Московский ордена Ленина, ордена Октябрьской Революции

и ордена Трудового Красного Знамени

государственный технический университет им. Н.Э. Баумана

Факультет «Робототехники и комплексной автоматизации»

Кафедра «Системы автоматизированного проектирования»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту на тему:

*«Оптимизация курса малого парусного судна с обходом препятствия»*

Студент \_\_\_\_\_\_\_\_\_\_ (Яблоков В.) Группа РК6-61

Руководитель проекта \_\_\_\_\_\_\_\_\_\_ (Берчун Ю.В.)

Москва, 2014

# **Содержание**

Содержание 1

Введение 2

Общая схема решения 2

Динамическая модель судна 2

Испытание модели 3

Метод оптимизации курса 4

Обработка ограничений (обход препятствия) 5

Средства реализации 6

Результаты работы алгоритма 7

Заключение 11

Использованные источники 12

Приложение 1. Листинг кода программы 13

# 

# Введение

В данной работе рассмотрена задача поиска наискорейшего курса парусного судна до некоторой удаленной точки с учетом препятствия на пути следования. Рассматриваются суда с площадью паруса до 15 кв. м и массой до 100 кг. Это ограничение позволяет упростить физическую модель, при моделировании более крупных судов начинает играть заметную роль множество факторов, сложных в расчете. Выбрана простая динамическая модель судна, в которой учтены только самые важные факторы. В качестве основного метода оптимизации был выбран метод случайного поиска.

# Общая схема решения

Задача разбивается на следующие подзадачи:

1. Составление динамической модели парусного судна;
2. Реализация испытания модели;
3. Составление алгоритма метода оптимизации для выбора приближенно наилучшего решения.

# Динамическая модель судна

Прежде всего необходимо определить те факторы, которые будут учитываться при построении модели. Данная модель строилась как механическая система из одной материальной точки. Таким образом, путь упрощенно представлен ломаной линией. Повороты динамически не моделируются, однако, время, которое на них затрачивается, учитывается по приближенной формуле, которая будет приведена ниже. При моделировании прямолинейного движения судна учитывались следующие внешние силы:

* сила сопротивления воды
* сила сопротивления воздуху (парусом)
* «подъемная» сила паруса

Сила сопротивления воздуху является основной движущей силой при следовании не против ветра. Подъемная сила паруса направлена перпендикулярно нему и во многом похожа на одноименную силу, рассматриваемую в аэродинамике. Она проявляется наиболее заметно при боковом ветре, т.е. при небольших углах атаки. Именно за счет этой силы возможно движение т.н. острыми галсами – т.е. под острым углом к ветру. Формулы вычисления данных сил приведены на листе №1 Приложения.

Каждая из сил определяется скоростью судна относительно той среды, для которой записана сила, а также углом атаки.

Таким образом, можно записать систему дифференциальных уравнений движения судна:

,

где – вектор, указывающий направление движения судна. Тогда скорость суда в следующий момент времени определяется как:

Радиус-вектор судна:

Таким образом, задав направление движения и угол установки паруса, можно получить динамическую характеристику движения судна.

# Испытание модели

Испытание модели заключается в решении уравнений и нахождении оптимального угла установки паруса для выбранного курса. Экспериментально было установлено, что для малых судов время, затрачиваемое на приобретение постоянной скорости после смены галса, достаточно мало, чтобы рассматривать все движение в неизменном направлении как равномерное.

В ходе решения задачи оптимизации рассматривается целевая функция, вычисляющая время движения из исходной точки в конечную через точку поворота. Метод, реализующий эту функцию, работает по следующей схеме:

1. Определить направление движения из исходной точки до точки поворота.
2. Вычислить оптимальный угол установки паруса методом.
3. Вычислить установившуюся скорость движения при оптимальной установке паруса
4. Время поворота на заданное направление подсчитать по формуле:

*,*

где – усредненное время совершения судном полного разворота, – угол между текущим и новым направлением.

1. Повторить пункты 1-3 для прямого курса из точки поворота в конечной точки.

Попутно, он выполняет проверку на достижимость очередной точки назначения прямым курсом.

# Метод оптимизации курса

Для многомерной оптимизации целевой функции используется метод случайного поиска. Опишем его принципиальную схему.

В рамках одной итерации метода координаты начальной и конечной точек движения являются константами. Варьируемые параметры – координаты точки, через которую будет проложен путь (ломаная) с одним поворотом от начальной точки до конечной.

Целевой функцией оптимизации является время пути из начальной точки в конечную через выбранную точку поворота, считая время, затрачиваемое судном на сам поворот.

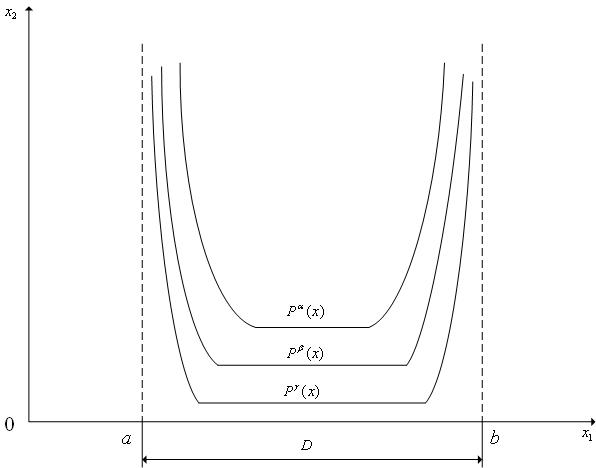
1. Прежде всего примем начальную точку движения судна за оптимальную точку поворота в первом приближении и вычислим значение целевой функции для нее, если это возможно. В дальнейшем будем сравнивать получаемые значения целевой функции с этим временем. Для первой итерации примем .
2. Выберем некоторое количество случайных точек на одном расстоянии от точки . Проведем испытания целевой функции для этих точек и отберем ту из них, в которой целевая функция минимальна .
3. Затем перейдем к одномерной оптимизации и найдем минимум целевой функции в направлении от начальной точки к выбранной в пункте 2: , где – числовой коэффициент. В качестве метода одномерной оптимизации используется метод Паулла, который относится к методам квадратичной аппроксимации.
4. Проверим, выполняется ли условие сходимости и если нет, то вернемся к пункту 2, приняв полученную в пункте 3 точку вместо начальной. В процессе сходимости алгоритма радиус случайной генерации точек сокращается.

В результате работы алгоритма, описанного в предыдущем разделе, мы получаем траекторию, представляющую собой ломаную линию с одним поворотом. Во время решения задачи безусловной оптимизации курса были проведены испытания по рекурсивному дроблению получаемой траектории и оптимизации курса на получаемых отрезках ломаной, которые показали, что в таком случае траектория стремится к уже полученному пути с одним поворотом. Поэтому для безусловной оптимизации задачу можно считать решенной с допустимым приближением.

# Обработка ограничений (обход препятствия)

Задача поиска оптимального курса в данной постановке осложняется наличием препятствий на пути судна. Препятствие реализовано в виде окружности, внутреннее пространство которой является для судна запретной зоной. Таким образом задача переходит в категорию задач условной оптимизации.

Для учета ограничений используется метод барьерных функций. Он заключается в модификации целевой функции таким образом, что ее значение стремительно возрастает около границ области ограничения (на рис. 1 пример для одномерной оптимизации).



