ELEN90097 Workshop 1 – Modelling and Analysis for AI

This workshop is allocated 3 weeks.

Topics Covered

- Modelling engineering and physical systems using ODEs
- Solving ODEs numerically
- Identifying system parameters
- Simulation of systems
- Software systems, abstract machines: Finite State Machines and Finite Automata

Workflow and Assessment

This subject follows a problem- and project-oriented approach. This learning workflow focuses on solving practical (engineering) problems, which motivates acquiring theoretical (background) knowledge.

Objectives

- Use these problems as a motivation to learn the fundamentals covered in lectures.
- Learn how to formulate and solve problems in practice.
- Familiarise yourself with practical software tools.
- Connect theoretical and practical knowledge by doing it yourself.

The goal is to gain hands-on experience and learn by doing! Understand how theoretical knowledge discussed in lectures relates to practice. Develop motivation for gaining further theoretical and practical knowledge beyond the subject material.

> **Self-learning** is one of the most important skills you should acquire as a student. Today, self-learning is much easier than it used to be thanks to many online resources.

Assessment Process

1. Follow the procedures described below, perform the given tasks, and answer the workshop questions **in this Python/Jupyter notebook! The resulting notebook will be your Workshop Report!**
2. Submit the workshop report at the announced deadline
3. Demonstrators will conduct a brief oral quiz on your submitted report in the subsequent weeks.
4. Your workshop marks will be based on the report you will submit and oral quiz results (with a heavy weight on the oral component).

> **The goal is to learn**, NOT blindly follow the procedures in the fastest possible way! **Do not simply copy-paste answers (from the Internet, friends, etc.). You can and should use all available resources but only to improve your understanding. If you copy-paste, you may perform poorly on the oral quiz!**

Notes

- Use the table of contents feature of Jupyter Notebook for your convenience, see View menu above, Table of Contents. You can also turn on section numbers there if you wish.

Modelling Physical Systems using ODEs

ODE Basics and Reduction of Order

In engineering, we extensively use Ordinary Differential Equations (ODEs) to describe the evolution of a dynamical system. A *differential equation* is a relationship between a function, $f(x)$, its independent variable, x, and any number of its derivatives. The independent variable in dynamical systems is often time t. Note that this is a continuous(-time) model. If time (or independent variable) is discretised, we end up with a difference equation instead of a differential equation.

As a general example, consider

$$\frac{d^n f(x)}{dx^n} = F\left(x, f(x), \frac{df(x)}{dx}, \frac{d^2 f(x)}{dx^2}, \frac{d^3 f(x)}{dx^3}, \ldots, \frac{d^{n-1} f(x)}{dx^{n-1}}\right),$$

where F is an arbitrary function that incorporates one or all of the input arguments, and n is the *order* of the differential equation. This equation is said to be an n^{th} *order ODE*.

A *general solution* to a differential equation is a $g(x)$ that satisfies the differential equation. Although there are usually many solutions to a differential equation, they are still hard to find. For an ODE of order n, a *particular solution* is a $p(x)$ that satisfies the differential equation *and n* explicitly *known values* of the solution, or its derivatives, at certain points. A common set of known values for an ODE solution is the *initial value*. For an ODE of order n, the initial value is a known value for the 0^{th} to $(n-1)^{\text{th}}$ derivatives at $x = 0, f(0), f^{(1)}(0), f^{(2)}(0), \ldots, f^{(n-1)}(0)$. In some cases, the initial value is sufficient to find a unique particular solution. Finding a solution to an ODE given an initial value is called the *initial value problem*.

It is useful to convert higher-order ODEs to *first-order* ODEs, which is also known as *reducing the order* of the ODE to first order. To reduce the order of the ODE of a dynamical system, we define the *state* of the system as a function of time, which we can represent as a vector, $S(t)$. In general, the state of a system is a deep concept. In this case,

$$f^{(n)}(t) = F\left(t, f(t), f^{(1)}(t), f^{(2)}(t), f^{(3)}(t), \ldots, f^{(n-1)}(t)\right).$$

For initial value problems, it is useful to take the state to be

$$S(t) = \begin{bmatrix} f(t) \\ f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ \cdots \\ f^{(n-1)}(t) \end{bmatrix}.$$

Then, the derivative of the state is

$$\frac{dS(t)}{dt} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \cdots \\ f^{(n)}(t) \end{bmatrix} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \cdots \\ F\left(t, f(t), f^{(1)}(t), \ldots, f^{(n-1)}(t)\right) \end{bmatrix} = \begin{bmatrix} S_2(t) \\ S_3(t) \\ S_4(t) \\ S_5(t) \\ \cdots \\ F\left(t, S_1(t), S_2(t), \ldots, S_{n-1}(t)\right) \end{bmatrix},$$

where $S_i(t)$ is the i^{th} element of $S(t)$. With the state written in this way, $\frac{dS(t)}{dt}$ can be written using only $S(t)$ (i.e., no $f(t)$) or its derivatives. In particular, $\frac{dS(t)}{dt} = \mathcal{F}(t, S(t))$, where $\mathcal{F}$ is a function that appropriately assembles the vector describing the derivative of the state. This equation is in the form of a first-order differential equation in S. Essentially, what we have done is turn an n^{th} order ODE into n first order ODEs that are *coupled* together, meaning they share the same terms.

