МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по курсовой работе

по дисциплине «Дифференциальные уравнения»

Тема: Солнечная система

Студенты гр. 8383 _____ Федоров И.А.
Гречко В.Д.

Преподаватель _____ Павлов Д.А.

Санкт-Петербург
2021

Цель работы

Реализовать с помощью различных численных методов движение объектов солнечной системы (планет и Солнца) как материальных точек, подчиненных законам Ньютона.

Выполнение работы

Симуляция движения планет рассматривается как гравитационная задача *n* тел. Имеется *n* материальных точек, массы которых известны. Попарное взаимодействие точек подчинено закону тяготения Ньютона. Известны начальные положения и скорости каждой точки. Требуется найти положения точек в последующие моменты времени.

Задачу можно кратко описать следущими уравнениями:

$$\frac{d\bar{r}_i}{dt} = \bar{\upsilon}_i,$$

$$\frac{d\bar{\upsilon}_i}{dt} = G\sum_{j\neq i}^{n} m_j \frac{\bar{r}_j - \bar{r}_i}{r_{ij}^3} \Rightarrow$$

$$\frac{d^2 x_i}{dt^2} = G\sum_{j\neq i}^{n} m_j \frac{x_j - x_i}{r_{ij}^3}$$

$$\frac{d^2 y_i}{dt^2} = G\sum_{j\neq i}^{n} m_j \frac{y_j - y_i}{r_{ij}^3} \ldots$$

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

Ниже кратко описываются некоторые детали реализации.

Был реализован класс ***OdeIntegrator***, которые предоставляет небольшой набор численных методов (РК4, РК8 и Дормана-Принса 8(7)). Основные поля и методы класса:

- `calc_diff_eqs` – функция *f* (вычисление правой части дифференциального уравнения), передается пользователем.
- `set_f_params()` - метод для установки дополнительных параметров для функции *f*.
- `set_f()` – метод для установки пользовательской функции *f*.

- `set_method_params()` – метод для установки дополнительных параметров для численного метода.
- `set_integrator()` – метод для выбора численного методы по переданной строке.
- `set_init_params()` – метод для задания начальных условий (*y, t, dt*).
- `set_solution_out()` – метод для установки функции (callback), которая будет вызываться на каждом шаге метода.
- `integrate()` – метод, который вычисляет $y(total\_time)$, используя заданный численный метод.
- `rk4_method()`,`rk8_method()`,`prince_dormand_87_method()` – функции, реализующие соответствующие численные методы.

Был реализован класс ***Body***, который отвечает за хранение информации об одном теле (материальной точке). Необходим для создания экземпляра материальной точки и дальнейшей передачи в класс ***Simulation*** (описан ниже). Основные поля и методы данного класса:

- `mass` – масса точки.
- `x_vec` – вектор длины 3, хранящий местоположение точки.
- `v_vec` – вектор длины 3, хранящий вектор скорости точки.
- `name` – строка с именем точки.
- `has_units` – атрибут, отвечающий за систему единиц измерения характеристик точки (СГС, СИ либо же безразмерная).
- `return_vec()` – метод, возвращающий вектор, склеенный из `x_vec` и `v_vec`.

Был создан класс ***Simulation***, который отвечает за проведение симуляции и имеет соответствующие для этого методы. Основные поля и методы класса:

- `bodies` – вектор экземпляров класса ***Body***.
- `mass_vec` – вектор масс всех точек.
- `name_vec` – вектор всех имен.
- `ode_integrator` – экземпляр класса `OdeIntegrator`, который предоставляет численные методы.
- `set_diff_eqs()` – метод для установки пользователем функции f (вычисление дифференциального уравнения). Переданная функция и дополнительные параметры будут переданы полю `ode_integrator`.
- `set_ode_integator()` – метод, позволяющий пользователю установить в поле `ode_integrator` свой экземпляр класса `OdeIntegrator`.
- `set_numeric_method()` – метод для установки (выбора) численного метода. Получает строку, в зависимости от которой устанавливает нужный метод.
- `run_simulation()` – метод, запускающий симуляцию.
- `history` – хранит в себе историю вычислений в ходе выполнения метода `run_simulation()`.
- `quant_vec` – вектор, хранящий `x_vec` и `v_vec` для всех точек, имеет следующий вид (пример для 3 точек, x_i имеет размерность 3):

$$y = \begin{pmatrix} x_1 \\ \dot{x_1} \\ x_2 \\ \dot{x_2} \\ x_3 \\ \dot{x_3} \end{pmatrix}$$

Т.к. данная задача не имеет аналитического решения, то качество интегрирование будет оцениваться с помощью инвариантов барицентра системы.

Положение барицентра системы материальных точек можно вычислить следующим образом:

$$\bar{r}_c = \frac{\sum_i m_i \bar{r}_i}{M_c}, M_c = \sum_i mi$$

где M_c – масса всей системы.

Существует теорема *о движении центра масс системы*, одна из формулировок которой гласит: центр масс движется так, как двигалась бы материальная точка, масса которой равна массе системы, под действием силы, равной сумме всех внешних сил, действующих на систему. Следствием данной теоремы является *закон сохранения движения центра масс*: если сумма внешних сил, действующих на систему, равна нулю, то центр масс такой системы движется с постоянной скоростью. Т.к. в нашей системы отсутствуют внешние силы, то данный закон должен выполняться.

Скорость барицентра системы материальных точек можно вычислить следующим образом:

$$\bar{\upsilon}_c = \frac{\sum_i m_i \bar{\upsilon}_i}{M_c}, M_c = \sum_i mi$$

Еще одним инвариантом является полная энергия системы *n* тел, которая является постоянной. Вычислить ее можно следующим образом:

$$T - U = E,$$

где *E* – полная энергия системы, *T* – кинетическая энергия системы, -*U* – потенциальная энергия. Кинетическая и потенциальная энергии рассчитываются следующим образом:

$$T = \frac{1}{2}\sum_{i=1}^{n} m_i V_i^2,$$

$$U = \frac{1}{2} G \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{m_i m_j}{r_{ij}},\ (i \neq j).$$

Для оценивания методов и сравнения их между собой реализованы функции-методы класса ***Simulation***: `check_barycent_inv()` и `_calc_barycent_v()`. Они рассчитывают скорость барицентра для каждой итерации работы метода (результаты хранятся в поле `history`) и сравнивают с первоначальной скоростью барицентра. Т.к. скорости векторные величины, то оценивается расстояние между ними:

$$dist(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2}\ .$$

Функция-метод `check_energy_inv()` необходима для рассчета относительного изменения полной энергии системы для каждой итерации.

Т.к. число итераций у некоторых методов велико, то для большей наглядности реализована возможность построения графиков расстояний между начальной скоростью барицентра и скоростью на текущей итерации с помощью функции `smooth_graph_points()`, которая заменяет каждое значение экспоненциальным скользящим средним по предыдущим значениям, чтобы получить более "гладкий" график. Вычисляется по формуле:

$$EMA_t = \alpha \cdot P_t + (1 - \alpha) \cdot EMA_{t-1}$$

где α – коэффиент в интервале от 0 до 1.

Был реализован явный одношаговый метод Рунге-Кутты 4-го порядка. Таблица Бутчера для него имеет следующий вид:

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
	1/6	1/3	1/3	1/6

Рисунок 1 – таблица Бутчера для РК4

Код реализации данного метода представлен ниже:

```
    def rk4_method(self, total_time):
        '''RK4 method. Returns new [y] vector for total_time.'''
        while self.t <= total_time:
            k1 = self.dt * self.calc_diff_eqs(self.t,
self.y_prev, **self.f_params)
            k2 = self.dt * self.calc_diff_eqs(self.t +
0.5*self.dt, self.y_prev + 0.5*k1, **self.f_params)
            k3 = self.dt * self.calc_diff_eqs(self.t +
0.5*self.dt, self.y_prev + 0.5*k2, **self.f_params)
            k4 = self.dt * self.calc_diff_eqs(self.t + self.dt,
self.y_prev + k3, **self.f_params)

            y_new = self.y_prev + ((k1 + 2*k2 + 2*k3 + k4) /
6.0)
            self.y_prev = y_new
            if self._call_solution_out(self.t, y_new) == -1:
                return self.y_prev
            self.t += self.dt
        return self.y_prev
```

На рис. 2 изображен график разности скоростей барицентра от начальной на каждой итерации. На рис. 3 изображены графики значений полной энергии системы и относительных изменений энергии на каждой итерации. В таблице 1 приведен результат.

Таблица 1

T симуляции (*years*)	Число вызовов *f*	Значение dE/E_0 в конце
4	11688	4.2047990875331473e-10