*Рисунок 1. Иллюстрация метода штрафных функций*

Целевая функция модифицирована следующим образом. Пусть – координаты центра препятствия, *r* – радиус препятствия, – координаты точки поворота (варьируемые параметры целевой функции Ф(*x,y*)). Тогда

После первого выполнения метода оптимизации и получения оптимальной траектории выполняется проверка, не проходят ли отрезки этой траектории через препятствие. Если отрезок ломаной проходит через запретную зону, то для него рекурсивно повторно выполняется оптимизация курса до тех пор, пока все отрезки общей траектории будут лежать вне области препятствия. В результате, возможны следующие случаи:

* На территории препятствия находится точка отправления или точка назначения – ошибочные входные данные;
* Прямой курс без поворотов является оптимальным и не проходит через препятствие;
* Прямой курс проходит через препятствие, но оптимальный курс с одним поворотом его избегает;
* Точка поворота для оптимального курса с одним поворотом лежит вне препятствия, но один или оба отрезка ломаной его «задевают». В этом случае судно, упираясь в запретную зону, огибает ее по границе.

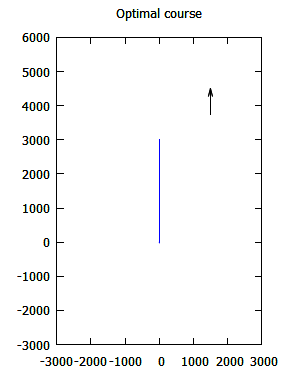
# Средства реализации

Для выполнения требуемых расчетов была разработана программа на языке C++ с использованием библиотек Qt. Для визуализации результатов была использована открытая программа Gnuplot.

# Результаты работы алгоритма

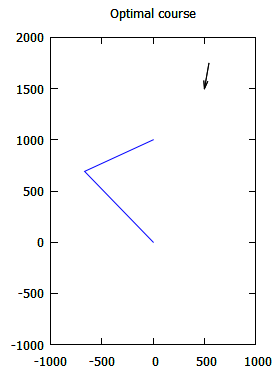
Алгоритм был протестирован при различных направлениях ветра и положениях препятствия. Ниже представлены некоторые случаи. Во всех приведенных ниже примерах судно следует из точки (0,0) строго на север (на схемах ниже северное направление сверху) и ориентировано в ту же сторону. Направление ветра обозначено черной стрелкой. Масса судна – 75 кг, площадь паруса – 10 кв. м.

1. Ветер дует по ходу судна, прямой курс является оптимальным (рис. 2).



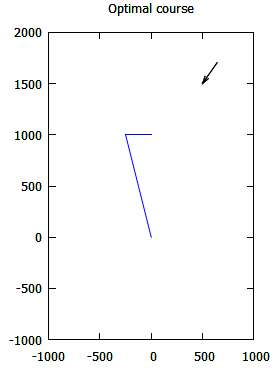
*Рисунок 2*

1. Ветер дует под слишком острым углом к ходу судна, чтобы идти прямым курсом, выбран оптимальный курс с одним поворотом (рис. 3):



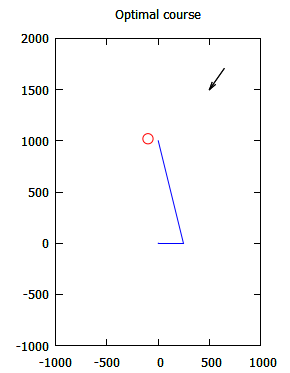
*Рисунок 3*

1. Ветер позволяет идти прямым курсом, однако, он не оптимален. Выбран курс с одним поворотом (рис. 4).



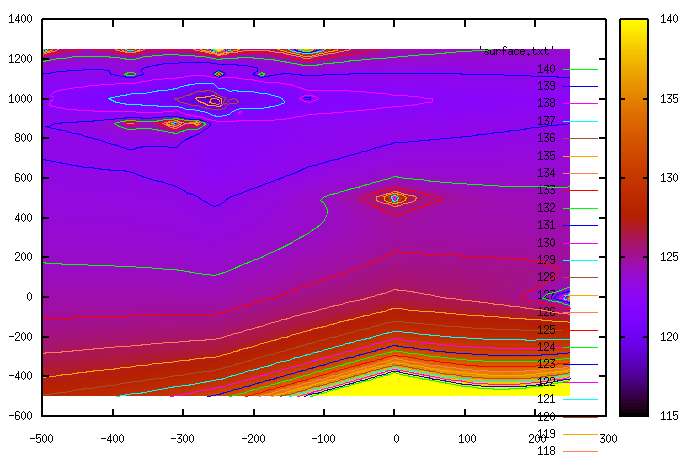
*Рисунок 4*

1. К условиям примера 3 добавилась зона ограничения, выбран другой курс, не попадающий в нее (рис. 5).



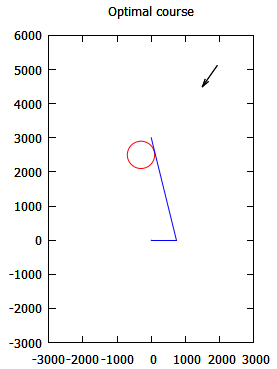
*Рисунок 5*

На диаграмме (рис. 6) значений целевой функции в области определения для примера 3 (и рисунка 4 соответственно) заметно 2 локальных минимума (наиболее темные области). После ввода ограничения в примере 4 в районе одного из них, оптимальный курс стал проходить точку поворота, соответствующую другому минимуму.



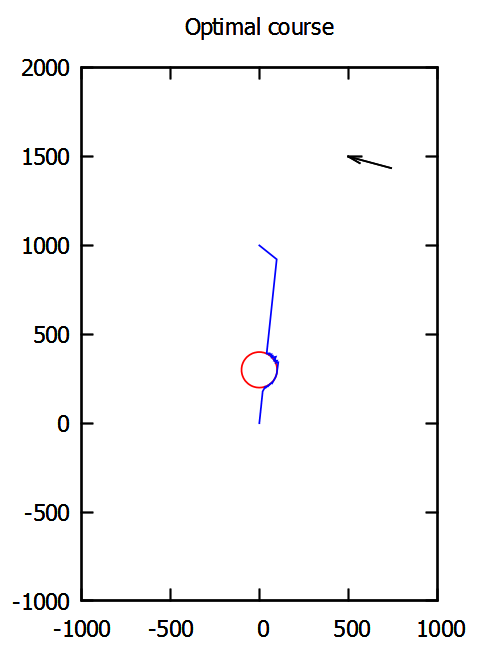
*Рисунок 6*

1. Ограничение присутствует на карте, но оптимальный курс, построенные для безусловной задачи, не проходит через него, поэтому не корректируется (рис. 7).



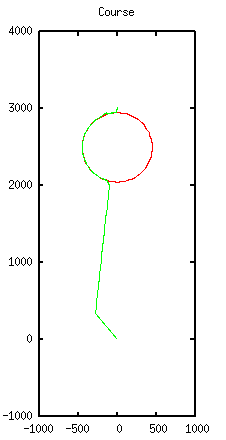
*Рисунок 7*

1. Оптимальный курс проходит через препятствие, и судно огибает его (рис. 8).



*Рисунок 8*

1. Судно огибает обширное препятствие вблизи точки назначения, метеорологическое направление ветра, или азимут точки, откуда дует ветер – 105° (рис. 9):



*Рисунок 9*

# Заключение

Представленный метод решения задачи основан на значительном числе упрощений, которые, однако, не повлияли на принципиальную правильность полученных решений – построенные курсы достаточно точно соответствуют курсам, которые прокладываются на практике, что свидетельствует о допустимости принятых упрощений в рамках данной постановки задачи. Разработанные на языке C++ классы могут стать основой для дальнейших работ, в данной предметной области.

