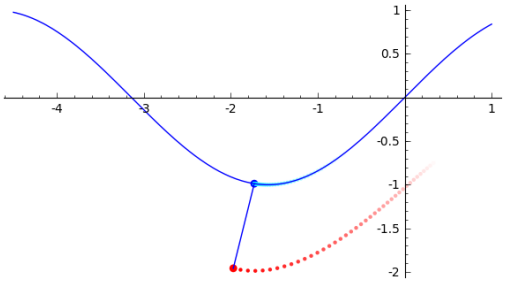
Mechanics with SageMath



Marcin Kostur, Jerzy Łuczka

Institute of Physics
University of Silesia
Poland

August 30, 2019

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

Contents

1 Preface

Computer algebra system provides a useful tool for analyzing mechanical systems. Why?

Problem solving problem in classical mechanics often require an hour or more of paper and pencil algebra. It is mostly connected with formulas which appear during change of coordinates in various expressions and differentiation. Derivation of Euler-Lagrange equations for double pendulum in polar coordinates can be a prominent example, in which expressions easily take two lines of handwriting. It used to be done on the blackboard but the same equations for five pendulums hanging in one line gives following expression:

$$
\begin{aligned}
&-\left(l_1\dot{\varphi_1}^2\sin(\varphi_1)-l_1\ddot{\varphi_1}\cos(\varphi_1)\right)\delta\varphi_1 l_1 m_1\cos(\varphi_1)+\left(\left(l_1\dot{\varphi_1}^2\cos(\varphi_1)+l_1\ddot{\varphi_1}\sin(\varphi_1)\right)m_1+g m_1\right)\delta\varphi_1 l_1\sin(\varphi_1)-\left(l_1\dot{\varphi_1}^2\sin(\varphi_1)+l_2\dot{\varphi_2}^2\sin(\varphi_2)-l_1\ddot{\varphi_1}\cos(\varphi_1)-l_2\ddot{\varphi_2}\cos(\varphi_2)\right)\left(\delta\varphi_1 l_1\cos(\varphi_1)+\delta\varphi_2 l_2\cos(\varphi_2)\right)m_2\\
&-\left(l_1\dot{\varphi_1}^2\sin(\varphi_1)+l_2\dot{\varphi_2}^2\sin(\varphi_2)+l_3\dot{\varphi_3}^2\sin(\varphi_3)-l_1\ddot{\varphi_1}\cos(\varphi_1)-l_2\ddot{\varphi_2}\cos(\varphi_2)-l_3\ddot{\varphi_3}\cos(\varphi_3)\right)\left(\delta\varphi_1 l_1\cos(\varphi_1)+\delta\varphi_2 l_2\cos(\varphi_2)+\delta\varphi_3 l_3\cos(\varphi_3)\right)m_3\\
&-\left(l_1\dot{\varphi_1}^2\sin(\varphi_1)+l_2\dot{\varphi_2}^2\sin(\varphi_2)+l_3\dot{\varphi_3}^2\sin(\varphi_3)+l_4\dot{\varphi_4}^2\sin(\varphi_4)-l_1\ddot{\varphi_1}\cos(\varphi_1)-l_2\ddot{\varphi_2}\cos(\varphi_2)-l_3\ddot{\varphi_3}\cos(\varphi_3)-l_4\ddot{\varphi_4}\cos(\varphi_4)\right)\left(\delta\varphi_1 l_1\cos(\varphi_1)+\delta\varphi_2 l_2\cos(\varphi_2)+\delta\varphi_3 l_3\cos(\varphi_3)+\delta\varphi_4 l_4\cos(\varphi_4)\right)m_4\\
&-\left(l_1\dot{\varphi_1}^2\sin(\varphi_1)+l_2\dot{\varphi_2}^2\sin(\varphi_2)+l_3\dot{\varphi_3}^2\sin(\varphi_3)+l_4\dot{\varphi_4}^2\sin(\varphi_4)+l_5\dot{\varphi_5}^2\sin(\varphi_5)-l_1\ddot{\varphi_1}\cos(\varphi_1)-l_2\ddot{\varphi_2}\cos(\varphi_2)-l_3\ddot{\varphi_3}\cos(\varphi_3)-l_4\ddot{\varphi_4}\cos(\varphi_4)-l_5\ddot{\varphi_5}\cos(\varphi_5)\right)\left(\delta\varphi_1 l_1\cos(\varphi_1)+\delta\varphi_2 l_2\cos(\varphi_2)+\delta\varphi_3 l_3\cos(\varphi_3)+\delta\varphi_4 l_4\cos(\varphi_4)+\delta\varphi_5 l_5\cos(\varphi_5)\right)m_5\\
&+\left(\delta\varphi_1 l_1\sin(\varphi_1)+\delta\varphi_2 l_2\sin(\varphi_2)\right)\left(\left(l_1\dot{\varphi_1}^2\cos(\varphi_1)+l_2\dot{\varphi_2}^2\cos(\varphi_2)+l_1\ddot{\varphi_1}\sin(\varphi_1)+l_2\ddot{\varphi_2}\sin(\varphi_2)\right)m_2+g m_2\right)\\
&+\left(\delta\varphi_1 l_1\sin(\varphi_1)+\delta\varphi_2 l_2\sin(\varphi_2)+\delta\varphi_3 l_3\sin(\varphi_3)\right)\left(\left(l_1\dot{\varphi_1}^2\cos(\varphi_1)+l_2\dot{\varphi_2}^2\cos(\varphi_2)+l_3\dot{\varphi_3}^2\cos(\varphi_3)+l_1\ddot{\varphi_1}\sin(\varphi_1)+l_2\ddot{\varphi_2}\sin(\varphi_2)+l_3\ddot{\varphi_3}\sin(\varphi_3)\right)m_3+g m_3\right)\\
&+\left(\delta\varphi_1 l_1\sin(\varphi_1)+\delta\varphi_2 l_2\sin(\varphi_2)+\delta\varphi_3 l_3\sin(\varphi_3)+\delta\varphi_4 l_4\sin(\varphi_4)\right)\left(\left(l_1\dot{\varphi_1}^2\cos(\varphi_1)+l_2\dot{\varphi_2}^2\cos(\varphi_2)+l_3\dot{\varphi_3}^2\cos(\varphi_3)+l_4\dot{\varphi_4}^2\cos(\varphi_4)+l_1\ddot{\varphi_1}\sin(\varphi_1)+l_2\ddot{\varphi_2}\sin(\varphi_2)+l_3\ddot{\varphi_3}\sin(\varphi_3)+l_4\ddot{\varphi_4}\sin(\varphi_4)\right)m_4+g m_4\right)\\
&+\left(\delta\varphi_1 l_1\sin(\varphi_1)+\delta\varphi_2 l_2\sin(\varphi_2)+\delta\varphi_3 l_3\sin(\varphi_3)+\delta\varphi_4 l_4\sin(\varphi_4)+\delta\varphi_5 l_5\sin(\varphi_5)\right)\left(\left(l_1\dot{\varphi_1}^2\cos(\varphi_1)+l_2\dot{\varphi_2}^2\cos(\varphi_2)+l_3\dot{\varphi_3}^2\cos(\varphi_3)+l_4\dot{\varphi_4}^2\cos(\varphi_4)+l_5\dot{\varphi_5}^2\cos(\varphi_5)+l_1\ddot{\varphi_1}\sin(\varphi_1)+l_2\ddot{\varphi_2}\sin(\varphi_2)+l_3\ddot{\varphi_3}\sin(\varphi_3)+l_4\ddot{\varphi_4}\sin(\varphi_4)+l_5\ddot{\varphi_5}\sin(\varphi_5)\right)m_5+g m_5\right)
\end{aligned}
$$

Although very informative, such derivations may first of all discourage the student from further analyzing the system, and secondly, there is a growing chance of making a mistake with the length of equations.

Instead of scaring students with lengthy formulas these tasks can be quickly done with the help of Computer Algebra Systems (CAS).

Mechanics with SageMath consist of a set of notebooks in which we solve classical problems in mechanics. Our approach differs from the traditional one in:

1. It is insisted that all derivations are done automatically with help of CAS,
2. When equation of motion are obtained algebraically, we use numerical and algebraic method to analyze the system.
3. Wherever possible we create visually appealing pictures or animations which illustrate the key properties of the system.

This approach requires the reader to take time to learn about the computer algebra system and programming. However we believe that this knowledge will give disproportionate benefits and will not only make problem solving in mechanics a pleasure but prepare to solve real life problems where analytically tractable formulas are not so common.

Marcin & Jerzy

In [ ]:

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

There are three approaches to describe classical mechanical systems. Historically, the first theory was formulated by Newton and is called Netwon Mechanics. The second is Lagrange Mechanics and is convenient for description of systems with constraints. The third is Hamilton Mechanics. Lagrange mechanics is a basis for the modern theory of elementary particles. In turn, the Hamilton approach is used in formulation of quantum physics.

2.1 Harmonic Oscillator

We demostrate three approaches to classical mechanical systems by considering 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 Lagrange mechanics

In the Lagrange approach, the mechanical system is described by the scalar function L called the Lagrange 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.

```
In [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])
```

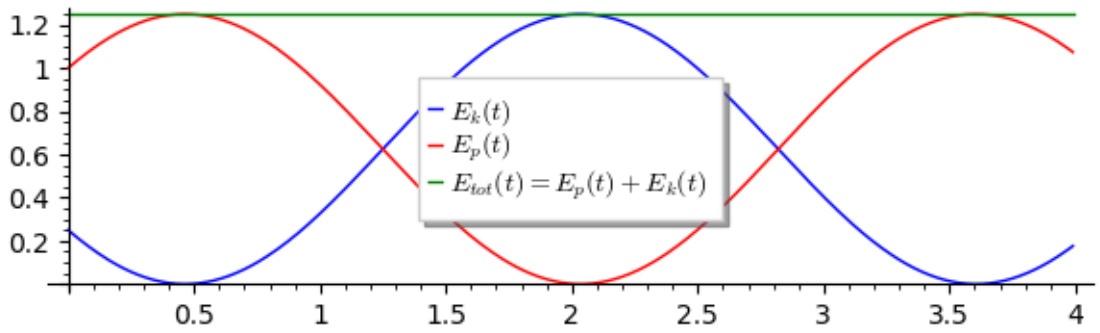
We can compute E_k and E_p:

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

And plot the results:

```
In [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
```

Out[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 Lagrange mechanics

In the Lagrange approach, in this case 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 Newton equation can be rewritten in the form:

$$\begin{cases} \dfrac{dx}{dt} = v \\ \dfrac{dv}{dt} = -\omega_0^2 x - \gamma_0 v, \quad \gamma_0 = \gamma/m \end{cases} \tag{2}$$

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

```
In [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])
```

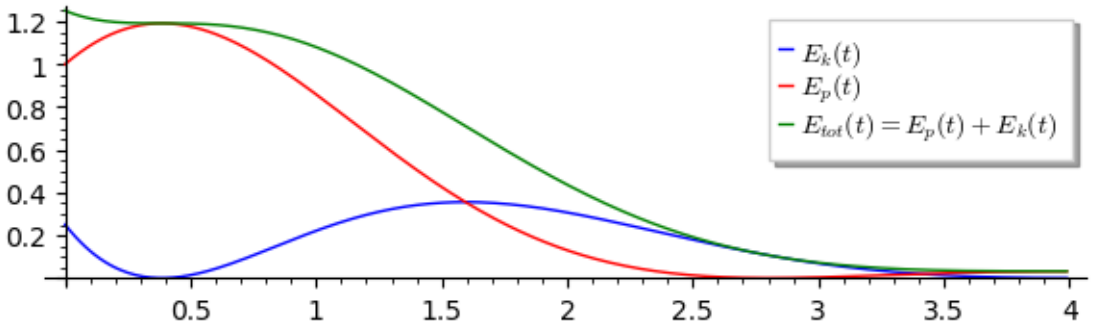
We can compute E_k and E_p:

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

And plot the results:

```
In [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
```

Out[6]:



3 Newton equations: Conservation of energy

3.1 One degree of freedom

Consider the Newton equation for a particle with one degree of freedom in the form of:

$$m\ddot{x} = F(x) = -\frac{dU(x)}{dx} = -U'(x)$$

where $F(x)$ is the force acting on the particle and $U(x)$ defined by the relation

$$F(x) = -U'(x)$$

is called potential energy (we will write shortly - potential). If we know the force $F(x)$, then the potential can be found by calculating the integral:

$$U(x) = -\int_a^x F(y)dy$$

where a is any number which does not influence dynamics of the system. For example, we can choose so that at some point the potential is zero or infinite.

The Newton equation is a second-order differential. We will re-write it in the form

$$\begin{aligned} \dot{x} &= v, \qquad x(0) = x_0 \\ m\dot{v} &= F(x) = -U'(x), \qquad v(0) = v_0 \end{aligned}$$

It is a set of 2 autonomous differential equations of first order. It means that the phase space $\{x, v\}$ is 2-dimensional. In this phase space (which is a plane) we can analyze phase curves: $\{x(t), v(t)\}$ as a parametric curves with the parameter being time t and the coordinate $x(t)$ and the velocity $v(t)$ change over time according to the Newton equation.

Theorem: There is a function (combination) of $x(t)$ and $v(t)$ that does not change over time:

$$E = E[x(t), v(t)] = \frac{1}{2}mv^2(t) + U(x(t)) = \frac{1}{2}mv^2(0) + U(x(0)) = E[x(0), v(0)]$$

The quantity E is called in physics the total energy of the system. It consists of two parts: the kinetic energy of the particle: $E_k = mv^2/2$ and the potential energy of the particle: $E_p = U(x)$. The total ebergy is determined by initial conditions for the coordinate $x(0$ and velocity $v(0)$ which are in the expression $E[x(0), v(0)]$.

Proof: If E does not change in time, it means that it is a constant function with respect to time and the derivative with respect to time should be zero. Indeed:

$$\frac{dE}{dt} = \frac{d}{dt}E[x(t), v(t)] = \frac{\partial E}{\partial x}\frac{dx}{dt} + \frac{\partial E}{\partial v}\frac{dv}{dt} = U'(x)\,\dot{x} + mv\,\dot{v} = -F(x)v + vF(x),$$

where we used the relationship between the force and the potential energy as well as we exploited the Newton equation of motion. Because E does not change in time, we say that it is a constant of motion or the integral of motion, or the first integral of the system (the two last names seem to be bizarre, because in the expression for E no integral is visible). The existence of constants or integrals of the movement facilitates the analysis of systems.

First conclusion - phase curves

The equation

$$\frac{1}{2}mv^2 + U(x) = E$$

defines a curve on the plane $\{x, v\}$.

Second conclusion - admisible particle positions

The equation

$$U(x) = E$$

is fulfilled when the velocity $v = v(t)$ of the particle is zero. It determines the interval of possible positions $x = x(t)$ of the particle. The curve on the plane is symmetrical with respect to the horizontal axis x. It follows from the relation

$$v = \pm\frac{2}{m}\sqrt{E - U(x)}$$

Let us apply the above to a harmonic oscillator for which the form of the force is well known:

$$F(x) = -kx = -m\omega^2 x, \qquad U(x) = \frac{1}{2}kx^2 = \frac{1}{2}m\omega^2 x^2, \qquad \omega^2 = \frac{k}{m}$$

The law of energy conservation says that

$$E = \frac{1}{2}mv^2(t) + \frac{1}{2}kx^2(t) = const. = \frac{1}{2}mv^2(0) + \frac{1}{2}kx^2(0)$$

We note that the above equation in variables $\{x, v\}$ has the form

$$mv^2 + kx^2 = 2E$$

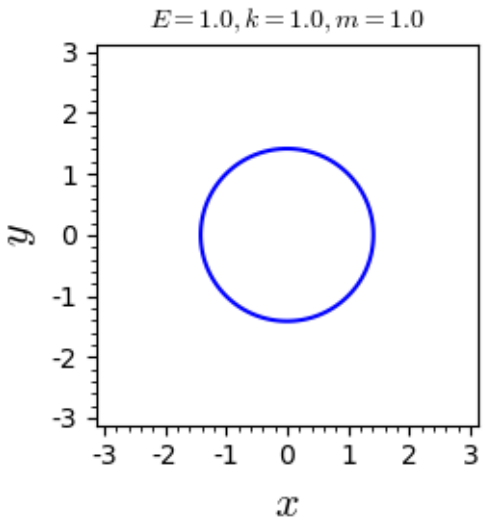
This is the equation of the ellipse:

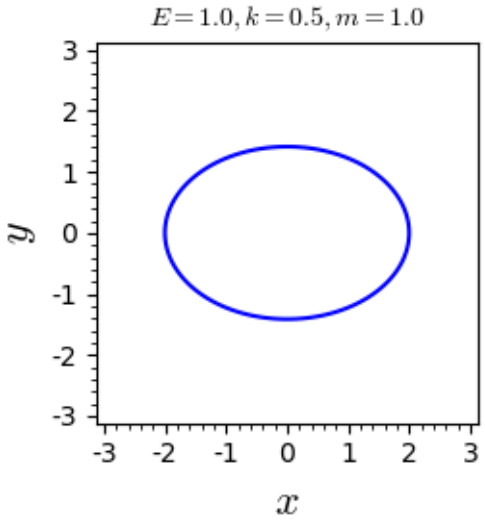
$$\frac{x^2}{(2E/k)} + \frac{v^2}{(2E/m)} = 1$$

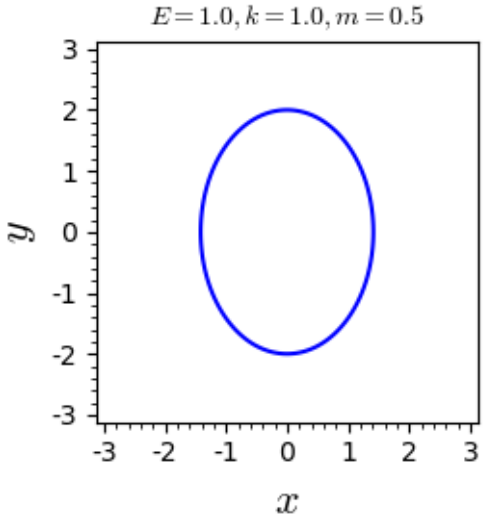
with $a = 2E/k$ and $b = 2E/m$ axes. Let's draw an ellipse for, say, $E = 2, k = 0.2$ and $m = 1$.

(Almost) everyone knows how this ellipse looks but we will do more in order to develop a natural ability to use Sage programs to visualize and interpret results.

```
In [1]: var('x v')
        import os
        PDF = 'PDF' in os.environ.keys()
        def draw_phase_curve(E,k,m):

            plt = implicit_plot( x^2/(2*E/k)+v^2/(2*E/m) == 1, \
                                 (x,-3,3),(v,-3,3),\
                                 axes_labels=[r'$x$',r'$y$'],
                                figsize=4,
                                title=r'$E=%0.1f, k=%0.1f,m=%0.1f$'%(E,k,m))
            plt.show()

        if PDF:
            draw_phase_curve(E=1,k=1,m=1)
            draw_phase_curve(E=1,k=.5,m=1)
            draw_phase_curve(E=1,k=1,m=.5)
        else:
            @interact
            def _(E=slider(0.5,2,.1,label=r'$E$'), \
                  k=slider(0.2,1.0,0.1,label=r'$k$'), \
                  m=slider(0.2,2,.1,label=r'$m$')):
                draw_phase_curve(E,k,m)
```







The particle moves in such a way that $\{x(t), v(t)\}$ moves on the ellipse. Because the ellipse is a closed curve, the motion is periodic and its period can be calculated from the law of conservation energy. Below we show step by step what to do to analyse the system using the law of energy conservation.

- We draw a graph showing the potential $U(x)$ - Below this graph, with the vertical axis set as in the potential graph, we draw 2 symmetrical curves given by the energy conservation law. The two curves $= v(x, E)$ define the phase curves. - The particle moves to the right when the speed is positive $v > 0$ (green curve) and to the left when the speed is negative $v < 0$.

We will try to analyze Newton's equation to obtain phase curves.

$$m\ddot{x} = F$$

If the force will be linear $F = -kx$, then we will get the above described problem of the harmonic oscillator. At the beginning, we must declare the names of variables and parameters used in the model. Remember - each time, if you want to calculate something symbolically, you have to write a line and do it.

```
In [2]: var('x v')
```

```
Out[2]: (x, v)
```

We will now set the system parameters.

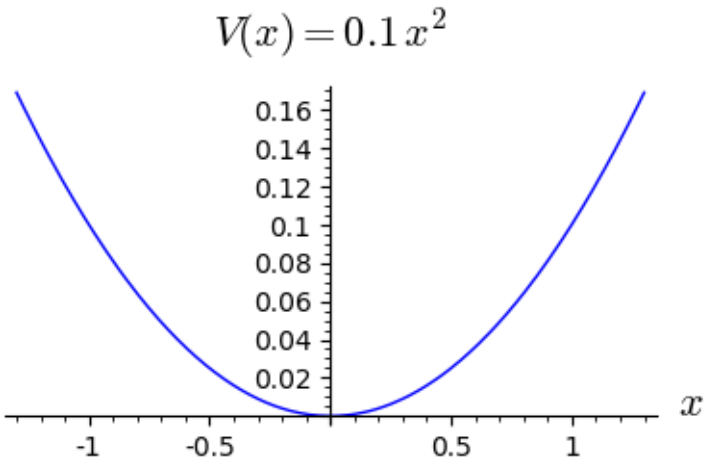
```
In [3]: x0 = 1.3
        v0 = 0.3
        k = 0.2
        m = 1
```

Now we declare the form of the force. In our case it will be simply

```
In [4]: F = -k*x
```

The potential is defined as an integral of the force (with minus sign - see above). We'll calculate it using Sage.

```
In [5]: V = -integral(F,x)
        p1 = plot(V, xmin=-x0, xmax=x0)
        p1.show(figsize=4, axes_labels=[r'$x$',r'$V(x)=%s$'%latex(V)])
```



From the law of energy conservation, we now calculate how the velocity depends on the position (these phase curves).

```
In [6]: E = m*v0^2 + V(x=x0)
        PZE = m*v^2 + V == E
        rozw = solve(PZE, v); show(rozw)
        v1=rozw[0].rhs()
        v2=rozw[1].rhs()
```

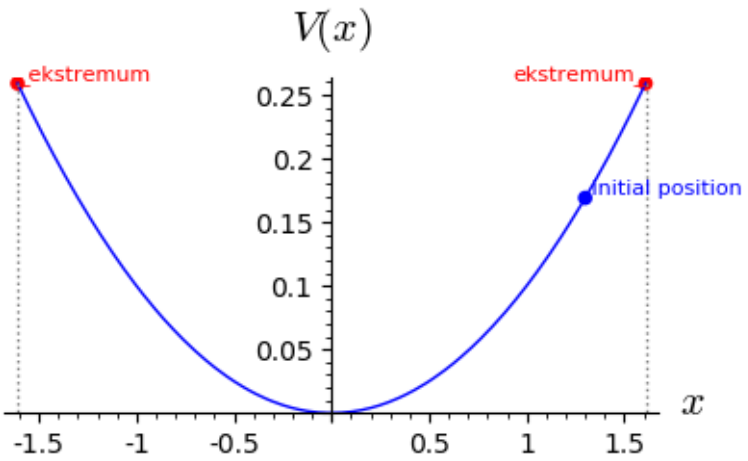
```
[v == -1/10*sqrt(-10*x^2 + 259/10), v == 1/10*sqrt(-10*x^2 + 259/10)]
```

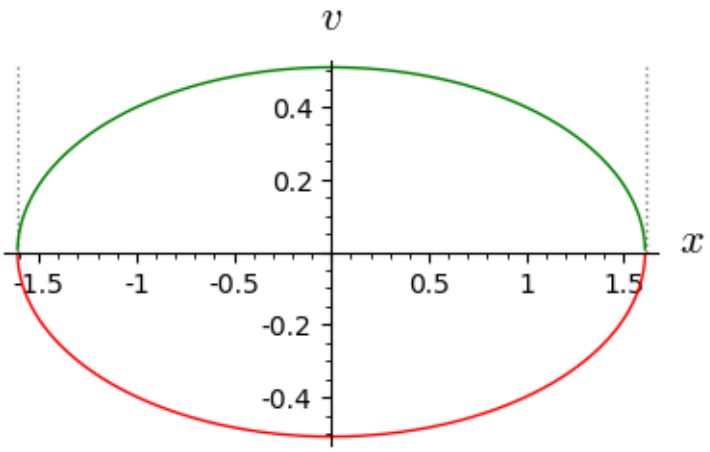
For $v = 0$ we will calculate the extreme deflections of the particle.

```
In [7]: #ekstremalne wychylenie
        #prawo zachowania energii dla v=0
        rozw = solve(PZE(v=0), x); show(rozw)
        xmin = rozw[0].rhs()
        xmax = rozw[1].rhs()
```

```
[x == -1/10*sqrt(259), x == 1/10*sqrt(259)]
```

```
In [8]: #punkt pocztkowy (tak jak powyej)
        ball = (x0,V(x=x0))
        p0  = point(ball,size=30)
        p0 += text(r" initial position",ball,vertical_alignment='bottom',horizontal_alignmer

        #ekstrema
        ball = (xmax,V(x=xmax))
        p0 += point(ball,size=30,color='red')
        p0 += text("ekstremum_",ball,vertical_alignment='bottom',horizontal_alignment='right
        p12a = line((ball,(xmax,0)),linestyle='dotted',color='grey')
        ball = (xmin,V(x=xmin))
        p0 += point(ball,size=30,color='red')
        p0 += text("_ekstremum",ball,vertical_alignment='bottom',horizontal_alignment='left
        p12a += line((ball,(xmin,0)),linestyle='dotted',color='grey')

        #potencja
        p1 = plot(V, xmin=xmin, xmax=xmax)

        #krzywe fazowe
        p12b = line(((xmin,0),(xmin,v2(x=0))),linestyle='dotted',color='grey')
        p12b += line(((xmax,0),(xmax,v2(x=0))),linestyle='dotted',color='grey')
        p2 =  plot(v1, (x,xmin,xmax), color='red')
        p2 += plot(v2, (x,xmin,xmax), color='green')


        (p0+p1+p12a).show(figsize=4, axes_labels=['$x$','$V(x)$'])
        (p12b+p2).show(figsize=4,xmax=xmax, axes_labels=['$x$','$v$'])
```





3.2 Many degrees of freedom

A system of one degree of freedom is always potential (i.e. there is a function $U(x)$ provided that the force depends only on the position of the particle and time). If the force also depends on the particle velocity, i.e. when $F = F(x, v)$, there is no such a function $F(x, v) = -U'(x) = -dU(x)/dx$. For a system of many degrees of freedom, described by the set of Newton's equations:

$$m_i \frac{d^2\vec{r}_i}{dt^2} = \vec{F}_i(\vec{r}_1, \vec{r}_2, \vec{r}_3, ..., \vec{r}_N)$$

for N particles, the system is potential if there exists a scalar function $V(\vec{r}_1, \vec{r}_2, \vec{r}_3, ..., \vec{r}_N)$ such that the force acting on the i-th particle is a gradient of the potential U with a minus sign. We can explain it for the example of one particle moving in the 3-dimensional space:

$$m\frac{d^2x}{dt^2} = F_1(x,y,z) = -\frac{\partial}{\partial x}U(x,y,z)$$

$$m\frac{d^2y}{dt^2} = F_2(x,y,z) = -\frac{\partial}{\partial y}U(x,y,z)$$

$$m\frac{d^2z}{dt^2} = F_3(x,y,z) = -\frac{\partial}{\partial z}U(x,y,z)$$

In the general case, when there are three components of the force $[F_1, F_2, F_3]$, the potential cannot exist. Then we say that the system is non-potential. There is a simple criterion to check whether the system is potential or not. If the system is potential, i.e. if

$$\vec{F} = -grad\ U \qquad \text{then} \qquad rot\ \vec{F} = -rot\ grad\ U = -\vec{\nabla} \times \vec{\nabla} U \equiv 0$$

where the nabla operator $\vec{\nabla}$ is a vector differentiation operator in the form

$$\vec{\nabla} = \hat{e}_x\frac{\partial}{\partial x} + \hat{e}_y\frac{\partial}{\partial y} + \hat{e}_y\frac{\partial}{\partial y}$$

It is sufficient to check whether rotation of the force is zero.

Experiment with Sage!

Check if $\vec{F}(x,y,z)$ forces with components

$$A. \qquad F_1(x,y,z) = \frac{y}{x^2+y^2+z^2}, \quad F_2(x,y,z) = -\frac{x}{x^2+y^2+z^2}, \quad F_3(x,y,z) = \frac{z}{x^2+y^2+z^2}$$

$$B. \qquad F_1(x,y,z) = \frac{x-z}{x^2+y^2}, \quad F_2(x,y,z) = xe^{-y^2}, \quad F_3(x,y,z) = z+5$$

$$C. \qquad F_1(x,y,z) = 25x^4y - 3y^2, \quad F_2(x,y,z) = 5x^5 - 6xy - 5, \quad F_3(x,y,z) = 0$$

are potential.

```
In [9]: var('x y z')
        F = vector([ y/(x^2+y^2+z^2),-x/(x^2+y^2+z^2),z/(x^2+y^2+z^2) ])
        # U = (x+exp(x*y*z))/(x^2+z*y^2+z^2)
        # F = [-diff(U,x_) for x_ in [x,y,z]]

        def curl(F):
            assert(len(F) == 3)
            return vector([diff(F[2],y)-diff(F[1],z), diff(F[0],z)-diff(F[2],x), diff(F[1],x

        show( simplify(curl(F)) )
```

```
(-2*x*z/(x^2 + y^2 + z^2)^2 - 2*y*z/(x^2 + y^2 + z^2)^2, 2*x*z/(x^2 + y^2 + z^2)^2 - 2*y*z/
```

If the system is potential then one can check that, as in the case of a 1-degree system of freedom, there is an integral of motion - the total energy of the system:

$$E = \sum_i \frac{m\vec{v}^2}{2} + U(\vec{r}_1, \vec{r}_2, \vec{r}_3, ..., \vec{r}_N) = constant, \qquad \frac{dE}{dt} = 0$$

Therefore, this field of forces is called a conservative force field. All forces related to the potential force field are conservative forces. Of course, there are forces that are not potential forces.

For a system of more than one degree of freedom, the law of energy conservation is not so useful as for the former. E.g. for a system of two degrees of freedom:

$$E = \frac{mv_1^2}{2} + \frac{mv_2^2}{2} + U(x, y) = constant$$

is a function of four variables $\{x, y, v_1, v_2\}$ and the above relation defines a hyper-surface in the 4-dimensional space. So, the analysis of such a geomertic object is not simple (even not possible in a general case). The eminent mathematician V. I. Arnold has written that this system is beyond capabilities of modern mathematics.

4 Harmonic oscillator with computer algebra

4.1 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

4.1.1 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 point particle of mass m in a square potential:

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

The Newton equation for this system reads:

$$ma = -\frac{dU(x)}{dx} = -kx,$$

and because acceleration a is the second derivative of the position with respect to time one gets the differential equation:

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

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 $m\vec{a} = \vec{F}$.

4.1.2 Small oscillations

The harmonic oscillator is an extremely important model that appears in many real situations, both in classical and quantum systems. We now show why? Note that for any potential $U(x)$, which at some point x_0, say $x_0 = 0$, has a local minimum, its Taylor series takes the form:

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

Since we have assumed that at $x_0 = 0$ the potential has a local minimum, the derivative at this point is zero, i.e., $U'(0) = 0$ and therefore:

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

The Taylor's series can be approximated by keeping only the first two non-vanishing terms. The first term $U(0)$ is constant and can be neglected because the corresponding force does not depend on this constant. As a result, the potential is approximated by the expression:

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

The potential is identical to the potential of the harmonic oscillator and the second derivative of the potential at the point of its minimum $U''(0)$ can be identified with the elastic constant k. So we see that the movement of the particle in the vicinity of the potential minimum can be approximated by a harmonic oscillator. This fact is the reason why the harmonic oscillator is such frequently used in physics.

4.2 Free oscillator

Consider the Newton equation for a harmonic oscillator in a vacuum (there is no dissipation,no a frictional force) It has a form:

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

It is conveniet to convert this equation to the form

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

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

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

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

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

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

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

Out[2]:

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

Because we use a set of variables for operations in expressions containing derivatives, the above cell gives 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:

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

Out[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 the so-called a general solution depends on two constants K_1 and K_2. We can determine these two constants knowing the position and velocity of the oscillator at some given time t_0. Because the equation of motion do not depend explicitly on time, the solution $x(t) = x(t, t_0) = x(t-t_0)$ depends on the time difference $t-t_0$. Therefore, without the limitation of generality, $t = 0$ can be used as an initial moment.