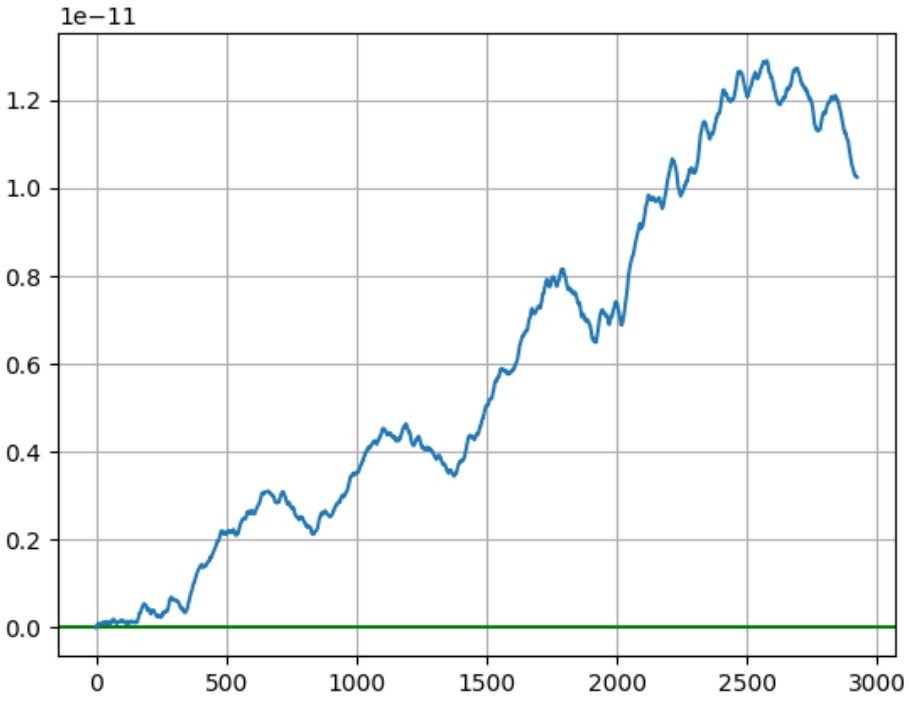

Рисунок 2 – График разности скоростей барицентра от начального для РК4

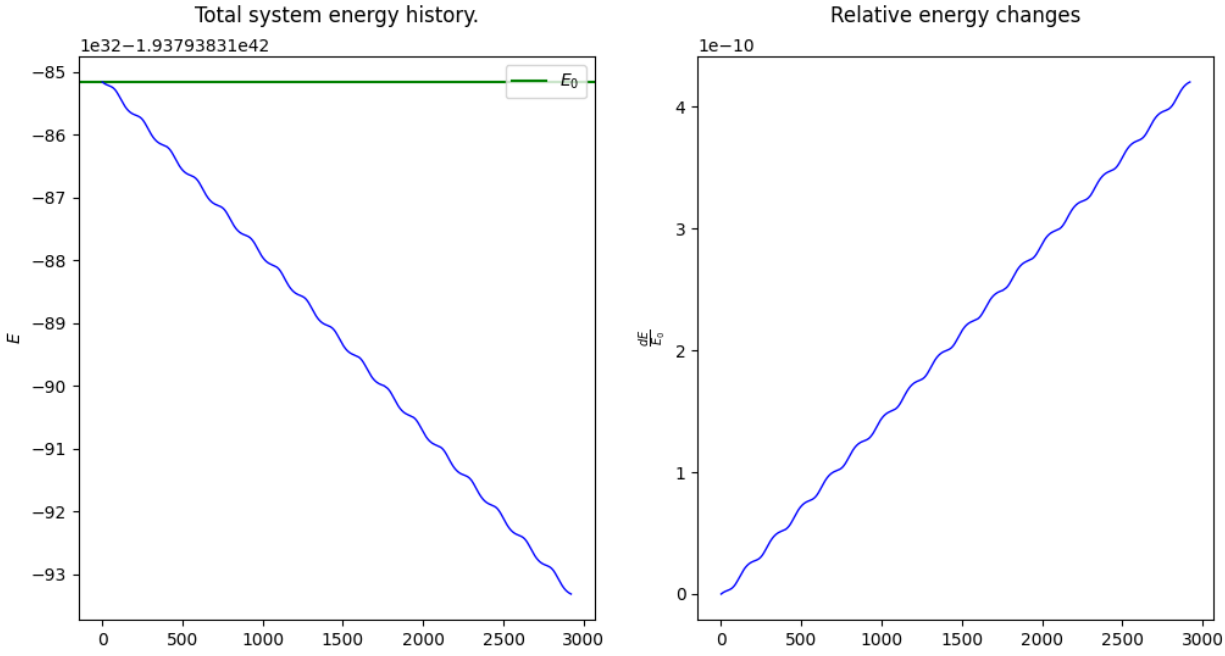

Рисунок 3 – Графики значений полной энергии и относительных изменений для РК4

Пример графического результата (отрисовка орбит) приведен в конце, т.к. визуально результаты разных методов не отличаются.

Был реализован явный одношаговый метод Рунге-Кутты 8-го порядка.

Таблица Бутчера для него имеет следующий вид:

0	0									
4/27	4/27									
2/9	1/18	3/18								
1/3	1/12	0	3/12							
1/2	1/8	0	0	3/8						
2/3	13/54	0	−27/54	42/54	8/54					
1/6	389/4320	0	−54/4320	966/4320	−824/4320	243/4320				
1	−231/20	0	81/20	−1164/20	656/20	−122/20	800/20			
5/6	−127/288	0	18/288	−678/288	456/288	−9/288	576/288	4/288		
1	1481/820	0	−81/820	7104/1481	−3376/820	72/820	−5040/820	−60/820	720/820	
	41/840	0	0	27/840	272/840	27/840	216/840	0	216/840	41/840

Рисунок 4 – Таблица Бутчера для РК8

Код реализации метода приведен ниже:

```
    def rk8_method(self, total_time):
        '''RK8 method. Returns new [y] vector for total_time.'''
        while self.t <= total_time:
            k1 = self.calc_diff_eqs(self.t, self.y_prev,
**self.f_params)
            k2 = self.calc_diff_eqs(self.t + self.dt*(4/27),
self.y_prev+(self.dt*4/27)*k1, **self.f_params)
            k3 = self.calc_diff_eqs(self.t + self.dt*(2/9),
self.y_prev+(self.dt/18)*(k1 + 3*k2), **self.f_params)
            k4 = self.calc_diff_eqs(self.t + self.dt*(1/3),
self.y_prev+(self.dt/12)*(k1+3*k3), **self.f_params)
            k5 = self.calc_diff_eqs(self.t + self.dt*(1/2),
self.y_prev+(self.dt/8)*(k1+3*k4), **self.f_params)
            k6 = self.calc_diff_eqs(self.t + self.dt*(2/3),
self.y_prev+(self.dt/54)*(13*k1-27*k3+42*k4+8*k5),
**self.f_params)
            k7 = self.calc_diff_eqs(self.t + self.dt*(1/6),
self.y_prev+(self.dt/4320)*(389*k1-54*k3+966*k4-824*k5+243*k6),
**self.f_params)
            k8 = self.calc_diff_eqs(self.t + self.dt,
self.y_prev+(self.dt/20)*(-231*k1+81*k3-1164*k4+656*k5-
122*k6+800*k7), **self.f_params)
            k9 = self.calc_diff_eqs(self.t + self.dt*(5/6),
self.y_prev+(self.dt/288)*(-127*k1+18*k3-678*k4+456*k5-
9*k6+576*k7+4*k8), **self.f_params)
            k10 = self.calc_diff_eqs(self.t + self.dt,
self.y_prev+(self.dt/820)*(1481*k1-81*k3+7104*k4-3376*k5+72*k6-
5040*k7-60*k8+720*k9), **self.f_params)

            y_new = self.y_prev +
self.dt/840*(41*k1+27*k4+272*k5+27*k6+216*k7+216*k9+41*k10)
```

```
            self.y_prev = y_new
            if self._call_solution_out(self.t, y_new) == -1:
                return self.y_prev
            self.t += self.dt
        return self.y_prev
```

В таблице 2 приведены результаты.

Таблица 2

T симуляции (*years*)	Число вызовов f	Значение dE/E_0 в конце
4	29220	1.5826136241377246e-14

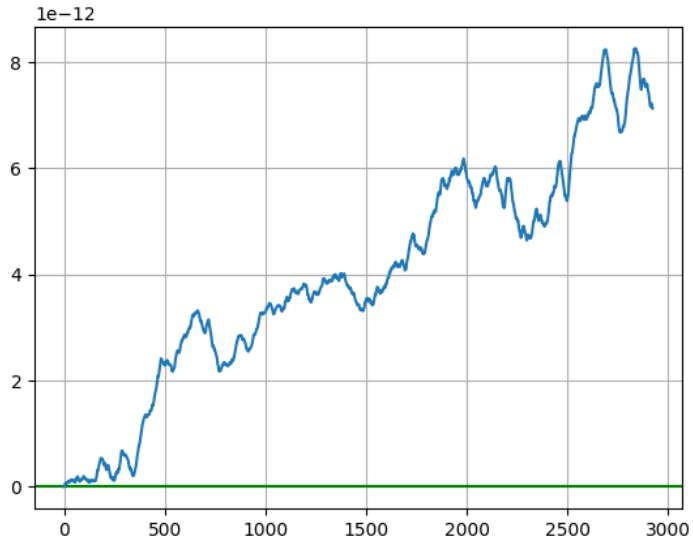

Рисунок 5 – График разности скоростей барицентра от начального для РК8

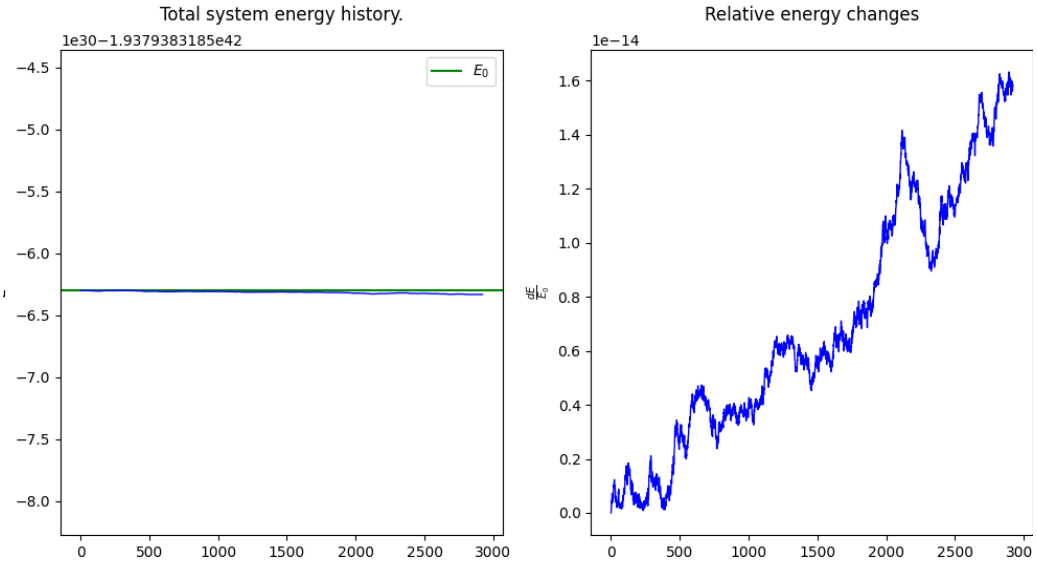

Рисунок 6 – Графики значений полной энергии и относительных изменений для РК8

Был реализован встроенный метод Дормана-Принса порядка 8(7). Таблица Бутчера данного метода показана в приложении. Рассматриваются две схемы:

$$\overline{y}_{k+1} = \overline{y_k} + h(b_1 k_1 + b_2 k_2 + ... + b_s k_s)$$
$$\overline{\hat{y}}_{k+1} = \overline{y_k} + h(\hat{b}_1 k_1 + \hat{b}_2 k_2 + ... + \hat{b}_s k_s)$$

Первый метод имеет порядок p, и является основным (его решение принимают за численное решение уравнений), а второй (вспомогательный) имеет порядок $\hat{p}$, необходим для оценки локальной погрешности.

