Київський національний університет імені Тараса Шевченка

Факультет кібернетики

Кафедра обчислювальної математики

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему:

ПОБУДОВА ОПТИМАЛЬНОЇ ТРАЄКТОРІЇ РУХУ ЧОТИРИКОЛІСНОГО АВТОМОБІЛЯ ПО ПОШКОДЖЕНІЙ ДОРОЗІ

Виконав студент 6-го курсу

Башук Олександр Олексійович

Науковий керівник:

доктор фізико-математичних наук, професор

Рубльов Богдан Владиславович

Роботу заслухано на засіданні кафедри обчислювальної математики

та рекомендовано до захисту в ДЕК.

Протокол №__ від «__» _____ 2015 року.

Завідувач кафедри обчислювальної математики проф.. Ляшко С. І. ___

Київ – 2015

Зміст

Вступ

Задача автоматизації керування автомобілем займає важливе місце серед актуальних задач прикладної математики. Сферу роботизованих автомобілів без сумніву можна назвати одним з найперспективніших плодів таких розділів прикладної математики, як розпізнавання образів, машинне навчання та чисельні методи оптимізації.

Ще декілька десятиліть тому автомобілі без водіїв були лише фантастикою з художньої літератури, проте вже сьогодні ми можемо бачити перші прототипи роботизованих автомобілів. Одним з найяскравіших прикладів є розробка такого IT-гіганта, як Google, а саме Google Self-Driving Car. Автомобіль обладнано різноманітними сенсорами, які сканують оточення та передають інформацію на головний процесор. Процесор в режимі реального часу проводить обробку інформації та приймає рішення щодо подальшого маневрування автомобіля. В той час, як «під капотом» здійснюється ціла сукупність складних обчислень, робота водія зведена до мінімуму. Очевидною стає важливість розробки програмного забезпечення для таких автомобілів, оскільки майбутнє саме за ними.

Серйозність намірів щодо введення роботизованих автомобілів в експлуатацію підтверджує також те, що на сьогодні в ряді країн вже є дозвіл на дослідження таких автомобілів на дорогах загального користування. Так, наприклад, у штаті Каліфорнія, США, компанія Google отримала ліцензію на тестування автомобілів в межах штату. Таким чином, мешканці цього штату вже сьогодні можуть стати свідками майбутнього.

Однак роботизовані автомобілі все ще не є ідеальними та мають чимало вад, від яких необхідно позбавитись. Однією з таких проблем є проїзд по ушкодженим дорогам. Звісно, ця проблема не є першочерговою зараз, на перших стадіях побудови таких автомобілів. Тим не менш, проблема пошкоджених доріг є доволі розповсюдженою у цілому ряді країн, і, очевидно, стане перешкодою для роботизованих автомобілів.

Метою цієї дипломної роботи є створення алгоритму побудови оптимальної траєкторії руху чотириколісного автомобіля по ушкодженій дорозі. Такий алгоритм може знайти застосування у сфері роботизованих автомобілів та сприяти розвитку цієї галузі.

Формулювання проблеми та її актуальність

За відомою інформацією про стан певної частини дороги у напрямку руху автомобіля, побудувати оптимальну траєкторію його руху. Врахувати габаритні характеристики автомобіля, геометрію його руху та особливості ушкоджень.

Актуальність такої проблеми пояснюється відсутністю механізму врахування перешкод подібного характеру прототипами роботизованих автомобілів компанії Google [1]. Також відсутні алгоритми, які б розв’язували цю задачу з врахуванням специфіки руху чотириколісного автомобіля по звичайній автомобільній дорозі.

Огляд існуючих розв’язків

Задача побудови оптимальної траєкторії є відомою та є однією з основних проблем теорії керування. Її частинний випадок, а саме побудова оптимальних траєкторій руху транспортних засобів, також є відомим. Тим не менш, існуючі алгоритми розв’язку такої задачі мають власну специфіку. Не враховуються деякі ключові особливості, які притаманні саме задачі руху автомобіля по дорозі.

Так, наприклад, у роботі М. Гаселіча та Н. Ганджийського «про швидку побудову траєкторії руху в неструктурованому середовищі» [2] розглядається проблема руху позашляхового автомобіля по пересічній місцевості. Ключовою особливістю побудованого алгоритму є класифікація місцевості за допомогою алгоритмів розпізнавання. Побудова траєкторії здійснюється виключно виходячи з проведеної класифікації «дорога» / «не дорога». Така класифікація не знаходить застосування у задачі побудови траєкторії на дорозі з асфальтовим покриттям, оскільки досить часто провести візуальну класифікацію неможливо.

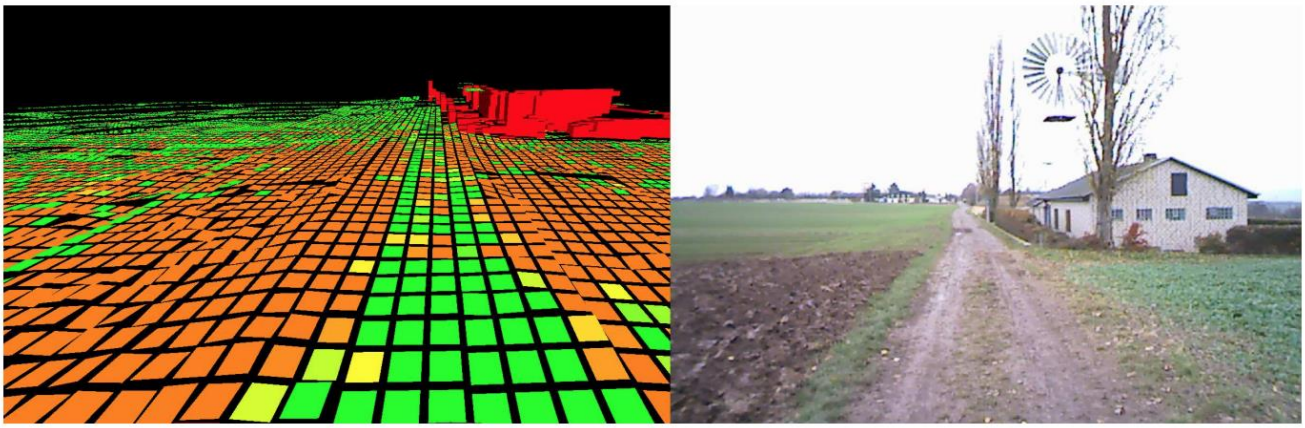


Рис. 1: Ілюстрація класифікатора Н. Гаселіча та Н. Ганджийського

Ще одним прикладом є робота Х. Янга, що розглядає «побудову оптимальної траєкторії руху наземного та повітряного транспорту у динамічному середовищі» [3]. Особливість цієї роботи притаманна цілому ряду інших робіт, та відокремлює їх від проблеми проїзду по пошкодженій дорозі. Характерною особливістю є те, що геометрія руху транспортного засобу врахована поверхнево. Розглядається рух матеріальної точки з певним фіксованим околом. Такий підхід не може бути застосовано по відношенню до автомобіля.

Також, перешкоди розглядаються як певні геометричні фігури з чіткими границями. Інакше кажучи, в кожній точці простору є лише інформація про те, чи є в цій точці перешкода, чи її немає. Такий підхід буде дуже грубим при застосуванні до пошкоджень на автомобільній дорозі, які значно відрізняються за своїм характером.

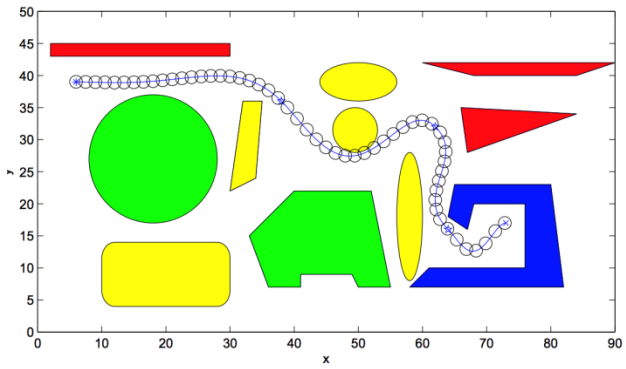


Рис. 2: Побудова оптимальної траєкторії у роботі Х. Янга

Ще одним близьким класом задач є задачі побудови оптимальної траєкторії руху роботів-розвідників. Так, наприклад, у роботі Т. Говарда розглядається побудова таких алгоритмів для роботи марсоходу. На прикладі цієї роботи зручно ілюструвати ще одну суттєву особливість роботизованих автомобілів: побудова оптимальної траєкторії повинна враховувати низьку маневрову здатність автомобіля в умовах руху на дорозі. На відміну від роботів-розвідників, в роботизованих автомобілів зазвичай немає змоги зупинитись на дорозі, або здійснити надто крутий поворот. Також, алгоритми роботів-розвідників переважно розраховані саме на пересічну місцевість, а в критерій оптимальності покладені такі фактори, як витрати пального та спроможність безперешкодного підйому по схилу або безпечного спуску. Такі критерії не мають місця для звичайного чотириколісного автомобіля в умовах руху по автомобільній дорозі, де витратами пального та перепадом висоти на достатньо короткому шматку дороги можна знехтувати.

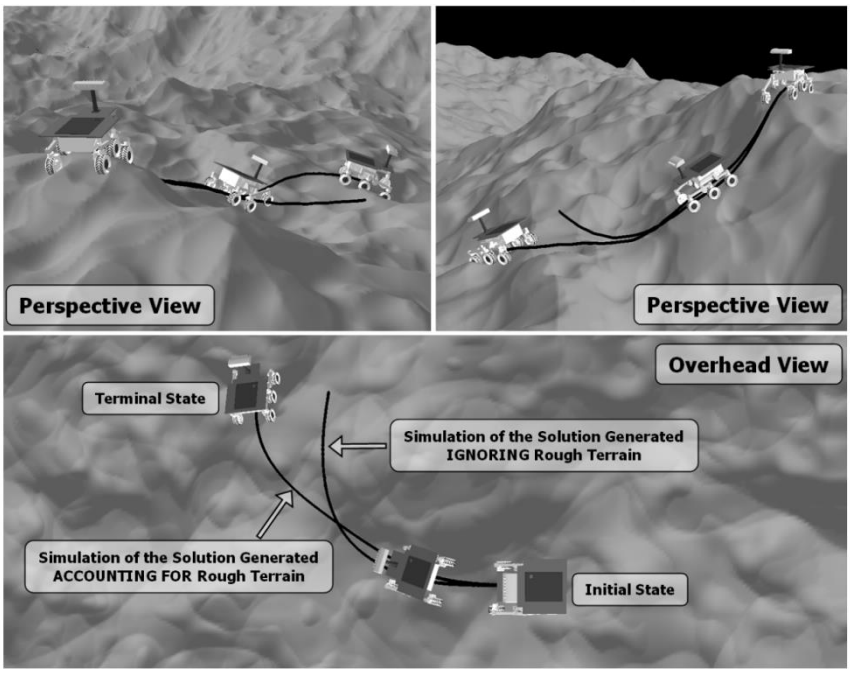


Рис. 3: Ілюстрація особливостей руху роботів-розвідників

Постановка задачі

У даній дипломній роботі розглядається сукупність пов’язаних між собою задач:

- Дослідити особливості руху чотириколісного транспортного засобу з системою керування Акермана по дорогам з пошкодженнями різного характеру;
- Сформулювати критерій оптимальності траєкторії руху чотириколісного автомобіля по пошкодженій дорозі;
- Створити алгоритм, який будуватиме оптимальну траєкторію руху на основі інформації про габаритні характеристики автомобіля та інформації про дорожнє покриття.