Sage can also determine constants. If at the moment $t = 0$ the oscillator is in the rest state: $x(0) = x_0$ and $v(0) = 0$, then we have:

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

Out[4]:

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

If, at $t = 0$, the initial conditions are: $x(0) = 0$ and $v(0) = v_0$, then we have:

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

Out[5]:

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

Finally, for a general case when $x(0) = x_0$ and $v(0) = v_0$ the result reads

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

Out[6]:

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

We can also assume specific numerical values $x(0) = 2$ and $v(0) = 3$. Then

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

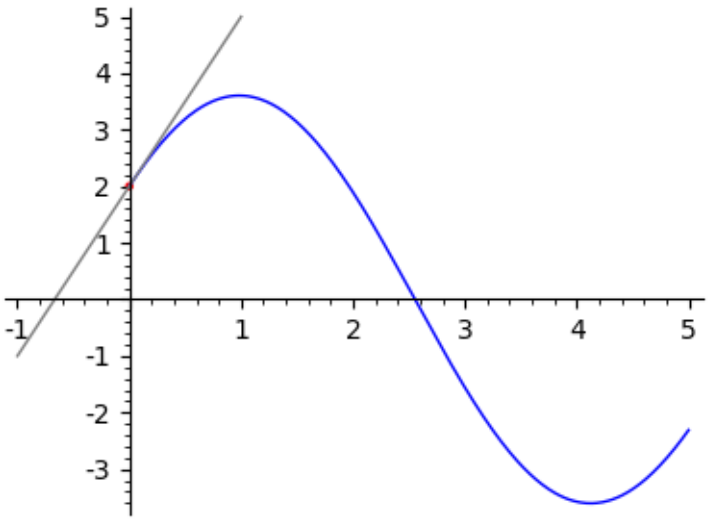
Out[7]:

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

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

```
In [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')
```

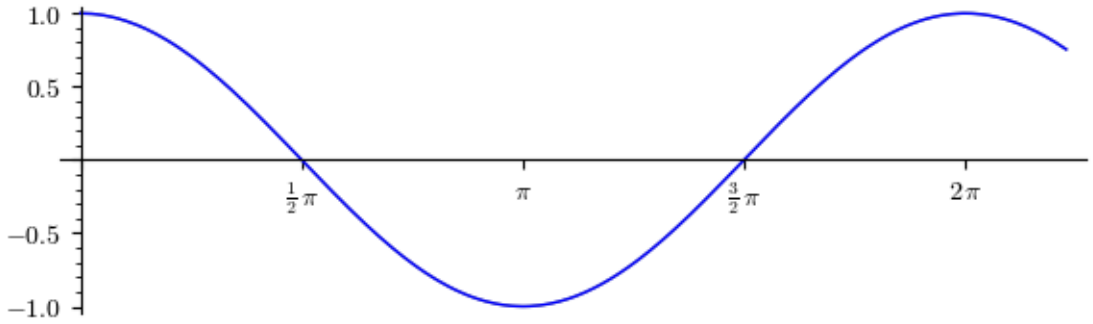
Out[8]:

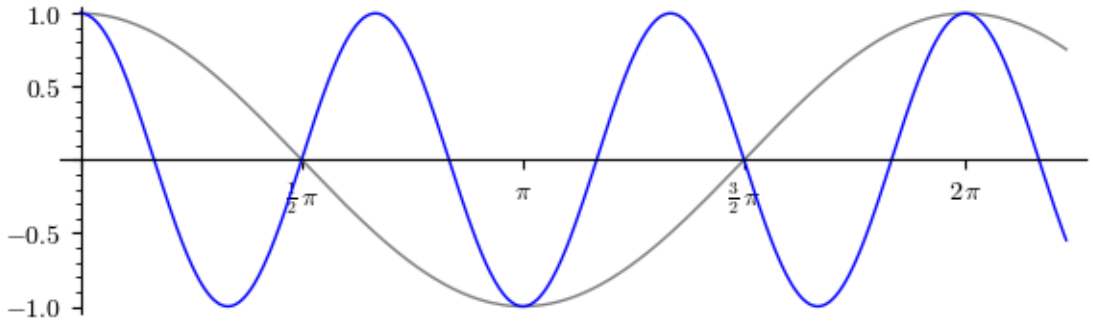


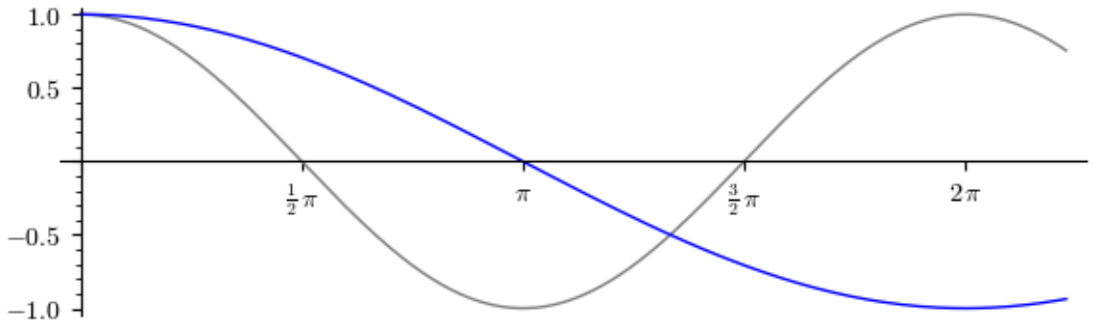
We can examine how the solution depends on the parameters ω_0 and x_0 (with $v_0 = 0$).

```
In [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
```

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







4.3 Damped oscillator

As we have seen from previous considerations, the free harmonic oscillator exhibits oscillations infinitively long. In real situations, systems are in contact with its surroundings (environment) which introduces a dissipation process. In other words, the system is subjected to damping forces, which cause the oscillations to decay in time. It is the 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 frictional force proportional to the velocity:

$$\vec{F}_D = -\Gamma\vec{v},$$

where Γ is a damping (friction) constant. It is so called the Stokes force or the drag force 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 in a more complicated way on the velocity. E.g. in aerodynamics, a quadratic relationship is a good model.

Consideration of a linearly rate-dependent friction force has the basic advantage that, despite adding an additional term to the equation of motion, we still deal with a linear differential 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 introduce the rescaled damping constant $\gamma = \Gamma/m$ and $\omega_0 = k/m$. Then

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

From 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 obtain various different solutions. It can be easily done assuming that the solution of the Newton equation in an exponent e^{kt} and substitute this form into ODE:

```
In [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)
```

Out[11]:

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

Now we can factorize this expression:

```
In [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 a quadratic equation which has the following solutions:

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

Out[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 the determinant:

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

for which the solution has qualitatively different properties:

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

4.3.1 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

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

```
In [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]
```

Out[14]:

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

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

Out[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 the above solution contains two constants which have no "handles":

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

Out[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:

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

Out[17]: [_K1, _K2]

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

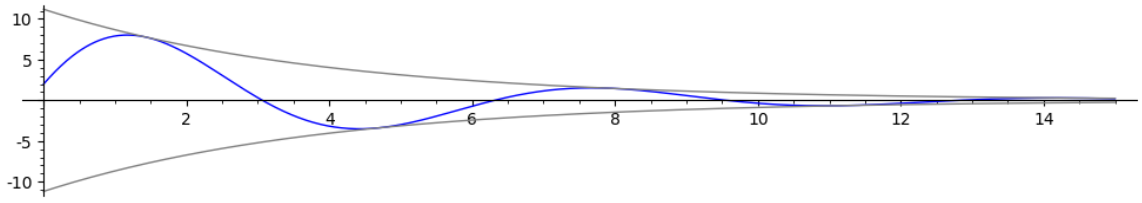
```
In [18]: _K1
```

Out[18]: _K1

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

```
In [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' )
```

Out[19]:



Where did the formula for A come from?

We can use the trigonometric identity 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:

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

Out[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.

```
In [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)) }
```

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

Out[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:

In [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)) }
```

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

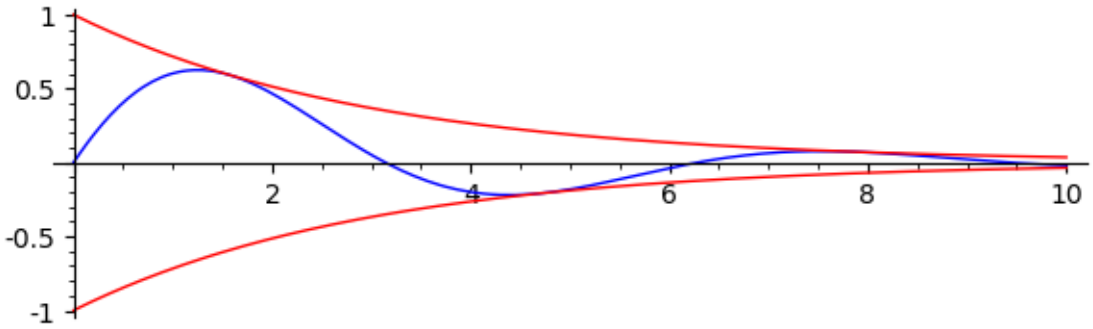
Out[24]:

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

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

In [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')
```

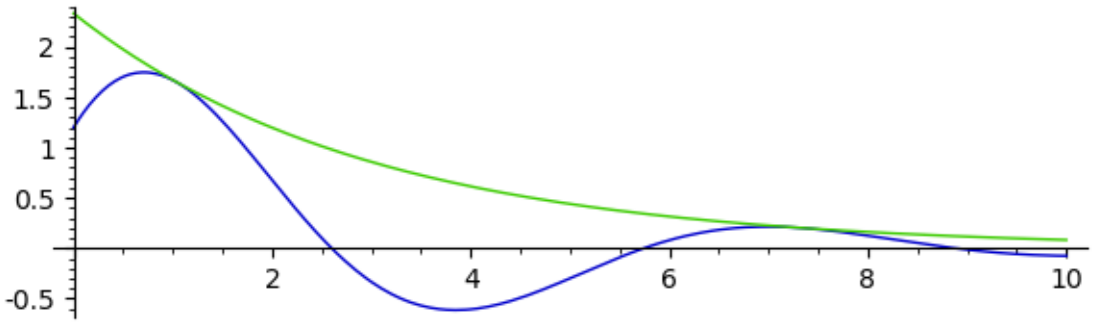
Out[25]:



we have therefore:

In [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))
```

Out[26]:



4.3.2 Two real roots

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

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

then we have:

```
In [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]
```

Out[27]:

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

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

Out[28]:

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

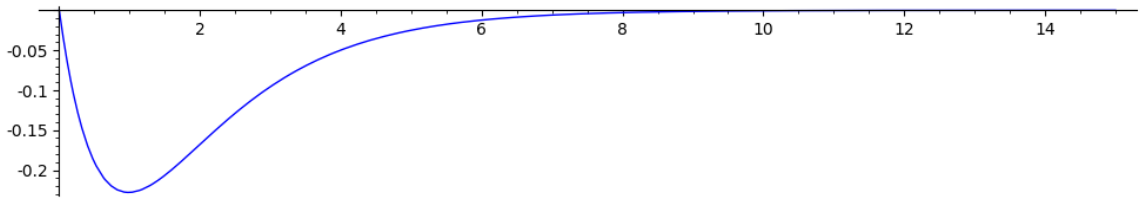
We see that the solution does not contain periodic functions but is the sum of two exponents with negative arguments.

Task: Prove that the argument of the exponent $k_1 e^{\left(-\frac{1}{2}\left(\gamma-\sqrt{\gamma^2-4\omega_0^2}\right)t\right)}$, for $t > 0$, is negative.

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

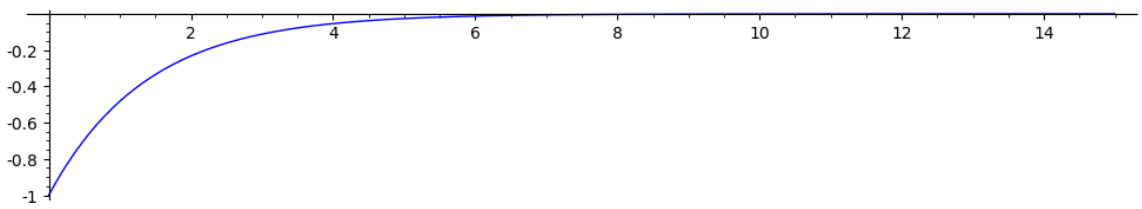
```
In [29]: var('t')
         plot( phi_anal.subs({_K1:-1,_K2:1,omega0:1,g:2.1}), (t,0,15), figsize=(10,2))
```

Out[29]:




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

Out[30]:



4.3.3 The degenerate case

Consider the case when the determinant is zero,

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

Let us have a look how the general solution looks:

```
In [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]
```

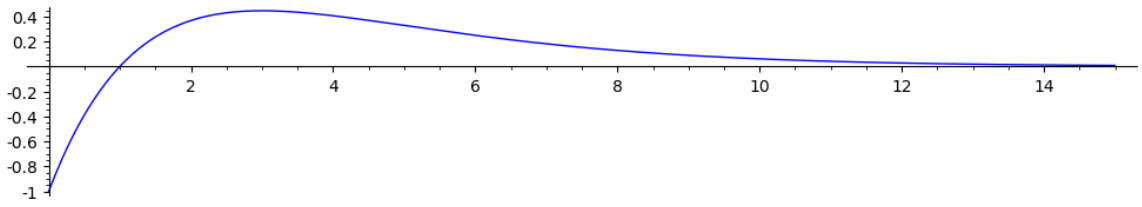
Out[31]:

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

This case is called: critically damped.

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

Out[32]:



4.3.4 Dissipated power

If we know analytical solutions then one can evaluate how much of system energy is dissipated to the environment. For a given value of ω_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 $v_0 = 1$:

```
In [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 characterize dissipated power P (it is energy or work per unit time):

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

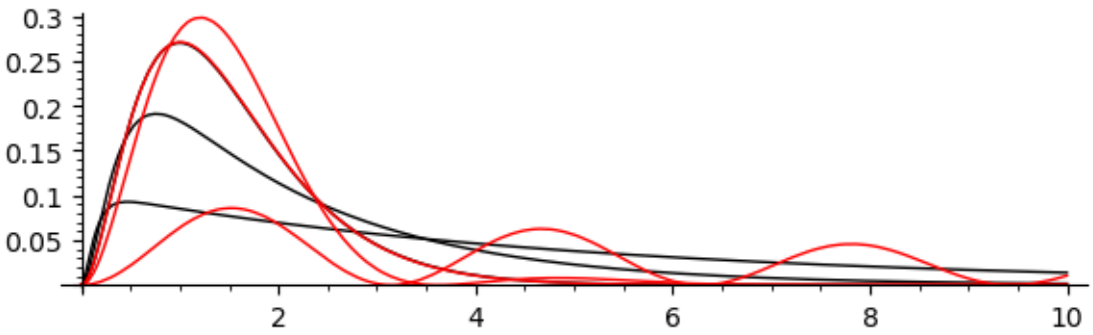
and total energy (for $m = 1$):

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

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

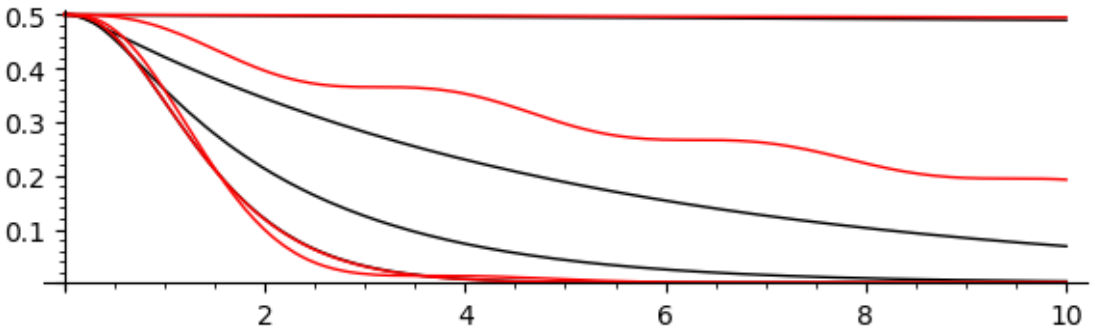
Now we can plot power and trajectory as a function of time:

```
In [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))
```



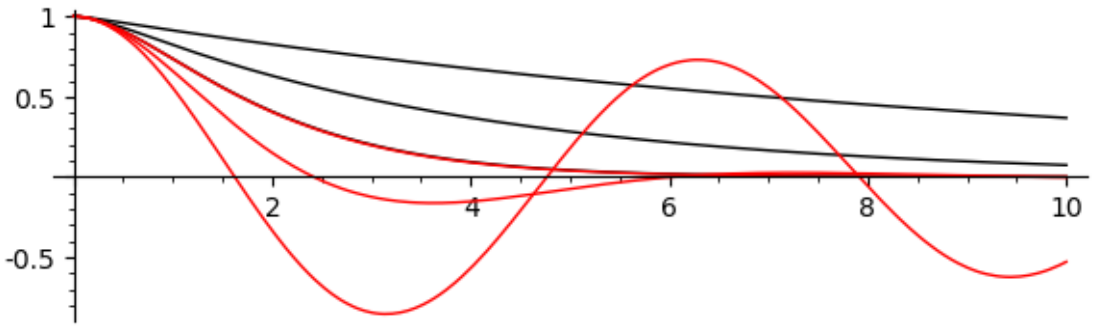
In the above figure, the dissipated power P is shown for selected values of the damping constant.

```
In [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))
```



In the above figure, the total energy E_{tot} is depicted for various damping constants.

```
In [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))
```



The above figure shows the solutions $x(t)$ for different damping constant.

4.4 Forced harmonic oscillator

Now, we take into account the driving in the form of a time-dependent periodic force $\sin(\omega t)$. 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:

In [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)
```

Out[38]:

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

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

```
variationofparameters
```

Out[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 which is a sum of two terms: 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 limit $t \to \infty$.

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

Out[40]:

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

Let's transform a special solution to the form:

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

To do this, we extract the numerator and denominator:

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

Out[41]:

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

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

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

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

```
In [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:

```
In [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:

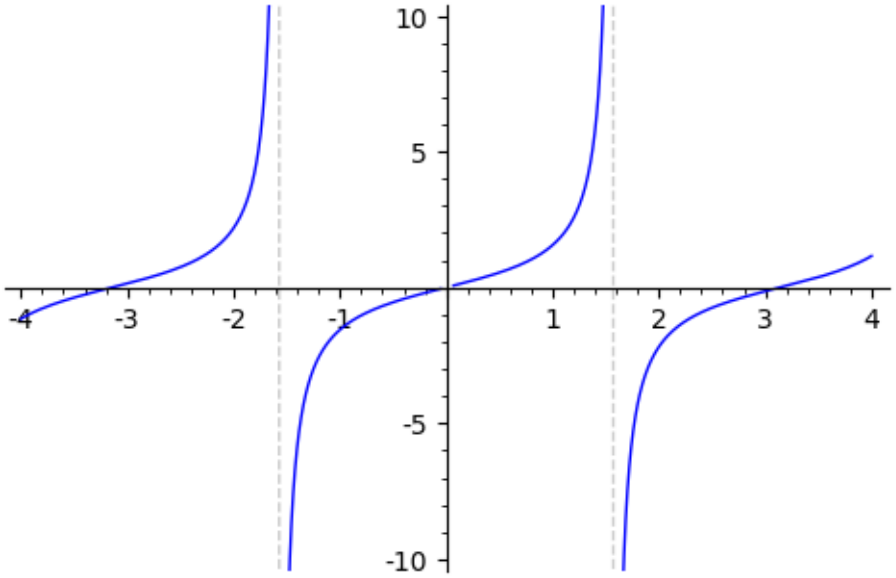
```
In [44]: assume(a>0)
         (a*sin(x)+b*cos(x)).subs(sub3).full_simplify().show()
```

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

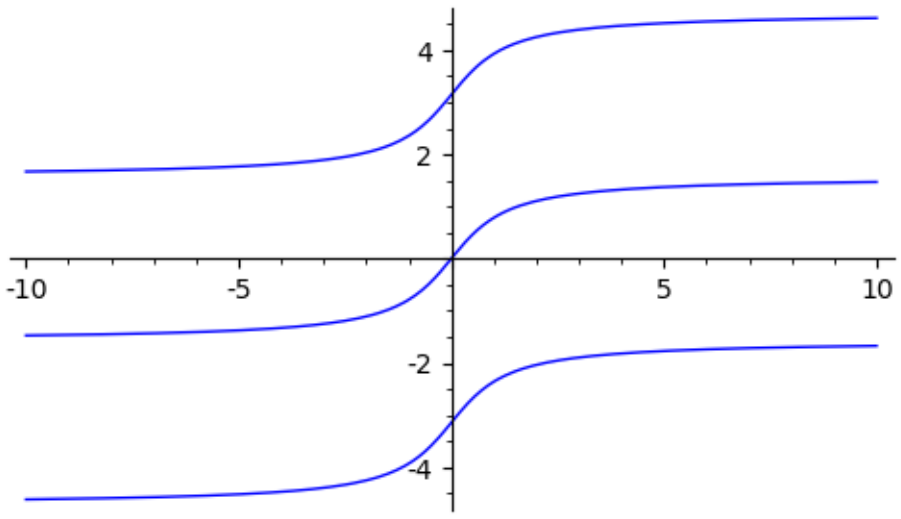
Everything looks nice, but we have a classic problem to choose the function's branch. Let's look at the graph of the $\tan(x)$ function:

```
In [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
```



Out[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`:

```
In [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.

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

```
Out[47]:
```

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

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

```
Out[48]:
```

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

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

```
Out[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:

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

```
Out[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:

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

Out[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:

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

Out[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:

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

Out[53]:

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

It is a famous formula related to 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 efficient way if the frequency of the external force is consistent with the eigen-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.

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

Out[54]:

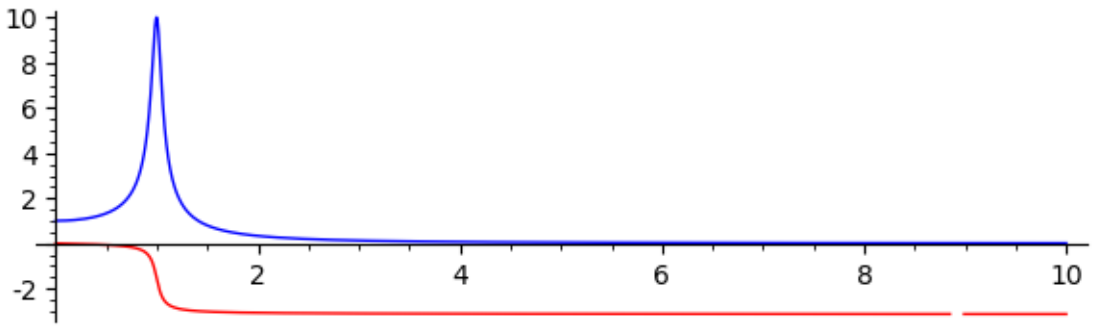
$$-\pi + \arctan\left(a\gamma\omega, a\omega^2 - a\omega_0^2\right)$$

The maximal amplitude depends on the attenuation coefficient:

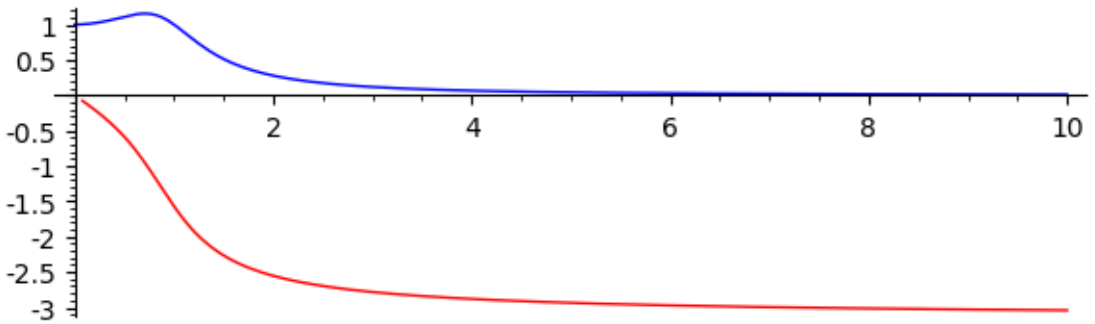
```
In [55]: #@interact
         def plot_A_phase(g_=slider(0.01,1,0.01,label="$\gamma$",default=0.2)):
             pars = {g:g_,a:1,omega0:1}
             print ( g^2*omega^2 + omega^4 - 2*omega^2*omega0^2 + omega0^4 ).subs(pars)
             p = plot(A.subs(pars),(omega,0,10))
             p += plot(phase.subs(pars),(omega,0,10),color='red',detect_poles="show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

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

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




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



Out[56]: (None, None)

4.4.1 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:

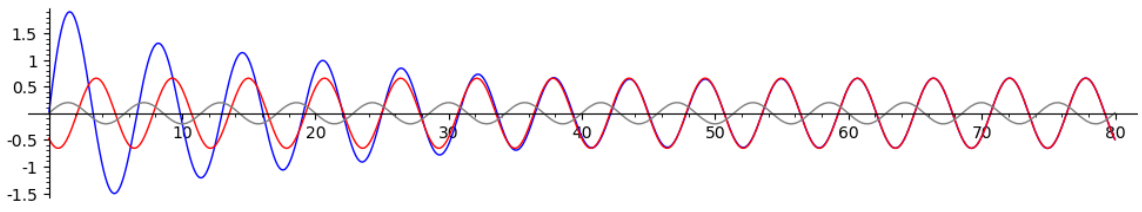
In [57]:
```
var('phid',latex_name=r'\dot\varphi')
var('phi',latex_name=r'\varphi')
```

Out[57]: phi

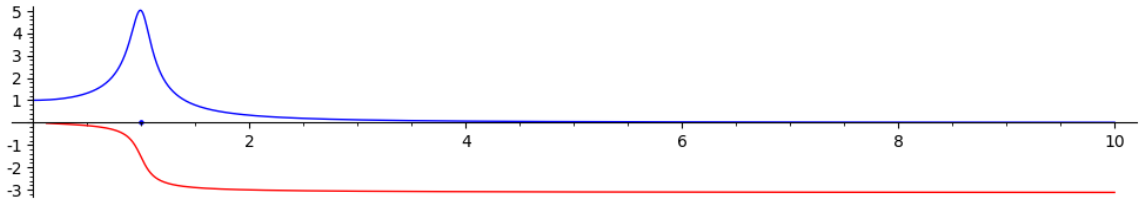
In [58]:
```
rhs = solve(osc,diff(Phi(t), t, 2))[0].rhs()
#@interact
def plot_A_traj(g_=slider(0.01,2.2,0.01,label="$\gamma$",default=0.2),\
                omega_=slider(0.01,5.134,0.01,label="$\omega$",default=1.4)):
    pars = {g:g_,a:1,omega0:1}
    print ( g^2*omega^2 + omega^4 - 2*omega^2*omega0^2 + 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_}
```

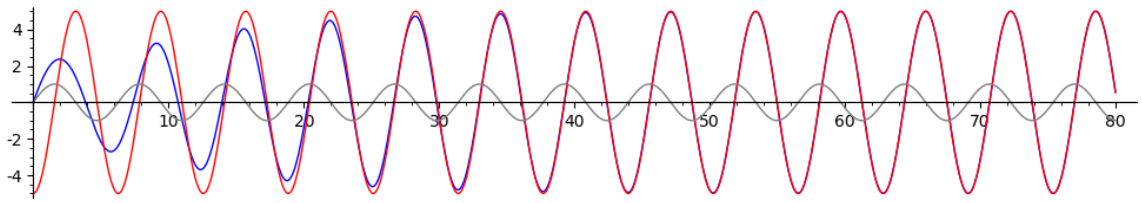
In [59]:
```
pars = {omega:1+0.1,omega0:1,a:0.2,g:.2}
ode = [phid,rhs.subs({Phi:phi,diff(Phi,t):phid}).subs(pars)]
times = srange(0,80,0.001)
ics = [0.0,2.1]
sol = desolve_odeint(ode,ics,times,[phi,phid])
line( zip(times,sol[::1,0]),figsize=(10,2) ) + \
 plot( r_szczegolne.subs(pars), (t,0,80),color='red') +\
 plot( (a*sin(omega*t)).subs(pars), (t,0,80),color='gray')
```

Out[59]:

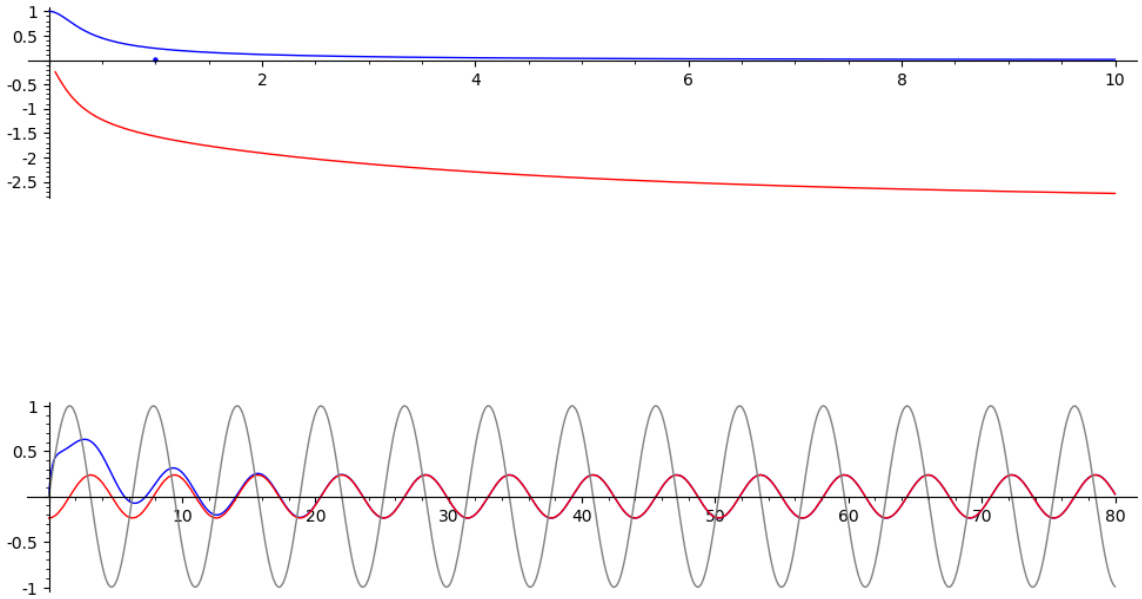


```
In [60]: rhs = solve(osc,diff(Phi(t), t, 2))[0].rhs()
         #@interact
         def plot_A_traj(g_=slider(0.01,2.2,0.01,label="$\gamma$",default=0.2),\
                         omega_=slider(0.01,5.134,0.01,label="$\omega$",default=1.4)):
             pars = {g:g_,a:1,omega0:1}
             print ( 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()

In [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*:

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

Out[62]:

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

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

Out[63]:

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

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

Out[64]:

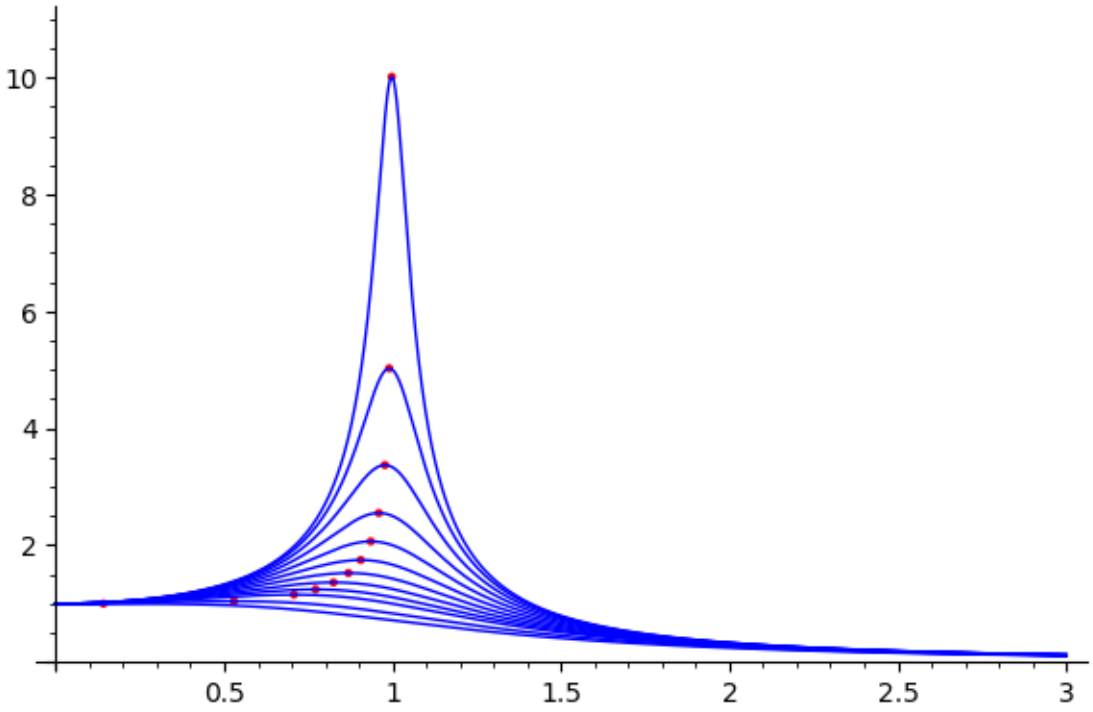
$$\sqrt{-\frac{1}{2}\gamma^2+\omega_0^2}$$

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

Out[65]:

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

In [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) ),colo
    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:

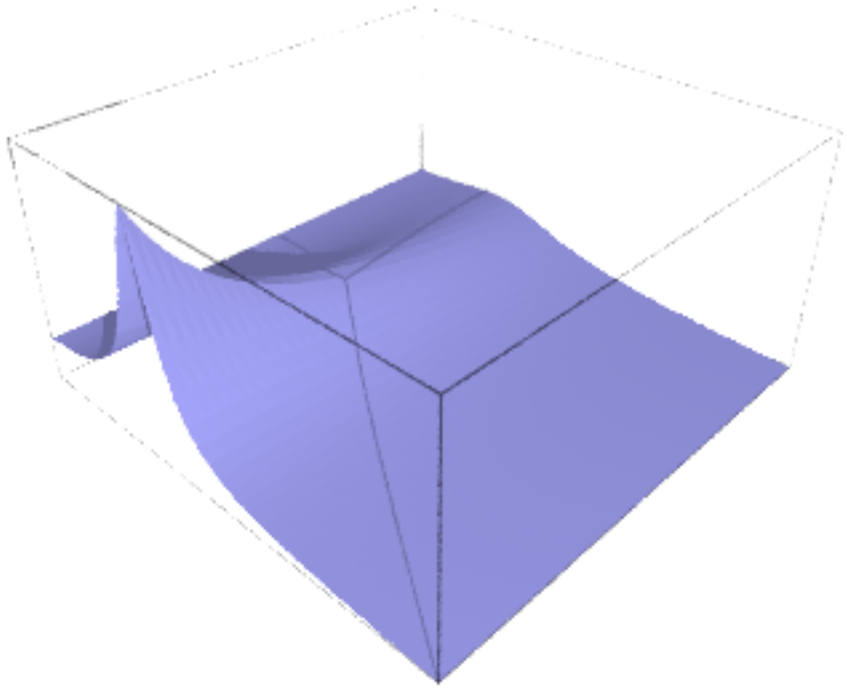
$$-\gamma^2 + 2\omega_0^2 = (\sqrt{2}\omega_0 - \gamma)(\sqrt{2}\omega_0 + \gamma)$$

for

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

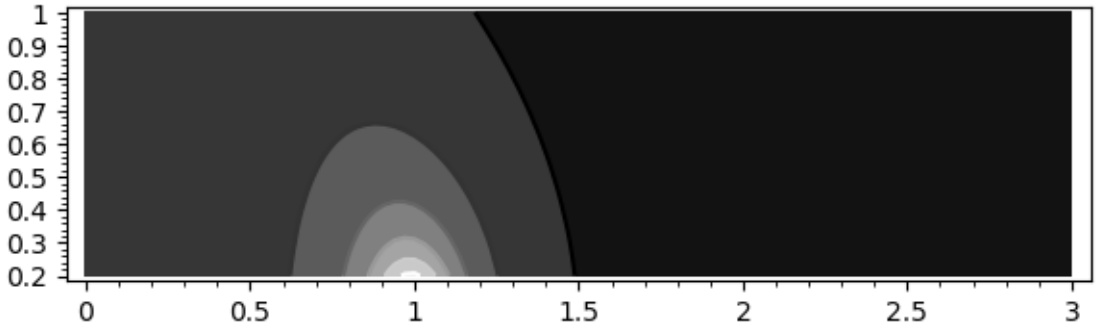
the resonance disappears - i.e. there is not maxima in $A(\omega)$ dependence.

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



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

Out[68]:



4.4.2 Absorbed power

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

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

Out[69]:

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

Dissipated power is equal to: force times the velocity, so:

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

We want to integrate the power over one period of oscillations

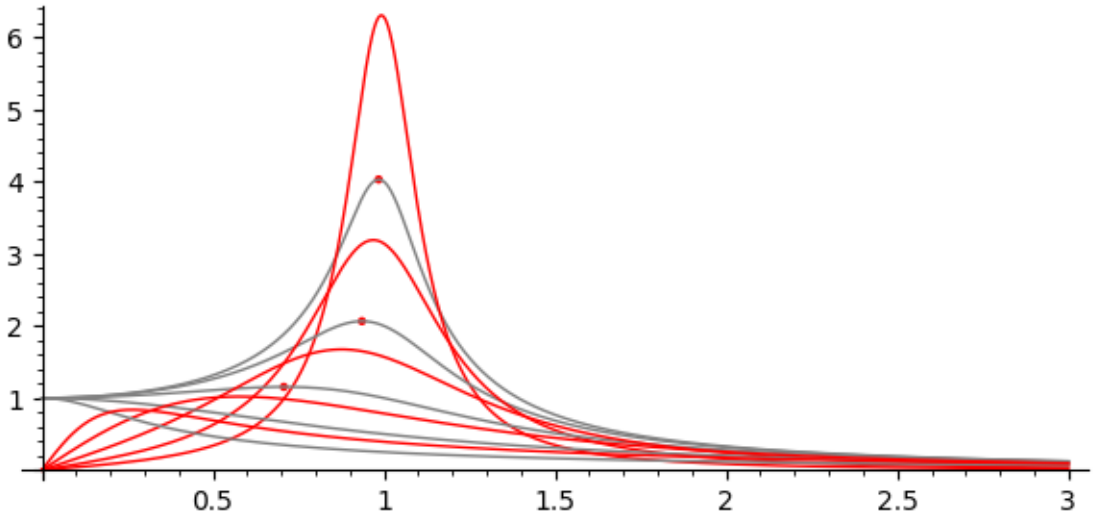
In [71]:
```
showmath( P.integrate(t,0,2*pi) )
```

Out[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.