Шаг в данном методе не является постоянным на всем промежутке интегрирования, а меняется так, чтобы ошибка держалась в требуемых пределах:

$$\left|y_{j,k+1} - \hat{y}_{j,k+1}\right| \le tol_j = Atol_j + \max(\left|y_{j,k+1}\right|, \left|\hat{y}_{j,k+1}\right|) \cdot Rtol_j$$

Абсолютные и относительные допуски $Atol_j, Rtol_j$ задаются пользователем. Получается, что необходимо выполнение следующего условия:

$$E_j = \frac{\left|y_{j,k+1} - \hat{y}_{j,k+1}\right|}{tol_j} \le 1$$

Нужно подобрать шаг так, чтобы данное условие выполнялось:

$$\overline{E} = (E_1, ..., E_n), \left\|E\right\| = err$$
$$y_{j,k+1} \approx y_j(t_{k+1}) + c_j h^{p+1}$$
$$\hat{y}_{j,k+1} \approx y_j(t_{k+1}) + \hat{c}_j h^{\hat{p}+1} \Rightarrow$$
$$err \approx Dh^{\min(p,\hat{p})+1},\ D - const$$
$$1 \approx Dh_{opt}{}^{\min(p,\hat{p})+1} \Rightarrow h_{opt} = h \cdot \left(\frac{1}{err}\right)^{\frac{1}{\min(p,\hat{p})+1}}$$

Иногда в формулу оптимального шага добавляют "смягчающий" множитель:

$$h_{opt} = \alpha \cdot h \cdot \left(\frac{1}{err}\right)^{\frac{1}{\min(p,\hat{p})+1}},\ (*)\ \alpha = 0.7, 0.8, 0.95...$$

Выражения для *err* могут быть разными, одним из оптимальных является следующее:

$$err = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_{j,k+1} - \hat{y}_{j,k+1}}{tol_i}\right)^2}$$

В методе `prince_dormand_87_method()` предусмотрены возможность выбора другого выражения для *err*, задание смягчающего параметра, а также использование метода как неадаптивного.

В итоге, алгоритм управления шага следующий:

1) Вычислить $\overline{y}_{k+1} - \overline{\hat{y}}_{k+1}$ для текущего шага *h*.
2) Вычислить $\overline{E}$ и *err*.
3) Если $err \le 1$, то шаг *h* принять и взять следующий по формуле (*). Иначе шаг отклонить, взять новый по (*) и вернуться к пункту 1.

Фрагмент реализации метода показан ниже. Реализации вспомогательных функций-методов `calc_tol_()`, `calc_err_()`, `calc_err_norm()` приведены в приложении.

```
            _nsteps = 0
        while self.t <= total_time:
            k1 = self.calc_diff_eqs(self.t, self.y_prev,
**self.f_params)
            k2 = self.calc_diff_eqs(self.t+c2*self.dt,
self.y_prev+self.dt*(a21*k1), **self.f_params)
            k3 = self.calc_diff_eqs(self.t+c3*self.dt,
self.y_prev+self.dt*(a31*k1+a32*k2), **self.f_params)
            k4 = self.calc_diff_eqs(self.t+c4*self.dt,
self.y_prev+self.dt*(a41*k1+a43*k3), **self.f_params)
            k5 = self.calc_diff_eqs(self.t+c5*self.dt,
self.y_prev+self.dt*(a51*k1+a53*k3+a54*k4), **self.f_params)
            k6 = self.calc_diff_eqs(self.t+c6*self.dt,
self.y_prev+self.dt*(a61*k1+a64*k4+a65*k5), **self.f_params)
            k7 = self.calc_diff_eqs(self.t+c7*self.dt,
self.y_prev+self.dt*(a71*k1+a74*k4+a75*k5+a76*k6), **self.f_params)
            k8 = self.calc_diff_eqs(self.t+c8*self.dt,
self.y_prev+self.dt*(a81*k1+a84*k4+a85*k5+a86*k6+a87*k7),
**self.f_params)
            k9 = self.calc_diff_eqs(self.t+c9*self.dt,
self.y_prev+self.dt*(a91*k1+a94*k4+a95*k5+a96*k6+a97*k7+a98*k8),
**self.f_params)
            k10 = self.calc_diff_eqs(self.t+c10*self.dt,
self.y_prev+self.dt*(a10_1*k1+a10_4*k4+a10_5*k5+a10_6*k6+a10_7*k7+a10_
8*k8+a10_9*k9), **self.f_params)
            k11 = self.calc_diff_eqs(self.t+c11*self.dt,
self.y_prev+self.dt*(a11_1*k1+a11_4*k4+a11_5*k5+a11_6*k6+a11_7*k7+a11_
8*k8+a11_9*k9+a11_10*k10), **self.f_params)
```

```
            k12 = self.calc_diff_eqs(self.t+c12*self.dt,
self.y_prev+self.dt*(a12_1*k1+a12_4*k4+a12_5*k5+a12_6*k6+a12_7*k7+a12_
8*k8+a12_9*k9+a12_10*k10+a12_11*k11), **self.f_params)
            k13 = self.calc_diff_eqs(self.t+c13*self.dt,
self.y_prev+self.dt*(a13_1*k1+a13_4*k4+a13_5*k5+a13_6*k6+a13_7*k7+a13_
8*k8+a13_9*k9+a13_10*k10+a13_11*k11), **self.f_params)

            y_new = self.y_prev +
self.dt*(b1*k1+b6*k6+b7*k7+b8*k8+b9*k9+b10*k10+b11*k11+b12*k12)
            y_new_ = self.y_prev +
self.dt*(b_1*k1+b_6*k6+b_7*k7+b_8*k8+b_9*k9+b_10*k10+b_11*k11+b_12*k12
+b_13*k13)

            if adaptive:
                tol = calc_tol(y_new, y_new_, a_tol, r_tol,
**calc_tol_kwargs)
                err = calc_err_(tol, y_new, y_new_)
                err_n = calc_err_norm(err, ord)
                prev_t = self.dt

                if err_n == 0.0:
                    err_n = 1e-6
                self.dt = mitig_param * self.dt * (1/err_n)**(1/8)

                if self.dt/prev_t > ifactor:
                    self.dt = prev_t * ifactor
                if prev_t/self.dt > dfactor:
                    self.dt = prev_t/dfactor

                _nsteps += 1
                if _nsteps > nsteps:
                    raise RuntimeError('Limit exceeded of nsteps.')

                if err_n <= 1.000:
                    _nsteps = 0
                    self.t += self.dt
                    self.y_prev = y_new_
                    if self._call_solution_out(self.t, y_new) == -1:
                        return self.y_prev
            else:
                self.t += self.dt
                self.y_prev = y_new_
                if self._call_solution_out(self.t, y_new) == -1:
                    return self.y_prev
        return self.y_prev
```

Результаты аналогичного запуска приведены в таблице 3.

Таблица 3

T симуляции (*years*)	Число вызовов *f*	Значение dE/E_0 в конце
4	7107	5.628562406190808e-13

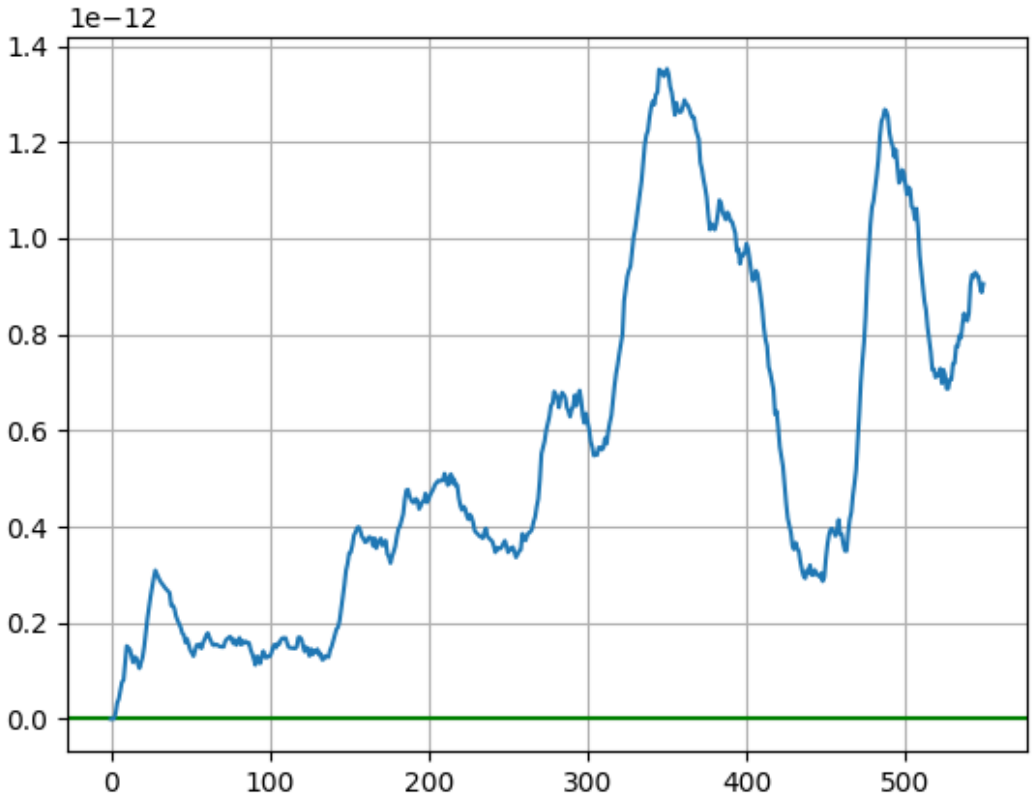

Рисунок 7 – График разности скоростей барицентра от начального для ДП8(7)