# Использованные источники

1. Леонтьев Е. П. «Школа яхтейного капитана», М. «Физкультура и спорт», 1983
2. В.Н. Белоозеров. [http://parusa.narod.ru/bib/books/kuzn/03.htm](http://parusa.narod.ru/bib/books/kuzn/03.htm" \t "_blank)
3. Карпенко А.П. Методы оптимизации <http://bigor.bmstu.ru/?cnt/?doc=MO/base.cou>
4. Методы оптимизации <http://www.theweman.info/index.html>
5. Жиглявский А.А., Жилинскас А.Г. Методы поиска глобального экстремума. — М.: Наука, 1991.

# Приложение 1. Листинг кода программы

Файл nature.h:

//Simulation of physical conditions

#ifndef NATURE\_H

#define NATURE\_H

#include <QString>

class Nature

{

double airWeight;

double waterWeight;

double windVelocity;

double windAngle;

double k;

bool useWindB;

double gAcceleration;

double PI;

public:

Nature();

void setK(double kk) { k = kk; }

void setWindKind(bool a) { useWindB = a; }

void setwindVelocity(double windVel) { windVelocity = windVel; }

double getWindVelocity() { return windVelocity; }

double getK() { return k; }

double getAirWeight () {return airWeight;}

double getWaterWeight () {return waterWeight;}

double getWindAngleN() {return windAngle;}

void setWindAngle(double angle) { windAngle = angle; }

double add360(double angle1, double angle2);

double getWindVelocityN ();

double sub360(double angle1, double angle2);

double getWindAngleToBoat (double azimuth);

double windToAzimuth (double wind);

double getWindAngle(double boatVelocity, double azimuth);

double getWindVelocity(double boatVelocity, double azimuth);

double getWindVelocityB (double boatVelocity, double azimuth);

double getAngleB (double boatVelocity, double azimuth);

QString getWindToBoatOrientation(double azimuth);

double getFrictionForce (double cx, double sailSurface, double windVel);

double getTractionForce (double cy, double sailSurface, double windVel);

double getWaterResistForce (double boatVelocity, double s);

double toRadians (double angle);

double toDegrees(double angle);

double getSafeArcsin (double sinOfAngle);

double getSafeArccos(double cosOfAngle);

double getCosFromDegrees(double angle);

double getSinFromDegrees(double angle);

};

#endif // NATURE\_H

Файл nature.cpp:

#include "nature.h"

#include <cmath>

Nature::Nature ()

{

airWeight = 1.29;

waterWeight = 1000;

windVelocity = 10;

windAngle = 0;

k = 2.0 \* 0.001;

useWindB = false;

PI = 3.1415;

gAcceleration = 9.8;

}

double Nature::add360(double angle1, double angle2)

{

double res = angle1 + angle2;

if (res < 0) res += 360;

if (res > 360) res -= 360;

return res;

}

double Nature::getWindVelocityN ()

{

return windVelocity;

}

double Nature::sub360(double angle1, double angle2)

{

double res = angle1 - angle2;

if (res < 0) res += 360;

if (res > 360) res -= 360;

return res;

}

double Nature::getWindAngleToBoat (double azimuth)

{

return sub360(azimuth, windAngle);

}

double Nature::windToAzimuth (double wind)

{

return add360(wind, 180);

}

double Nature::getWindAngle(double boatVelocity, double azimuth)

{

if (useWindB) return getAngleB(boatVelocity, azimuth);

else return getWindAngleN();

}

double Nature::getWindVelocity(double boatVelocity, double azimuth)

{

if (useWindB) return getWindVelocityB(boatVelocity, azimuth);

else return getWindVelocityN();

}

double Nature::getWindVelocityB(double boatVelocity, double azimuth)

{

double windAngleToBoat = getWindAngleToBoat(azimuth);

return sqrt(boatVelocity \* boatVelocity + windVelocity \* windVelocity

- 2 \* boatVelocity \* windVelocity \* getCosFromDegrees(180 - windAngleToBoat));

}

double Nature::getAngleB (double boatVelocity, double azimuth)

{

if (boatVelocity == 0 || azimuth == windAngle ||

fabs(azimuth - windAngle) == 180)

return getWindAngleN();

// if (getWindAngleToBoat(azimuth) > 0)

if (getWindToBoatOrientation(azimuth) == "L")

return getSafeArccos((getWindVelocityB (boatVelocity, azimuth) \* getWindVelocityB (boatVelocity, azimuth)

+ boatVelocity \* boatVelocity - windVelocity \* windVelocity)

/ (2 \* getWindVelocityB (boatVelocity, azimuth) \* boatVelocity));

else

return 360.0 - getSafeArccos((getWindVelocityB(boatVelocity, azimuth) \* getWindVelocityB(boatVelocity, azimuth)

+ boatVelocity \* boatVelocity - windVelocity \* windVelocity)

/ (2 \* getWindVelocityB(boatVelocity, azimuth) \* boatVelocity));

}

QString Nature::getWindToBoatOrientation(double azimuth)

{

if (getWindAngleToBoat(azimuth) > 180) return "L";

else return "R";

}

double Nature::getFrictionForce (double cx, double sailSurface, double windVel)

{

return gAcceleration \* cx \* windVel \* windVel \* airWeight \* sailSurface / 2.0;

}

double Nature::getTractionForce (double cy, double sailSurface, double windVel)

{

return gAcceleration \* cy \* windVel \* windVel \* airWeight \* sailSurface / 2.0;

}

double Nature::getWaterResistForce(double boatVelocity, double s)

{

return gAcceleration \* k \* boatVelocity \* boatVelocity \* waterWeight \* s /2.0;

}

double Nature::toRadians (double angle)

{

return angle \* PI / 180.0;

}

double Nature::toDegrees(double angle)

{

return angle \* 180.0 / PI;

}

double Nature::getSafeArcsin(double sinOfAngle)

{

double deltaAngle = 0.999;

if (sinOfAngle <= deltaAngle)

return toDegrees(asin(sinOfAngle));

else

return toDegrees(PI / 2.0);

}

double Nature::getSafeArccos(double cosOfAngle)

{

if (cosOfAngle >= 1) return 0.0;

else return toDegrees(acos(cosOfAngle));

}

double Nature::getCosFromDegrees(double angle)

{

double rad\_angle = toRadians(angle);

return cos(rad\_angle);

}

double Nature::getSinFromDegrees(double angle)

{

double rad\_angle = toRadians(angle);

return sin(rad\_angle);

}

Файл boat.h:

#ifndef BOAT\_H

#define BOAT\_H

#include "interpolation.h"

#include "nature.h"

class Boat

{

double mass;

double sailSurface;

double s;

double azimuth;

double rudderAngle;

double sailToBoatAngle;

QString sailToBoatOrientation;

double attackAngle;

QString sailSide;

double boatVelocity;

double a;

double frictionForce;

double tractionForce;

double waterResistForce;

double x;

double y;

Nature \*World;

Interpolation \*Cx;

Interpolation \*Cy;

void turnSail(QString dir);

public:

Boat(double m, double sailSurf, double s1, double azimuth1, Nature \*N);

double getMass () {return mass;}

double getSailSurface () {return sailSurface;}

double getS () {return s;}

double getAzimuth () {return azimuth;}

double getX () {return x;}

double getY() {return y;}

double getBoatVelocity() { return boatVelocity;}

double getA() {return a;}

double getSailAngle() { return sailToBoatAngle; }

double getFrictionForce() { return frictionForce; }

double getTractionForce() { return tractionForce; }

double getWaterResistForce() { return waterResistForce; }

QString getSailSide() { return sailSide; }

double getAttackAngle() { return attackAngle; }

QString getSailToBoatOrientation() { return sailToBoatOrientation; }

double getRudderAngle() { return rudderAngle; }

void setSailSurface(double s) { sailSurface = s; }

void setS(double s1) { s = s1; }

void setMass(double m) { mass = m; }

void setAzimuth (double az) { azimuth = az; }

void setBoatVelocity (double v) { boatVelocity = v;}

void setA (double a1) {a = a1;}

void setSailToBoatAngle (double angle) { sailToBoatAngle = angle;}

void setSailToBoatOrientation(QString orientation) { sailToBoatOrientation = orientation; }

void setX(double x1) { x = x1;}

void setY(double y1) { y = y1;}

void setRudderAngle(double angle) { rudderAngle = angle; }

double decRudderAngle();

double incRudderAngle();

void optimizeSailAngle (int windAngle);

double boatSpeed ();

double sailAzimuth();

void updateSailOrientation();

void calculateForces();

void step();

void changeAzimuth();

void turnSailRight() { turnSail("R"); }

void turnSailLeft() { turnSail("L"); }

};

#endif // BOAT\_H

Файл boat.cpp

#include "boat.h"

Boat::Boat (double m, double sailSurf, double s1, double azimuth1, Nature \*N)

{

World = N;

mass = m;

sailSurface = sailSurf;

s = s1;

azimuth = azimuth1;

boatVelocity = 0;

a = 0;

x = 0.0;

y = 0.0;

sailToBoatAngle = 0;

sailToBoatOrientation = "0";

updateSailOrientation();

rudderAngle = 0;

QVector<double> \*cx = new QVector<double>();

cx->push\_back(0.1);

cx->push\_back(0.1);

cx->push\_back(0.15);

cx->push\_back(0.18);

cx->push\_back(0.23);

cx->push\_back(0.3);

cx->push\_back(0.4);

cx->push\_back(0.6);

cx->push\_back(0.7);

cx->push\_back(0.8);

cx->push\_back(0.85);

cx->push\_back(0.93);

cx->push\_back(1);

cx->push\_back(1.07);

cx->push\_back(1.12);

cx->push\_back(1.19);

cx->push\_back(1.2);

cx->push\_back(1.23);

cx->push\_back(1.25);

cx->push\_back(1.23);

cx->push\_back(1.2);

cx->push\_back(1.19);

cx->push\_back(1.12);

cx->push\_back(1.07);

cx->push\_back(1);

cx->push\_back(0.93);

cx->push\_back(0.85);

cx->push\_back(0.8);

cx->push\_back(0.7);

cx->push\_back(0.6);

cx->push\_back(0.4);

cx->push\_back(0.3);

cx->push\_back(0.23);

cx->push\_back(0.18);

cx->push\_back(0.15);

cx->push\_back(0.1);

cx->push\_back(0.1);

// fprintf(stdout, "%d\n", cx->size());

Cx = new Interpolation (cx, cx->size(), 5, 0, 180);

QVector<double> \*cy = new QVector<double>();

cy->push\_back(0);

cy->push\_back(0.3);

cy->push\_back(0.5);

cy->push\_back(0.9);

cy->push\_back(1.2);

cy->push\_back(1.4);

cy->push\_back(1.6);

cy->push\_back(1.3);

cy->push\_back(1.14);

cy->push\_back(1.09);

cy->push\_back(1);

cy->push\_back(0.87);

cy->push\_back(0.78);

cy->push\_back(0.66);

cy->push\_back(0.55);

cy->push\_back(0.4);

cy->push\_back(0.3);

cy->push\_back(0.2);

cy->push\_back(0);

cy->push\_back(0.3);

cy->push\_back(0.35);

cy->push\_back(0.4);

cy->push\_back(0.45);

cy->push\_back(0.5);

cy->push\_back(0.6);

cy->push\_back(0.8);

cy->push\_back(1.0);

cy->push\_back(1.2);

cy->push\_back(1.1);

cy->push\_back(0.9);

cy->push\_back(0.8);

cy->push\_back(0.7);

cy->push\_back(0.4);

cy->push\_back(0.35);

cy->push\_back(0.2);

cy->push\_back(0.1);

cy->push\_back(0);

Cy = new Interpolation (cy, cy->size(), 5, 0, 180);

}

double Boat::decRudderAngle()

{

if (rudderAngle > -30) --rudderAngle;

return rudderAngle;

}

double Boat::incRudderAngle()

{

if (rudderAngle < 30) ++rudderAngle;

return rudderAngle;

}

void Boat::optimizeSailAngle (int windAngle)

{

double speed = 0;

azimuth = 0;

World->setWindAngle(windAngle);

sailToBoatOrientation = "L";

sailToBoatAngle = 0;

double nextSpeed = boatSpeed();

do

{

sailToBoatAngle += 5;

speed = nextSpeed;

nextSpeed = boatSpeed();

} while (speed <= nextSpeed && sailToBoatAngle < 90);

sailToBoatAngle -= 5;

double leftMax = sailToBoatAngle;

double leftMaxVelocity = speed;

sailToBoatOrientation = "R";

sailToBoatAngle = 0;

nextSpeed = boatSpeed();

do

{

sailToBoatAngle += 5;

speed = nextSpeed;

nextSpeed = boatSpeed();

} while (speed <= nextSpeed && sailToBoatAngle < 90);

sailToBoatAngle -= 5;

double rightMax = sailToBoatAngle;

double rightMaxVelocity = speed;

if (leftMaxVelocity >= rightMaxVelocity)

{

sailToBoatAngle = leftMax;

sailToBoatOrientation = "L";

boatVelocity = leftMaxVelocity;

}

else

{

sailToBoatAngle = rightMax;

sailToBoatOrientation = "R";

boatVelocity = rightMaxVelocity;

}

}

double Boat::boatSpeed ()

{

double delta\_a = 0.001;

boatVelocity = 0;

a = 0;

do

{

step();

} while (a > delta\_a);

return boatVelocity;

}

double Boat::sailAzimuth()

{

if (sailToBoatOrientation == "L")

return World->add360(azimuth, sailToBoatAngle);

else

return World->sub360(azimuth, sailToBoatAngle);

}

void Boat::updateSailOrientation()

{

attackAngle = World->sub360(sailAzimuth(), World->getWindAngle(boatVelocity, azimuth));

if (attackAngle > 180)

{

sailSide = "Out";

attackAngle = 360 - attackAngle;

}

else sailSide = "In";

if (sailToBoatOrientation == "L")

{

if (sailSide == "In") sailSide = "Out";

else sailSide = "In";

}

}

void Boat::calculateForces()

{

waterResistForce = -World->getWaterResistForce(boatVelocity, s);

if (boatVelocity < 0) waterResistForce \*= -1;

frictionForce = World->getFrictionForce(Cx->get(attackAngle), sailSurface, World->getWindVelocity(boatVelocity, azimuth));

tractionForce = World->getTractionForce(Cy->get(attackAngle), sailSurface, World->getWindVelocity(boatVelocity, azimuth));

if (getSailSide() == "Out") tractionForce \*= -1;

}

void Boat::step()

{

changeAzimuth();

updateSailOrientation();

calculateForces();

a = (

World->getSinFromDegrees(sailToBoatAngle) \* tractionForce

- frictionForce \* World->getCosFromDegrees(World->getWindAngleToBoat(azimuth))