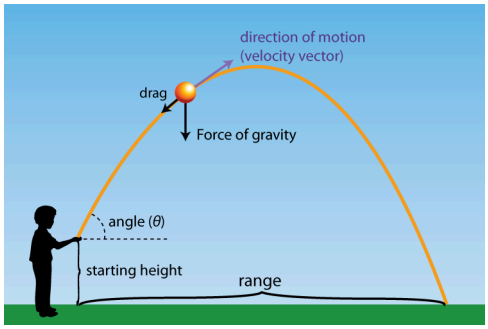
Note that several notations are commonly used for the derivative of $f(t)$ such as $f'(t)$, $f^{(1)}(t)$, $\frac{df(t)}{dt}$, and $\dot{f}$.

You can read further about this topic here.

In [3]:
```
import numpy as np
import scipy.integrate as spi
import matplotlib.pyplot as plt
%matplotlib inline
```

In [4]:
```
m = 1.  # particle's mass
k = 0.5  # drag coefficient
g = 9.81  # gravity acceleration

# The initial position is (0, 0).
v0 = np.zeros(4)
# The initial speed vector is oriented
# to the top right.
v0[2] = 4.
v0[3] = 8.
```

Example: Simple Projectile Motion



We horizontally throw a mass in a 2D setting where we only care about location x and height y. Let $u = (x, y)$ encode the 2D position of our particle with mass m. This particle is subject to two forces: gravity $mg = -9.81m$ and air drag $F = -ku'$. This last term is a simplifying assumption, which depends on the particle's speed and is only valid at low speeds.

Remember *Newton's second law of motion* from classical mechanics. This law states that, in an inertial reference frame, the mass multiplied by the acceleration of the particle is equal to the sum of all forces applied to that particle. You can show all forces acting on a particle in a free-body diagram. Here, we have:

$$m \cdot u'' = F + (0, 1)mg,$$

which is simplified for convenience, and immediately yields a second-order ODE:

$$u'' = -\frac{k}{m}u' - (0, 1)g.$$

Next, we reduce order and transform it into a single-order system of ODEs, with the state defined as $v = (u, u')$:

$$v' = (u', u'') = (u', [-\frac{k}{m}x', -\frac{k}{m}y' - g]).$$

The last term can be expressed as a function of the state v only.

You may have solved these types of simple ODEs by hand in mathematics subjects before. In most cases, however, the ODEs you will encounter in real life will not be solvable by hand! Then, you need to either simulate them or solve them numerically. Let's solve this ODE using the well-known Python library SciPy to simulate it numerically.

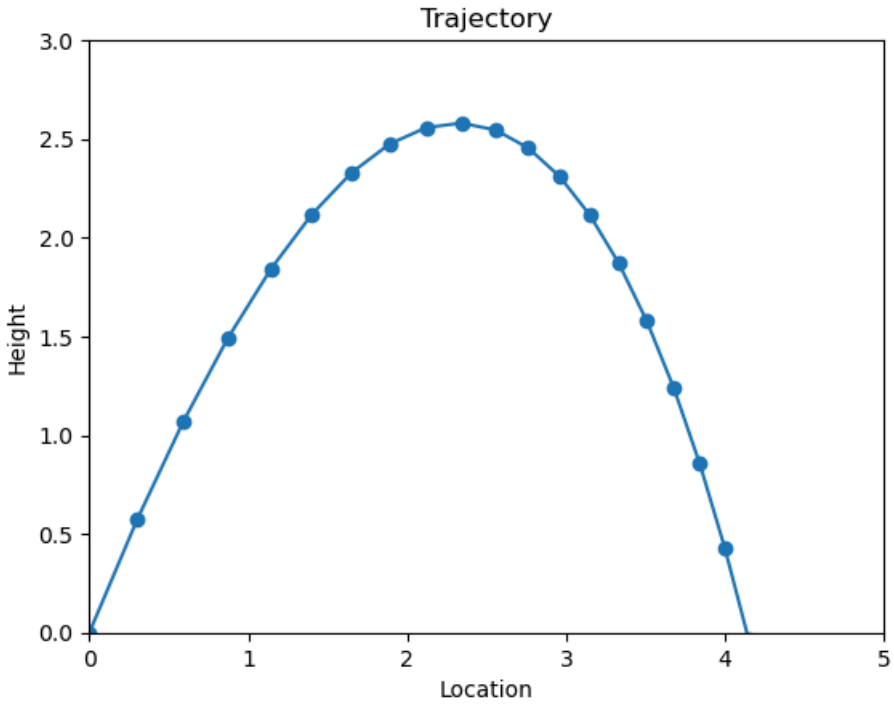
Note that there are many DE solvers. For example, diffeqpy is a nice project that uses Julia libraries (known for their DE-solving capabilities).

