Документация по проекту

# Динамическая библиотека libPendulum.dll

*Программирование, механика, 1 курс*

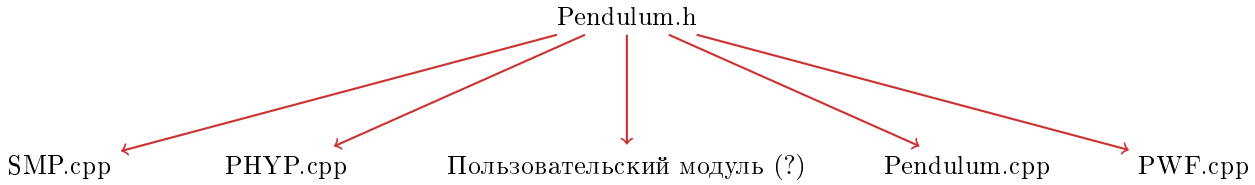
17 мая 2018 г.

# Оглавление

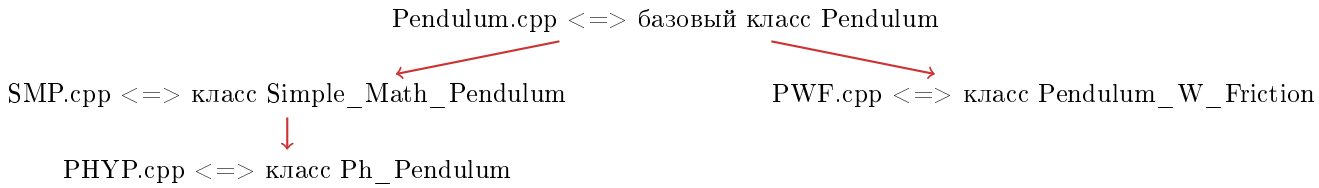
Динамическая библиотека libPendulum.dll содержит в себе классы моделей маятников, а так же некоторые необходимые константы, функции и т.д. Все физические величины далее приведены в СИ, если указано другое. Пока что здесь всего три более-менне полноценных модели: простой математический маятник, физический 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

```
#include "Pendulum.h"
#include <cmath>
using namespace std;
void Pendulum::put_x0(double d) {   //меняет начальную координату на переданное значение и переопределяет остальные характеристики
  x0 = d;
  def();
}
void Pendulum::put_v0(double d) {   //меняет начальную скорость на переданное значение и переопределяет остальные характеристики
  v0 = d;
  def();
}
void Pendulum::put_m(double d) {   //меняет массу на переданное значение, если оно корректное
```

```
  try {
    if (d <= 0) throw "Масса маятника должна быть положительным числом\n";
    m = d;
    def();  //переопределяет остальные характеристики
  }
  catch (const char* s) {
    cerr << s;
  }
}
void Pendulum::put_l(double d) {   //меняет длину на переданное значение, если о
    но корректное
  try {
    if (d <= 0) throw "Длина маятника должна быть положительным числом\n";
    l = d;
    def();  //переопределяет остальные характеристики
  }
  catch (const char* s) {
    cerr << s;
  }
}
double Pendulum::get_x0() { //возвращает начальную координату
  return x0;
}
double Pendulum::get_v0() { //возвращает начальную скорость
  return v0;
}
double Pendulum::get_m() { //возвращает массу
  return m;
}
double Pendulum::get_l() { //возвращает длину
  return l;
}
double Pendulum::get_a() {  //возвращает начальную фазу
  return a;
}
double Pendulum::get_A() {  //возвращает амплитуду
  return A;
}
```

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 определяютя `def()`, а так же методы `double get_w()` и `double get_T()` соответственно.

```
double SimpleMathPendulum::get_w(){
  return w;
}
double SimpleMathPendulum::get_T(){
  return T;
}
```

У класса два конструктора.

