Dokumentacja programu zaliczeniowego

Język Python 2025/2026

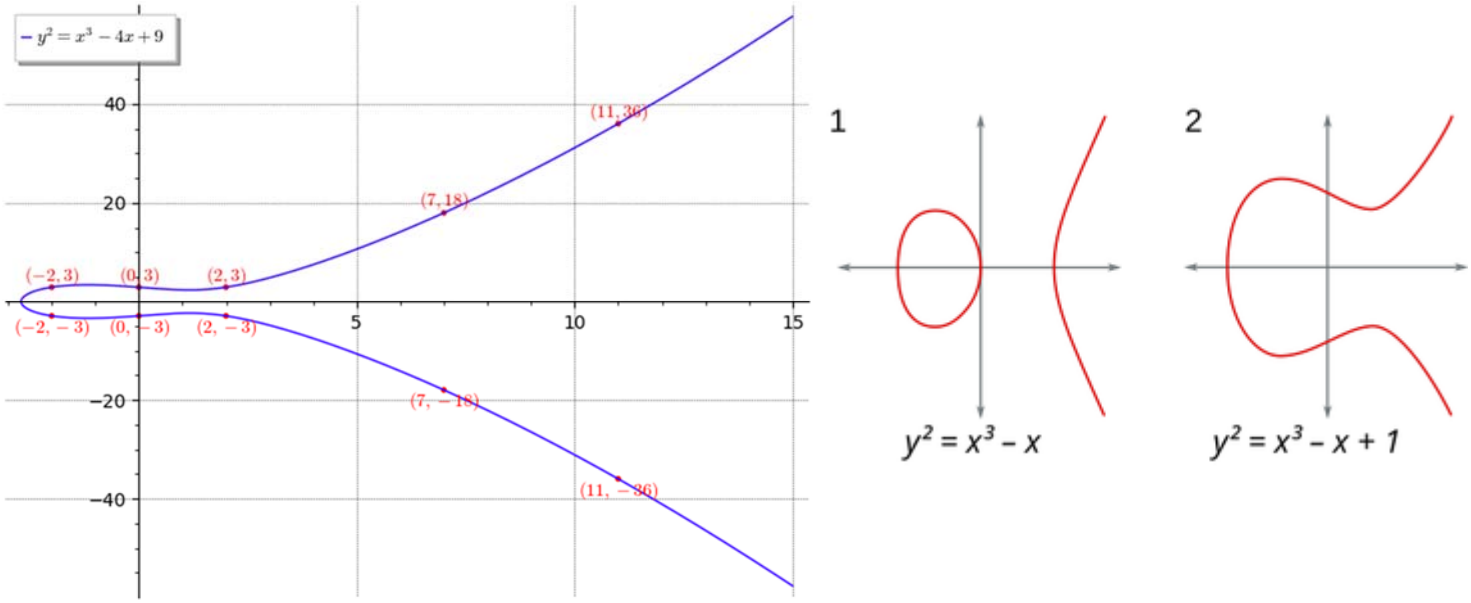
Mój program zaliczeniowy umożliwia wykonywanie operacji na krzywych eliptycznych. Program działa w formie importowanego modułu języka Python, dzięki czemu może być w prosty sposób dołączony do innych projektów, w celu wzbogacenia ich o nowe funkcjonalności. Program wykorzystuje algorytmy kryptograficzne oraz zagadnienia z działu arytmetyki modularnej.

Krzywe eliptyczne są zbiorami punktów na płaszczyźnie określonymi równaniem matematycznym. Równanie to, dla krzywych eliptycznych w postaci Weierstrassa ma postać: $y^2 = x^3 + ax + b$ gdzie a oraz b to dowolne współczynniki determinujące kształt krzywej. Parametry te muszą jednak spełniać następujący warunek. Niech $\Delta = 4a^3 + 27b^2$ Wyróżnik równania krzywej czyli Δ musi być różny od *0*. Jeśli wyróżnik jest równy *0* to krzywa staje się osobliwa i traci swoje właściwości.

