

Orbital Mechanics Library for C++ Documentation

Generated by Doxygen 1.9.5

1 General Information	1
1.0.0.1 C++ library for numerical calculations related to basic spacecraft motion issues, taking into account influence of gravity and vehicle propulsion.	1
1.1 Instalation	1
1.2 Operation	1
1.3 Units	2
1.4 json file	2
1.4.1 control	2
1.4.2 data	2
1.4.3 planets	3
1.4.4 ship	3
1.5 Example of Json file (Ship on the Earth orbit with a constant velocity):	3
2 Class Index	5
2.1 Class List	5
3 File Index	7
3.1 File List	7
4 Class Documentation	9
4.1 Control Class Reference	9
4.1.1 Detailed Description	9
4.1.2 Constructor & Destructor Documentation	10
4.1.2.1 Control()	10
4.1.3 Member Function Documentation	10
4.1.3.1 Check_input()	10
4.1.3.2 Print_Interval()	10
4.1.4 Member Data Documentation	10
4.1.4.1 engforce	10
4.1.4.2 timeend	11
4.1.4.3 timestart	11
4.2 Planet Class Reference	11
4.2.1 Detailed Description	12
4.2.2 Constructor & Destructor Documentation	12
4.2.2.1 Planet()	12
4.2.3 Member Function Documentation	12
4.2.3.1 Move_Planet()	12
4.2.3.2 Print_info()	12
4.2.4 Member Data Documentation	12
4.2.4.1 ang_velocity	13
4.2.4.2 isOrb	13
4.2.4.3 mass	13
4.2.4.4 name	13

4.2.4.5 orb_data	13
4.2.4.6 orb_pos	13
4.2.4.7 orb_radius	14
4.2.4.8 position	14
4.2.4.9 radius	14
4.2.4.10 start_ang	14
4.3 Solver Class Reference	14
4.3.1 Detailed Description	17
4.3.2 Member Enumeration Documentation	17
4.3.2.1 ode	17
4.3.3 Member Function Documentation	17
4.3.3.1 Adams_Bashforth()	17
4.3.3.2 Calculate_Grav()	18
4.3.3.3 Calculate_Net()	18
4.3.3.4 Check_Collision()	18
4.3.3.5 dvdt()	19
4.3.3.6 dxdt()	19
4.3.3.7 Euler()	20
4.3.3.8 Load_data()	20
4.3.3.9 Load_file()	20
4.3.3.10 Midpoint()	21
4.3.3.11 Move_Orbit()	21
4.3.3.12 Populate()	21
4.3.3.13 Print_Pauses()	22
4.3.3.14 Push_Back()	22
4.3.3.15 Recalculate_Forces()	22
4.3.3.16 Reset_Param()	22
4.3.3.17 Runge_Kutta()	23
4.3.3.18 Save_data()	23
4.3.3.19 Save_json()	23
4.3.3.20 Save_Planets()	23
4.3.3.21 Setup()	24
4.3.3.22 Solve()	24
4.3.3.23 UseEngine()	24
4.3.3.24 Validate_Json()	24
4.3.4 Member Data Documentation	25
4.3.4.1 a	25
4.3.4.2 a_1	25
4.3.4.3 a_2	25
4.3.4.4 distance	25
4.3.4.5 engine_data	25
4.3.4.6 engine_used	25

4.3.4.7 force_data	26
4.3.4.8 fuel_data	26
4.3.4.9 fuel_used	26
4.3.4.10 G	26
4.3.4.11 grav_forces	26
4.3.4.12 index	26
4.3.4.13 kinetic_data	27
4.3.4.14 mass_data	27
4.3.4.15 method	27
4.3.4.16 n_steps	27
4.3.4.17 Planets	27
4.3.4.18 position_data	27
4.3.4.19 potential_data	28
4.3.4.20 Ship	28
4.3.4.21 step	28
4.3.4.22 T	28
4.3.4.23 temp_force	28
4.3.4.24 temp_mass	28
4.3.4.25 time	29
4.3.4.26 time_data	29
4.3.4.27 TimeVect	29
4.3.4.28 v	29
4.3.4.29 v_1	29
4.3.4.30 v_2	29
4.3.4.31 velocity_data	30
4.4 Vector3D Class Reference	30
4.4.1 Detailed Description	31
4.4.2 Constructor & Destructor Documentation	31
4.4.2.1 Vector3D() [1/2]	31
4.4.2.2 Vector3D() [2/2]	31
4.4.3 Member Function Documentation	31
4.4.3.1 Add()	31
4.4.3.2 Divide()	32
4.4.3.3 Multiply()	32
4.4.3.4 operator*()	33
4.4.3.5 operator*=()	33
4.4.3.6 operator+=()	33
4.4.3.7 operator-=()	34
4.4.3.8 operator/()	34
4.4.3.9 operator/=()	34
4.4.3.10 Subtract()	35
4.4.3.11 VectorsEqual()	35

4.4.3.12 Zero()	36
4.4.4 Friends And Related Function Documentation	36
4.4.4.1 operator*	36
4.4.4.2 operator+	36
4.4.4.3 operator-	37
4.4.4.4 operator/	37
4.4.4.5 operator<<	38
4.4.5 Member Data Documentation	38
4.4.5.1 x	38
4.4.5.2 y	38
4.4.5.3 z	38
4.5 Vehicle Class Reference	38
4.5.1 Detailed Description	39
4.5.2 Constructor & Destructor Documentation	40
4.5.2.1 Vehicle() [1/2]	40
4.5.2.2 Vehicle() [2/2]	40
4.5.3 Member Function Documentation	40
4.5.3.1 Print_info()	40
4.5.3.2 User_set()	41
4.5.4 Member Data Documentation	41
4.5.4.1 CalculatedEnergy	41
4.5.4.2 displacement	41
4.5.4.3 engine	41
4.5.4.4 force	42
4.5.4.5 fuel	42
4.5.4.6 fuel_usage	42
4.5.4.7 KineticEnergy	42
4.5.4.8 mass	42
4.5.4.9 name	42
4.5.4.10 position	43
4.5.4.11 PotentialEnergy	43
4.5.4.12 velocity	43
5 File Documentation	45
5.1 Engineer_Thesis/Engineer_Thesis/Engineer_Thesis.cpp File Reference	45
5.1.1 Function Documentation	45
5.1.1.1 main()	45
5.2 Engineer_Thesis/Engineer_Thesis/Instructions.md File Reference	45
5.3 Engineer_Thesis/Engineer_Thesis/Planet.h File Reference	45
5.4 Planet.h	46
5.5 Engineer_Thesis/Engineer_Thesis/Solver.cpp File Reference	46
5.6 Engineer_Thesis/Engineer_Thesis/Solver.h File Reference	46

5.7 Solver.h	47
5.8 Engineer_Thesis/Engineer_Thesis/Vector3D.cpp File Reference	48
5.8.1 Function Documentation	48
5.8.1.1 operator*()	48
5.8.1.2 operator+()	49
5.8.1.3 operator-()	49
5.8.1.4 operator/()	50
5.8.1.5 operator<<()	50
5.9 Engineer_Thesis/Engineer_Thesis/Vector3D.h File Reference	50
5.10 Vector3D.h	50
5.11 Engineer_Thesis/Engineer_Thesis/Vehicle.cpp File Reference	51
5.12 Engineer_Thesis/Engineer_Thesis/Vehicle.h File Reference	51
5.13 Vehicle.h	51
Index	53

Chapter 1

General Information

1.0.0.1 C++ library for numerical calculations related to basic spacecraft motion issues, taking into account influence of gravity and vehicle propulsion.

1.1 Instalation

To use the library you need to:

1. Clone repository using: git clone https://github.com/Spectyte5/Engineer_Thesis
2. Make sure your compiler "sees" the libraries neccessary (which are located in the external folder):
 - json for modern C++: <https://github.com/tristanpenman/valijson>
 - valijson: <https://github.com/nlohmann/json>

1.2 Operation

There are two options of using the library:

1. Provide a command line argument of type string, which is a path to your *.json* file for simulation. (exemplary *.json* file can be found lower in this document) *example*:
`.\Engineer_Thesis.cpp "./JSON_files/Sim1.json"`
2. Run the code with your compiler of choice and then choose if you want to create a new simulation or load a *.json* from this directory:
`<Library directory>/JSON_files/`

1.3 Units

Units used are SI units:

- mass in kilograms [kg]
- position in meters [m]
- velocity in meters per second [m/s]
- force in Newtons [N]
- energy in Joule [J]
- angle in radians [rad]
- angular velocity in radians per second [rad/s]

1.4 json file

.json file has one object with 4 separate parts:

1.4.1 control

- **starttime** - *array* of three double type elements, time when engine will be turned on for x, y, z axis, allowing user to set each one independently.
- **endtime** - *array* of three double type elements, time when engine will be turned off for x, y, z axis, allowing user to set each one independently.
- **force** - *array* of three double type elements, magnitude and the direction of engine thrust for x, y, z axis, allowing user to set each one independently.