Листинг 2: Так определится простой математический маятник по умолчанию

```
SimpleMathPendulum::SimpleMathPendulum() {
  x0 = 1;
  v0 = 0;
  m = 0.1;
  l = 5;
  def();
}
```

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

```
SimpleMathPendulum::SimpleMathPendulum(ifstream & f1) {
  try {
    f1 >> x0;
    f1 >> m;
    f1 >> l;
    f1 >> v0;
    if (m <= 0) throw "Масса маятника должна быть положительным числом \n";
    if (l <= 0) throw "Длина маятника должна быть положительным числом \n";
    def();
  }
  catch (const char* s) {
    cout << s;
    existence = false;
  }
  catch (...) {
    cout << "Неверный формат исходных данных \n";
    existence = false;
  }
}
```

Листинг 4: Определение SMP. Метод def()

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

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

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

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

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

```
void SimpleMathPendulum::Fou(double &x, double &y, double t, double f) {
  if (existence) {
    double p = 0; double tt = 0;
    double w1 = pow(w*w + pow(o*sin(f), 2), 0.5);
    tt = o*sin(f)*t;
    double i, j;
    if (x0) {
      i = atan((w1*cos(tt)) / (v0 / x0 + o*sin(f)*sin(tt)));
      if (sin(i))
        j = x0 / sin(i);
      else j = 0;
    }
    else {
      i = 0;
      j = v0 / (w1*cos(tt));
    }
    p = j*sin(w1*t+i);
    x = p*cos(tt);
    y = p*sin(tt);
  }
  else {
    x = y = 0;
    cout << "Маятник не существует \n";
  }
}
```

### 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

```
PendulumWFrict::PendulumWFrict() { //тот же самый маятник ''по умолчанию'', что
    и y Simple_Math_Pendulum
  x0 = 1;
  v0 = 0;
  m = 0.1;
  l = 5;
  k = 0;
  def();
}
PendulumWFrict::PendulumWFrict(ifstream & f1) { //Ничего принципиально не поменя
    лось
  try {
    f1 >> x0;
    f1 >> m;
    f1 >> l;
    f1 >> v0;
    f1 >> k;
    if (m <= 0) throw "Масса маятника должна быть положительным числом\n";
    if (l <= 0) throw "Длина маятника должна быть положительным числом\n";
    if (k < 0) throw "Вязкость должна быть неотрицательным числом\n";
    def();
```

```
  }
  catch (const char* s) {
    existence = false;
    cerr << s;
  }
  catch (...) {
    cerr << "Неверный формат исходных данных \n";
    existence = false;
  }
}
```

Листинг 8: Определение PWF. Метод def()

```
void PendulumWFrict::def() {
  w0 = pow(g / l, 0.5);
  A = pow((x0*x0 + (v0 / w0)*(v0 / w0)), 0.5);
  if (v0 == 0)
    a = pi / 2;
  else
    a = atan(x0*w0 / v0);
  c = k / (2 * m);
  if (m*g*sin(a) <= k*v0) {
    xt = NULL;
    v0 = 0;
  }
  else {
    if (w0 > c)
      xt = x1;
    if (w0 == c)
      xt = x2;
    if (w0 < c)
      xt = x3;
  }
}
```

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:

```
double x1(double t, double c, double v0, double w0, double x0) {
  double A;
  double w = pow(w0*w0 - c*c, 0.5);
  A = pow((x0*x0 + pow((v0 + c*x0)/(w),2)), 0.5);
  double a;
  if (A == 0)
    a = 0;
  else
    a = asin(x0/A);
  return A*exp(-c*t)*sin(w*t + a);
}
double x2(double t, double c, double v0, double w0, double x0) {

  double a1 = c + pow(c*c - w0*w0, 0.5);
  double a2 = c - pow(c*c - w0*w0, 0.5);
  double C2 = (v0 + a1*x0) / (a1 - a2);
  double C1 = x0 - C2;
  return C1*exp(-a1*t) + C2*exp(-a2*t);

}
double x3(double t, double c, double v0, double w0, double x0) {
  double C1 = x0*c + v0;
  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:

```
double PendulumWFrict::x(double t) {
  if (existence) {
    if (xt == NULL) return x0;
    return xt(t, c, v0, w0, x0);
  }
  cout << "Маятник не существует \n";
  return NULL;
}
```

У затухающих колебаний, которые описывает первый случай (x1), амплитуда зависит от времени:

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

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

```
void PendulumWFrict::put_k(double d) {
  try {
    if(d<0) throw "Вязкость должна быть неотрицательным числом\n";
    k = d;
    def();
  }
  catch (const char* s) {
    cerr << s;
  }
}
double PendulumWFrict::get_k() {
  return k;
}
```

### 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:

```
void PhPendulum::put_r(double d) {
  try {
    if(d<=0) throw "Радиус тела должна быть положительным числом \n";
    r = d;
    def();
  }
  catch (string s) {
    cerr << s;
  }
}
double PhPendulum::get_r() {
  return r;
}
double PhPendulum::get_J() {
  return J;
```

```
}
form PhPendulum::get_form() {
  return f;
}
```

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. Функция (не метод), записывающая форму:

Листинг 9: Для файлового ввода

```
void getform(form& f, ifstream &f1) {
  string s;
  try {
    f1 >> s;
    if (s == "шар") f = ball;
    if (s == "куб") f = cube;
    if (s == "цилиндр") f = cilinder;
    if (s == "диск") f = disk;
    if (s != "шар" && s != "куб" && s != "цилиндр" && s != "диск") throw "Неопоз
        нанная форма: " + s + '\n';
  }
  catch (string s) {
    cerr << s;
    f = none;
  }
  catch (...) {
    cerr << "Неверный формат исходных данных\n";
    f = none;
  }

}
```

Листинг 10: Метод

```
void PendulumWFrict::put_k(double d) {
  try {
    if(d<0) throw "Вязкость должна быть неотрицательным числом\n";
    k = d;
    def();
  }
  catch (const char* s) {
    cerr << s;
  }
}
```

Конструкторы физического маятника:

Листинг 11: Все тот же маятник “по умолчанию”

```
PhPendulum::PhPendulum() {
  f = none;
  r = 0;
}
```

Листинг 12: Физический маятник конструируется из файлового потока

```
PhPendulum::PhPendulum(ifstream & f1) {
  try {
    f1 >> x0;
    f1 >> m;
```

```
    f1 >> l;
    f1 >> v0;
    getform(f, f1);
    f1 >> r;
    if (r <= 0) throw "Радиус тела должна быть положительным числом \n";
    if (m <= 0) throw "Масса маятника должна быть положительным числом \n";
    if (l <= 0) throw "Длина маятника должна быть положительным числом \n";
    if (f == none)
      SimpleMathPendulum::def();
    else
      def();
  }
  catch (const char* s) {
    existence = false;
    cerr << s;
  }
  catch (...) {
    existence = false;
    cerr << "Неверный формат исходных данных \n";
  }
}
```

Наконец, так определен `def()` для Ph_Pendulum:

```
void PhPendulum::def() {
  if (f == ball) {
    J = 0.4*m*r*r;
  }
  if (f == cube) {
    J = 2 * m*r*r / 3;
  }
  if (f == cilinder) {
    J = m*r*r / 2;
  }
  if (f == disk) {
    J = m*r*r / 4; //вычислили момент инерции в зависимости от формы
  }
  L = l + r + J / ((l + r)*m); //вычислили приведенную длину
  T = 2 * pi*pow(L / g, 0.5); //теперь это математический маятник
  w = pow(g / L, 0.5);
  A = pow((x0*x0 + (v0 / w)*(v0 / w)), 0.5);
  if (v0 == 0)
    a = pi / 2;
  else
    a = atan(x0*w / v0);
}
```

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