Punkt w przestrzeni dwuwymiarowej o współrzędnych *(x, y)* należy do krzywej wtedy, gdy spełnia jej równanie. Jeśli punkt *(x, y)* należy do krzywej, to punkt *(x, -y)* również należy do krzywej.

Każda krzywa eliptyczna posiada również punkt O nazywany punktem w nieskończoności. Punkt ten nie należy do krzywej i nie spełnia jej równania. Punkt w nieskończoności jest elementem neutralnym względem dodawania punktów.

Krzywe eliptyczne mogą być zdefiniowane na zbiorze liczb rzeczywistych oraz w ciele skończonym (ciele Galois). Krzywe eliptyczne zdefiniowane na zbiorze liczb rzeczywistych nie mają ograniczeń na wartości współrzędnych punktów oraz parametrów krzywej. Każda z wartości: a, b, x, y może być dowolną liczbą rzeczywistą.



Wykresy przykładowych krzywych eliptycznych

Krzywe eliptyczne zdefiniowane na ciałach skończonych mają nieco inną postać równania. Krzywa w tej postaci jest określona wzorem: $y^2 = x^3 + ax + b \pmod p$

W takim przypadku krzywa jest określona nad ciałem ***GF(p)***

Większość krzywych eliptycznych wykorzystywanych w praktyce jest określona na ciałach, dla których ***p*** jest liczbą pierwszą. Gwarantuje to istnienie odwrotności modularnej dla każdej liczby większej od 0 i mniejszej od ***p***

Taka postać narzuca restrykcje na wartości współczynników. Zgodnie z zasadami arytmetyki modularnej, zarówno parametry krzywej: ***a***, ***b***, ***p*** jak i współrzędne punktów ***x***, ***y*** leżących na krzywej powinny być liczbami całkowitymi.

Na każdej krzywej eliptycznej są zdefiniowane 4 podstawowe operacje, pozwalające wykonywać działania na punktach do niej należących.

- Negacja punktu. Posiadając punkt **P** należący do krzywej chcę otrzymać punkt przeciwny równy **–P**

- Podwojenie punktu. Posiadając punkt $\boldsymbol{P}$ należący do krzywej chcę otrzymać punkt $\boldsymbol{Q}$ leżący na krzywej: $\boldsymbol{Q} = \boldsymbol{2P}$

- Dodawanie dwóch punktów. Posiadając dwa różne punkty należące do krzywej: $\boldsymbol{P}$ oraz $\boldsymbol{Q}$ chcę otrzymać punkt $\boldsymbol{R}$ należący do krzywej będący sumą tych dwóch punktów: $\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{Q}$

Operacja odejmowania dwóch punktów na krzywej jest równoważna z dodaniem do punktu pierwszego negacji punktu drugiego:

$\boldsymbol{R} = \boldsymbol{P} - \boldsymbol{Q} \Leftrightarrow \boldsymbol{R} = \boldsymbol{P} + (-\boldsymbol{Q})$

- Pomnożenie punktu przez skalar. Posiadając punkt $\boldsymbol{P}$ leżący na krzywej oraz dodatnią liczbę całkowitą $\boldsymbol{n}$ chcę otrzymać punkt $\boldsymbol{Q}$ leżący na krzywej będący iloczynem punktu $\mathbf{P}$ $\boldsymbol{n}$ razy: $\boldsymbol{Q} = \boldsymbol{P} * \boldsymbol{n}$ Operacja ta jest równoważna z dodaniem punktu P do siebie samego n razy.

Wszystkie z powyższych operacji powinny uwzględniać działania z punktem w nieskończoności $\boldsymbol{O}$

Krzywe eliptyczne mają wiele zastosowań praktycznych. Przy użyciu krzywych eliptycznych w 1993 roku udowodniono wielkie twierdzenie Fermata. Algorytmy wykorzystujące krzywe są także używane do znajdowania dzielników dużych liczb złożonych. Głównym zastosowaniem krzywych eliptycznych jest jednak kryptografia, gdzie są one wykorzystywane podczas wymiany kluczy (ECDH) oraz w podpisach cyfrowych (ECDSA). Podpisy cyfrowe oparte o krzywe eliptyczne są jednym z głównych elementów mojego programu.