+ waterResistForce

) / mass;

double delta\_t = 0.001;

// It seems, this calculations were correct for SCREEN coords, not for REAL

// x += (boatVelocity \* delta\_t + a \* delta\_t \* delta\_t / 2) \* World->getSinFromDegrees(azimuth) / 100000.0;

// y -= (boatVelocity \* delta\_t + a \* delta\_t \* delta\_t / 2) \* World->getCosFromDegrees(azimuth) / 100000.0;

x += (boatVelocity \* delta\_t + a \* delta\_t \* delta\_t / 2) \* World->getSinFromDegrees(azimuth);

y += (boatVelocity \* delta\_t + a \* delta\_t \* delta\_t / 2) \* World->getCosFromDegrees(azimuth);

boatVelocity += a \* delta\_t;

}

void Boat::changeAzimuth()

{

double c = 2000;

double delta\_azimuth = boatVelocity / 100000.0 \* rudderAngle \* c;

azimuth = World->add360(azimuth, delta\_azimuth);

}

void Boat::turnSail(QString dir)

{

QString opdir = "L";

if (dir == "L") opdir = "R";

if (sailToBoatOrientation == dir && sailToBoatAngle < 90)

++sailToBoatAngle;

else if (sailToBoatOrientation == "0")

{

sailToBoatAngle = 1;

sailToBoatOrientation = dir;

}

else if (sailToBoatOrientation == opdir)

{

--sailToBoatAngle;

if (sailToBoatAngle == 0) sailToBoatOrientation = "0";

}

}

Файл interpolation.h:

//A class to interpolate phase characteristics of boat coefficients

#ifndef INTERPOLATION\_H

#define INTERPOLATION\_H

#include <QVector>

class Interpolation

{

QVector<double> \*table;

// int n;

int step;

double rightLimit;

double leftLimit;

public:

Interpolation(QVector<double> \*c, int n, int s, double ll, double rl);

double get (double angle);

};

#endif // INTERPOLATION\_H

Файл interpolation.cpp:

#include "interpolation.h"

Interpolation::Interpolation(QVector<double> \*c, int n, int s, double ll, double rl)

{

step = s;

leftLimit = ll;

rightLimit = rl;

// table = new double[n];

table = new QVector<double>();

for (int i = 0; i < n; i++)

{

// Table.SetValue(c.GetValue(i), i);

table->push\_back(c->value(i));

}

}

double Interpolation::get(double angle)

{

if ( ! (leftLimit <= angle && angle <= rightLimit) )

return 0;

// int i = Convert.ToInt32((angle - leftLimit) / step);

int i = (int)((angle - leftLimit) / step);

if (angle == (angle / step) \* step)

// return Convert.ToDouble(table.GetValue(i));

return table->value(i);

if (angle > 175)

//return Convert.ToDouble(table.GetValue(36));

return table->value(36);

// int angle1 = Convert.ToInt32(i \* step);

int angle1 = (int)(i \* step);

int angle2 = angle1 + step;

// double c1 = Convert.ToDouble(Table.GetValue(i));

double c1 = table->value(i);

// double c2 = Convert.ToDouble(Table.GetValue(i + 1));

double c2 = table->value(i + 1);

return ((c2 - c1) \* (angle - angle1)) / (angle2 - angle1) + c1;

}

Файл main.cpp:

#include <QCoreApplication>

#include <iostream>

#include <iomanip>

#include <cmath>

#include <cstdlib>

#include <ctime>

#include <fstream>

#include "boat.h"

#define RESOLUTION 4

#define DELTA\_X 2

#define DELTA\_F 0.1

#define k 2.0

#define RANDOM\_POINTS 20

#define VERTEX\_NUM 5

#define WAYPOINTS\_FILE "waypoints.txt"

#define PRINT\_WAYPOINTS

//#define USE\_ADVANCED\_METHOD

double mass = 75, sailSurface = 10, s = 2, fullTurnTime = 15;

Nature \*world;

Boat \*boat;

int windAngle;

std::ofstream waypoints;

double fmin2 (double a, double b)

{

if(a > b) return b;

else return a;

}

double fmax2 (double a, double b)

{

if(a > b) return a;

else return b;

}

double fmin3 (double a, double b, double c)

{

return fmin2(fmin2(a, b), c);

}

double fmax3 (double a, double b, double c)

{

return fmax2(fmax2(a, b), c);

}

// Проверка вхождения точки в пределы многоугольника (1 если входит, 0 если нет)

int pointInPoly( double pgon[][2], int numverts, double\* point )

{

int i, crossings = 0;

for ( i = 0; i < numverts; i++ ) {

double x1 = pgon[i][0];

double y1 = pgon[i][1];

double x2 = pgon[(i + 1) % numverts][0];

double y2 = pgon[(i + 1) % numverts][1];

double d = (point[1] - y1) \* (x2 - x1) - (point[0] - x1) \* (y2 - y1);

if ( (y1 >= point[1]) != (y2 >= point[1]) ) {

crossings += y2 - y1 >= 0 ? d >= 0 : d <= 0;

}

if ( !d && fmin2( x1, x2 ) <= point[0] && point[0] <= fmax2( x1, x2 )

&& fmin2( y1, y2 ) <= point[1] && point[1] <= fmax2( y1, y2 ) ) {

return 1;

}

}

return crossings & 1;

}

// Проверка пересечения отрезков

bool intersectLines (double\* a, double\* b, double\* c, double\* d)

{

double common = (b[0] - a[0])\*(d[1] - c[1]) - (b[1] - a[1])\*(d[0] - c[0]);

if (common == 0) return false;

double rH = (a[1] - c[1])\*(d[0] - c[0]) - (a[0] - c[0])\*(d[1] - c[1]);

double sH = (a[1] - c[1])\*(b[0] - a[0]) - (a[0] - c[0])\*(b[1] - a[1]);

double r = rH / common;

double s = sH / common;

if (r >= 0 && r <= 1 && s >= 0 && s <= 1)

return true;

else

return false;

}

// Проверка пересечения отрезка и окружности

bool intersectLineAndCircle ( double\* point1, double\* point2, double\* center, double radius)

{

double p1[2], p2[2];

p1[0] = point1[0] - center[0];

p1[1] = point1[1] - center[1];

p2[0] = point2[0] - center[0];

p2[1] = point2[1] - center[1];

double dx = p2[0] - p1[0];

double dy = p2[1] - p1[1];

//составляем коэффициенты квадратного уравнения на пересечение прямой и окружности.