1.4.2 data

- **ode** - integer type *number* meaning which ODE solving method should be used: 0. Adams-Bashford

1. Euler
2. Midpoint
3. Runge-Kutta IV

- **step** - double type *number* equal to timesteps used for simulation.
- **n** - integer type *number*, the amount of steps in the simulation.

1.4.3 planets

- **name** - *string* type, name of the planet
- **mass** - double type *number*, mass of the planet
- **radius** - double type *number*, radius of the planet
- **orbit** - *boolean* type, is orbiting or strationary [true/false] For orbiting Planets (orbit = true):
- **start_angle** - double type *number*, phase used for orbital motion (start angle)
- **orbit_radius** - double type *number*, radius of the orbit
- **ang_velocity** - double type *number*, constant angular velocity of the planet
- **orbit_pos** - *array* of three double type elemetents, position of the center of the orbit (x,y,z) **NOTE:** only x,z are taken into account y value is always 0 For stationary Planets (orbit = false):
- **position** - *array* of three double type elemetents, magnitude and the direction of position vector (x,y,z)

1.4.4 ship

- **fuel** - double type *number*, fuel mass of the ship
- **fuel_usage** - double type *number*, constant ammount of fuel used when engines are turned on
- **mass** - double type *number*, total mass of the ship
- **name** - *string* type, name of the ship
- **position** - *array* of three double type elemetents, magnitude and the direction of position vector (x,y,z)
- **velocity** - *array* of three double type elemetents, magnitude and the direction of velocity vector (x,y,z)

1.5 Example of Json file (Ship on the Earth orbit with a constant velocity):

```
{
  "control": [
    {
      "endtime": [
        0.0,
        0.0,
        0.0
      ],
      "force": [
        0.0,
        0.0,
        0.0
      ],
      "starttime": [
        0.0,
        0.0,
        0.0
      ]
    }
  ],
  "data": {
    "ode": 3,
    "step": 1,
    "time": 42500
  },
  "planets": [
    {
      "name": "Earth",
      "mass": 5.972e24,
      "radius": 6.378e6,
      "orbit": false,

```

```
        "position": [
            0.0,
            0.0,
            0.0
        ]
    },
    ],
    "ship": {
        "fuel": 0.0,
        "fuel_usage": 0.0,
        "mass": 3000.0,
        "name": "Sim1",
        "position": [
            12742000.0,
            0.0,
            0.0
        ],
        "velocity": [
            0.0,
            5592.27,
            0.0
        ]
    }
}
```

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Control	Class for engine intervals used in simulation	9
Planet	Class for different Planets	11
Solver	Main class, used for solving	14
Vector3D	Class for Three-Dimensional Vectors	30
Vehicle	Class for different spaceship objects	38

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Engineer_Thesis/Engineer_Thesis/ Engineer_Thesis.cpp	45
Engineer_Thesis/Engineer_Thesis/ Planet.h	45
Engineer_Thesis/Engineer_Thesis/ Solver.cpp	46
Engineer_Thesis/Engineer_Thesis/ Solver.h	46
Engineer_Thesis/Engineer_Thesis/ Vector3D.cpp	48
Engineer_Thesis/Engineer_Thesis/ Vector3D.h	50
Engineer_Thesis/Engineer_Thesis/ Vehicle.cpp	51
Engineer_Thesis/Engineer_Thesis/ Vehicle.h	51

Chapter 4

Class Documentation

4.1 Control Class Reference

Class for engine intervals used in simulation.

```
#include <Engineer_Thesis/Engineer_Thesis/Solver.h>
```

Public Member Functions

- [Control](#) ()
- void [Print_Interval](#) ()
- bool [Check_input](#) ([Solver](#) &method)

Function for checking if intervals given by user are possible to be implemented.

Public Attributes

- [Vector3D](#) [timestart](#) = { 0,0,0 }
- [Vector3D](#) [timeend](#) = { 0,0,0 }
- [Vector3D](#) [engforce](#) = { 0,0,0 }

4.1.1 Detailed Description

Class for engine intervals used in simulation.

This class is used to create engine intervals, print and check input given by user for them.

Parameters

<i>timestart</i>	is Vector3D object at what time the engine will be turned on
<i>timeend</i>	is Vector3D object at what time the engine will be turned off
<i>engforce</i>	is Vector3D object the direction and magnitude of thrust force vector

See also

[Vector3D](#) for information about three-dimensional vectors class

4.1.2 Constructor & Destructor Documentation

4.1.2.1 Control()

```
Control::Control ( ) [inline]
```

4.1.3 Member Function Documentation

4.1.3.1 Check_input()

```
bool Control::Check_input (
    Solver & method ) [inline]
```

Function for checking if intervals given by user are possible to be implemented.

Iterates through all intervals given by the user, checks if the engine start, end and thrust values are correct and don't intersect each other.

Returns

true if all conditions are satisfied

4.1.3.2 Print_Interval()

```
void Control::Print_Interval ( ) [inline]
```

4.1.4 Member Data Documentation

4.1.4.1 engforce

```
Vector3D Control::engforce = { 0,0,0 }
```

4.1.4.2 timeend

```
Vector3D Control::timeend = { 0,0,0 }
```

4.1.4.3 timestart

```
Vector3D Control::timestart = { 0,0,0 }
```

4.2 Planet Class Reference

Class for different Planets.

```
#include <Engineer_Thesis/Engineer_Thesis/Planet.h>
```

Public Member Functions

- [Planet](#) ()
default constructor
- void [Print_info](#) ()
Function printing information about planet.
- void [Move_Planet](#) (bool save, double time)
Move [Planet](#) around orbit.

Public Attributes

- std::vector< [Vector3D](#) > [orb_data](#)
vector used for storing position data of orbiting planets
- double [mass](#) = 0
mass of the planet [kg]
- double [radius](#) = 0
radius of the planet [m]
- double [orb_radius](#) = 0
radius of the orbit [m]
- double [ang_velocity](#) = 0
angular velocity [rad/s]
- double [start_ang](#) = 0
phase [rad]
- [Vector3D](#) [position](#) = { 0,0,0 }
position of the ship
- [Vector3D](#) [orb_pos](#) = { 0,0,0 }
position of the orbit in space
- std::string [name](#) = ""
name of the planet
- bool [isOrb](#) = false
Variable connected to the orbiting of planet if true it means that the [Planet](#) orbits around a given point.

4.2.1 Detailed Description

Class for different Planets.

This class handles creating, moving and printing information about a planet

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Planet()

```
Planet::Planet ( ) [inline]
```

default constructor

4.2.3 Member Function Documentation

4.2.3.1 Move_Planet()

```
void Planet::Move_Planet (
    bool save,
    double time ) [inline]
```

Move [Planet](#) around orbit.

Function moving planet around orbit with given angular velocity starting from given angle

Parameters

<i>time</i>	is current time of simulation
-------------	-------------------------------

4.2.3.2 Print_info()

```
void Planet::Print_info ( ) [inline]
```

Function printing information about planet.

4.2.4 Member Data Documentation

4.2.4.1 ang_velocity

```
double Planet::ang_velocity = 0
```

angular velocity [rad/s]

4.2.4.2 isOrb

```
bool Planet::isOrb = false
```

Variable connected to the orbiting of planet if true it means that the [Planet](#) orbits around a given point.