Bezpieczeństwo algorytmów kryptograficznych opartych na krzywych eliptycznych jest zagwarantowane poprzez problem dyskretnego logarytmu przeniesiony na krzywe eliptyczne. Jeśli liczba elementów w grupie jest

wystarczająco duża – liczba ***p*** ma długość przynajmniej 128 bitów lub większą niemożliwe jest, przy obecnej mocy obliczeniowej komputerów stwierdzenie, posiadając jedynie dwa punkty na krzywej eliptycznej ile razy jeden z tych punktów jest większy od drugiego. Posiadając dwa punkty: ***P*** oraz ***G*** takie że

P = ***G*** * ***n*** gdzie ***n*** jest skalarem a * operacją mnożenia punktu przez skalar niemożliwe jest odzyskanie wartości ***n***.

Aby móc wykorzystywać podpisy cyfrowe na krzywej należy ustalić punkt bazowy ***G*** zwany generatorem oraz jej rząd ***v*** czyli liczbę wszystkich punktów na niej leżących wraz z punktem w nieskończoności.

W celu korzystania z podpisów cyfrowych potrzebna jest para kluczy: klucz prywatny oraz klucz publiczny. Klucz prywatny służy do tworzenia podpisu cyfrowego. Podpis cyfrowy dla danej wiadomości to para liczb (***r***, ***s***)

Każdy kto posiada klucz publiczny odpowiadający danemu kluczowi prywatnemu może zweryfikować, czy dana wiadomość została podpisana przez posiadacza tego klucza, a więc czy pochodzi od prawidłowego nadawcy i nie została zmodyfikowana podczas transmisji.

Klucz prywatny ***d*** jest skalarem zawierającym się w przedziale [1, ***v*** – 1]

Klucz publiczny ***P*** jest punktem znajdującym się na krzywej. Para kluczy spełnia zależność ***P*** = ***d*** * ***G*** Jeśli v jest odpowiednio duże i parametry krzywej zostały dobrane odpowiednio to nie jest możliwe odzyskanie klucza prywatnego posiadając jedynie klucz publiczny i tym samym sfałszowanie wiadomości.

Używając krzywych eliptycznych można również dokonać bezpiecznej wymiany kluczy w niezaufanym środowisku. Algorytm wymiany kluczy również bazuje na zagadnieniu dyskretnego logarytmu i jest rozwinięciem algorytmu Diffiego – Hellmana na krzywe eliptyczne.

Ważną zaletą algorytmów opartych o krzywe eliptyczne jest mały rozmiar kluczy. 256 – bitowe klucze na krzywych eliptycznych mają taką samą

kryptograficzną siłę jak 3072 – bitowe klucze używane w algorytmie RSA. Dodatkowo algorytmy podpisu cyfrowego i weryfikacji na krzywych eliptycznych są szybsze i bardziej wydajne niż te używające RSA.

Struktura programu

Program składa się z 4 plików. Zawartość każdego z plików jest opisana poniżej.

- *elliptic.py* Plik zawiera dwie klasy. Klasa *point* opisuje punkt w przestrzeni dwuwymiarowej. Klasa *ellipticcurve* jest główną klasą programu. Pozwala wykonywać operacje na krzywych eliptycznych oraz generować pary kluczy, podpisywać i sprawdzać poprawność podpisanej wiadomości.

- *specialcurve.py* Plik zawiera cztery krzywe eliptyczne wykorzystywane w praktyce. Krzywe są odpowiednio 128, 256, 384 oraz 521 bitowe. Są one gotowe do użycia w celu podpisywania i weryfikacji wiadomości.

