Gezeitenreibung - Computational Physics - Documentation

Bennet Weiss, Nico Alt

Contents

Module `gezeiten`

This module contains all the business logic of the project “Gezeitenreibung”, created during the lecture “Computational Physics” at TU Darmstadt in 2020.

Various gezeiten.differential_equations may be solved by using a Solver from gezeiten.solvers.

Sub-modules

- gezeiten.constants
- gezeiten.differential_equation
- gezeiten.differential_equations
- gezeiten.exercises
- gezeiten.solver
- gezeiten.solvers

Module `gezeiten.constants`

This file contains various (mostly natural) constants used all around this project.

Variables

Variable `DEFAULT_PLOT_TITLE`

Default title used for plots and animations

Variable `DEFAULT_TIME_BOUNDARIES`

Default time boundaries used for solving a DifferentialEquation

Variable `G`

Gravitational constant

Variable `T_E`

Time of Earth’s intrinsic rotation

Variable `T_M`

Time for moon to do a full turn around Earth

Variable `k`

Constant of friction in complex 4 body problem

Variable `m_E`

Mass of Earth

Variable `m_M`

Mass of Moon

Variable `m_O`

Mass of Oceans

Variable `r_C`

Distance of center of mass of earth-moon system, from Earth

Variable `r_E`

Radius of Earth

Variable `r_M`

Radius of Moon's orbit around Earth

Variable `tau`

Amount of Earth day length's increment in 100 years

Module `gezeiten.differential_equation`

Contains base class used for differential equations.

Classes

Class `DifferentialEquation`

```
class DifferentialEquation(
    differential_equation_function,
    initial_conditions,
    time_boundaries,
    data_points_amount
)
```

Base class used for differential equations.

Attributes

`data_points_amount` : int Amount of points created by gezeiten.solvers.

`differential_equation_function` : callable Right-hand side of the system. The calling signature is fun(t, y). Here t is a scalar, and there are two options for the ndarray y: It can either have shape (n,); then fun must return array_like with shape (n,). Alternatively, it can have shape (n, k); then fun must return an array_like with shape (n, k), i.e., each column corresponds to a single column in y. The choice between the two options is determined by vectorized argument (see below). The vectorized implementation allows a faster approximation of the Jacobian by finite differences (required for stiff solvers).

(Copied from scipy.integrate.solve_ivp's docs) `initial_conditions` : **array_like, shape (n,)** Initial state. For problems in the complex domain, pass y0 with a complex data type (even if the initial value is purely real).

(Copied from scipy.integrate.solve_ivp's docs)

`time_boundaries` : `2-tuple` **of floats** Interval of integration (t0, tf). The solver starts with t=t0 and integrates until it reaches t=tf.

(Copied from scipy.integrate.solve_ivp's docs)

`solution` : `array_like, shape (n + 1,)` Array containing the solution computed by Solver.

Descendants

- gezeiten.differential_equations.two_body_problem.TwoBodyProblem

Class variables

Variable `data_points_amount`

Variable `differential_equation_function`

Variable `initial_conditions`

Variable `solution`

Variable `time_boundaries`

Methods

Method `plot`

```
def plot(
    self
)
```

Plots the differential equation.

Method `solve`

```
def solve(
    self,
    solver=<gezeiten.solvers.magic_solver.MagicSolver object>
)
```

Solves the differential equation by using the solver passed as an argument.

Once finished, the differential equation object will have a solution field which contains the solution.

Attributes

`solver` : **Solver** Solver which solves the differential equation; by default MagicSolver

Module `gezeiten.differential_equations`

This module contains all DifferentialEquations that can be solved by a Solver.

Sub-modules

- gezeiten.differential_equations.four_body_problem_complex
- gezeiten.differential_equations.four_body_problem_simple
- gezeiten.differential_equations.n_body_problem
- gezeiten.differential_equations.two_body_problem

Module gezeiten.differential_equations.four_body_problem_complex

Contains DifferentialEquation for "complex" four body problem.

Classes

Class FourBodyProblemComplex

```
class FourBodyProblemComplex(
    time_boundaries,
    data_points_amount
)
```

Although being named generically as "four body problem", this class actually is quite specific to the earth-moon system by default. The initial conditions provided and constants used in the differential equation are based on values observed in space and should be changed for other four body problems. In contrast to FourBodyProblemSimple, this class takes into account the intrinsic rotation of Earth and the friction caused by the high tides on the surface of Earth.

Initializes the four body problem, by default with initial conditions of the earth-moon system.