4.2.4.3 mass

```
double Planet::mass = 0
```

mass of the planet [kg]

4.2.4.4 name

```
std::string Planet::name = ""
```

name of the planet

4.2.4.5 orb_data

```
std::vector<Vector3D> Planet::orb_data
```

vector used for storing position data of orbiting planets

4.2.4.6 orb_pos

```
Vector3D Planet::orb_pos = { 0,0,0 }
```

position of the orbit in space

Note

orbit is actually only x,z so the y component will always be 0

4.2.4.7 orb_radius

```
double Planet::orb_radius = 0
```

radius of the orbit [m]

4.2.4.8 position

```
Vector3D Planet::position = { 0,0,0 }
```

position of the ship

4.2.4.9 radius

```
double Planet::radius = 0
```

radius of the planet [m]

4.2.4.10 start_ang

```
double Planet::start_ang = 0
```

phase [rad]

4.3 Solver Class Reference

Main class, used for solving.

```
#include <Engineer_Thesis/Engineer_Thesis/Solver.h>
```

Public Types

- enum `ode` { `adams` , `euler` , `midpoint` , `runge` }

Enum of different solving ODEs methods.

Public Member Functions

- [Vector3D](#) [dvdt](#) ([Vector3D](#) f, double m)
Derivative of Velocity.
- [Vector3D](#) [dxdt](#) ([Vector3D](#) v)
Derivative of Position.
- void [Populate](#) ()
Define planets in simulation.
- void [Setup](#) ()
Create all simulation elements.
- bool [Validate_Json](#) (std::string &filename)
Json Validation function.
- void [Save_json](#) ()
Save simulation as a Json file.
- std::ifstream [Load_file](#) (std::string sys_path, std::string filepath, std::string extension)
Function for Loading file from a directory.
- void [Load_data](#) (std::string &filename)
Function Setting parameters from the file.
- bool [Check_Collision](#) ([Planet](#) &[Planet](#))
Collision checking function.
- bool [UseEngine](#) ()
Applying the thrust force from the engines.
- void [Calculate_Grav](#) ()
Function for calculating gravity.
- void [Calculate_Net](#) ()
Function for Calculating Net force.
- void [Reset_Param](#) ()
Function for Resetting Parameters in RKIV method.
- void [Recalculate_Forces](#) (double time, double &mass, [Vector3D](#) position, [Vector3D](#) &force)
Recalculating force in RKIV method.
- void [Euler](#) ([Vector3D](#) &velocity, [Vector3D](#) &position, [Vector3D](#) force, double mass)
Euler method.
- void [Runge_Kutta](#) ([Vector3D](#) &velocity, [Vector3D](#) &position, [Vector3D](#) &force, double &mass)
Runge-Kutta IV method.
- void [Midpoint](#) ([Vector3D](#) &velocity, [Vector3D](#) &position, [Vector3D](#) force, double mass)
Midpoint method.
- void [Adams_Bashforth](#) ([Vector3D](#) &velocity, [Vector3D](#) &position, [Vector3D](#) &force, double &mass)
Adams-Bashforth's method.
- void [Solve](#) ()
Main solving function.
- void [Push_Back](#) ()
Put all parameters in vectors.
- void [Move_Orbit](#) (bool save)
Method for changing position of orbiting planets.
- void [Save_Planets](#) ()
Function for saving planets.
- void [Save_data](#) ()
Save simulation data.
- void [Print_Pauses](#) ()
Pauses between simulation elements printing.

Public Attributes

- const double `G` = 6.67259e-11
Universal Gravitational constant.
- int `index` = 0
Index of current interval for RKIV.
- double `temp_mass`
variable to store mass for RKIV
- `Vector3D` `temp_force`
variable to store force for RKIV
- int `n_steps`
Number of steps for simulation.
- `Vector3D` `a`
variable to store current acceleration for AB
- `Vector3D` `v`
variable to store current velocity for AB
- `Vector3D` `a_1`
variable to store previous step acceleration for AB
- `Vector3D` `a_2`
variable to store acceleration from two steps before for AB
- `Vector3D` `v_1`
variable to store previous step velocity for AB
- `Vector3D` `v_2`
variable to store velocity from two steps before for AB
- `std::vector< double >` `time_data`
vectors for storing current time value
- `std::vector< double >` `mass_data`
vectors for storing current mass
- `std::vector< double >` `fuel_data`
vectors for storing current fuel value
- `std::vector< double >` `kinetic_data`
vectors for storing current kinetic energy value
- `std::vector< double >` `potential_data`
vectors for storing current potential energy value
- `std::vector< Vector3D >` `position_data`
vector used for storing current position data
- `std::vector< Vector3D >` `velocity_data`
vector used for storing current velocity data
- `std::vector< Vector3D >` `engine_data`
vector used for storing current engine data
- `std::vector< Vector3D >` `force_data`
vector used for storing current force data
- `std::vector< Planet >` `Planets`
vector storing planets in the simulation
- `std::vector< Control >` `TimeVect`
vector storing force intervals of type `Control`
- `Vehicle` `Ship` = `Vehicle`("", 0, 0, 0, 0, 0, 0, 0, 0, 0)
Ship member used in simulation.
- `Vector3D` `grav_forces`
Gravitational force and distance from `Planet` at the time `T`.
- `Vector3D` `distance`

- bool `engine_used` = false
Boolean used for removing fuel used.
- int `method` = 0
Method stored as an int used for enum.
- double `T`
time of simulation
- double `step` = 0
time step between increments
- double `time` = 0
current simulation time
- double `fuel_used` = 0
ammount of fuel_used at the iteration

4.3.1 Detailed Description

Main class, used for solving.

Class taking care of loading, validating, saving files and calculating results using different solvers

4.3.2 Member Enumeration Documentation

4.3.2.1 ode

```
enum Solver::ode
```

Enum of different solving ODEs methods.

Enumerator

adams	
euler	
midpoint	
runge	

4.3.3 Member Function Documentation

4.3.3.1 Adams_Bashforth()

```
void Solver::Adams_Bashforth (
    Vector3D & velocity,
```

```

    Vector3D & position,
    Vector3D & force,
    double & mass )

```

Adams-Bashforth's method.

Function solving and ODE using the Adams-Bashforth predictor and corrector method for calculating and setting parameters of the ship

Parameters

<i>velocity</i>	is velocity at current time
<i>position</i>	is position at current time
<i>force</i>	is force acting on the spaceship
<i>mass</i>	is mass of the spaceship

4.3.3.2 Calculate_Grav()

```
void Solver::Calculate_Grav ( )
```

Function for calculating gravity.

Iterate through planets and calculate Gravitation forces acting on the ship and it's potential energy

4.3.3.3 Calculate_Net()

```
void Solver::Calculate_Net ( )
```

Function for Calculating Net force.

Sets the value of net force taking in to account gravitational and thrust forces

See also

[Calculate_Grav\(\)](#) and [UseEngine\(\)](#) for more information about forces calculation

4.3.3.4 Check_Collision()

```
bool Solver::Check_Collision (
    Planet & Planet )
```

Collision checking function.

Function calculating distance between the Ship and [Planet](#).

Parameters

<code>Planet</code>	is a <code>Planet</code> Class object which we are checking ships collision with
---------------------	--

Returns

true if we have a collision and false if there not

Attention

If Ship is exactly on the `Planet`'s surface it does not count as a collision

4.3.3.5 `dvdtd()`

```
Vector3D Solver::dvdtd (
    Vector3D f,
    double m ) [inline]
```

Derivative of Velocity.

Function returning the derivative of velocity.

Parameters

f	is force at time t
m	is mass of the object

Returns

`Vector3D` describing velocity change (acceleration) in the last interval

4.3.3.6 `dxdt()`

```
Vector3D Solver::dxdt (
    Vector3D v ) [inline]
```

Derivative of Position.

Function returning the derivative of position.

Parameters

v	is velocity at time t
-----	-----------------------

Returns

[Vector3D](#) discribing position (velocity) change in the last interval

4.3.3.7 Euler()

```
void Solver::Euler (
    Vector3D & velocity,
    Vector3D & position,
    Vector3D force,
    double mass )
```

Euler method.

Function solving and ODE using the Euler's method and setting parameters of the ship

Parameters

<i>velocity</i>	is velocity at current time
<i>position</i>	is position at current time
<i>force</i>	is force acting on the spaceship
<i>mass</i>	is mass of the spaceship

4.3.3.8 Load_data()

```
void Solver::Load_data (
    std::string & filename )
```

Function Setting parameters from the file.

Loaded file is used to set paramaters

See also

[Load_file\(\)](#) for information about loading file

4.3.3.9 Load_file()

```
std::ifstream Solver::Load_file (
    std::string sys_path,
    std::string filepath,
    std::string extenstion )
```

Function for Loading file from a directory.

