

ДОКУМЕНТАЦИЯ ПО ПРОЕКТУ

Динамическая библиотека libPendulum.dll

Программирование, механика, 1 курс

17 мая 2018 г.

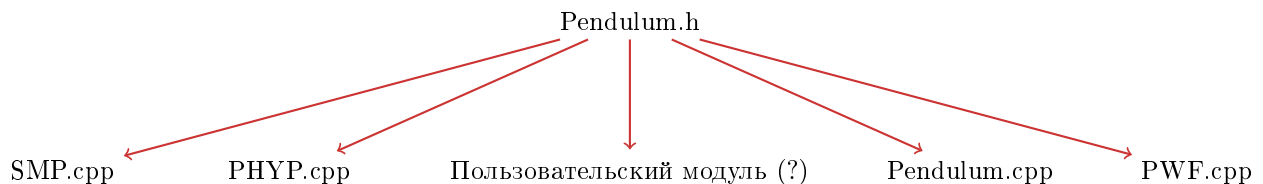
Оглавление

0.1	Структура библиотеки	2
0.2	Модули	2
0.2.1	Pendulum.cpp и класс Pendulum	2
0.2.2	Простой математический маятник. Класс Simple_Math_Pendulum и SMP.cpp	3
0.2.3	Маятник с вязким трением. Класс Pendulum_W_Friction и PWP.cpp	5
0.2.4	Физический маятник. Класс Ph_Pendulum и PHYR.cpp	7

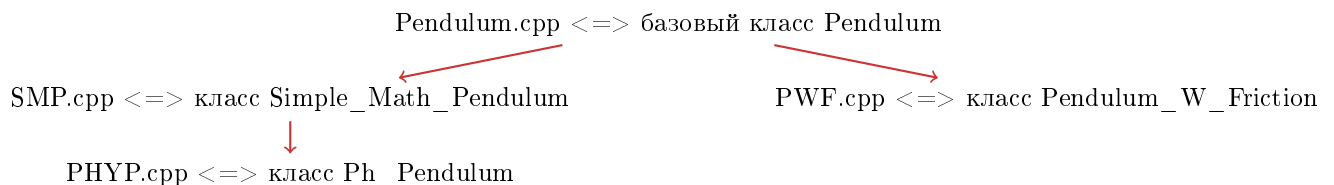
Динамическая библиотека libPendulum.dll содержит в себе классы моделей маятников, а так же некоторые необходимые константы, функции и т.д. Пока что здесь всего три более-менее полноценных модели: простой математический маятник, физический маятник, и маятник, на который действует вязкое трение. Кроме того, метод для нахождения координат маятника Фуко. Автор библиотеки надеется расширить ее. Когда-нибудь...

0.1 Структура библиотеки

Составляющие (на данный момент, еще можно добавить пользовательские модули):



Внутренние зависимости:



Чтобы подключить библиотеку, достаточно заголовочного файла Pendulum.h и, собственно, библиотеки.

0.2 Модули

0.2.1 Pendulum.cpp и класс Pendulum

Содержит описание методов базового класса Pendulum: функции ввода-вывода стандартных характеристик — начальных координаты и скорости, длины подвеса и массы (и вывод, и ввод с проверкой на корректность), угла отклонения (фазы) и амплитуды (только вывод). bool-поле existence определяет, существует маятник или нет. Маятник может перестать существовать в результате некорректного ввода данных.

Листинг 1: Pendulum.cpp

```

1 #include "Pendulum.h"
2 #include <cmath>
3 using namespace std;
4 void Pendulum::put_x0(double d) { //меняет начальную координату на переданное
    значение и переопределяет остальные характеристики
5     x0 = d;
6     def();
7 }
8 void Pendulum::put_v0(double d) { //меняет начальную скорость на переданное зн
    ачение и переопределяет остальные характеристики
9     v0 = d;
10    def();
11 }
12 void Pendulum::put_m(double d) { //меняет массу на переданное значение, если о
    но корректное
13    try {
  