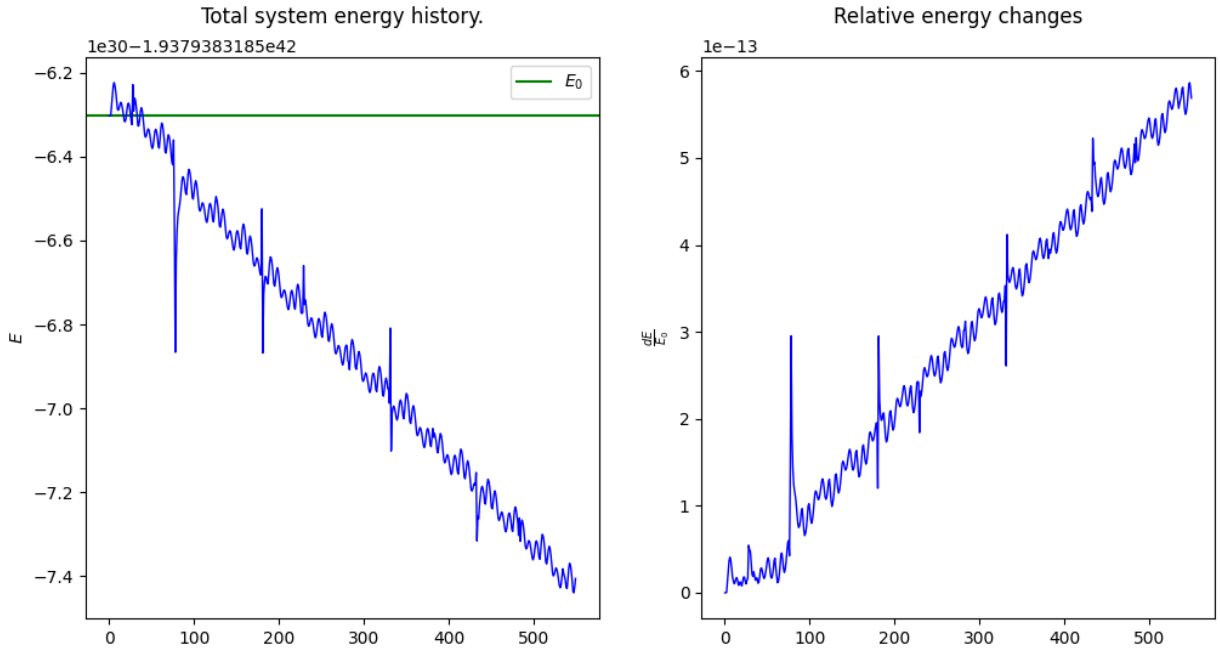

Рисунок 8 – Графики значений полной энергии и относительных изменений для ДП8(7)

Было проведено сравнение числа вызовов функции f для каждого метода. Время полного моделирования $T = 40(лет)$. При этом проводились варьирования значений шага и параметров методов и замерялось количество вызовов функции *f*. Будем считать лучшим тот метод, который при меньшем числе вызовов привел к меньшим потерям. Сравнительный график относительных изменений полной энергии системы в конце моделирования в зависимости от числа вызовов функции f для разных методов приведен на рис. 10. Результаты для каждого метода в отдельности приведены в таблицах 4-7.

Таблица 4 – Результаты для ДП8(7)

Число вызовов f	Значение dE/E_0 в конце симуляции
36646	1.8135492375766347e-09
51045	3.274439672716814e-11
89843	4.397983554912578e-13
123370	1.876355536409898e-14
149279	2.040151266652787e-15

Таблица 5 – Результаты для РК4

Число вызовов f	Значение dE/E_0 в конце симуляции
29220	4.383417130418988e-06
58440	1.3511477564635645e-07
140256	1.6936688095536425e-09
467520	4.131435566365387e-12
935040	8.372951912866681e-14
1402560	1.0679349060574833e-15

Таблица 6 – Результаты для РК8

Число вызовов f	Значение dE/E_0 в конце симуляции
48700	7.548337352586976e-10

73050	4.2926348403976076e-11
146100	3.3711997329430785e-13
292200	2.6316130191565197e-14
350640	3.3821241171788843e-15

Таблица 7 – Результаты для ДП8(7) (неадаптивный)

Число вызовов f	Значение dE/E_0 в конце симуляции
35607	2.4548218022381444e-09
63310	1.5855497402919484e-11
89659	7.826067617405032e-13
142441	1.644540123188448e-14
189930	8.16267373766118e-15
455832	6.494238507512885e-15

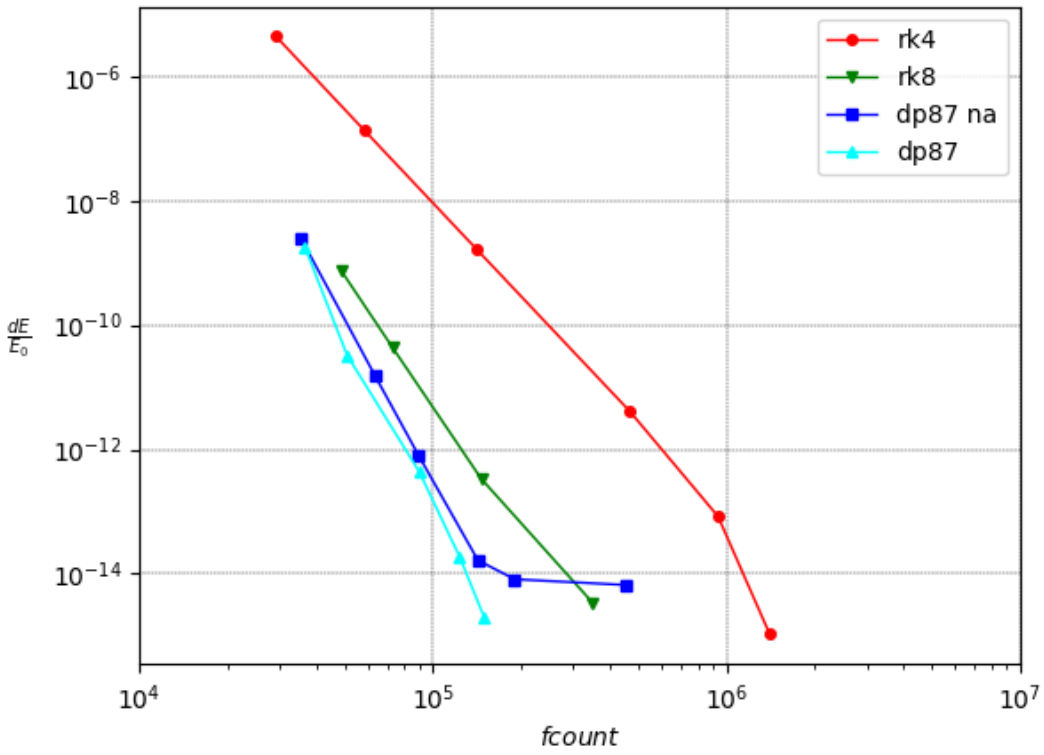

Рисунок 9 – Относительное изменение энергии в конце моделирования в зависимости от количества вызовов f для разных методов

Таблица 8 – Сравнительная таблица методов

Метод	Число вызовов f	Значение dE/E_0 в конце симуляции
РК4	935040	8.372951912866681e-14
РК8	292200	2.6316130191565197e-14
ДП8(7) н	142441	1.644540123188448e-14
ДП8(7)	123370	1.876355536409898e-14

В данном случае вложенный метод Дормана-Принса 8(7) оказался лучшим, т.к. он потребовал меньшее число вызовов функции f для достижения приемлемого значения ошибки. Неадаптивный метод Дормана-Принса является альтернативным вариантом метода РК8, однако он дает результаты лучше (вероятно за счет лучшего подбора коэффициентов таблицы), чем "стандартный" метод Рунге-Кутта 8 порядка. Метод Рунге-Кутта 4 порядка оказался самым медленным среди всех реализованных методов, он требует значительно большее число вызовов функции f для достижения такого же качества ошибки.

На рисунках ниже приведены примеры графического отображения орбит.