Формалізація вхідних та вихідних даних

Вхідні дані алгоритму:

- W, H - довжина та ширина шматка дорожньої полоси, в межах якого повинен рухатись автомобіль, та інформація про покриття якої є відомою;
- $Q(x, y), x \in [0; W], y \in [0; H]$ - інформація про дорожнє покриття («функція якості»). Є дискретною функцією своїх змінних, значення якої відповідають відхиленню висоти дороги в заданій точці від нормальної висоти дорожнього покриття (глибина ями). Вважається доступною як результат роботи скануючого пристрою роботизованого автомобіля у вигляді монохромного зображення.

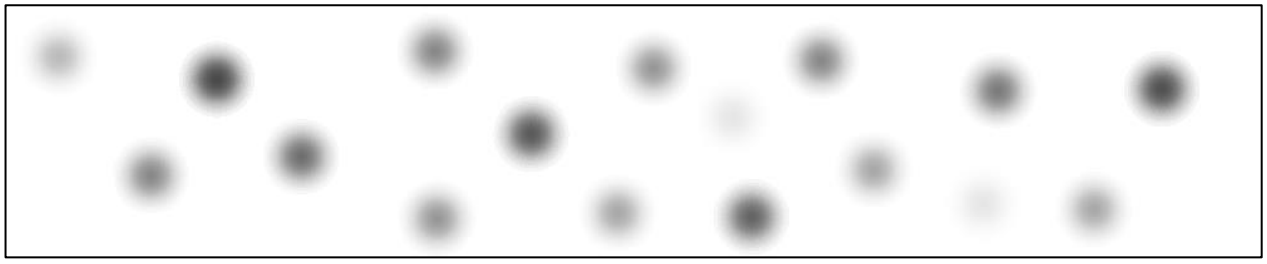


Рис. 4: Приклад функції якості, поданої у вигляді монохромного зображення з значеннями в інтервалі $[0; 255]$

- $w_{car}, l_{car}, wheel_{car}$ - габаритні характеристики автомобіля: колія коліс (відстань між центрами коліс однієї вісі автомобіля), колісна база (відстань між передньою та задньою віссю), ширина контакту автомобільної покришки й дороги;
- $y_L \in [0; H], y_R \in [0; H]$ - крайові умови положення автомобіля на кінцях полоси дороги, на якій розглядається задача. Визначають положення середини задньої вісі автомобіля при $x_L = 0$ та $x_R = W$ відповідно;
- $y'_L \in [0; H], y'_R \in [0; H]$ - крайові умови орієнтації автомобіля на кінцях полоси дороги, на якій розглядається задача. Визначають орієнтацію руху автомобіля (тангенс кута напрямку руху автомобіля по відношенню до напрямку дороги) при $x_L = 0$ та $x_R = W$ відповідно.

Результатом (вихідними даними) алгоритму є оптимальна траєкторія руху середини задньої вісі автомобіля, що задається у вигляді функції $f(x), x \in [0; W]$. За умови реалізації системи керування автомобіля за правилами Акермана, таке задання траєкторії однозначно визначає рух автомобіля, а саме положення всіх чотирьох коліс та їх орієнтацію.

Функцію $f(x)$ будемо називати траєкторією автомобіля.

Побудова розв'язку

Аналіз особливостей руху автомобіля

Для розв'язання поставленої задачі використовується спрощена модель автомобіля. З точки зору побудови оптимальної траєкторії проїзду по пошкодженій дорозі не має сенсу розглядати такі фізичні характеристики, як вага чи форма автомобіля. Значення має лише слід автомобіля, тобто множина точок контакту автомобіля з поверхнею дороги; фізичними параметрами, що не впливають на слід автомобіля, або впливають зовсім незначною мірою, було знехтувано.

У використаній моделі руху в основу покладені основні габаритні характеристики автомобіля (ширина покришок коліс, та взаємне розташування коліс), а також залежність траєкторії руху від керування автомобілем (система керування Акермана).

Габаритні характеристики автомобіля зображено на Рисунку 5.

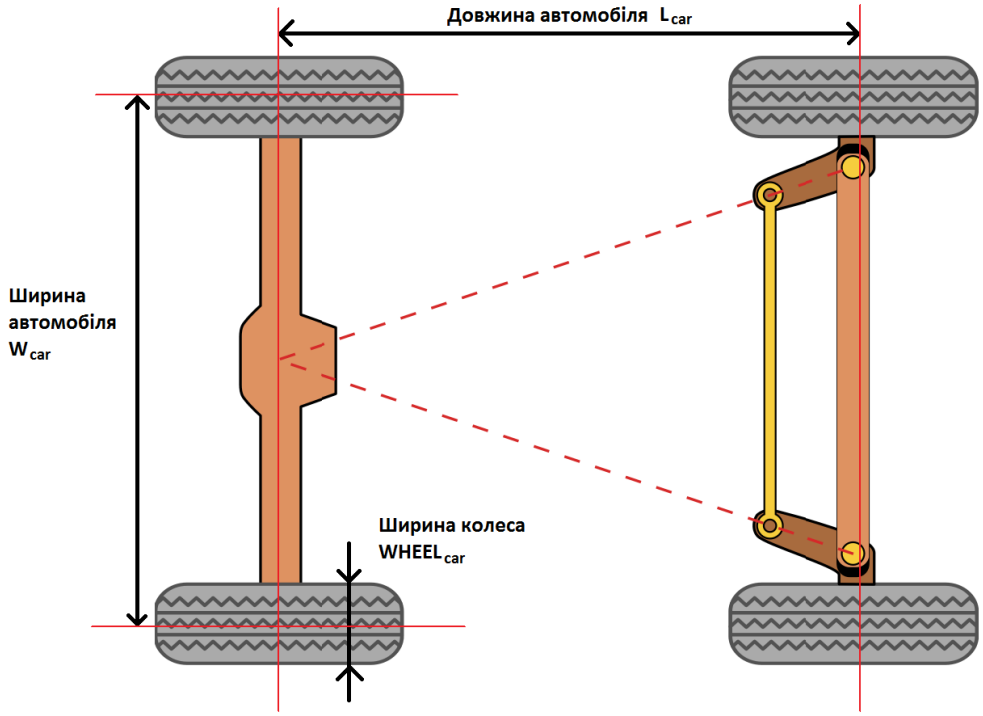


Рис. 5: Габаритні характеристики чотириколісного автомобіля

Ідеальна модель керування Анкермана [5] полягає в тому, що в режимі стаціонарного повороту траєкторії всіх коліс є концентричними колами. Інакше кажучи, вісі всіх чотирьох коліс перетинаються в одній точці, що буде центром обертання.

В такому випадку центральна (головна) вісь автомобіля, що проходить через центри осей автомобіля, буде дотичною до кола з центром в центрі обертання, що проходить через центр задньої вісі.

Така властивість системи керування Акермана дозволяє прийти до наступного твердження: в режимі стаціонарного повороту, траєкторія руху точки центру задньої вісі однозначно задає положення та орієнтацію всіх чотирьох коліс. Дійсно, побудувавши дотичну до траєкторії в заданій точці отримаємо центральну вісь автомобіля, а відтак і положення всіх чотирьох коліс. В той же час, траєкторія є дугою з відомим центром, що дозволяє однозначно визначити орієнтацію всіх коліс, знаючи їх положення на дорозі.

Ілюстрація системи керування Акермана зображена на Рисунку 6.

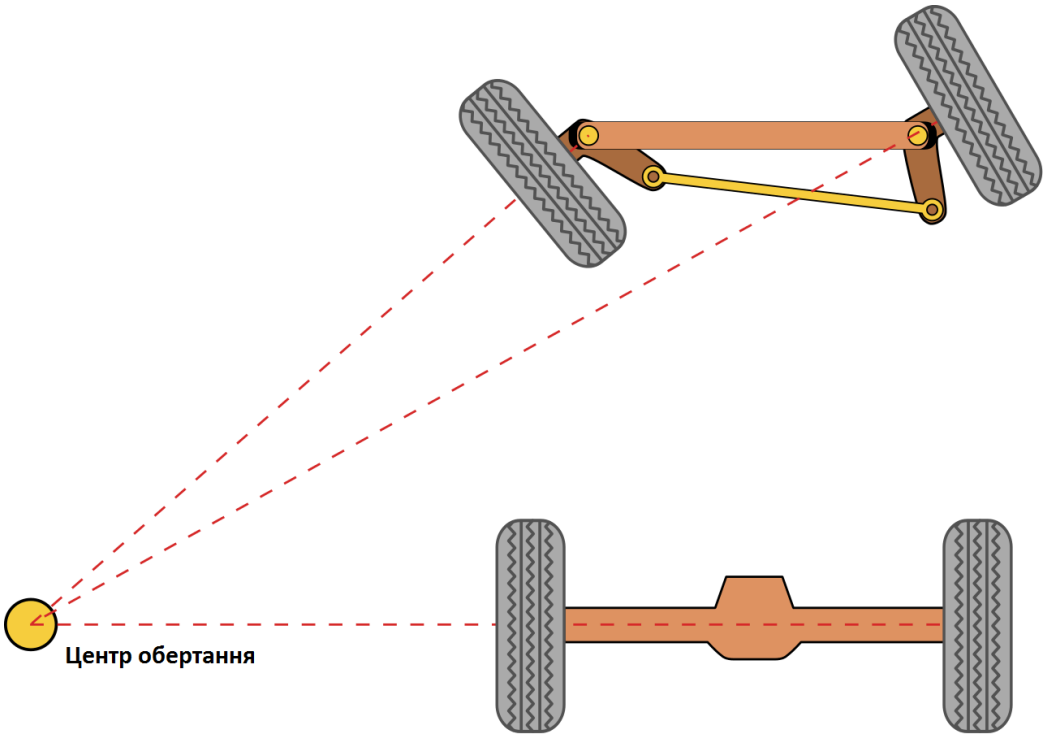


Рис. 6: Схема система керування Акермана

Аналіз особливостей проїзду пошкоджень. Критерій якості траєкторії

При побудові критерію якості траєкторії, варто виходити з природних тверджень та спостережень:

- Часткове потрапляння колеса в дорожню яму наносить менше шкоди, аніж повне потрапляння;
- Залежність шкоди, заподіяної ямою, від її глибини перевершує лінійну. В даній роботі шкода від проїзду ями пропорційна квадрату її глибини;
- За відсутності пошкоджень дороги, пряма траєкторія має перевагу над всіма іншими;

Виходячи з цих міркувань, вводиться функціонал загального штрафу на множині допустимих траєкторій:

$$P[f] = \alpha_1 Q_1[f] + \alpha_2 Q_2[f] + \alpha_3 Q_3[f],$$
$$f(0) = y_L,\ f(W) = y_R,$$
$$f'(0) = y'_L,\ f(W) = y'_R$$

в якому:

- $Q_1[f]$ - функціонал штрафу за проїзд перешкод:

$$Q_1[f] = \begin{cases} \sum_{i=1}^{4} \iint_{D_i} Q^2(x,y)\,dxdy, \text{ якщо } D_i \subseteq [0;W] \times [0;H] \\ \infty, \text{ інакше} \end{cases}$$

де D_i - область, що відповідає сліду відповідного колеса. Цей функціонал відповідає за акумуляцію штрафу за проїзд перешкод при русі по заданій траєкторії. При цьому виїзд за рамки полоси вважається неприпустимим, і в такому випадку штраф стає нескінченно великим;

- $Q_2[f]$- функціонал штрафу за довжину траєкторії:

$$Q_2[f] = \int_0^W \sqrt{1 + \left(f'(x)\right)^2}\, dx$$

- $Q_3[f]$ - функціонал штрафу за маневреність вздовж траєкторії:

$$Q_3[f] = \sum_{i=1}^{N} \frac{1}{R_i}$$

де R_i - радіус дуги кола, що є наближенням до відповідного шматка траєкторії руху при її розбитті на N шматків, що рівні за довжиною;

- $\alpha_1, \alpha_2, \alpha_3$ - вагові коефіцієнти, що встановлюються експериментальним шляхом та залежать від габаритних характеристик автомобіля.