```

```

14     if (d <= 0) throw "Масса маятника должна быть положительным числом\n";
15     m = d;
16     def(); //переопределяет остальные характеристики
17 }
18 catch (const char* s) {
19     cerr << s;
20 }
21 }
22 void Pendulum::put_l(double d) { //меняет длину на переданное значение, если о
    но корректное
23     try {
24         if (d <= 0) throw "Длина маятника должна быть положительным числом\n";
25         l = d;
26         def(); //переопределяет остальные характеристики
27     }
28     catch (const char* s) {
29         cerr << s;
30     }
31 }
32 double Pendulum::get_x0() { //возвращает начальную координату
33     return x0;
34 }
35 double Pendulum::get_v0() { //возвращает начальную скорость
36     return v0;
37 }
38 double Pendulum::get_m() { //возвращает массу
39     return m;
40 }
41 double Pendulum::get_l() { //возвращает длину
42     return l;
43 }
44 double Pendulum::get_a() { //возвращает начальную фазу
45     return a;
46 }
47 double Pendulum::get_A() { //возвращает амплитуду
48     return A;
49 }

```

Класс Pendulum и Pendulum.cpp необходимы для остальных трех существующих “моделей”. Pendulum включает в себя виртуальный метод `void def()`, который определяет характеристики “модели” на основании начальных данных. Для каждой существующей модели `def()` определен по-разному.

0.2.2 Простой математический маятник. Класс Simple_Math_Pendulum и SMP.cpp

Зависит от Pendulum.cpp, т.к. Simple_Math_Pendulum является наследником Pendulum. Кроме полей базового класса добавляются частота (`double w`) и период (`double T`), которые определяют `def()`, а так же методы `double get_w()` и `double get_T()` соответственно.

```

1 double SimpleMathPendulum::get_w(){
2     return w;
3 }
4 double SimpleMathPendulum::get_T(){
5     return T;
6 }

```

У класса два конструктора.

Листинг 2: Так определится простой математический маятник по умолчанию

```
1 SimpleMathPendulum::SimpleMathPendulum() {
2     x0 = 1;
3     v0 = 0;
4     m = 0.1;
5     l = 5;
6     def();
7 }
```

Почему какие-то заранее заданные значения? Чтобы не возникало неопределенности. Поэтому маятник “по умолчанию” — это маятник массой 100 грамм, с длиной подвеса 5 метров, начальной координатой 1 м и нулевой начальной скоростью.

Листинг 3: Так можно считать маятник из файлового потока

```
1 SimpleMathPendulum::SimpleMathPendulum(istream & f1) {
2     try {
3         f1 >> x0;
4         f1 >> m;
5         f1 >> l;
6         f1 >> v0;
7         if (m <= 0) throw "Масса маятника должна быть положительным числом \n";
8         if (l <= 0) throw "Длина маятника должна быть положительным числом \n";
9         def();
10    }
11    catch (const char* s) {
12        cout << s;
13        existence = false;
14    }
15    catch (...) {
16        cout << "Неверный формат исходных данных \n";
17        existence = false;
18    }
19 }
```

Листинг 4: Определение SMP. Метод def()

```
1 void SimpleMathPendulum::def() {
2     T = 2 * pi*pow(l / g, 0.5); //период
3     w = pow(g / l, 0.5); //частота
4     A = pow((x0*x0 + (v0 / w)*(v0 / w)), 0.5); //амплитуда
5     if (v0 == 0)
6         a = pi / 2;
7     else //фаза (угол начального отклонения)
8         a = atan(x0*w / v0);
9 }
```

И, наконец, метод double x (double t), находящий координату x маятника в момент времени t.

Листинг 5: Уравнение движения. Метод x(t).

```
1 double SimpleMathPendulum::x(double t) {
2     if (existence)
3         return A*sin(w*t + a);
4     cerr << "Маятник не существует \n";
5     return NULL;
6 }
```

В Simple_Math_Pendulum есть метод void Fou(double &, double &, double, double), который возвращает координаты x и y маятника Фуко с теми же начальными данными, что и у данного

простого математического. //хотя можно было бы создать отдельный класс(-наследник?) маятник Фуко.

Метод принимает в себя переменные, в которые нужно записать значения x и y , момент времени t и географическую широту (в радианах).

Листинг 6: Координаты маятника Фуко

```
1 void SimpleMathPendulum::Fou(double &x, double &y, double t, double f) {
2     if (existence) {
3         double p = 0; double tt = 0;
4         double w1 = pow(w*w + pow(o*sin(f), 2), 0.5);
5         tt = o*sin(f)*t;
6         double i, j;
7         if (x0) {
8             i = atan((w1*cos(tt)) / (v0 / x0 + o*sin(f)*sin(tt)));
9             if (sin(i))
10                 j = x0 / sin(i);
11             else j = 0;
12         }
13         else {
14             i = 0;
15             j = v0 / (w1*cos(tt));
16         }
17         p = j*sin(w1*t+i);
18         x = p*cos(tt);
19         y = p*sin(tt);
20     }
21     else {
22         x = y = 0;
23         cout << "Маятник не существует \n";
24     }
25 }
```

0.2.3 Маятник с вязким трением. Класс Pendulum_W_Friction и PWP.cpp

Так же, как и предыдущий класс, является наследником Pendulum. Новые поля — динамическая вязкость k (в СИ — Па*с), указатель на функцию xt . Возвращает k — `double get_k()`, а принимает — `void put_k(double)`.