- *utils.py* Plik zawiera pomocnicze algorytmy dotyczące sprawdzania pierwszości liczb oraz arytmetyki modularnej – np. modularny pierwiastek

- *test_module.py* Plik zawiera testy sprawdzające poprawność modułu.

elliptic.py

Klasa *point*

point(x, y) – konstruktor. Tworzy nową instancję klasy point opisującej punkt w kartezjańskiej przestrzeni dwuwymiarowej. Oba parametry *x* oraz *y* powinny być liczbami całkowitymi

__eq__(other) – operator porównania z innym punktem. Zwraca *True* jeśli punkty są identyczne, *False* w przeciwnym wypadku

__str__(), __repr__() – umożliwiają w prosty sposób dla użytkownika wyświetlenie reprezentacji punktu.

Klasa *ellipticcurve*

ellipticcurve(a, b, p, basepoint = None, order = None)

Konstruktor krzywej eliptycznej. Zmienne *a*, *b* oraz *p* są parametrami krzywej. Zmienna *p* musi być liczbą pierwszą. Zmienne *a* oraz *b* muszą spełniać warunek nieosobliwości krzywej: $4a^3 + 27b^2 \pmod p \neq 0$

Podanie zmiennych *basepoint* oraz *order* w konstruktorze nie jest wymagane, jednak bez nich nie jest możliwe korzystanie z kryptograficznych funkcji dotyczących podpisów cyfrowych. Zmienne te mogą być uzupełnione później.

basepoint – zmienna klasy *point*. Jest to punkt bazowy na krzywej, czyli generator ***G***

order – rząd krzywej, czyli liczba wszystkich punktów na nie leżących wraz z punktem w nieskończoności. Zmienna jest liczbą całkowitą

setbasepoint(basepoint), setorder(order)

Funkcje umożliwiają dodanie zmiennych *basepoint* oraz *order* jeśli nie podano ich w konstruktorze

bisoncurve(pt)

Sprawdza czy punkt *pt* leży na krzywej czyli spełnia jej równanie. Funkcja zwraca *True* jeśli punkt leży na krzywej, *False* w przeciwnym wypadku

pneg(pt)

Zwraca punkt przeciwny – negację punktu *pt*

pdbl(pt)

Zwraca podwojony punkt *pt* – 2 * *pt*

Punkt wejściowy *pt* musi leżeć na krzywej

pointadd(pt1, pt2)

Dodaje punkty *pt1* oraz *pt2* i zwraca punkt *P* = *pt1* + *pt2*

Oba punkty wejściowe muszą leżeć na krzywej albo muszą być punktami w nieskończoności. Zwracany punkt *P* również leży na krzywej albo jest punktem w nieskończoności.

pointsub(pt1, pt2)

Odejmuje punkty *pt1* oraz *pt2* i zwraca punkt *P* = *pt1* - *pt2*

Oba punkty wejściowe muszą leżeć na krzywej albo muszą być punktami w nieskończoności. Zwracany punkt *P* również leży na krzywej albo jest punktem w nieskończoności.

pointmul(pt, num)

Wykonuje operację mnożenia punktu przez skalar i zwraca punkt P = pt * num

Punkt pt musi leżeć na krzywej albo być punktem w nieskończoności. Zmienna num powinna być dodatnią liczbą całkowitą.

Zwracany punkt *P* również leży na krzywej albo jest punktem w nieskończoności. Funkcja działa szybko i jest efektywnie napisana – działa w czasie $\log_2(num)$ Dla 256 – bitowej liczby wejściowej mającej wartość > 2^{256} wykonywanych jest jedynie 256 operacji.

randpoint()

Zwraca losowy punkt leżący na krzywej.

recovery(x)

Na podstawie podanej w parametrze koordynaty x funkcja próbuje odzyskać odpowiadającą mu wartość y taką że punkt (x, y) leży na krzywej.

