Diff:

Differences between given skeleton and solution
In order to make the sample solution easier to understand, the differences between it and the given skeleton source code were highlighted with the help of the program `diff`.

Legend:

- Gray: unchanged text (only excerpts).
- Green: new lines
- Yellow: changed lines
- Red: deleted lines

Note: Files not listed have not been changed.

This document was created with the help of diff2html erstellt.

diff -u ../course03-numerical-computation/exercise/code/01_simulation.py ../course03-numerical-computation/exercise/solution/01_simulation.py

../course03-numerical-computation/exercise/code/01_simulation.py

```
⁝
import sys
import numpy as np
from numpy import r_, pi
from matplotlib import pyplot as plt  # used for the plotting at the end


# Move the following line further down as you are advancing.
# Background: actively exiting the program here prevents errors,
# due to usage of undefined names like `XXX`.


sys.exit() # Ends the program here. Otherwise: error messages




# Task 1:

# import the function solve_ivp from the package scipy.integrate
from scipy.XXX import XXX

# import two functions for calculating the accelerations
# (look inside the file `equations_of_motion.py`!)
from equations_of_motion import xdd_fnc #, XXX


# Task 2:

def rhs(XXX, XXX):
   # This function calculates the time derivative z_dot from the state z
   # the 1st argument (the time t) is not needed here

   x, phi, xd, phid = z # unpacking (see overview slides in course01)
   F = 0

   xdd = xdd_fnc(XXX, XXX, ...)
   phidd = XXX


   # Return the derivative of the state vector
   z_dot = r_[xd, phid, XXX, XXX]
   return z_dot


⁝

# Task 3:

zz0 = np.array([XXX,  pi*0.5, XXX, XXX])


# Task 4:

# do the numerical integration
res = solve_ivp(XXX, (tt[0], tt[-1]), XXX, t_eval=tt, rtol=1e-5)
```

../course03-numerical-computation/exercise/solution/01_simulation.py

```
⁝
import sys
import numpy as np
from numpy import r_, pi
from matplotlib import pyplot as plt # used for the plotting at the end


# Task 1:

# import the function solve_ivp from the package scipy.integrate
from scipy.integrate import solve_ivp

# import two functions for calculating the accelerations
# (look inside the file `equations_of_motion.py`!)
from equations_of_motion import xdd_fnc, phidd_fnc


# Task 2:

def rhs(t, z):
   # This function calculates the time derivative z_dot from the state z
   # the 1st argument (the time t) is not needed here

   x, phi, xd, phid = z # unpacking (see overview slides in course01)
   F = 0

   xdd = xdd_fnc(x, phi, xd, phid, F)
   phidd = phidd_fnc(x, phi, xd, phid, F)

   # Return the derivative of the state vector
   z_dot = r_[xd, phid, xdd, phidd]
   return z_dot


⁝

# Task 3:

zz0 = np.array([0,  pi*0.5, 0, 0])


# Task 4:

# do the numerical integration
res = solve_ivp(rhs, (tt[0], tt[-1]), zz0, t_eval=tt, rtol=1e-5)
```

```
# res: result container
# res.y: result array with shape (4, 1001)
# rows -> state components, columns -> time steps.


# Task 5:

# Unpacking of individual state components.
# Arrays are always unpacked along the 1st axis (rows).
x, phi, xd, phid = XXX

# visualization (more on this in course04):
plt.plot(tt, x)
plt.plot(XYZ)
plt.show()
```

```
# res: result container
# res.y: result array with shape (4, 1001)
# rows -> state components, columns -> time steps.

# Task 5:

# Unpacking of individual state components.
# Arrays are always unpacked along the 1st axis (rows).
x, phi, xd, phid = res.y


# visualization (more on this in course04):
plt.plot(tt, x)
plt.plot(tt, phi)
plt.show()


## The following code is not part of the exercise03.1 but conveniently stored here
## by the supervisor. It is used to generate "pseudo-measurement data"
## for exercise03.2.
## The block will not be executed, but can be quickly converted to "active code"
## by replacing `0` with `1` in the if-statement.

if 0:
   # binary format:
   np.save('measurement-data.npy', res.y)
   # text fomat (human readable, but needs more memory):
   np.savetxt('measurement-data.txt', res.y)

   print ("Files written.")
```

diff -u ../course03-numerical-computation/exercise/code/02_identification.py ../course03-numerical-computation/exercise/solution/02_identification.py

../course03-numerical-computation/exercise/code/02_identification.py | **../course03-numerical-computation/exercise/solution/02_identification.py**

```
⋮

# Move the following line further down as you are advancing.
# Background: actively exiting the program here prevents errors,
# due to usage of undefined names like `XXX`.


sys.exit() # Ends the program here. Otherwise: error messages








import sys
import numpy as np

⋮
# Task 1:

# import the function solve_ivp from the package scipy.integrate


```

```
⋮
import sys
import numpy as np

⋮
# Task 1:

# import the function solve_ivp from the package scipy.integrate
from scipy.integrate import solve_ivp