```
In [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
             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))
```



5 Particle in one-dimensional potential well

5.1 Period of oscillations in potential well

Dynamics of a particle of mass m moving in one dimension OX is described the Newton equation

$$m\ddot{x} = m\dot{v} = F(x) = -U'(x),$$

where $F(x)$ is a force acting on tha particle and $U(x)$ is potential energy of the particle. We assume that there are no dissipative forces. Therefore, as we know from the previous section, the total energy E of the particle is conserved, i.e.,

$$\frac{mv^2}{2} + U(x) = E = const. \tag{8}$$

If we treat the velocity $v = \frac{dx}{dt}$ as an independent variable, it is implicit equation of the orbit (the phase curve) in **phase space** (x, v) of the particle with energy E. This equation can be rewritten in the form

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

It is a first-order differential equation and can be solved by the method of variable separation. The result reads

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

Here, a is the initial position $x(0) = a$ and b is the final position $x(t) = b$ of the particle. The time t is time for moving the particle from the point a to the point b under the condition that $E \geq U(x)$ for all $x \in (a, b)$.

This is a useful formula which allows to calculate period of oscillations of the particle in a potential well.

In this section we will consider a class of potentials in the form

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

where n is a positive real number.

These potentials are similar: they are bounded from below, have only one minimum at $x = 0$ and tends to infinity when $x \to \pm\infty$. In such potential the particle motion is bounded and the particle oscillates between two symmetrical positions x_0 and $-x_0$ which in general depend on the total energy E and are determined by the equation

$$U(\pm x_0) = E$$

Because of symmetry, the period T of oscillations can be determined by the equation

$$T = 4\sqrt{\frac{m}{2}}\int_0^{x_0}\frac{dx}{\sqrt{E-U(x)}} \tag{11}$$

This class of potentials is simple but nevertheless analysis of T in dependence of the index n and the total energy E is very interesting and instructive.

We will use computer algebra and numerical methods to investigate properties of motion is such potential wells.

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

```
In [2]: t = var('t')
        m = var('m')
        A = var('A')
        assume(A > 0)
        assume(m > 0)
        y = function('y')(t)
        de = m*diff(y,t,2) + 2*A*y == 0
        showmath( desolve(de, y,ivar=t) )
```

Out[2]:

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

It is an analytical solution of the Newton equation in the case when $n = 2$ (a harmonic potential).

5.2 Particle in potential x^2

For $n = 2$ the system is a harmonic oscillator:

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

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

To obtain the integration limit x_0 in the formula for the period of oscillations, we must solve the equation:

$$U(x) = E$$

So for the Ax^2 potential, we have:

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

Out[4]:

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

These formulas describe the values of the oscillator's extremal positions for a given energy. Let's put them into the formula for T: :

```
In [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)
```

Out[5]:

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

We see that the period T does not depend on energy of the oscillator. It means that it does not depend on the initial conditions because the total energy of the particle depends on them. In turn, it means that it does not depend on the distance between the points $-x_0$ and x_0. It seems to be unusual behavior: time to travel from -1 to 1 and back is the same as time to travel from -10000 to 10000 and back. In the second case the distance is much, much longer but time is the same. This is an exceptional property valid only for the harmonic potential!

5.3 Particle in $|x|^n$ potential

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

$$T = 4\sqrt{\frac{m}{2}} \int_0^{x_0} \frac{dx}{\sqrt{E - Ax^n}}$$

or in the equivalent form:

$$T = 4\sqrt{\frac{m}{2}} \frac{1}{\sqrt{E}} \int_0^{x_0} \frac{dx}{\sqrt{1 - Ax^n/E}}$$

This integral can be transformed to a dimensionless form by substitution

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

It 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 integration variable to y. To do this, we use SAGE to transform the expression under integral in the following way:

```
In [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)
```

Out[6]:

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

and we substitute:

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

Out[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:

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

Out[8]:

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

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

Out[9]:

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

Finally, we obtain:

$$T = 4\sqrt{\frac{m}{2}}\frac{1}{A^{1/n}}E^{\frac{1}{n}-\frac{1}{2}}\int_0^{y_0}\frac{dy}{\sqrt{1-y^n}}$$

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

We still need to calculate the upper limit y_0 of integration. In the integral, the upper limit is the position in which the total energy is the potential energy:

$$U(x) = E$$

In this case

$$Ax^n = E.$$

By changing the variables we get:

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

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

That is, the integration limit is $y_0 = 1$.

Therefore the period of oscillations is given by the relation:

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

We note that only for the case $n = 2$, the period T does not depend on E (i.e. on initial conditions, i.e. on the distance). In other cases it depends on the total energy E, i.e. it depends on initial conditions, i.e. it depends on the distance between the points $-x_0$ and x_0.

The above equation shows how much time the particle needs to travel the distance for one oscillation in dependence on E and in consequence on the distance: If energy E is higher then the distance $4x_0$ is longer.

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

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

```
[A > 0, E > 0, E is real, n is integer, n >= 0, a is integer, a > 0]
```

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

```
Out[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.

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

```
Out[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 $U(x) = ax^n$.

```
In [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:

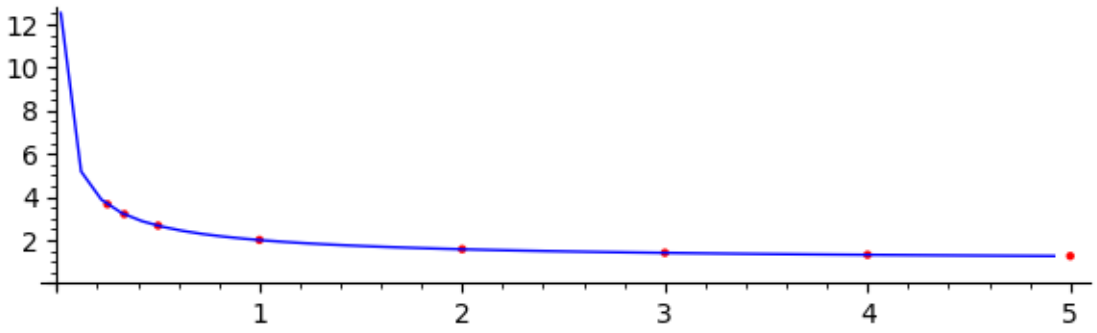
```
In [15]: showmath( integr_list_analytical )
```

```
Out[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:

```
In [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:

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

```
Out[17]: 1
```

```
In [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

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

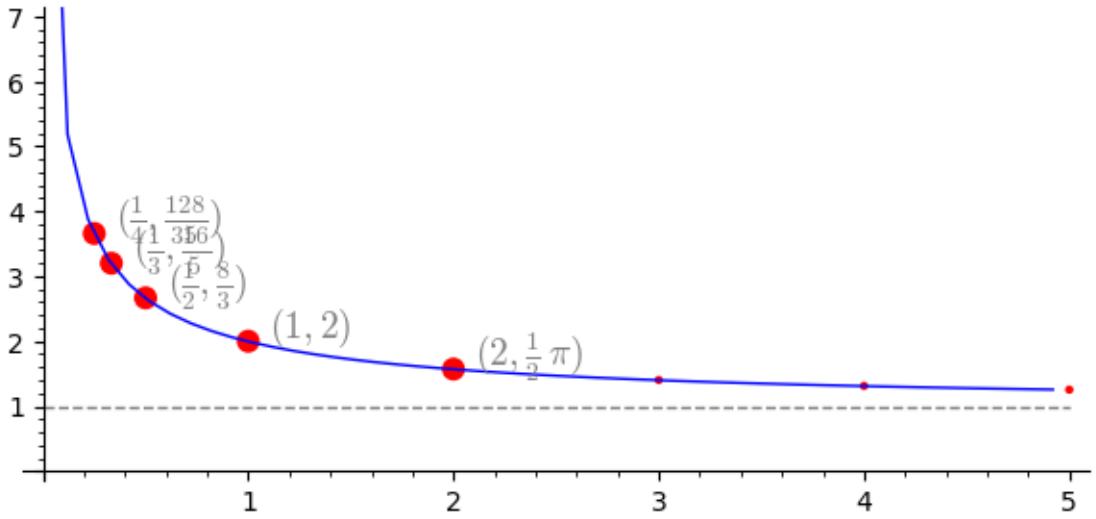
```
Out[19]:
```

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

```
In [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
                 p.append( point ( (x,y),size=75,color='red' ) )
             return sum(p)
```

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

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



5.4 Numerical convergence

The integral

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

seems to be divergent for n :

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

Out[23]:

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

However, choosing the right algorithm gives the correct result:

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

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

lets check it out with an exact formula:

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

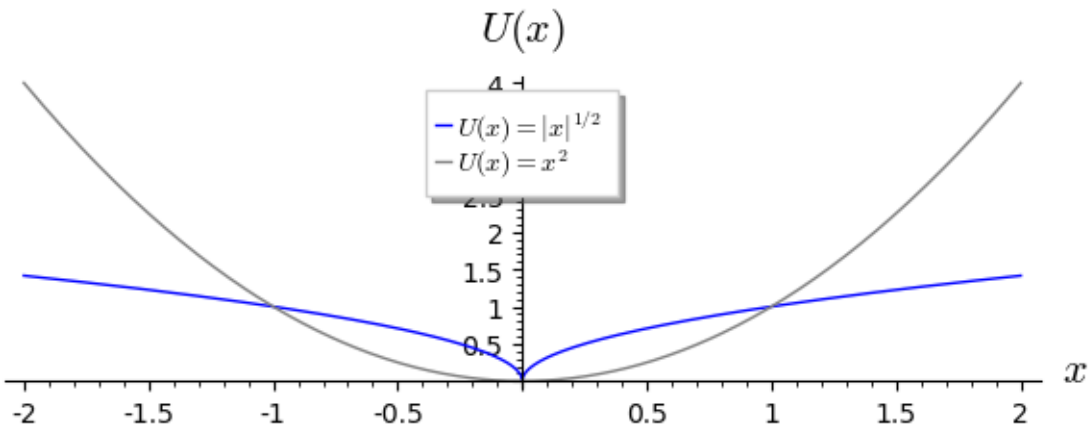
```
Out[25]: 3.65714285714286
```

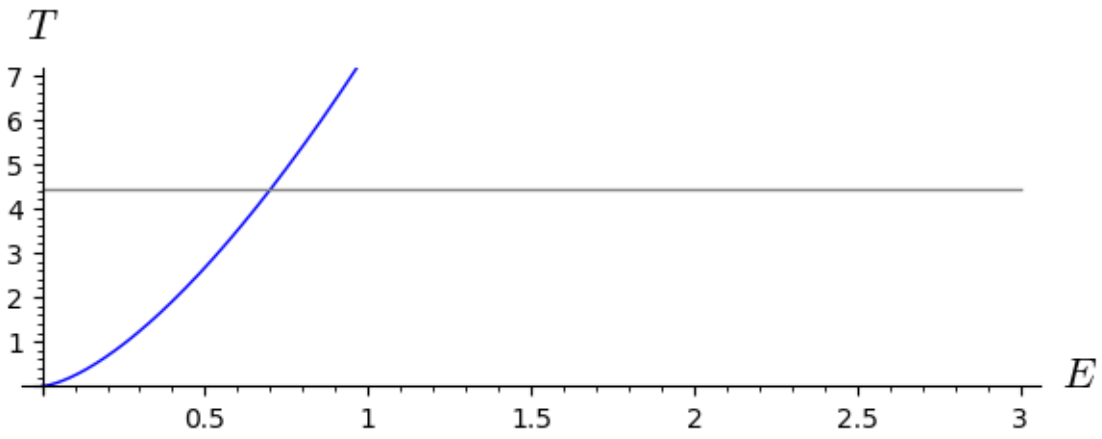
Indeed, we see that carefull numerical integration gives finite result.

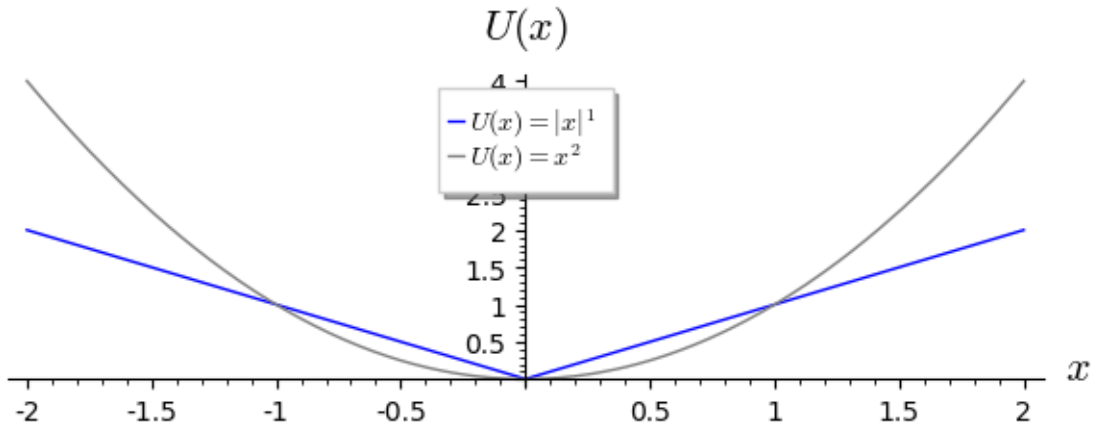
5.5 The dependence of period on energy for different n.

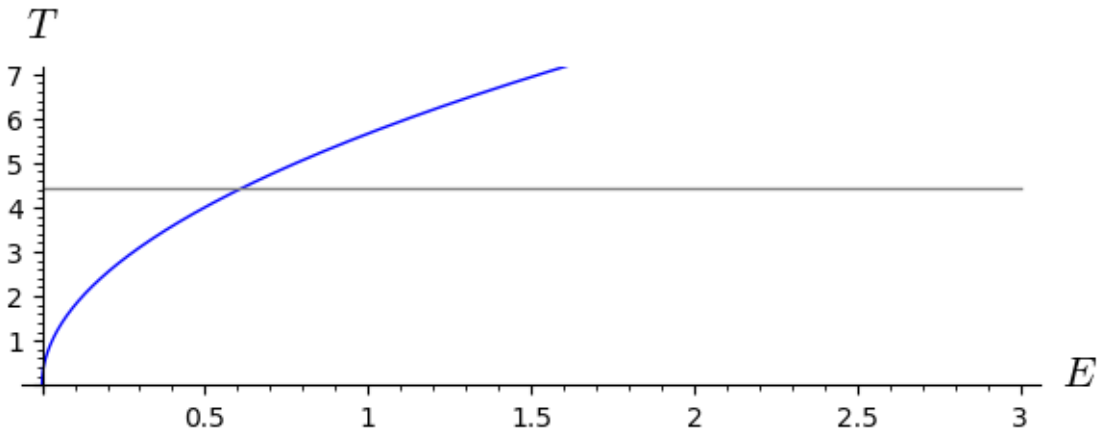
```
In [26]: var('E x n')
         def draw_E(n,figsize=(6,2.5)):
             p = []
             p2 = []
             p.append( plot(abs(x)^n,(x,-2,2),\
                            ymin=0,ymax=4,legend_label=r"$U(x)=|x|^{%s}$" % n )   )
             p.append( plot( (x)^2,(x,-2,2),\
               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

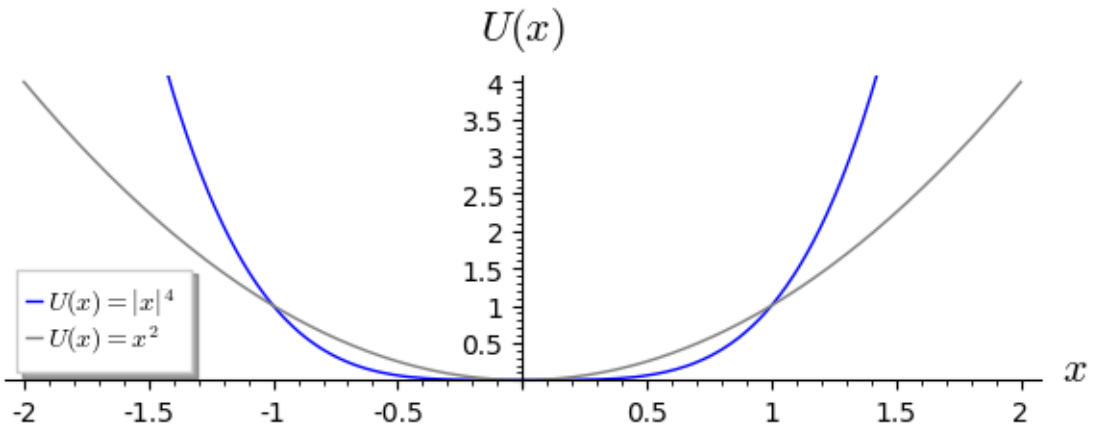
```
In [28]: import os
         if 'PDF' in os.environ.keys():
             draw_E(1/2)
             draw_E(1)
             draw_E(4)
```

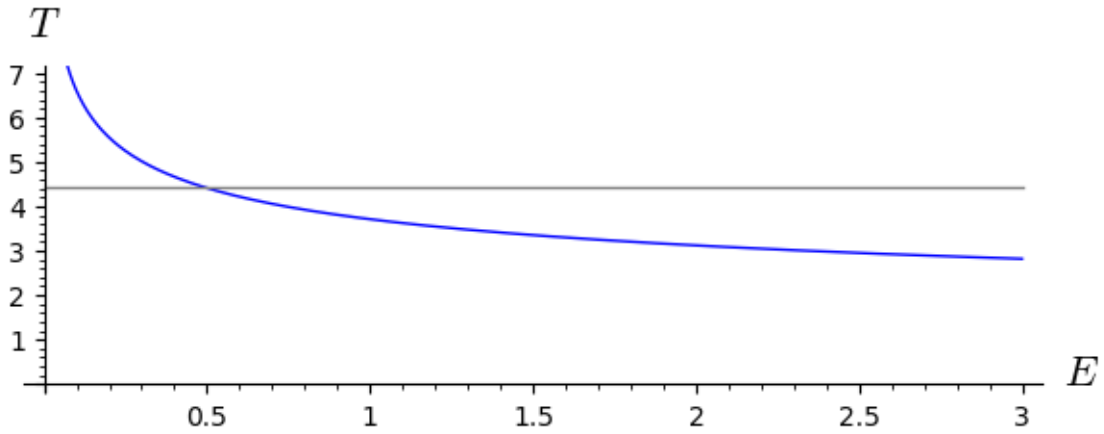



T
7
6
5
4
3
2
1
0.5
1
1.5
2
2.5
3
E


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


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



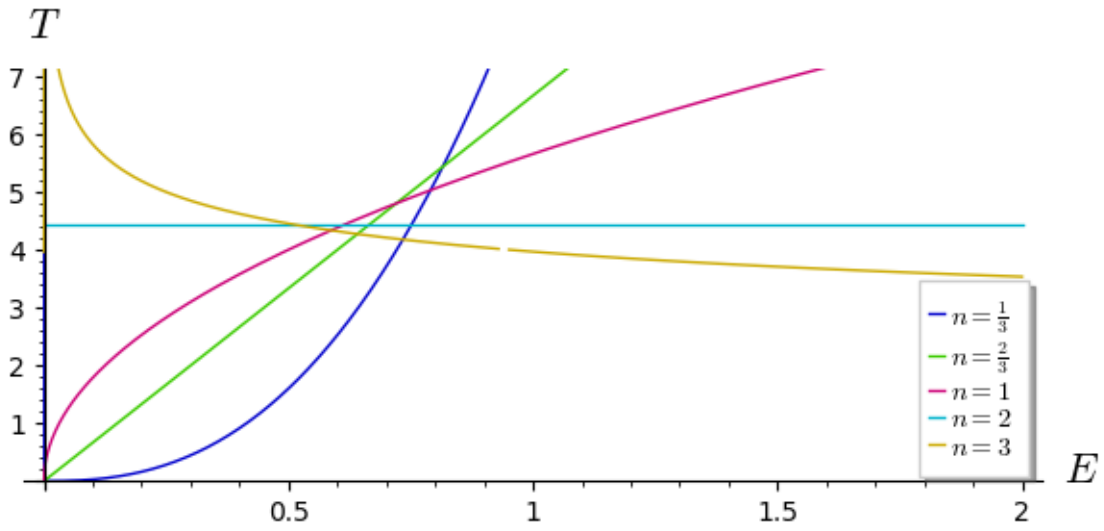


We can plot the dependence of period T on energy (i.e. amplitude) $T(E)$ for different values of n. In figure belowe we see that if $n > 2$ then oscillations are faster as energy grows. On the other hand if $n < 1$, oscillations are getting slower with growing energy.

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

```
In [29]: 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) )
```

Out[29]:



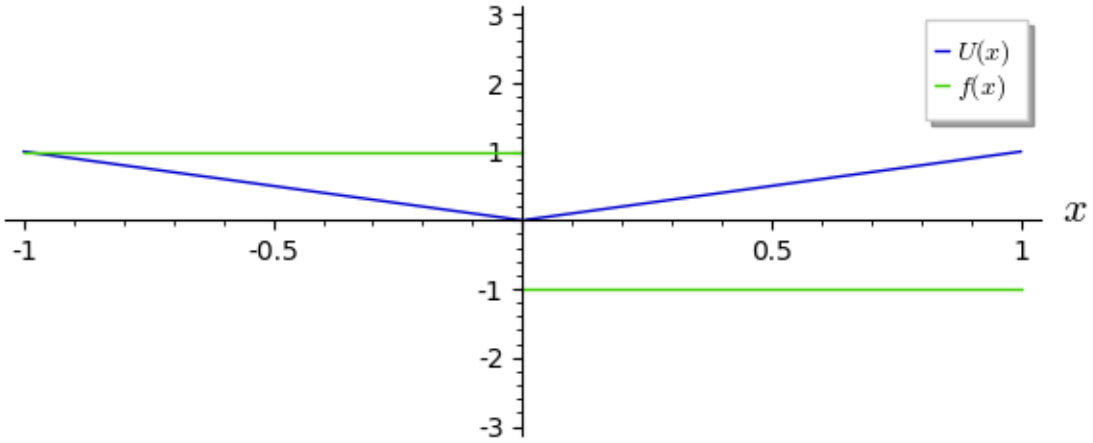
5.6 Numerical integration of equations of motion

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

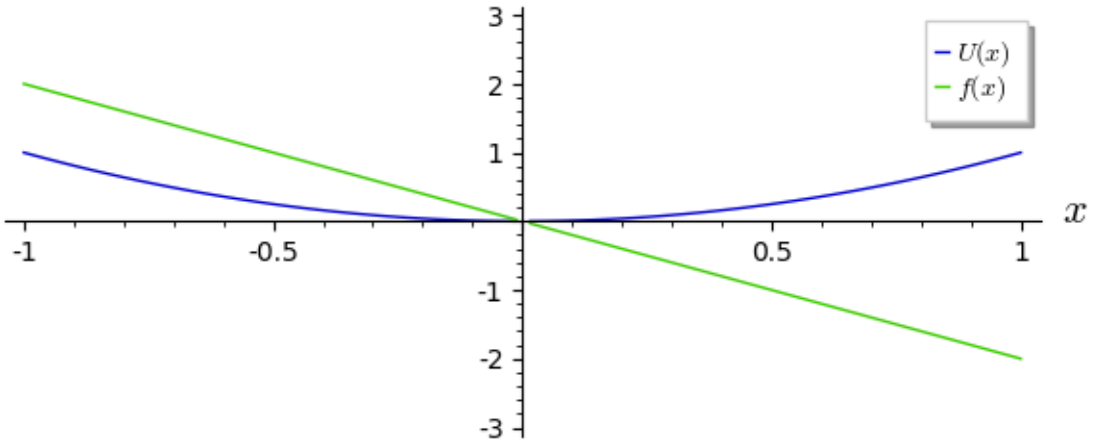
```
In [30]: def plot_Uf(n_):
             U(x) = abs(x)^(n_)
             plt = plot( [U(x),-diff( U(x),x)],(x,-1,1),\
                        detect_poles='show',ymin=-3,ymax=3,
                       legend_label=[r"$U(x)$",'$f(x)$'])
             plt.axes_labels([r"$x$",r"$U(x)=|x|^%s$"%latex(n_)])
             show(plt,figsize=(6,3))
```

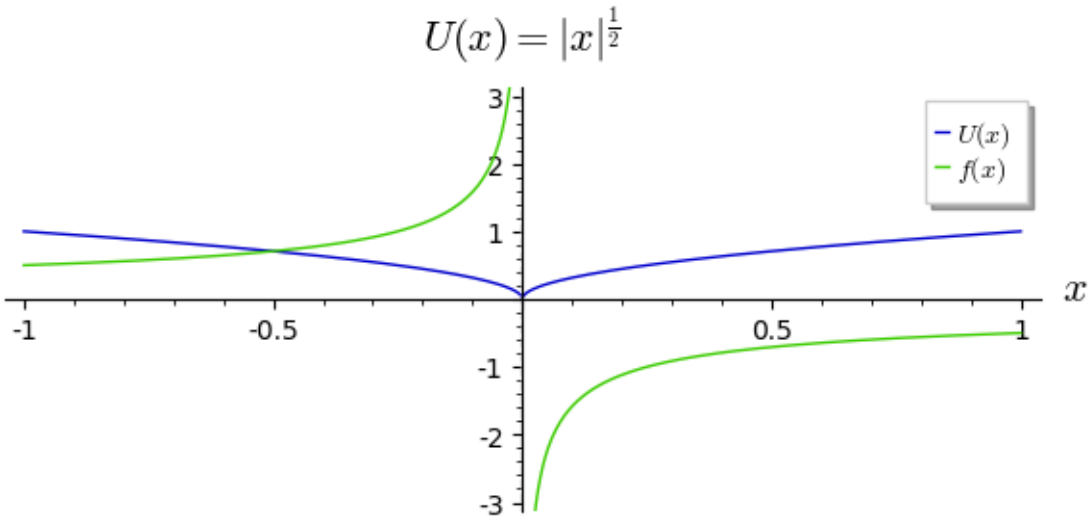
```
In [32]: if 'PDF' in os.environ.keys():
             plot_Uf(1)
             plot_Uf(2)
             plot_Uf(1/2)
```

$U(x) = |x|^1$



$U(x) = |x|^2$





We can see that for $n \geq 1$ the force and potential are continuous. 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 $x = 0$.