In [5]:
```
# the function to integrate
def f(t, v, k, m, g):
    # v has four components: v=[u, u']. Note u has two components itself, so 2x2=4!
    u, udot = v[:2], v[2:]
    # We compute the second derivative u'' of u in a fancy shortcut way
    udotdot = -k / m * udot   # air drag affects both x and y
    udotdot[1] -= g           # gravity only affects y in reverse direction
    # We return v'=[u', u''].
    return np.r_[udot, udotdot]  # concatenate, https://numpy.org/devdocs/reference/generated/numpy.r_.h
```

In [6]:
```
# We want to evaluate the system between t=0 and t=3.
tspan = [0, 3]
t_pts = np.linspace(tspan[0],tspan[1], 40)  # 40 points
```

```
#v = spi.odeint(f, v0, t, args=(k,))
solution = spi.solve_ivp(f, tspan, v0, args=(k, m, g), t_eval=t_pts, dense_output=True, method="RK45")
#solution.y

# We plot the particle's trajectory.
fig, ax = plt.subplots()
ax.plot(solution.y[0,:], solution.y[1,:], 'o-')
ax.set(xlabel='Location', ylabel='Height', title='Trajectory')
ax.set_xlim(0, 5)
ax.set_ylim(0, 3)
# fig.savefig("trajectory.png")  # you can save the figure if you wish
```

Out[6]: (0.0, 3.0)



In [7]:
```
solution
```

Out[7]:
```
  message: The solver successfully reached the end of the integration interval.
  success: True
   status: 0
        t: [ 0.000e+00  7.692e-02 ...  2.923e+00  3.000e+00]
        y: [[ 0.000e+00  3.019e-01 ...  6.145e+00  6.215e+00]
            [ 0.000e+00  5.750e-01 ... -1.492e+01 -1.595e+01]
            [ 4.000e+00  3.849e+00 ...  9.275e-01  8.925e-01]
            [ 8.000e+00  6.958e+00 ... -1.322e+01 -1.346e+01]]
      sol: <scipy.integrate._ivp.common.OdeSolution object at 0x0000020134355DB0>
 t_events: None
 y_events: None
     nfev: 44
     njev: 0
      nlu: 0
```

Let's save our results! There are many ways of doing it in Python, see how to save Numpy arrays to csv files. Python also has built-in support for csv files.

After saving the results, which are the state variables over time (4 variables, given time steps), to a universal csv (comma separated value) file, you can open it in Excel or import it to Matlab.

In [8]:
```
# Let's save the results

print(solution.y.shape)
# solnarray = np.array(solution.y) # solution.y is already a numpy array!
np.savetxt("projectile.csv", solution.y, delimiter = ",")
```

```
(4, 40)
```

Question: Solving ODEs using Trajector Example

Use the projectile motion above as an example to explore how to solve initial value problems in Python. Specifically,

1. Write the ODEs in matrix form by using state definition $v = [x, y, x', y']$.
2. Change parameters of the model (m, g, k) and observe their effects.
3. Change initial conditions (location, speed), i.e. state at time $t = 0$ and observe how solutions change.
4. Practice saving your solutions in CSV format.
5. Implement your simple solver using Euler's method.
6. Try different solvers including your own and parameters (e.g. number of evaluation points). Discuss (if you see) any differences.

Hint: do not try to be exhaustive and spend too much time on these tasks.

Answer as text here

In [9]:
```
''' Answer as code here '''
```

Out[9]:
```
' Answer as code here '
```

Question: Solve an ODE with step input

Consider this set of ODEs with the initial conditions and step input function given. Solve it numerically using scipy.integrate.solve_ivp as you have done before. Note that you may need to use a different approach here due to discontinuity in the function $u(t)$.

$$2\,\frac{dx(t)}{dt} = -x(t) + u(t)$$

$$5\,\frac{dy(t)}{dt} = -y(t) + x(t)$$

$$u = 2\,S(t-5)$$

$$x(0) = 0$$

$$y(0) = 0$$

where $S(t-5)$ is a step function that changes from zero to one at $t = 5$. When it is multiplied by two, it changes from zero to two at that same time, $t = 5$.

Hint: *the trick here is to define the step function u and include it in the system of ODEs.*

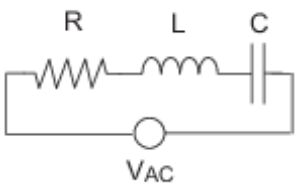
Further Exploration: You can use `u(t) = 2 * np.random.random_sample()` for more fun!

Answer as text here

In [10]:
```
''' Answer as code here '''
```

Out[10]:
```
' Answer as code here '
```

Question: Solve and Simulate a Simple RLC Circuit



You probably have seen this simple RLC circuit before. It can be solved using a circuit simulator such as SPICE. Alternatively, you can use Matlab/Simulink to model

and solve it. To save time, we will skip the underlying KVL/KCL (Kirchoff Voltage and Current Laws) and provide the reduced first-order set of ODEs below:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{V_C} \\ \dot{V_L} \\ \dot{V_R} \\ \dot{I_L} \\ \dot{I_{AC}} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \frac{1}{C} & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & R & 0 \\ 0 & \frac{1}{L} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} V_C \\ V_L \\ V_R \\ I_L \\ I_{AC} \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} V_{AC}$$

Here the state vector is $[V_C, V_L, V_R, I_L, I_{AC}]$. It is possible to use a more optimise it if you wish by removing I_{AC}.

1. Simulate this circuit in your favourite circuit simulator (or even in Matlab) after choosing **your own** reasonable values for R, L, C. Provide the current and voltages as outputs.
2. Solve the ODEs in Python numerically as above and check your results against the "ground truth" from your simulation in Step 1.
3. Try different solvers and check if you see any difference (should be similar).

Note that in many cases when real-world data is not available, system simulation provides synthetic but useful data, e.g. for machine learning.