Display files in directory and open file with a given filepath

Parameters

<i>sys_path</i>	is directory in which we are looking for files
<i>filepath</i>	is path to the file
<i>extension</i>	is extension of the file ex. ".txt"

Returns

loaded file as ifstream

4.3.3.10 Midpoint()

```
void Solver::Midpoint (
    Vector3D & velocity,
    Vector3D & position,
    Vector3D force,
    double mass )
```

Midpoint method.

Function solving and ODE using the modified Euler's method (Midpoint method) and setting parameters of the ship

Parameters

<i>velocity</i>	is velocity at current time
<i>position</i>	is position at current time
<i>force</i>	is force acting on the spaceship
<i>mass</i>	is mass of the spaceship

4.3.3.11 Move_Orbit()

```
void Solver::Move_Orbit (
    bool save )
```

Method for changing position of orbiting planets.

Checks if planets is orbiting around a point and if yes changes its position and saves it to vector

4.3.3.12 Populate()

```
void Solver::Populate ( )
```

Define planets in simulation.

Gets ammount of planets in simulation, sets parameters for planet and puts it in the planets vector

4.3.3.13 Print_Pauses()

```
void Solver::Print_Pauses ( ) [inline]
```

Pauses between simulation elements printing.

Function printing '=' signs to allow better separation between simulation elements and improve comfort of reading the text displayed.

4.3.3.14 Push_Back()

```
void Solver::Push_Back ( )
```

Put all parameters in vectors.

Save all necessary values at time t into corresponding vectors

4.3.3.15 Recalculate_Forces()

```
void Solver::Recalculate_Forces (
    double time,
    double & mass,
    Vector3D position,
    Vector3D & force )
```

Recalculating force in RKIV method.

Sets the value of net force taking in to account gravitational and thrust forces

Parameters

<i>time</i>	is time at which the force and mass should be recalculated
<i>position</i>	is position at the time given
<i>force</i>	is the force that will be recalculated
<i>mass</i>	is the mass that will be recalculated

4.3.3.16 Reset_Param()

```
void Solver::Reset_Param ( )
```

Function for Resetting Parameters in RKIV method.

Resets the parameters changed for K2, K3, K4 coefficients of RKIV.

See also

[Reset_Param\(\)](#) for the function changing parameters

4.3.3.17 Runge_Kutta()

```
void Solver::Runge_Kutta (
    Vector3D & velocity,
    Vector3D & position,
    Vector3D & force,
    double & mass )
```

Runge-Kutta IV method.

Function solving and ODE using the Runge-Kutta IV-order method and setting parameters of the ship

Parameters

<i>velocity</i>	is velocity at current time
<i>position</i>	is position at current time
<i>force</i>	is force acting on the spaceship
<i>mass</i>	is mass of the spaceship

4.3.3.18 Save_data()

```
void Solver::Save_data ( )
```

Save simulation data.

Saves all parameters and calls the function for saving planets' data.

See also

[Save_Planets\(\)](#) for more information about saving planets

4.3.3.19 Save_json()

```
void Solver::Save_json ( )
```

Save simulation as a Json file.

Function used in create a simulation mode to save all parameters of the ship and planets in a json file which then can be reloaded in load mode.

4.3.3.20 Save_Planets()

```
void Solver::Save_Planets ( )
```

Function for saving planets.

Save all planets' paramaters to a seprate file

4.3.3.21 Setup()

```
void Solver::Setup ( )
```

Create all simulation elements.

Setup Particle and planets in simulation, fill all engine intervals

4.3.3.22 Solve()

```
void Solver::Solve ( )
```

Main solving function.

This function loops through time interval calling all functions used for calculation and prints result on screen.

See also

[Adams_Bashforth\(\)](#), [Midpoint\(\)](#), [Euler\(\)](#), [Runge_Kutta\(\)](#) for more information about solving ODE's

4.3.3.23 UseEngine()

```
bool Solver::UseEngine ( )
```

Applying the thrust force from the engines.

Checks if we are in any of intervals defined by user and if yes and fuel is available it applies engine force

Returns

true if engine was used and no if not

4.3.3.24 Validate_Json()

```
bool Solver::Validate_Json (
    std::string & filename )
```

Json Validation function.

Check if vector file validates against the schema

Parameters

<i>filename</i>	is a filepath for the json file that will be validated
-----------------	--

4.3.4 Member Data Documentation

4.3.4.1 a

`Vector3D Solver::a`

variable to store current acceleration for AB

4.3.4.2 a_1

`Vector3D Solver::a_1`

variable to store previous step acceleration for AB

4.3.4.3 a_2

`Vector3D Solver::a_2`

variable to store acceleration from two steps before for AB

4.3.4.4 distance

`Vector3D Solver::distance`

4.3.4.5 engine_data

`std::vector<Vector3D> Solver::engine_data`

vector used for storing current engine data

4.3.4.6 engine_used

`bool Solver::engine_used = false`

Boolean used for removing fuel used.

4.3.4.7 force_data

```
std::vector<Vector3D> Solver::force_data
```

vector used for storing current force data

4.3.4.8 fuel_data

```
std::vector<double> Solver::fuel_data
```

vectors for storing current fuel value

4.3.4.9 fuel_used

```
double Solver::fuel_used = 0
```

ammount of fuel_used at the iteration

4.3.4.10 G

```
const double Solver::G = 6.67259e-11
```

Universal Gravitational constant.

4.3.4.11 grav_forces

```
Vector3D Solver::grav_forces
```

Gravitational force and distance from [Planet](#) at the time T.

4.3.4.12 index

```
int Solver::index = 0
```

Index of current interval for RKIV.

4.3.4.13 kinetic_data

```
std::vector<double> Solver::kinetic_data
```

vectors for storing current kinetic energy value

4.3.4.14 mass_data

```
std::vector<double> Solver::mass_data
```

vectors for storing current mass

4.3.4.15 method

```
int Solver::method =0
```

Method stored as an int used for enum.

4.3.4.16 n_steps

```
int Solver::n_steps
```

Number of steps for simulation.

4.3.4.17 Planets

```
std::vector<Planet> Solver::Planets
```

vector storing planets in the simulation

4.3.4.18 position_data

```
std::vector<Vector3D> Solver::position_data
```

vector used for storing current position data

4.3.4.19 potential_data

```
std::vector<double> Solver::potential_data
```

vectors for storing current potential energy value

4.3.4.20 Ship

```
Vehicle Solver::Ship = Vehicle("", 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Ship member used in simulation.

4.3.4.21 step

```
double Solver::step = 0
```

time step between increments

4.3.4.22 T

```
double Solver::T
```

time of simulation

4.3.4.23 temp_force

```
Vector3D Solver::temp_force
```

variable to store force for RKIV

4.3.4.24 temp_mass

```
double Solver::temp_mass
```

variable to store mass for RKIV

4.3.4.25 time

```
double Solver::time = 0
```

current simulation time

4.3.4.26 time_data

```
std::vector<double> Solver::time_data
```

vectors for storing current time value

4.3.4.27 TimeVect

```
std::vector<Control> Solver::TimeVect
```

vector storing force intervals of type [Control](#)

4.3.4.28 v

```
Vector3D Solver::v
```

variable to store current velocity for AB

4.3.4.29 v_1

```
Vector3D Solver::v_1
```

variable to store previous step velocity for AB

4.3.4.30 v_2

```
Vector3D Solver::v_2
```

variable to store velocity from two steps before for AB

4.3.4.31 velocity_data

```
std::vector<Vector3D> Solver::velocity_data
```

vector used for storing current velocity data

4.4 Vector3D Class Reference

Class for Three-Dimensional Vectors.