Оптимальна траєкторія визначається як така, що мінімізує заданий функціонал загального штрафу на множині допустимих траєкторій. Таким чином, задача побудови оптимальної траєкторії зводиться до задачі мінімізації функціонала.

Продовження функції якості дороги

За постановкою задачі, функція якості дороги $Q(x, y)$ задана дискретно. Фактично, оскільки вона надається на вхід у вигляді зображення, маємо функцію, що визначена лише у точках з цілочисельними координатами, та набуває в кожній точці одного з 256 значень.

З метою точнішого підрахунку функціоналу штрафу, продовжимо функцію якості дороги на довільні точки з області $[0; W] \times [0; H]$ таким чином, щоб функція була неперервною. При цьому обчислення такого продовження повинно здійснюватись швидко, оскільки підрахування функції якості $Q(x, y)$ є одною з базових операцій, а тому здійснюється дуже часто.

Розіб'ємо задану область на трикутники, вершини яких матимуть цілочисельні координати. Трійку вершин будемо знаходити наступним чином:

Якщо $\{x\}+\{y\}\le 1$, то $A=([x],[y]), B=([x]+1,[y]), C=([x],[y]+1)$;

Якщо $\{x\}+\{y\}>1$, то $A=([x]+1,[y]+1), B=([x]+1,[y]), C=([x],[y]+1)$.

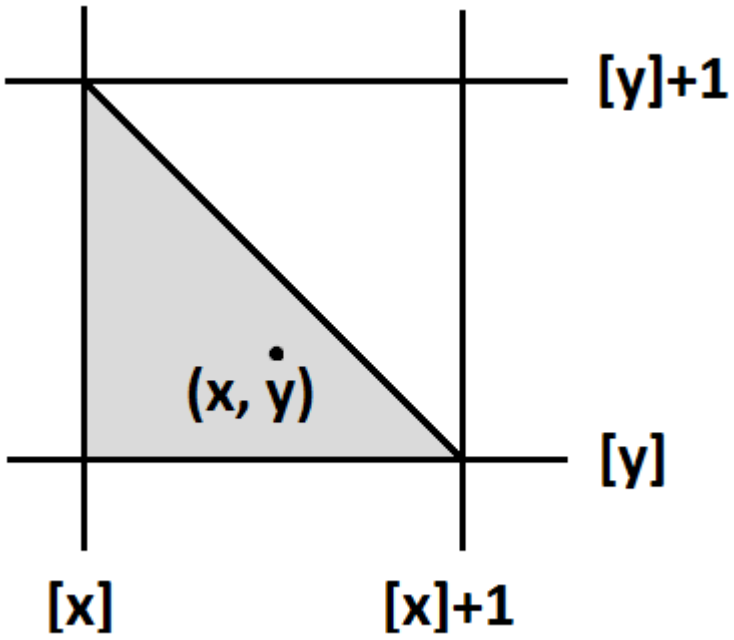


Рис. 7: Ілюстрація розбиття області на трикутники

Обчисливши три вершини трикутника, додамо у якості третього виміру значення функції якості у кожній з точок. В результаті отримаємо три тривимірні точки:

$$x_1 = x_A, y_1 = y_A, z_1 = Q(x_A, y_A)$$
$$x_2 = x_B, y_2 = y_B, z_2 = Q(x_B, y_B)$$
$$x_3 = x_C, y_3 = y_C, z_3 = Q(x_C, y_C)$$

Проведемо через них площину L , що існує та єдина. Знайдемо точку, що належить цій площині, а перші дві координати якої збігаються з заданою точкою:

$$(x, y, z) \in L, x = x, y = y$$

Тоді значення z третьої координати і будемо вважати значенням розширеної функції якості $Q(x, y)$ в точці (x, y).

Загальна ідея пошуку оптимальної траєкторії

Задача задана у досить загальному вигляді: функція якості дороги $Q(x, y)$ може набувати довільного вигляду. У зв'язку з цим і розв'язання задачі вимагає застосування загальних методів, що не прив'язані до вигляду функції та можуть бути застосовані до широкого класу вхідних параметрів – наприклад, методів градієнтного спуску.

Розіб'ємо задану область вздовж осі абсцис на N рівних шматків, де N - параметр розв'язку. Припустимо, що на границях цих шматків траєкторія автомобіля набуває відомих значень:

$$f(x_i) = y_i, i = \overline{0, N}$$

При цьому також виконуються крайові умови:

$$f(0) = y_0 = y_L,\ f(W) = y_N = y_R$$
$$f'(0+0) = y'_L,\ f'(W-0) = y'_R$$

Будемо наближати шукану функцію кубічними сплайнами, що задовольняють описані умови на границях та всередині області. Для кожного набору $y_i\left(i = \overline{0, N}\right), y'_0, y'_N$ кубічний сплайн існує та єдиний [6].

Таке наближення є достатнім з точки зору управління транспортним засобом, та в той же час досить простим з точки зору обчислень. Окрім того, в такому разі досить швидко й просто обчислюються значення похідних траєкторії, які є компонентами функціонала штрафу.

Таким чином, задачу пошуку оптимальної траєкторії зведено до задачі підбору вдалих значень траєкторії всередині області, оскільки крайові умови задачі задані за умовою. Інакше кажучи, необхідно знайти вектор:

$$\vec{y}^* = (y_1, y_2, ..., y_{N-1}),$$
$$y_i \in [0; H], i = \overline{1, N-1}$$

такий, що кубічний сплайн

$$f(x) = f^*(y_0, y_1, ..., y_N, y'_0, y'_N, x) = f(\vec{y}^*, x)$$

буде надавати найменшого значення функціоналу штрафу:

$$f(x) = \arg\min_g P[g].$$

Вектор y^* обчислюється з наперед заданою точністю ε (по кожній з координат) методом покоординатного спуску з сталим коефіцієнтом зменшення кроку α [7].

Покажемо, що цей метод зійдеться. Для фіксованого покоординатного кроку алгоритму λ існує не більш, ніж $(N-1)\times\left(\left[\frac{H}{\lambda}\right]+1\right)$ векторів, для яких буде знайдено значення функціоналу загального штрафу. Отже, ітерація алгоритму для сталого координатного кроку λ рано чи пізно закінчиться. В той же час, кількість ітерацій алгоритму також є скінченною, оскільки при $\alpha < 1$ значення координатного кроку λ рано чи пізно стане меншим за параметр ε, після чого завершиться пошук оптимального вектора $\vec{y}^*$.

Коефіцієнт зменшення координатного кроку α, параметр точності алгоритму ε, початковий координатний крок λ_0, а також кількість відрізків розбиття N є параметрами алгоритму. Вони мають бути задані перед початком обчислень, та залежать від потужності процесора, на якому проводяться обчислення, оскільки суттєво впливають на швидкість знаходження наближеного розв’язку.

Алгоритм пошуку оптимальної траєкторії

Враховуючи всі описані вище етапи знаходження оптимальної траєкторії руху, алгоритм пошуку оптимальної траєкторії руху чотириколісного роботизованого автомобіля набуває наступного вигляду (подано у вигляді псевдокоду):

Зчитати вхідні дані та **задати** параметри алгоритму:

- дискретна функція якості $Q(x,y)$;
- габаритні характеристики автомобіля $w_{car}, l_{car}, wheel_{car}$;
- значення машинного нуля θ та машинної нескінченності ∞;
- значення вагових коефіцієнтів функціоналу загального штрафу $\alpha_1, \alpha_2, \alpha_3$;
- параметри методу покоординатного спуску: початковий координатний крок λ_0, коефіцієнт зменшення координатного кроку α, точність пошуку оптимальних координат ε;
- кількість спроб запуску методу покоординатного спуску T;
- кількість ключових точок сплайну N;
- розмір розбиття траєкторії на рівні за довжиною шматки M;
- розмір розбиття траєкторії для чисельного інтегрування K;

Виконати для $t : 1 \to T$:

Згенерувати вектор $\vec{y} = (y_1, y_2, ..., y_{N-1})$ випадковим чином;

Задати $\lambda = \lambda_0$;

Повторювати поки $\lambda > \varepsilon$:

Виконати для $i : 1 \to N-1$:

Задати $\vec{y}_{new} = (y_1, y_2, ..., y_{i-1}, y_i + \lambda, y_{i+1}, ..., y_{N-1})$;

Задати $f(x) = f(\vec{y}_{new}, x)$; // кубічний сплайн

Розбити $f(x)$ на M рівновеликих шматків за допомогою чисельного інтегрування, обчислити загальну довжину Q_2;

Задати $Q_1 = Q_2 = Q_3 = 0$;

Виконати для $j : 1 \to M$:

Наблизити шматок траєкторії $f_j(x)$ відрізком або дугою кола;

Якщо дуга кола, **задати** $Q_3 = Q_3 + \frac{1}{R_j}$;

Обчислити положення та орієнтацію кожного з коліс при русі середини задньої вісі вздовж шматка траєкторії $f_j(x)$;

Виконати для кожного з коліс:

Наблизити слід колеса D_{ij} прямокутником або кільцевим сегментом;

Задати $Q_1 = Q_1 + \iint\limits_{D_{ij}} Q^2(x, y)\,dxdy$ за допомогою чисельного інтегрування;

Задати $P\left[f\left(\vec{y}_{new}, x\right)\right] = \sum\limits_{k=1}^{3} \alpha_k Q_k$;

Якщо $P\left[f\left(\vec{y}_{new}, x\right)\right] < P\left[f\left(\vec{y}, x\right)\right]$:

Задати $\vec{y} = \vec{y}_{new}$;

Перейти до $i \to i+1$;

Задати $\vec{y}_{new} = (y_1, y_2, ..., y_{i-1}, y_i - \lambda, y_{i+1}, ..., y_{N-1})$;

... // повторити аналогічні обчислення P

Якщо $P\left[f\left(\vec{y}_{new}, x\right)\right] < P\left[f\left(\vec{y}, x\right)\right]$:

Задати $\vec{y} = \vec{y}_{new}$ **;**

Перейти до $i \to i+1$**;**

Якщо $P\left[f\left(\vec{y}, x\right)\right] < P\left[f\left(\vec{y}^*, x\right)\right]$ **або** $\vec{y}^*$ не визначений:

Задати $\vec{y}^* = \vec{y}$ **;**

Якщо вектор $\vec{y}$ на даній ітерації не зазнав змін:

Задати $\lambda = \alpha\lambda$ **;**

Повернути $f^*(x) = f\left(\vec{y}^*, x\right)$.