Jeśli na krzywej istnieje punkt o podanej wartości x funkcja zwraca dwa punkty: (x, y) oraz (x, ***p*** – y)

Jeśli na krzywej nie istnieje taki punkt, funkcja rzuca wyjątek ValueError

getkeypair()

Funkcja zwraca parę (d, P) – klucz prywatny (liczba) oraz klucz publiczny (punkt)

Funkcja może być użyta jedynie w przypadku, gdy na krzywej jest ustalony punkt bazowy ***G***

sign(privatekey, msg, hashfunc = hashlib.sha256)

Funkcja zwraca podpis cyfrowy wiadomości *msg* podpisanej przy użyciu klucza prywatnego *privatekey*

Podpis cyfrowy jest parą liczb (r, s)

Argument *privatekey* jest dodatnią liczbą całkowitą. Wiadomość do podpisania jest ciągiem znaków typu *str* albo *bytes*

Z racji iż krzywe eliptyczne używają kluczy o stosunkowo małym rozmiarze w porównaniu do długości wiadomości wiadomość musi zostać przetworzona przez funkcję hashującą przed podpisaniem. Domyślnie używana funkcja to

sha256 ale użytkownik może wybrać dowolną funkcję hashującą z modułu *hashlib*

Dozwolone wartości to: *md5*, *sha1*, *sha224*, *sha256*, *sha384*, *sha512*

Funkcja może być użyta jedynie w przypadku, gdy na krzywej jest ustalony punkt bazowy ***G*** oraz jej rząd

verify(publickey, msg, sig, hashfunc = hashlib.sha256)

Funkcja sprawdza poprawność podpisu cyfrowego kluczem publicznym w argumencie *publickey*

Klucz publiczny jest punktem leżącym na krzywej. Wiadomość w parametrze *msg* jest ciągiem znaków typu *str* albo *bytes*

Parametr *sig* jest podpisem cyfrowym wiadomości przekazywanym jako para dwóch dodatnich liczb całkowitych

Podobnie jak w funkcji *sign* istnieje możliwość ustawienia innej funkcji hashującej do weryfikacji podpisu wiadomości. Funkcja hashująca musi być taka sama jaka została użyta do podpisania wiadomości, w przeciwnym razie podpis będzie uznany za nieprawidłowy.

Funkcja zwraca *True* jeśli podpis jest prawidłowy oraz *False* w przeciwnym wypadku.

Funkcja może być użyta jedynie w przypadku, gdy na krzywej jest ustalony punkt bazowy ***G*** oraz jej rząd.

recoverpub(msg, sig, hashfunc = hashlib.sha256)

Na podstawie podanej wiadomości oraz jej podpisu cyfrowego funkcja zwraca dwa klucze publiczne, które poprawnie weryfikują daną wiadomość. Jeden z nich odpowiada kluczowi prywatnemu, którym dana wiadomość została podpisana. Algorytm bazuje na odwróconym algorytmie weryfikacji podpisu. Zwracane są dwa potencjalne klucze, gdyż wykorzystuje on pierwiastek modularny, który zwraca dwie wartości odpowiadające punktom

(x, y) oraz (x, ***p*** – y)

Wiadomość w parametrze *msg* jest ciągiem znaków typu *str* albo *bytes*

Parametr *sig* jest podpisem cyfrowym wiadomości przekazywanym jako para dwóch dodatnich liczb całkowitych

Funkcja hashująca przekazana w parametrze *hashfunc* musi być taka sama jaka została użyta do podpisania wiadomości, w przeciwnym razie algorytm odzyska niepoprawne klucze.

specialcurve.py

Plik zawiera 4 krzywe eliptyczne, które są używane w kryptografii w praktyce.

secp128r1 – 128 bitowa

secp256k1 – 256 bitowa, używana m. in. w kryptowalucie Bitcoin

brainpoolp384r1 – 384 bitowa

p521 – 521 bitowa

Wszystkie krzywe mają parametry zgodne z oficjalnymi standardami i mają ustalone punkty bazowe oraz rzędy, dzięki czemu są gotowe do działań kryptograficznych.

utils.py

Plik zawiera pomocnicze funkcje matematyczne umożliwiające wykonywanie operacji w arytmetyce modularnej.

bisprime(num, ntrial = 10)

Funkcja sprawdza czy liczba podana w argumencie *num* jest liczbą pierwszą. Algorytm sprawdzający działa szybko i bazuje na małym twierdzeniu Fermata.