```
#include <Engineer_Thesis/Engineer_Thesis/Vector3D.h>
```

Public Member Functions

- [Vector3D](#) ()
default constructor
- [Vector3D](#) (double [x](#), double [y](#), double [z](#))
constructor with x, y and z values
- [Vector3D](#) & [Add](#) (const [Vector3D](#) &vect)
Add two vectors.
- [Vector3D](#) & [Subtract](#) (const [Vector3D](#) &vect)
Subtract two vectors.
- [Vector3D](#) & [Multiply](#) (const [Vector3D](#) &vect)
Multiply two vectors.
- [Vector3D](#) & [Divide](#) (const [Vector3D](#) &vect)
Divide two vectors.
- [Vector3D](#) & [operator+=](#) (const [Vector3D](#) &vect)
Add two vectors with += operator.
- [Vector3D](#) & [operator-=](#) (const [Vector3D](#) &vect)
Subtract two vectors with -= operator.
- [Vector3D](#) & [operator*=](#) (const [Vector3D](#) &vect)
*Multiply two vectors with *= operator.*
- [Vector3D](#) & [operator/=](#) (const [Vector3D](#) &vect)
Divide two vectors with /= operator.
- [Vector3D](#) [operator*](#) (const double &d)
Multiply vector by scale.
- [Vector3D](#) [operator/](#) (const double &d)
Multiply vector by scale.
- [Vector3D](#) & [Zero](#) ()
Sets values of the x,y,z to 0.
- bool [VectorsEqual](#) (const [Vector3D](#) &vect)
Function checking if two vectors have the same x,y,z components.

Public Attributes

- double [x](#)
x component of the vector
- double [y](#)
y component of the vector
- double [z](#)
z component of the vector

Friends

- `Vector3D operator+` (const `Vector3D` &v1, const `Vector3D` &v2)
Add two vectors with + operator.
- `Vector3D operator-` (const `Vector3D` &v1, const `Vector3D` &v2)
Subtract two vectors with - operator.
- `Vector3D operator*` (const `Vector3D` &v1, const `Vector3D` &v2)
*Multiply two vectors with * operator.*
- `Vector3D operator/` (const `Vector3D` &v1, const `Vector3D` &v2)
Divide two vectors with / operator.
- `std::ostream & operator<<` (std::ostream &output, const `Vector3D` &vect)
overload of << operator for printing vectors

4.4.1 Detailed Description

Class for Three-Dimensional Vectors.

This class is used for operations and storing parameters of the three-dimensional vectors

4.4.2 Constructor & Destructor Documentation

4.4.2.1 `Vector3D()` [1/2]

```
Vector3D::Vector3D ( )
```

default constructor

4.4.2.2 `Vector3D()` [2/2]

```
Vector3D::Vector3D (
    double x,
    double y,
    double z )
```

constructor with x, y and z values

4.4.3 Member Function Documentation

4.4.3.1 `Add()`

```
Vector3D & Vector3D::Add (
    const Vector3D & vect )
```

Add two vectors.

4.4.3.1.1 Example `v1.Add(v2) // which equals to v1 + v2`

Parameters

<i>vect</i>	is vector being added to the vector calling this method
-------------	---

Returns

[Vector3D](#) that is a vector on which method was called with *vect* value added to it

4.4.3.2 Divide()

```
Vector3D & Vector3D::Divide (  
    const Vector3D & vect )
```

Divide two vectors.

4.4.3.2.1 Example `v1.Divide(v2) // which equals to v1 / v2`

Parameters

<i>vect</i>	is vector which the vector calling this method is divided by
-------------	--

Returns

[Vector3D](#) that is a vector on which method was called divided by *vect* value

4.4.3.3 Multiply()

```
Vector3D & Vector3D::Multiply (  
    const Vector3D & vect )
```

Multiply two vectors.

4.4.3.3.1 Example `v1.Multiply(v2) // which equals to v1 * v2`

Parameters

<i>vect</i>	is vector which the vector calling this method is multiplyied by
-------------	--

Returns

[Vector3D](#) that is a vector on which method was called multiplyied by *vect* value

4.4.3.4 operator*()

```
Vector3D Vector3D::operator* (
    const double & d )
```

Multiply vector by scale.

Parameters

<i>d</i>	is double value by which we want to multiply our vector
----------	---

Returns

[Vector3D](#) with values multiplied by d

4.4.3.5 operator*=()

```
Vector3D & Vector3D::operator*= (
    const Vector3D & vect )
```

Multiply two vectors with *= operator.

4.4.3.5.1 Example `v1 *= v2`

Parameters

<i>v1</i>	is vector multiplied
<i>v2</i>	is vector we are multiplying by

Returns

v1 multiplied by v2 value

4.4.3.6 operator+=()

```
Vector3D & Vector3D::operator+= (
    const Vector3D & vect )
```

Add two vectors with += operator.

4.4.3.6.1 Example `v1 += v2`

Parameters

<i>v1</i>	is vector which we are adding into
<i>v2</i>	is vector being added

Returns

v1 increased by v2 value

4.4.3.7 operator-=()

```
Vector3D & Vector3D::operator-= (
    const Vector3D & vect )
```

Subtract two vectors with -= operator.

4.4.3.7.1 Example `v1 -= v2`**Parameters**

<code>v1</code>	is vector which we are subtracting from
<code>v2</code>	is vector being subtracted

Returns

v1 decreased by v2 value

4.4.3.8 operator/()

```
Vector3D Vector3D::operator/ (
    const double & d )
```

Multiply vector by scale.

Parameters

<code>d</code>	is double value by which we want to divide our vector
----------------	---

Returns

`Vector3D` with values divided by d

4.4.3.9 operator/=()

```
Vector3D & Vector3D::operator/= (
    const Vector3D & vect )
```

Divide two vectors with /= operator.

4.4.3.9.1 Example `v1 /= v2`

Parameters

<i>v1</i>	is vector divided
<i>v2</i>	is vector we are dividing by

Returns

v1 divided by *v2* value

4.4.3.10 Subtract()

```
Vector3D & Vector3D::Subtract (
    const Vector3D & vect )
```

Subtract two vectors.

4.4.3.10.1 Example `v1.Subtract(v2) // which equals to v1 - v2`

Parameters

<i>vect</i>	is vector being subtracted from the vector calling this method
-------------	--

Returns

[Vector3D](#) that is a vector on which method was called with *vect* value subtracted from it

4.4.3.11 VectorsEqual()

```
bool Vector3D::VectorsEqual (
    const Vector3D & vect )
```

Function checking if two vectors have the same x,y,z components.

Parameters

<i>vect</i>	is vector compared to the vector calling this method
-------------	--

Returns

true if two vectors are the same, false if not

4.4.3.12 Zero()

```
Vector3D & Vector3D::Zero ( )
```

Sets values of the x,y,z to 0.

4.4.4 Friends And Related Function Documentation

4.4.4.1 operator*

```
Vector3D operator* (
    const Vector3D & v1,
    const Vector3D & v2 ) [friend]
```

Multiply two vectors with * operator.

4.4.4.1.1 Example `result = v1 * v2`

Parameters

<code>v1</code>	is first vector being multiplied
<code>v2</code>	is second vector we are multiplying by

Returns

vector equal to `v1 * v2`

4.4.4.2 operator+

```
Vector3D operator+ (
    const Vector3D & v1,
    const Vector3D & v2 ) [friend]
```

Add two vectors with + operator.

4.4.4.2.1 Example `result = v1 + v2`

Parameters

<code>v1</code>	is first vector being added
<code>v2</code>	is second vector being added

Returns

vector equal to $v1 + v2$

4.4.4.3 operator-

```
Vector3D operator- (
    const Vector3D & v1,
    const Vector3D & v2 ) [friend]
```

Subtract two vectors with - operator.

4.4.4.3.1 Example `result = v1 - v2`**Parameters**

<code>v1</code>	is first vector being subtracted
<code>v2</code>	is second vector being subtracted

Returns

vector equal to $v1 - v2$

4.4.4.4 operator/

```
Vector3D operator/ (
    const Vector3D & v1,
    const Vector3D & v2 ) [friend]
```

Divide two vectors with / operator.

4.4.4.4.1 Example `result = v1 / v2`**Parameters**

<code>v1</code>	is first vector being divided
<code>v2</code>	is second vector we are dividing by

Returns

vector equal to $v1 / v2$

4.4.4.5 operator<<

```
std::ostream & operator<< (
    std::ostream & output,
    const Vector3D & vect ) [friend]
```

overload of << operator for printing vectors

Parameters

<i>output</i>	is ostream where we will print data
<i>vect</i>	is a vector being printed

4.4.5 Member Data Documentation

4.4.5.1 x

```
double Vector3D::x
```

x component of the vector

4.4.5.2 y

```
double Vector3D::y
```

y component of the vector

4.4.5.3 z

```
double Vector3D::z
```

z component of the vector

4.5 Vehicle Class Reference

Class for different spaceship objects.

```
#include <Engineer_Thesis/Engineer_Thesis/Vehicle.h>
```

Public Member Functions

- [Vehicle](#) ()
Default Constructor.
- [Vehicle](#) (std::string n, double rx, double ry, double rz, double vx, double vy, double vz, double m, double [fuel](#), double [fuel_usage](#))
Constructor assigning given paramaters.
- void [Print_info](#) ()
Print information about the Ship.
- void [User_set](#) ()
User set ships parameters.

Public Attributes

- std::string [name](#)
name of the ship
- [Vector3D](#) [position](#)
[Vector3D](#) position on x,y,z axis [m].
- [Vector3D](#) [velocity](#)
[Vector3D](#) velocity on x,y,z axis [m/s].
- [Vector3D](#) [engine](#) = { 0,0,0 }
engine is a [Vector3D](#) thrust force on x,y,z axis [N]
- [Vector3D](#) [force](#) = { 0,0,0 }
[Vector3D](#) net force acting on spaceship on x,y,z axis [N].
- [Vector3D](#) [displacement](#) = { 0,0,0 }
displacement of the ship from initial position[m]
- double [mass](#) = 0
total mass of the ship with fuel [kg]
- double [fuel](#) = 0
mass of fuel carried by the ship[kg]
- double [fuel_usage](#) = 0
ammount of fuel used by engines [kg/s]
- double [PotentialEnergy](#) = 0
total potential energy from all planets acting on the spaceship [J]
- double [KineticEnergy](#) = 0
energy from velocity whith which spaceship is moving [J]
- bool [CalculatedEnergy](#) = 0
true or false depending on whether the planets where already initialized

4.5.1 Detailed Description

Class for different spaceship objects.

Ship is a body having no size, no rotation (Point-mass)

Note

In few places the mass actually is a mass without fuel, inputed by user that then has the fuel mass added to it.

See also

[Vector3D](#) for more information about three-dimenstional vector objects

4.5.2 Constructor & Destructor Documentation

4.5.2.1 Vehicle() [1/2]

```
Vehicle::Vehicle ( ) [inline]
```

Default Constructor.

4.5.2.2 Vehicle() [2/2]

```
Vehicle::Vehicle (
    std::string n,
    double rx,
    double ry,
    double rz,
    double vx,
    double vy,
    double vz,
    double m,
    double fuel,
    double fuel_usage )
```

Constructor assiging given paramaters.

Parameters

<i>n</i>	is name of the ship
<i>rx</i>	is position on x axis [m]
<i>ry</i>	is position on y axis [m]
<i>rz</i>	is position on z axis [m]
<i>vx</i>	is velocity on x axis [m/s]
<i>vy</i>	is velocity on y axis [m/s]
<i>vz</i>	is velocity on z axis [m/s]
<i>m</i>	is mass of the ship [kg]
<i>fuel</i>	is mass of fuel carried by the ship [kg]
<i>fuel_usage</i>	is ammount of fuel used in [kg/s]

4.5.3 Member Function Documentation

4.5.3.1 Print_info()

```
void Vehicle::Print_info ( )
```


Print information about the Ship.

Function for printing each paramter of the ship on screen

4.5.3.2 User_set()

```
void Vehicle::User_set ( )
```

User set ships parameters.

Function allowing user to set values of the Ship, used in create a simulation mode.

4.5.4 Member Data Documentation

4.5.4.1 CalculatedEnergy

```
bool Vehicle::CalculatedEnergy = 0
```

true or false depending on whether the planets where already initialized

See also

[Planet](#) more info about planets

4.5.4.2 displacement

```
Vector3D Vehicle::displacement = { 0,0,0 }
```

displacement of the ship from initial position[m]

4.5.4.3 engine

```
Vector3D Vehicle::engine = { 0,0,0 }
```

engine is a [Vector3D](#) thrust force on x,y,z axis [N]

4.5.4.4 force

```
Vector3D Vehicle::force = { 0,0,0 }
```

Vector3D net force acting on spaceship on x,y,z axis [N].

4.5.4.5 fuel

```
double Vehicle::fuel = 0
```

mass of fuel carried by the ship[kg]

4.5.4.6 fuel_usage

```
double Vehicle::fuel_usage = 0
```

ammount of fuel used by engines [kg/s]

4.5.4.7 KineticEnergy

```
double Vehicle::KineticEnergy = 0
```

energy from velocity whith which spaceship is moving [J]

4.5.4.8 mass

```
double Vehicle::mass = 0
```

total mass of the ship with fuel [kg]

4.5.4.9 name

```
std::string Vehicle::name
```

name of the ship

4.5.4.10 position

`Vector3D Vehicle::position`

`Vector3D` position on x,y,z axis [m].

4.5.4.11 PotentialEnergy

`double Vehicle::PotentialEnergy = 0`

total potential energy from all planets acting on the spaceship [J]

4.5.4.12 velocity

`Vector3D Vehicle::velocity`

`Vector3D` velocity on x,y,z axis [m/s].

Chapter 5

File Documentation

5.1 Engineer_Thesis/Engineer_Thesis/Engineer_Thesis.cpp File Reference

```
#include <iostream>
#include "Solver.h"
```

Functions

- int [main](#) (int argc, char *argv[])

5.1.1 Function Documentation

5.1.1.1 main()

```
int main (
    int argc,
    char * argv[] )
```

5.2 Engineer_Thesis/Engineer_Thesis/Instructions.md File Reference

5.3 Engineer_Thesis/Engineer_Thesis/Planet.h File Reference

```
#include "Vector3D.h"
```

Classes

- class [Planet](#)

Class for different Planets.

5.4 Planet.h

[Go to the documentation of this file.](#)

```

1 #pragma once
2 #include "Vector3D.h"
3
4
5
6
7 class Planet {
8
9 public:
10
11     std::vector <Vector3D> orb_data;
12
13     double mass = 0;
14     double radius = 0;
15     double orb_radius = 0;
16     double ang_velocity = 0;
17     double start_ang = 0;
18
19     Vector3D position = { 0,0,0 };
20     Vector3D orb_pos = { 0,0,0 };
21     std::string name = "";
22     bool isOrb = false;
23
24     Planet() {}
25
26     void Print_info() {
27
28         std::cout << "\nName: " << name << "\nMass: " << mass << " kg" << "\nRadius: " << radius << " m" <<
29         "\nPosition: " << position << " m";
30
31         if (isOrb) {
32             std::cout << "\nOrbit Radius: " << orb_radius << " m" << "\nOrbit Velocity: " << ang_velocity <<
33             " rad/s" << "\nOrbit Center: " << orb_pos << " m";
34         }
35
36         std::cout << "\n";
37     }
38
39     void Move_Planet(bool save, double time) {
40
41         position.x = orb_pos.x + orb_radius * cos(start_ang + ang_velocity * time);
42         position.z = orb_pos.z + orb_radius * sin(start_ang + ang_velocity * time);
43
44         if (save) {
45             orb_data.push_back(position);
46         }
47     }
48 };

```

5.5 Engineer_Thesis/Engineer_Thesis/Solver.cpp File Reference

```

#include "Solver.h"
#include <filesystem>
#include <json.hpp>
#include <valijson_nlohmann_bundled.hpp>

```

5.6 Engineer_Thesis/Engineer_Thesis/Solver.h File Reference

```

#include <fstream>
#include <iomanip>
#include "Vehicle.h"
#include "Planet.h"

```

Classes

- class [Solver](#)
Main class, used for solving.
- class [Control](#)
Class for engine intervals used in simulation.

5.7 Solver.h

[Go to the documentation of this file.](#)