```

```python
# import two functions for calculating the accelerations
# (look inside the file `equations_of_motion.py`!)
from equations_of_motion import xdd_fnc, XXX


# load pseudo measurement data in binary format
zz_res_target = np.load(XXX)

# -> this is a 2d array with shape = (4, 1001), i.e. 4 rows, 1001 cols.
# meaning of rows: x, phi, xd, phid
# meaning of cols: time instant


## alternatively: load data in text format:
# zz_res_target = np.loadtxt(XXX)


# Task 2:
⋮
    :returns:   err – non-negative real valued error measure
                (how "wrong the simulation result is")
    """

    m2, l = XXX # unpacking the parameter vector

    # Task 3:

⋮
        Righthand side of the equations of motion
        (Note: this depends on m2 and l from the surrounding namespace).
        """
        x, phi, xd, phid = XXX # unpacking
        F = 0

        # m2 and l come from the namespace one level higher
        # (you might want to look again into `eqations_of_motion.py`
        # to check the signature of these functions:)
        xdd = xdd_fnc(XXXXXXXX, m2, l)
        phidd = phidd_fnc(XXXXXXXXX,  XXX, XXX)

        # return derivative of the state vector
        return XXX

    # end of the inner function definition of rhs

⋮
    # them here:

    # array with evaluation times (should be consistent with the measured data)
    tt = np.linspace(0, 10, XXX)

    # select a consistent initial state (4 values) for the simulation
    # from the measurement data (-> choose the first column)
    zz0 = XXX

    # do the simulation (get result container)
    sim_res = solve_ivp(XXX, (XXXX, XX), YYY, t_eval=tt, rtol=1e-5)

```

```python
# import two functions for calculating the accelerations
# (look inside the file `equations_of_motion.py`!)
from equations_of_motion import xdd_fnc, phidd_fnc

# load pseudo measurement data in binary format
zz_res_target = np.load('measurement-data.npy')

# -> this is a 2d array with shape = (4, 1001), i.e. 4 rows, 1001 cols.
# meaning of rows: x, phi, xd, phid
# meaning of cols: time instant

## alternatively: load data in text format:
# zz_res_target = np.loadtxt('measurement-data.txt')


# Task 2:
⋮
    :returns:   err – non-negative real valued error measure
                (how "wrong the simulation result is")
    """
    m2, l = p # unpacking the parameter vector

    # Task 3:

⋮
        Righthand side of the equations of motion
        (Note: this depends on m2 and l from the surrounding namespace).
        """
        x, phi, xd, phid = z # unpacking
        F = 0

        # m2 and l come from the namespace one level higher
        # (you might want to look again into `eqations_of_motion.py`
        # to check the signature of these functions:)
        xdd = xdd_fnc(x, phi, xd, phid, F, m2, l)
        phidd = phidd_fnc(x, phi, xd, phid, F, m2, l)

        # return derivative of the state vector
        return np.array([xd, phid, xdd, phidd])

    # end of the inner function definition of rhs

⋮
    # them here:

    # array with evaluation times (should be consistent with the measured data)
    tt = np.linspace(0, 10, 1001)

    # select a consistent initial state (4 values) for the simulation
    # from the measurement data (-> choose the first column)
    zz0 = zz_res_target[:, 0]

    # do the simulation (get result container)
    sim_res = solve_ivp(rhs, (tt[0], tt[-1]), zz0, t_eval=tt, rtol=1e-5)

```

```python
    # select the state vector (which we call "z" but scipy calls "y")
    zz_res = XXX.y

    # Task 5:

    # Calculate the difference of the x-positions (first line in each case)
    # then square (...**2),
    # then add up (applying np.sum)
    err = np.sum( (XXX[YYY, ZZZ] - XXX[YYY, ZZZ])**XXX )

    # Status message and output (to assess progress of optimization)
    print("simulation ready. p =", p, " equation error:", err)
⋮

# Task 6:

p0 = np.array([.5, .7]) # Startschätzung für m2 und l

# import the function minimize from the module scipy.optimize
from XXX.XXX import XXX

# do the optimization (call the algorithm, which internally repeatedly calls min_target)
min_res = minimize(XXX, XXX, method="Nelder-Mead")

print("\n", "minimization result (data structure):", min_res, "\n")
print("estimated paremeters (m2, l):", min_res.x, "\n")
```

```python
    # select the state vector (which we call "z" but scipy calls "y")
    zz_res = sim_res.y

    # Task 5:

    # Calculate the difference of the x-positions (first line in each case)
    # then square (...**2),
    # then add up (applying np.sum)
    err = np.sum( (zz_res[0, :] - zz_res_target[0, :])**2 )

    # Status message and output (to assess progress of optimization)
    print("simulation ready. p =", p, " equation error:", err)
⋮

# Task 6:

p0 = np.array([.5, .7]) # initial guess for m2 and l

# import the function minimize from the module scipy.optimize
from scipy.optimize import minimize

# do the optimization (call the algorithm, which internally repeatedly calls min_target)
min_res = minimize(min_target, p0, method="Nelder-Mead")


print("\n", "minimization result (data structure):", min_res, "\n")
print("estimated paremeters (m2, l):", min_res.x, "\n")
```