//если на отрезке [0..1] есть отрицательные значения, значит отрезок пересекает окружность

double a = dx\*dx + dy\*dy;

double b = 2.\*(p1[0]\*dx + p1[1]\*dy);

double c = p1[0]\*p1[0] + p1[1]\*p1[1] - radius\*radius;

//а теперь проверяем, есть ли на отрезке [0..1] решения

if (-b < 0)

return (c < 0);

if (-b < (2.\*a))

return ((4.\*a\*c - b\*b) < 0);

return (a+b+c < 0);

}

// Вычисление геометрического ограничения (окружность)

double calcCircleRestriction(double\* point, double\* center, double radius)

{

double r = (point[0] - center[0])\*(point[0] - center[0]) + (point[1] - center[1])\*(point[1] - center[1]) - radius\*radius;

return r;

}

// Вычисление барьерной функции

double barrierFunction(double\* point, double\* center, double radius)

{

double g = calcCircleRestriction(point, center, radius);

double phi = 0;

double n = 2;

if(g >= 0)

phi = 1 / pow(g, n);

else

phi = INFINITY;

return phi;

}

// Расстояние между точками

double calcDistance(double\* startPoint, double\* destPoint) {

return sqrt((destPoint[1] - startPoint[1])\*(destPoint[1] - startPoint[1]) + (destPoint[0] - startPoint[0])\*(destPoint[0] - startPoint[0]));

}

// Азимут от точки до точки

int calcAzimuth(double\* startPoint, double\* destPoint) {

int azimuth = world->sub360(90, world->toDegrees(acos((destPoint[0] - startPoint[0]) / calcDistance(startPoint, destPoint))));

if(destPoint[1] < startPoint[1])

azimuth = world->sub360(-1 \* azimuth, 180);

return azimuth;

}

// Угол ветра относительно азимута

int calcRelativeWindAngle(double\* startPoint, double\* destPoint)

{

return world->sub360(windAngle, calcAzimuth(startPoint, destPoint));

}

// Время прямого пути от точки до точки

double calcTravelTime(double\* startPoint, double\* destPoint)

{

boat->optimizeSailAngle(calcRelativeWindAngle(startPoint, destPoint));

double travelTime = calcDistance(startPoint, destPoint) / boat->getBoatVelocity();

if(travelTime < 0)

{

std::cout<<"Whoops! velocity is "<<boat->getBoatVelocity()<<

" on ["<<startPoint[0]<<";"<<startPoint[1]<<"] => ["<<destPoint[0]<<";"<<destPoint[1]<<"]"<<

" when rel. wind angle is "<<calcRelativeWindAngle(startPoint, destPoint)<<"\n";

}

return travelTime;

}

// Расчет времени поворота

double calcTurnTime(double\* startPoint, double\* turnPoint, double\* destPoint)

{

return ( abs(calcAzimuth(turnPoint, destPoint) - calcAzimuth(startPoint, turnPoint)) / 360.0 ) \* fullTurnTime;

}

// Общее время пути до и после поворота

double calcTotalTime(double\* startPoint, double\* turnPoint, double\* destPoint)

{

return calcTravelTime(startPoint, turnPoint) + calcTravelTime(turnPoint, destPoint) + calcTurnTime(startPoint, turnPoint, destPoint);

}

double calcTotalTimeWithRestriction(double\* startPoint, double\* turnPoint, double\* destPoint, double\* islandCenter, double islandRadius)

{

return calcTotalTime(startPoint, turnPoint, destPoint) + barrierFunction(turnPoint, islandCenter, islandRadius);

}

// Проверка на слишком острый курс

bool tooSharpCourse(double\* startPoint, double\* destPoint)

{

int relativeWindAngle = calcRelativeWindAngle(startPoint, destPoint);

if(relativeWindAngle < 30 || relativeWindAngle > 330)

return true;

else

return false;

}

// Определение оптимальной точки поворота простым способом

void getTurnPointSimple(double\* bestTurnPoint, double\* startPoint, double\* destPoint, double badTravelTime, double\* islandCenter, double islandRadius)

{

//Method I (равномерный поиск)

double totalTime = badTravelTime;

double bestTime;

double curTurnPoint[2];

bestTurnPoint[0] = destPoint[0];

bestTurnPoint[1] = destPoint[1];

bool firstIter = true;

double leftBound, rightBound, upBound, downBound;

double margin = calcDistance(startPoint, destPoint) / 2;

if (startPoint[0] < destPoint[0])

{

leftBound = startPoint[0] - margin;

rightBound = destPoint[0] + margin;

}

else

{

leftBound = destPoint[0] - margin;

rightBound = startPoint[0] + margin;

}

if (startPoint[1] < destPoint[1])

{

downBound = startPoint[1] - margin;

upBound = destPoint[1] + margin;

}

else

{

downBound = destPoint[1] - margin;

upBound = startPoint[1] + margin;

}

double step\_x = fabs((rightBound - leftBound) / RESOLUTION);

double step\_y = fabs((upBound - downBound) / RESOLUTION);

std::ofstream surface;

surface.open("surface.txt", std::ios::out);

surface<<std::fixed<<std::setprecision(2);

while(step\_x > DELTA\_X/2 && step\_y > DELTA\_X/2)

{

for (curTurnPoint[0] = leftBound; curTurnPoint[0] < rightBound; curTurnPoint[0]+=step\_x)

for (curTurnPoint[1] = downBound; curTurnPoint[1] < upBound; curTurnPoint[1]+=step\_y)

{

if(tooSharpCourse(startPoint, curTurnPoint) || tooSharpCourse(curTurnPoint, destPoint))

{

continue;

}

//totalTime = calcTotalTime(startPoint, curTurnPoint, destPoint);

totalTime = calcTotalTimeWithRestriction(startPoint, curTurnPoint, destPoint, islandCenter, islandRadius);

surface<<curTurnPoint[0]<<"\t"<<curTurnPoint[1]<<"\t"<<totalTime<<"\n\n";

if(firstIter)

{

bestTime = totalTime;

firstIter = false;

}

if (totalTime < bestTime)

{

bestTime = totalTime;

bestTurnPoint[0] = curTurnPoint[0];

bestTurnPoint[1] = curTurnPoint[1];

}

}

//surface<<"\n";

leftBound = bestTurnPoint[0] - step\_x;

rightBound = bestTurnPoint[0] + step\_x;

upBound = bestTurnPoint[1] + step\_y;

downBound = bestTurnPoint[1] - step\_y;

step\_x = fabs((rightBound - leftBound) / RESOLUTION);

step\_y = fabs((upBound - downBound) / RESOLUTION);

}

surface.close();

}

// Определение оптимальной точки поворота

void getTurnPoint(double\* bestTurnPoint, double\* startPoint, double\* destPoint, double badTravelTime, double\* islandCenter, double islandRadius)

{

// Method II (случ. поиск с пост. радиусом + метод Паулла)

double totalTime = badTravelTime;

double bestTime;

double curTurnPoint[2];

bestTurnPoint[0] = destPoint[0];

bestTurnPoint[1] = destPoint[1];

bool firstIter = true;

double radius;

double prevTotalTime = 0;

int azimuth = 0;

double bestAzimuth = 0;

double point1[2], point2[2], point3[2], sqRootPoint[2], prevBestPoint[2];

double time1 = 0, time2 = 0, time3 = 0;

double x1=0, x2=0, x3=0, xSqRoot=0, e23=0, e13=0, e12=0, r23=0, r13=0, r12=0;

int sign=1;

double prevTurnPoint[2];

sqRootPoint[0] = startPoint[0];

sqRootPoint[1] = startPoint[1];

//srand(time(NULL));

double radius\_k = 25;

do

{

prevTotalTime = totalTime;

prevTurnPoint[0] = sqRootPoint[0];

prevTurnPoint[1] = sqRootPoint[1];

radius = calcDistance(startPoint, destPoint) / radius\_k;

firstIter = true;

/\*

double a[2] = {0, 0}, b[2];

int real\_az = 0;

for(;;)

{

azimuth = rand() % 360 ;

b[0] = a[0] + 100 \* world->getCosFromDegrees(world->sub360(90, azimuth));

b[1] = a[1] + 100 \* world->getSinFromDegrees(world->sub360(90, azimuth));

real\_az = calcAzimuth(a, b);

}

\*/

for(int i = 0; i < RANDOM\_POINTS; i++) //выбираем направление поиска из случайных точек на окружности