Problems with numerical integration 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:

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

```
In [33]: var('x',domain='real')
         var('v t')
         eps = 1e-6
         U(x) = (abs(x)+eps)^(1/2)
         showmath( U.diff(x).expand().simplify() )
```

Out[33]:

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

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

```
In [34]: w0 = SR.wild(0)
         w1 = SR.wild(1)
         f = -U.diff(x).subs({w0*w1/abs(w1):w0*sign(w1)})
```

```
In [35]: showmath( f(x) )
```

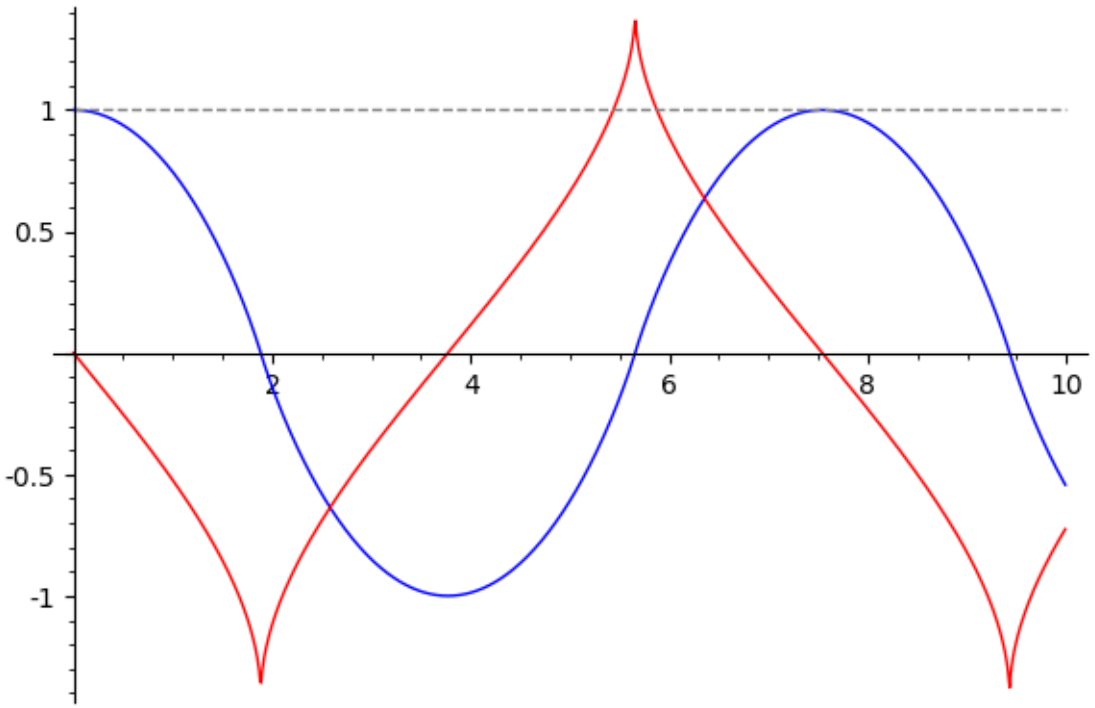
Out[35]:

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

```
In [36]: ode_pot = [v,f(x)]

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

```
In [37]: 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')
```

Out[37]:



We can evaluate the period T from the trajectory obtained via numerical solutions. For this purpose one might need an interpolation of numerical table returned by `desolve_odeint`:

```
In [38]: 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()
```

In [39]:
```
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
```

Out[39]:

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

We find numerically a period of trajectory:

In [40]:
```
T = find_period( sol[:,0],t_lst)
T
```

Out[40]: 7.54250742200179

Exact results for comparison:

In [42]:
```
# 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
                        for n_ in [1/4,1/3,1/2,2/3,1,2,3,4,5] ] )
```

Out[42]:

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

5.7 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 $T(E)$.

In [43]:
```
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) )
```

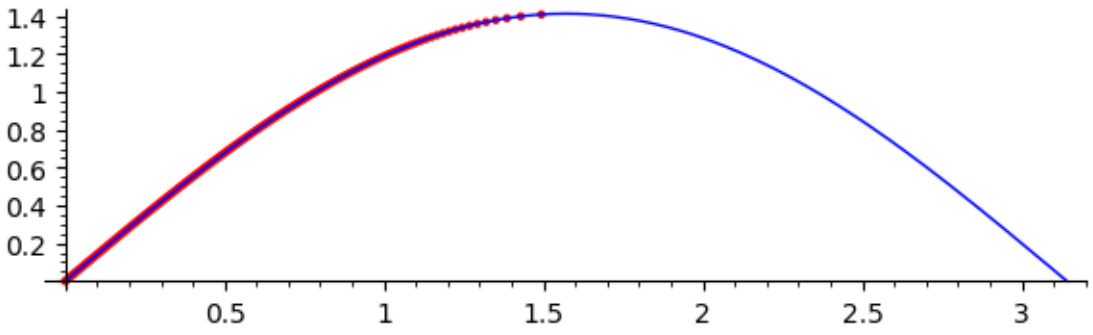
Out[43]:

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

In [44]:
```
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)]
```

In [45]:
```
point(t_lst ,color='red')+\
 plot(sqrt(2)*sin(x),(x,0,pi),figsize=(6,2))
```

Out[45]:



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

6 Particle in multistable potentials

6.1 Phase portrait for a one-dimensional system

The Newton equation for a point particle in a one-dimensional potential $U(x)$ can be written as a set of two first-order differential equations:

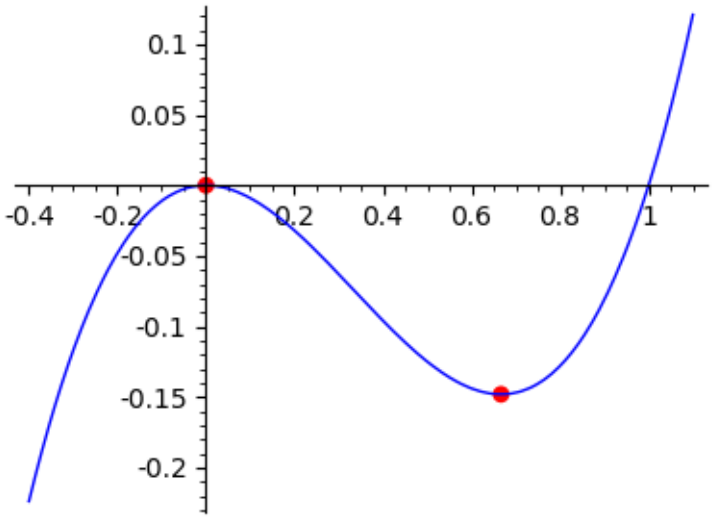
$$\begin{cases} \dot{x} = v \\ \dot{v} = -\frac{1}{m}U'(x) \end{cases}$$

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

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

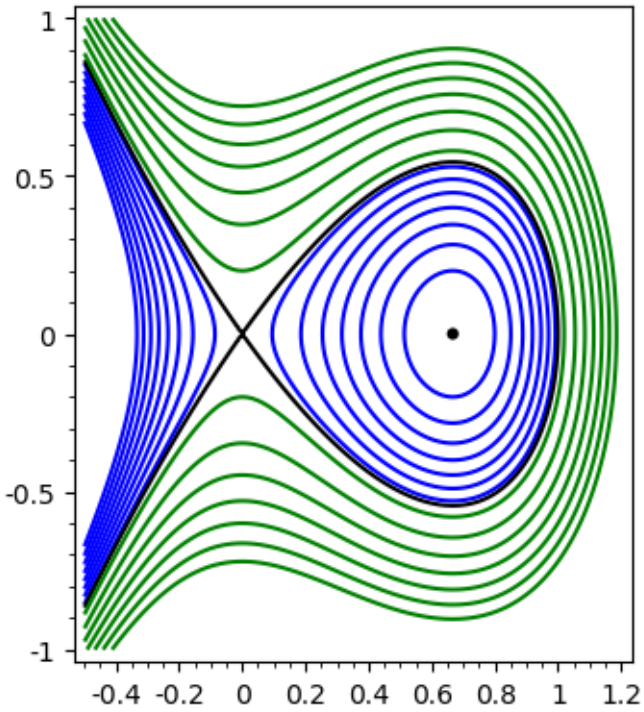
In [1]:
```
var('v')
m = 1
U(x) = x^3-x^2
xmax,xmin = sorted([s_.rhs() for s_ in solve(U.diff(x)==0,x)])
Emin = U(xmin)
Etot = 1/2*m*v^2 + U(x)

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)
```

Out[1]:



The potential $U(x) = x^3 - x^2$.

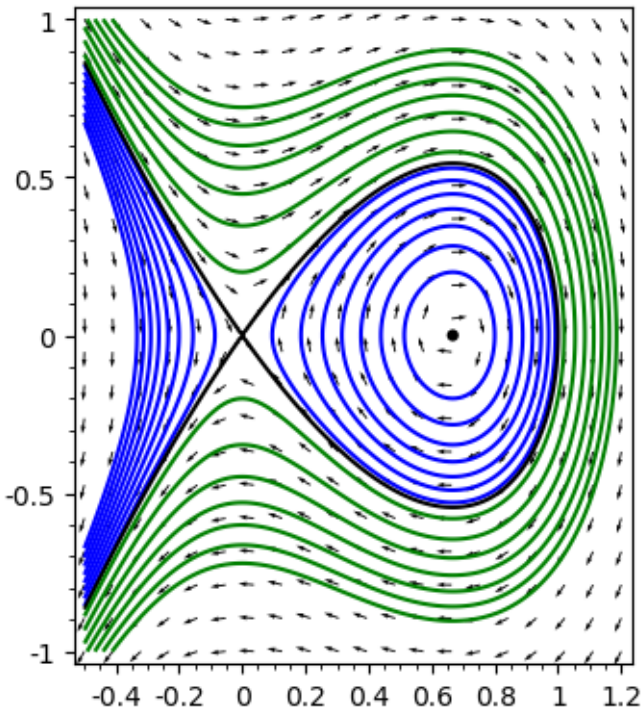
```
In [2]: pkt = point((xmin,0),size=20,color='black')
        plt =sum([ implicit_plot(Etot==E0,(x,-1/2,1.2),(v,-1,1),color='blue')\
                  for E0 in srange(Emin,0.0,0.02)])
        plt +=implicit_plot(Etot==0,(x,-1/2,1.2),(v,-1,1),color='black') +pkt
        plt +=sum([ implicit_plot(Etot==E0,(x,-1/2,1.2),(v,-1,1),color='green')\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



The phase curves (x, v) depends on initial conditions (x_0, v_0). There are two types of phase curves: closed (periodic motion of the particle in a bounded interval) and open (the motion is unbounded: the particle can escape to -infinity or can return from -infinity.

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

```
Out[3]:
```



The vector field shows direction of motion of the particle on the (x, v)-plane.

6.3 Harmonic oscillations limit for one-dimensional systems

Consider a conservative one-dimensional system. In this case, the force $f(x)$ can always be represented as gradient of the potential $U(x)$, namely,

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

Consider a certain potential that has a minium at some point x_0. The necessary condition for minimum of the function is its zero first derivative at this point. Let's expand the potential in the Taylor series around the minimum. We obtain:

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

For small deviation from the minimum this series can be approximated by the function

$$U(x) = \frac{1}{2}k(x - x_0)^2,$$

The Newton equation for such motion is as follows:

$$m\ddot{x} = ma = F = -U'(x) = -k(x - x_0)$$

For the new variable $y = x - x_0$ it takes the form

$$m\ddot{y} = -ky$$

This is the already known equation for the harmonic oscillator with the shifted equilibrium point x_0.

Now, let us return to the system with the potential $U(x) = x^3 - x^2$.

```
In [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
```

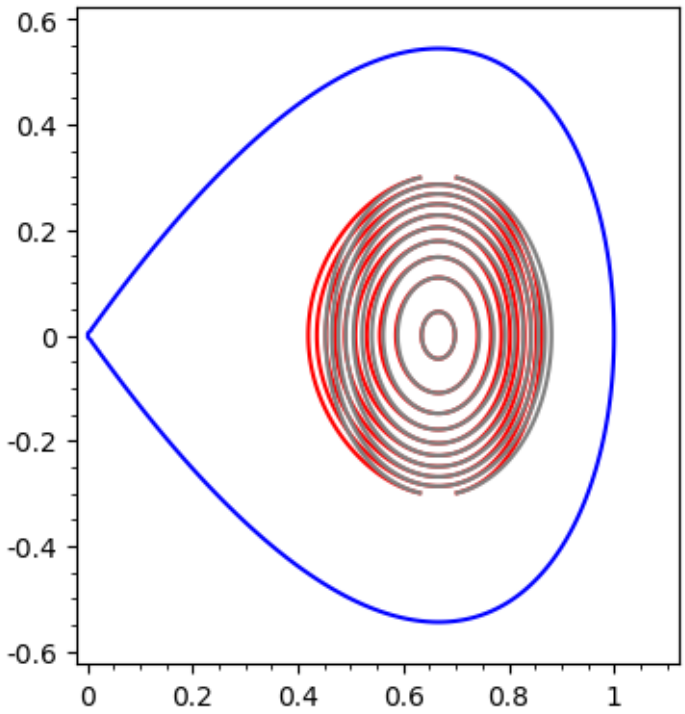
```
In [5]: Emin = Etot(x=xmin,v=0)
        Emin
```

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

Let's have a look at the trajectories for the exact system with $U(x) = x^3 - x^2$ and the linearized system with $U(x) = (1/2)x^2$. The blue line below is a separatrix - i.e. a solution with $E = 0$

```
In [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
```

```
Out[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.

6.4 Time to reach the hill

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:

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

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

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

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

```
Out[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.

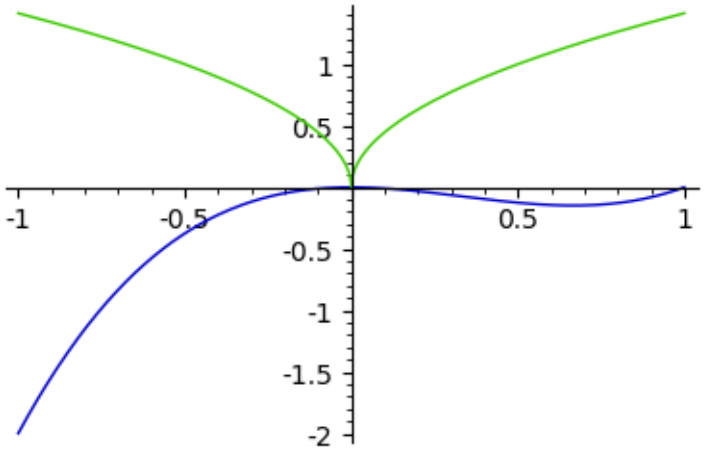
Let analyze it:

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

we can plot velocity and potential:

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

Out[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:

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

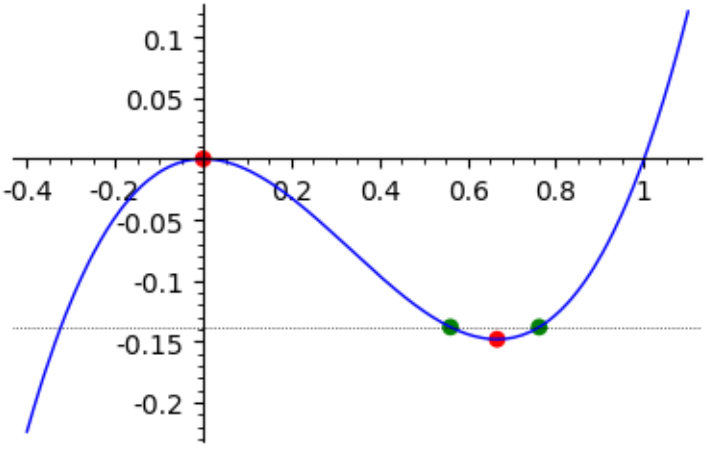
```
Out[11]: 1.41421356237310
```

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.

```
In [12]: dE = 0.01

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

Out[13]: -0.138148148148148

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

Out[14]: (0.560919215938882, 0.762206802325432)

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

Out[15]: 0.0223211406255026

In [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

Out[16]:



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

```
Out[17]: (3.1721596231106926, 2.332448399655124e-06)
```

```
In [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
```

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

```
Out[19]: 4.48611131060635
```

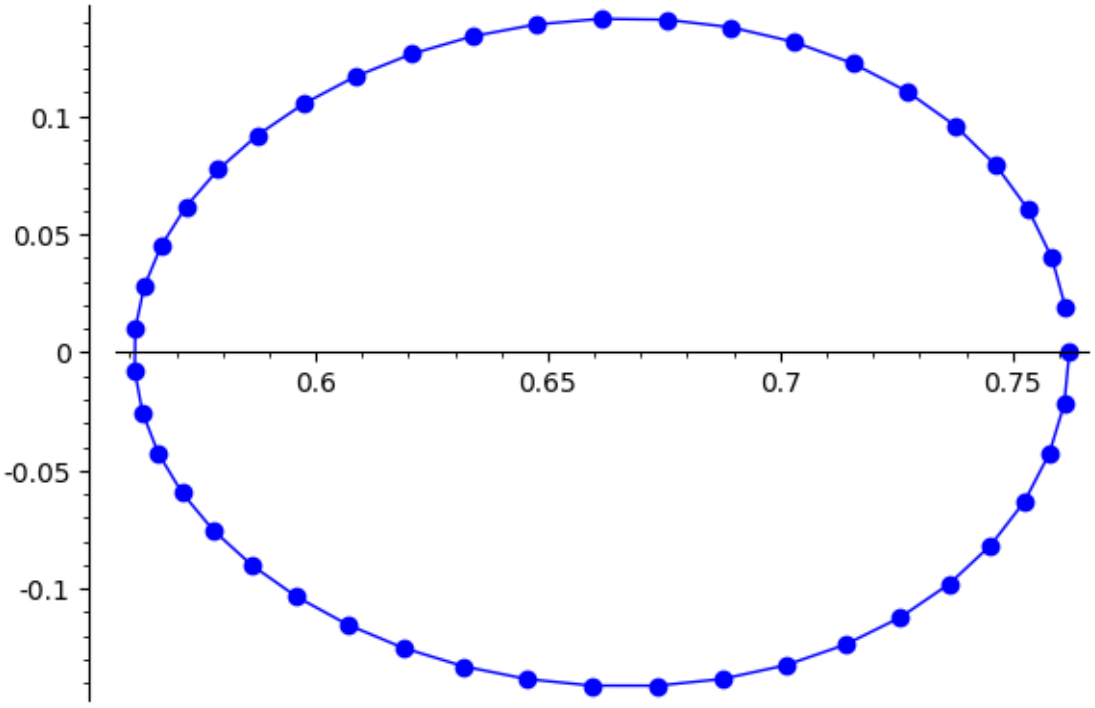
```
In [20]: omega = sqrt(U(x).diff(x,2).subs(x==xmin.n()))
         period_harm = 2*pi.n()/omega
         period_harm
```

```
Out[20]: 4.44288293815837
```

```
In [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])
```

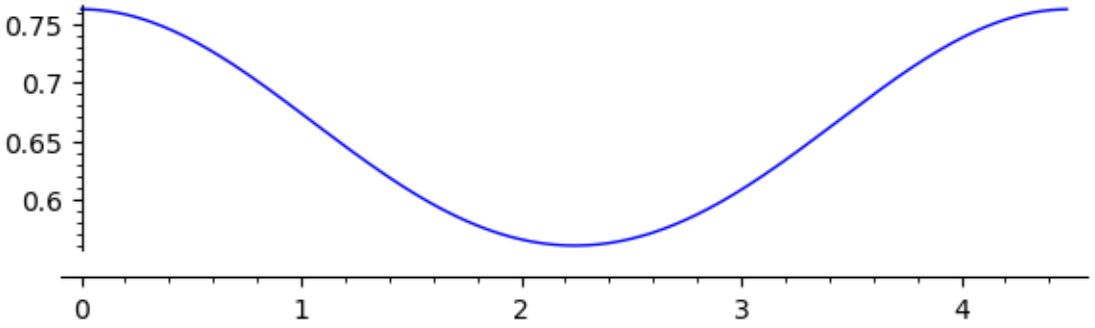
```
In [22]: line(sol[::10,:],marker='o')
```

```
Out[22]:
```



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

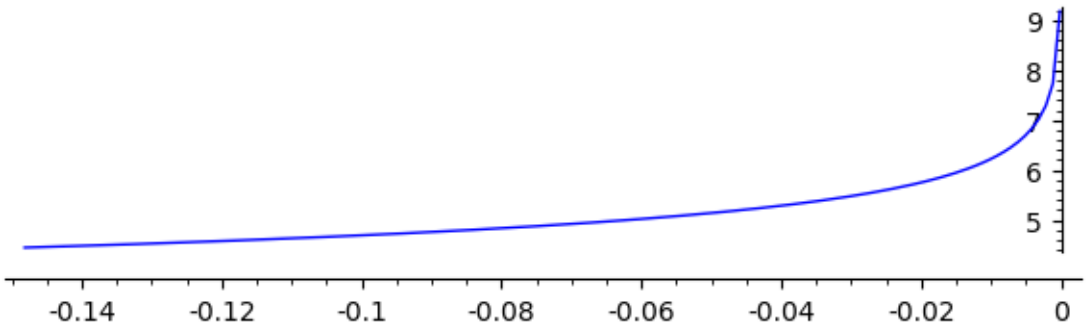
Out[23]:



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

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

Out[25]:



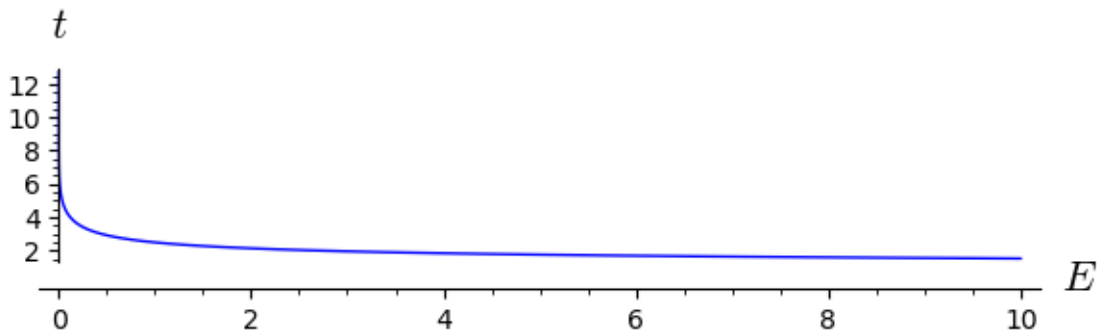
In [26]:
```
def t_hill(E0):
    m = 1

    x2, = [s_.rhs().n().real() for s_ in solve(U(x)==E0,x) if s_.rhs().n().imag().a

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

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

Out[27]:
```
1.53679467633445
```

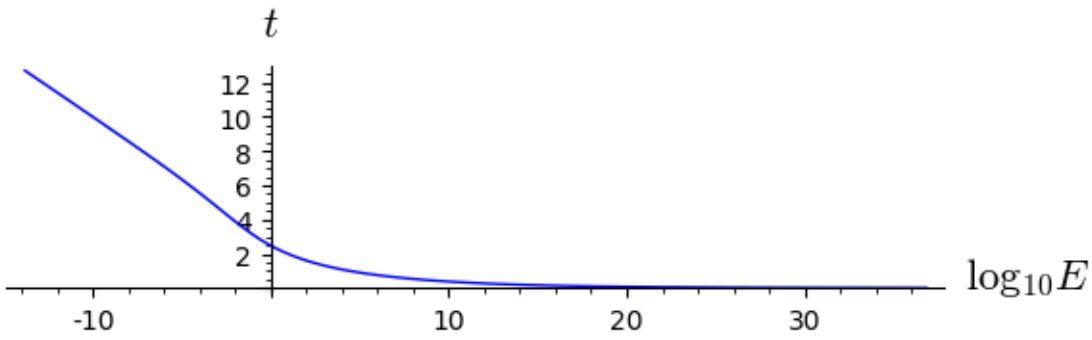
In [28]:
```
import numpy as np
t_E = [(E_,t_hill(E_)) for E_ in np.logspace(-6,1,120)]
line(t_E,axes_labels=['$E$','$t$'], figsize=(6,2))
```

Out[28]:



```
In [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))
```

Out[29]:



6.5 Excercise 1

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

6.6 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.

7 d'Alembert with computer algebra system

7.1 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.

7.2 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`

7.2.1 Example - step by step

Let a denote some generalized coordinate in out dynamical system:

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

```
Out[1]: a
```

We add symbols representing its derivatives with nice LaTeX representation:

```
In [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.

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

```
A(t)
```

Now, we can do following:

```
In [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:

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

we change variables to explicit function of time:

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

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

and calculate derivative:

```
In [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`

```
In [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 $\dot{a}$:

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

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

7.2.2 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:

In [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='_po
        """
        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
```

In [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
```

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

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

```
In [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:

```
In [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
```

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

```
4*yd
```

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

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

7.3 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:

```
In [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())
```

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

```
[x, y]
```

```
[dx, dy]
```

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

```
In [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:

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

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

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

```
[x, y]
```

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

```
[dx, dy]
```

Having constraints, one can obtain its differential form:

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

```
In [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:

```
In [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:

```
In [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:

```
In [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:

```
In [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:

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

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

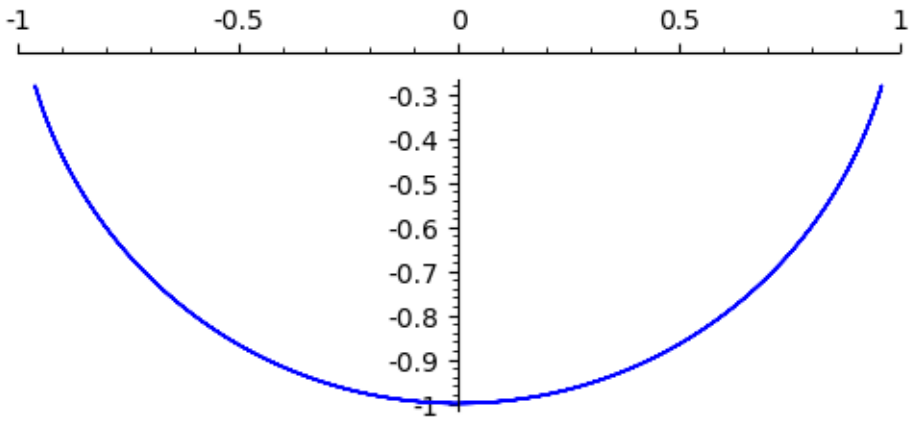
```
In [29]: sol[0][1].rhs()
```

```
Out[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):

$$\frac{dx}{dt} = \dot{x} \tag{12}$$
$$\frac{dy}{dt} = \dot{y} \tag{13}$$
$$\frac{d\dot{x}}{dt} = \frac{gxy - (\dot{x}^2 + \dot{y}^2)x}{x^2 + y^2} \tag{14}$$
$$\frac{d\dot{y}}{dt} = -\frac{gx^2 + (\dot{x}^2 + \dot{y}^2)y}{x^2 + y^2} \tag{15}$$

```
In [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:

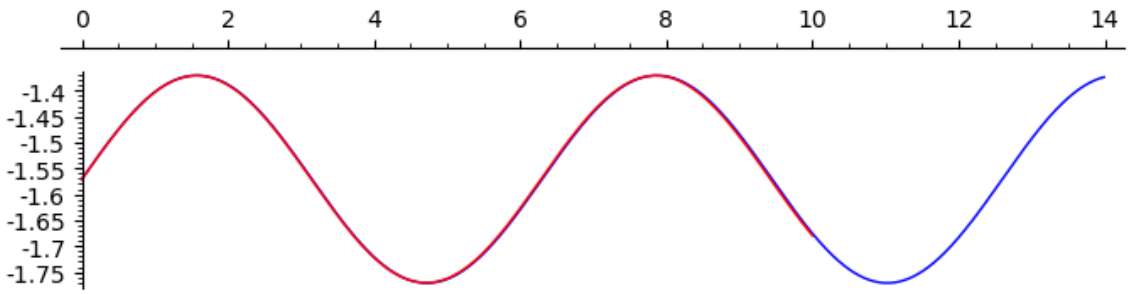
$$\phi = -\pi/2 + 0.2\sin(\omega_0 t),$$

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

```
In [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')
```

Out[31]:



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

```
In [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
```

7.3.1 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:

```
In [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
```

```
In [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:

```
In [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:

```
In [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)
```

```
In [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:

```
In [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:

```
In [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:

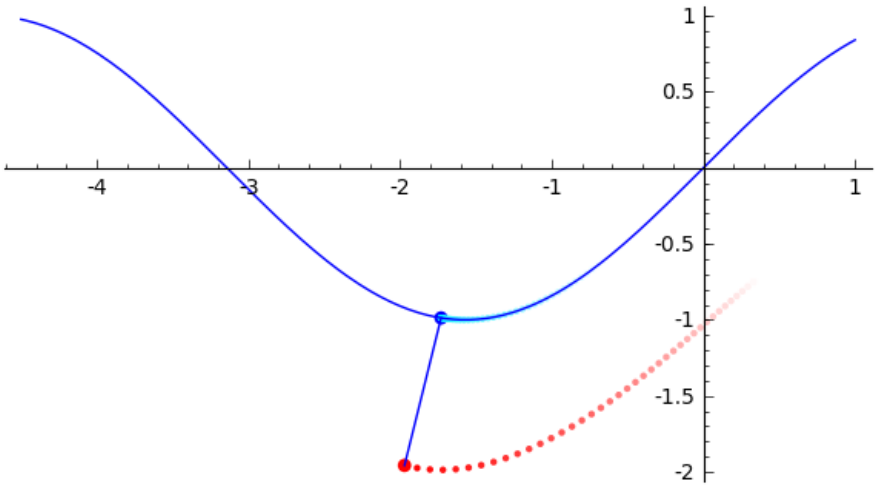
```
In [40]: taylor(-g/l*cos(phi),phi,-pi/2,1).show()
```

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

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

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

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



Pendulum

8 Pendulum on $\sin(x)$.

8.1 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 $y = f(x)$

We will need some helpers for the algebra:

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

```
In [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.

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

Out[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: $\delta x, \delta\phi$.

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

Now we can write d'Alembert principle:

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

In [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))
```

Out[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:

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

Out[6]:

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

In [7]: showmath(w2)

Out[7]:

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

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

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

Out[8]:

$$0$$

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

Out[9]:

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

We obtain mathematical pendulum if

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

Out[10]:

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

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

Out[11]:

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

8.2 Numerical analysis of the system

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

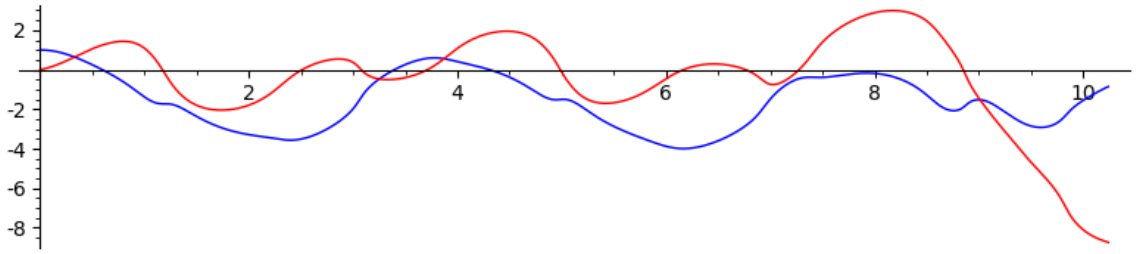
In [12]:
```
import numpy as np
```

In [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 295 ms, sys: 28.4 ms, total: 324 ms
Wall time: 319 ms
```

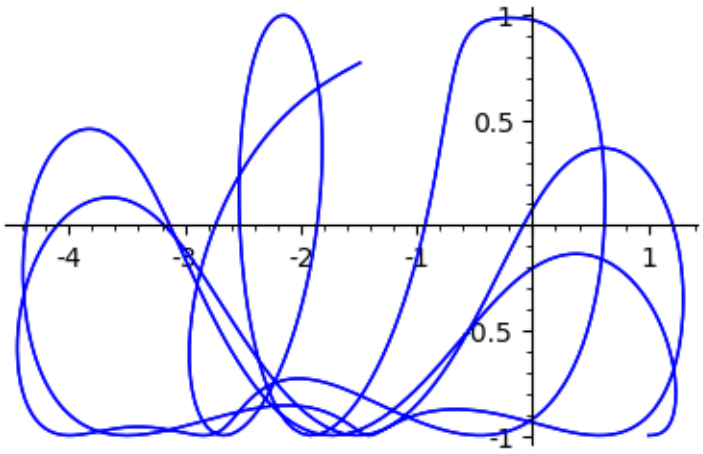
In [14]:
```
line( zip(times,sol[::1,0]),figsize=(8,2) )+\
 line( zip(times,sol[::1,1]),color='red')
```

Out[14]:



In [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 )
```

Out[15]:



8.2.1 Visualization

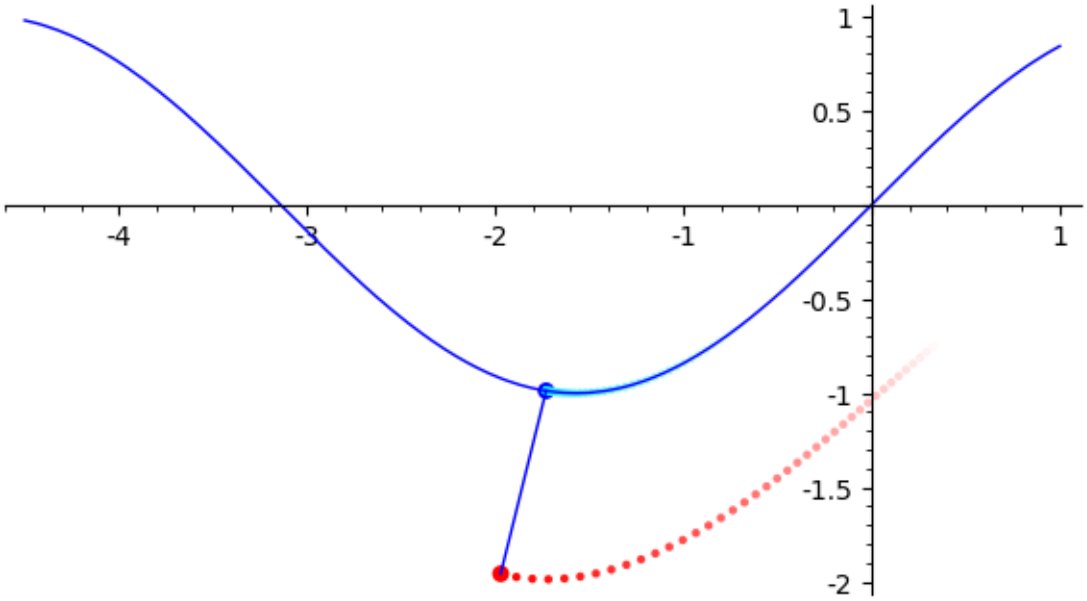
It is helpfull to write simple function displaying configuration of the system for given set of variables.

```
In [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
             trace2 = sum([point((x,f(x)),hue=(.51,(i)/n,1)) for i,(x,phi) in enumerate(sol

             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:

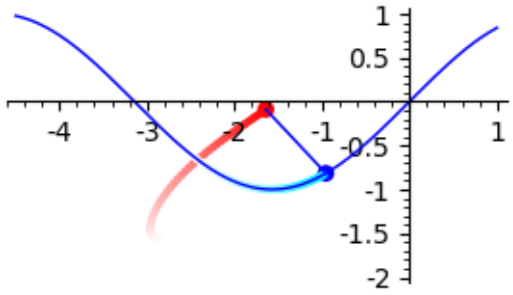
```
In [17]: draw_system(120)
```

Out[17]:



We can animate:

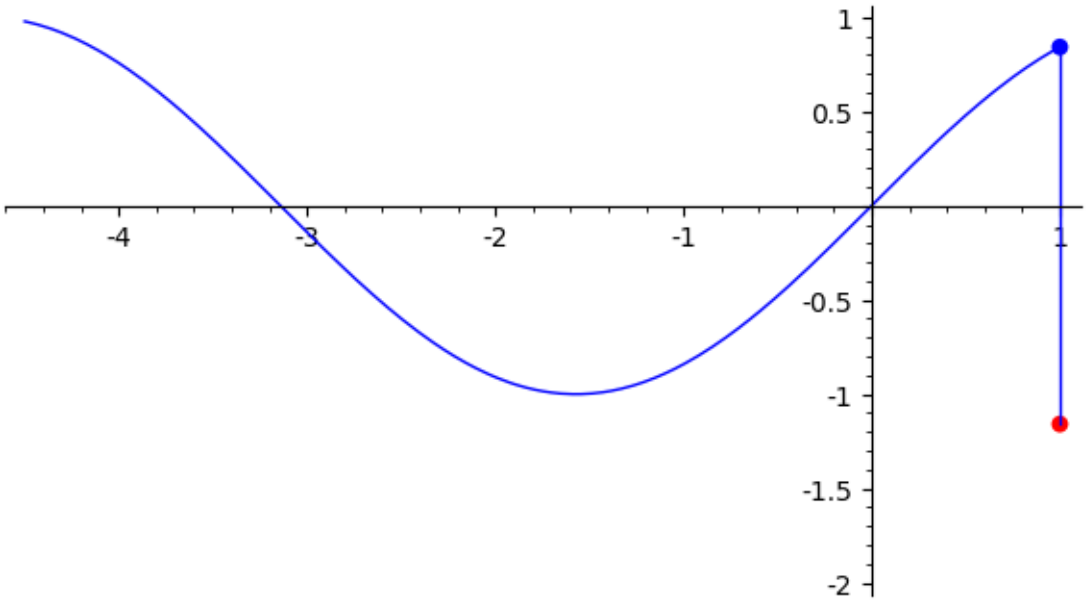
```
In [18]: from IPython.display import clear_output
         import time
         for ith in range(0,len(sol),20):
             plt = draw_system(ith=ith,l=1)
             clear_output(wait=True)
             plt.show(figsize=3)
             time.sleep(0.021)
```



Alternatively one can use slider:

```
In [19]: @interact
         def _(ith=slider(range(len(sol)))):
```

```
plt = draw_system(ith=ith,l=2)
plt.show(figsize=6)
```



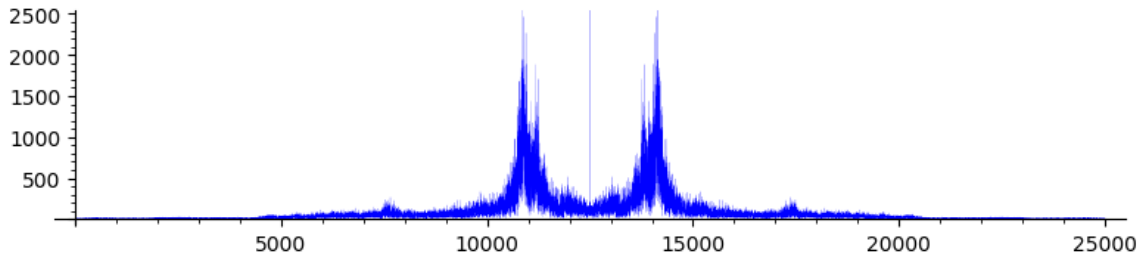
8.2.2 Chaotic properties of the solution

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.