Результати роботи алгоритму

Тестування роботи алгоритму проводилось з наступними параметрами:

- Розмір шматка дороги (довжина 15 метрів, ширина 3 метри):

$$W = 1500, H = 300$$

- Габаритні характеристики автомобіля:

$$w_{car} = 171$$
$$l_{car} = 271$$
$$wheel_{car} = 31.5$$

Характеристики відповідають характеристикам автомобіля Bugatti Veyron Super Sport, заданим в сантиметрах;

- Параметри методу покоординатного спуску: $\lambda_0 = 33.0, \alpha = 0.5, \varepsilon = 1.0$
- Кількість спроб запуску методу покоординатного спуску: $T = 5$
- Параметри машинного нуля та нескінченності: $\theta = 10^{-5}, \infty = 10^{9}$
- Параметри розбиття: $N = 10, M = 100, K = 1000$
- Вагові коефіцієнти функціоналу загального штрафу:

$$\alpha_1 = 100.0, \alpha_2 = 0.5, \alpha_3 = 100.0$$

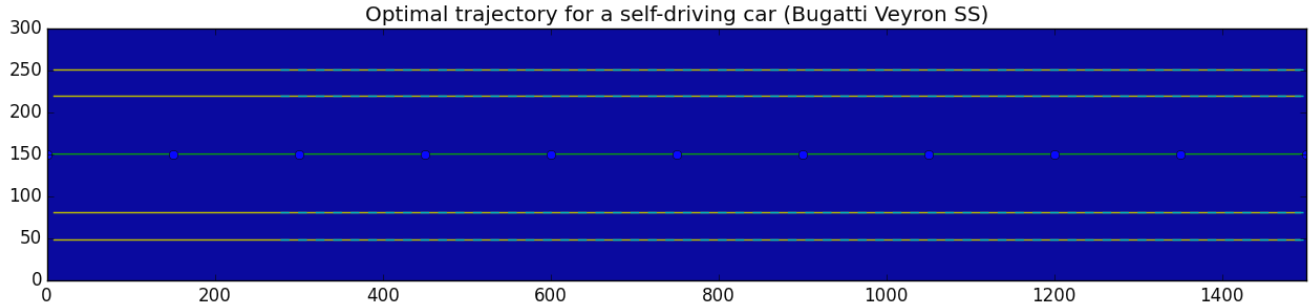
- Крайові умови: $y_L = \frac{H}{2}, y_R = \frac{H}{2}, y_L = 0, y_R = 0$

Тестування здійснювалось для різних варіантів доріг: сильно та слабко ушкоджених, з можливістю або неможливістю повного уникнення контакту з пошкодженням. Також проведено тестування для неушкодженої дороги.

Дорога без перешкод

Вхідні дані:

Результат:

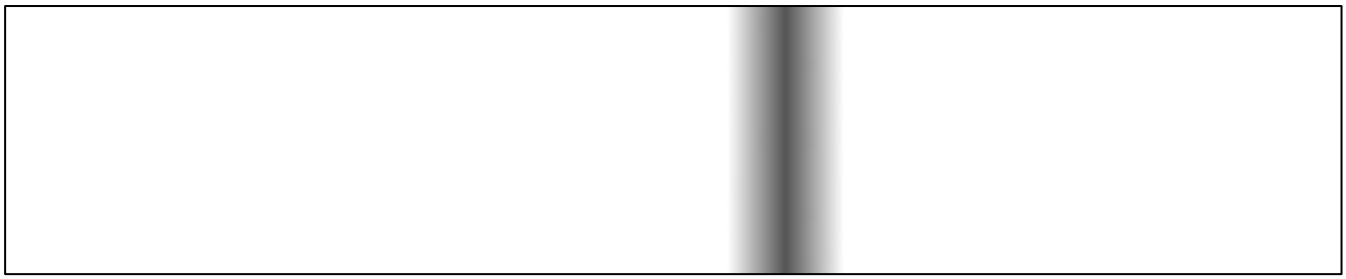


Час роботи: 40.61 сек.

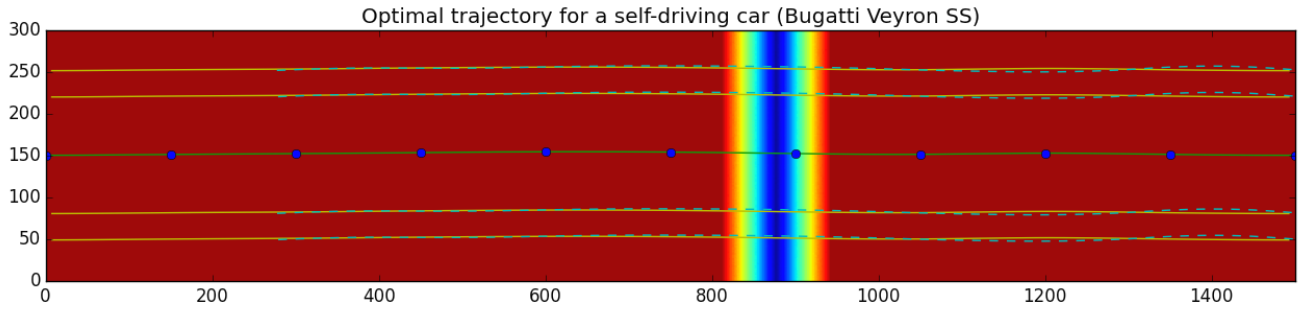
Аналіз адекватності: результатом роботи алгоритму на дорозі без перешкод є пряма траєкторія руху, що повністю узгоджується з очікуваним результатом.

Дорога з поперечною канавою

Вхідні дані:



Результат:

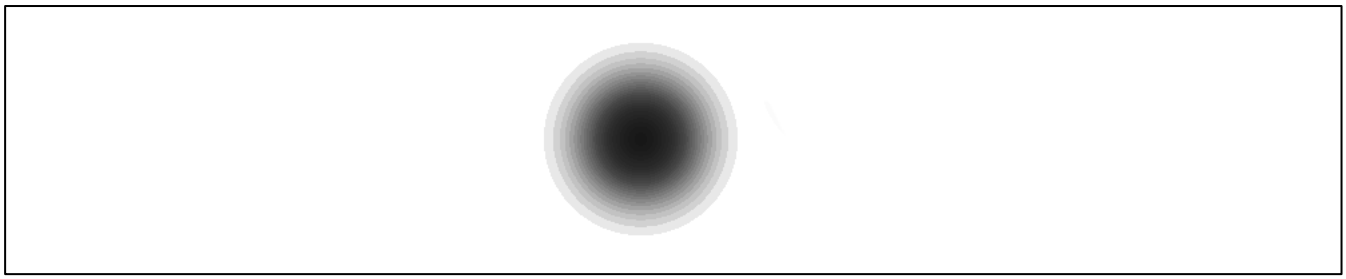


Час роботи: 32.12 сек.

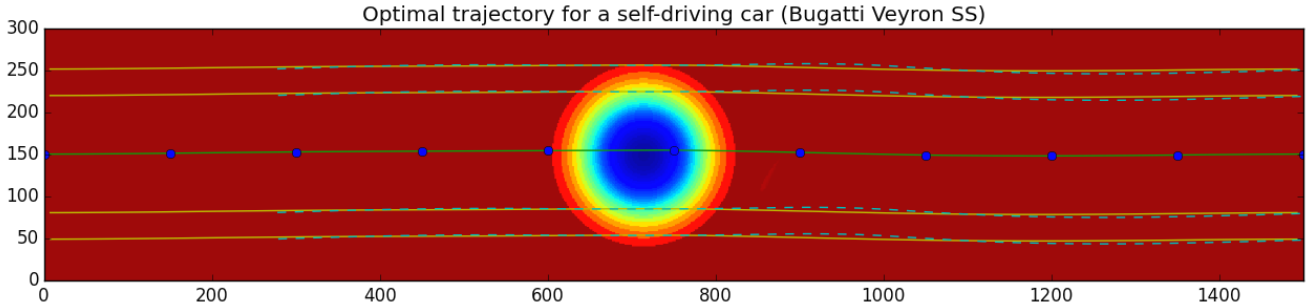
Аналіз адекватності: для дороги з ідеально поперечною перешкодою, результатом роботи алгоритму є майже пряма дорога. Присутнє незначне відхилення від прямолінійної траєкторії руху. Відхилення пояснюється похибкою обчислення чисельного інтеграла штрафу Q_1, а саме через приближення області інтегрування кільцевими сегментами. Загалом відхилення за величиною можна прирівняти до незначних коливань при русі автомобіля, тому така похибка не буде відчутною для пасажира.

Дорога з однією ямою, що лежить близько до центру

Вхідні дані:



Результат:



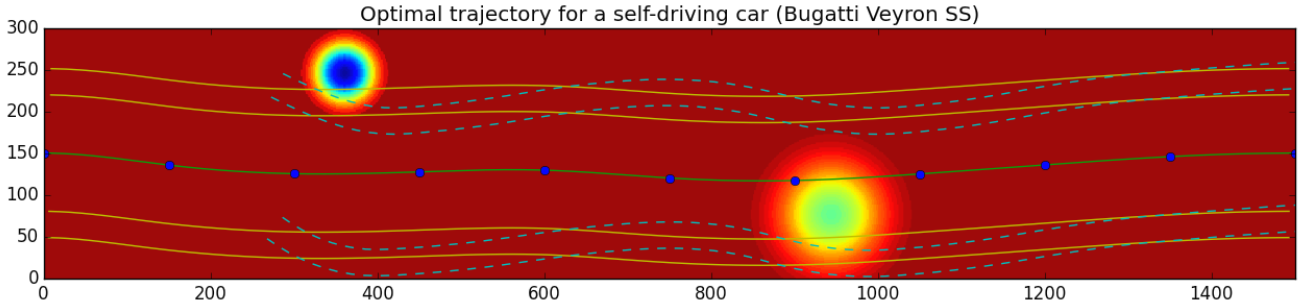
Час роботи: 32.78 сек.

Аналіз адекватності: результатом роботи алгоритму на дорозі з однією ямою є гладка траєкторія, що виглядає задовільно. Незначні відхилення від прямої траєкторії є виправданими та очікуваними, і пояснюються тим, що центр ями лежить поза центром дороги.

Дорога з двома ямами, віддаленими від центру

Вхідні дані:

Результат:



Час роботи: 44.01 сек.

Аналіз адекватності: результатом роботи алгоритму на дорозі з двома ямами є траєкторія, яка максимально оминає обидві ями, але в той же час частково зачіпає обидві. Алгоритмом врахована неможливість повного об’їзду даних ям.