Attributes

`time_boundaries` : `2-tuple` **of floats** Describes start and end time
`data_points_amount` : **int** Amount of points solvers should use as data points

Ancestors (in MRO)

- gezeiten.differential_equations.four_body_problem_simple.FourBodyProblemSimple
- gezeiten.differential_equations.two_body_problem.TwoBodyProblem
- gezeiten.differential_equation.DifferentialEquation

Descendants

- gezeiten.differential_equations.n_body_problem.NBodyProblem

Class variables

Variable initial_conditions

Methods

Method plot_intrinsic_rotation

```
def plot_intrinsic_rotation(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots intrinsic rotation of Earth with matplotlib

Attributes

`plot_title` : **string** Title to be attached to the plots
`window_title` : **string** Title to be attached to the window

Method `plot_velocity_high_tides`

```
def plot_velocity_high_tides(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots velocity of high tides with matplotlib

Attributes

`plot_title` : **string** Title to be attached to the plots
`window_title` : **string** Title to be attached to the window

Method `solve`

```
def solve(
    self,
    solver=<gezeiten.solvers.magic_solver.MagicSolver object>,
    m_0=1.4e+21
)
```

Solves the differential equation of the four body problem by using the solver passed as an argument.

Once finished, the two body problem will have a solution field which is a dictionary with entries t, x_E, y_E, vx_E, vy_E, x_M, y_M, vx_M, vy_M, phi1, vphi1, phi2, vphi2, phi_E, vphi_E, each containing a list of floats.

Attributes

`solver` : **Solver** Solver which solves the differential equation; by default MagicSolver

Module `gezeiten.differential_equations.four_body_problem_simple`

Contains DifferentialEquation for “simple” four body problem.

Classes

Class `FourBodyProblemSimple`

```
class FourBodyProblemSimple(
    time_boundaries,
    data_points_amount
)
```

Although being named generically as “four body problem”, this class actually is quite specific to the earth-moon system by default. The initial conditions provided and constants used in the differential equation are based on values observed in space and should be changed for other four body problems. In contrast to TwoBodyProblem, this class respects the influence of tidal forces on the system.

Initializes the four body problem, by default with initial conditions of the earth-moon system.

Attributes

`time_boundaries` : `2-tuple` **of floats** Describes start and end time
`data_points_amount` : **int** Amount of points solvers should use as data points

Ancestors (in MRO)

- gezeiten.differential_equations.two_body_problem.TwoBodyProblem
- gezeiten.differential_equation.DifferentialEquation

Descendants

- gezeiten.differential_equations.four_body_problem_complex.FourBodyProblemComplex

Class variables

Variable `initial_conditions`

Static methods

`Method f`

```
def f(
    t,
    r
)
```

Actual differential equation of four body problem

Attributes

`t` : float Time
`r` : array Vector containing positions and velocities of earth and moon and angles and angular velocity of high tides

Returns

array Updated r vector

Methods

Method `animate`

```
def animate(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Animates solution with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `plot_2d`

```
def plot_2d(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots positions of Earth, Moon and high tides with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `plot_distance_earth_tide`

```
def plot_distance_earth_tide(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots distance of high tides to Earth with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `plot_high_tide_angles`

```
def plot_high_tide_angles(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots angles of high tides with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `plot_high_tide_velocity_of_angles`

```
def plot_high_tide_velocity_of_angles(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots high tide velocity with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `plot_phase_high_tide`

```
def plot_phase_high_tide(
    self,
    phi_points,
    vphi_points,
    number,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of Moon with matplotlib

Method `plot_phase_high_tide_1`

```
def plot_phase_high_tide_1(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of Moon with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method **`plot_phase_high_tide_2`**

```
def plot_phase_high_tide_2(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of Moon with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method **`plot_time_series_high_tide`**

```
def plot_time_series_high_tide(
    self,
    phi_points,
    vphi_points,
    number,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of high tide with matplotlib

Attributes

`phi_points` : array Containing values of angle phi
`vphi_points` : array Containing values of angular velocity phi
`number` : string "1st" or "2nd" high tide
`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method **`plot_time_series_high_tide_1`**

```
def plot_time_series_high_tide_1(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of first high tide with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method **`plot_time_series_high_tide_2`**

```
def plot_time_series_high_tide_2(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of first high tide with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `solve`

```
def solve(
    self,
    solver=<gezeiten.solvers.magic_solver.MagicSolver object>
)
```

Solves the differential equation of the four body problem by using the solver passed as an argument.

Once finished, the two body problem will have a solution field which is a dictionary with entries t, x_E, y_E, vx_E, vy_E, x_M, y_M, vx_M, vy_M, phi1, vphi1, phi2, vphi2, each containing a list of floats.

Attributes

`solver` **:** **Solver** Solver which solves the differential equation; by default MagicSolver

Module `gezeiten.differential_equations.n_body_problem`

Contains DifferentialEquation for n body problem.

Classes

Class `NBodyProblem`