```
In [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 9.03 s, sys: 1.4 s, total: 10.4 s
Wall time: 8.97 s
```

```
In [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

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

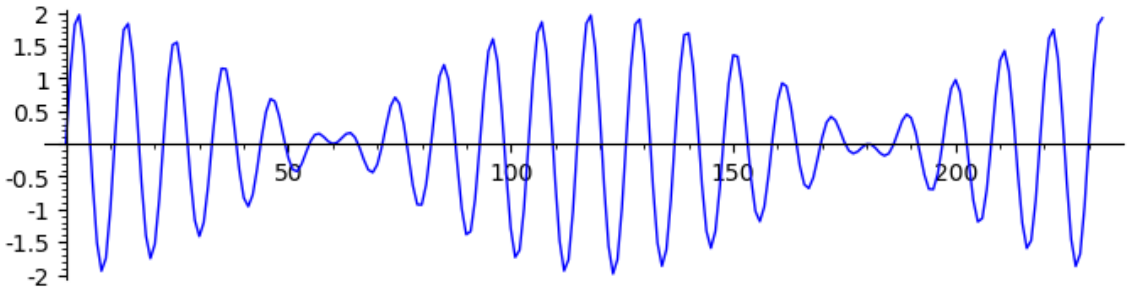
```
In [23]: import numpy as np
         import sympy
         expr_np = np.vectorize( sympy.lambdify(t, sympy.sympify( expr ) ) )
```

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

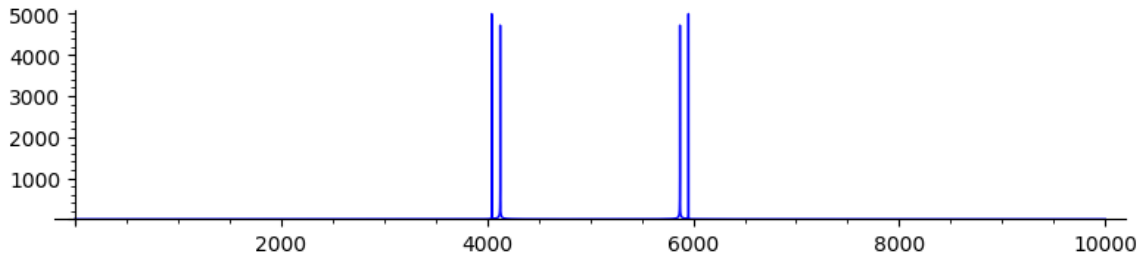
```
In [25]: %time nonchaotic_fft = np.fft.fft(nonchaotic)
```

```
CPU times: user 1.48 ms, sys: 0 ns, total: 1.48 ms
Wall time: 904 ţs
```

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




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



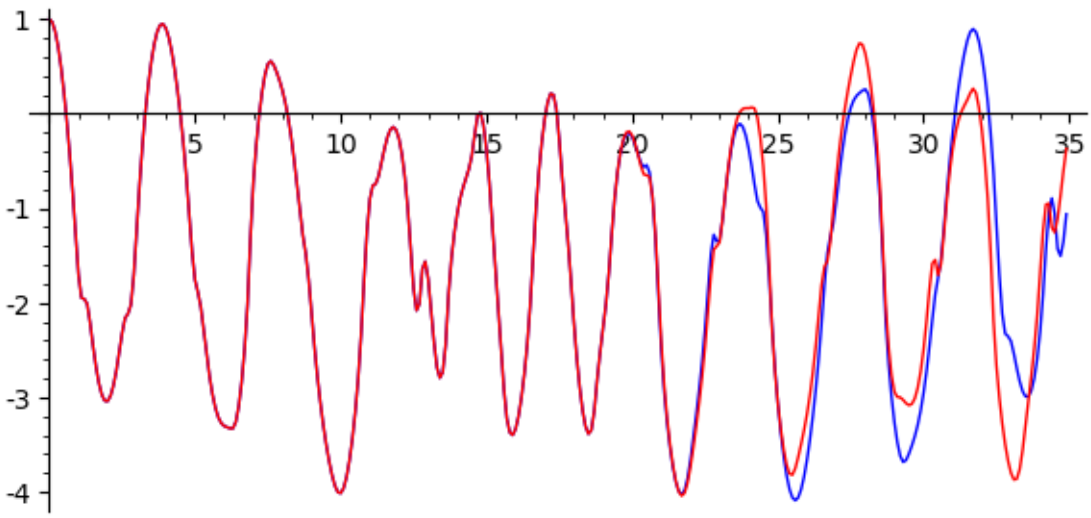
Sensitivity to initial conditions Let us compare two solution which differ by $\frac{1}{1000000}$ in initial velocity.

```
In [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 138 ms, sys: 37.4 ms, total: 175 ms
Wall time: 150 ms
```

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

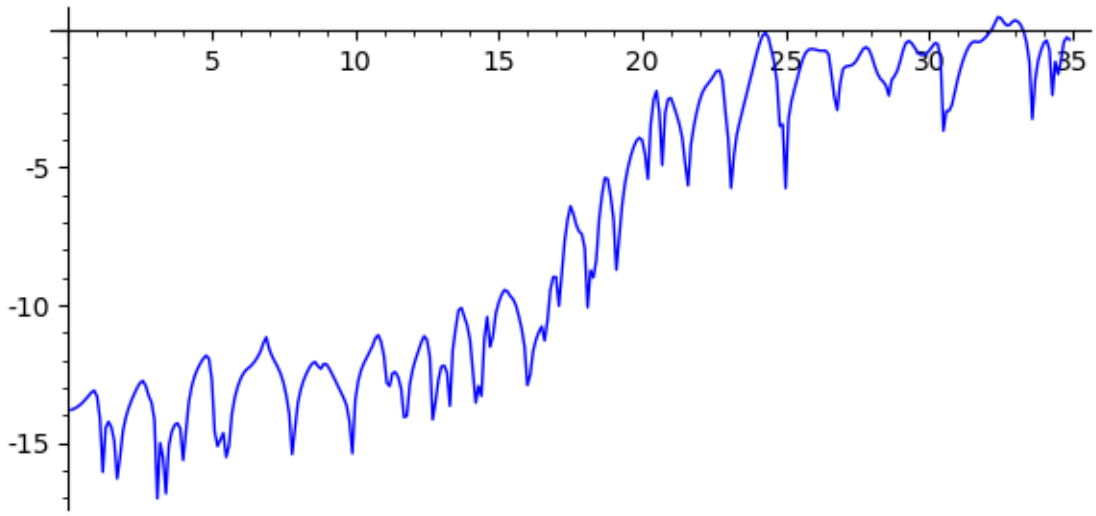
Out[29]:



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

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

Out[30]:



9 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.

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

```
In [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
```

9.1 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:

```
In [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)
```

Out[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:

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

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.

```
In [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)
```

Out[4]:

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

```
In [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:

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

Out[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:

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

Out[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.

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

Out[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:

```
In [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 )
```

Out[9]:

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

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

Out[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:

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

Out[11]:

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

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:

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

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

Out[13]:

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

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

Out[14]:

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

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

Out[15]:

$$\ddot{y}_2 = -\frac{\left(gm_1 + gm_2\right){x_1}^2 - 2\left(gm_1 + gm_2\right)x_1x_2 + \left(gm_1 + gm_2\right){x_2}^2 + \left(\dot{x}_1^2 - 2\,\dot{x}_1\dot{x}_2 + \dot{x}_2^2 + \dot{y}_2^2\right)m_1y_2}{\left(m_1 + m_2\right){x_1}^2 - 2\left(m_1 + m_2\right)x_1x_2 + \left(m_1 + m_2\right){x_2}^2 + m_1{y_2}^2}$$

9.1.1 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

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

Out[16]:

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

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:

In [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)]
```

```
Out[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

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

```
Out[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:

```
In [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:

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

Out[20]:

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

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:

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

Out[21]:

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

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

Out[22]:

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

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:

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

Out[23]:

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

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

Out[24]:

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

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

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

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

Out[26]:

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

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

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

Out[28]:

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

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

Out[29]:

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

9.1.2 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:

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

Out[30]:

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

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

Out[31]:

$$\frac{\frac{lm_2\dot{\phi}^2\cos(\phi)\sin(\phi)}{m_1}+g\sin(\phi)+\frac{gm_2\sin(\phi)}{m_1}}{\frac{lm_2\cos(\phi)^2}{m_1}-l-\frac{lm_2}{m_1}}$$

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:

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

```
Out[32]: 0
```

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

```
Out[33]:
```

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

9.1.3 Case study $m_2 >> m_1$

In this case, the first mass is negligible.

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

```
Out[34]:
```

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

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

```
Out[35]:
```

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

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

```
Out[36]:
```

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

9.1.4 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

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

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:

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

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

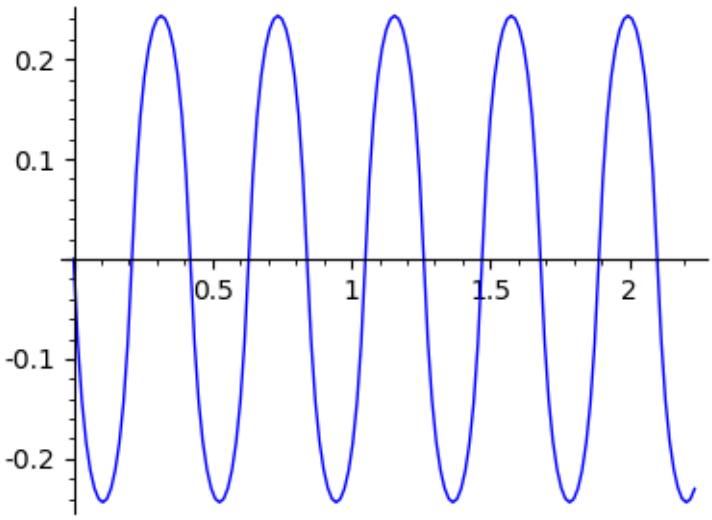
Out[37]:

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

```
In [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])
```

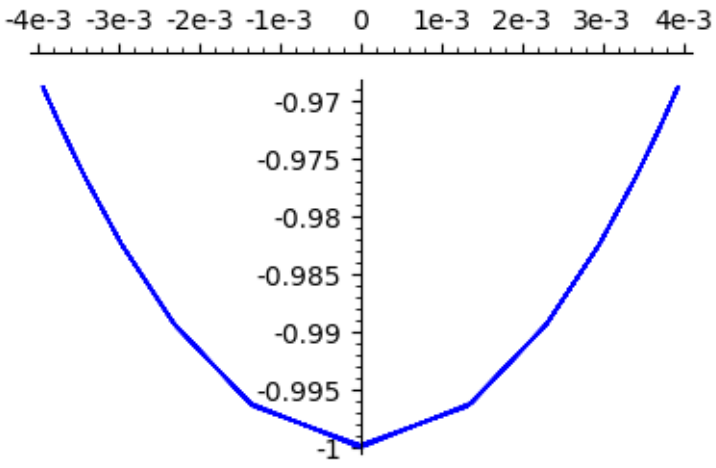
```
In [39]: line( zip(times,sol[:,0]),figsize=4 )
```

Out[39]:



In [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 )
```

Out[40]:



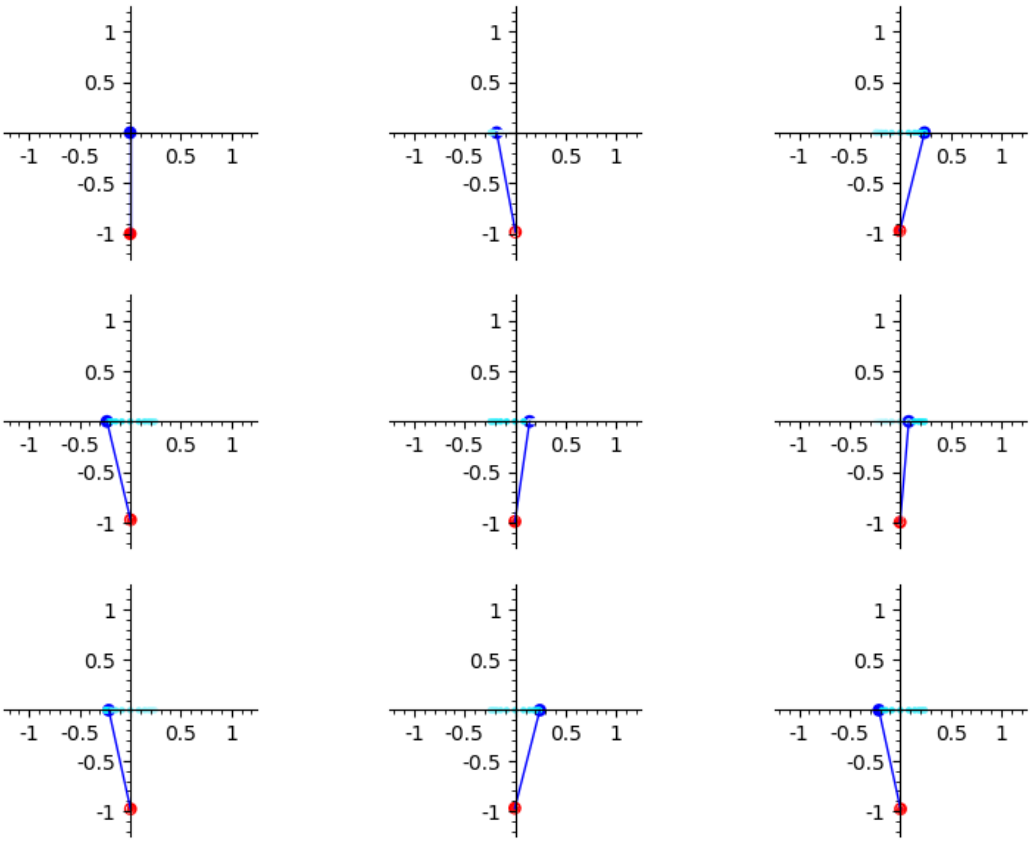
In [41]:
```
def draw_system(ith=0,l=1):
    x,phi = sol[ith,:2]
    x1,y1,x2,y2 =x, 0,  l*sin(phi) + x,-l*cos(phi)
```

```
            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)
            trace2 = sum([point((x,0),hue=(.51,(i)/n,1)) for i,(x,phi) in enumerate(sol[i0
            #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
```

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

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

In [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)
```

9.1.5 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.

10 Euler Lagrange - pendulum with oscillating support

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

$$L = E_k - E_p \tag{16}$$

Then equation of motion are given by:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\varphi}}\right) - \frac{\partial L}{\partial \varphi} = 0 \tag{17}$$

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.

10.1 System definition

Let us define a system,

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

```
In [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
```

10.1.1 Horizontal oscillations of a support point

We parametrize the system similarily to mathematical pendulum,

$$x = a\sin(\omega t) + l\sin(\varphi) \tag{18}$$
$$y = -l\cos(\varphi) \tag{19}$$

```
In [3]: # horizontal
        var('a omega t')
        x2u = {x:l*sin(phi)+a*sin(omega*t),y:-l*cos(phi)}
        showmath(x2u)
```

Out[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:

$$E_k = \frac{1}{2}(\dot{x}^2 + \dot{y}^2) \tag{20}$$

Using transformation dictionary, to generalized coordinates we have $E_k(\varphi, , \dot{\varphi})$:

```
In [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)
```

Out[4]:

$$\frac{1}{2}a^2\omega^2\cos(\omega t)^2 + al\omega\dot{\varphi}\cos(\omega t)\cos(\varphi) + \frac{1}{2}l^2\dot{\varphi}^2$$

Step 2: Potential energy

Similarily we have to express potential energy:

$$E_p = gy \tag{21}$$

as a function of $E_P(\varphi)$

```
In [5]: Ep = g*y.subs(x2u)
        showmath(Ep)
```

Out[5]:

$$-gl\cos(\varphi)$$

Step 3: Langranian

Now we have Lagrangian $L(\varphi, \dot{\varphi})$:

```
In [6]: L = Ek - Ep
        showmath(L)
```

Out[6]:

$$\frac{1}{2}a^2\omega^2\cos(\omega t)^2 + al\omega\dot{\varphi}\cos(\omega t)\cos(\varphi) + \frac{1}{2}l^2\dot{\varphi}^2 + gl\cos(\varphi)$$

10.2 Derivation of equations of motion

Using Euler-Lagrange formulas 17 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.

```
In [7]: EL1 = L.diff(phid).subs(to_fun).diff(t).subs(to_var) - L.diff(phi)
```

```
In [8]: showmath(EL1)
```

```
Out[8]:
```

$$-al\omega^2\cos\left(\varphi\right)\sin\left(\omega t\right)+l^2\ddot{\varphi}+gl\sin\left(\varphi\right)$$

10.3 Analysis

10.3.1 Small angle approximation

Let see what happens if oscillations are small. We can expand in Taylor seried the equations of motion:

```
In [9]: eq_lin = EL1.taylor(phi,0,1)
        showmath(eq_lin)
```

```
Out[9]:
```

$$-al\omega^2\sin\left(\omega t\right)+gl\varphi+l^2\ddot{\varphi}$$

```
In [10]: var('alpha,omega0')
         eq_lin2  = (eq_lin/l^2).expand().subs({a:l*alpha,g:l*omega0^2})
         showmath(eq_lin2)
```

```
Out[10]:
```

$$-\alpha\omega^2\sin\left(\omega t\right)+\omega_0^2\varphi+\ddot{\varphi}$$

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.

```
In [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)
```

Out[11]:

$$-\frac{\alpha\omega^2 \sin(\omega t)}{\omega^2 - \omega_0^2} + K_2 \cos(\omega_0 t) + K_1 \sin(\omega_0 t)$$

```
In [12]: showmath(eq_lin2)
```

Out[12]:

$$-\alpha\omega^2 \sin(\omega t) + \omega_0^2 \varphi + \ddot{\varphi}$$

10.3.2 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:

$$\frac{d\varphi}{dt} = \dot{\varphi} \tag{22}$$

$$\frac{d\dot{\varphi}}{dt} = \frac{a\omega^2 \cos(\varphi) \sin(\omega t)}{l} - \frac{g \sin(\varphi)}{l} \tag{23}$$

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.

```
In [13]: rhs = EL1.solve(phidd)[0].rhs()
         showmath(rhs().expand())
```

Out[13]:

$$\frac{a\omega^2 \cos(\varphi) \sin(\omega t)}{l} - \frac{g \sin(\varphi)}{l}$$

Linear system can be derived in similar way:

```
In [14]: rhs_lin = eq_lin.solve(phidd)[0].rhs()
         showmath(rhs_lin)
```

Out[14]:

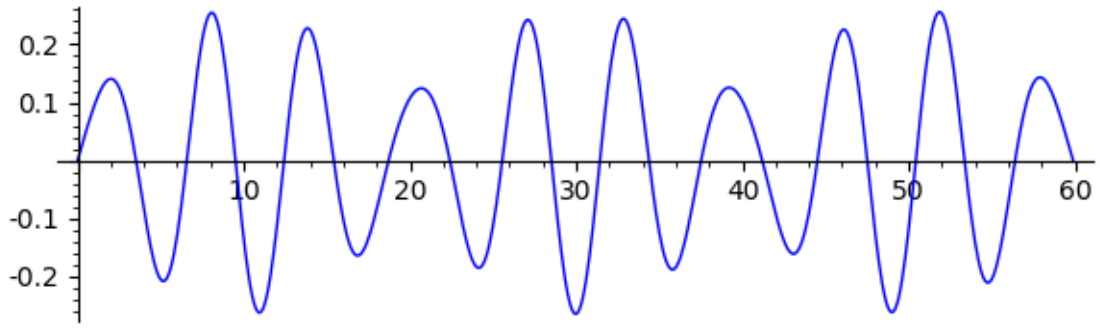
$$\frac{a\omega^2 \sin(\omega t) - g\varphi}{l}$$

In [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:

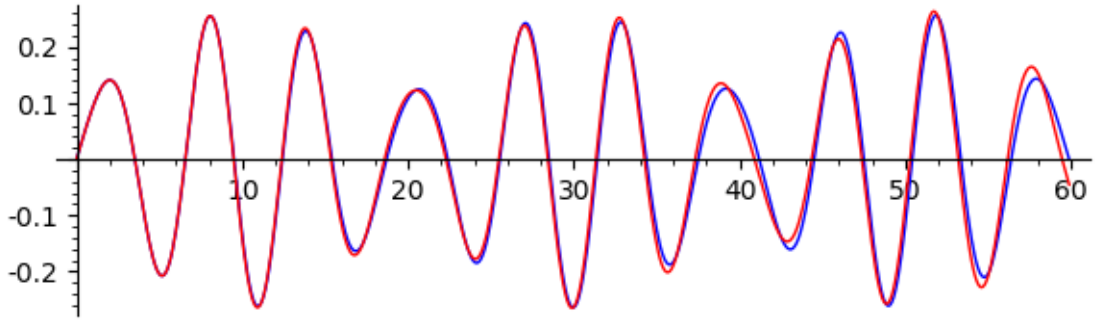
In [16]:
```
ode = [phid, rhs.subs(pars)]
times = srange(0,t_end,0.1)
ics = [0.0, 0.1]
sol = desolve_odeint(ode, ics, times, [phi, phid])
line( zip(times,sol[::1,0]),figsize=(6,2), )
```

Out[16]:



In [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')
```

Out[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.

10.3.3 Vertical oscillations

In the case of vertical oscillations of a support point, the transformation to generalized coordinates reads:

$$x = l \sin(\varphi) \tag{24}$$
$$y = -a \cos(\omega t) - l \cos(\varphi), \tag{25}$$

```
In [18]: # vertical
         var('a omega t')
         x2u = {x:l*sin(phi), y:-l*cos(phi)-a*cos(omega*t)}
         showmath(x2u)
```

Out[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:

```
In [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)
```

Out[19]:

$$al\omega^2 \cos(\omega t) \sin(\varphi) + l^2\ddot{\varphi} + gl \sin(\varphi)$$

In [20]:
```
rhs = EL1.solve(phidd)[0].rhs()
showmath( rhs.expand().collect(sin(phi)) )
```

Out[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:

In [21]:
```
showmath(EL1.taylor(phi,0,1) )
```

Out[21]:

$$l^2\ddot{\varphi} + \left(al\omega^2 \cos(\omega t) + gl\right)\varphi$$

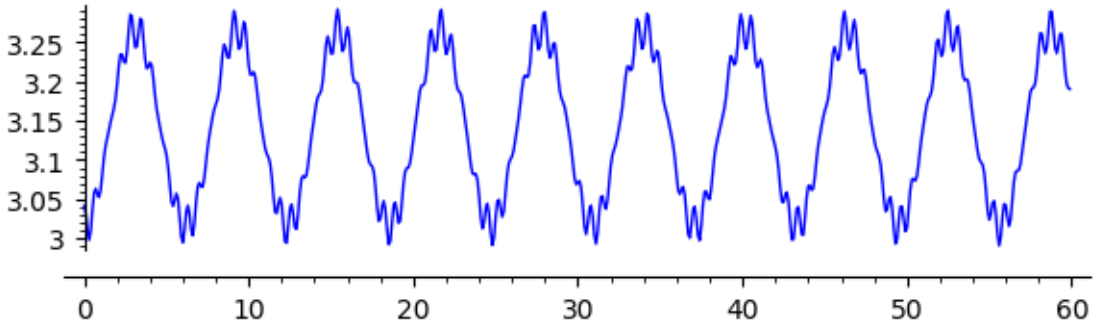
We see that in this case we do not obtain a harmonic oscillator. Forcing term is multiplied by φ, i.e. the linearized equation is fundamentally different.

10.3.4 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:

In [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.

```
In [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 ]
```

```
In [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
```

```
In [25]: #draw_pendulum(1200).save('inverted_pend.png',figsize=8)
```

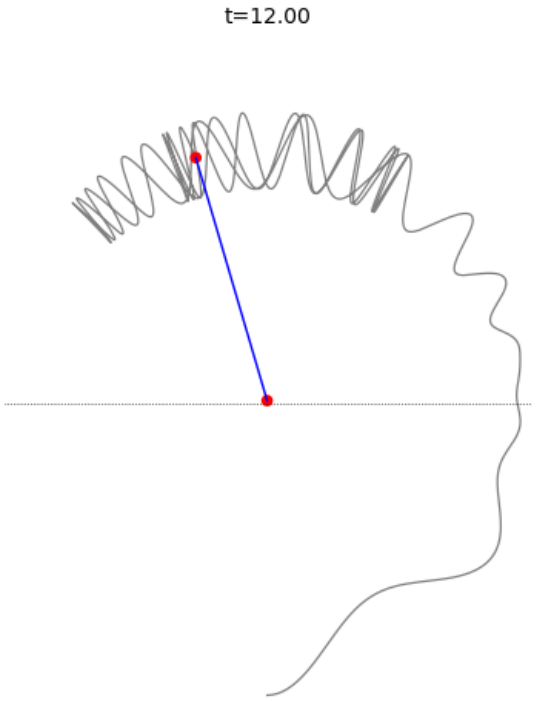
Time evolution of the inverted stable pendulum:

10.3.5 System with damping

Adding damping to the system will make inverted state an stable atractor.

$$\ddot{\varphi} = -2\gamma\dot{\varphi} + (-\omega_0^2 - \frac{a}{l}\omega^2\cos(\omega t))\sin(\varphi)$$

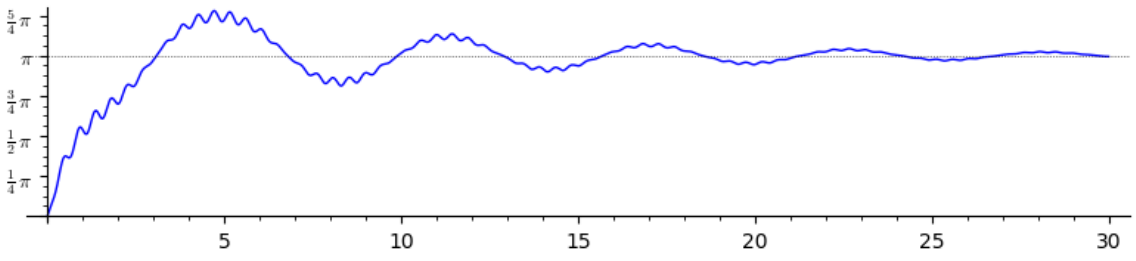
```
In [26]: var('omega,omega0,gama,t,a')
         pars = {l:1,a:0.152,omega0:1,omega:14.,gama:.1}
         ode = [phid,\
                (-2*gama*phid+(-omega0^2-a/l*omega^2*cos(omega*t))*sin(phi)).subs(pars)]
```



Inverted pendulum stalibized by oscillations of a support point

```
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)]])
```

Out[26]:



11 Point particle on rotating curve

11.1 Point particle on arbitrary curve

Let us assume that point with mass $m = 1$ moves freely on a flat curve which spins around vertical axis with angular velocity ω_0. In general case we have a curve given by an implicit equation in a form:

$$f(x, y) = 0$$

We can treat the problem in 2d in the rotating frame of reference. However, we need to take into account additional force in non-inertial system which acts in x direction:

$$F_x = mx\omega_0^2.$$

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

```
In [2]: var('t')
        var('g')
        var('w0', latex_name='\omega_0')
        xy_names = [('x','x'),('y','y')]
```

```
In [3]: to_fun, to_var = make_symbols(xy_names)
```

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

dAlembert principle takes the following form:

```
In [4]: dAlemb = (X.diff(t,2)-x*w0^2)*dx + (Y.diff(t,2)+g)*dy
```

```
In [5]: showmath(dAlemb.subs(to_var))
```

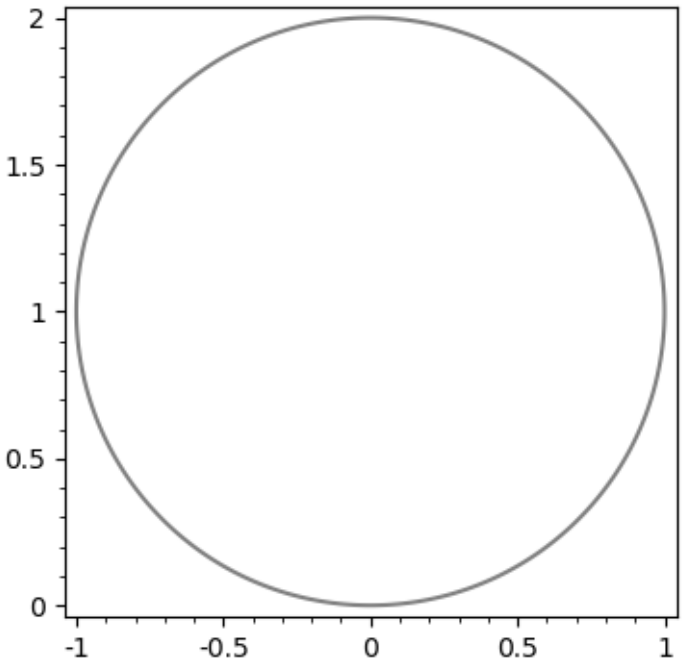
```
Out[5]:
```

$$-\left(\omega_0{}^2 x - \ddot{x}\right)\delta x + \delta y(g + \ddot{y})$$

We will need differential form of contraints, which can also be obtained by computer algebra. We will use implicit equation of a circle of unit radius. In this case we can compare solutions obtained in generic procedure with one computer in generalized coordinates.

```
In [6]: f = x^2+ (y-1)^2 - 1

        plt_constraints = implicit_plot(f,(x,-1,1),(y,0,2),figsize=5,color='gray')
        plt_constraints
```

Out[6]:



We will use constaints to connect variations δx and δy with each other, therefore we use formula for complete differential:

In [7]:
```
df = diff(f,x)*dx+diff(f,y)*dy
showmath(df)
```

Out[7]:

$$2\,\delta x x + 2\,\delta y (y-1)$$

We solve above equation for δx and substitute it to dAlembert equation. Since there is only one variation left, then its coeficcient must be zero. It is a first differential equation.