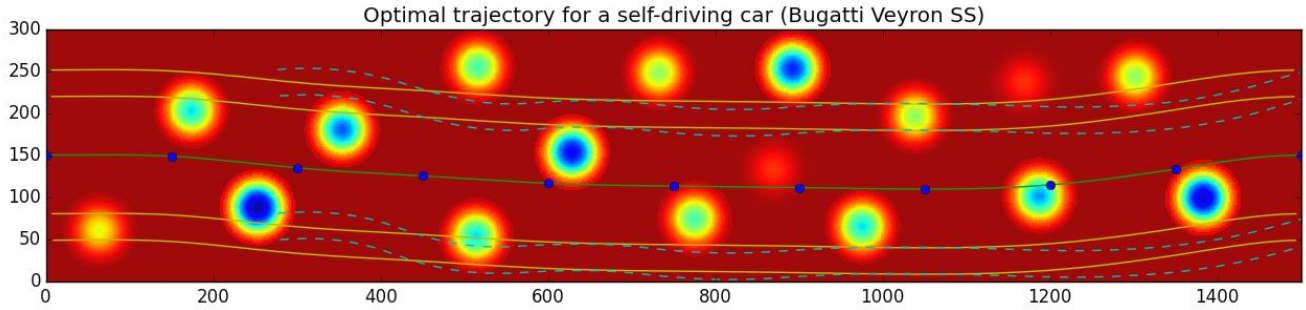
На шматку дороги між ямами спостерігається надлишкове маневрування у вигляді повороту вліво. Потенційною причиною такої особливості траєкторії є вирівнювання напрямку руху автомобіля для уникнення виїзду за праву границю дорожньої полоси.

На результаті також видно, що при об’їзді дальньої ями автомобіль передньою віссю досяг свого максимально допустимого правого положення, що позитивно характеризує знайдений розв’язок в даних умовах.

Дорога з значними пошкодженнями (1)

Вхідні дані:

Результат:



Час роботи: 42.84 сек.

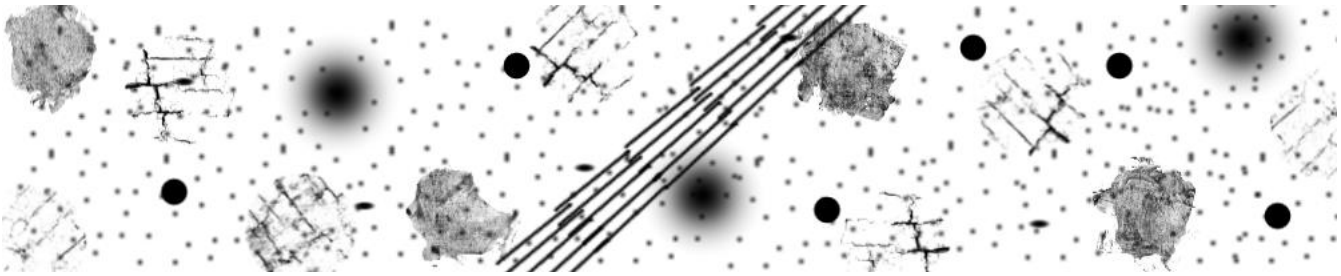
Аналіз адекватності: результатом роботи алгоритму на дорозі із значною кількістю ям є траєкторія, що проходить біля правої границі дорожньої полоси та досить ефективно оминає найглибші ями, які несуть в собі найбільшу загрозу.

Спостерігається миттєва реакція на ями, що знаходяться в безпосередній близькості до автомобіля на момент початку руху. Також слід відзначити стабільність руху вздовж правого краю до того граничного моменту, коли необхідно повернути автомобіль на центр полоси задля задоволення правих граничних умов.

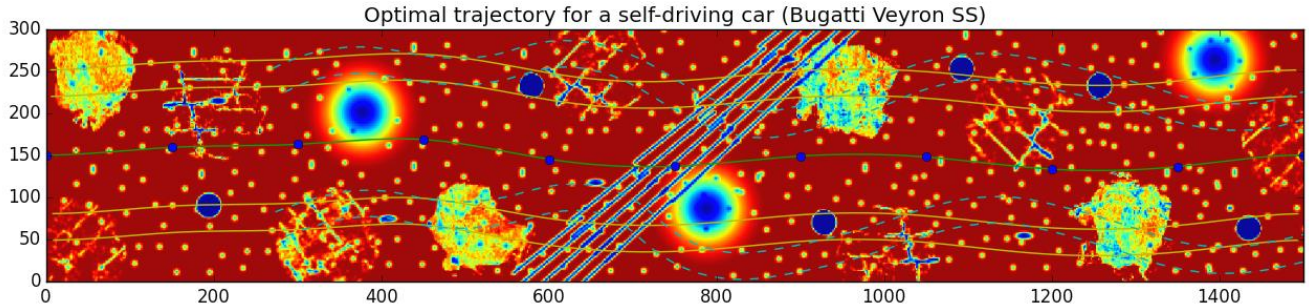
Приклад є гарною ілюстрацією важливості практичного застосування алгоритму, оскільки прокладає ефективну траєкторію для ситуації, яка є складною навіть для досвідченого водія.

Дорога з значними пошкодженнями (2)

Вхідні дані:



Результат:



Час роботи: 29.60 сек.

Аналіз адекватності: результатом роботи алгоритму на дорозі із значною кількістю ям є траєкторія, що оминає найбільші ями. Очевидною є неможливість абсолютного уникнення ям автомобілем в такому випадку.

Приклад ілюструє, що алгоритм може бути слабко чутливим до таких ям, що мають малий розмір, проте є глибокими. Причиною такої низької чутливості є недостатньо велике значення параметрів розбиття M , через що при чисельному інтегруванні для обчислення штрафу Q_1 малі ями залишаються поза ключовими точками інтегрування, таким чином не впливаючи на результат. Потенційним вирішенням проблеми є збільшення значення параметру M , що негативно вплине на швидкодію алгоритму. Проте в реальній ситуації ями такого характеру практично не зустрічаються.

В той же час необхідно відзначити, що високий рівень пошкоджень дороги не впливає на швидкодію алгоритму.

Висновок

Результатом дипломної роботи є формулювання критерію оптимальності траєкторії руху чотириколісного роботизованого автомобіля по ушкодженій дорозі, постановка оптимізаційної задачі на основі створеного критерію, а також побудова алгоритму, який знаходить наближений розв’язок поставленої оптимізаційної задачі.

Реалізація алгоритму мовою Python 2.7.9 будує траєкторії для штучно створених даних, що відображають можливі ушкодження різного характеру реальних автомобільних доріг. Побудовані траєкторії значною мірою відповідають природним очікуванням щодо оптимальної траєкторії руху, що є особливо важливим у тих ситуаціях, коли оптимальна траєкторія руху не є очевидною для водія.

Швидкодія створеної програми не відповідає вимогам щодо генерації траєкторії в режимі живого часу. На побудову траєкторії для створених тестових даних в середньому витрачається біля 30 секунд. Втім, варто пам’ятати, що швидкодія може бути значною мірою покращена у випадку реалізації алгоритму іншими мовами – наприклад, мовою C++. Вибір мови Python в даному випадку пояснюється в першу чергу простотою використання графічних бібліотек для демонстрації результату, що жодним чином не є важливим при реальному практичному застосуванні в роботизованих автомобілях.

Важливою особливістю даного алгоритму є його незалежність від ступеня ушкодження дороги. Так, при тестуванні дороги з надзвичайно великою кількістю ям час роботи був меншим, ніж у випадку дороги без пошкоджень.

Загалом, побудований розв’язок готовий до використання та може бути застосований в практичних задачах побудови оптимальної траєкторії руху роботизованого автомобіля.

Список використаних джерел

1) “Hidden Obstacles for Google’s Self-Driving Cars”, L. Gomes

(http://www.technologyreview.com/news/530276/hidden-obstacles-for-googles-self-driving-cars/)

2) “Spline Templates for Fast Path Planning in Unstructured Environments”, M. Haselich, N. Handzhiyski

(http://www.uni-koblenz.de/~agas/Public/Haeselich2011STF.pdf)

3) “Real-time Trajectory Planning for Ground and Aerial Vehicles in A Dynamic Environment”, J. Yang

(http://etd.fcla.edu/CF/CFE0002031/Jian_Yang_200804_PhD.pdf)

4) “Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots”, T. Howard

(https://www.ri.cmu.edu/pub_files/pub4/howard_thomas_2007_1/howard_thomas_2007_1.pdf)

5) “Ackermann steering geometry”, Wikipedia

(http://en.wikipedia.org/wiki/Ackermann_steering_geometry)

6) “Кубический сплайн”, Википедия

(https://ru.wikipedia.org/wiki/Кубический_сплайн)

7) “Метод покоординатного спуска”, Википедия

(https://goo.gl/fU2lrF)

8) «Численные методы», Н. Н. Калиткин. М., Наука, 1978

Додатки

Код програми

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-

""" Optimal trajectory builder (Alex Bashuk's thesis work)."""

import copy
import bisect
import numpy as np
import matplotlib.pyplot as plt
import random
import PIL
import os
import sys
import math
import profilehooks

__author__ = "Alex Bashuk"
__copyright__ = "Copyright (c) 2015 Alex Bashuk"
__credits__ = "Alex Bashuk"
__license__ = "MIT"
__version__ = "1.0.0"
__maintainer__ = "Alex Bashuk"
__email__ = "alex@bashuk.tk"
__status__ = "Development"

# Constants
DEBUG = True

INFINITY = 10 ** 9
EPSILON = 10 ** -5

DEFAULT_INTEGRATION_PIECES_PER_MILE = 10
DEFAULT_MILES_PER_POINT = 10
DEFAULT_POINTS = 10
DEFAULT_ATTEMPTS = 5
DEFAULT_SAVE_TO_FILE_SCALE = 1.0
DEFAULT_FILENAME = 'samples/3.png'

JUMP_STEP_INITIAL_VALUE = 33.0
JUMPING_THRESHOLD = 1.0
JUMP_STEP_MULTIPLIER = 0.5

ROAD_BOTTOM_VALUE = -100.0
Q1_SCALE = 100.0
Q2_SCALE = 0.5
Q3_SCALE = 100.0

def log(message, force = False):
    """
    Helper function. Prints message to STDOUT in debug mode, or if forced.
```

```
    """
    message = str(message)
    if DEBUG or force:
        sys.stdout.write(message)
        sys.stdout.flush()

class SplineBuilder:
    """
    Spline builder class.
    This class helps to build a cubic spline based on finite set of points,
    function values in these points, and boundary conditions of first type
    given on left and right sides of the interval.
    """
    def __init__(self):
        # Amount of points
        self._n = 0
        # Coordinates and function values
        self._x = []
        self._y = []
        # Cubic spline parameters
        self._a = []
        self._b = []
        self._c = []
        self._d = []
        # Spline miles
        self.miles = 0
        self.mile_length = 0
        self.mile = []

    def _tdma_solve(self, a, b, c, d):
        """
        Tri Diagonal Matrix Algorithm(a.k.a Thomas algorithm) solver

        TDMA solver, a b c d can be NumPy array type or Python list type.
        refer to http://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm

        Source: https://gist.github.com/ofan666/1875903
        """
        # preappending missing parts
        a = [0] + a
        c = c + [0]

        nf = len(a)     # number of equations
        ac, bc, cc, dc = map(np.array, (a, b, c, d))     # copy the array
        for it in xrange(1, nf):
            mc = ac[it]/bc[it-1]
            bc[it] = bc[it] - mc*cc[it-1]
            dc[it] = dc[it] - mc*dc[it-1]

        xc = ac
        xc[-1] = dc[-1]/bc[-1]

        for il in xrange(nf-2, -1, -1):
            xc[il] = (dc[il]-cc[il]*xc[il+1])/bc[il]
```

```
        del bc, cc, dc  # delete variables from memory

        return list(xc)

    def build(self, x, y, der_left = 0, der_right = 0):
        """
        Builds a spline.
        Source: http://matlab.exponenta.ru/spline/book1/12.php
        """
        if len(x) != len(y):
            raise ValueError("x and y have to be of the same size.")
        if len(x) < 2:
            raise ValueError("x must be at least of size 2.")

        x = map(float, x)
        y = map(float, y)

        self._n = len(x) - 1
        self._x = copy.deepcopy(x)
        self._y = copy.deepcopy(y)

        h = []
        for k in range(self._n):
            h.append(x[k + 1] - x[k])

        if self._n > 1:
            # here must be (n - 1) equations for b
            alpha = []
            for k in range(1, self._n - 1):
                alpha.append(1.0 / h[k])
            beta = []
            for k in range(0, self._n - 1):
                beta.append(2.0 * (1.0 / h[k] + 1.0 / h[k + 1]))
            gamma = []
            for k in range(0, self._n - 2):
                gamma.append(1.0 / h[k + 1])
            delta = []
            for k in range(0, self._n - 1):
                delta.append(3.0 * (
                    (y[k + 2] - y[k + 1]) / (x[k + 2] - x[k + 1]) / h[k + 1] +
                    (y[k + 1] - y[k]) / (x[k + 1] - x[k]) / h[k]
                    ))
            # boundary conditions of 1st type
            delta[0] -= der_left * 1.0 / h[0]
            delta[self._n - 2] -= der_right * 1.0 / h[self._n - 1]
            self._b = self._tdma_solve(alpha, beta, gamma, delta)
        else:
            self._b = []

        self._a = copy.deepcopy(self._y)
        self._b = [der_left] + self._b + [der_right]
        self._c = []
        for k in range(0, self._n):
            self._c.append(
                (3.0 * (y[k + 1] - y[k]) / (x[k + 1] - x[k]) -
```

```
                self._b[k + 1] - 2.0 * self._b[k]) /
            h[k]
            )
        self._d = []
        for k in range(0, self._n):
            self._d.append(
                (self._b[k] + self._b[k + 1] -
                    2.0 * (y[k + 1] - y[k]) / (x[k + 1] - x[k])) /
                (h[k] * h[k])
                )
        self._a = self._a[:-1]
        self._b = self._b[:-1]

    def f(self, x):
        """
        Calculates the value of a spline approximation in a given point.
        """
        if self._n == 0:
            raise Exception("Spline not built yet.")
        if x < self._x[0] or self._x[-1] < x:
            raise ValueError("Given point is out of interval.")

        x = float(x)

        if x == self._x[-1]:
            index = self._n - 1
        else:
            index = bisect.bisect_right(self._x, x) - 1

        x0 = self._x[index]
        a = self._a[index]
        b = self._b[index]
        c = self._c[index]
        d = self._d[index]
        value = float(a + b * (x - x0) + c * (x - x0) ** 2 + d * (x - x0) ** 3)

        return value

    def der_f(self, x):
        """
        Calculates the derivative of spline approximation in a given point.
        """
        if self._n == 0:
            raise Exception("Spline not built yet.")
        if x < self._x[0] or self._x[-1] < x:
            raise ValueError("Given point is out of interval.")

        x = float(x)

        if x == self._x[-1]:
            index = self._n - 1
        else:
            index = bisect.bisect_right(self._x, x) - 1

        x0 = self._x[index]
```

```
        a = self._a[index]
        b = self._b[index]
        c = self._c[index]
        d = self._d[index]
        der = b + 2.0 * c * (x - x0) + 3.0 * d * (x - x0) ** 2

        return der

    def split_by_length(self, miles, integration_pieces_per_mile =
        DEFAULT_INTEGRATION_PIECES_PER_MILE):
        """
        Splits calculated spline into m pieces, equal by lenght.
        """
        M = miles
        N = miles * integration_pieces_per_mile

        x = map(float, np.linspace(self._x[0], self._x[-1], 2 * N + 1))
        y = map(lambda x: math.sqrt(1.0 + self.der_f(x) ** 2), x)

        l = [0.0]
        for i in range(N):
            intgr = (x[2 * i + 2] - x[2 * i]) / 6.0 * \
                (y[2 * i] + 4.0 * y[2 * i + 1] + y[2 * i + 2])
            l.append(intgr)
        for i in range(1, len(l)):
            l[i] = l[i] + l[i - 1]

        mile_length = l[-1] / M
        self.mile = [0.0]
        wanted = mile_length
        for i in range(N):
            if l[i] <= wanted and wanted < l[i + 1]:
                # alpha == 1.0 means left, alpha == 0.0 means right
                alpha = (l[i + 1] - wanted) / (l[i + 1] - l[i])
                self.mile.append(
                    alpha * x[2 * i] + (1.0 - alpha) * x[2 * (i + 1)])
                wanted += mile_length
        if len(self.mile) == M:
            self.mile.append(x[-1])
        self.miles = M
        self.mile_length = mile_length

        if len(self.mile) != M + 1:
            raise Exception("Ooops... I think we didn't manage to split it.")

    def show(self):
        """
        Show spline.
        """
        x = np.linspace(self._x[0], self._x[-1], self._x[-1] - self._x[0] + 1)
        y = [self.f(xi) for xi in x]

        dot_x = self._x
        dot_y = [self.f(xi) for xi in dot_x]
```

```
        plt.plot(x, y, 'g', dot_x, dot_y, 'ro')
        plt.show()

class QualityFunctionBuilder:
    """
    Quality function builder class.
    This class helps to load the terrain quality function Q(x, y) from file.
    Also it provides methods for calculating the values of points between
    the pixel centers.
    """
    def __init__(self):
        # image weight and height
        self._imw = 0
        self._imh = 0
        # image data
        self._im = []
        # terrain weight and height
        self.w = 0.0
        self.h = 0.0

    def _build_plane_by_3_points(self, p1, p2, p3):
        """
        Builds a plane that goes through three given points in 3-D space.
        Plane is defined by equation: Ax + By + Cz + D = 0.
        Coefficients A, B, C, D are returned as a result.
        """
        p1 = np.array(p1)
        p2 = np.array(p2)
        p3 = np.array(p3)

        v1 = p1 - p3
        v2 = p2 - p3
        norm = np.cross(v1, v2)

        A, B, C = map(float, norm)
        D = float(- np.dot(norm, p3))

        return A, B, C, D

    def load_from_image(self, filename):
        """
        Load quality function from image file.
        Pixels with high values (white) correspond to bigger height on the
        terrain.
        This method sets terrain size to be equal to the image size.
        To change the terrain size, use set_custom_terrain_size() method.
        """
        im = PIL.Image.open(filename)
        if im.mode != "L":
            im = im.convert("L")

        pix = im.load()
        self._imw, self._imh = im.size
        self.w, self.h = float(self._imw - 1), float(self._imh - 1)
        if self.w < 1 or self.h < 1:
```

```
            self.__init__()
            raise Exception("Image size should be at least 2x2.")
        self._im = [
            [float(pix[x, y]) / 255.0 for y in xrange(self._imh - 1, -1, -1)]
            for x in xrange(self._imw)
        ]

    def set_custom_terrain_size(self, size):
        """
        Sets custom terrain size (terrain sizes equal image size by default).
        """
        if self.w == 0:
            raise Exception("You should load an image before setting size.")

        w, h = size
        if w <= 0 or h <= 0:
            raise ValueError("Terrain size must be positive.")
        self.w = float(w)
        self.h = float(h)

    def Q(self, x, y):
        """
        Quality function, linearly interpolated from the image pixel values.
        """
        if self.w == 0:
            raise Exception("Quality funcion not loaded yet.")

        # Going beyond the boundaries of the road line is not acceptable.
        if y < 0 or y > self.h:
            # I thought of the exception here at first, but then I realised
            # that splines can go beyond the road, and when calculating
            # the quality for that spline - we don't want it to raise any
            # exceptions, we just want it to make an enormously big jump
            return - INFINITY
            # raise ValueError("Given point is out of the terrain.")
        # Since we have no info about the quality of the road there, we just
        # have to assume that those values are the same as on the boundaries.
        if x < 0:
            x = 0
        if x > self.w:
            x = self.w

        # Scaling
        x = 1.0 * x * (self._imw - 1) / self.w
        y = 1.0 * y  * (self._imh - 1) / self.h

        # Handling boundaries
        eps = EPSILON
        if abs(x) < eps and x <= 0.0:
            x += eps
        if abs(x - (self._imw - 1)) < eps:
            x -= eps
        if abs(y) < eps and y <= 0.0:
            y += eps
        if abs(y - (self._imh - 1)) < eps:
```

```
            y -= eps

        # From now on, coordinates here must be strictly inside the image:
        # 0.0 < x < _imw, 0.0 < y < _imh
        return self._im[int(x)][int(y)] # optimization
        p1 = (int(x) + 1, int(y))
        p2 = (int(x), int(y) + 1)
        if x - int(x) + y - int(y) <= 1.0:
            p3 = (int(x), int(y))
        else:
            p3 = (int(x) + 1, int(y) + 1)

        p1 = (p1[0], p1[1], self._im[p1[0]][p1[1]])
        p2 = (p2[0], p2[1], self._im[p2[0]][p2[1]])
        p3 = (p3[0], p3[1], self._im[p3[0]][p3[1]])
        a, b, c, d = self._build_plane_by_3_points(p1, p2, p3)

        res = - (a * x + b * y + d) / c
        return res

class CarBuilder:
    """
    Simple class for containing car parameters.
    """
    def __init__(self, width = 171, length = 271, wheel = 31.5,
        title = 'Bugatti Veyron SS'):
        """
        Default parameters (width = 171, length = 271, wheel = 315) are
        the actual parameters of Bugatti Veyron SS, given in cm.
        """
        self.width = width
        self.length = length
        self.wheel = wheel
        self.title = title

class VehicleTrajectoryBuilder:
    """
    Vehicle Trajectory Builder.
    This class builds the trajectory for a 4-wheels vehicle. Vehicle's movement
    is defined by Ackermann steering geometry.
    """
    def __init__(self, qfb, car, fl = None, fr = None, dfl = None, dfr = None):
        # Helper classes
        self._qfb = qfb
        self._car = car
        self._sb = SplineBuilder()
        # Terrain parameters
        self._w = qfb.w
        self._h = qfb.h
        # Boundaries
        self._fl = fl if fl is not None else qfb.h * 0.5
        self._fr = fr if fr is not None else qfb.h * 0.5
        self._dfl = dfl if dfl is not None else 0.0
        self._dfr = dfr if dfr is not None else 0.0
        # Quality along trajectory
```

```
        self._qat = - INFINITY
        self._qs = ()
        # Trained flag
        self._trained = False
        # wheels traces
        # 0, 1 - rear left inner, outer
        # 2, 3 - rear right inner, outer
        # 4, 5 - front left inner, outer
        # 6, 7 - front right inner, outer
        self._wtrace = [[] for i in xrange(8)]

    def _generate_straight_trajectory(self, points, miles_per_point =
        DEFAULT_MILES_PER_POINT, rebuild_spline = True):
        """
        Configures the spline so that it corresponds to simplest straight
        movement.
        """
        x = map(float, np.linspace(0.0, self._w, points + 1))
        y = [self._h / 2.0 for i in xrange(points + 1)]
        y[0] = self._fl
        y[points] = self._fr

        if rebuild_spline:
            self._sb.build(x, y, self._dfl, self._dfr)
            miles = points * miles_per_point
            self._qat, self._qs = \
                self._quality_along_trajectory(miles)

        return x, y

    def _generate_random_trajectory(self, points, miles_per_point =
        DEFAULT_MILES_PER_POINT, rebuild_spline = True):
        """
        Configures the spline so that it defines some random trajectory.
        """
        margin = (self._car.width + self._car.wheel) / 2.0
        x = map(float, np.linspace(0.0, self._w, points + 1))
        y = [random.uniform(0.0 + margin, self._h - margin)
            for i in xrange(points + 1)]
        y[0] = self._fl
        y[points] = self._fr

        if rebuild_spline:
            self._sb.build(x, y, self._dfl, self._dfr)
            miles = points * miles_per_point
            self._qat, self._qs = \
                self._quality_along_trajectory(miles)

        return x, y

    def _try_alternative_trajectory(self, new_x, new_y, miles, force = False):
        """
        Tries alternative trajectory.
        If the quality along the new trajectory is higher, or if forced,
        the spline is rebuilt along the new trajectory.
```

```
        Otherwise, the spline is not changed.
        Return True if the spline was changed, False otherwise.
        """
        cur_x = copy.deepcopy(self._sb._x)
        cur_y = copy.deepcopy(self._sb._y)

        self._sb.build(new_x, new_y, self._dfl, self._dfr)
        new_qat, new_qs = self._quality_along_trajectory(miles)
        if new_qat > self._qat or force:
            self._qat = new_qat
            self._qs = new_qs
            return True
        else:
            self._sb.build(cur_x, cur_y, self._dfl, self._dfr)
            return False

    def _quality_along_trajectory(self, miles, save_trace = False):
        """
        Calculates the quality of current trajectory (defined by spline).
        """
        self._sb.split_by_length(miles)

        # wheels traces
        self._wtrace = [[] for i in xrange(8)]

        # Q1 is integral over quality function.
        # This corresponds to quality of the road.
        # The more - the better.
        # Implemented with the method of right rectangles
        Q1 = 0.0
        # Q2 corresponds to the length of the trajectory.
        # The less - the better.
        Q2 = 0.0
        # Q3 corresponds to the curvature of the trajectory.
        # The less - the better.
        Q3 = 0.0
        for i in xrange(1, len(self._sb.mile)):
            # Start and end points of the arc
            x1 = self._sb.mile[i - 1]
            y1 = self._sb.f(x1)
            x2 = self._sb.mile[i]
            y2 = self._sb.f(x2)

            # Derivatives on the ends, corresponding angles
            d1 = self._sb.der_f(x1)
            a1 = math.atan(d1)
            d2 = self._sb.der_f(x2)
            a2 = math.atan(d2)

            # Additional vectors for easy computation of coordinates of
            # all 4 wheels
            start = np.array([x1, y1])
            finish = np.array([x2, y2])
            front_move = (self._car.length) * np.array([np.cos(a1), np.sin(a1)])
            wheel_move = (self._car.width * 0.5) * np.array(
```

```
        [np.cos(a1 + np.pi * 0.5), np.sin(a1 + np.pi * 0.5)])
tire_move = (self._car.wheel * 0.5) * np.array(
        [np.cos(a1 + np.pi * 0.5), np.sin(a1 + np.pi * 0.5)])

# Checking if we are not going beyond the road lane
low_limit = 0.0 + (self._car.wheel + self._car.width) * 0.5
high_limit = self._qfb.h - (self._car.wheel + self._car.width) * 0.5
if start[1] < low_limit or start[1] > high_limit or \
    (start + front_move)[1] < low_limit or \
    (start + front_move)[1] > high_limit:
    Q1 -= INFINITY

# rbv = road bottom value
# When Q(x, y) == 1.0, we want to take this point, so it counts
# as 1.0.
# But when it's bottom (i.e., Q == 0.0), this is very bad for
# the car, so we want to count it not as 0.0, but a much smaller
# value. For example, - 100.0.
rbv = ROAD_BOTTOM_VALUE

if abs(a1 - a2) < EPSILON:
    # Case 1: line
    # All wheels will cover the same area in this case.
    # The only difference is where we take the average quality -
    # this point will be individual for each wheel.
    area = self._car.wheel * self._sb.mile_length
    mile_move = (self._sb.mile_length * 0.5) * \
        np.array([np.cos(a1), np.sin(a1)])

    # For a line, there is no curvature.
    Q3 += 0.0

    # Rear wheels
    rear = start + mile_move # rear suspention center
    w = rear + wheel_move # left wheel
    qfbQ = self._qfb.Q(*w)
    value = qfbQ ** 2 - rbv * qfbQ + rbv if qfbQ > 0.0 else qfbQ
    Q1 += float(area * value)
    if save_trace:
        self._wtrace[0].append(w - tire_move)
        self._wtrace[1].append(w + tire_move)

    w = rear - wheel_move # right wheel
    qfbQ = self._qfb.Q(*w)
    value = qfbQ ** 2 - rbv * qfbQ + rbv if qfbQ > 0.0 else qfbQ
    Q1 += float(area * value)
    if save_trace:
        self._wtrace[2].append(w + tire_move)
        self._wtrace[3].append(w - tire_move)

    # Front wheels
    front = start + mile_move + front_move # front suspention center
    w = front + wheel_move # left wheel
    qfbQ = self._qfb.Q(*w)
    value = qfbQ ** 2 - rbv * qfbQ + rbv if qfbQ > 0.0 else qfbQ
```

```
            Q1 += float(area * value)
            if save_trace:
                self._wtrace[4].append(w - tire_move)
                self._wtrace[5].append(w + tire_move)

            w = front - wheel_move # right wheel
            qfbQ = self._qfb.Q(*w)
            value = qfbQ ** 2 - rbv * qfbQ + rbv if qfbQ > 0.0 else qfbQ
            Q1 += float(area * value)
            if save_trace:
                self._wtrace[6].append(w + tire_move)
                self._wtrace[7].append(w - tire_move)

            # All 4 wheels move straight forward
            Q2 += 4.0 * self._sb.mile_length
        else:
            # Case 2: arc
            # Wheels are moving according to the Ackermann steering.

            # Finding parameters of the arc
            # Radial lines
            a1p = a1 + np.pi * 0.5
            l1a = - np.sin(a1p) # = y1 - (y1 + np.sin(a1p))
            l1b = np.cos(a1p) # = (x1 + np.cos(a1p)) - x1
            l1c = - (l1a * x1 + l1b * y1)

            a2p = a2 + np.pi * 0.5
            l2a = - np.sin(a2p) # = y2 - (y2 + np.sin(a2p))
            l2b = np.cos(a2p) # = (x2 + np.cos(a2p)) - x2
            l2c = - (l2a * x2 + l2b * y2)

            # Center is the intersection of the lines
            cx = float((l1b * l2c - l2b * l1c) / (l1a * l2b - l2a * l1b))
            cy = float((l1c * l2a - l2c * l1a) / (l1a * l2b - l2a * l1b))
            co = np.array([cx, cy])

            # Arc radius
            # Not always there is an arc that perfectly fits the given
            # boundary conditions (x, y, a). That's why this circular
            # arc is just an approximation.
            R1 = np.linalg.norm(start - co)
            R2 = np.linalg.norm(finish - co)
            cr = float(R1 + R2) / 2.0

            # Arc radial size
            v1 = start - co # radial vector
            v2 = finish - co # radial vector
            v1u = v1 / np.linalg.norm(v1) # unit vector
            v2u = v2 / np.linalg.norm(v2) # unit vector
            ca = np.arccos(np.dot(v1u, v2u)) # circular arc angle
            if np.isnan(ca):
                if (v1u == v2u).all():
                    return 0.0
                else:
                    return np.pi
```

```
ca = float(ca)

# Orienting the angle
# http://e-maxx.ru/algo/oriented_area
s = (cx - x1) * (y2 - y1) - (cy - y1) * (x2 - x1)
if s > 0:
    # counter-clockwise order - angle must be negative
    ca = - ca

# For an arc, the smaller is radius and the bigger is
# the angle of the arc – the bigger is the curvature.
Q3 += 1.0 / cr

# Rotation matrix (rotates the half of the arc angle)
rot_a = ca * 0.5
rot_m = np.array([[np.cos(rot_a), - np.sin(rot_a)],
    [np.sin(rot_a), np.cos(rot_a)]])

# clock = - sgn(ca)
# this is needed to compute wheel traces
if ca > 0.0:
    clock = 1.0
else:
    clock = - 1.0

# Rear left wheel
start_w = start + wheel_move
rel_start_w = start_w - co
rel_mid_w = np.dot(rot_m, rel_start_w)
wheel_r = np.linalg.norm(rel_mid_w)
inner_r = wheel_r - self._car.wheel * 0.5
outer_r = wheel_r + self._car.wheel * 0.5
Q2 += wheel_r * abs(ca)
area = (outer_r ** 2 - inner_r ** 2) * abs(ca) / 2.0
mid_w = rel_mid_w + co
qfbQ = self._qfb.Q(*mid_w)
value = qfbQ ** 2 - rbv * qfbQ + rbv if qfbQ > 0.0 else qfbQ
Q1 += float(area * value)
if save_trace:
    tire_move = rel_mid_w / np.linalg.norm(rel_mid_w)
    tire_move *= self._car.wheel * 0.5
    self._wtrace[0].append(mid_w + clock * tire_move)
    self._wtrace[1].append(mid_w - clock * tire_move)

# Rear right wheel
start_w = start - wheel_move
rel_start_w = start_w - co
rel_mid_w = np.dot(rot_m, rel_start_w)
wheel_r = np.linalg.norm(rel_mid_w)
inner_r = wheel_r - self._car.wheel * 0.5
outer_r = wheel_r + self._car.wheel * 0.5
Q2 += wheel_r * abs(ca)
area = (outer_r ** 2 - inner_r ** 2) * abs(ca) / 2.0
mid_w = rel_mid_w + co
qfbQ = self._qfb.Q(*mid_w)
```

```
            value = qfbQ ** 2 - rbv * qfbQ + rbv if qfbQ > 0.0 else qfbQ
            Q1 += float(area * value)
            if save_trace:
                tire_move = rel_mid_w / np.linalg.norm(rel_mid_w)
                tire_move *= self._car.wheel * 0.5
                self._wtrace[2].append(mid_w - clock * tire_move)
                self._wtrace[3].append(mid_w + clock * tire_move)

            # Front left wheel
            start_w = start + front_move + wheel_move
            rel_start_w = start_w - co
            rel_mid_w = np.dot(rot_m, rel_start_w)
            wheel_r = np.linalg.norm(rel_mid_w)
            inner_r = wheel_r - self._car.wheel * 0.5
            outer_r = wheel_r + self._car.wheel * 0.5
            Q2 += wheel_r * abs(ca)
            area = (outer_r ** 2 - inner_r ** 2) * abs(ca) / 2.0
            mid_w = rel_mid_w + co
            qfbQ = self._qfb.Q(*mid_w)
            value = qfbQ ** 2 - rbv * qfbQ + rbv if qfbQ > 0.0 else qfbQ
            Q1 += float(area * value)
            if save_trace:
                tire_move = rel_mid_w / np.linalg.norm(rel_mid_w)
                tire_move *= self._car.wheel * 0.5
                self._wtrace[4].append(mid_w + clock * tire_move)
                self._wtrace[5].append(mid_w - clock * tire_move)

            # Front right wheel
            start_w = start + front_move - wheel_move
            rel_start_w = start_w - co
            rel_mid_w = np.dot(rot_m, rel_start_w)
            wheel_r = np.linalg.norm(rel_mid_w)
            inner_r = wheel_r - self._car.wheel * 0.5
            outer_r = wheel_r + self._car.wheel * 0.5
            Q2 += wheel_r * abs(ca)
            area = (outer_r ** 2 - inner_r ** 2) * abs(ca) / 2.0
            mid_w = rel_mid_w + co
            qfbQ = self._qfb.Q(*mid_w)
            value = qfbQ ** 2 - rbv * qfbQ + rbv if qfbQ > 0.0 else qfbQ
            Q1 += float(area * value)
            if save_trace:
                tire_move = rel_mid_w / np.linalg.norm(rel_mid_w)
                tire_move *= self._car.wheel * 0.5
                self._wtrace[6].append(mid_w - clock * tire_move)
                self._wtrace[7].append(mid_w + clock * tire_move)

    # Total quality along the trajectory.
    # The more - the better.
    # Q1: [0.0 .. 4 * wheel * w]
    # Q2: [qfb.w .. sinusoidal_length]
    # Q3: [0.0 .. 0.1-ish]
    Q1 = Q1 / (4.0 * self._qfb.w * self._car.wheel) * Q1_SCALE
    Q2 = - Q2 * Q2_SCALE
    Q3 = - Q3 * Q3_SCALE
```

```
    res = Q1 + Q2 + Q3

    return res, (Q1, Q2, Q3)

@profilehooks.profile
def train_trajectory(self, attempts = DEFAULT_ATTEMPTS, points =
    DEFAULT_POINTS, miles_per_point = DEFAULT_MILES_PER_POINT):
    """
    Trains the spline so that it has the best quality.
    """
    miles = points * miles_per_point

    # Choosing from given number of random trajectories
    self._generate_straight_trajectory(points, miles_per_point)
    x = copy.deepcopy(self._sb._x)
    best_qat = self._qat
    best_y = copy.deepcopy(self._sb._y)
    log("\rAttempt #0: quality = {} {}\n".format(self._qat, self._qs))

    for attempt in xrange(attempts):
        # These two parameters define the process of optimization
        # Also, they define the number of iterations
        jump_step = JUMP_STEP_INITIAL_VALUE
        threshold = JUMPING_THRESHOLD

        self._generate_random_trajectory(points, miles_per_point)
        while jump_step > threshold:
            changed = True
            while changed:
                changed = False
                for point in xrange(1, points):
                    y = copy.deepcopy(self._sb._y)
                    init_point_y = y[point]

                    # trying to jump up
                    y[point] = init_point_y + jump_step
                    if self._try_alternative_trajectory(x, y, miles):
                        changed = True
                    # trying to jump down
                    y[point] = init_point_y - jump_step
                    if self._try_alternative_trajectory(x, y, miles):
                        changed = True

                    # Clear current line, then write again
                    sys.stdout.write('\x1b[2K\r')
                    log("Attempt #{}: jump = {}, quality = {} {}".format(
                        attempt + 1, jump_step, self._qat, self._qs))
                    # self.show()
            jump_step *= JUMP_STEP_MULTIPLIER
        sys.stdout.write("\n")

        if self._qat > best_qat:
            best_qat = self._qat
            best_y = copy.deepcopy(self._sb._y)
```

```
        self._sb.build(x, best_y, self._dfl, self._dfr)
        self._qat, self._qs = self._quality_along_trajectory(
            miles, save_trace = True)
        self._trained = True
        print 'Best: {}'.format(best_qat)

    def f(self, x):
        """
        Returns the values of trajectory function.
        """
        return self._sb.f(x)

    def Q(self, x, y):
        """
        Returns the values of quality function.
        """
        return self._qfb.Q(x, y)

    def show(self):
        """
        Show quality function and current trajectory.
        """
        Q_x = np.linspace(0, self._qfb.w, self._qfb.w)
        Q_y = np.linspace(0, self._qfb.h, self._qfb.h)
        img = [[self.Q(x, y) for x in Q_x] for y in Q_y]
        plt.imshow(img)

        f_x = np.linspace(0, self._qfb.w, self._qfb.w)
        f_y = [self.f(xi) for xi in f_x]
        plt.plot(f_x, f_y, 'g', label='Trajectory (spline)')

        plt.plot(self._sb._x, self._sb._y, 'bo', label='Key points')

        # wheels traces
        # rear wheels
        for trace in self._wtrace[:4]:
            wx = [w[0] for w in trace]
            wy = [w[1] for w in trace]
            plt.plot(wx, wy, 'y-')
        # front wheels
        for trace in self._wtrace[4:]:
            wx = [w[0] for w in trace]
            wy = [w[1] for w in trace]
            plt.plot(wx, wy, 'c--')

        plt.title('Optimal trajectory for a self-driving car ({})'.format(
            self._car.title))
        plt.axis([Q_x[0], Q_x[-1], Q_y[0], Q_y[-1]])
        plt.show()

    def save_to_file(self, filename, scale = DEFAULT_SAVE_TO_FILE_SCALE):
        """
        Saves an image of the terrain combined with the trajectory curves.
        """
        w, h = int(self._w * scale), int(self._h * scale)
```

```
        im = PIL.Image.new("RGB", (w, h))
        pix = im.load()

        # Drawing the quality of the terrain
        for i in xrange(w):
            for j in xrange(h):
                px, py = i * 1.0 / scale, j * 1.0 / scale
                R, G, B = [int(self._qfb.Q(px, py) * 255.0)] * 3
                pix[i, j] = (R, G, B)

        # Drawing the trajectory
        alpha = 0.7
        for i in xrange(w):
            x = i * 1.0 / scale
            y = self._sb.f(x)

            j0 = int(y * scale)
            low = max(0, j0)
            high = min(h, j0 + 1)
            for j in range(low, high):
                R, G, B = pix[i, j]
                R = int(alpha * 0   + (1.0 - alpha) * R)
                G = int(alpha * 255 + (1.0 - alpha) * G)
                B = int(alpha * 0   + (1.0 - alpha) * B)
                pix[i, j] = (R, G, B)

        im.save(filename)

class Tester:
    """
    Tester class.
    Contains methods for testing purposes.
    """
    def _log(self, message):
        print message
        sys.stdout.flush()

    def test_spline_builder(self, sample_size = 5):
        x = np.linspace(0, 2 * np.pi, 100)
        y = np.sin(x)

        sub_x = [x[0]] + sorted(random.sample(x[1 : -1], sample_size)) + [x[-1]]
        sub_y = np.sin(sub_x)

        sb = SplineBuilder()
        sb.build(sub_x, sub_y, 1.0, 1.0)
        z = [sb.f(point) for point in x]

        plt.plot(x, y, 'b--', sub_x, sub_y, 'ro', x, z, 'g')
        plt.show()

    def test_spline_split_by_length(self, split_pieces = 5):
        x = np.linspace(0, 2 * np.pi, 500)
        y = np.sin(x)
```

```
        sb = SplineBuilder()
        sb.build(x, y, 1.0, 1.0)
        z = [sb.f(point) for point in x]

        sb.split_by_length(split_pieces)
        mx = sb.mile
        my = np.sin(mx)

        plt.plot(x, y, 'b--', x, z, 'g', mx, my, 'ro')
        plt.show()

    def test_image_loading(self, filename = 'samples/2.png'):
        qfb = QualityFunctionBuilder()
        qfb.load_from_image(filename)
        print qfb.w, qfb.h
        print qfb._im[0][0], qfb._im[400][100]
        print qfb._im[220][170], qfb._im[625][50]

    def test_Q_calculation(self, filename = 'samples/2.png'):
        qfb = QualityFunctionBuilder()
        qfb.load_from_image(filename)
        qfb.set_custom_terrain_size((2403, 1993))
        axis_x = np.linspace(0, 2403, 240)
        axis_y = np.linspace(0, 1993, 199)
        img = [[qfb.Q(x, y) for x in axis_x] for y in axis_y]
        plt.imshow(img)
        plt.show()

    def test_trajectory_drawing(self, filename = 'samples/2.png'):
        qfb = QualityFunctionBuilder()
        qfb.load_from_image(filename)
        car = CarBuilder()
        vtb = VehicleTrajectoryBuilder(qfb, car)
        vtb._generate_random_trajectory(10)
        vtb.save_to_file('samples/result.png')

    def test_quality_along_trajectory(self):
        qfb = QualityFunctionBuilder()
        qfb.load_from_image('samples/2.png')
        qfb.set_custom_terrain_size((1500, 300))
        car = CarBuilder()
        vtb = VehicleTrajectoryBuilder(qfb, car)

        x, y = vtb._generate_straight_trajectory(10, rebuild_spline = False)
        y[1] -= 15.0
        y[2] -= 25.0
        y[3] -= 25.0
        y[4] -= 30.0
        y[5] -= 35.0
        y[6] -= 40.0
        y[7] -= 35.0
        y[8] -= 20.0
        y[9] -= 5.0
        vtb._sb.build(x, y)
        qat, qs = vtb._quality_along_trajectory(100, save_trace = True)
```

```
        log("Attempt #0: quality = {} {}\n".format(qat, qs))
        print

        # vtb.show()

    def test_train_trajectory(self):
        qfb = QualityFunctionBuilder()
        qfb.load_from_image('samples/2.png')
        qfb.set_custom_terrain_size((1500, 300))
        car = CarBuilder()
        vtb = VehicleTrajectoryBuilder(qfb, car)

        vtb.train_trajectory(10)
        vtb.show()
        # vtb.save_to_file('samples/result.png')

def main(filename):
    qfb = QualityFunctionBuilder()
    qfb.load_from_image(filename)
    qfb.set_custom_terrain_size((1500, 300))
    car = CarBuilder()

    vtb = VehicleTrajectoryBuilder(qfb, car)
    vtb.train_trajectory(attempts = DEFAULT_ATTEMPTS)
    vtb.show()

if __name__ == '__main__':
    # Tester().test_spline_builder()
    # Tester().test_spline_split_by_length(5)
    # Tester().test_image_loading()
    # Tester().test_Q_calculation()
    # Tester().test_trajectory_drawing()
    # Tester().test_quality_along_trajectory()
    # Tester().test_train_trajectory()

    if len(sys.argv) == 1:
        main(DEFAULT_FILENAME)
    else:
        filename = sys.argv[1]
        main(filename)
    # main('samples/3.png')
    pass
```