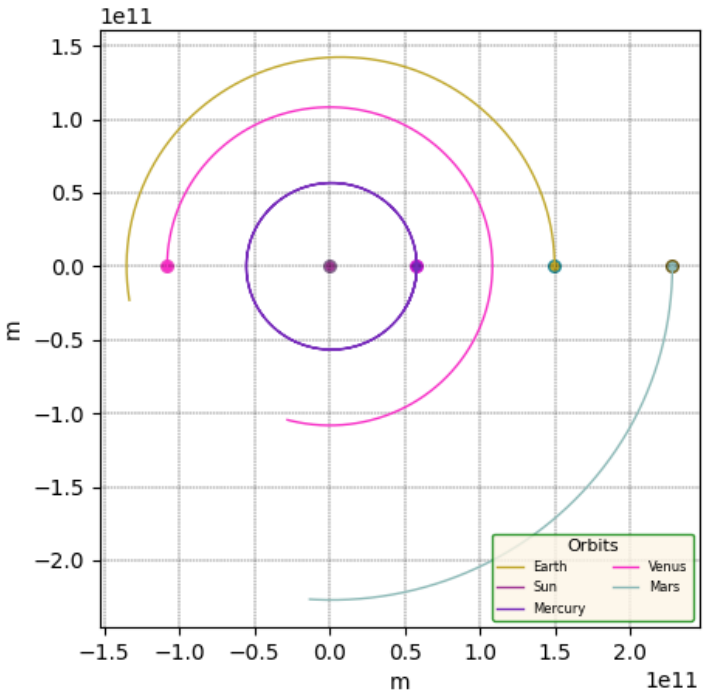

Рисунок 11

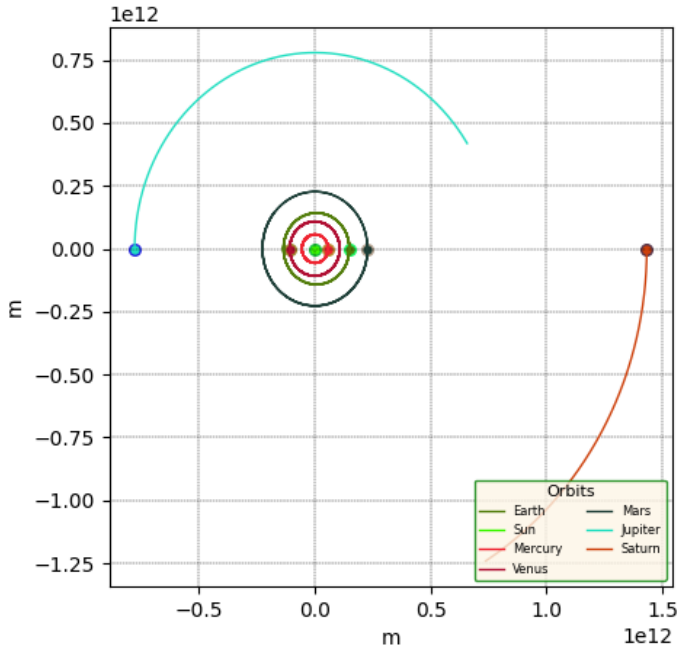

Рисунок 12

Graphical results

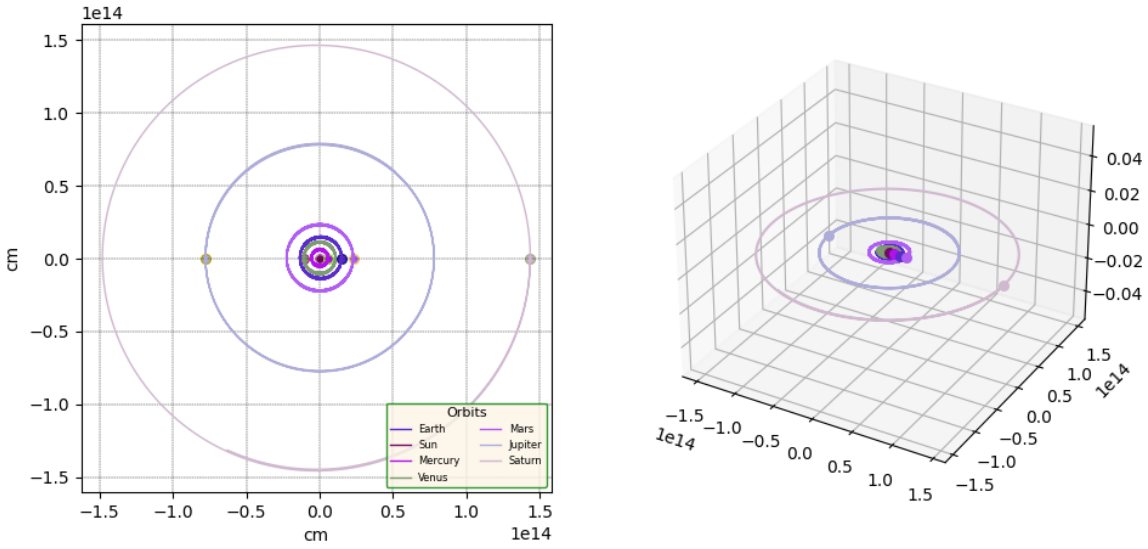

Рисунок 13

Выводы

В ходе выполнения работы была реализована симуляция движения материальных точек под действием гравитационных сил. Были реализованы 3 численных метода решения дифференциальных уравнений: РК4, РК8 и метод Дормана-Принса 8(7). Было проведено краткое сравнение работы методов, в результате чего был сделан вывод, что самым оптимальным является вложенный метод, т.к. он дает достаточно точные результаты и не требует значительного числа вызовов функции *f*, в отличие от методов с постоянным шагом.

Список источников

1) Mathematical theory of the Goddard trajectory determination system, Cappellari, J. O., Velez, C. E., Fuchs, A. J., 1976, p. 288.
2) https://www.maths.ed.ac.uk/~heggie/lecture1.pdf
3) Рой А. Движение по орбитам. 1981., 544 с.
4) https://github.com/the21composer/Dynamic-systems-modeling-course

Приложение

Таблица Бутчера для метода Дорманда-Принса:

c_i	a_{ij}												$\hat{b}_i$	b_i
0													14005451 335480064	13451932 455176623
$\frac{1}{18}$	$\frac{1}{18}$												0	0
$\frac{1}{12}$	$\frac{1}{48}$	$\frac{1}{16}$											0	0
$\frac{1}{8}$	$\frac{1}{32}$	0	$\frac{3}{32}$										0	0
$\frac{5}{16}$	$\frac{5}{16}$	0	$\frac{-75}{64}$	$\frac{75}{64}$									0	0
$\frac{3}{8}$	$\frac{3}{80}$	0	0	$\frac{3}{16}$	$\frac{3}{20}$								$\frac{-59238493}{1068277825}$	$\frac{-808719846}{976000145}$
$\frac{59}{400}$	$\frac{29443841}{614563906}$	0	0	$\frac{77736538}{692538347}$	$\frac{-28693883}{1125000000}$	$\frac{23124283}{1800000000}$							$\frac{181606767}{758867731}$	$\frac{1757004468}{5645159321}$
$\frac{93}{200}$	$\frac{16016141}{946692911}$	0	0	$\frac{61564180}{158732637}$	$\frac{22789713}{633445777}$	$\frac{545815736}{2771057229}$	$\frac{-180193667}{1043307555}$						$\frac{561292985}{797845732}$	$\frac{656045339}{265891186}$
$\frac{5490023248}{9719169821}$	$\frac{39632708}{573591083}$	0	0	$\frac{-433636366}{683701615}$	$\frac{-421739975}{2616292301}$	$\frac{100302831}{723423059}$	$\frac{790204164}{839813087}$	$\frac{800635310}{3783071287}$					$\frac{-1041891430}{1371343529}$	$\frac{-3867574721}{1518517206}$
$\frac{13}{20}$	$\frac{246121993}{1340847787}$	0	0	$\frac{-37695042795}{15268766246}$	$\frac{-309121744}{1061227803}$	$\frac{-12992083}{490766935}$	$\frac{6005943493}{2108947869}$	$\frac{393006217}{1396673457}$	$\frac{123872331}{1001029789}$				$\frac{760417239}{1151165299}$	$\frac{465885868}{322736535}$
$\frac{1201146811}{1299019798}$	$\frac{-1028468189}{846180014}$	0	0	$\frac{8478235783}{508512852}$	$\frac{1311729495}{1432422823}$	$\frac{-10304129995}{1701304382}$	$\frac{-48777925059}{3047939560}$	$\frac{15336726248}{1032824649}$	$\frac{-45442868181}{339846769 6}$	$\frac{3065993473}{597172653}$			$\frac{118820643}{751138087}$	$\frac{53011238}{667516719}$
1	$\frac{185892177}{718116043}$	0	0	$\frac{-3185094517}{667107341}$	$\frac{-477755414}{1098053517}$	$\frac{-703635378}{230739211}$	$\frac{5731566787}{1027545527}$	$\frac{5232866602}{850066563}$	$\frac{-4093664535}{80868825 7}$	$\frac{3962137247}{1805957418}$	$\frac{65686358}{487910083}$		$\frac{-528747749}{2220607170}$	$\frac{2}{45}$
1	$\frac{403863854}{491063109}$	0	0	$\frac{-5068492393}{434740067}$	$\frac{-411421997}{543043805}$	$\frac{652783627}{914296604}$	$\frac{11173962825}{925320556}$	$\frac{-13158990841}{6184727034}$	$\frac{3936647629}{197804968 0}$	$\frac{-160528059}{685178525}$	$\frac{248638103}{1413531060}$	0	$\frac{1}{4}$	0

Реализация класса ***Body***:

```
class Body():
    '''
    Body class for particle, used in Simulaion
    ------------------------------------------
    Params:
        mass: mass of particle.
        x_vec: vector len(3) containing the x, y, z positions.
        v_vec: vector len(3) containing the v_x, v_y, v_z velocities.
        name: string, name of body.
        has_units: type of dimension?

    Example of using:
        Mars = Body(name='Mars', x_vec=mars_x, v_vec=mars_v,
mass=mars_mass)
    '''
    def __init__(self, mass, x_vec, v_vec, name='Unknown',
has_units=None):
        self.name = name
        self.has_units = has_units
        if self.has_units and self.has_units.upper() == 'CGS':
            self.mass = mass.cgs
            self.x_vec = x_vec.cgs.value
            self.v_vec = v_vec.cgs.value
        elif self.has_units and self.has_units.upper() == 'SI':
            self.mass = mass.si
            self.x_vec = x_vec.si.value
            self.v_vec = v_vec.si.value
        else:
            self.mass = mass
            self.x_vec = x_vec
            self.v_vec = v_vec

    def return_vec(self):
        '''
        return concatenates x and v into "y" vector of positions and
velocities
        '''
        return np.concatenate((self.x_vec, self.v_vec))

    def return_mass(self):
        if self.has_units and self.has_units.upper() == 'CGS':
            return self.mass.cgs.value
        elif self.has_units and self.has_units.upper() == 'SI':
            return self.mass.si.value
        else:
            return self.mass

    def return_name(self):
        return self.name
```

Реализация класса ***OdeIntegrator***:

```
class OdeIntegrator():

    def __init__(self, f=None):
        self.calc_diff_eqs = f
        self.dt = None
        self.t = None
        self.f_params = {}
        self.method_params = {}
        self.y_prev = None
        self.solution_out = None

    def set_solution_out(self, sol_out):
        '''Set callback, which be called at every step.'''
        self.solution_out = sol_out
        return self

    def set_f_params(self, **kwargs):
        '''Set any additional hyperparameters for function f.'''
        self.f_params = kwargs
        return self

    def add_f_params(self, **kwargs):
        '''Add any additional hyperparameters for function f.'''
        self.f_params.update(kwargs)
        return self

    def set_f(self, f, **kwargs):
        '''
        Assigns an external solver function as the diff-eq solver for method.
        ---------------------------------
        Params:
            f: function which returns a [y] vector for method.
            **kwargs: Any additional hyperparameters.
        '''
        if kwargs:
            self.f_params = kwargs
        self.calc_diff_eqs = f
        return self

    def set_method_params(self, **kwargs):
        '''Set any additional hyperparameters for numeric method.'''
        self.method_params = kwargs
        return self

    def add_method_params(self, **kwargs):
        '''Add any additional hyperparameters for numeric method.'''
        self.method_params.update(kwargs)
        return self

    def set_init_params(self, y, t=0.0, dt=0.001):
        '''Set initial parameters: y, t, dt.'''
        self.dt = dt
        if np.isscalar(y):
            y = [y]
        self.y_prev = np.array(y)
        self.t = t
        return self

    def set_integrator(self, method_name, **kwargs):
        '''
        Assigns a numeric method to solve diff-eq
```

```
        ---------------------------------
        Params:
            method_name: string name of method.
            **kwargs: Any additional hyperparameters.
        '''
        if kwargs:
            self.method_params = kwargs

        if method_name.upper() == 'RK4':
            self.curr_num_method = self.rk4_method
        elif method_name.upper() == 'RK8':
            self.curr_num_method = self.rk8_method
        elif method_name.upper() == 'PD87' or method_name.upper() == 'DP87':
            self.curr_num_method = self.prince_dormand_87_method
        else:
            raise AttributeError('Not find method {}.'.format(method_name))

    def integrate(self, total_time):
        '''Using the numerical method, find y(total_time).'''
        if not np.isscalar(total_time):
            raise ValueError('Need parametr "t".')
        if not hasattr(self, 'curr_num_method') or not self.curr_num_method:
            raise ValueError('You need to set method. Use function
"set_integrator".')
        if not hasattr(self, 'calc_diff_eqs') or not self.calc_diff_eqs:
            raise ValueError('You need to set f function. Use "set_f".')

        self._check_init_params()

        if total_time < self.t:
            return self.y_prev

        if self.dt > (total_time - self.t):
            self.dt = total_time - self.t
        self.t += self.dt
        return self.curr_num_method(total_time)

    def _check_init_params(self):
        if self.y_prev is None or not hasattr(self, 'y_prev'):
            raise ValueError('You need to set start param "y0"!')
        if self.dt is None or not hasattr(self, 'dt'):
            raise ValueError('You need to set start param "dt"!')

    def _call_solution_out(self, t, y):
        ret = 0
        if self.solution_out:
            ret = self.solution_out(t, y)
        return ret

    def rk4_method(self, total_time):
        '''RK4 method. Returns new [y] vector for total_time.'''
        while self.t <= total_time:
            k1 = self.dt * self.calc_diff_eqs(self.t, self.y_prev,
**self.f_params)
            k2 = self.dt * self.calc_diff_eqs(self.t + 0.5*self.dt,
self.y_prev + 0.5*k1, **self.f_params)
            k3 = self.dt * self.calc_diff_eqs(self.t + 0.5*self.dt,
self.y_prev + 0.5*k2, **self.f_params)
            k4 = self.dt * self.calc_diff_eqs(self.t + self.dt, self.y_prev +
k3, **self.f_params)

            y_new = self.y_prev + ((k1 + 2*k2 + 2*k3 + k4) / 6.0)
            self.y_prev = y_new
```

```
            if self._call_solution_out(self.t, y_new) == -1:
                return self.y_prev
            self.t += self.dt
        return self.y_prev

    def rk8_method(self, total_time):
        '''RK8 method. Returns new [y] vector for total_time.'''
        while self.t <= total_time:
            k1 = self.calc_diff_eqs(self.t, self.y_prev, **self.f_params)
            k2 = self.calc_diff_eqs(self.t + self.dt*(4/27),
self.y_prev+(self.dt*4/27)*k1, **self.f_params)
            k3 = self.calc_diff_eqs(self.t + self.dt*(2/9),
self.y_prev+(self.dt/18)*(k1 + 3*k2), **self.f_params)
            k4 = self.calc_diff_eqs(self.t + self.dt*(1/3),
self.y_prev+(self.dt/12)*(k1+3*k3), **self.f_params)
            k5 = self.calc_diff_eqs(self.t + self.dt*(1/2),
self.y_prev+(self.dt/8)*(k1+3*k4), **self.f_params)
            k6 = self.calc_diff_eqs(self.t + self.dt*(2/3),
self.y_prev+(self.dt/54)*(13*k1-27*k3+42*k4+8*k5), **self.f_params)
            k7 = self.calc_diff_eqs(self.t + self.dt*(1/6),
self.y_prev+(self.dt/4320)*(389*k1-54*k3+966*k4-824*k5+243*k6),
**self.f_params)
            k8 = self.calc_diff_eqs(self.t + self.dt,
self.y_prev+(self.dt/20)*(-231*k1+81*k3-1164*k4+656*k5-122*k6+800*k7),
**self.f_params)
            k9 = self.calc_diff_eqs(self.t + self.dt*(5/6),
self.y_prev+(self.dt/288)*(-127*k1+18*k3-678*k4+456*k5-9*k6+576*k7+4*k8),
**self.f_params)
            k10 = self.calc_diff_eqs(self.t + self.dt,
self.y_prev+(self.dt/820)*(1481*k1-81*k3+7104*k4-3376*k5+72*k6-5040*k7-
60*k8+720*k9), **self.f_params)

            y_new = self.y_prev +
self.dt/840*(41*k1+27*k4+272*k5+27*k6+216*k7+216*k9+41*k10)
            self.y_prev = y_new
            if self._call_solution_out(self.t, y_new) == -1:
                return self.y_prev
            self.t += self.dt
        return self.y_prev

    def prince_dormand_87_method(self, total_time):
        '''
        DOPRI 8(7) method. Explicit runge-kutta method with stepsize control.
        Method accepts the following hyperparameters:
            - atol : absolute tolerance for solution
            - rtol : relative tolerance for solution
            - mitig_param: "softening" factor on new step selection
            - ord: order of the norm (type of norm) in calc_err_norm().
            - calc_tol: can be set by the user
            - calc_tol_params: hyperparameters for calc_tol()
        '''
        c2=1/18;                    a21=1/18
        c3=1/12;                    a31=1/48;                    a32=1/16
        c4=1/8;                     a41=1/32;                    a43=3/32
        c5=5/16;                    a51=5/16;                    a53=-75/64;
a54=75/64
        c6=3/8;                     a61=3/80;                    a64=3/16;
a65=3/20
        c7=59/400;                  a71=29443841/614563906;
a74=77736538/692538347;         a75=-28693883/1125000000;
a76=23124283/1800000000
        c8=93/200;                  a81=16016141/946692911;
a84=61564180/158732637;         a85=22789713/633445777;
```

```
a86=545815736/2771057229;      a87=-180193667/1043307555
        c9=5490023248/9719169821;  a91=39632708/573591083;      a94=-
433636366/683701615;       a95=-421739975/2616292301;
a96=100302831/723423059;       a97=790204164/839813087;
a98=800635310/3783071287
        c10=13/20;                  a10_1=246121993/1340847787;  a10_4=-
37695042795/15268766246; a10_5=-309121744/1061227803; a10_6=-
12992083/490766935;     a10_7=6005943493/2108947869;
a10_8=393006217/1396673457;    a10_9=123872331/1001029789
        c11=1201146811/1299019798; a11_1=-1028468189/846180014;
a11_4=8478235783/508512852;     a11_5=1311729495/1432422823; a11_6=-
10304129995/1701304382; a11_7=-48777925059/3047939560;
a11_8=15336726248/1032824649;  a11_9=-45442868181/3398467696;
a11_10=3065993473/597172653
        c12=1.0;                    a12_1=185892177/718116043;   a12_4=-
3185094517/667107341;    a12_5=-477755414/1098053517; a12_6=-
703635378/230739211;    a12_7=5731566787/1027545527;
a12_8=5232866602/850066563;    a12_9=-4093664535/808688257;
a12_10=3962137247/1805957418; a12_11=65686358/487910083
        c13=1.0;                    a13_1=403863854/491063109;   a13_4=-
5068492393/434740067;    a13_5=-411421997/543043805;
a13_6=652783627/914296604;     a13_7=11173962825/925320556;   a13_8=-
13158990841/6184727034; a13_9=3936647629/1978049680;   a13_10=-
160528059/685178525;  a13_11=248638103/1413531060

        b1=13451932/455176623; b6=-808719846/976000145;
b7=1757004468/5645159321; b8=656045339/265891186; b9=-3867574721/1518517206;
b10=465885868/322736535; b11=53011238/667516719; b12=2/45
        b_1=14005451/335480064; b_6=-59238493/1068277825;
b_7=181606767/758867731; b_8=561292985/797845732; b_9=-1041891430/1371343529;
b_10=760417239/1151165299; b_11=118820643/751138087; b_12=-
528747749/2220607170; b_13=1/4


        adaptive = self.method_params['adapt'] if 'adapt' in
self.method_params else False

        if adaptive:
            if 'atol' in self.method_params:
                a_tol = self.method_params['atol']
            else:
                raise AttributeError('You need to set "atol" param for PD8(7)
method!')

            if 'rtol' in self.method_params:
                r_tol = self.method_params['rtol']
            else:
                raise AttributeError('You need to set "rtol" param for PD8(7)
method!')

            if 'mitig_param' in self.method_params:
                mitig_param = self.method_params['mitig_param']
            else:
                mitig_param = 1.0

            if 'ord' in self.method_params:
                ord = self.method_params['ord']
            else:
                ord = None

            if 'ifactor' in self.method_params:
                ifactor = self.method_params['ifactor']
            else:
```

```
                ifactor = 10.0
            if 'dfactor' in self.method_params:
                dfactor = self.method_params['dfactor']
            else:
                dfactor = 10.0

            if 'nsteps' in self.method_params:
                nsteps = self.method_params['nsteps']
            else:
                nsteps = 1e3

            if 'calc_tol' in self.method_params:
                calc_tol = self.method_params['calc_tol']
                calc_tol_kwargs = self.method_params['calc_tol_params'] if
'calc_tol_params' in self.method_params else ()
            else:
                calc_tol = calc_tol_

        _nsteps = 0

        while self.t <= total_time:
            k1 = self.calc_diff_eqs(self.t, self.y_prev, **self.f_params)
            k2 = self.calc_diff_eqs(self.t+c2*self.dt,
self.y_prev+self.dt*(a21*k1), **self.f_params)
            k3 = self.calc_diff_eqs(self.t+c3*self.dt,
self.y_prev+self.dt*(a31*k1+a32*k2), **self.f_params)
            k4 = self.calc_diff_eqs(self.t+c4*self.dt,
self.y_prev+self.dt*(a41*k1+a43*k3), **self.f_params)
            k5 = self.calc_diff_eqs(self.t+c5*self.dt,
self.y_prev+self.dt*(a51*k1+a53*k3+a54*k4), **self.f_params)
            k6 = self.calc_diff_eqs(self.t+c6*self.dt,
self.y_prev+self.dt*(a61*k1+a64*k4+a65*k5), **self.f_params)
            k7 = self.calc_diff_eqs(self.t+c7*self.dt,
self.y_prev+self.dt*(a71*k1+a74*k4+a75*k5+a76*k6), **self.f_params)
            k8 = self.calc_diff_eqs(self.t+c8*self.dt,
self.y_prev+self.dt*(a81*k1+a84*k4+a85*k5+a86*k6+a87*k7), **self.f_params)
            k9 = self.calc_diff_eqs(self.t+c9*self.dt,
self.y_prev+self.dt*(a91*k1+a94*k4+a95*k5+a96*k6+a97*k7+a98*k8),
**self.f_params)
            k10 = self.calc_diff_eqs(self.t+c10*self.dt,
self.y_prev+self.dt*(a10_1*k1+a10_4*k4+a10_5*k5+a10_6*k6+a10_7*k7+a10_8*k8+a1
0_9*k9), **self.f_params)
            k11 = self.calc_diff_eqs(self.t+c11*self.dt,
self.y_prev+self.dt*(a11_1*k1+a11_4*k4+a11_5*k5+a11_6*k6+a11_7*k7+a11_8*k8+a1
1_9*k9+a11_10*k10), **self.f_params)
            k12 = self.calc_diff_eqs(self.t+c12*self.dt,
self.y_prev+self.dt*(a12_1*k1+a12_4*k4+a12_5*k5+a12_6*k6+a12_7*k7+a12_8*k8+a1
2_9*k9+a12_10*k10+a12_11*k11), **self.f_params)
            k13 = self.calc_diff_eqs(self.t+c13*self.dt,
self.y_prev+self.dt*(a13_1*k1+a13_4*k4+a13_5*k5+a13_6*k6+a13_7*k7+a13_8*k8+a1
3_9*k9+a13_10*k10+a13_11*k11), **self.f_params)

            y_new = self.y_prev +
self.dt*(b1*k1+b6*k6+b7*k7+b8*k8+b9*k9+b10*k10+b11*k11+b12*k12)
            y_new_ = self.y_prev +
self.dt*(b_1*k1+b_6*k6+b_7*k7+b_8*k8+b_9*k9+b_10*k10+b_11*k11+b_12*k12+b_13*k
13)

            if adaptive:
                tol = calc_tol(y_new, y_new_, a_tol, r_tol,
**calc_tol_kwargs)
                err = calc_err_(tol, y_new, y_new_)
                err_n = calc_err_norm(err, ord)
```

```
            prev_t = self.dt

            if err_n == 0.0:
                err_n = 1e-6
            self.dt = mitig_param * self.dt * (1/err_n)**(1/8)

            if self.dt/prev_t > ifactor:
                self.dt = prev_t * ifactor
            if prev_t/self.dt > dfactor:
                self.dt = prev_t/dfactor

            _nsteps += 1
            if _nsteps > nsteps:
                raise RuntimeError('Limit exceeded of nsteps.')

            if err_n <= 1.000:
                _nsteps = 0
                self.t += self.dt
                self.y_prev = y_new_
                if self._call_solution_out(self.t, y_new) == -1:
                    return self.y_prev
        else:
            self.t += self.dt
            self.y_prev = y_new_
            if self._call_solution_out(self.t, y_new) == -1:
                return self.y_prev
    return self.y_prev
```

Реализация класса ***Simulation***:

```
class Simulation():
    '''
    Simulation object.
    -----------------
    Params:
        bodies: list of Body() objects.
        has_units: type of bodies dimension.
        num_bodies: number of particles.
        _n_dim: len(y) for 'y' vector for 1 particle.
        quant_vec: vector of 'y' vectors for every body.
        mass_vec: vector of masses for all bodies.
        name_vec: vector of names for all bodies.
    '''
    def __init__(self, bodies, has_units=None):
        self.has_units = has_units
        self.bodies = bodies
        self.num_bodies = len(self.bodies)
        self._n_dim = 6
        self.quant_vec = np.concatenate(np.array([elem.return_vec() for elem
in self.bodies]))
        self.mass_vec = np.array([elem.return_mass() for elem in
self.bodies])
        self.name_vec = [elem.return_name() for elem in self.bodies]
        self.system_mass = np.sum(self.mass_vec)
        self.ode_integrator = OdeIntegrator()

    def set_ode_integator(self, integrator):
        if not isinstance(integrator, OdeIntegrator):
            raise ValueError('Error type of integrator.')
        self.ode_integrator = integrator
        return self

    def set_diff_eqs(self, calc_diff_eqs, **kwargs):
        self.ode_integrator.set_f(calc_diff_eqs, **kwargs)

    def set_numeric_method(self, method_name, **kwargs):
        self.ode_integrator.set_integrator(method_name, **kwargs)

    def run_simulation(self, total_time, dt, t_0=0, G=None, logging=False):
        '''
        Method for running simulation.
        ------------------------------
        Params:
            total_time: total time for simulation.
            dt: timestep.
            t_0: start time.
        '''

        if G is None:
            if self.has_units.upper() == 'CGS':
                G = astr_const.G.cgs.value
            elif self.has_units.upper() == 'SI':
                G = astr_const.G.si.value
            else:
                raise TypeError('You need to set value of "G" if you use
dimensionless system.')
        else:
            if self.has_units:
                try:
                    _ = G.unit
                except:
```

```
                    G = (G * astr_const.G.unit)
                G = G.cgs.value if self.has_units.upper() == 'CGS' else
G.si.value
        self.G = G

        if self.has_units:
            try:
                _ = t_0.unit
            except:
                t_0 = (total_time.unit * t_0)#.cgs.value
            if self.has_units.upper() == 'CGS':
                t_0 = t_0.cgs.value
                total_time = total_time.cgs.value
                dt = dt.cgs.value
            elif self.has_units.upper() == 'SI':
                t_0 = t_0.si.value
                total_time = total_time.si.value
                dt = dt.si.value
            else:
                raise TypeError('Unintended unit system type
{}!'.format(self.has_units))

        self.num_diff_eq_calls = 0
        self.quant_vec = np.concatenate(np.array([elem.return_vec() for elem
in self.bodies]))
        self.history = [self.quant_vec]
        self.ode_integrator.add_f_params(G=G, num_calls=[0],
masses=self.mass_vec)
        self.ode_integrator.set_init_params(self.quant_vec, t_0, dt)
        self.ode_integrator.add_method_params(calc_tol=calc_tol_n,
calc_tol_params={'nbody':self.num_bodies})
        def solout(t,y):
            self.history.append(y)
            self.quant_vec = y
        self.ode_integrator.set_solution_out(solout)
        start_time = time.time()

        self.ode_integrator.integrate(total_time)

        end_time = time.time() - start_time
        print('Simulation passed in {} seconds'.format(end_time))
        self.history = np.array(self.history)
        self.num_diff_eq_calls = self.ode_integrator.f_params['num_calls'][0]

    def get_num_calls_diff_eq(self):
        if hasattr(self, 'num_diff_eq_calls'):
            return self.num_diff_eq_calls
        else:
            return None

    def _calc_barycent_v(self, index):
        barycent_v = np.zeros(3)
        for i in range(self.num_bodies):
            offset = i * 6
            barycent_v += self.mass_vec[i] *
self.history[index][offset+3:offset+6]
        return barycent_v / self.system_mass

    def plot_barycent_v_dist_history(self, barycent_v0, barycent_v_history,
need_save_plt=False, save_plt_name=None, smooth=True):
        dist_hist = ([np.linalg.norm(barycent_v0-barycent_v_history[i]) for i
in range(len(barycent_v_history))])
        plt.axhline(y=0, xmin=0, xmax=len(barycent_v_history), color
```

```
="green")
        if smooth:
            plt.plot(np.arange(0, len(barycent_v_history), 1),
smooth_graph_points(dist_hist))
        else:
            plt.plot(np.arange(0, len(barycent_v_history), 1), dist_hist)
        plt.grid()
        if need_save_plt:
            if save_plt_name:
                plt.savefig(save_plt_name)
            else:
                now = datetime.datetime.now()
                plt.savefig(r"barycent_history"+now.strftime("%d-%m-%Y_%H-%M-
%S")+".png")
        plt.show()

    def check_barycent_inv(self, need_plot=True, need_save_plt=False,
save_plt_name=None, smooth=True):
        if not hasattr(self, 'history'):
            raise AttributeError('Missing attribute "history", maybe need run
simulation?')
        barycent_v0 = self._calc_barycent_v(0)
        barycent_v_history = []
        for i in range(len(self.history)):
            barycent_v_history.append(self._calc_barycent_v(i))
        if need_plot:
            self.plot_barycent_v_dist_history(barycent_v0,
np.array(barycent_v_history), need_save_plt, save_plt_name, smooth)
        return np.array(barycent_v_history)


    def _calc_barycent_r(self, index):
        barycent_r = np.zeros(3)
        for i in range(self.num_bodies):
            offset = i * 6
            barycent_r += self.mass_vec[i] *
self.history[index][offset:offset+3]
        return barycent_r / self.system_mass

    def get_barycent_r_history(self):
        if not hasattr(self, 'history'):
            raise AttributeError('Missing attribute "history", maybe need run
simulation?')
        barycent_r_history = []
        for i in range(len(self.history)):
            barycent_r_history.append(self._calc_barycent_r(i))
        return np.array(barycent_r_history)


    def check_energy_inv(self, need_smooth=True, factor=0.5, title_pad=20):
        if not hasattr(self, 'history'):
            raise AttributeError('Missing attribute "history", maybe need run
simulation?')
        T_history, U_history, E_history = [], [], []
        for i in range(len(self.history)):
            T_history.append(self._calc_k_energy(self.history[i]))
            U_history.append(self._calc_p_energy(self.history[i]))
            E_history.append(T_history[i] - U_history[i])
        E_history = np.array(E_history)

        E0 = E_history[0]
        rel_ch = np.array(np.abs(E_history-E0)/np.abs(E0))
```

```
        if need_smooth:
            E_history = smooth_graph_points(E_history, factor=factor)
            rel_ch = smooth_graph_points(rel_ch, factor=factor)
        fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(12, 6))
        #ax1.axhline(y=E0, xmin=0, xmax=len(self.history), color ="green")
        ax1.plot(range(0, len(self.history), 1), E_history, color="blue",
linestyle='-', linewidth=1)
        ax1.set_ylabel(r'$E$')
        ax1.set_title('Total system energy history.', pad=title_pad)

        ax2.plot(range(0, len(self.history), 1), rel_ch, color="blue",
linestyle='-', linewidth=1)
        ax2.set_ylabel(r'$\frac{dE}{E_0}$')
        ax2.set_title('Relative energy changes', pad=title_pad)
        plt.show()
        return E_history, E0, rel_ch[-1]


    def _calc_k_energy(self, y_vec):
        res = []
        for i in range(self.num_bodies):
            offset = i * 6
            v_vec = y_vec[offset+3:offset+6]
            res.append(self.mass_vec[i] * np.dot(v_vec,v_vec))
        res = kahan_sum(np.array(res))
        return res / 2.0

    def _calc_p_energy(self, y_vec):
        res_p = []
        for i in range(self.num_bodies):
            ioffset = i * 6
            res = 0
            for j in range(self.num_bodies):
                joffset = j * 6
                if i != j:
                    dx = y_vec[ioffset] - y_vec[joffset]
                    dy = y_vec[ioffset+1] - y_vec[joffset+1]
                    dz = y_vec[ioffset+2] - y_vec[joffset+2]
                    r = (dx**2+dy**2+dz**2)**0.5
                    res += self.mass_vec[i] * self.mass_vec[j] / r
            res_p.append(res)
        res_p = kahan_sum(np.array(res_p))
        return res_p * self.G / 2.0


    def plot(self, point_size = 30, orbit_width = 1,
            orbit_alpha = 1, point_alpha = 1, point_marker = 'o',
            linestyle = '-', point_edge_width=1):
        if not hasattr(self, 'history'):
            raise AttributeError('Missing attribute "history", maybe need run
simulation?')
        fig = plt.figure(figsize=(11,5))
        fig.suptitle('Graphical results',fontsize=14, fontstyle='italic')
        ax1 = fig.add_subplot(1, 2, 1)
        ax2 = fig.add_subplot(1, 2, 2, projection='3d')
        if self.has_units and self.has_units.upper() == 'CGS':
            unit = 'cm'
        elif self.has_units and self.has_units.upper() == 'SI':
            unit = 'm'
        else:
            unit = 'unit'
        ax1.set_xlabel(unit)
        ax1.set_ylabel(unit)
```

```
        for i in range(len(self.bodies)):
            offset = i * 6
            x_ = self.history[0][offset + 0]
            y_ = self.history[0][offset + 1]
            z_ = self.history[0][offset + 2]
            x, y, z = [], [], []
            colors_1 = np.random.rand(3)
            colors_2 = np.random.rand(3)
            ax1.scatter(x=x_, y=y_, marker=point_marker, color=colors_1,
                        linewidths=point_edge_width, edgecolor=colors_2,
alpha=point_alpha, s=point_size)
            ax2.scatter(x_, y_, z_, marker=point_marker, color=colors_1,
alpha=point_alpha, s=point_size)
            for j in range(len(self.history)):
                x.append(self.history[j][offset])
                y.append(self.history[j][offset + 1])
                z.append(self.history[j][offset + 2])
            ax1.plot(x,y, color=colors_1,
                     linewidth=orbit_width, alpha=orbit_alpha,
                     label='{}'.format(self.name_vec[i]),
                     linestyle=linestyle)
            ax2.plot(x,y,z, color=colors_1,)
        ax1.grid(color='black', linewidth=0.3, linestyle='--')
        ax1.legend(fontsize=6, ncol=2, facecolor='oldlace',
                   edgecolor='green', title='Orbits', title_fontsize='8',
loc='lower right')
        plt.show()


    def print_unit_system(self):
        if self.has_units.upper() == 'CGS' or self.has_units.upper() == 'SI':
            print(' "{}" unit system'.format(self.has_units.upper()))
        else:
            print('Dimensionless system.')
```