In [8]:
```
eq1 = dAlemb.subs(df.solve(dx)).coefficient(dy).subs(to_var)
showmath(eq1)
```

Out[8]:

$$g + \frac{\left(\omega_0{}^2 x - \ddot{x}\right)(y-1)}{x} + \ddot{y}$$

The second equation can be obtained, for example by differentiation of contraints in time two times.

```
In [9]: eq2 = f.subs(to_fun).diff(t,2).subs(to_var)
        showmath(eq2)
```

Out[9]:

$$2\,\dot{x}^2 + 2\,x\ddot{x} + 2\,\dot{y}^2 + 2\,(y-1)\ddot{y}$$

We solve above system for $\ddot{x}$ and $\ddot{y}$:

```
In [10]: sol = solve( [eq1,eq2], [xdd,ydd])[0]
         showmath( sol[0] )
```

Out[10]:

$$\ddot{x} = \frac{{\omega_0}^2 x y^2 - \left(2\,{\omega_0}^2 - g\right) x y + \left({\omega_0}^2 - \dot{x}^2 - \dot{y}^2 - g\right) x}{x^2 + y^2 - 2\,y + 1}$$

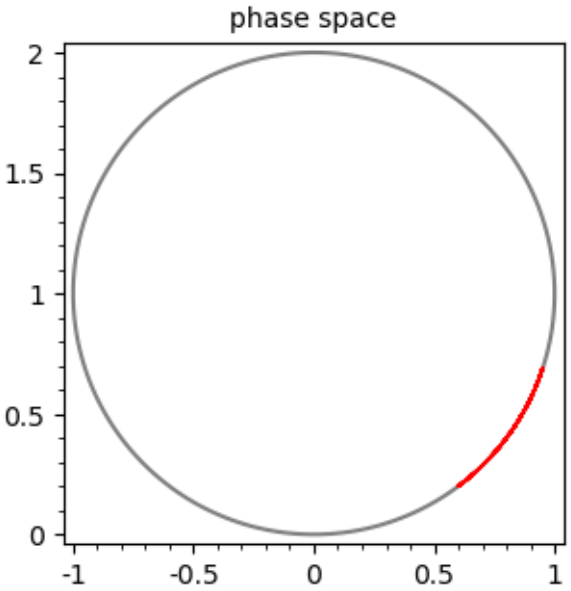
```
In [11]: showmath( sol[1] )
```

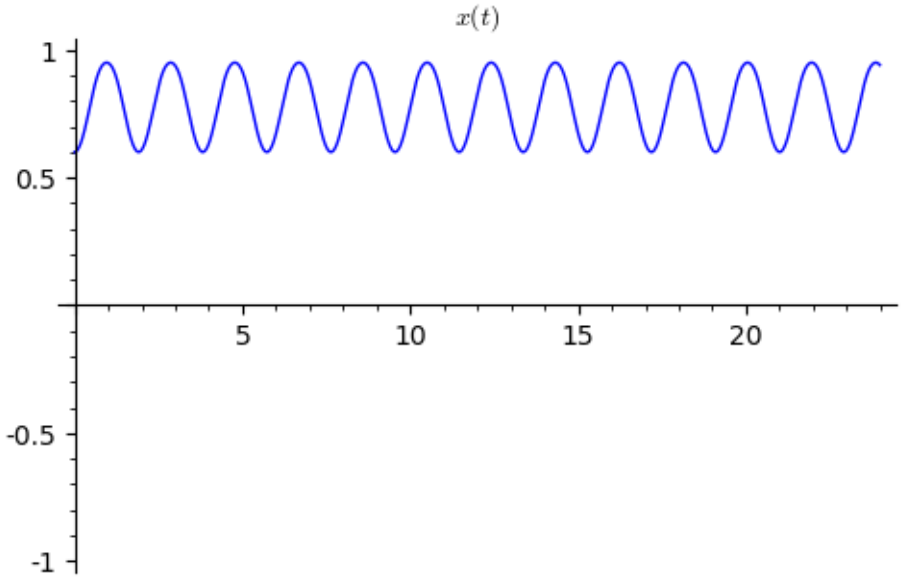
Out[11]:

$$\ddot{y} = \frac{\left({\omega_0}^2 - g\right) x^2 + \dot{x}^2 - \left({\omega_0}^2 x^2 + \dot{x}^2 + \dot{y}^2\right) y + \dot{y}^2}{x^2 + y^2 - 2\,y + 1}$$

We are ready to set numerical values of parameters and solve system of ODEs:

```
In [12]: times = srange(0,24,0.01)

         ode = [xd,yd] + [ sol_.rhs().subs({w0:4.2,g:9.81}) for sol_ in sol]

         x0 = 0.6
         y0 = f.subs({x:x0}).solve(y)[1].rhs().n()
         dx0 = 0.0
         dy0 = df.subs({x:x0,y:y0,dx:dx0}).solve(dy)[0].rhs().n()

         numsol = desolve_odeint(ode,[x0,y0,dx0,dy0], times, [x,y,xd,yd])
         p = line(zip(numsol[:,0],numsol[:,1]), color='red')
         p2 = line(zip(times,numsol[:,0]), figsize=5, ymin=-1, ymax=1)

         (p+plt_constraints).show(title='phase space')
         p2.show(title=r'$x(t)$')
```





Experiment with Sage!

Change parameters to see if other types of solution can be found!

11.1.1 Explicit form of contraints

If contraits are in explicit for we can simplify the procedure.

In [13]:
```
# f = 1/2*(x)^2 - y
f = 1-sqrt(1-x^2) - y
```

In [14]:
```
df = diff(f,x)*dx+diff(f,y)*dy
```

In [15]:
```
eq1 = dAlemb.subs(df.solve(dx)).coefficient(dy).subs(to_var)
showmath(eq1)
```

Out[15]:

$$g+\ddot{y}-\frac{\left(\omega_0{}^2x-\ddot{x}\right)\sqrt{-x^2+1}}{x}$$

In [16]:
```
eq2 = f.subs(to_fun).diff(t,2).subs(to_var)
showmath(eq2)
```

Out[16]:

$$\frac{x^2\dot{x}^2}{\left(-x^2+1\right)^{\frac{3}{2}}}+\frac{\dot{x}^2}{\sqrt{-x^2+1}}+\frac{x\ddot{x}}{\sqrt{-x^2+1}}-\ddot{y}$$

In [17]:
```
sol = solve( [eq1,eq2], [xdd,ydd])[0]
showmath( sol[0] )
```

Out[17]:

$$\ddot{x}=-\frac{\omega_0{}^2x^5-2\,\omega_0{}^2x^3+\left(\omega_0{}^2-\dot{x}^2\right)x+\left(gx^3-gx\right)\sqrt{-x^2+1}}{x^2-1}$$

We see that the first equation does not depent on y. We can solve it numericaly.

In [18]:
```
y_on_curve = y.subs(f.solve(y)[0])
```

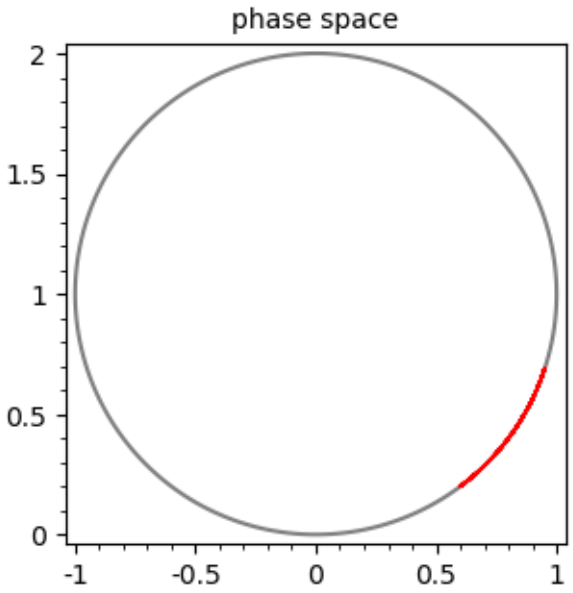
In [19]:
```
y_fun = fast_callable(y_on_curve, vars = [x])
```

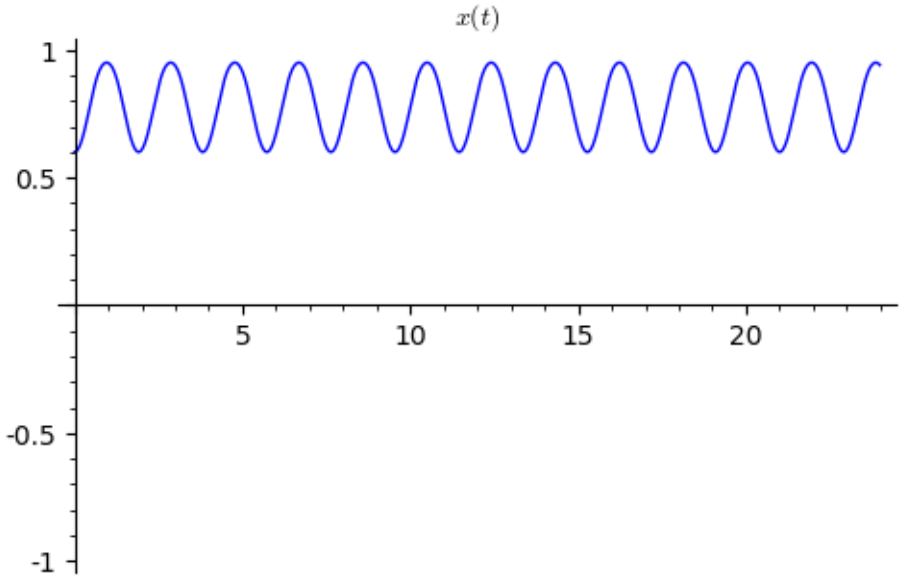
In [20]:
```
y_fun(numsol[:,0])
```

Out[20]:
```
array([0.2       , 0.20007744, 0.20030982, ..., 0.66806648, 0.66409114,
       0.65977775])
```

In [21]:
```
ode = [xd,sol[0].rhs().subs({w0:4.2,g:9.81})]

times = srange(0,24,0.01)
```

```
numsol = desolve_odeint(ode,[.6,.0],times,[x,xd])
p = line(zip(numsol[:,0],y_fun(numsol[:,0])),color='red')
p2 = line(zip(times,numsol[:,0]),figsize=5,ymin=-1,ymax=1)

(p+plt_constraints).show(title='phase space')
p2.show(title=r'$x(t)$')
```





We see that it was possible to obtain the same solution. However, the implicit form allows to solve the system for $y > 1$. The explicit formulation will fail in wuch a case.

11.2 Point particle on rotating circle

If particle moves on the rotating circle, we can use the symmetry and transform dAlembert principle to polar coordinates in rotating frame.

```
In [22]: 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
```

```
In [23]: x2u = {x:l*sin(phi),y:1-l*cos(phi)}
```

```
In [24]: showmath(x2u)
```

```
Out[24]:
```

$$\{y : -l\cos(\phi) + 1, x : l\sin(\phi)\}$$

```
In [25]: 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)]
```

```
Out[25]: [dphi*l*cos(phi), dphi*l*sin(phi)]
```

```
In [26]: 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
```

```
In [27]: sol = dAlemb.expand().coefficient(dphi).trig_simplify().solve(phidd)
         show( sol)
```

```
[phidd == (l*w0^2*cos(phi)*sin(phi) - g*sin(phi))/l]
```

11.2.1 Efffective potential

In polar coordinates we can analyze the system without solving. Namely we can observe that the generalized force depends only on ϕ, thus:

$$\ddot{\phi} = \underbrace{\frac{l\omega_0{}^2 \cos(\phi)\sin(\phi) - g\sin(\phi)}{l}}_{-\frac{\partial U_{eff}(\phi)}{\partial \phi}} \tag{26}$$

we can interpret the equation of motion as a motion in an effective potential.

Let's extrat this potential and see how it behaves for different parameters:

```
In [28]: Ueff = -sol[0].rhs().expand().integrate(phi)
         showmath(Ueff)
```

Out[28]:

$$\frac{1}{2}\,\omega_0{}^2 \cos(\phi)^2 - \frac{g\cos(\phi)}{l}$$

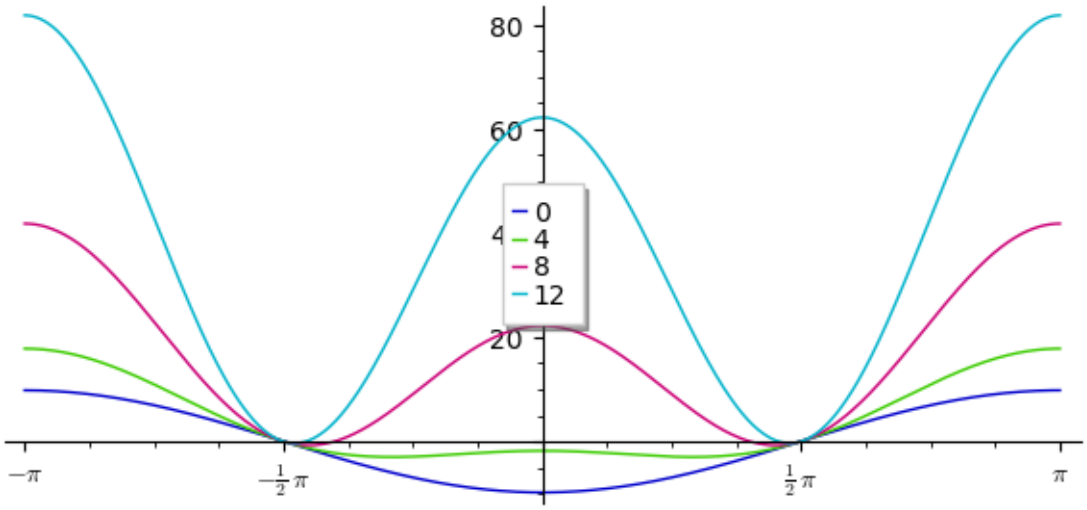
```
In [29]: Ueff = -sol[0].rhs().expand().subs({g:9.81,l:1}).integrate(phi)
         showmath(Ueff)
```

Out[29]:

$$\frac{1}{2}\,\omega_0{}^2 \cos(\phi)^2 - 9.81\,\cos(\phi)$$

```
In [30]: 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])
```

Out[30]:



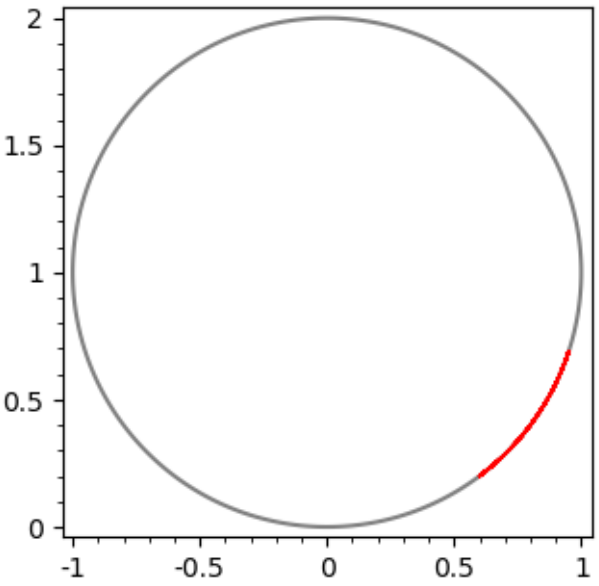
11.2.2 Numerical solutions

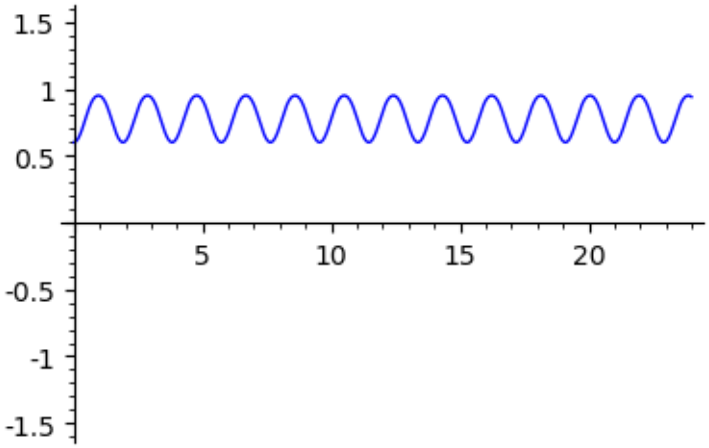
```
In [31]: arcsin(0.4)
```

```
Out[31]: 0.411516846067488
```

```
In [32]: times = srange(0,24,0.01)
         pars = {l:1, w0:4.2, g:9.81}

         x_phi = fast_callable( x.subs(x2u).subs(pars),vars=[phi])
         y_phi = fast_callable( y.subs(x2u).subs(pars),vars=[phi])

         ode = [phid,sol[0].rhs().subs(pars)]
         show(ode)
         numsol = desolve_odeint(ode,[arcsin(0.6),0],times,[phi,phid])
         p = line(zip(x_phi(numsol[:,0]),y_phi(numsol[:,0])),color='red')
         p2 = line(zip(times,x_phi(numsol[:,0])),figsize=4,ymin=-pi/2,ymax=pi/2)

         (p+plt_constraints).show()
         p2.show()
```

```
[phid, 17.6400000000000*cos(phi)*sin(phi) - 9.81000000000000*sin(phi)]
```





11.3 Lagrange approach

11.3.1 Rotating system of coordinates

In a rotating system of coordinates, the bead moves on the circle in (x, y) plane. At the same time there is a rotation with angular velocity ω_0 around vetical axis. It means that a bead will have addtional velocity, perpedicular to (x, y) plane, which magnitute is:

$$v_{rot} = x\omega_0 = l\omega_0 \sin(\phi)$$

In [33]:
```
Ek = 1/2*sum([x_.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2 for x_ in [x,y]])
Ek += 1/2*(x.subs(x2u)*w0)^2
Ek = Ek.trig_simplify()
showmath(Ek)
```

Out[33]:

$$\frac{1}{2}\,l^2{\omega_0}^2\sin\left(\phi\right)^2+\frac{1}{2}\,l^2\dot{\phi}^2$$

In [34]:
```
Ep = g*y.subs(x2u)
showmath(Ep)
```

Out[34]:

$$-(l\cos\left(\phi\right)-1)g$$

In [35]:
```
L = Ek-Ep
showmath(L)
```

Out[35]:

$$\frac{1}{2}\,l^2{\omega_0}^2\sin\left(\phi\right)^2+\frac{1}{2}\,l^2\dot{\phi}^2+(l\cos\left(\phi\right)-1)g$$

In [36]:
```
EL = L.diff(phid).subs(to_fun).diff(t).subs(to_var) - L.diff(phi)
```

In [37]:
```
showmath(EL)
```

Out[37]:

$$-l^2{\omega_0}^2\cos\left(\phi\right)\sin\left(\phi\right)+l^2\ddot{\phi}+gl\sin\left(\phi\right)$$

11.3.2 Code generation

We can readily generate code which can be used in external programs.

En example can be found in simulation and 3d vis

In [38]:
```
from sympy import ccode
oderhs = sol[0].rhs()
ccode(oderhs._sympy_())
```

Out[38]:
```
'(-g*sin(phi) + l*pow(w0, 2)*sin(phi)*cos(phi))/l'
```

12 Bead on a rotating circle

We consider a movement of a material point (bead) with the mass m in a gravitational field, along a vertical circle with a radius l rotating with the frequency ω around a vertical axis, passing through the center of the circle.

Turning circle creates a virtual sphere with a R radius and a bead moves in this sphere. Therefore, it is most convenient to choose a reference system in (r, θ, ϕ) spherical variables:

$$x = r\sin\theta\cos\phi \tag{27}$$
$$y = r\sin\theta\sin\phi \tag{28}$$
$$z = r\cos\theta \tag{29}$$

It is a system with constraints and therefore it is convenient to analyze this issue in the framework of Lagrange mechanics.

The Lagrange function is the difference in the kinetic energy of the bead and its potential energy (particles in the gravitational field):

$$L = L(r, \theta, \phi, \dot{r}, \dot{\theta}, \dot{\phi}) = E_k - E_p$$

The kinetic energy of the bead is given by the formula:

$$E_k = \frac{mv^2}{2} = \frac{m}{2}[\dot{x}^2 + \dot{y}^2 + \dot{z}^2] = \frac{m}{2}[\dot{r}^2 + r^2\dot{\theta}^2 + r^2\sin^2(\theta)\dot{\phi}^2] \tag{30}$$

The potential energy of the bead has the form

$$E_p = mgz = mgr\cos(\theta)$$

12.1 Lagrange aproach in spherical coordinates

We can write Lagrangian of the system in spherical coordinates. Contraints in spherical coordinates have simple form:

- $r = l$ - radius is constant.
- $\phi(t) = \omega_0 t$ - the azimuthal angle is forced by the constraint.

Effectively, the only degree of freedom is a polar angle θ.

We will used CAS to derive both the form of the Lagrangian as well as equation of motion.

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

```
In [2]: var('t')
        var('l g w0')
        xy_names = [('x','x'),('y','y'),('z','z')]
        uv_names = [('r','r'),('phi',r'\phi'),('theta',r'\theta')]

        load('cas_utils.sage')

        to_fun, to_var = make_symbols(xy_names,uv_names)
        x2u = {x:r*sin(theta)*cos(phi),
               y: r*sin(theta)*sin(phi),
               z: r*cos(theta)}
        _  = transform_virtual_displacements(xy_names,uv_names,suffix='_uv')
```

```
r  :: has been  processed
phi  :: has been  processed
theta  :: has been  processed
x  :: has been  processed
y  :: has been  processed
z  :: has been  processed
```

```
In [3]: Ek = 1/2*sum([x_.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2 \
                      for x_ in [x,y,z]])
```

The symbolic variable is kinetic energy in spherical coordinates 30,

```
In [4]: showmath(Ek.trig_simplify())
```

```
Out[4]:
```

$$\frac{1}{2}\,\dot{\phi}^2 r^2 \sin\left(\theta\right)^2 + \frac{1}{2}\,r^2\dot{\theta}^2 + \frac{1}{2}\,\dot{r}^2$$

At this point we can substitute conditions resulting from constraints. Note that since we have already time derivatives $\dot{r}$ and $\dot{phi}$, we have to also substitute them explicitely.

```
In [5]: Ek = Ek.subs({phi:w0*t,phid:w0,rd:0,r:l}).trig_simplify()
        showmath(Ek)
```

```
Out[5]:
```

$$\frac{1}{2}\,l^2 w_0^2 \sin\left(\theta\right)^2 + \frac{1}{2}\,l^2\dot{\theta}^2$$

The potential energy can be computer in a similar way:

```
In [6]: Ep = g*z.subs(x2u).subs({r:l})
        showmath(Ep)
```

Out[6]:

$$gl\cos(\theta)$$

Langrange function is a difference between E_k and E_{pot}:

In [7]:
```
L = Ek-Ep
showmath(L.trig_simplify())
```

Out[7]:

$$\frac{1}{2}l^2w_0^2\sin(\theta)^2+\frac{1}{2}l^2\dot{\theta}^2-gl\cos(\theta)$$

We have single Euler-Lagrange equation in generalized coodinate: polar angle θ.

In [8]:
```
EL = L.diff(thetad).subs(to_fun).diff(t).subs(to_var) - L.diff(theta)
```

In [9]:
```
showmath(EL.trig_simplify())
```

Out[9]:

$$-l^2w_0^2\cos(\theta)\sin(\theta)+l^2\ddot{\theta}-gl\sin(\theta)$$

12.2 Analysis of the system

12.2.1 Effective potential

Fist, we can notice that the form of Lagrangian contains kinetic part, with term contatining $\dot{\theta}$ and two terms dependent only on θ. We can interpret the latter as an effective potential of 1d motion in θ coordinate. The term comes from kinetic energy.

We can extract it symbolically in SageMath by setting $\dot{\theta}=0$ in Lagrangian:

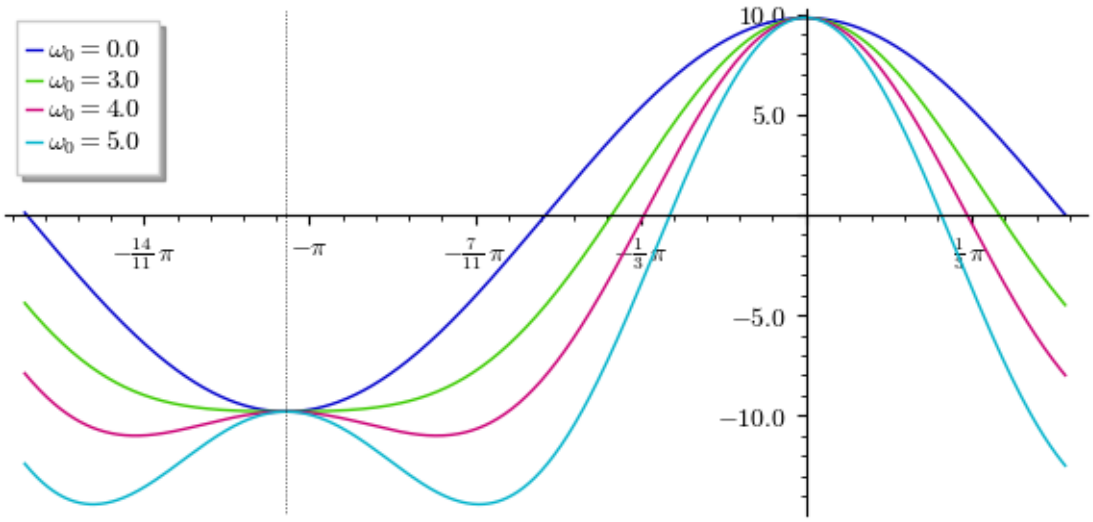
In [10]:
```
Ueff = -L.subs(thetad==0)
showmath(Ueff)
```

Out[10]:

$$-\frac{1}{2}l^2w_0^2\sin(\theta)^2+gl\cos(\theta)$$

In [11]:
```
plot?
```

```
In [12]: w0s = [0,3,4,5]
         p_u1 = plot([Ueff.subs({w0:w0_,l:1,g:9.81}) for w0_ in w0s],\
                     (theta,-3*pi/2-0.01,pi/2),
                    legend_label=[r'$\omega_0=%0.1f$'%w_ for w_ in w0s],
                   tick_formatter=pi, gridlines=[[-pi],None])
         p_u1.show(figsize=(6,3))
```



13 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.

```
In [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')]
```

```
In [2]: load('cas_utils.sage')
```

```
In [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
```

```
In [4]: x2u = {x1:l1*sin(phi1),\
               y1:-l1*cos(phi1),\
               x2:l1*sin(phi1)+l2*sin(phi2),\
               y2:-l1*cos(phi1)-l2*cos(phi2)}
```

```
In [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)]
```

```
Out[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)]
```

```
In [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)
```

```
In [7]: showmath(dAlemb)
```

```
Out[7]:
```

$$-\left(l_1\dot{\varphi}_1^2\sin\left(\varphi_1\right)-l_1\ddot{\varphi}_1\cos\left(\varphi_1\right)\right)\delta\varphi_1 l_1 m_1\cos\left(\varphi_1\right)+\left(\left(l_1\dot{\varphi}_1^2\cos\left(\varphi_1\right)+l_1\ddot{\varphi}_1\sin\left(\varphi_1\right)\right)m_1+gm_1\right)\delta\varphi_1 l_1\sin\left(\varphi_1\right)-\left(l\right.$$

```
In [8]: eq1 = dAlemb.expand().coefficient(dphi1).trig_simplify()
        eq2 = dAlemb.expand().coefficient(dphi2).trig_simplify()
        showmath(eq1)
```

```
Out[8]:
```

$$\left(l_1 l_2 m_2\cos\left(\varphi_2\right)\sin\left(\varphi_1\right)-l_1 l_2 m_2\cos\left(\varphi_1\right)\sin\left(\varphi_2\right)\right)\dot{\varphi}_2^2+\left(l_1^2 m_1+l_1^2 m_2\right)\ddot{\varphi}_1+\left(l_1 l_2 m_2\cos\left(\varphi_1\right)\cos\left(\varphi_2\right)+l_1 l_2 m_2\sin\right.$$

```
In [9]: showmath(eq2)
```

```
Out[9]:
```

$$l_2^2 m_2\ddot{\varphi}_2-\left(l_1 l_2 m_2\dot{\varphi}_1^2\sin\left(\varphi_1\right)-l_1 l_2 m_2\ddot{\varphi}_1\cos\left(\varphi_1\right)\right)\cos\left(\varphi_2\right)+\left(l_1 l_2 m_2\dot{\varphi}_1^2\cos\left(\varphi_1\right)+l_1 l_2 m_2\ddot{\varphi}_1\sin\left(\varphi_1\right)+gl_2 m_2\right)\sin$$

```
In [10]: sol =  solve([eq1,eq2],[phi1dd,phi2dd])[0]
```

```
In [11]: showmath(sol)
```

Out[11]:

$$\left[\ddot{\varphi}_1 = \frac{l_1 m_2 \dot{\varphi}_1^2 \cos(\varphi_1)\cos(\varphi_2)^2 \sin(\varphi_1) - \left(l_1 m_2 \dot{\varphi}_1^2 \cos(\varphi_1) + g m_2\right) \sin(\varphi_1)\sin(\varphi_2)^2 + \left(l_2 m_2 \cos(\varphi_2) \sin(\varphi_1)\right.}{l_1 m_2 \cos(\varphi_1)^2 \cos(\varphi_2)^2 + 2\, l_1 m_2 \cos(\varphi_1)\, c}$$

```
In [12]: showmath( sol[0].rhs().denominator() )
```

Out[12]:

$$l_1 m_2 \cos(\varphi_1)^2 \cos(\varphi_2)^2 + 2\, l_1 m_2 \cos(\varphi_1)\cos(\varphi_2)\sin(\varphi_1)\sin(\varphi_2) + l_1 m_2 \sin(\varphi_1)^2 \sin(\varphi_2)^2 - l_1 m_1 - l_1 m_2$$

```
In [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)

In [14]: bool ( -2*sol[0].rhs().denominator()==(l1*(2*m1+m2-m2*cos(2*phi1-2*phi2))).expand_t

Out[14]: True
```

Since the "textbook" solution contains a slightly different form, let's check if we have these formulas:

$$T(\varphi_1, \varphi_2, \dot{\varphi}_1, \dot{\varphi}_2) = \frac{m_1}{2} l_1^2 \dot{\varphi}_1^2 + \frac{m_2}{2}\left(l_1^2 \dot{\varphi}_1^2 + l_2^2 \dot{\varphi}_2^2 + 2 l_1 l_2 \dot{\varphi}_1 \dot{\varphi}_2 \cos(\varphi_1 - \varphi_2)\right)$$

$$V(\varphi_1, \varphi_2) = -(m_1 + m_2) g l_1 \cos\varphi_1 - m_2 g l_2 \cos\varphi_2$$

$$m_2 l_2 \ddot{\varphi}_2 \cos(\varphi_1 - \varphi_2) + (m_1 + m_2)\, l_1 \ddot{\varphi}_1 + m_2 l_2 \dot{\varphi}_2^2 \sin(\varphi_1 - \varphi_2) + (m_1 + m_2)\, g \sin\varphi_1 = 0$$

$$l_2 \ddot{\varphi}_2 + l_1 \ddot{\varphi}_1 \cos(\varphi_1 - \varphi_2) - l_1 \dot{\varphi}_1^2 \sin(\varphi_1 - \varphi_2) + g \sin\varphi_2 = 0$$

```
In [15]: rown_wiki = [m2*l2*cos(phi1-phi2)*phi2dd+(m1+m2)*l1*phi1dd+m2*l2*phi2d^2 * sin(phi
          l2*phi2dd+l1*cos(phi1-phi2)*phi1dd-l1*phi1d^2*sin(phi1-phi2)+g*sin(phi2)]

In [16]: showmath(rown_wiki[0])
```

Out[16]:

$$l_2 m_2 \dot{\varphi}_2^2 \sin(\varphi_1 - \varphi_2) + l_2 m_2 \ddot{\varphi}_2 \cos(\varphi_1 - \varphi_2) + l_1(m_1 + m_2)\ddot{\varphi}_1 + g(m_1 + m_2)\sin(\varphi_1)$$

```
In [17]: showmath(rown_wiki[1])
```

Out[17]:

$$-l_1\dot{\varphi}_1^2\sin(\varphi_1-\varphi_2)+l_1\ddot{\varphi}_1\cos(\varphi_1-\varphi_2)+l_2\ddot{\varphi}_2+g\sin(\varphi_2)$$

```
In [18]: rown_wiki[0].show()
```

```
l2*m2*phi2d^2*sin(phi1 - phi2) + l2*m2*phi2dd*cos(phi1 - phi2) + l1*(m1 + m2)*phi1dd + g*(m
```

```
In [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
```

```
In [20]: rown_wiki[0].show()
```

```
l2*m2*phi2d^2*sin(phi1 - phi2) + l2*m2*phi2dd*cos(phi1 - phi2) + l1*(m1 + m2)*phi1dd + g*(m
```

```
In [21]: bool((eq1/l1) == rown_wiki[0] )
```

```
Out[21]: True
```

```
In [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)
```

```
In [23]: bool((eq2/l2/m2) == rown_wiki[1] )
```

```
Out[23]: True
```

13.1 Euler -Langrange

```
In [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)
```

In [25]:
```
showmath( Epot.collect(cos(phi1)) )
```

Out[25]:

$$-gl_2m_2\cos(\varphi_2) - (gl_1m_1 + gl_1m_2)\cos(\varphi_1)$$

In [26]:
```
showmath( Epot )
```

Out[26]:

$$-gl_1m_1\cos(\varphi_1) - (l_1\cos(\varphi_1) + l_2\cos(\varphi_2))gm_2$$

In [27]:
```
showmath( Ekin.trig_simplify() )
```

Out[27]:

$$\frac{1}{2}l_2^2m_2\dot{\varphi}_2^2 + \frac{1}{2}\left(l_1^2m_1 + l_1^2m_2\right)\dot{\varphi}_1^2 + (l_1l_2m_2\dot{\varphi}_1\cos(\varphi_1)\cos(\varphi_2) + l_1l_2m_2\dot{\varphi}_1\sin(\varphi_1)\sin(\varphi_2))\dot{\varphi}_2$$