```

1 #pragma once
2 #include <fstream>
3 #include <iomanip>
4 #include "Vehicle.h"
5 #include "Planet.h"
6
7
8 //forward declare class
9 class Control;
10
11
12 class Solver {
13 public:
14
15     const double G = 6.67259e-11;
16     int index = 0;
17     double temp_mass;
18     Vector3D temp_force;
19     int n_steps;
20     Vector3D a;
21     Vector3D v;
22     Vector3D a_1;
23     Vector3D a_2;
24     Vector3D v_1;
25     Vector3D v_2;
26     std::vector<double> time_data;
27     std::vector<double> mass_data;
28     std::vector<double> fuel_data;
29     std::vector<double> kinetic_data;
30     std::vector<double> potential_data;
31     std::vector<Vector3D> position_data;
32     std::vector<Vector3D> velocity_data;
33     std::vector<Vector3D> engine_data;
34     std::vector<Vector3D> force_data;
35     std::vector<Planet> Planets;
36     std::vector<Control> TimeVect;
37     Vehicle Ship = Vehicle("", 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
38     Vector3D grav_forces, distance;
39     bool engine_used = false;
40     enum ode { adams, euler, midpoint, runge};
41     int method=0;
42     double T;
43     double step = 0;
44     double time = 0;
45     double fuel_used = 0;
46     inline Vector3D dvdt(Vector3D f, double m) { return f/m; }
47     inline Vector3D dxdt(Vector3D v) { return v; }
48     void Populate();
49     void Setup();
50     bool Validate_Json(std::string& filename);
51     void Save_json();
52     std::ifstream Load_file(std::string sys_path, std::string filepath, std::string extenstion);
53     void Load_data(std::string& filename);
54     bool Check_Collision(Planet& Planet);
55     bool UseEngine();
56     void Calculate_Grav();
57     void Calculate_Net();
58     void Reset_Param();
59
60     void Recalculate_Forces(double time, double& mass, Vector3D position, Vector3D& force);
61     void Euler(Vector3D& velocity, Vector3D& position, Vector3D force, double mass);
62     void Runge_Kutta(Vector3D& velocity, Vector3D& position, Vector3D& force, double& mass);
63     void Midpoint(Vector3D& velocity, Vector3D& position, Vector3D force, double mass);
64     void Adams_Bashforth(Vector3D& velocity, Vector3D& position, Vector3D& force, double& mass);
65     void Solve();
66     void Push_Back();
67     void Move_Orbit(bool save);
68     void Save_Planets();

```

```

211     void Save_data();
212
216     void Print_Pauses() {
217         std::cout << std::setfill('=') << std::setw(120) << "\n";
218     }
219 };
220
228 class Control{
229
230 public:
231     Vector3D timestart= { 0,0,0 }, timeend = { 0,0,0 }, engforce = { 0,0,0 };
232
233     Control() {};;
234
235     void Print_Interval() {
236         std::cout << "\nStart times(x,y,z): " << timestart << " s"
237             << "\nEnd times(x,y,z): " << timeend << " s"
238             << "\nEngine force(x,y,z): " << engforce << " N" << std::endl;
239     }
244     bool Check_input(Solver& method) {
245         //initial check:
246         if (method.TimeVect.empty()) {
247             //timestart
248             if (timestart.x < 0 || timestart.y < 0 || timestart.z < 0) return false;
249             //timeend
250             if (timeend.x > method.T || timeend.y > method.T || timeend.z > method.T) return false;
251             if (timeend.x < timestart.x || timeend.y < timestart.y || timeend.z < timestart.z) return
false;
252         }
253         else {
254             if (timestart.x < 0 || timestart.y < 0 || timestart.z < 0) return false;
255             if (timeend.x > method.T || timeend.y > method.T || timeend.z > method.T) return false;
256             if (timeend.x < timestart.x || timeend.y < timestart.y || timeend.z < timestart.z) return
false;
257             //check if interval does not intersect previous interval
258             if (timestart.x < method.TimeVect.back().timeend.x || timestart.y <
method.TimeVect.back().timeend.y || timestart.z < method.TimeVect.back().timeend.z) return false;
259         }
260         return true;
261     }
262 };

```

5.8 Engineer_Thesis/Engineer_Thesis/Vector3D.cpp File Reference

```
#include "Vector3D.h"
```

Functions

- [Vector3D operator+](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- [Vector3D operator-](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- [Vector3D operator*](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- [Vector3D operator/](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- [std::ostream & operator<<](#) (std::ostream &output, const [Vector3D](#) &vect)

5.8.1 Function Documentation

5.8.1.1 operator*()

```

Vector3D operator* (
    const Vector3D & v1,
    const Vector3D & v2 )

```

5.8.1.1.1 Example result = v1 * v2

Parameters

<i>v1</i>	is first vector being multiplyied
<i>v2</i>	is second vector we are multiplying by

Returns

vector equal to $v1 * v2$

5.8.1.2 operator+()

```
Vector3D operator+ (
    const Vector3D & v1,
    const Vector3D & v2 )
```

5.8.1.2.1 Example `result = v1 + v2`

Parameters

<i>v1</i>	is first vector being added
<i>v2</i>	is second vector being added

Returns

vector equal to $v1 + v2$

5.8.1.3 operator-()

```
Vector3D operator- (
    const Vector3D & v1,
    const Vector3D & v2 )
```

5.8.1.3.1 Example `result = v1 - v2`

Parameters

<i>v1</i>	is first vector being subtracted
<i>v2</i>	is second vector being subtracted

Returns

vector equal to $v1 - v2$

5.8.1.4 operator/()

```
Vector3D operator/ (
    const Vector3D & v1,
    const Vector3D & v2 )
```

5.8.1.4.1 Example `result = v1 / v2`

Parameters

<code>v1</code>	is first vector being divided
<code>v2</code>	is second vector we are dividing by

Returns

vector equal to `v1 / v2`

5.8.1.5 operator<<()

```
std::ostream & operator<< (
    std::ostream & output,
    const Vector3D & vect )
```

Parameters

<code>output</code>	is ostream where we will print data
<code>vect</code>	is a vector being printed

5.9 Engineer_Thesis/Engineer_Thesis/Vector3D.h File Reference

```
#include <iostream>
```

Classes

- class `Vector3D`
Class for Three-Dimensional Vectors.

5.10 Vector3D.h

[Go to the documentation of this file.](#)

```
1 #pragma once
2 #include <iostream>
3
```

```

7  class Vector3D {
8
9  public:
11     double x;
13     double y;
15     double z;
16
18     Vector3D();
20     Vector3D(double x, double y, double z);
21
30     Vector3D& Add(const Vector3D& vect);
39     Vector3D& Subtract(const Vector3D& vect);
48     Vector3D& Multiply(const Vector3D& vect);
57     Vector3D& Divide(const Vector3D& vect);
67     friend Vector3D operator+ (const Vector3D& v1, const Vector3D& v2);
77     friend Vector3D operator- (const Vector3D& v1, const Vector3D& v2);
87     friend Vector3D operator* (const Vector3D& v1, const Vector3D& v2);
97     friend Vector3D operator/ (const Vector3D& v1, const Vector3D& v2);
107    Vector3D& operator+=(const Vector3D& vect);
117    Vector3D& operator-=(const Vector3D& vect);
127    Vector3D& operator*=(const Vector3D& vect);
137    Vector3D& operator/=(const Vector3D& vect);
138
143    Vector3D operator*(const double& d);
144
149    Vector3D operator/(const double& d);
150
152    Vector3D& Zero();
153
158    friend std::ostream& operator < (std::ostream& output, const Vector3D& vect);
159
164    bool VectorsEqual(const Vector3D& vect);
165 };

```

5.11 Engineer_Thesis/Engineer_Thesis/Vehicle.cpp File Reference

```
#include "Vehicle.h"
```

5.12 Engineer_Thesis/Engineer_Thesis/Vehicle.h File Reference

```
#include "Vector3D.h"
#include <vector>
```

Classes

- class [Vehicle](#)
Class for different spaceship objects.

5.13 Vehicle.h

[Go to the documentation of this file.](#)

```

1  #pragma once
2  #include "Vector3D.h"
3  #include <vector>
4
10 class Vehicle {
11
12 public:
14     std::string name;
16     Vector3D position;
18     Vector3D velocity;

```

