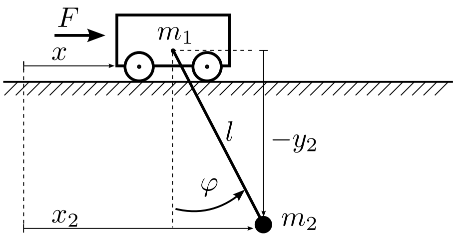
Exercise 03: `numpy + scipy`
Numerical Calculations with Python

The goal of the exercise is to get familiar with the packages `numpy` and `scipy` by means of simulation and identification of dynamical systems.

Exercise 03.1: Simulation with `solve_ivp`

We consider the following mechanical system:



with coordinates: $q_1(t) = x(t), \;\; q_2(t) = \varphi(t)$,
auxiliary geometric quantities:
$x_2(t) := x(t) + l \sin \varphi(t), \; y_2(t) := -l \cos \varphi(t)$,

and with the following equations of motion:

$$\ddot{x} = \frac{F + g m_2 \sin(\varphi) \cos(\varphi) + l m_2 \dot{\varphi}^2 \sin(\varphi)}{m_1 - m_2 \cos^2(\varphi) + m_2}, \tag{1a}$$

$$\ddot{\varphi} = -\frac{F \cos(\varphi) - g(m_1 + m_2) \sin(\varphi) - l m_2 \dot{\varphi}^2 \sin(\varphi) \cos(\varphi)}{l m_1 - l m_2 \cos^2(\varphi) + l m_2}. \tag{1b}$$

Note: In course09 (symbolic computation) we will derive these equations with `sympy` based on the so called "Euler-Lagrange-Equations". For this exercise we will simply use their implementation as executable functions (`xdd_fnc` , `phidd_fnc`) in the module `equations_of_motion.py` .
An analytical solution to these equations does not exist. With numerical integration methods, however, an approximative solution (i.e. a time evolution of all motion quantities) for given initial values can be determined.
Hints:

- Edit the given file `skeleton-code/01_simulation.py` (i.e. replace the occurences of `XXX` and add your own code.
- Make sure that you have read and understood the contents of the notebook simulation_of_dynamical_systems.ipynb.
- This exercise is quite hard. Feel free to sneek to `solution-code` . See also `exercise03-_solution_diff.pdf`.

1. Import the function `solve_ivp` from the appropriate sub package of `scipy` (see lecture slides). Then import the functions for numerical calculation of accelerations $\ddot{x}$ and $\ddot{\varphi}$ from the module `equations_of_motion` ,
2. Write a function `rhs(t, z)` which calculates the derivative $\dot{\mathbf{z}}$ of the state vector z. Assume the following definition of the state vector:
$$\mathbf{z} = (z_1, z_2, z_3, z_4)^T = (x, \varphi, \dot{x}, \dot{\varphi})^T$$
Note: The right hand side of the (vectorial) ODE does not depend on time.
3. Create an array for the simulation time and define reasonable initial conditions (e.g. $x(0) = 0$, $\varphi(0) = \frac{\pi}{2}$, $\dot{x}(0) = 0$, $\dot{\varphi}(0) = 0$).
4. Use the function `solve_ivp` from the module `scipy.integrate` (see docs) to determine (approximately) the time evolution of the four state variables in the time interval $t \in [0, 10]s$. Specify the optional argument `rtol=1e-5` as an upper bound on the relative error tolerance to obtain sufficient accuracy.
5. Plot $x(t)$ and $\varphi(t)$ using the existing code.

Exercise 03.2: Parameter identification with `minimize`

Now assume the parameters m_2 and l are unknown, but measured values of the motion exist. Using `minimize` (see docs) you shall find those values for m_2 and l, with which the measured values are best reproducible by simulation.
Note: For these tasks you are supposed to edit the file `sekeleton-code/identification.py` and use your knowledge from exercise 03.1!

1. Import the necessary functions (`solve_ivp` , `xdd_fnc` , ...). Then load the (fictitious) measured values from the corresponding file using `np.load(..)` or `np.loadtxt` . **Note:** For practicability reasons, this is also simulation data. Same sorting as in exercise 03.1, simulation duration 10s.

2. Create a function `min_target(p)` which expects a parameter array as argument. You can assume that the array p has two elements. Inside `min_target(p)` create the local variables `m2` , `l` and assign them the contents of p.

3. Define the function `rhs(z, t)` analogous to part 1, but this time *inside of the function* `min_target` taking into account the parameter values for m_2 and l defined in the parent (outer) function.
   Note: Use nested namespaces, i.e., functions within function, see Course02.

4. For each call to `min_target` (i.e. within this function), run a simulation with the corresponding parameter values for m_2 and l. Select the initial values and sampling step size to match the measured data. Use the `rtol=1e-5` option for `solve_ivp` as in exercise 03.1.

5. Calculate a squared measure of the carriage position error (simulated values minus measured values, squared and then summed using `np.sum`) and return this measure as the result of the function `min_target` .

6. Apply `scipy.optimize.minimize` to `min_target` with `method="Nelder-Mead"` as an optional keyword argument to find the optimal values for m_2 and l. Inside `min_target` print status information or intermediate results to `min_target` if necessary. As starting estimate for p you can use `p0 = [0.5, 0.7]` .

   Note: The function `scipy.optimize.minimize` provides a unified interface to many different minimization algorithms with and without constraints, see docs. For the problem at hand, the so-called "downhill simplex method" of Nelder and Mead is more suitable than the default method ("BFGS").