In [28]:
```
L = Ekin - Epot
```

In [29]:
```
len(L.expand().operands())
```

Out[29]: 11

In [30]:
```
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)
```

In [31]:
```
EL1.expand().operands()
```

Out[31]:
```
[l1*l2*m2*phi2d^2*cos(phi2)*sin(phi1),
 -l1*l2*m2*phi2d^2*cos(phi1)*sin(phi2),
 l1^2*m1*phi1dd*cos(phi1)^2,
 l1^2*m2*phi1dd*cos(phi1)^2,
 l1*l2*m2*phi2dd*cos(phi1)*cos(phi2),
 l1^2*m1*phi1dd*sin(phi1)^2,
 l1^2*m2*phi1dd*sin(phi1)^2,
 l1*l2*m2*phi2dd*sin(phi1)*sin(phi2),
 g*l1*m1*sin(phi1),
 g*l1*m2*sin(phi1)]
```

In [32]:
```
EL1 = (EL1/l1).trig_reduce()
EL2 = (EL2/l2).trig_reduce()
showmath(EL1)
```

Out[32]:

$$-l_2 m_2 \dot{\varphi}_2^2 \sin\left(-\varphi_1+\varphi_2\right)+l_2 m_2 \ddot{\varphi}_2 \cos\left(-\varphi_1+\varphi_2\right)+l_1 m_1 \ddot{\varphi}_1+l_1 m_2 \ddot{\varphi}_1+g m_1 \sin\left(\varphi_1\right)+g m_2 \sin\left(\varphi_1\right)$$

In [33]:
```
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
```

In [34]:
```
expr = sol[0].rhs()
```

In [35]:
```
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)
```

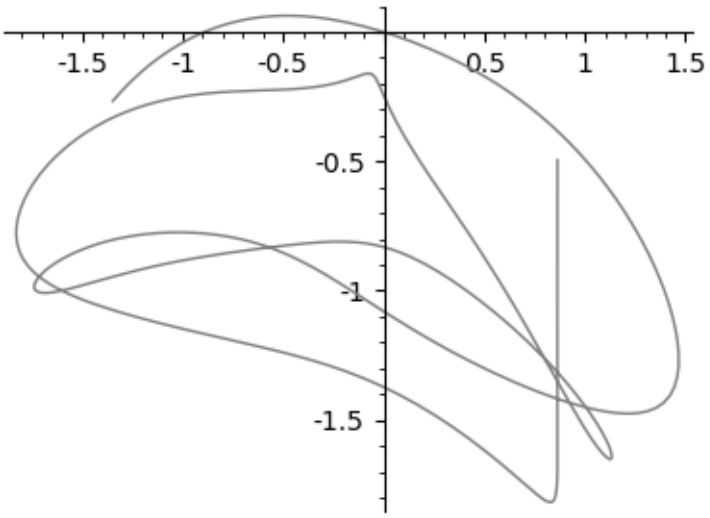
```
-g*m2*cos(-phi1 + phi2)*sin(phi2)
```

```
g*m1*sin(phi1)
```

```
g*m2*sin(phi1)
```

13.2 Numerical analysis

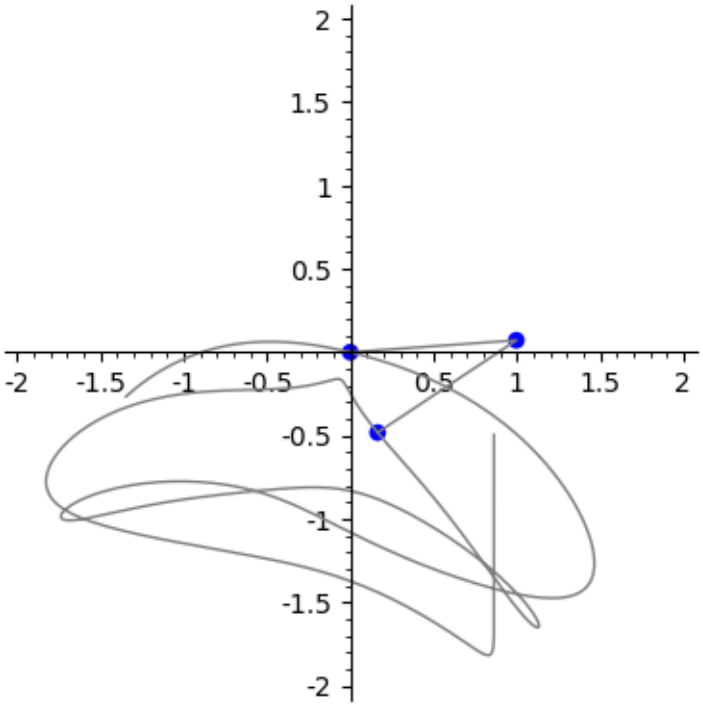
In [36]:
```
import numpy as np

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)
```



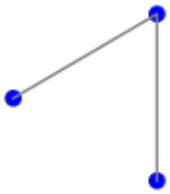

```
In [37]: def plot_dp(f1,f2,pars):
             mass1 = vector([x1,y1]).subs(x2u).subs(pars).subs({phi1:f1,phi2:f2})
             mass2 = vector([x2,y2]).subs(x2u).subs(pars).subs({phi1:f1,phi2:f2})
             plt = point([(0,0),mass1],aspect_ratio=1,size=40)
             plt += point(mass2,xmin=-2,xmax=2,ymin=-2,ymax=2,size=40)
             plt += line([(0,0),mass1,mass2],color='gray')

             return plt
```

```
In [38]: plot_dp(numsol[213,0],numsol[213,1],{l1:1,l2:1})+p
```

Out[38]:




```
In [39]: @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)
```



14 N pendula

Consider a system of N pendula suspended on the pendula.

```
In [1]: import numpy as np
```

```
In [2]: var('t,g')
```

```
Out[2]: (t, g)
```

```
In [3]: load('cas_utils.sage')
```

14.1 Triple pendulum

We can easily generalize our symbolic scheme to cases where the number of points is arbitrary.

```
In [4]: N = 3
        var('t g')
        for i in range(1,1+N):
            var('l%d m%d'%(i,i))
```

```
In [5]: 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)]
```

```
In [6]: load('cas_utils.sage')
```

```
In [7]: to_fun, to_var = make_symbols(xy_names,uv_names)
```

```
phi1  :: has been  processed
phi2  :: has been  processed
phi3  :: has been  processed
x1  :: has been  processed
x2  :: has been  processed
x3  :: has been  processed
y1  :: has been  processed
y2  :: has been  processed
y3  :: has been  processed
```

```
In [8]: 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)]
        ms = [vars()['m%d'%i] for i in range(1,1+N)]
```

```
phis = [vars()['phi%d'%i] for i in range(1,1+N)]
phids = [vars()['phi%dd'%i] for i in range(1,1+N) ]
phidds = [vars()['phi%ddd'%i] for i in range(1,1+N) ]

showmath(phis)
```

Out[8]:

$$[\varphi_1, \varphi_2, \varphi_3]$$

```
In [9]: 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_)
```

14.2 dAlembert

```
In [10]: transform_virtual_displacements(xy_names,uv_names,verbose=False)
```

```
Out[10]: [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*sin(phi1),
          dphi1*l1*sin(phi1) + dphi2*l2*sin(phi2),
          dphi1*l1*sin(phi1) + dphi2*l2*sin(phi2) + dphi3*l3*sin(phi3)]
```

```
In [11]: 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) ]
```

```
In [12]: dAlemb = sum( (m_*x_.subs(x2u).subs(to_fun).diff(t,2))*dx_ for m_,x_,dx_ in zip(ms
         dAlemb += sum( (m_*x_.subs(x2u).subs(to_fun).diff(t,2) + m_*g)*dx_ for m_,x_,dx_ in
         dAlemb = dAlemb.subs(to_var)
```

```
In [13]: #showmath(dAlemb)
```

```
In [14]: dphis = [vars()['dphi%d'%i] for i in range(1,1+N) ]
```

```
In [15]: eqs = [dAlemb.expand().coefficient(dphi_).trig_simplify() for dphi_ in dphis]
```

```
In [16]: showmath(eqs[1].trig_reduce())
```

Out[16]:

$$l_1 l_2 m_2 \dot{\varphi}_1^2 \sin(-\varphi_1+\varphi_2) + l_1 l_2 m_3 \dot{\varphi}_1^2 \sin(-\varphi_1+\varphi_2) - l_2 l_3 m_3 \dot{\varphi}_3^2 \sin(-\varphi_2+\varphi_3) + l_1 l_2 m_2 \ddot{\varphi}_1 \cos(-\varphi_1+\varphi_2) + l_1 l_2 m$$

In [ ]:

In [17]:
```
sol =  solve(eqs,phidds)[0]
```

In [18]:
```
len(sol[1].rhs().trig_reduce().operands())
```

Out[18]: 21

In [19]:
```
pars= {m_:1 for m_ in ms}

for i,l_ in enumerate(ls):
    pars[l_]  = 1/ (i+1)
pars[g] = 9.81
```

In [20]:
```
phidds
```

Out[20]: [phi1dd, phi2dd, phi3dd]

In [21]:
```
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)
```

In [ ]:

In [22]:
```
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)
```

14.3 Euler Lagrange formulation

```
In [23]: Ekin = 1/2 * sum(
                  m_*x_.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2 +\
                  m_*y_.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2 \
                           for m_,x_,y_ in zip(ms,xs,ys))

         Ekin = Ekin.trig_simplify()
         Epot = sum(m_*g*y_.subs(x2u) for m_,y_ in zip(ms,ys))
```

```
In [24]: showmath(Ekin)
```

Out[24]:

$$\frac{1}{2}l_3^2m_3\dot{\varphi}_3^2+\frac{1}{2}\left(l_1^2m_1+l_1^2m_2+l_1^2m_3\right)\dot{\varphi}_1^2+\frac{1}{2}\left(l_2^2m_2+l_2^2m_3\right)\dot{\varphi}_2^2+\left(\left(l_1l_2m_2+l_1l_2m_3\right)\dot{\varphi}_1\cos\left(\varphi_1\right)\cos\left(\varphi_2\right)+\left(l_1l_2m_2+\right.\right.$$

```
In [25]: showmath(Epot)
```

Out[25]:

$$-gl_1m_1\cos\left(\varphi_1\right)-\left(l_1\cos\left(\varphi_1\right)+l_2\cos\left(\varphi_2\right)\right)gm_2-\left(l_1\cos\left(\varphi_1\right)+l_2\cos\left(\varphi_2\right)+l_3\cos\left(\varphi_3\right)\right)gm_3$$

In [26]:
```
L = Ekin - Epot
```

In [27]:
```
ELs = [L.diff(phid_).subs(to_fun).diff(t).subs(to_var) - L.diff(phi_)\
        for (phi_,phid_) in zip(phis,phids)]
```

In [28]:
```
ELs = [(EL_/l_).trig_reduce() for EL_,l_ in zip(ELs,ls)]
```

In [29]:
```
showmath(ELs[0])
```

Out[29]:

$$-l_2 m_2 \dot{\varphi}_2^2 \sin(-\varphi_1+\varphi_2) - l_2 m_3 \dot{\varphi}_2^2 \sin(-\varphi_1+\varphi_2) - l_3 m_3 \dot{\varphi}_3^2 \sin(-\varphi_1+\varphi_3) + l_2 m_2 \ddot{\varphi}_2 \cos(-\varphi_1+\varphi_2) + l_2 m_3 \ddot{\varphi}_2 \cos(-\varphi_1+\varphi_2) + l_3 m_3 \ddot{\varphi}_3 \cos(-\varphi_1+\varphi_3) + g m_2 \sin\varphi_1 + g m_3 \sin\varphi_1$$

In [ ]:

In [30]:
```
sol =  solve(ELs,phidds)[0]
#show(sol)
```

In [31]:
```
pars= {m_:1 for m_ in ms}

for i,l_ in enumerate(ls):
    pars[l_]  = 0.3+1/ (i+1)
pars[g] = 9.81
```

In [32]:
```
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)
```

In [ ]:

In [33]:
```
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)
```

In [ ]:

15 Spherical pendulum

```
In [1]: import numpy as np
```

```
In [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')
```

```
In [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
```

```
In [4]: x2u = {x:l*sin(theta)*cos(phi),y: l*sin(theta)*sin(phi),z: l*cos(theta)}
```

```
In [5]: transform_virtual_displacements(xy_names,uv_names,suffix='_uv')
```

```
Out[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)]
```

```
In [6]: dx_uv
```

```
Out[6]: dtheta*l*cos(phi)*cos(theta) - dphi*l*sin(phi)*sin(theta)
```

```
In [ ]:
```

```
In [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)
```

```
In [8]: show(dAlemb)
```

```
(l*thetad^2*cos(theta) + l*thetadd*sin(theta) - g)*dtheta*l*sin(theta) - (2*l*phid*thetad*co
```

```
In [9]: sol = solve(\
            [dAlemb.expand().coefficient(dtheta).trig_simplify(),\
              dAlemb.expand().coefficient(dphi).trig_simplify()],\
                    [phidd,thetadd])[0]
```

```
In [10]: show(sol)
```

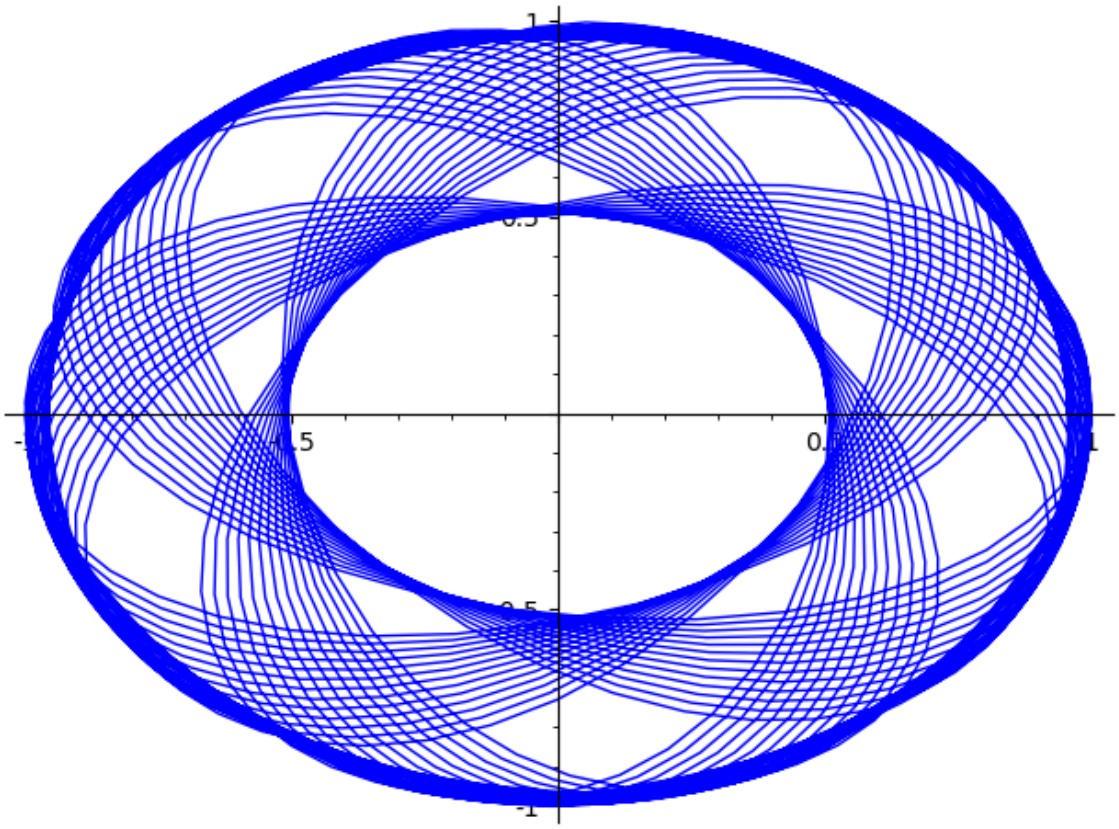
```
[phidd == -2*phid*thetad*cos(theta)/sin(theta),
 thetadd == (l*phid^2*cos(theta) + g)*sin(theta)/l]
```

```
In [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]
```

```
In [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_ra
         p = line ( zip(np.sin(numsol[:,1])*np.cos(numsol[:,0]),np.sin(numsol[:,0])*np.sin(n
         p.show()
```

```
[phid,
 thetad,
 -2*phid*thetad*cos(theta)/sin(theta),
 (phid^2*cos(theta) + 1)*sin(theta)]
```

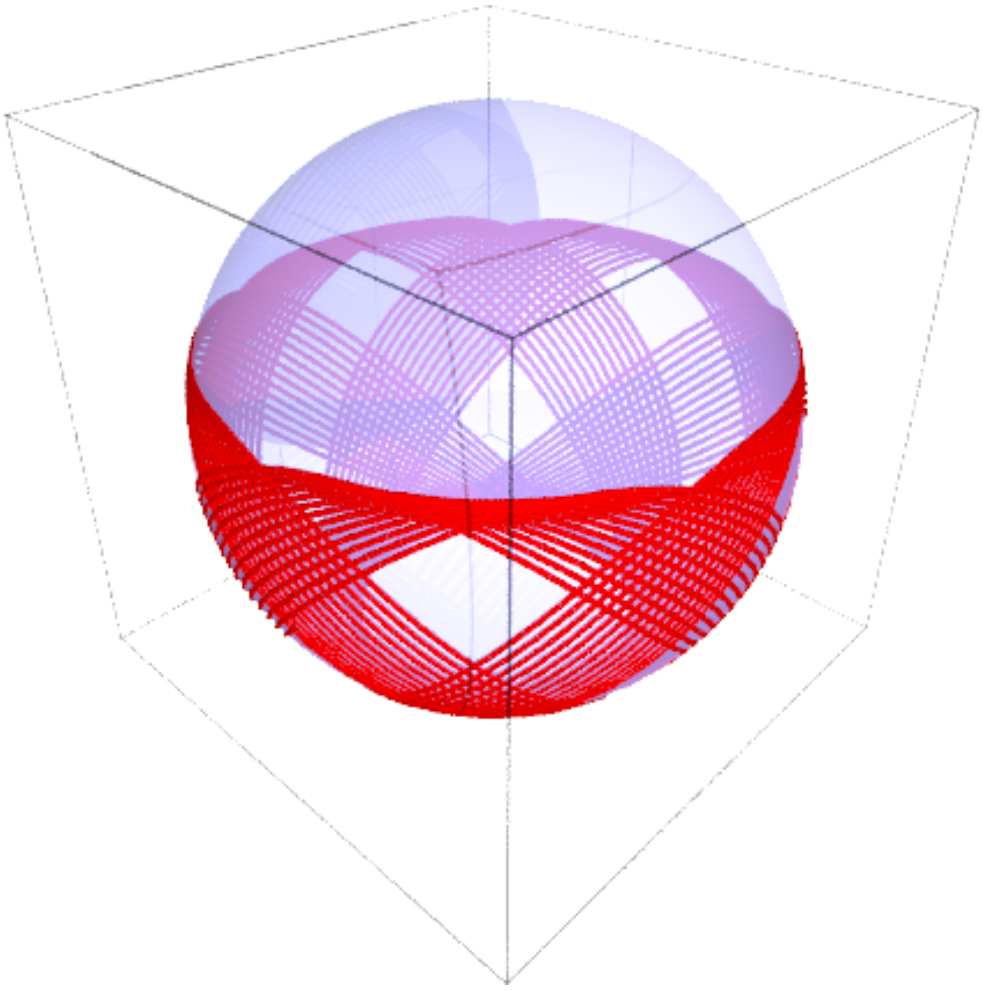


$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\phi = \text{arctg}\frac{y}{x}$$

$$\theta = \arcsin\frac{z}{r}$$

```
In [13]: p3d = line3d( zip(np.sin(numsol[:,1])*np.cos(numsol[:,0]),np.sin(numsol[:,0])*np.si
         p3d += sphere(opacity=0.5)
         p3d.show(viewer='tachyon')
```



16 Paraboloidal pendulum

16.1 System definition

In [1]:
```
import numpy as np
```

In [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
```

In [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)
```

In [4]:
```
showmath(dAlemb)
```

Out[4]:

$$\delta z(g+\ddot{z})+\delta x\ddot{x}+\delta y\ddot{y}$$

In [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)
```

Out[5]:

$$\delta xx+\delta yy-\delta z$$

In [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])
```

Out[6]:

$$[gx^2 + x^2\ddot{z} + x\ddot{x}, gxy + xy\ddot{z} + x\ddot{y}]$$

In [7]:
```
sol = solve([f.subs(to_fun).diff(2).subs(to_var),eq1,eq2],[xdd,ydd,zdd])[0][:2]
```

In [8]:
```
showmath(sol)
```

Out[8]:

$$\left[\ddot{x} = -\frac{x\dot{x}^2 + x\dot{y}^2 + gx}{x^2 + y^2 + 1}, \ddot{y} = -\frac{y\dot{y}^2 + (\dot{x}^2 + g)y}{x^2 + y^2 + 1}\right]$$

In [9]:
```
ode = [xd,yd] + [s_.rhs() for s_ in sol]
```

16.2 Numerical analysis

In [10]:
```
ode = map(lambda x:x.subs({l:1,g:1}),ode)
showmath(ode)
```

Out[10]:

$$\left[\dot{x}, \dot{y}, -\frac{x\dot{x}^2 + x\dot{y}^2 + x}{x^2 + y^2 + 1}, -\frac{y\dot{y}^2 + (\dot{x}^2 + 1)y}{x^2 + y^2 + 1}\right]$$

In [11]:
```
times = srange(0,237,.1)
```

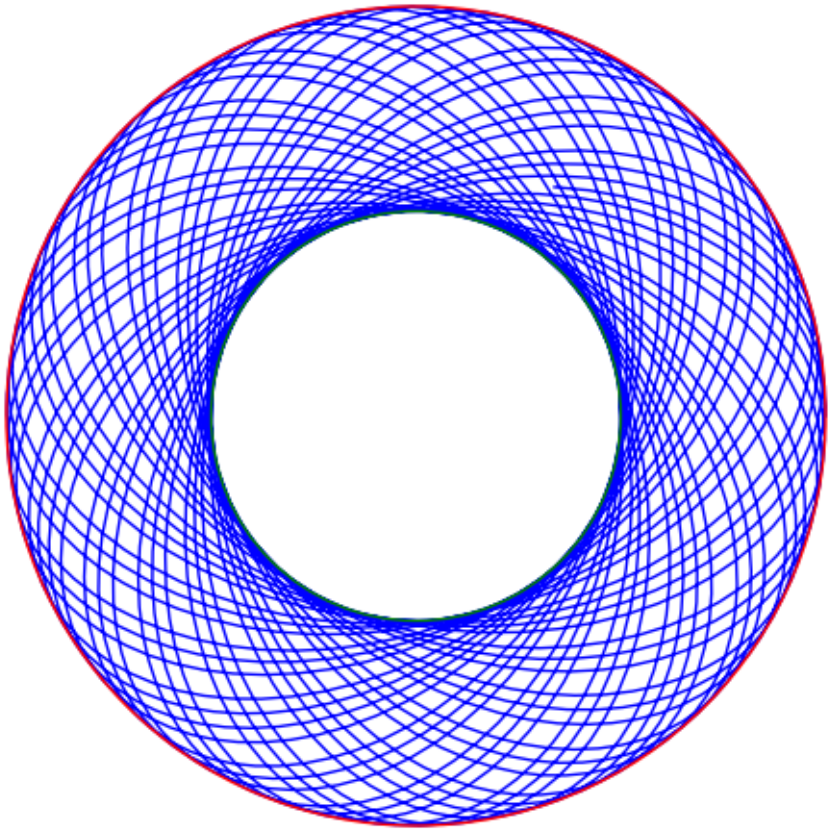
In [12]:
```
z_prime = (1/2*(x^2+y^2)).subs(to_fun).diff(t).subs(to_var)
z_prime.show()
```

```
x*xd + y*yd
```

In [13]:
```
Ekin = (1/2*(xd^2+yd^2+z_prime^2))
```

In [ ]:

In [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)
```



```
In [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!
```

```
In [16]: Ekin_num + Epot_num
```

```
Out[16]: array([0.625, 0.6249999943305325, 0.6249999979623393, ...,
                0.625004240168609, 0.6250042526868687, 0.6250042649374952],
               dtype=object)
```

16.3 Angular momentum

We can calculate symbolically angular momentum and chech if its z-component is conserved in time.

```
In [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])
```

In [18]:
```
p = e_r.cross_product(v)
```

In [19]:
```
showmath(p[0].full_simplify())
```

Out[19]:

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

In [20]:
```
Ekin_num = [Ekin.subs({x:d[0],y:d[1],xd:d[2],yd:d[3]}) for d in numsol]
```

In [21]:
```
P_num = [(p[2]).subs({x:d[0],y:d[1],xd:d[2],yd:d[3]}) for d in numsol]
```

In [22]:
```
P_num[0]
```

Out[22]:
```
0.5
```

In [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
```

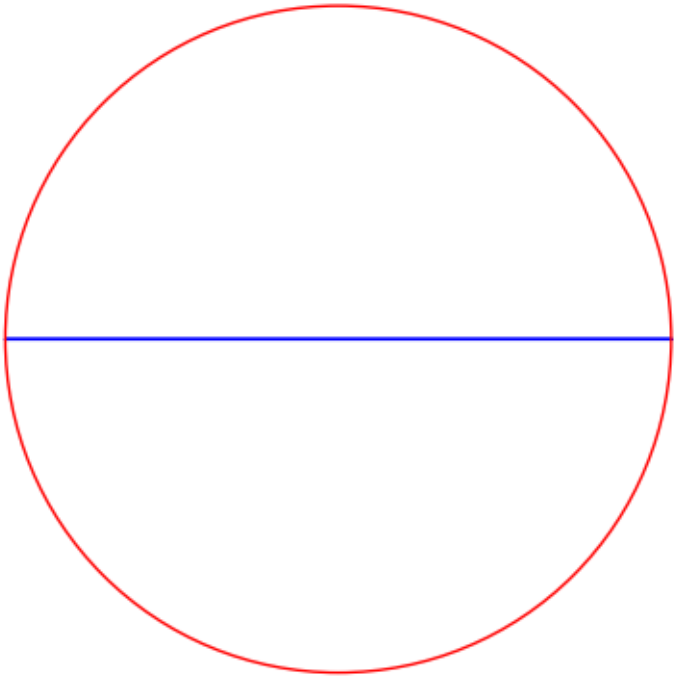
In [24]:
```
Ekin.show()
```

```
1/2*(x*xd + y*yd)^2 + 1/2*xd^2 + 1/2*yd^2
```

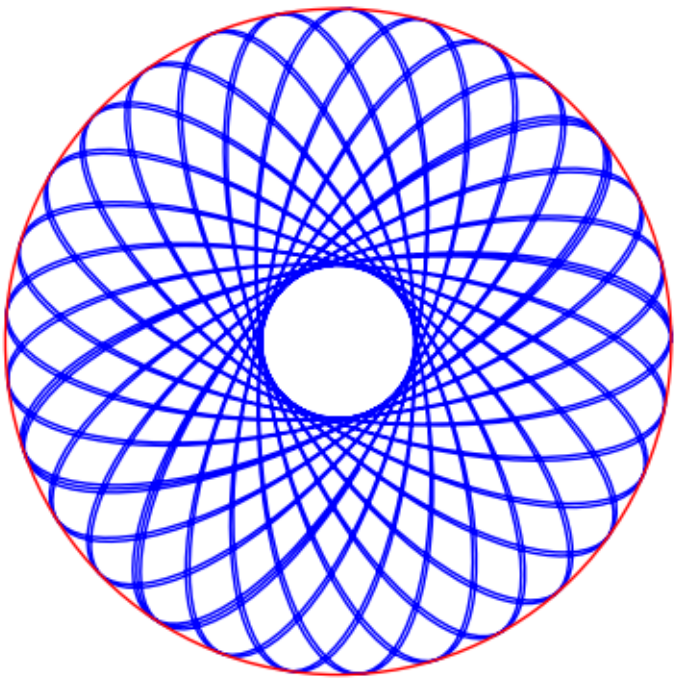
In [25]:
```
P_num[:4]
```

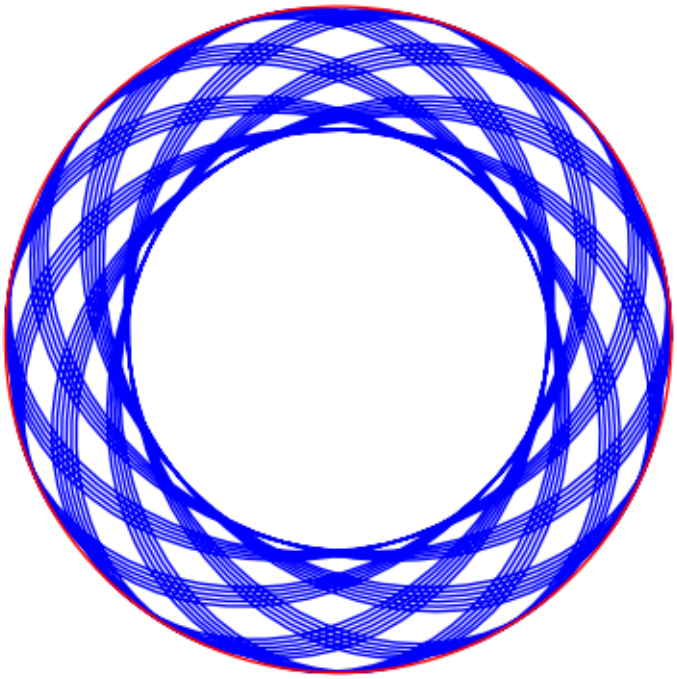
Out[25]:
```
[0.5, 0.4999999909879266, 0.49999999698576963, 0.4999999949420133]
```

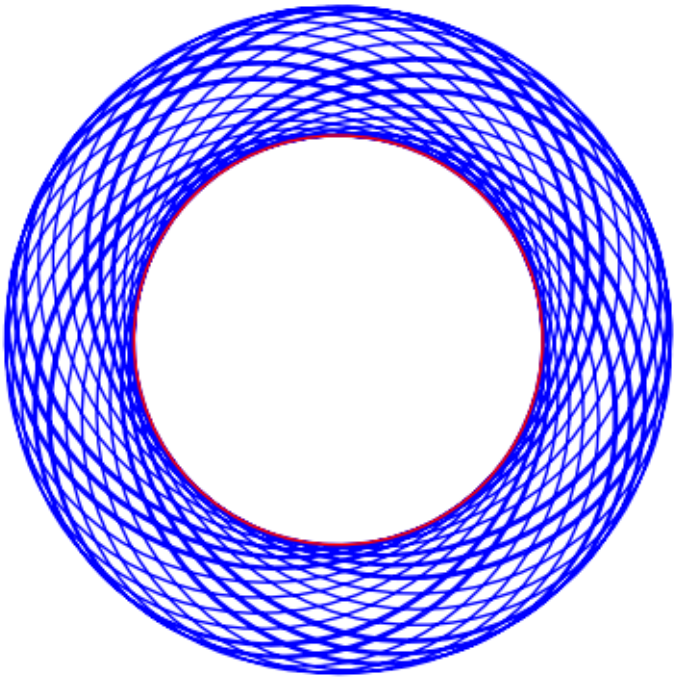
In [26]:
```
@interact
def plot2d_traj(v0=slider(0,3,0.001)):
    numsol = desolve_odeint(ode,[1,0,0,v0],times,[x,y,xd,yd])
    p = line( zip(numsol[:,0],numsol[:,1]), aspect_ratio=1)
    p += circle( (0,0),1,color='red')
    p.show(axes=False)
```



In [27]:
```
plot2d_traj(v0=0.23),plot2d_traj(v0=0.63),plot2d_traj(v0=1.63),
```

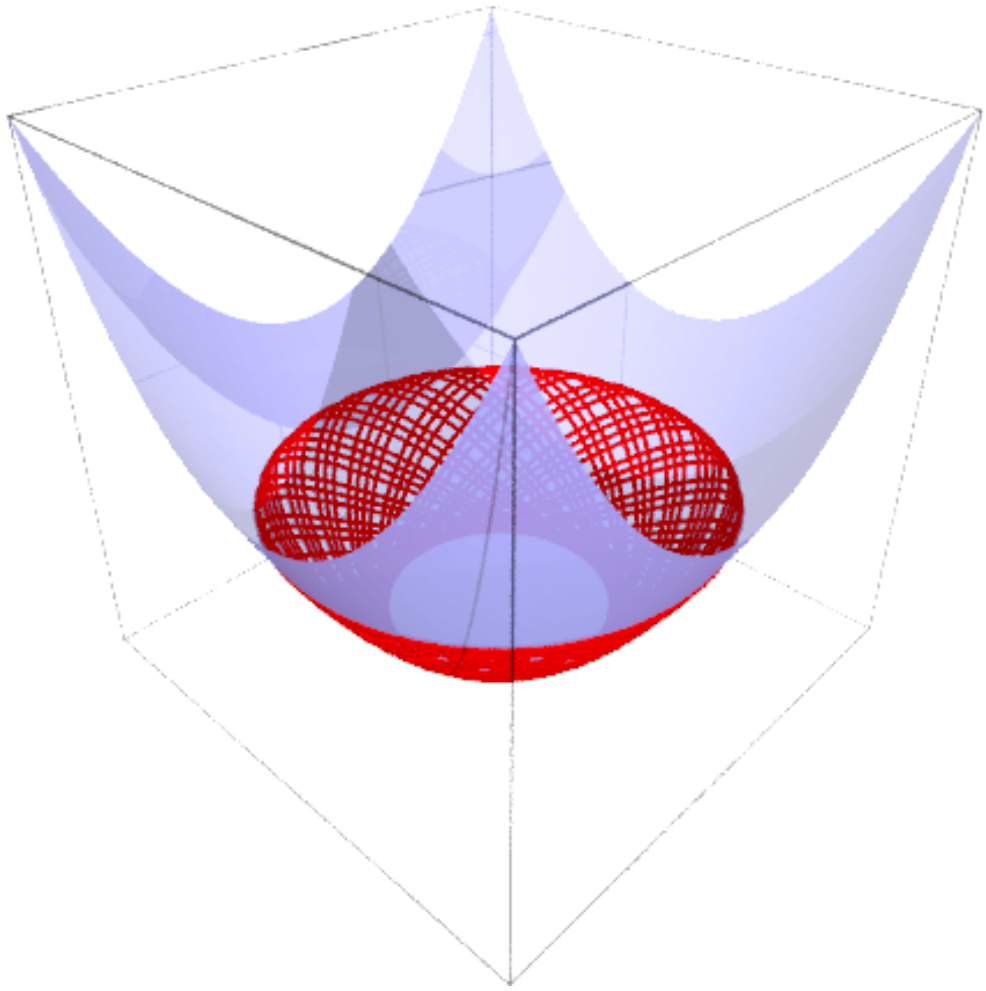






```
Out[27]: (None, None, None)
```

```
In [28]: p3d = line3d( zip(numsol[:,0],numsol[:,1],(1/2*(numsol[:,0]**2+numsol[:,1]**2))),th
         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')
```