```
20     Vector3D engine = { 0,0,0 };
22     Vector3D force = { 0,0,0 };
24     Vector3D displacement = { 0,0,0 };
26     double mass = 0;
28     double fuel = 0;
30     double fuel_usage = 0;
32     double PotentialEnergy = 0;
34     double KineticEnergy = 0;
38     bool CalculatedEnergy = 0;
39
41     Vehicle() {}
42
44     Vehicle(std::string n, double rx, double ry, double rz, double vx, double vy, double vz, double m,
45             double fuel, double fuel_usage);
46     void Print_info();
47     void User_set();
48 };
```

Index

- a
 - Solver, [25](#)
- a_1
 - Solver, [25](#)
- a_2
 - Solver, [25](#)
- adams
 - Solver, [17](#)
- Adams_Bashforth
 - Solver, [17](#)
- Add
 - Vector3D, [31](#)
- ang_velocity
 - Planet, [12](#)
- Calculate_Grav
 - Solver, [18](#)
- Calculate_Net
 - Solver, [18](#)
- CalculatedEnergy
 - Vehicle, [41](#)
- Check_Collision
 - Solver, [18](#)
- Check_input
 - Control, [10](#)
- Control, [9](#)
 - Check_input, [10](#)
 - Control, [10](#)
 - engforce, [10](#)
 - Print_Interval, [10](#)
 - timeend, [10](#)
 - timestart, [11](#)
- displacement
 - Vehicle, [41](#)
- distance
 - Solver, [25](#)
- Divide
 - Vector3D, [32](#)
- dvdt
 - Solver, [19](#)
- dxdt
 - Solver, [19](#)
- engforce
 - Control, [10](#)
- engine
 - Vehicle, [41](#)
- engine_data
 - Solver, [25](#)
- engine_used
 - Solver, [25](#)
- Engine_Thesis.cpp
 - main, [45](#)
- Engine_Thesis/Engine_Thesis/Engine_Thesis.cpp,
 - [45](#)
- Engine_Thesis/Engine_Thesis/Instructions.md, [45](#)
- Engine_Thesis/Engine_Thesis/Planet.h, [45](#), [46](#)
- Engine_Thesis/Engine_Thesis/Solver.cpp, [46](#)
- Engine_Thesis/Engine_Thesis/Solver.h, [46](#), [47](#)
- Engine_Thesis/Engine_Thesis/Vector3D.cpp, [48](#)
- Engine_Thesis/Engine_Thesis/Vector3D.h, [50](#)
- Engine_Thesis/Engine_Thesis/Vehicle.cpp, [51](#)
- Engine_Thesis/Engine_Thesis/Vehicle.h, [51](#)
- Euler
 - Solver, [20](#)
- euler
 - Solver, [17](#)
- force
 - Vehicle, [41](#)
- force_data
 - Solver, [25](#)
- fuel
 - Vehicle, [42](#)
- fuel_data
 - Solver, [26](#)
- fuel_usage
 - Vehicle, [42](#)
- fuel_used
 - Solver, [26](#)
- G
 - Solver, [26](#)
- grav_forces
 - Solver, [26](#)
- index
 - Solver, [26](#)
- isOrb
 - Planet, [13](#)
- kinetic_data
 - Solver, [26](#)
- KineticEnergy
 - Vehicle, [42](#)
- Load_data
 - Solver, [20](#)
- Load_file
 - Solver, [20](#)

- main
 - Engineer_Thesis.cpp, 45
- mass
 - Planet, 13
 - Vehicle, 42
- mass_data
 - Solver, 27
- method
 - Solver, 27
- Midpoint
 - Solver, 21
- midpoint
 - Solver, 17
- Move_Orbit
 - Solver, 21
- Move_Planet
 - Planet, 12
- Multiply
 - Vector3D, 32
- n_steps
 - Solver, 27
- name
 - Planet, 13
 - Vehicle, 42
- ode
 - Solver, 17
- operator<<
 - Vector3D, 37
 - Vector3D.cpp, 50
- operator*
 - Vector3D, 32, 36
 - Vector3D.cpp, 48
- operator*=
 - Vector3D, 33
- operator+
 - Vector3D, 36
 - Vector3D.cpp, 49
- operator+=
 - Vector3D, 33
- operator-
 - Vector3D, 37
 - Vector3D.cpp, 49
- operator-=
 - Vector3D, 34
- operator/
 - Vector3D, 34, 37
 - Vector3D.cpp, 49
- operator/=
 - Vector3D, 34
- orb_data
 - Planet, 13
- orb_pos
 - Planet, 13
- orb_radius
 - Planet, 13
- Planet, 11
 - ang_velocity, 12
 - isOrb, 13
 - mass, 13
 - Move_Planet, 12
 - name, 13
 - orb_data, 13
 - orb_pos, 13
 - orb_radius, 13
 - Planet, 12
 - position, 14
 - Print_info, 12
 - radius, 14
 - start_ang, 14
- Planets
 - Solver, 27
- Populate
 - Solver, 21
- position
 - Planet, 14
 - Vehicle, 42
- position_data
 - Solver, 27
- potential_data
 - Solver, 27
- PotentialEnergy
 - Vehicle, 43
- Print_info
 - Planet, 12
 - Vehicle, 40
- Print_Interval
 - Control, 10
- Print_Pauses
 - Solver, 21
- Push_Back
 - Solver, 22
- radius
 - Planet, 14
- Recalculate_Forces
 - Solver, 22
- Reset_Param
 - Solver, 22
- runge
 - Solver, 17
- Runge_Kutta
 - Solver, 22
- Save_data
 - Solver, 23
- Save_json
 - Solver, 23
- Save_Planets
 - Solver, 23
- Setup
 - Solver, 23
- Ship
 - Solver, 28
- Solve
 - Solver, 24

Solver, 14
 a, 25
 a_1, 25
 a_2, 25
 adams, 17
 Adams_Bashforth, 17
 Calculate_Grav, 18
 Calculate_Net, 18
 Check_Collision, 18
 distance, 25
 dvdt, 19
 dxdt, 19
 engine_data, 25
 engine_used, 25
 Euler, 20
 euler, 17
 force_data, 25
 fuel_data, 26
 fuel_used, 26
 G, 26
 grav_forces, 26
 index, 26
 kinetic_data, 26
 Load_data, 20
 Load_file, 20
 mass_data, 27
 method, 27
 Midpoint, 21
 midpoint, 17
 Move_Orbit, 21
 n_steps, 27
 ode, 17
 Planets, 27
 Populate, 21
 position_data, 27
 potential_data, 27
 Print_Pauses, 21
 Push_Back, 22
 Recalculate_Forces, 22
 Reset_Param, 22
 runge, 17
 Runge_Kutta, 22
 Save_data, 23
 Save_json, 23
 Save_Planets, 23
 Setup, 23
 Ship, 28
 Solve, 24
 step, 28
 T, 28
 temp_force, 28
 temp_mass, 28
 time, 28
 time_data, 29
 TimeVect, 29
 UseEngine, 24
 v, 29
 v_1, 29
 v_2, 29
 Validate_Json, 24
 velocity_data, 29
 start_ang
 Planet, 14
 step
 Solver, 28
 Subtract
 Vector3D, 35

T
 Solver, 28
temp_force
 Solver, 28
temp_mass
 Solver, 28
time
 Solver, 28
time_data
 Solver, 29
timeend
 Control, 10
timestart
 Control, 11
TimeVect
 Solver, 29

UseEngine
 Solver, 24
User_set
 Vehicle, 41

v
 Solver, 29
v_1
 Solver, 29
v_2
 Solver, 29
Validate_Json
 Solver, 24
Vector3D, 30
 Add, 31
 Divide, 32
 Multiply, 32
 operator<<, 37
 operator*, 32, 36
 operator*==, 33
 operator+, 36
 operator+=, 33
 operator-, 37
 operator-=, 34
 operator/, 34, 37
 operator/=, 34
 Subtract, 35
 Vector3D, 31
 VectorsEqual, 35
 x, 38
 y, 38
 z, 38

- Zero, [35](#)
- Vector3D.cpp
 - operator<<, [50](#)
 - operator*, [48](#)
 - operator+, [49](#)
 - operator-, [49](#)
 - operator/, [49](#)
- VectorsEqual
 - Vector3D, [35](#)
- Vehicle, [38](#)
 - CalculatedEnergy, [41](#)
 - displacement, [41](#)
 - engine, [41](#)
 - force, [41](#)
 - fuel, [42](#)
 - fuel_usage, [42](#)
 - KineticEnergy, [42](#)
 - mass, [42](#)
 - name, [42](#)
 - position, [42](#)
 - PotentialEnergy, [43](#)
 - Print_info, [40](#)
 - User_set, [41](#)
 - Vehicle, [40](#)
 - velocity, [43](#)
- velocity
 - Vehicle, [43](#)
- velocity_data
 - Solver, [29](#)
- x
 - Vector3D, [38](#)
- y
 - Vector3D, [38](#)
- z
 - Vector3D, [38](#)
- Zero
 - Vector3D, [35](#)