Реализация вспомогательных функций:

```
@njit
def kahan_sum(input_vec):
    sum_ = 0.0
    c = 0.0
    for i in range(len(input_vec)):
        y = input_vec[i] - c
        t = sum_ + y
        c = (t - sum_) - y
        sum_ = t
    return sum_

@njit
def calc_tol_(y, y_, a_tol, r_tol, **kwargs):
    if np.isscalar(a_tol) and np.isscalar(r_tol):
        return np.array([a_tol+max(abs(y[i]), abs(y_[i]))*r_tol for i in
range(len(y))])

    if len(a_tol) == len(r_tol) == 1:
        return np.array([a_tol[0]+max(abs(y[i]), abs(y_[i]))*r_tol[0] for i
in range(len(y))])


def calc_tol_n(y, y_, a_tol, r_tol, **kwargs):
    if 'nbody' in kwargs:
        nbody = kwargs['nbody']
    else:
        raise ValueError('You need to set "nbody" in calc_tol_n()!')

    if len(a_tol) == len(r_tol) == 2:
        res = np.zeros(len(y))
        for i in range(nbody):
            offset = i * 6
            res[offset] = a_tol[0] + max(abs(y[offset]),
abs(y_[offset]))*r_tol[0]
            res[offset+1] = a_tol[0] + max(abs(y[offset+1]),
abs(y_[offset+1]))*r_tol[0]
            res[offset+2] = a_tol[0] + max(abs(y[offset+2]),
abs(y_[offset+2]))*r_tol[0]

            res[offset+3] = a_tol[1] + max(abs(y[offset+3]),
abs(y_[offset+3]))*r_tol[1]
            res[offset+4] = a_tol[1] + max(abs(y[offset+4]),
abs(y_[offset+4]))*r_tol[1]
            res[offset+5] = a_tol[1] + max(abs(y[offset+5]),
abs(y_[offset+5]))*r_tol[1]
        return res

    raise AttributeError('Error size of atol/rtol!')


@njit
def calc_err_(tol, y, y_):
    res = np.array([abs(y[i]-y_[i])/tol[i] for i in range(y.size)])
    return res


def calc_err_norm(err, ord=None):
    '''
    Function for calculate one of the types of norms for err vector.
    ------------------------------
    Params:
```

```
        err: vector of errors / tol_i.
        ord: order of the norm (type of norm).
            ============
=================================================================
            None            calculated as in a lecture: sqrt{1/n * sum[(xi -
x_i) / tol_i]}
            'std'           similarly None
            2/'2'/'eucl'    euclidean norm
            inf/'oo'        infinite norm
            1/'1'           unit norm: sum[abs(xi)]
    '''
    if ord==None or ord=='std':
        res = np.sum(err**2) / err.size
        return sqrt(res)

    if ord==2 or ord=='2' or ord=='eucl':
        return np.sum(np.abs(err)**2)**(1./2)

    if ord==np.inf or ord=='oo':
        return np.max(np.abs(err))

    if ord==1 or ord=='1':
        return np.sum(np.abs(err))

    raise ValueError('Invalid parameter "ord": {}'.format(ord))


def grav_nbody_calc_diff_eqs(t, y, **kwargs):
    if 'num_calls' in kwargs:
        kwargs['num_calls'][0] += 1

    if 'masses' in kwargs:
        masses = kwargs['masses']
    else:
        raise ValueError('You need to pass masses of points!')

    if 'G' in kwargs:
        G = kwargs['G']
    else:
        G = astr_const.G.cgs.value

    has_units = False

    if 'dimension' in kwargs:
        dimension = kwargs['dimension']
    else:
        dimension = 6

    n_bodies = int(len(y) / dimension)
    solved_vec = np.zeros(y.size)

    for i in range(n_bodies):
        i_offset = i * dimension
        solved_vec[i_offset] = y[i_offset + 3]
        solved_vec[i_offset + 1] = y[i_offset + 4]
        solved_vec[i_offset + 2] = y[i_offset + 5]
        for j in range(n_bodies):
            j_offset = j * dimension

            if i != j:
                dx = y[i_offset] - y[j_offset]
                dy = y[i_offset + 1] - y[j_offset + 1]
                dz = y[i_offset + 2] - y[j_offset + 2]
```

```
                r = (dx**2 + dy**2 + dz**2) ** 0.5
                ax = (-G * masses[j] / r**3) * dx
                ay = (-G * masses[j] / r**3) * dy
                az = (-G * masses[j] / r**3) * dz
                if has_units:
                    ax = ax.value
                    ay = ay.value
                    az = az.value
                solved_vec[i_offset + 3] += ax
                solved_vec[i_offset + 4] += ay
                solved_vec[i_offset + 5] += az
    return solved_vec

def smooth_graph_points(x, factor=0.9):
    smoothed_points = []
    for point in x:
        if smoothed_points:
            prev = smoothed_points[-1]
            smoothed_points.append(prev * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points


def save_np_array(arr, file_name, delimeter=",", fmt='%.8e', newline='\n',
header='', curr_script_dir=False):
    if curr_script_dir:
        np.savetxt(os.path.dirname(os.path.abspath(__file__))+'\\'+file_name,
arr, delimiter=delimeter, fmt=fmt, newline=newline, header=header)
    else:
        np.savetxt(file_name, arr, delimiter=delimeter, fmt=fmt,
newline=newline, header=header)
```