Листинг 7: Конструкторы Pendulum_W_Friction

```
1 PendulumWFrict::PendulumWFrict() { //тот же самый маятник “по умолчанию”, что
2     u y Simple_Math_Pendulum
3     x0 = 1;
4     v0 = 0;
5     m = 0.1;
6     l = 5;
7     k = 0;
8     def();
9 }
10 PendulumWFrict::PendulumWFrict(ifstream & f1) { //Ничего принципиально не поменя
11     лось
12     try {
13         f1 >> x0;
14         f1 >> m;
15         f1 >> l;
16         f1 >> v0;
17         f1 >> k;
18         if (m <= 0) throw "Масса маятника должна быть положительным числом\n";
19         if (l <= 0) throw "Длина маятника должна быть положительным числом\n";
20         if (k < 0) throw "Вязкость должна быть неотрицательным числом\n";
21         def();
22     }
```

```

20 }
21 catch (const char* s) {
22     existence = false;
23     cerr << s;
24 }
25 catch (...) {
26     cerr << "Неверный формат исходных данных \n";
27     existence = false;
28 }
29 }

```

Листинг 8: Определение PWF. Метод def()

```

1 void PendulumWFrict::def() {
2     w0 = pow(g / l, 0.5);
3     A = pow((x0*x0 + (v0 / w0)*(v0 / w0)), 0.5);
4     if (v0 == 0)
5         a = pi / 2;
6     else
7         a = atan(x0*w0 / v0);
8     c = k / (2 * m);
9     if (m*g*sin(a) <= k*v0) {
10         xt = NULL;
11         v0 = 0;
12     }
13     else {
14         if (w0 > c)
15             xt = x1;
16         if (w0 == c)
17             xt = x2;
18         if (w0 < c)
19             xt = x3;
20     }
21 }

```

Так как решение уравнения движения маятника с вязким трением зависит от величин динамической вязкости и “начальной” частоты, три возможных варианта реализованы отдельными функциями x1, x2 и x3:

```

1 double x1(double t, double c, double v0, double w0, double x0) {
2     double A;
3     double w = pow(w0*w0 - c*c, 0.5);
4     A = pow((x0*x0 + pow((v0 + c*x0)/(w), 2)), 0.5);
5     double a;
6     if (A == 0)
7         a = 0;
8     else
9         a = asin(x0/A);
10    return A*exp(-c*t)*sin(w*t + a);
11 }
12 double x2(double t, double c, double v0, double w0, double x0) {
13
14     double a1 = c + pow(c*c - w0*w0, 0.5);
15     double a2 = c - pow(c*c - w0*w0, 0.5);
16     double C2 = (v0 + a1*x0) / (a1 - a2);
17     double C1 = x0 - C2;
18     return C1*exp(-a1*t) + C2*exp(-a2*t);
19 }
20 }
21 double x3(double t, double c, double v0, double w0, double x0) {
22     double C1 = x0*c + v0;
23     double C2 = x0;

```

```

24     return (C1*t + C2)*exp(-c*t);
25 }

```

Какую функцию использовать решает сам метод `def()`, изменить это извне нельзя (или все-таки можно?).

Тогда так будет выглядеть метод, возвращающий координату x маятника с вязким трением:

```

1 double PendulumWFrict::x(double t) {
2     if (existence) {
3         if (xt == NULL) return x0;
4         return xt(t, c, v0, w0, x0);
5     }
6     cout << "Маятник не существует \n";
7     return NULL;
8 }

```

У затухающих колебаний, которые описывает первый случай (x_1), амплитуда зависит от времени:

```

1 double PendulumWFrict::get_A(double t = 0) {
2     return A*exp(-c*t);
3 }

```

А вот методы для редактирования и получения динамической вязкости. Совершенно аналогично с предыдущими методами...

```

1 void PendulumWFrict::put_k(double d) {
2     try {
3         if(d<0) throw "Вязкость должна быть неотрицательным числом\n";
4         k = d;
5         def();
6     }
7     catch (const char* s) {
8         cerr << s;
9     }
10 }
11 double PendulumWFrict::get_k() {
12     return k;
13 }

```

0.2.4 Физический маятник. Класс `Ph_Pendulum` и `PHYP.cpp`

Если заменить длину подвеса приведенной длиной, уравнение физического маятника будет совпадать с уравнением математического. Поэтому эта модель реализована как наследник `Simple_Math_Pendulum` — простого математического маятника. Что здесь нового?