Parametr *ntrial* jest liczbą prób testu pierwszości. Domyślnie jest ustawiony na 10.

Funkcja zwraca *True* jeśli wszystkie próby testu pierwszości się powiodą oraz *False* jeżeli przynajmniej 1 próba nie powiedzie się.

legendre(x, p)

Funkcja zwraca symbol Legendre'a dla liczby pierwszej *p* i liczby *x*

Funkcja jest wykorzystywana do sprawdzania czy dla danej liczby *x* istnieje pierwiastek modularny modulo *p* czyli taka liczba *a* że $a^2 \bmod p = x$

Funkcja zwraca 1 jeżeli pierwiastek modularny dla liczby *x* modulo *p* istnieje

p - 1 jeżeli nie istnieje i 0 jeżeli *x* (mod *p*) = 0

hasmodularsqrt(x, p)

Funkcja sprawdza czy istnieje pierwiastek modularny *x* modulo *p* i zwraca wartość *True* jeżeli istnieje oraz *False* w przeciwnym razie.

modularsqrt(x, p)

Funkcja zwraca listę pierwiastków modularnych dla liczby *x* modulo liczbę pierwszą *p* czyli takich liczb *a* spełniających równanie $a^2 \bmod p = x$

Skoro liczba *a* jest prawidłowym rozwiązaniem równania to drugim rozwiązaniem jest liczba *p* - *a*

Funkcja zwraca więc dwie liczby: *a* oraz *p* – *a* gdyż $(p - a)^2 \bmod p = a^2 \bmod p = x$

Generalnym sposobem obliczanie pierwiastka modularnego jest algorytm Tonelliego – Shanksa. Jednakże ten algorytm jest kosztowny obliczeniowo więc zastosowałem 2 optymalizacje.

1) Jeżeli *p* (mod 4) = 3 to pierwiastek można szybko obliczyć w następujący sposób: $a = x^{\frac{p+1}{4}} \ (mod\ p)$ Ten wzór wynika z małego twierdzenia Fermata. Rozwiązaniami są *a* oraz *p* – *a*

2) Jeżeli p (mod 8) = 5 to pierwiastek można szybko obliczyć w następujący sposób: $v_1 = (2x)^{\frac{p-5}{8}} \ (mod\ p)$ $v_2 = 2xv_1^2 \ (mod\ p)$

$a = xv_1(v_2 - 1) \text{ (mod p)}$ Rozwiązaniami są *a* oraz *p* – *a*

W przeciwnym wypadku należy zastosować algorytm Tonelliego – Shanksa

tonelli(x, p)

Funkcja oblicza pierwiastek modularny z *x* modulo *p* algorytmem Tonelliego – Shanksa. Funkcja zwraca dwa rozwiązania.

Testy programu w formacie dla modułu *pytest* znajdują się w pliku *test_module.py* Można je uruchomić poleceniem: *python3 test_module.py*

Podczas testowania programu nie napotkałem żadnych błędów. Wszystkie testy zwróciły poprawne wyniki

Mój program pozwala w prosty sposób wykorzystać krzywe eliptyczne w celach kryptograficznych i matematycznych w dowolnym projekcie w języku Python dzięki prostocie importowania modułu. Funkcje są zoptymalizowane, aby program działał w możliwie najkrótszym czasie i mają jasne przeznaczenie. Kod programu jest napisany w sposób przejrzysty, najważniejsze funkcje są opatrzone komentarzami, które zwięźle opisują ich zastosowanie.

Marek Moroński