Answer as text here

In [43]:
```
''' Answer as code here '''
```

Out[43]:
```
' Answer as code here '
```

System Identification using Least Squares Regression

Given a set of independent data points x_i and dependent data points $y_i, i = 1, \ldots, m$, we would like to find an *estimation function*, $\hat{y}(x)$, that describes the data as well as possible. In the least squares regression, the estimation function must be a linear combination of *basis functions*, $f_i(x)$. That is, the estimation function must be of the form $\hat{y}(x) = \sum_{i=1}^{n} \alpha_i f_i(x)$ The scalars α_i are the *parameters* of the estimation function, and each basis function must be linearly independent of the others.

The goal of **least squares regression** is to find the parameters of the estimation function that minimize the *total squared error*, E, defined by $E = \sum_{i=1}^{m} (\hat{y} - y_i)^2$. The *residuals* are defined as $e_i = (\hat{y} - y_i)$. If e is the vector containing all the individual errors, then we are also trying to minimize $E = \|e\|_2^2$, which is the L_2 norm.

If we assume **noisy measurements** at each data point, x_i, this gives us the following system of equations:

$$\begin{aligned} \hat{y}(x_1) &= \alpha_1 f_1(x_1) + \alpha_2 f_2(x_1) + \cdots + \alpha_n f_n(x_1), \\ \hat{y}(x_2) &= \alpha_1 f_1(x_2) + \alpha_2 f_2(x_2) + \cdots + \alpha_n f_n(x_2), \\ &\cdots \\ \hat{y}(x_m) &= \alpha_1 f_1(x_m) + \alpha_2 f_2(x_m) + \cdots + \alpha_n f_n(x_m). \end{aligned}$$

We can write the *total squared error* as

$$E = \sum_{i=1}^{m} e_i^2 = \sum_{i=1}^{m} (\hat{y}(x_i) - y_i)^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} \alpha_j f_j(x_i) - y_i \right)^2.$$

Minimising E with respect to parameters α (using basic convex optimisation), we obtain for each row k:

$$\sum_{j=1}^{n} \alpha_j \sum_{i=1}^{m} f_j(x_i) f_k(x_i) = \sum_{i=1}^{m} y_i f_k(x_i).$$

Rearranging the terms, we have

$$\sum_{i=1}^{m} \left(\sum_{j=1}^{n} \alpha_j f_j(x_i) \right) f_k(x_i) = \sum_{i=1}^{m} y_i f_k(x_i), \;\; k = 1, \ldots, m$$

Let us define the matrix

$$F = \begin{bmatrix} f_1(x_1), f_1(x_2), \ldots, f_1(x_m) \\ f_2(x_1), f_2(x_2), \ldots, f_2(x_m) \\ \ldots \ \ldots \\ f_m(x_1), f_m(x_2), \ldots, f_m(x_m) \end{bmatrix}$$

Then, the equations can be written for each row k as $F_{k,*}\, F_{*,k}\, \alpha = F_{k,*}\, y$. Thus, we obtain the matrix equation

$$FF^T \alpha = Fy.$$

Solving this matrix equation (either by inverting FF^T or posing it as an optimisation problem), we find the parameter vector α and achieve our goal!

You can read further in this nice chapter on Least Squares Regression using Python.

Example: Identifying Projectile Mass

Let us imagine (as a simple hypothetical example) that we have a measurement device that records the position and acceleration of our projectile in the projectile trajectory example above. If we could measure these perfectly and we know the air drag constant k, all we would need is a single perfect data point to find the mass m of the projectile based on the equation $[x'', y''] = -\frac{k}{m}[x', y'] - [0, 1]g$. However, in real life measurements are imperfect so our data points will be noisy. Because our data comes from a ``perfect" model, we add a bit of noise to represent measurement errors or imprecision. Let's try!

In [44]:
```
from numpy.random import default_rng
from scipy.optimize import least_squares

rng = default_rng(65988165)
error = 0.1 * rng.standard_normal((4,40))

# generate imperfect measurements from previous solution
vx = solution.y[2]+error[0,:]
vy = solution.y[3]+error[1,:]
ax = -(k/m)*vx + +error[2,:]
ay = -(k/m)*vy +error[3,:] # that includes the constant g as simplification

# Initial parameter estimate
km0 = np.array([2])

#print(vx.shape, vy.shape, ax.shape, ay.shape, km0.shape)
```

In [45]:
```
# We define a function for computing residuals and the initial estimate of parameter k/m

def func(km, vx, vy, ax, ay):
    return (ax + km[0]*vx)**2+(ay + km[0]*vy)**2

#a = func(km0, vx, vy, ax, ay)
#a
est_lsq = least_squares(func, km0, args=(vx, vy, ax, ay))
est_lsq
```

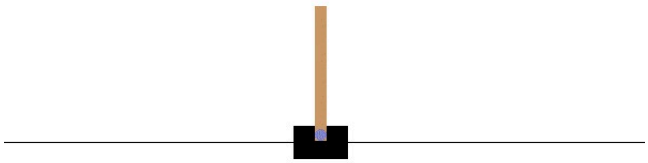
Out[45]:
```
      message: `ftol` termination condition is satisfied.
      success: True
       status: 2
          fun: [ 7.268e-03  9.223e-03 ...  4.615e-03  1.596e-02]
            x: [ 5.017e-01]
         cost: 0.012845606699258518
          jac: [[-1.275e+00]
                [-2.782e-01]
                ...
                [-1.266e+00]
                [-3.225e+00]]
         grad: [-2.356e-04]
  optimality: 0.00023561290220602826
 active_mask: [ 0.000e+00]
        nfev: 33
        njev: 33
```

In [46]:
```
print('estimated value is',est_lsq.x)
print('real value is',k/m)
```

```
print('estimation error due to noise',est_lsq.x-k/m)
```

```
estimated value is [0.50168501]
real value is 0.5
estimation error due to noise [0.00168501]
```

Example: Identify Parameters of a Pole-Cart Object



In this example, we will use MuJoCo System Identification (mujoco-sysid) Python module that is designed to perform system identification using the MuJoCo physics engine.