Новые поля — форма f , “радиус” формы r , момент импульса J , а так же приведенная длина L . Как эти поля редактировать и получать:

```

1 void PhPendulum::put_r(double d) {
2     try {
3         if(d<=0) throw "Радиус тела должна быть положительным числом \n";
4         r = d;
5         def();
6     }
7     catch (string s) {
8         cerr << s;
9     }
10 }
11 double PhPendulum::get_r() {
12     return r;
13 }
14 double PhPendulum::get_J() {
15     return J;

```



```

16 }
17 form PhPendulum::get_form() {
18     return f;
19 }

```

У этого физического маятника существует только четыре возможных формы: куб, шар, диск и цилиндр. За “радиус” берем половину стороны куба, радиус шара, малый радиус диска, радиус цилиндра. Если же форма неизвестная, то считается, что физический маятник - математический. Метод, записывающий форму:

Листинг 9: Для файлового ввода

```

1 void getform(form& f, ifstream &f1) {
2     string s;
3     try {
4         f1 >> s;
5         if (s == "шар") f = ball;
6         if (s == "куб") f = cube;
7         if (s == "цилиндр") f = cilinder;
8         if (s == "диск") f = disk;
9         if (f != ball && f != cube && f != cilinder && f != disk) throw "Неопознанна
            я форма: " + s + '\n';
10    }
11    catch (string s) {
12        cerr << s;
13        f = none;
14    }
15    catch (...) {
16        cerr << "Неверный формат исходных данных\n";
17        f = none;
18    }
19
20
21 }

```

Листинг 10: И для любого

```

1 void PendulumWFrict::put_k(double d) {
2     try {
3         if(d<0) throw "Вязкость должна быть неотрицательным числом\n";
4         k = d;
5         def();
6     }
7     catch (const char* s) {
8         cerr << s;
9     }
10 }

```

Конструкторы физического маятника:

Листинг 11: Все тот же маятник “по умолчанию”

```

1 PhPendulum::PhPendulum() {
2     f = none;
3     r = 0;
4 }

```

Листинг 12: Физический маятник конструируется из файлового потока

```

1 PhPendulum::PhPendulum(ifstream & f1) {
2     try {
3         f1 >> x0;

```

```

4   f1 >> m;
5   f1 >> l;
6   f1 >> v0;
7   getform(f, f1);
8   f1 >> r;
9   if (r <= 0) throw "Радиус тела должна быть положительным числом \n";
10  if (m <= 0) throw "Масса маятника должна быть положительным числом \n";
11  if (l <= 0) throw "Длина маятника должна быть положительным числом \n";
12  if (f == none)
13      SimpleMathPendulum::def();
14  else
15      def();
16  }
17  catch (const char* s) {
18      existence = false;
19      cerr << s;
20  }
21  catch (...) {
22      existence = false;
23      cerr << "Неверный формат исходных данных \n";
24  }
25  }

```

Наконец, так определен def() для Ph_Pendulum:

```

1  void PhPendulum::def() {
2      if (f == ball) {
3          J = 0.4*m*r*r;
4      }
5      if (f == cube) {
6          J = 2 * m*r*r / 3;
7      }
8      if (f == cilinder) {
9          J = m*r*r / 2;
10     }
11     if (f == disk) {
12         J = m*r*r / 4; //вычислили момент инерции в зависимости от формы
13     }
14     L = l + r + J / ((1 + r)*m); //вычислили приведенную длину
15     T = 2 * pi*pow(L / g, 0.5); //теперь это математический маятник
16     w = pow(g / L, 0.5);
17     A = pow((x0*x0 + (v0 / w)*(v0 / w)), 0.5);
18     if (v0 == 0)
19         a = pi / 2;
20     else
21         a = atan(x0*w / v0);
22 }

```

Функция double x(double t) родителя Ph_Pendulum — Simple_Math_Pendulum возвращает координату центра качания данного физического маятника.