{

azimuth = rand() % 360 ;

curTurnPoint[0] = prevTurnPoint[0] + radius \* world->getCosFromDegrees(world->sub360(90, azimuth));

curTurnPoint[1] = prevTurnPoint[1] + radius \* world->getSinFromDegrees(world->sub360(90, azimuth));

//double real\_az = calcAzimuth(prevTurnPoint, curTurnPoint);

if(tooSharpCourse(startPoint, curTurnPoint) || tooSharpCourse(curTurnPoint, destPoint))

{

continue;

}

totalTime = calcTotalTimeWithRestriction(startPoint, curTurnPoint, destPoint, islandCenter, islandRadius);

if(firstIter)

{

bestTime = totalTime;

firstIter = false;

}

if (totalTime < bestTime)

{

bestTime = totalTime;

bestAzimuth = azimuth;

bestTurnPoint[0] = curTurnPoint[0];

bestTurnPoint[1] = curTurnPoint[1];

}

}

if (bestTime > prevTotalTime)

{

radius\_k = radius\_k / 2;

continue;

}

point1[0] = bestTurnPoint[0]; //метод Паулла

point1[1] = bestTurnPoint[1];

point2[0] = point1[0] + k \* radius \* world->getCosFromDegrees(90 - bestAzimuth);

point2[1] = point1[1] + k \* radius \* world->getSinFromDegrees(90 - bestAzimuth);

time1 = calcTotalTimeWithRestriction(startPoint, point1, destPoint, islandCenter, islandRadius);

time2 = calcTotalTimeWithRestriction(startPoint, point2, destPoint, islandCenter, islandRadius);

if (time1 > time2)

{

point3[0] = point1[0] + 2 \* k \* radius \* world->getCosFromDegrees(90 - bestAzimuth);

point3[1] = point1[1] + 2 \* k \* radius \* world->getSinFromDegrees(90 - bestAzimuth);

}

else

{

point3[0] = point1[0] - k \* radius \* world->getCosFromDegrees(90 - bestAzimuth);

point3[1] = point1[1] - k \* radius \* world->getSinFromDegrees(90 - bestAzimuth);

}

do

{

time1 = calcTotalTimeWithRestriction(startPoint, point1, destPoint, islandCenter, islandRadius);

time2 = calcTotalTimeWithRestriction(startPoint, point2, destPoint, islandCenter, islandRadius);

time3 = calcTotalTimeWithRestriction(startPoint, point3, destPoint, islandCenter, islandRadius);

if(time1 == fmin3(time1, time2, time3))

{

prevBestPoint[0] = point1[0];

prevBestPoint[1] = point1[1];

}

if(time2 == fmin3(time1, time2, time3))

{

prevBestPoint[0] = point2[0];

prevBestPoint[1] = point2[1];

}

if(time3 == fmin3(time1, time2, time3))

{

prevBestPoint[0] = point3[0];

prevBestPoint[1] = point3[1];

}

//if(fabs(calcAzimuth(prevTurnPoint, point1) - bestAzimuth) > 90) sign = -1;

if(abs(calcAzimuth(prevTurnPoint, point1) - bestAzimuth) > 2) sign = -1;

else sign = 1;

x1 = sign \* calcDistance(prevTurnPoint, point1);

if(abs(calcAzimuth(prevTurnPoint, point2) - bestAzimuth) > 2) sign = -1;

else sign = 1;

x2 = sign \* calcDistance(prevTurnPoint, point2);

if(abs(calcAzimuth(prevTurnPoint, point3) - bestAzimuth) > 2) sign = -1;

else sign = 1;

x3 = sign \* calcDistance(prevTurnPoint, point3);

e23 = x2\*x2 - x3\*x3;

e13 = x1\*x1 - x3\*x3;

e12 = x1\*x1 - x2\*x2;

r23 = x2 - x3;

r13 = x1 - x3;

r12 = x1 - x2;

xSqRoot = (time1\*e23 - time2\*e13 + time3\*e12) / ((time1\*r23 - time2\*r13 + time3\*r12) \* 2.0);

sqRootPoint[0] = prevTurnPoint[0] + xSqRoot \* world->getCosFromDegrees(90 - bestAzimuth);

sqRootPoint[1] = prevTurnPoint[1] + xSqRoot \* world->getSinFromDegrees(90 - bestAzimuth);

totalTime = calcTotalTimeWithRestriction(startPoint, sqRootPoint, destPoint, islandCenter, islandRadius);

if (totalTime > fmin3(time1, time2, time3) || totalTime < 0)

{

totalTime = fmin3(time1, time2, time3);

sqRootPoint[0] = prevBestPoint[0];

sqRootPoint[1] = prevBestPoint[1];

break;

}

if(xSqRoot >= x2 && xSqRoot <= x3)

{

point1[0] = point2[0];

point1[1] = point2[1];

point2[0] = sqRootPoint[0];

point2[1] = sqRootPoint[1];

point3[0] = point3[0];

point3[1] = point3[1];

continue;

}

if(xSqRoot >= x1 && xSqRoot <= x2)

{

point1[0] = point1[0];

point1[1] = point1[1];

point3[0] = point2[0];

point3[1] = point2[1];

point2[0] = sqRootPoint[0];

point2[1] = sqRootPoint[1];

continue;

}

if(xSqRoot >= x3 && xSqRoot <= x1)

{

double temp[2];

temp[0] = point1[0];

temp[1] = point1[1];

point1[0] = point3[0];

point1[1] = point3[1];

point2[0] = sqRootPoint[0];

point2[1] = sqRootPoint[1];

point3[0] = temp[0];

point3[1] = temp[1];

continue;

}

if(xSqRoot > x3 && x3 > x2)

{

point1[0] = point2[0];

point1[1] = point2[1];

point2[0] = point3[0];

point2[1] = point3[1];

point3[0] = sqRootPoint[0];

point3[1] = sqRootPoint[1];

continue;

}

if(xSqRoot < x3 && x3 < x1)

{

point3[0] = point2[0];

point3[1] = point2[1];

point2[0] = point1[0];

point2[1] = point1[1];

point1[0] = sqRootPoint[0];

point1[1] = sqRootPoint[1];

continue;

}

}

//while (calcDistance(point1, point3) > DELTA\_X);

while (fabs(calcTotalTimeWithRestriction(startPoint, point2, destPoint, islandCenter, islandRadius) - calcTotalTimeWithRestriction(startPoint, point3, destPoint, islandCenter, islandRadius)) > DELTA\_F);

}

while (fabs(totalTime - prevTotalTime) > DELTA\_F && totalTime < prevTotalTime && radius > DELTA\_X);

bestTurnPoint = prevTurnPoint;

}

// Метафункция для выбора метода оптимизации

void optimizeTurn(double\* bestTurnPoint, double\* startPoint, double\* destPoint, double badTravelTime, double\* islandCenter, double islandRadius)

{

// Получение курса с одним поворотом

#ifdef USE\_ADVANCED\_METHOD

getTurnPoint(bestTurnPoint, startPoint, destPoint, badTravelTime, islandCenter, islandRadius); // Метод случ. поиска + метод Паула

#endif

#ifndef USE\_ADVANCER\_METHOD

getTurnPointSimple(bestTurnPoint, startPoint, destPoint, badTravelTime, islandCenter, islandRadius); // Метод равномерного поиска

#endif

}

// Рекурсивная оптимизация курса с одним поворотом с учетом препятствия и запись точек поворота в файл

double processRestrictions(double\* startPoint, double\* turnPoint, double\* destPoint, double\* islandCenter, double islandRadius)