There is extensive documentation on MuJoCo, which comes with native Python bindings you can read about. You can also find a nice tutorial on there. Here is another nice notebook on least squares.

First, let's have a look at the MuJoCo implementation of the famous cart-pole model.

In [47]:
```
import mujoco
sim_model = mujoco.MjModel.from_xml_path("./files/cartmodel.xml")
sim_data = mujoco.MjData(sim_model)
```

In [48]:
```
# You can edit model properties directly on the XML file or access them via Python interface

# This is a trick to list objects in the model
try:
  sim_model.body()
except KeyError as e:
  print(e)
```

```
"Invalid name ''. Valid names: ['cart', 'pole', 'world']"
```

In [49]:
```
# let's look at the properties of the cart
sim_model.body("cart")
```

Out[49]:
```
<_MjModelBodyViews
  dofadr: array([0])
  dofnum: array([1])
  geomadr: array([0])
  geomnum: array([1])
  id: 1
  inertia: array([1.e-06, 1.e-06, 1.e-06])
  invweight0: array([0.66666133, 0.        ])
  ipos: array([0., 0., 0.])
  iquat: array([1., 0., 0., 0.])
  jntadr: array([0])
  jntnum: array([1])
  mass: array([0.5])
  mocapid: array([-1])
  name: 'cart'
  parentid: array([0])
  pos: array([0., 0., 1.])
  quat: array([1., 0., 0., 0.])
  rootid: array([1])
  sameframe: array([1], dtype=uint8)
  simple: array([0], dtype=uint8)
  subtreemass: array([0.6])
  user: array([], dtype=float64)
  weldid: array([1])
>
```

In [50]:
```
# we can change the properties of the cart, for example mass as below
sim_model.body("cart").mass=np.array([1.0])
```

Now, we gather data by randomly applying force to the cart. This type of "random walk" is useful in system identification as it allows us to explore the trajectory space randomly (instead of sticking to a subset of the environment).

In [51]:
```
# random forcing and measure data

# Reset data, set initial pose.
mujoco.mj_resetData(sim_model, sim_data)

identification_data = {
    "qhist": [],
    "vhist": [],
    "dvhist": [],
    "uhist": [],
}

for i in range(3000):
    identification_data["qhist"].append(sim_data.qpos.copy())
    identification_data["vhist"].append(sim_data.qvel.copy())

    u = np.random.randn(1) * 10
    identification_data["uhist"].append(u)

    sim_data.ctrl = u
    mujoco.mj_step(sim_model, sim_data)
    identification_data["dvhist"].append(sim_data.qacc.copy())

identification_data["qhist"] = np.array(identification_data["qhist"])
identification_data["vhist"] = np.array(identification_data["vhist"])
identification_data["dvhist"] = np.array(identification_data["dvhist"])
identification_data["uhist"] = np.array(identification_data["uhist"])
```

Cart Pole Model Equations and Parameters

Once we have gathered data, we can formulate the model-based system identification problem. A specific model of the cart pole system dynamics is given by:

$$\begin{cases} (M+m)\,\ddot{q}_1 - mL\ddot{q}_2\cos q_2 + mL\dot{q}_2^2\sin q_2 = u \\ L\ddot{q}_2 - g\sin q_2 = \ddot{q}_1\cos q_2 \end{cases} \tag{1}$$

where q_2 is the angle of the pendulum measured from the upper equilibrium and q_1 is the cart's position.

The notable property of any rigid body mechanical system (cart-pole in particular) is the inverse dynamics can be parametrised linearly to the set of constant inertial parameters, indeed one may introduce the parameters $\boldsymbol{\theta} = [M + m, mL, L]$ and derive linearly parametrised inverse dynamics in so-called regressor form:

$$\begin{bmatrix} \ddot{q}_1 & -\ddot{q}_2 \cos\theta + \dot{q}_2^2 \sin q_2 & 0 \\ 0 & 0 & \ddot{q}_2 \end{bmatrix} \begin{bmatrix} M+m \\ mL \\ L \end{bmatrix} = \begin{bmatrix} u \\ g \sin q_2 + \ddot{q}_1 \cos q_2 \end{bmatrix}$$

For given measurements of $\mathbf{q}(t), \dot{\mathbf{q}}(t), \ddot{\mathbf{q}}(t), \mathbf{u}(t)$ the above is the formulated as simple linear equation:

$$\mathbf{A}(t)\boldsymbol{\theta} = \mathbf{b}(t)$$

Now given the sequence of time samples (measured data above), we may build the $\mathbf{A}, \mathbf{b}$ and solve the associated linear least squares problem (possibly with bounds on parameters).