17 Point particle on the cone

Consider the movement of the material point on the surface of the cone.

```
In [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())

In [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)
```

$$-z^2\tan(a)^2 + x^2 + y^2$$

$$-2dz z\tan(a)^2 + 2dx x + 2dy y$$

```
In [3]: f.solve(z)
```

$$\left[z = -\frac{\sqrt{x^2 + y^2}}{\tan(a)}, z = \frac{\sqrt{x^2 + y^2}}{\tan(a)}\right]$$

```
In [4]: dAlemb_xy = dAlemb.subs(constr.solve(dz)[0]).subs(f.solve(z)[1])
```

```
In [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__
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__
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))
```

```
In [ ]:
```

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.

```
In [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
        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:

```
In [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)


In [8]: f_uv  = f.subs(x2u).trig_simplify()

In [9]: duv = [vars()['d'+repr(zm)] for zm in uv]
        constr =sum([dzm*f_uv.diff(zm) for zm,dzm in zip(uv,duv)])

In [10]: show(constr.solve(dz)[0])

dz == dr*r*cos(a)^2/(z*sin(a)^2)


In [11]: dAlemb.subs( constr.solve(dz)[0] )

Out[11]: -(phid^2*r*sin(phi) - phidd*r*cos(phi) - 2*phid*rd*cos(phi) - rdd*sin(phi))*(dphi*r

In [12]: zdd_rdd = (f_uv.solve(z)[1]).subs(to_fun).diff(t,2).subs(to_var)
         zdd_rdd

Out[12]: zdd == rdd*cos(a)/sin(a)

In [13]: dAlemb_uv = dAlemb.subs( constr.solve(dz)[0] ).subs(f_uv.solve(z)[1]).subs(zdd_rdd)

In [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__
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__
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) + (p


In [15]: table([rown.trig_simplify() for rown in solve([r1,r2],[rdd,phidd])[0]])

Out[15]:   rdd == phid^2*r*sin(a)^2 - g*cos(a)*sin(a)   phidd == -2*phid*rd/r
```

In [ ]:

In [ ]:

18 Paraglider flight mechanics in 2d

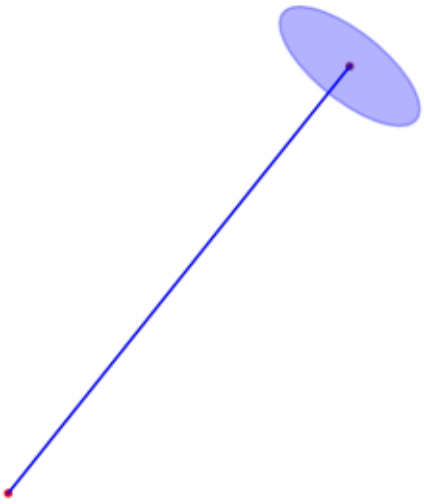
(m_1, x_1) pilot

(m_2, x_2) wing

F_x, F_y, aerodynamic force in the rest system

```
In [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
```

```
In [2]: plt = make_fig(1,2,3,4)
        plt.show(figsize=4,aspect_ratio=1,axes=false)
```



In [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
    #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)
```

In [4]:
```
show(dAlemb)
```

```
-(l*phid^2*sin(phi) - l*phidd*cos(phi) - xdd)*(dphi*l*cos(phi) + dx)*m1 + (dphi*l*sin(phi) -
```

In [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)
```

Out[5]:

l^2*m1*phidd + l*m1*xdd*cos(phi) + g*l*m1*sin(phi) + l*m1*ydd*sin(phi)	-l*m1*phi

In [6]:
```
#drugie =   [vars()[repr(v)+'dd'] for v in uv]
sol = solve( rown,drugie)
show( map(lambda x:x.trig_simplify(),sol[0]) )
```

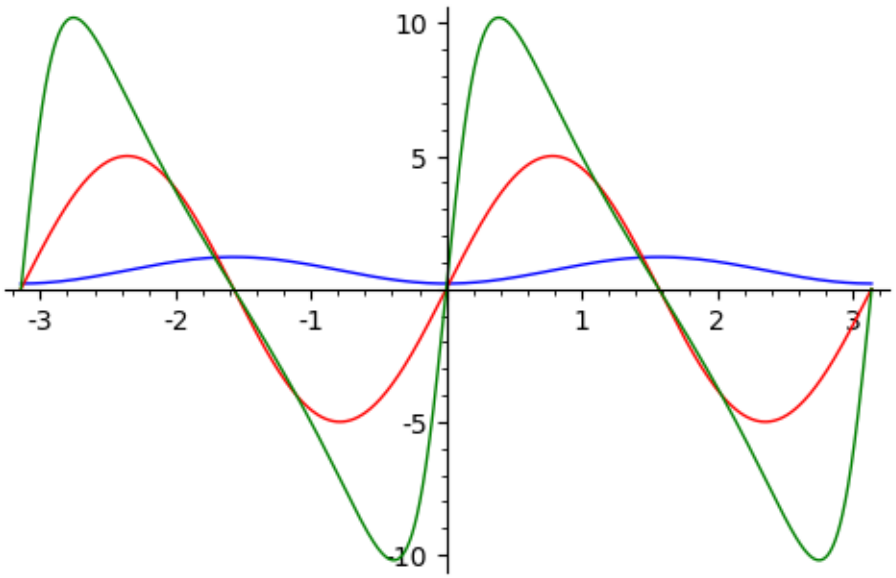
```
[phidd == -(Fx*cos(phi) + Fy*sin(phi))/(l*m2),
 xdd == (l*m1*m2*phid^2*sin(phi) + Fx*m1*cos(phi)^2 + Fy*m1*cos(phi)*sin(phi) + Fx*m2)/(m1*m
 ydd == -(l*m1*m2*phid^2*cos(phi) - Fx*m1*cos(phi)*sin(phi) - Fy*m1*sin(phi)^2 + g*m2^2 + (g
```

In [7]:
```
rhs = map(lambda x:x.rhs().trig_simplify(),sol[0])
show(rhs)
```

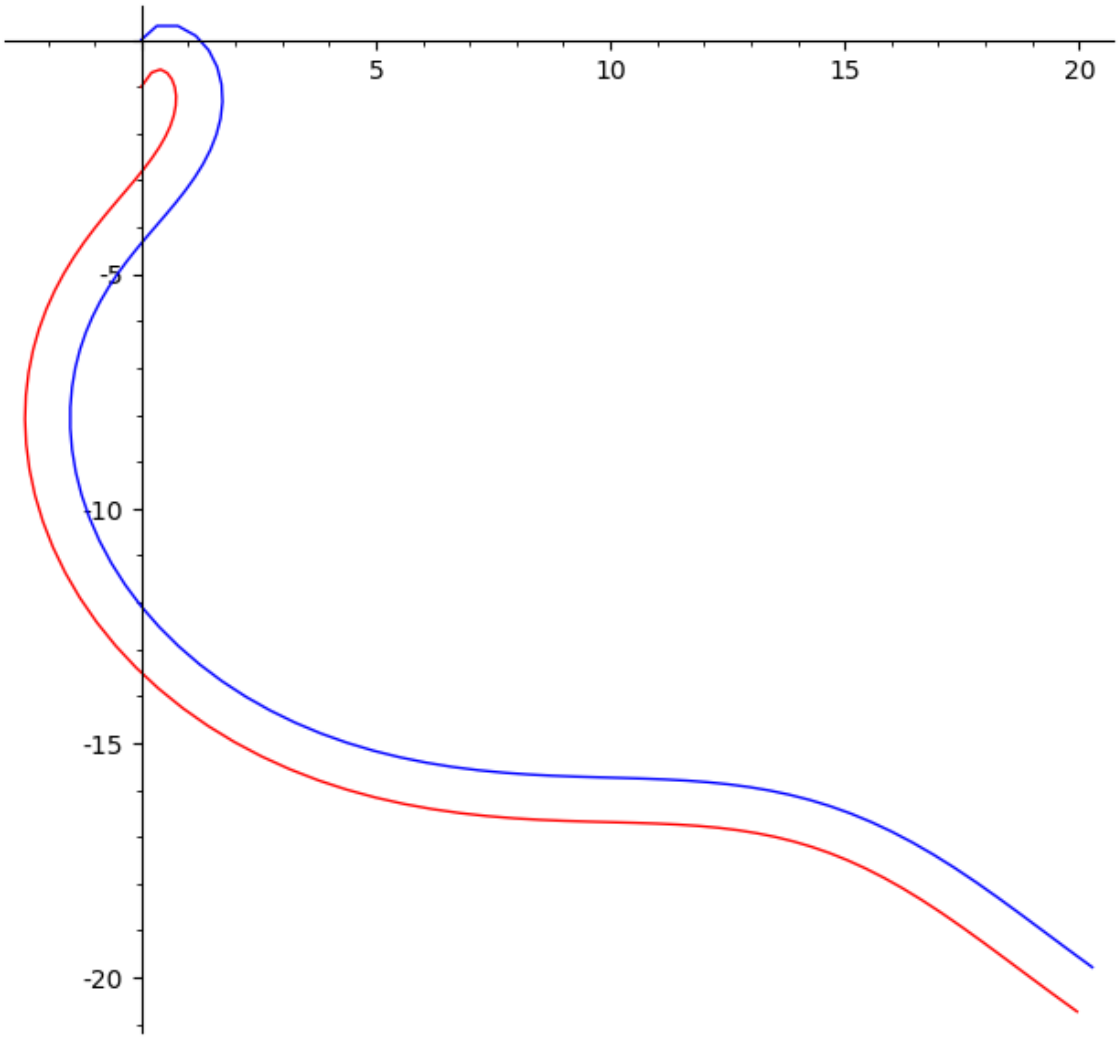
```
[-(Fx*cos(phi) + Fy*sin(phi))/(l*m2),
 (l*m1*m2*phid^2*sin(phi) + Fx*m1*cos(phi)^2 + Fy*m1*cos(phi)*sin(phi) + Fx*m2)/(m1*m2 + m21
 -(l*m1*m2*phid^2*cos(phi) - Fx*m1*cos(phi)*sin(phi) - Fy*m1*sin(phi)^2 + g*m2^2 + (g*m1 - F
```

In [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')
```

Out[8]:



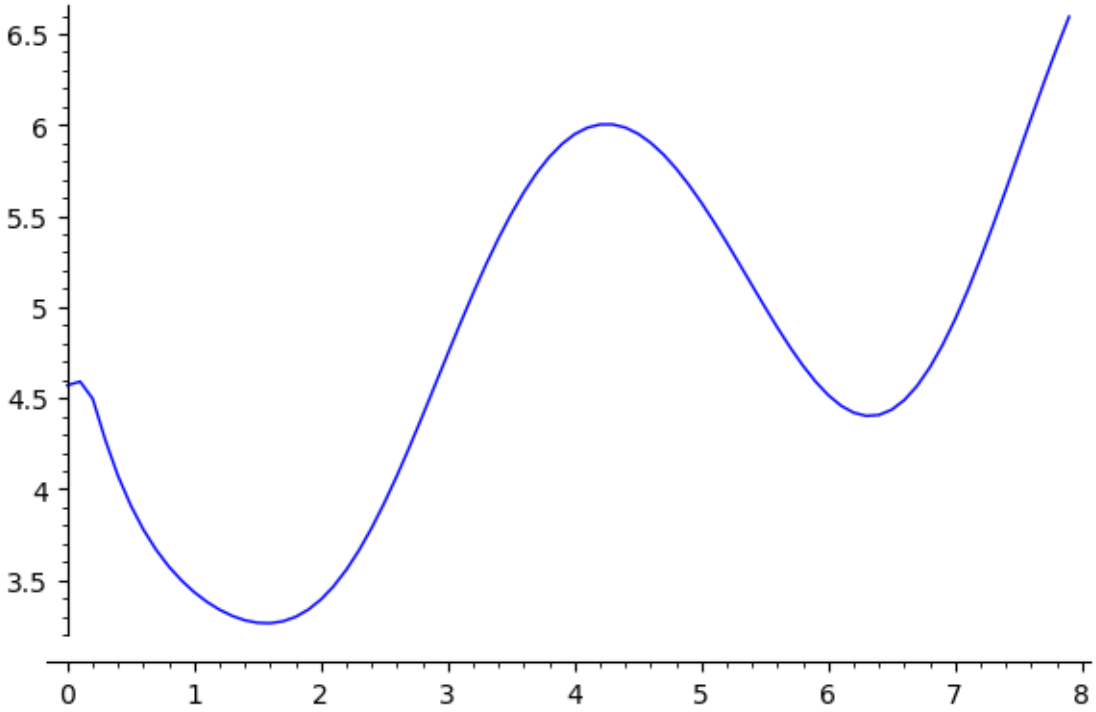
```
In [9]: #Cx = lambda x:1.0
        #Cz = lambda x:1.0
        trim = -0.24
        alpha = arctan2(yd,xd)
        forces={Fx:(-Cx(phi-alpha+trim)*cos(alpha) - Cz(phi-alpha+trim)*sin(alpha) )*(xd^2+y
        #show( map(lambda x:x.subs(forces),rhs) )
        dof3 = [phid,xd,yd] + map(lambda x:x.subs(forces),rhs)

In [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 1.1 s, sys: 43.5 ms, total: 1.14 s
Wall time: 1.12 s


In [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])),colo
         p.show()
```



In [ ]:

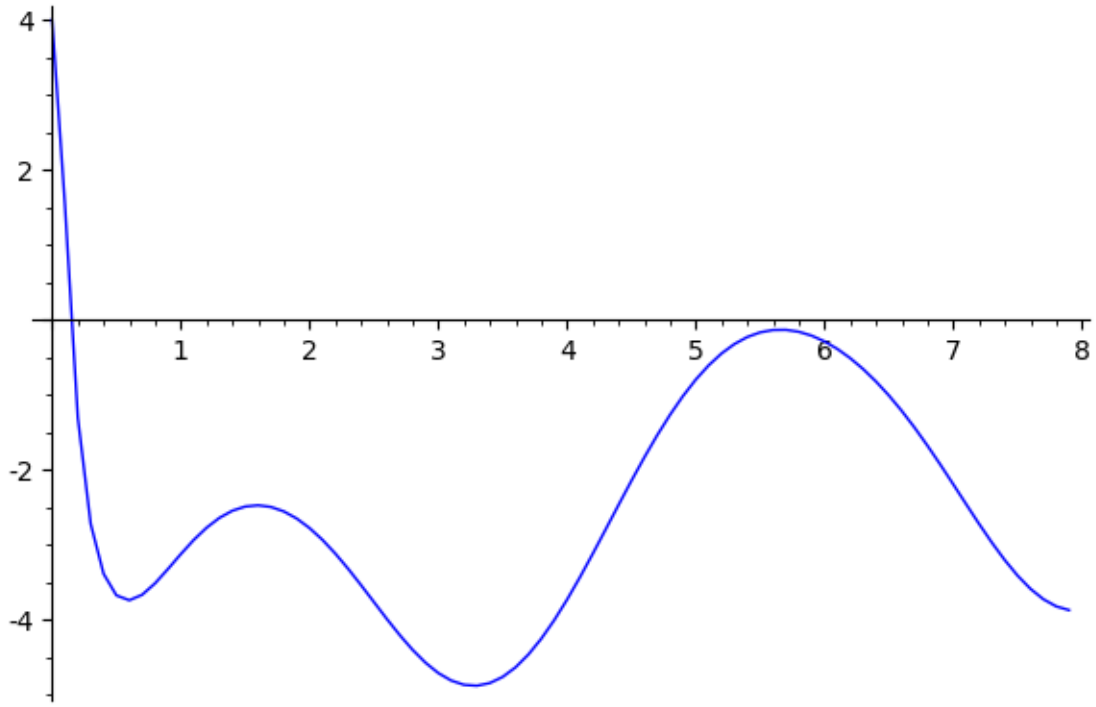
In [12]: line(zip(times,np.sqrt(numsol[:,4]**2+numsol[:,5]**2)))

Out[12]:




```
In [13]: line(zip(times,numsol[:,5]) )
```

Out[13]:



```
In [14]: every = max(int(numsol.shape[0]/25),1)
         every
```

```
Out[14]: 3
```

```
In [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
```

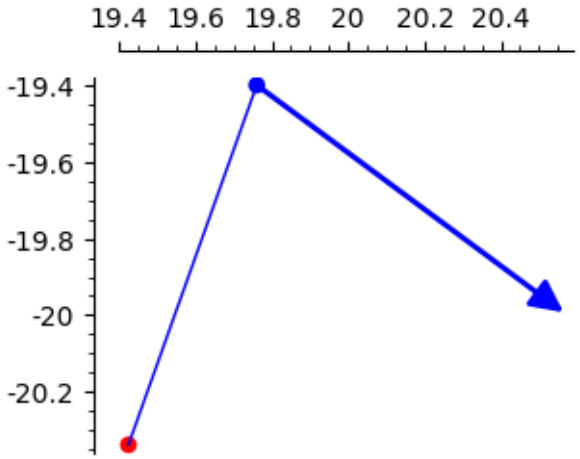
```
In [16]: plts = [p+plot_system(data,l=1) for data in numsol[::every,:]]
```

```
In [17]: %%time
         anim = animate(plts,xmax=125)
         anim.show()
```

```
Animation with 27 frames


CPU times: user 4.94 s, sys: 104 ms, total: 5.05 s
Wall time: 7.09 s
```

```
In [18]: plot_system(data,l=1)
```

```
Out[18]:
```

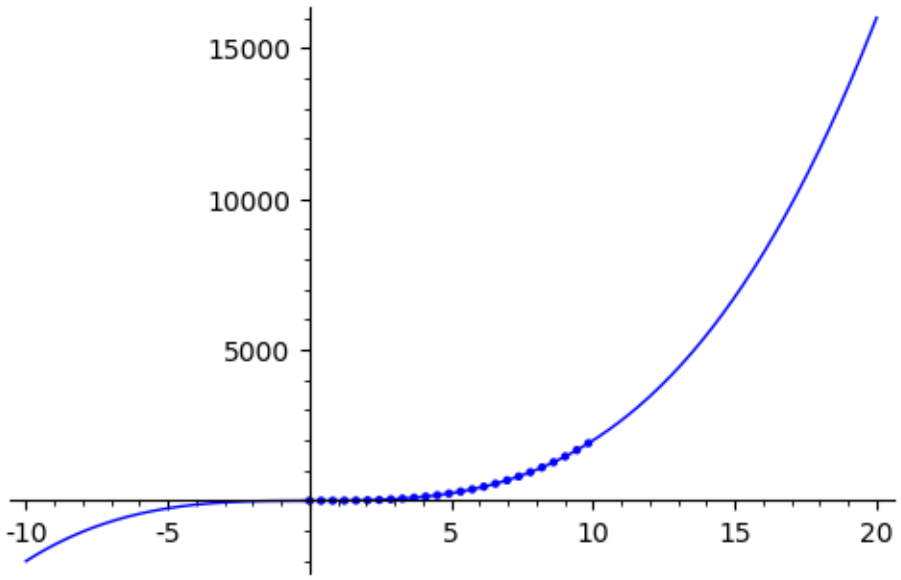


In [ ]:

In [ ]:

18.1 C_L from data

```
In [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)
```



Out[19]: 2.0000255313797446*x^3 + 0.30404394754745556

19 Appendix: a gentle introduction to differential equations

19.1 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.

19.2 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:

$$\underbrace{\frac{dx}{dt}}_{\text{derivative}} = \underbrace{f(x,t)}_{\text{Right Hand Side}}$$

19.3 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.

19.4 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:

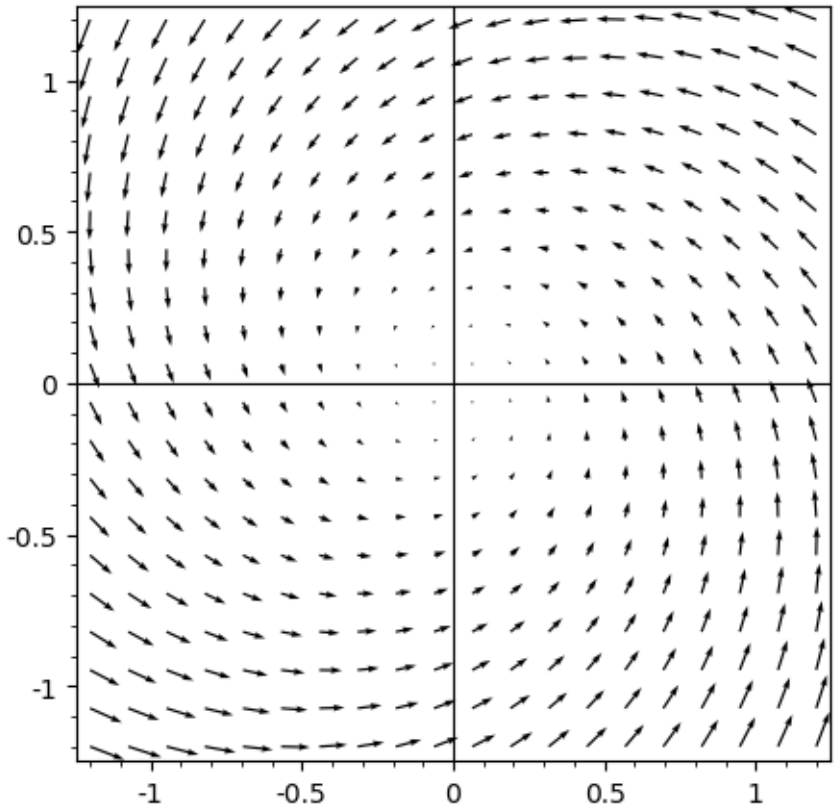
$$\vec{F}(x,y) = \begin{cases} f(x,y) \\ g(x,y) \end{cases}$$

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`

```
In [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
        plt_v
```

Out[1]:



19.5 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

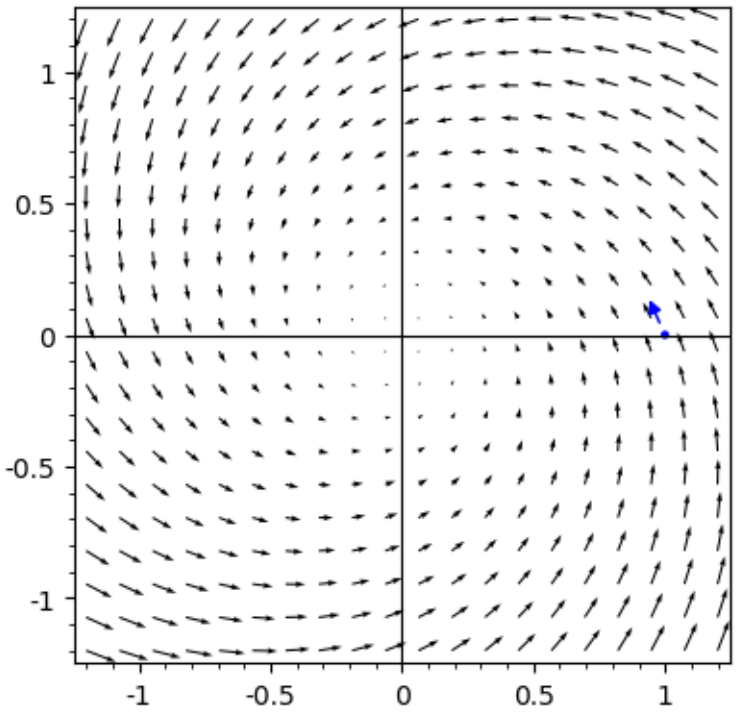
2. we calculate the point at the moment $t + h$
3. we draw the end point on the graph
4. we take the final point as the starting point
5. we go back to 1.

```
In [2]: x0,y0 = (1,0)
        h = 0.2
```

By executing this cell many times we get the next steps of the algrorytm:

```
In [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)
        x0,y0  = x1,y1
        plt_v
```

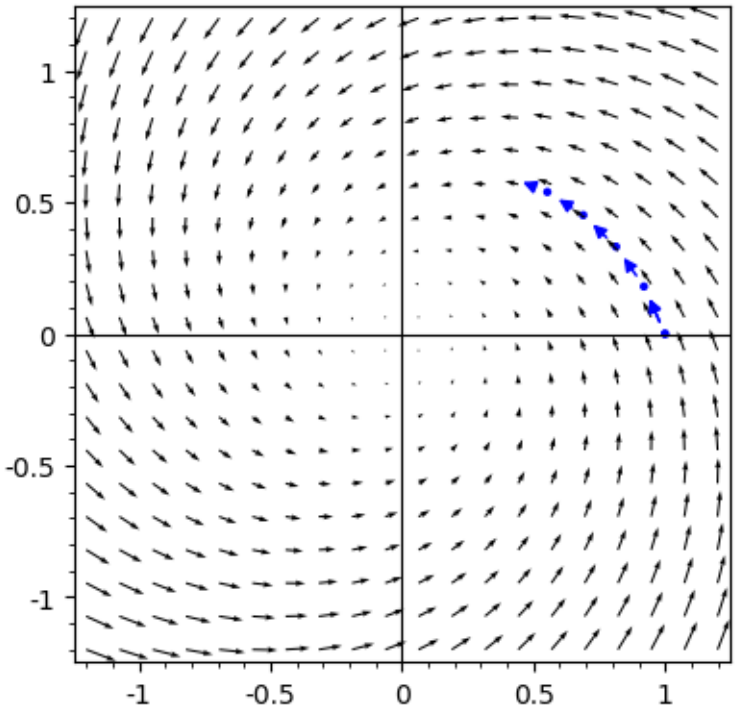
Out[3]:




```
In [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))
            x0,y0 = x1,y1
            plts.append(plt_v)
```

```
In [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.

19.6 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.

19.6.1 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:

$$x(t = 0) = e^{-k0+C} = e^C.$$

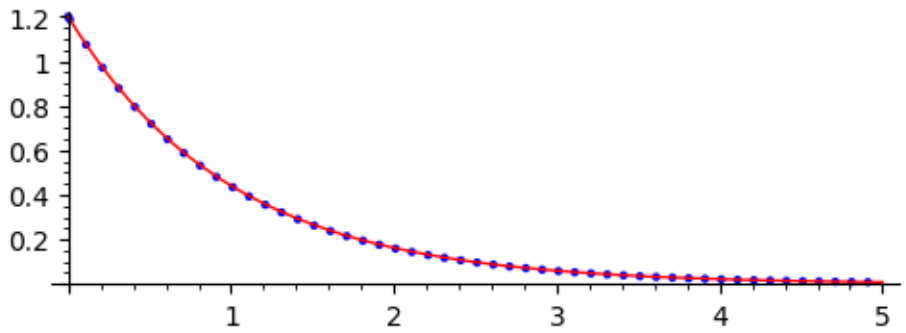
So we can save the solution with the initial condition $x(0) = x_0$ as:

$$x(t) = x_0 e^{-kt}.$$

Let's check if this solution agrees with the obtained approximate method:

```
In [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
```

Out[6]:



19.7 Solving ODEs using `desolve_odeint`

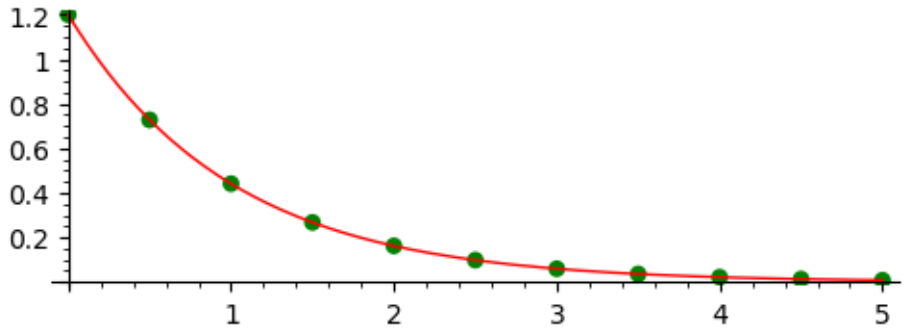
There are several algorithms built in to the Sage system that significantly more accurately and more efficiently solve differential equations. Without going into the details of their implementation, it is worth learning them use.

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:

```
In [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,
```

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:

```
In [8]: sol.shape
```

```
Out[8]: (11, 1)
```

```
In [9]: type(sol)
```

```
Out[9]: <type 'numpy.ndarray'>
```

19.7.1 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:

$$\begin{cases} \dot{x} = v \\ \dot{v} = -kx \end{cases}$$

```
In [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.

```
In [11]: sol [::200, :]
```

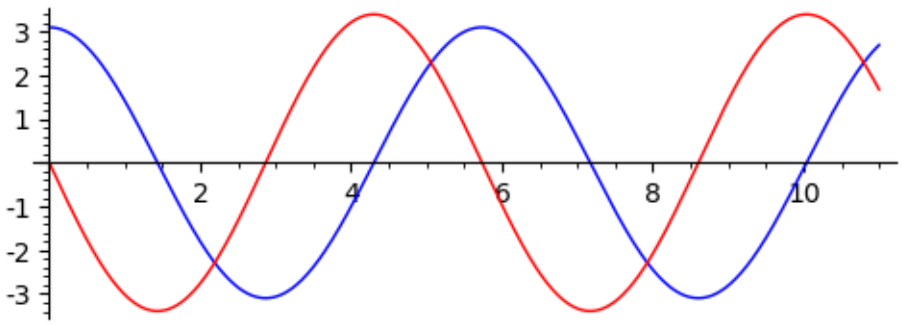
```
Out[11]: array([[ 1.        ,  0.        ],
                [ 0.69241901,  0.7903589 ],
                [-0.0411118 ,  1.09451919]])
```

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:

```
In [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
```

```
Out[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:

```
In [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)
```

```
Out[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 $v(0) = v_0$:

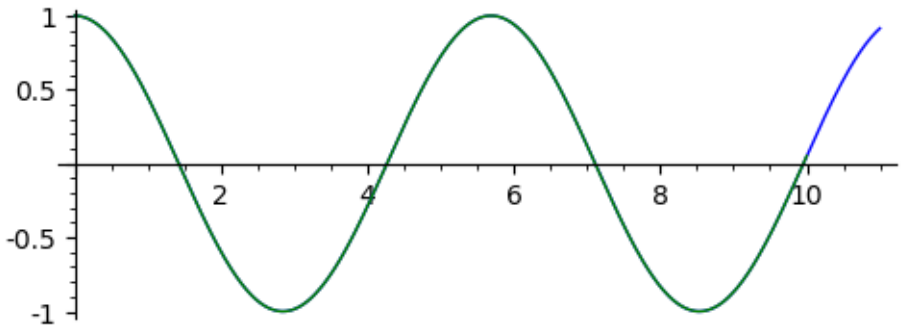
```
In [14]: var('t k')
         assume(k>0)
         x = function('x')(t)
         de = diff(x,t,2) == -k*x
         var('v0,x0')
         show( desolve(de, x,ics=[0,x0,v0],ivar=t))
```

```
x0*cos(sqrt(k)*t) + v0*sin(sqrt(k)*t)/sqrt(k)
```

Let's compare the numerical and analytic solution for the condition initial $x_0, v_0 = 0, 1$:

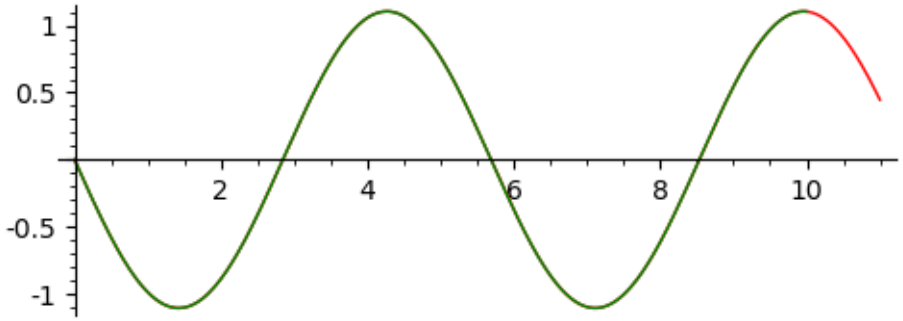
```
In [15]: sage: var('t x v')
         sage: k=1.22
         sage: sol = desolve_odeint([v, -k*(x)], [1.,0], times, [x,v])
         sage: px = line(zip(times,sol[:,0]),figsize=(5,2))
         sage: px+plot(cos(sqrt(k)*t),(t,0,10),color='green')
```

```
Out[15]:
```




```
In [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')
```

Out[16]:



19.7.2 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:

$$\begin{cases} \dot{x} = v \\ \dot{v} = -k\sin(x) \end{cases}$$