{

double newTurnPoint[2];

double complexTravelTime = 0;

if(intersectLineAndCircle(startPoint, turnPoint, islandCenter, islandRadius))

{

optimizeTurn(newTurnPoint, startPoint, turnPoint, 0x7ff0000000000000, islandCenter, islandRadius);

processRestrictions(startPoint, newTurnPoint, turnPoint, islandCenter, islandRadius);

}

else

complexTravelTime += calcTravelTime(startPoint, turnPoint);

waypoints<<turnPoint[0]<<"\t"<<turnPoint[1]<<"\n";

#ifdef PRINT\_WAYPOINTS

std::cout<<turnPoint[0]<<"\t"<<turnPoint[1]<<"\n";

#endif

complexTravelTime += calcTurnTime(startPoint, turnPoint, destPoint);

if(intersectLineAndCircle(turnPoint, destPoint, islandCenter, islandRadius))

{

optimizeTurn(newTurnPoint, turnPoint, destPoint, 0x7ff0000000000000, islandCenter, islandRadius);

processRestrictions(turnPoint, newTurnPoint, destPoint, islandCenter, islandRadius);

}

else

complexTravelTime += calcTravelTime(turnPoint, destPoint);

return complexTravelTime;

}

// Генерация скрипта для Gnuplot

void writeGnuplotScript(double\* startPoint, double\* destPoint, double\* islandCenter, double islandRadius)

{

double scale = calcDistance(startPoint, destPoint);

double xmin = fmin2(startPoint[0],destPoint[0]) - scale;

double xmax = fmax2(startPoint[0],destPoint[0]) + scale;

double ymin = fmin2(startPoint[1],destPoint[1]) - scale;

double ymax = fmax2(startPoint[1],destPoint[1]) + scale;

double ratio = (ymax - ymin) / (xmax - xmin);

std::ofstream script;

script.open("results.plt", std::ios::out);

//script<<"set term x11\n";

script<<"set multiplot\n";

script<<"set parametric\n";

script<<"set xrange ["<<xmin<<":"<<xmax<<"]\n";

script<<"set yrange ["<<ymin<<":"<<ymax<<"]\n";

script<<"set size ratio "<<ratio<<"\n";

script<<"set title \"Optimal course\"\n";

script<<"set arrow from "<<xmax - scale / 2 + scale / 4 \* world->getCosFromDegrees(90 - windAngle)<<","<<ymax - scale / 2 + scale / 4 \* world->getSinFromDegrees(90 - windAngle)<<" to "<<xmax - scale / 2<<","<<ymax - scale / 2<<"\n";

script<<"plot [0:2\*pi] "<<islandRadius<<"\*sin(t)+("<<islandCenter[0]<<"),"<<islandRadius<<"\*cos(t)+("<<islandCenter[1]<<") lt 1 notitle\n";

script<<"plot '"<<WAYPOINTS\_FILE<<"' w l lt 3 notitle\n";

script<<"pause mouse\n";

script.close();

}

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

{

QCoreApplication app(argc, argv);

if(argc < 7 || (argc > 7 && argc < 10))

{

std::cout<<"Usage: "<<argv[0]<<" BOAT\_AZIMUTH WIND\_METEO\_ANGLE START\_X START\_Y DESTINATION\_X DESTINATION\_Y [ISLAND\_CENTER\_X ISLAND\_CENTER\_Y ISLAND\_RADIUS]";

exit(0);

}

int azimuth = atoi(argv[1]);

windAngle = atoi(argv[2]);

double startPoint[2];

startPoint[0] = atof(argv[3]);

startPoint[1] = atof(argv[4]);

double destPoint[2];

destPoint[0] = atof(argv[5]);

destPoint[1] = atof(argv[6]);

double islandCenter[2];

double islandRadius;

if(argc > 7)

{

islandCenter[0] = atof(argv[7]);

islandCenter[1] = atof(argv[8]);

islandRadius = atof(argv[9]);

}

else

{

islandCenter[0] = 0x7ff0000000000000;

islandCenter[1] = 0x7ff0000000000000;

islandRadius = 1;

}

double travelTime = 0x7ff0000000000000;

double turnPoint[2];

world = new Nature();

boat = new Boat(mass, sailSurface, s, azimuth, world);

std::cout<<std::fixed<<std::setprecision(2);

// Проверка, не лежат ли точки отправления и назначения в запретной области

if(calcCircleRestriction(startPoint, islandCenter, islandRadius) <= 0)

{

std::cout<<"Start point is on restricted square!\n";

exit(1);

}

if(calcCircleRestriction(destPoint, islandCenter, islandRadius) <= 0)

{

std::cout<<"Destination point is on restricted square!\n";

exit(1);

}

// Вычисление времени прямого пути, если он возможен

if(!tooSharpCourse(startPoint, destPoint) && !intersectLineAndCircle(startPoint, destPoint, islandCenter, islandRadius))

{

travelTime = calcTravelTime(startPoint, destPoint);

std::cout<<"Parameters of straight course\n";

std::cout<<"Distance: "<<calcDistance(startPoint, destPoint)<<" m\n";

std::cout<<"Optimal sail angle: "<<boat->getSailAngle()<<" deg\n";

std::cout<<"Max. velocity: "<<boat->getBoatVelocity()<<" m/s\n";

std::cout<<"Estimated best travel time: "<<travelTime<<" s\n";

}

else

std::cout<<"Straight course is to sharp or would lead through the island.\n";

// Получение курса с одним поворотом

std::cout<<"\nCalculating complex optimal course...\n";

optimizeTurn(turnPoint, startPoint, destPoint, travelTime, islandCenter, islandRadius);

// Обработка обхода препятствий и сохранение точек поворота в файл

waypoints.open(WAYPOINTS\_FILE, std::ios::out);

waypoints<<std::fixed<<std::setprecision(2);

waypoints<<startPoint[0]<<"\t"<<startPoint[1]<<"\n";

#ifdef PRINT\_WAYPOINTS

std::cout<<startPoint[0]<<"\t"<<startPoint[1]<<"\n";

#endif

double complexTravelTime = processRestrictions(startPoint, turnPoint, destPoint, islandCenter, islandRadius); // Обход препятствия

waypoints<<destPoint[0]<<"\t"<<destPoint[1]<<"\n";

#ifdef PRINT\_WAYPOINTS

std::cout<<destPoint[0]<<"\t"<<destPoint[1]<<"\n";

#endif

waypoints.close();

writeGnuplotScript(startPoint, destPoint, islandCenter, islandRadius);

if(!tooSharpCourse(startPoint, destPoint) &&

!intersectLineAndCircle(startPoint, destPoint, islandCenter, islandRadius) &&

calcTravelTime(startPoint, destPoint) < complexTravelTime)

{

std::cout<<"Straight course is optimal."<<"\n";

waypoints.open(WAYPOINTS\_FILE, std::ios::out);

waypoints<<std::fixed<<std::setprecision(2);

waypoints<<startPoint[0]<<"\t"<<startPoint[1]<<"\n";

waypoints<<destPoint[0]<<"\t"<<destPoint[1]<<"\n";

}

else

{

std::cout<<"Best travel time for complex optimal course is ";

std::cout<<complexTravelTime<<" s\n";

}

std::cout<<"Optimal waypoints were saved to \'"<<WAYPOINTS\_FILE<<"\'\n";

system("gnuplot results.plt");

exit(0);

return app.exec();

}