In [52]:
```python
# formulate regressor
A = []
b = []

g = 9.81

for i in range(1, len(identification_data["qhist"])):
    q = identification_data["qhist"][i]
    v = identification_data["vhist"][i]
    dv = identification_data["dvhist"][i]
    u = identification_data["uhist"][i]

    regressor = [
        [dv[0], -dv[1] * np.cos(q[1]) + v[1] ** 2 * np.sin(q[1]), 0],
        [0, 0, dv[1]],
    ]

    A.extend(regressor)
    b.extend(
        [
            u[0],
            dv[0] * np.cos(q[1]) + g * np.sin(q[1]),
        ]
    )

A = np.array(A)
b = np.array(b)
```

Now, we solve the simple least squares problem to get the parameter estimates using MuJoCo's built-in least-squares implementation.

In [53]:
```python
import mujoco.minimize as minimize


def residual(x):
    # Ensure x is 2D for consistent processing
    if x.ndim == 1:
        x = x[:, np.newaxis]

    # Compute residuals using matrix operations
    return A @ x - b[:, np.newaxis]


result = minimize.least_squares(
    np.array((0.0, 0.0, 0.0)),
    residual,
)

identified_parameters = result[0]

newL = identified_parameters[2]
new_poleM = identified_parameters[1] / newL
new_cartM = identified_parameters[0] - new_poleM

identified_parameters
```

```
iter: 0    y: 2.985e+05  log10mu: -inf  ratio: 1        dx: 1.2      reduction: 2.9e+05
iter: 1    y: 1.346e+04  log10mu: -inf  ratio: 7.9e+06  dx: 6.3e-13  reduction: 1.8e-12
Terminated after 1 iterations: norm(dx) < tol. y: 1.346e+04, Residual evals: 9
total time 4.0ms of which residual 50.0%
```

Out[53]: `array([1.1       , 0.05      , 0.47785148])`

In [54]:
```
print("Total mass M+m = ",identified_parameters[0])
print("Cart mass M =", new_cartM)
print("Pole mass m =", new_poleM)
print("Pole length L =",newL)   # in xml file geom size 0.45
```

```
Total mass M+m =  1.0999999999996952
Cart mass M = 0.9953649775385469
Pole mass m = 0.10463502246114836
Pole length L = 0.47785147672507017
```

You can rerun this after changing some parameters such as cart mass or pole mass/length. **Try it yourself!**

Question: System Identification Mini Project

Now that you have seen system identification examples and libraries for deploying the well-known least square method, it is time to try things yourself!

The tasks are:

1. Choose **one** dynamical system with a known model. It can be a simplified model, e.g. linearised. (*Note: derivation of a new model is outside scope!*)
2. Generate an observation dataset from the dynamical system recording relevant variable values for a given set of fixed system parameters. This can be done either by (a) solving the ODEs of the model as above, (b) using a simulator, or (c) an environment like Gymnasium or MuJoCo.
3. Choose one or more system parameters and assume they are unknown. Then, use the **Least Squares method** to identify these parameters using the library of your choice, e.g. SciPy.
4. Compute the error between actual and estimated values, e.g. using MSE or MAE (Mean Square or Absolute Error).

Possible Dynamical Systems from different domains

- *[Mechanical]* Pendulum (possibly with friction and damping). There are multiple resources online such as relevant equations and background/II%3A_Dynamical_Systems_and_Chaos/11%3A_The_Damped%2C_Driven_Pendulum), a tutorial, an example project and a gym environment.
- *[Electrical/Energy]* Resistor-capacitor (RC) model of buildings from an energy perspective.
- *[Electrical]* Identification of an element, e.g. resistance in an (RLC) circuit from measurements. You can use the RLC circuit model above as a starting point.
- *[Robotics/Mechanical]* There are many options in Gymnasium classical control and MuJoCo Menagerie. You can choose any of these (as long as it is simple enough and you find the ODE model).
- *[Other]* If you have expertise in another field (e.g. Chemistry) and have an ODE model of a dynamical system, you can use it.

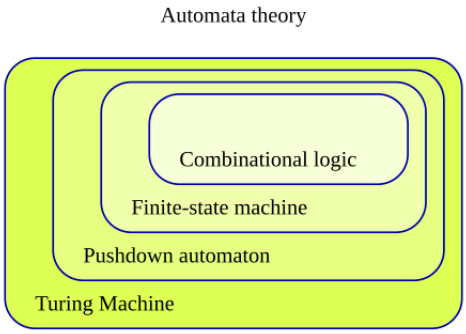
Requirement: Please first check your choice of dynamical system with your demonstrator and lecturer!

Answer as text here

In [55]:
```
''' Answer as code here '''
```

Out[55]: `' Answer as code here '`

Abstract Computing Machines



In the computing world, an **abstract machine** is a theoretical construct and an umbrella model for computing systems. Abstract machines are mathematical objects, so they are expected to perform correctly and independent of hardware. Abstract machines are "machines" because they allow step-by-step execution of programmes; they are "abstract" because they ignore many aspects of actual physical hardware. Therefore, they are fundamental to the field of computational complexity theory, such as finite state machines, automata, and Turing machines.

It is important to note a crucially important difference between abstract machines of the computing world and physical systems. Abstract machines are (up to some extent) not bound by the rules of physics. Therefore, they are different from mechanical, chemical, or electrical engineering systems. Maybe unsurprisingly, they are not easy to model using ODEs which work very well in modelling physical phenomena.