```
class NBodyProblem(
    time_boundaries,
    data_points_amount
)
```

Although being named generically as “n body problem”, this class actually is quite specific to the earth-moon system by default. The initial conditions provided and constants used in the differential equation are based on values observed in space and should be changed for other n body problems. In contrast to FourBodyProblemComplex, this class takes into account more than two tide particles.

Initializes the four body problem, by default with initial conditions of the earth-moon system.

Attributes

`time_boundaries` **:** `2-tuple` **of floats** Describes start and end time
`data_points_amount` **: int** Amount of points solvers should use as data points

Ancestors (in MRO)

- gezeiten.differential_equations.four_body_problem_complex.FourBodyProblemComplex
- gezeiten.differential_equations.four_body_problem_simple.FourBodyProblemSimple
- gezeiten.differential_equations.two_body_problem.TwoBodyProblem
- gezeiten.differential_equation.DifferentialEquation

Class variables

Variable `N`

Variable `i`

Variable `initial_conditions`

Static methods

`Method f`

```
def f(
    t,
    r
)
```

Actual differential equation of n-body problem

Attributes

`t` : float Time

`r` : array Vector containing positions and velocities of earth and moon and angles and angular velocity of high tides

Returns

array Updated r vector

Module `gezeiten.differential_equations.two_body_problem`

Contains DifferentialEquation for two body problem.

Classes

Class `TwoBodyProblem`

```
class TwoBodyProblem(
    time_boundaries,
    data_points_amount
)
```

Although being named generically as “two body problem”, this class actually is quite specific to the earth-moon system by default. The initial conditions provided and constants used in the differential equation are based on values observed in space and should be changed for other two body problems.

Initializes the two body problem, by default with initial conditions of the earth-moon system.

Attributes

`time_boundaries` : `2-tuple` of floats Describes start and end time

`data_points_amount` : int Amount of points solvers should use as data points

Ancestors (in MRO)

- gezeiten.differential_equation.DifferentialEquation

Descendants

- gezeiten.differential_equations.four_body_problem_simple.FourBodyProblemSimple

Class variables

Variable `initial_conditions`

Static methods

`Method f`

```
def f(
    t,
    r
)
```

Actual differential equation of two body problem

Attributes

t : float Time
r : array Vector containing positions and velocities of earth and moon and angles and angular velocity of high tides

Returns

array Updated r vector

Methods

Method plot

```
def plot(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Creates various plots with matplotlib

Attributes

plot_title : string Title to be attached to the plots
window_title : string Title to be attached to the window

Method plot_2d

```
def plot_2d(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots positions of Earth and Moon with matplotlib

Attributes

plot_title : string Title to be attached to the plots
window_title : string Title to be attached to the window

Method plot_center_of_mass

```
def plot_center_of_mass(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots position of center of mass with matplotlib

Attributes

plot_title : string Title to be attached to the plots
window_title : string Title to be attached to the window

Method plot_phase_earth

```
def plot_phase_earth(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of Moon with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `plot_phase_moon`

```
def plot_phase_moon(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of moon with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `plot_time_series_earth`

```
def plot_time_series_earth(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of Earth with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `plot_time_series_moon`

```
def plot_time_series_moon(
    self,
    plot_title='',
    window_title='Numerically solved earth-moon problem - Bennet Weiss and Nico Alt'
)
```

Plots time series of moon with matplotlib

Attributes

`plot_title` : string Title to be attached to the plots
`window_title` : string Title to be attached to the window

Method `solve`

```
def solve(
    self,
    solver=<gezeiten.solvers.magic_solver.MagicSolver object>
)
```

Solves the differential equation of the two body problem by using the solver passed as an argument.

Once finished, the two body problem will have a solution field which is a dictionary with entries t, x_E, y_E, vx_E, vy_E, x_M, y_M, vx_M, vy_M each containing a list of floats.

Attributes

`solver` : Solver Solver which solves the differential equation; by default MagicSolver

Module `gezeiten.exercises`

This module contains all exercises of the lecture Computational Physics.

Sub-modules

- gezeiten.exercises.exercise_2_1_b
- gezeiten.exercises.exercise_2_1_b_3d
- gezeiten.exercises.exercise_2_1_c
- gezeiten.exercises.exercise_2_2_c
- gezeiten.exercises.exercise_2_3_b
- gezeiten.exercises.exercise_2_3_c
- gezeiten.exercises.exerise_3_2

Module `gezeiten.exercises.exercise_2_1_b`

Contains plots for exercise 2.1b

Functions

Function `plot_2_1_b_euler`

```
def plot_2_1_b_euler()
```

Exercise 2.1b: plot Euler's solution of the two body problem

Function `plot_2_1_b_runge_kutta`

```
def plot_2_1_b_runge_kutta()
```

Exercise 2.1b: plot RungeKuttaSolver's solution of the two body problem

Function `plot_2_1_b_solve_ivp`

```
def plot_2_1_b_solve_ivp()
```

Exercise 2.1b: plot scipy.integrate.solve_ivp's solution of the two body problem

Module `gezeiten.exercises.exercise_2_1_b_3d`

Contains animations for exercise 2.1b

Functions

Function `animate_2_1_b_solve_ivp_3d`

```
def animate_2_1_b_solve_ivp_3d()
```

Exercise 2.1b: render 3D animation of two body problem with correct initial conditions.

Function `animate_2_1_b_solve_ivp_3d_moon_too_fast`

```
def animate_2_1_b_solve_ivp_3d_moon_too_fast()
```

Exercise 2.1b: render 3D animation of two body problem with modified velocity of the moon's orbit.

Module `gezeiten.exercises.exercise_2_1_c`

Contains plots for exercise 2.1c

Functions

Function `plot_2_1_c_solve_ivp`

```
def plot_2_1_c_solve_ivp()
```

Exercise 2.1c: plot solve_ivp's solution of the center of mass of the two body problem

Module `gezeiten.exercises.exercise_2_2_c`

Contains plots for exercise 2.2c

Functions

Function `animate_2_2_c_solve_ivp`

```
def animate_2_2_c_solve_ivp()
```

Exercise 2.2c: animate solve_ivp's solution of simple four body problem

Function `plot_2_2_c_solve_ivp`

```
def plot_2_2_c_solve_ivp()
```

Exercise 2.2c: plot solve_ivp's solution of simple four body problem

Module `gezeiten.exercises.exercise_2_3_b`

Contains plots for exercise 2.3b

Functions

Function `animate_2_3_b_solve_ivp`

```
def animate_2_3_b_solve_ivp()
```

Exercise 2.3b: animate solve_ivp's solution of complex four body problem

Function `plot_2_3_b_solve_ivp`

```
def plot_2_3_b_solve_ivp()
```

Exercise 2.3b: plot solve_ivp's solution of complex four body problem

Module `gezeiten.exercises.exercise_2_3_c`

Contains exercise 2.3c

Functions

Function `fit_2_3_c_solve_ivp`

```
def fit_2_3_c_solve_ivp()
```

Exercise 2.3c: fit mass of oceans to match tau

Module `gezeiten.exercises.exerise_3_2`

Contains plots for exercise 3.2

Functions

Function `animate_3_2_solve_ivp`

```
def animate_3_2_solve_ivp()
```

Exercise 3.2: animate solve_ivp's solution of complex n body problem

Function `plot_3_2_solve_ivp`

```
def plot_3_2_solve_ivp()
```

Exercise 3.2: plot solve_ivp's solution of complex n body problem

Module `gezeiten.solver`

Contains base class used for solvers.

Classes

Class `Solver`

```
class Solver
```

Base class for solvers implemented in gezeiten.solvers.

Descendants

- gezeiten.solvers.euler_solver.EulerSolver
- gezeiten.solvers.magic_solver.MagicSolver
- gezeiten.solvers.runge_kutta_solver.RungeKuttaSolver

Methods

Method `solve`

```
def solve(
    self,
    differential_equation
)
```

Integrate a system of ordinary differential equations.

Parameters

`differential_equation` : Instance of DifferentialEquation Differential equation to be solved

Returns

array of arrays, shape of 2nd array (len(t), len(y0)) 1st array containing the values of time. 2nd array containing the value of y for each desired time in t, with the initial value y0 in the first row.

Module `gezeiten.solvers`

This module contains all the different solvers that can be used to solve a DifferentialEquation.

Sub-modules

- gezeiten.solvers.euler_solver
- gezeiten.solvers.magic_solver
- gezeiten.solvers.runge_kutta_solver

Module `gezeiten.solvers.euler_solver`

Contains Solver based on Euler algorithm.

Classes

Class `EulerSolver`

```
class EulerSolver
```

Solver based on Euler algorithm.

Ancestors (in MRO)

- gezeiten.solver.Solver

Module `gezeiten.solvers.magic_solver`

Contains Solver based on scipy.integrate.solve_ivp function.

Classes

Class `MagicSolver`

```
class MagicSolver(
    method='Radau'
)
```

gezeiten.solver.Solver based on scipy.integrate.solve_ivp‘ function.

Attributes

`method` : string or OdeSolver, optional Integration method to use. See scipy.integrate.solve_ivp for full documentation.

Ancestors (in MRO)

- gezeiten.solver.Solver

Module `gezeiten.solvers.runge_kutta_solver`

Contains Solver based on Runge Kutta 4th order algorithm.

Classes

Class `RungeKuttaSolver`

```
class RungeKuttaSolver
```

Solver based on Runge Kutta 4th order algorithm.

Ancestors (in MRO)

- gezeiten.solver.Solver

Generated by *pdoc* 0.8.4 (https://pdoc3.github.io).