Example: Finite State Machine (FSM)

Let us make things a bit more concrete by looking at deterministic finite automata, which you may already know well as **Finite State Machine (FSM)**.

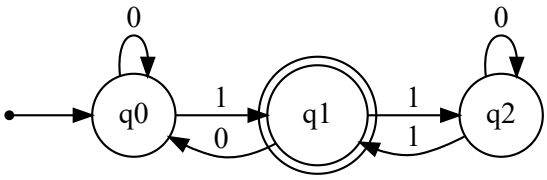
FSMs are widely used from communication protocols to industrial automation. Here are a few examples from language processing, embedded systems, and other applications. Here is another good free book chapter on this topic.

There are many FSM implementations in Python. We choose automata for its simplicity. Very useful information can be found at the homepage of the book Algorithms by Jeff Erickson.

In [56]:
```
from automata.fa.dfa import DFA

# DFA which matches all binary strings ending in an odd number of '1's
my_dfa = DFA(
    states={'q0', 'q1', 'q2'},
    input_symbols={'0', '1'},
    transitions={
        'q0': {'0': 'q0', '1': 'q1'},
        'q1': {'0': 'q0', '1': 'q2'},
        'q2': {'0': 'q2', '1': 'q1'}
    },
    initial_state='q0',
    final_states={'q1'}
)

# a helper function that takes in an automaton and input strings, printing whether the input was accepte
def read_user_input(my_automaton, input_sequence):
    if my_automaton.accepts_input(input_sequence):
        print("Accepted")
    else:
        print("Rejected")
```

In [57]:
```
# The visualisation is unimportant, don't worry if you cannot make it work due to package problems!

!pip install 'automata-lib[visual]'
# let's visualise this FSM
my_dfa.show_diagram()
```

Out[57]:



A sequence is accepted by the FSM (DFA) if it is a valid input sequence and (starting from the initial state) it ends in a legitimate final state.

In [58]:
```
read_user_input(my_dfa,"0111101")
```

```
Accepted
```

In [59]:
```
read_user_input(my_dfa,"1101010")
```

```
Rejected
```

Modify this FSM to your liking and try different inputs!

Example: Cellular Automata

Cellular automata are discrete dynamical systems evolving on a grid of cells. These cells can be in a finite number of states (for example, on/off). The evolution of a cellular automaton is governed by a set of rules, describing how the state of a cell changes according to the state of its neighbours.

Although extremely simple, these models can initiate highly complex and chaotic behaviours. Cellular automata can model real-world phenomena such as car traffic, chemical reactions, propagation of fire in a forest, epidemic propagation, and much more. Cellular automata are also found in nature. For example, the patterns of some seashells are generated by natural cellular automata.

An **elementary cellular automaton** is a binary, one-dimensional automaton, where the rules concern the immediate left and right neighbours of every cell. The Wikipedia article provides a very good overview. Famous Stephen Wolfram (founder of Mathematica) worked on this topic and created the Wolfram Code that efficiently encodes the rules.

Let's simulate this specific (binary, one-dimensional) automaton following Chapter 12 of IPython Cookbook.

In [60]:
```
u = np.array([[4], [2], [1]]) # vector to represent binary 3 bits
```

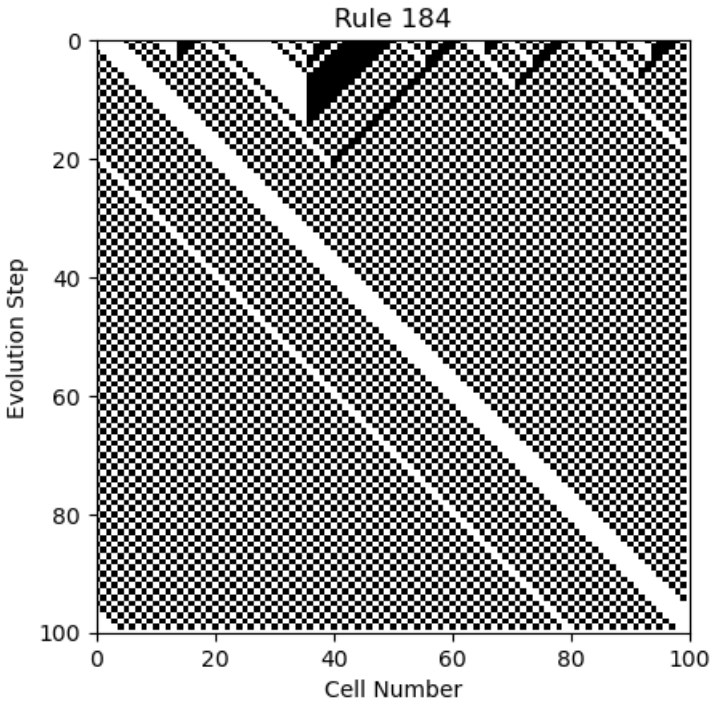
In [61]:
```
def step(x, rule_b):
    """Compute a single step of an elementary cellular
    automaton."""
    # The columns contain the L, C, and R values
    # of all cells.
    y = np.vstack((np.roll(x, 1), x,
                   np.roll(x, -1))).astype(np.int8)  # shift array elements, then stack them vertically
    # We get the LCR pattern numbers between 0 and 7.
    z = np.sum(y * u, axis=0).astype(np.int8)
    # We get the patterns given by the rule.
    return rule_b[7 - z]
```

In [62]:
```
def generate(rule, initial_state, size, steps):
    """Simulate an elementary cellular automaton given
    its rule (number between 0 and 255)."""
```

```
    # Compute the binary representation of the rule.
    rule_b = np.array(
        [int(_) for _ in np.binary_repr(rule, 8)],
        dtype=np.int8)
    x = np.zeros((steps, size), dtype=np.int8)
    # Random initial state.
    x[0, :] = initial_state
    # Apply the step function iteratively.
    for i in range(steps - 1):
        x[i + 1, :] = step(x[i, :], rule_b)
    return x
```

In [64]:
```
# fig, axes = plt.subplots(3, 3, figsize=(8, 8))
# rules = [3, 18, 30,
#          90, 106, 110,
#          158, 154, 184]
# for ax, rule in zip(axes.flat, rules):
#     x = generate(rule)
#     ax.imshow(x, interpolation='none',
#               cmap=plt.cm.binary)
#     ax.set_axis_off()
#     ax.set_title(str(rule))


# which rule to use
rule = 184  # try also 110
cell_array_size = 100
nbr_evolve_steps = 100
initial_state = np.zeros(cell_array_size)
initial_state = np.random.rand(cell_array_size) < .5  # random initial state
# initial_state[99]= 1  # just a single cell as starting state (comment above)
x = generate(rule, initial_state, cell_array_size, nbr_evolve_steps)

# We plot the particle's trajectory.
fig, ax = plt.subplots()
ax.imshow(x, interpolation='none', cmap=plt.cm.binary)
ax.set(xlabel='Cell Number', ylabel='Evolution Step', title="Rule "+str(rule))
ax.set_xlim(0, cell_array_size)
ax.set_ylim(nbr_evolve_steps, 0)
# fig.savefig("cellular_automata-"+str(rule)+".png")  # you can save the figure if you wish
```

Out[64]: (100.0, 0.0)



A special case of note is Rule 110, which has the perhaps surprising property that it is Turing complete, and hence capable of universal computation! **You should run it to see what it looks like.**

Another famous automaton with this property is **Game of Life** which is a two-dimensional cellular automaton. You can even code it yourself if you wish (as a hobby). People even built an entire digital processor that can run Tetris game out of this!

Question: FSM Mini Project

In this project, you will choose **one** simple problem and implement an FSM that solves it. Specifically,

1. Briefly describe the problem.
2. Implement the FSM(s) using the automaton library (used in the example above).
3. Visualise the FSM(s).
4. Test it with multiple input strings.

Here are possible problems to work on:

- A Lift Controller as described here.
- UART serial communication protocol.
- Any other simple problem you are interested in and that can be modelled using FSMs.

Answer as text here

In [65]:
```
''' Answer as code here '''
```

Out[65]:
```
' Answer as code here '
```

Turing Machines

A Turing machine is a mathematical model of computation describing an abstract machine. Although the simple machine described ``only" manipulates symbols on a strip of tape according to a table of rules, it is capable of implementing any computer algorithm (not easy but possible).

Since the Turing machine is essentially a mathematical model, it is not practical at all. Still, people have developed various implementations that visualise it.

In [ ]:

In [ ]:

In [ ]:

Workshop Assessment Instructions

You should complete the workshop tasks and answer the questions within the allocated session! **Submission deadline is usually the end of the last week of the workshop. Please check Canvas for the exact deadline!**

It is **mandatory to follow all of the submission guidelines** given below. *Don't forget the Report submission information on top of this notebook!*

1. The completed Jupyter Notebook and its PDF version (you can simply print-preview and then print as PDF from within your browser) should be uploaded to the right place in Canvas. *It is your responsibility to follow the announcements!* **Late submissions will be penalised (up to 100% of the total mark depending on the delay amount)!**
2. Filename should be "ELEN90088 Workshop **W: StudentID1-StudentID2** of session **Day-Time**", where **W** refers to the workshop number, **StudentID1-StudentID2** are your student numbers, **Day-Time** is your session day and time, e.g. *Tue-14*.
3. Answers to questions, simulation results and diagrams should be included in the Jupyter Notebook as text, code, and plots. *If you don't know latex, you can write formulas/text to a paper by hand, scan it and then include it as an image within Markdown cells.*
4. Please submit your report as a group.

Workshop Marking

- Each workshop has a fixed number of points corresponding to a certain percentage of the total subject mark inclusive individual oral examination. You can find the detailed rubrics on Canvas.
- Individual oral quizzes will be scheduled within the next two weeks following the report submission. They will be during workshop hours. Therefore, you must attend the workshops!
- The individual oral examination will assess your answers to workshop questions, what you have done in that workshop, and your knowledge of the subject material in association with the workshop.

Additional guidelines for your programs:

- Write modular code using functions.
- Properly indent your code. But Python forces you to do that anyway ;)
- Heavily comment on the code to describe your implementation and to show your understanding. No comments, no credit!
- Make the code your own! It is encouraged to find and get inspired by online examples but you should exactly understand, modify as needed, and explain your code via comments. If you resort to blind copy/paste, you will certainly not do well in the individual oral quizzes.

In [ ]:

In [ ]: