LICEO CANTONALE DI MENDRISIO

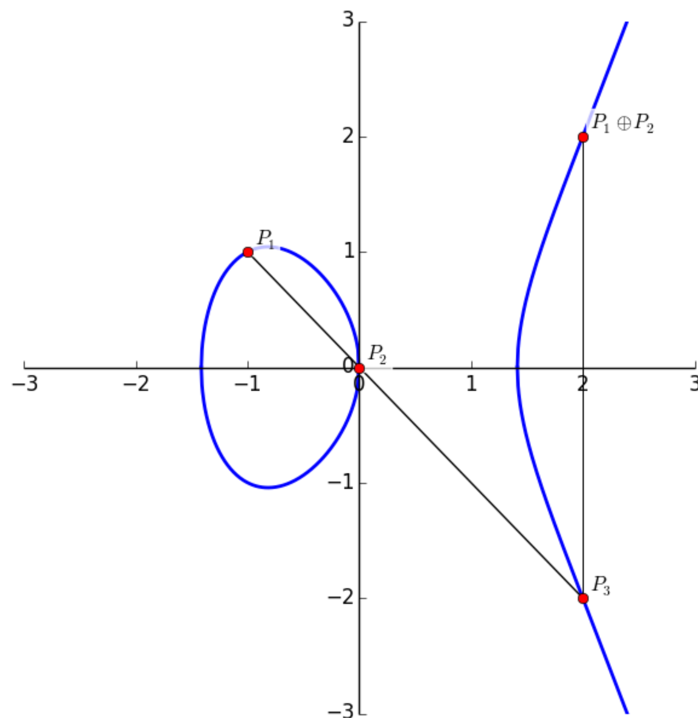
Sean Bone

Curve Ellittiche

Lavoro di maturità di matematica e informatica

Docenti: Claudio Marsan e Piero Antognini

Anno 2014



Curve ellittiche

Sean Bone

13 gennaio 2015

Indice

1 Strutture algebriche **3**
1.1 Gruppi commutativi . . . 3
1.2 Anelli . . . 3
1.3 Campi . . . 3

2 Aritmetica modulare **4**
2.1 Il modulo . . . 4
2.1.1 Congruenza modulare . . . 4
2.1.2 Proprietà di calcolo . . . 5
2.2 Inverso moltiplicativo modulo n . . . 5
2.2.1 Algoritmo esteso di Euclide . . . 5
2.2.2 Il piccolo teorema di Fermat . . . 6
2.2.3 Esponenziazione modulare rapida . . . 6
2.3 Gli anelli $\mathbb{Z}_n$ e $\mathbb{Z}_n^*$ . . . 7

3 Curve ellittiche **8**
3.1 Introduzione . . . 8
3.2 Gruppo ellittico su $\mathbb{R}$ . . . 10
3.3 Gruppi ellittici su $\mathbb{Z}_p$ . . . 12

4 Implementazione delle curve ellittiche in Python **14**
4.1 La classe `ECPoint` . . . 14
4.2 La classe `EC`: gruppo ellittico su $\mathbb{R}$ . . . 15
4.2.1 Somma di punti . . . 16
4.2.2 Il problema dell'approssimazione . . . 17
4.2.3 Multipli di un punto . . . 19
4.3 Tracciare le curve ellittiche su $\mathbb{R}$ . . . 21
4.3.1 Ottenere gli zeri . . . 23
4.3.2 Il metodo di Newton con Python . . . 23
4.3.3 Presenza della 'bolla' . . . 25
4.3.4 Disegno della curva in Python . . . 26
4.3.5 Disegno della somma di punti . . . 29
4.4 La classe `EC_modp`: gruppo ellittico su $\mathbb{Z}_p$ . . . 31

5 Bibliografia e sitografia **34**

Introduzione

In questo lavoro studieremo le curve ellittiche e le loro proprietà. Conosceremo alcune strutture algebriche e vedremo come introdurre un'operazione di somma fra i punti di una curva ellittica in modo tale da ottenere una struttura gruppo abeliano. Faremo poi un veloce giro d'orizzonte sull'aritmetica modulare, in modo da poter replicare la struttura di gruppo abeliano ellittico anche su $\mathbb{Z}_n$.

In seguito utilizzeremo il linguaggio di programmazione PYTHON per rappresentare questi gruppi ed i loro elementi al computer. In particolare, vogliamo poter calcolare la somma fra punti su una curva e rappresentare graficamente le curve e le somme di punti.

Sebbene si tratti di un'applicazione relativamente semplice, dovremo confrontarci con un problema classico della matematica implementata attraverso il computer: l'approssimazione dei decimali.

1 Strutture algebriche

1.1 Gruppi commutativi

Definizione 1.1. Un insieme $G \neq \varnothing$ munito di un'operazione binaria $+$, scritto $(G, +)$, è detto *gruppo abeliano* (o *gruppo commutativo*) se valgono le seguenti proprietà:

1. L'operazione è *chiusa*: $\forall x, y \in G$ anche $x + y \in G$.
2. L'operazione è *associativa*: $\forall x, y, z \in G$ vale $(x + y) + z = x + (y + z)$.
3. G possiede un *elemento neutro*, che denotiamo con e: per ogni $x \in G$, vale $x + e = e + x = x$.
4. Ogni elemento di G ha un *inverso*: per ogni $x \in G$ esiste un $y \in G$ tale che $x + y = y + x = e$.
5. L'operazione è *commutativa*: $\forall x, y \in G$ vale $x + y = y + x$.

I seguenti sono gruppi commutativi: $(\mathbb{Z}, +)$, $(\mathbb{Q}, +)$, $(\mathbb{R}, +)$, $(\mathbb{C}, +)$, $(\mathbb{Q} - \{0\}, \cdot)$, $(\mathbb{R} - \{0\}, \cdot)$, $(\mathbb{C} - \{0\}, \cdot)$.

1.2 Anelli

Definizione 1.2. Un insieme $A \neq \varnothing$ munito di due operazioni binarie $+$ (addizione) e $\cdot$ (moltiplicazione), scritto $(A, +, \cdot)$, è detto *anello commutativo unitario* se valgono le seguenti proprietà:

1. $(A, +)$ è un gruppo commutativo.
2. L'operazione $\cdot$ è *associativa*, *commutativa* e ha un *elemento neutro*.

I seguenti sono di anelli commutativi unitari: $(\mathbb{Z}, +, \cdot)$, $(\mathbb{Q}, +, \cdot)$, $(\mathbb{R}, +, \cdot)$, $(\mathbb{C}, +, \cdot)$.

1.3 Campi

Definizione 1.3. Un insieme $C \neq \varnothing$ munito di due operazioni binarie $+$ (addizione) e $\cdot$ (moltiplicazione), scritto $(C, +, \cdot)$, è detto *campo* o *corpo commutativo* se valgono le seguenti proprietà:

1. $(C, +)$ è un gruppo commutativo con elemento neutro e_1.
2. $(C - \{e_1\}, \cdot)$ è un gruppo commutativo con elemento neutro e_2, dove $e_1 \neq e_2$.
3. Vale la proprietà distributiva: se $x, y, z \in C$, $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.

Un campo è quindi un anello commutativo unitario per il quale tutti gli elementi diversi dall'elemento neutro dell'addizione possiedono un inverso moltiplicativo.

Come visto, $(\mathbb{Z}, +, \cdot)$ è un anello commutativo unitario ma non un campo: infatti in $\mathbb{Z}$ gli unici elementi ad avere l'inverso moltiplicativo sono 1 e -1. Sono invece campi: $(\mathbb{Q}, +, \cdot)$, $(\mathbb{R}, +, \cdot)$, $(\mathbb{C}, +, \cdot)$.

2 Aritmetica modulare

2.1 Il modulo

Pensiamo ad un orologio a lancette (12 ore): se adesso sono le 10:00, fra quattro ore l'orologio non segnerà le 14:00, bensì le 2:00. Se usiamo il simbolo $\oplus$ per indicare la somma di ore possiamo scrivere $10 \oplus 4 = 2$. Se invece fossero le 5:00 del mattino, fra ventiquattro ore saranno ancora le 5:00, non le 29:00: $5 \oplus 24 = 5$. Questi sono esempi di un'operazione nota come addizione *modulo 12*. Facciamo qualche esempio ulteriore, scrivendo $a + b \mod 12$ invece di $a \oplus b$:

$$\begin{aligned} 7 + 8 &\mod 12 = 3; \\ 4 + 3 &\mod 12 = 7; \\ 6 + 11 &\mod 12 = 5. \end{aligned}$$

Possiamo notare che il risultato delle addizioni modulo 12 coincide con il resto della somma quando questa viene divisa per 12.

Generalizzando possiamo dare la definizione seguente. Siano dati gli interi a, q, n, r con $n > 0$ e $r \geq 0$ tali che:

$$a = q \cdot n + r.$$

Allora diciamo che *a modulo n vale r* e scriveremo

$$a \mod n = r,$$

o, in modo equivalente:

$$a \equiv r \pmod{n}.$$

Il numero $a \mod n$ è quindi il resto della divisione di a per n. Negli esempi precedenti, $5 + 24 = 29 = 2 \cdot 12 + 5$, dunque $(5 + 24) \equiv 5 \pmod{12}$.

2.1.1 Congruenza modulare

Siano dati tre interi a, b, n con $n > 0$. a e b si dicono *congruenti modulo n* se hanno lo stesso resto quando vengono divisi per n:

$$a \mod n = b \mod n.$$

Scritto in modo più conciso:

$$a \equiv b \pmod{n}.$$

In modo equivalente, a e b sono congruenti modulo n se $a - b$ è un multiplo di n, cioè $a - b \equiv 0 \pmod{n}$.

Ad esempio:

$$\begin{aligned} 5 &\equiv 3 \pmod{2}; \\ 8 &\equiv 5 \pmod{3}; \\ -8 &\equiv 6 \pmod{7}. \end{aligned}$$

2.1.2 Proprietà di calcolo

Può tornarci utile a questo punto ricordare le seguenti proprietà di calcolo, che si dimostrano facilmente.

Siano dati tre interi a, b, n, con $n > 0$. Allora valgono:

$$\begin{aligned}(a+b) \mod n &\equiv (a \mod n) + (b \mod n);\\ (a \cdot b) \mod n &\equiv (a \mod n) \cdot (b \mod n).\end{aligned}$$

2.2 Inverso moltiplicativo modulo n

Dati gli interi $a, n > 0$, l'*inverso moltiplicativo modulo n* di a è, se esiste, l'intero a^{-1} tale che

$$a \cdot a^{-1} \equiv 1 \pmod{n}.$$

Il problema che ci poniamo ora è questo: come ricavare a^{-1} dati a e n?

2.2.1 Algoritmo esteso di Euclide

Una soluzione viene dalla *forma estesa dell'algoritmo di Euclide* per il calcolo del massimo comune divisore (nel seguito, utilizzando la terminologia inglese, gcd). Facciamo un esempio con $n = 37$ e $a = 14$:

$$\begin{aligned}37 &= 2 \cdot 14 + 9;\\ 14 &= 1 \cdot 9 + 5;\\ 9 &= 1 \cdot 5 + 4;\\ 5 &= 1 \cdot 4 + 1;\\ 4 &= 4 \cdot 1 + 0.\end{aligned}$$

Sappiamo ora che 1 è il massimo comune divisore fra 37 e 14. Ora procediamo a ritroso per trovare l'inverso moltiplicativo di $a \mod n$, ossia esprimiamo 1 come combinazione lineare di n e a:

$$\begin{aligned}1 &= 5 - 1 \cdot 4;\\ 1 &= 5 - (9 - 5) = 2 \cdot 5 - 9;\\ 1 &= 2 \cdot (14 - 9) - 9 = 2 \cdot 14 - 3 \cdot 9;\\ 1 &= 2 \cdot 14 - 3 \cdot (37 - 2 \cdot 14);\\ 1 &= 8 \cdot 14 - 3 \cdot 37.\end{aligned}$$

Così $1 \mod 37 = (8 \cdot 14 - 3 \cdot 37) \mod 37$, ossia $1 \equiv 8 \cdot 14 \pmod{37}$. Dunque possiamo dire che:

$$14^{-1} \equiv 8 \pmod{37}.$$

Infatti: $14 \cdot 8 = 112 \equiv 1 \pmod{37}$.

2.2.2 Il piccolo teorema di Fermat

La forma estesa dell'algoritmo di Euclide è parecchio laboriosa per il calcolo dell'inverso, ma il seguente risultato, noto come *piccolo teorema di Fermat*, può semplificare le cose in alcuni casi.

Teorema 2.1. *Siano dati due interi a, n, con $n > 0$. Se n è primo e $\gcd(a, n) = 1$, allora $a^{n-1} \equiv 1 \pmod{n}$.*

Quindi, nel nostro contesto possiamo dire che per n primo vale:

$$\boxed{1 \equiv a^{n-1} = a \cdot a^{n-2} \pmod{n}}$$

Dunque $a^{-1} \equiv a^{n-2} \pmod{n}$. Tornando all'esempio di prima, 37 è primo e $\gcd(37, 14) = 1$, quindi $14^{-1} \equiv 14^{35} \equiv 8 \pmod{37}$.

2.2.3 Esponenziazione modulare rapida

Ora, però, abbiamo un altro problema: calcolare $a^k \mod n$ è molto lungo per esponenti grandi. Se vogliamo poter usare il piccolo teorema di Fermat in un nostro futuro programma scritto in PYTHON dobbiamo assicurarci di avere un metodo rapido per fare questo genere di calcolo.

Ammettiamo di voler calcolare $a^{150} \mod n$. La forma binaria di 150 è $(10010110)_2$, dato che $150 = 2^7 + 2^4 + 2^2 + 2^1$. Il nostro calcolo diventa così:

$$\begin{aligned} a^{150} &= a^{2^7+2^4+2^2+2} = a^{2^7} a^{2^4} a^{2^2} a^2 \\ &= ((((((a^2)^2)^2)^2)^2)^2 \cdot (((a^2)^2)^2)^2 \cdot (a^2)^2 \cdot a^2 \\ &= ((((((a^2)^2)^2 \cdot a)^2)^2 \cdot a)^2 \cdot a)^2 \end{aligned}$$

Questa stessa forma può essere semplicemente ottenuta attraverso un metodo iterativo come il seguente:

$$\begin{array}{ll} \text{bit} & \text{potenze di } a \\ 1: & a \\ 0: & a^2 \\ 0: & (a^2)^2 \\ 1: & ((a^2)^2)^2 \cdot a \\ 0: & (((a^2)^2)^2 \cdot a)^2 \\ 1: & ((((a^2)^2)^2 \cdot a)^2)^2 \cdot a \\ 1: & (((((a^2)^2)^2 \cdot a)^2)^2 \cdot a)^2 \cdot a \\ 0: & ((((((a^2)^2)^2 \cdot a)^2)^2 \cdot a)^2 \cdot a)^2 \end{array}$$

Compiamo un'iterazione per ciascun bit che compone $150 = (10010110)_2$, partendo dal primo a sinistra. Per ciascun bit quadriamo il risultato e, se il bit in questione è pari a 1, moltiplichiamo per la base a. In questo modo raggiungiamo comunque il risultato desiderato, ma in modo più efficace: infatti possiamo calcolare

il modulo n del risultato a ciascuna iterazione, mantenendo i risultati fra 0 e $n-1$ e lavorando di conseguenza con dei numeri considerevolmente più piccoli.

Questo metodo può essere facilmente implementato in PYTHON, fornendo così, nel caso n sia primo, un metodo efficiente per il calcolo dell'inverso moltiplicativo modulo n.

2.3 Gli anelli $\mathbb{Z}_n$ e $\mathbb{Z}_n^*$

Se fissiamo un numero intero $n > 1$, i possibili resti delle divisioni per n sono i numeri 0, 1, ..., $n-1$. Se definiamo l'insieme $\mathbb{Z}_n = \{0, 1, ..., n-1\}$ è facile dimostrare che $(\mathbb{Z}_n, +)$ è un gruppo abeliano e che $(\mathbb{Z}_n, +, \cdot)$ è un anello commutativo unitario, l'*anello degli interi modulo n* ("+" è l'addizione modulo n, "·" è la moltiplicazione modulo n).

Ci poniamo ora la domanda seguente: è possibile definire un gruppo abeliano *moltiplicativo* su $\mathbb{Z}_n$? O più in generale: può $\mathbb{Z}_n$ avere una struttura di campo?

Vediamo facilmente che l'operazione della moltiplicazione è chiusa (modulo n), associativa, commutativa e che esiste l'elemento neutro (cioè 1). Non è però scontata l'invertibilità degli elementi: sicuramente dobbiamo escludere lo 0 (elemento neutro dell'addizione) poiché esso non possiede l'inverso (moltiplicativo) e inoltre vale:

$$\boxed{a \text{ è invertibile in } \mathbb{Z}_n \iff \gcd(a, n) = 1}$$

Possiamo così definire un nuovo insieme denominato con $\mathbb{Z}_n^*$:

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n | \gcd(a, n) = 1\}.$$

È poi facile verificare che $(\mathbb{Z}_n^*, \cdot)$ è un gruppo abeliano. Per $n = 10$, ad esempio, abbiamo:

$$\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}.$$

Infatti:

$$1^{-1} = 1, \quad 3^{-1} = 7, \quad 7^{-1} = 3, \quad 9^{-1} = 9,$$

mentre $2, 4, 5, 6, 8$ non hanno il massimo comune divisore con 10 uguale a 1.

Un teorema dell'algebra afferma inoltre che:

$$(\mathbb{Z}_n^*, +, \cdot) \text{ è un campo} \quad \iff \quad n \text{ è primo.}$$

Se n è un numero primo ovviamente avremo $\mathbb{Z}_n^* = \mathbb{Z}_n - \{0\}$.

3 Curve ellittiche

3.1 Introduzione

Una *curva ellittica* è una curva la cui equazione ha la forma seguente:

$$\boxed{\mathcal{E}\colon y^2 = x^3 + ax + b} \tag{1}$$

dove $4a^3 + 27b^2 \neq 0$. Questa condizione garantisce che il polinomio $x^3 + ax + b$ abbia tre zeri (prendiamo questo fatto per vero senza la dimostrazione, che sarebbe oltre la portata di questo lavoro, vedi per esempio [2]).

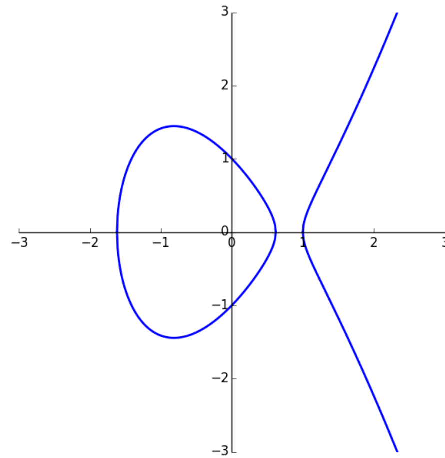


Figura 1: La curva $y^2 = x^3 - 2x + 1$.

Queste curve hanno una particolarità molto interessante: se una retta non verticale interseca una curva ellittica in due punti, allora ci sarà certamente anche un terzo punto d'intersezione (se la retta fosse tangente alla curva in un punto, quel punto verrà conteggiato due volte).

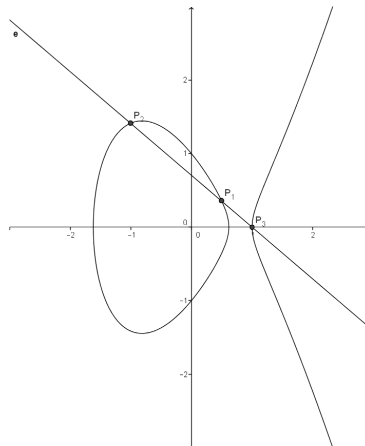


Figura 2: Intersezione di una retta con una curva ellittica.

Dati due punti $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$ appartenenti alla curva ellittica, possiamo quindi calcolare il terzo, $P_3 = (x_3, y_3)$, come segue.

Diamo per scontato che se $x_1 = x_2$, allora sarà $y_1 = y_2$, altrimenti la retta sarebbe verticale. Allo stesso modo, i due punti possono essere uguali (retta tangente), fintanto che $y_1 \neq 0$.

Se $x_1 \neq x_2$, la pendenza della retta passante per P_1 e P_2 è:

$$m = \frac{y_1 - y_2}{x_1 - x_2},$$

altrimenti, se $x_1 = x_2$ sarà:

$$m = \frac{3x_1^2 + a}{2y_1}.$$

Le coordinate del punto P_3 sono date da:

$$\begin{aligned} x_3 &= m^2 - x_1 - x_2, \\ y_3 &= m \cdot (x_3 - x_1) + y_1. \end{aligned}$$

Dimostrazione. m è la pendenza della retta passante per i due punti. È evidente per $x_1 \neq x_2$, dimostriamolo per $x_1 = x_2$. Dato che entrambi i punti soddisfano l'equazione (1), abbiamo:

$$\begin{aligned} y_1^2 - y_2^2 &= x_1^3 - x_2^3 + a \cdot (x_1 - x_2), \\ (y_1 - y_2) \cdot (y_1 + y_2) &= (x_1 - x_2)(x_1^2 + x_1 \cdot x_2 + x_2^2 + a), \end{aligned}$$

da cui:

$$m = \frac{y_1 - y_2}{x_1 - x_2} = \frac{x_1^2 + x_1 \cdot x_2 + x_2^2 + a}{y_1 + y_2}. \tag{2}$$

Con $x_1 = x_2$, si ha

$$m = \frac{3x_1^2 + a}{2y_1}.$$

L'equazione 2 vale per tutte le coppie di punti lungo la retta, quindi possiamo anche dire che:

$$\begin{aligned} m \cdot (y_3 + y_1) &= x_3^2 + x_3 \cdot x_1 + x_1^2 + a, \\ m \cdot (y_3 + y_2) &= x_3^2 + x_3 \cdot x_2 + x_2^2 + a. \end{aligned}$$

Sottraiamo la seconda equazione alla prima:

$$m \cdot (y_1 - y_2) = x_3 \cdot (x_1 - x_2) + (x_1^2 - x_2^2).$$

Dividiamo entrambi i lati per $(x_1 - x_2)$:

$$\begin{aligned} m \cdot m &= x_3 + x_1 + x_2, \\ x_3 &= m^2 - x_1 - x_2. \end{aligned}$$

Infine usiamo la definizione della pendenza della nostra retta, m, per calcolare y_3:

$$\begin{aligned} m &= \frac{y_3 - y_1}{x_3 - x_1}, \\ y_3 &= m \cdot (x_3 - x_1) + y_1. \end{aligned}$$

□

3.2 Gruppo ellittico su $\mathbb{R}$

Nel seguito vogliamo mostrare come sia possibile munire della struttura di gruppo abeliano l'insieme dei punti del piano $\mathbb{R}^2$ che giacciono su una data curva ellittica.

Definizione 3.1. Dati la curva ellittica $\mathcal{E}\colon y^2 = x^3+ax+b$ e due punti appartenenti ad essa, $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2) \neq (x_1, -y_1)$, definiamo l'operazione $\oplus$:

$$(x_1, y_1) \oplus (x_2, y_2) = (x_3, -y_3)$$

dove $(x_3, -y_3)$ è il **simmetrico rispetto all'asse** x di $P_3 = (x_3, y_3)$, il terzo punto d'intersezione della retta passante per P_1 e P_2 con la curva ellittica (vedi paragrafo precedente).

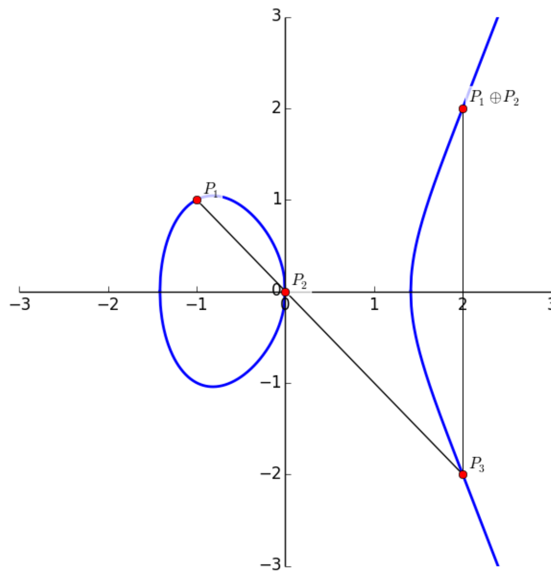


Figura 3: Somma di due punti sulla curva $y^2 = x^3 - 2x$.

Considerando i cinque assiomi che l'operazione $\oplus$ dovrebbe soddisfare affinché $(\mathcal{E}, \oplus)$ sia un gruppo abeliano, notiamo subito che la nostra operazione non è chiusa, poiché $(x, y) \oplus (x, -y)$ è indefinito; inoltre mancano elemento neutro e inversi. Si può invece verificare che $\oplus$ è associativa:

$$(P_1 \oplus P_2) \oplus P_3 = P_1 \oplus (P_2 \oplus P_3) \qquad \forall P_1, P_2, P_3 \in \mathcal{E}$$

(tralasciamo i calcoli perché sono molto lunghi e macchinosi) e commutativa:

$$P_1 \oplus P_2 = P_2 \oplus P_1 \qquad \forall P_1, P_2 \in \mathcal{E}$$

(i calcoli sono elementari).

Definizione 3.2. Sia $P = (x, y)$ un punto sulla curva ellittica $\mathcal{E}\colon y^2 = x^3 + ax + b$. Indichiamo il suo simmetrico rispetto all'asse x con P^{-1}; sarà $P^{-1} = (x, -y)$. Introduciamo inoltre ∞ come elemento neutro dell'operazione $\oplus$; per definizione valgono:

$$P \oplus \infty = \infty \oplus P = P \qquad \forall P \in \mathcal{E}$$

e

$$P \oplus P^{-1} = P^{-1} \oplus P = \infty.$$

Si può pensare a ∞, detto *punto all'infinito*, come a un punto con y infinitamente grande, in modo che ogni retta verticale passi per esso.

Se prendiamo la retta verticale passante per P e per ∞, il terzo punto d'intersezione con la curva sarà $P^{-1} = (x, -y)$, quindi $P \oplus \infty = (x, -(-y)) = (x, y) = P$. Questo significa che ∞ è effettivamente l'elemento neutro della somma di punti su una curva ellittica.

Dato che P e P^{-1} sono simmetrici rispetto all'asse x, la retta passante per questi due punti è verticale, quindi il terzo punto d'intersezione sarà ∞ (anche se non propriamente appartenente alla curva $\mathcal{E}$). Questo significa che P^{-1} è l'inverso di P, visto che $P \oplus P^{-1} = \infty$.

Abbiamo così risolto i nostri problemi: se al posto di $\mathcal{E}$ consideriamo $\mathcal{E} \cup \{\infty\}$, in questo insieme l'operazione $\oplus$ sarà chiusa, avrà l'elemento neutro e ogni elemento avrà il suo inverso.

Definizione 3.3. Siano a e b interi tali che

$$4a^3 + 27b^2 \neq 0$$

e sia

$$E(a, b) = \{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + ax + b\} \cup \{\infty\}.$$

Allora $(E(a, b), \oplus)$ è un gruppo abeliano, detto *gruppo ellittico* su $\mathbb{R}$; i suoi elementi sono i punti della curva ellittica $\mathcal{E}$ e il punto all'infinito ∞.

Facciamo un esempio semplice. Con $a = -2$ e $b = 1$, abbiamo la curva ellittica $y^2 = x^3 - 2x + 1$. L'insieme di partenza del nostro gruppo $(E(-2, 1), \oplus)$ è quindi

$$E(-2, 1) = \{(x, y) \in \mathbb{R}^2 | y^2 = x^3 - 2x + 1\} \cup \{\infty\}.$$

In $x = 1$ abbiamo un'intersezione con l'asse x, quindi solo un'immagine: $P_1(1, 0)$. In $x = -1$, invece, abbiamo due immagini:

$$y = \pm\sqrt{-1 + 2 + 1} = \pm\sqrt{2},$$

ossia due punti: $P_2(-1, \sqrt{2})$ e $P_2^{-1}(-1, -\sqrt{2})$.

Possiamo ora a calcolare la somma $P_1 \oplus P_2$. Dato che $x_1 \neq x_2$, la pendenza della retta passante per i due punti è

$$m = \frac{y_1 - y_2}{x_1 - x_2} = \frac{0 - \sqrt{2}}{1 - (-1)} = -\frac{\sqrt{2}}{2}.$$

Quindi calcoliamo il terzo punto d'intersezione fra la retta e la curva ellittica:

$$x_3 = m^2 - x_1 - x_2 = \frac{2}{4} - 1 - (-1) = \frac{1}{2},$$
$$y_3 = m \cdot (x_3 - x_1) + y_1 = -\frac{\sqrt{2}}{2} \cdot (\frac{1}{2} - 1) + 0 = \frac{\sqrt{2}}{4}.$$

Infine abbiamo:

$$P_1 \oplus P_2 = (x_3, -y_3) = (\frac{1}{2}, -\frac{\sqrt{2}}{4}).$$

3.3 Gruppi ellittici su $\mathbb{Z}_p$

L'aritmetica con le curve ellittiche che abbiamo visto finora si può benissimo applicare in *modulo* p, dove p è un numero primo: questa proprietà di p garantisce la calcolabilità del denominatore di m, che deve essere trattato come l'inverso moltiplicativo di un elemento di $\mathbb{Z}_p$. Nel seguito ammetteremo che p sia un numero primo con $p > 3$.

Definizione 3.4. Siano a, $b \in \mathbb{Z}_p$ con $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. L'insieme $\mathcal{E}$ dei punti $(x, y) \in \mathbb{Z}_p^2$ che soddisfano l'equazione

$$y^2 = x^3 + ax + b \pmod{p}$$

è detto *curva ellittica*. Su $\mathcal{E} \cup \{\infty\}$ definiamo l'operazione $\oplus$ *modulo* p nel modo seguente. Se $x_1 \equiv x_2 \pmod{p}$ e $y_1 \equiv -y_2 \pmod{p}$, allora

$$(x_1, y_1) \oplus (x_2, y_2) = \infty;$$

altrimenti:

- se $x_1 \not\equiv x_2 \pmod{p}$ e $\gcd(x_1 - x_2, n) = 1$, allora sia s l'inverso modulo p di $x_1 - x_2$ e sia
$$m = (y_1 - y_2) \cdot s \mod p;$$
- se $x_1 \equiv x_2 \pmod{p}$ e $\gcd(y_1 + y_2, n) = 1$, cioé $y_1 \equiv y_2 \pmod{p}$, allora sia s l'inverso modulo p di $2y_1$ e sia
$$m = (3x_1^2 + a) \cdot s \mod p,$$

e quindi definiamo:

$$(x_1, y_1) \oplus (x_2, y_2) \equiv (x_3, -y_3) \pmod{p},$$

dove

$$x_3 = (m^2 - x_1 - x_2) \mod p,$$
$$y_3 = (m \cdot (x_3 - x_1) + y_1) \mod p.$$

Definizione 3.5. Sia p un numero primo maggiore di 3 e siano $a, b \in \mathbb{Z}_p$ interi tali che sia soddisfatta la condizione

$$4a^3 + 27b^2 \not\equiv 0 \pmod{p};$$

sia inoltre

$$E(a,b)/p = \{(x,y) \in \mathbb{Z}_p^2 \,|\, y^2 = x^3 + ax + b \pmod{p}\} \cup \{\infty\}.$$

Si può verificare che $(E(a,b)/p, \oplus)$ è un gruppo abeliano; tale gruppo è detto *gruppo ellittico* modulo p.

Facciamo un esempio semplice. Poniamo $a = b = 16$ e $p = 17$. Abbiamo quindi a che fare con la curva ellittica $y^2 = x^3 + 16x + 16 \pmod{17}$ ed il gruppo ellittico $(E(16,16)/17, \oplus)$. Le immagini di 0, ad esempio, sono $y = \pm\sqrt{16} = \pm 4$, ossia $y = 4$ e $y = -4 \mod 17 = 13$. Otteniamo quindi i punti $P_1(0,4)$ e $P_1^{-1}(0,13)$, che sono uno l'inverso dell'altro: infatti $x_1 = x_2$ e $y_1 = 4 \equiv -13 = -y_2 \pmod{17}$.

Analogamente, per $x = 14$, si ha

$$y^2 = 2744 + 224 + 16 = 2984 \equiv 9 \pmod{17},$$

$$y = \pm\sqrt{9} = \pm 3 \pmod{17},$$

$$y = 3 \vee y = 14,$$

ossia altri due punti $P_2(14,3)$ e $P_2^{-1}(14,14)$.

Calcoliamo ora $P_1(0,4) \oplus P_2(14,3)$. Per calcolare m abbiamo bisogno dell'inverso modulo 17 di $x_1 - x_2$, cioé

$$x_1 - x_2 = -14 = 3 \pmod{17}.$$

Grazie al piccolo teorema di Fermat:

$$(x_1 - x_2)^{-1} = 3^{-1} \equiv 3^{17-2} \equiv 6 \pmod{17}.$$

Possiamo così calcolare m, x_3 e y_3:

$$m = (y_1 - y_2) \cdot (x_1 - x_2)^{-1} = 1 \cdot 6 = 6,$$

$$x_3 = m^2 - x_1 - x_2 = 36 - 14 \equiv 5 \pmod{17},$$

$$y_3 = m \cdot (x_3 - x_1) + y_1 = 0.$$

Quindi $P_1 \oplus P_2 = (x_3, -y_3) = (5, 0)$.

4 Implementazione delle curve ellittiche in Python

4.1 La classe `ECPoint`

Per poter calcolare facilmente nei gruppi delle curve ellittiche in Python abbiamo implementato la classe `ECPoint`, che rappresenta un punto del piano appartenente ad un gruppo ellittico. I gruppi ellittici possono essere rappresentati con le classi `EC` o `EC_modp`, che vedremo in dettaglio più avanti: ad esempio, `EC(-2, 1)` rappresenta la curva $y^2 = x^3 - 2x + 1$.

```
>>> e = EC(-2, 1) # y^2 = x^3 -2x + 1
>>> p = ECPoint(1, 0, e) # Il punto (1, 0) appartiene alla curva
    ellittica
>>> p
ECPoint(1.0, 0.0, EC(-2, 1))
>>> p2 = ECPoint(1, 6, e) # Il punto (1, 6) non appartiene alla curva
     ellittica
[...]
builtins.ValueError: Il punto (1.000000, 6.000000) non appartiene
    alla curva EC(-2, 1)
```

Inoltre, il metodo `EC.point` (`EC_modp.point`) permette di creare facilmente un punto appartenente ad un dato oggetto `EC` o `EC_modp`:

```
>>> e = EC(-2, 1)
>>> e.point(0)
ECPoint(0.0, 1.0, EC(-2, 1))
>>> e.point(0, -1)
ECPoint(0.0, -1.0, EC(-2, 1))
>>> e.point(0, e(0)[1])
ECPoint(0.0, -1.0, EC(-2, 1))
```

Se viene fornito un solo parametro, il metodo `point` userà sempre l'immagine positiva dell'ascissa fornita come parametro. Utilizzando i metodi `__add__` e `__mul__`, che fanno riferimento rispettivamente ai metodi `sum` e `mult` della classe `EC` (o `EC_modp`) di cui parleremo più avanti, possiamo comodamente sommare punti e moltiplicare punti per un intero, ad esempio:

```
>>> e = EC(-2, 1)
>>> p1 = e.point(1)
>>> p2 = e.point(0)
>>> p1 + p2
ECPoint(0.0, -1.0, EC(-2, 1))
>>> p2 * 2
ECPoint(1.0, 0.0, EC(-2, 1))
```

I metodi `__eq__` e `__ne__` ci permettono di confrontare due punti fra di loro:

```
>>> e1 = EC(-1, 0)
>>> e2 = EC(-2, 0)
>>> p1 = e1.point(0)
>>> p1
ECPoint(0.0, 0.0, EC(-1, 0))
```

```
>>> p2 = e2.point(0)
>>> p2
ECPoint(0.0, 0.0, EC(-2, 0))
>>> p1 == p2
True
>>> p1 != p2
False
```

I due punti confrontati non devono per forza appartenere alla stessa curva, basta che abbiano le medesime coordinate. È inoltre definito per le classi `EC` e `EC_modp` il metodo `__contains__`, che fa riferimento al metodo `P_in_EC` per permetterci di determinare se un punto appartiene ad una data curva:

```
>>> p1 in e1
True
>>> p1 in e2
True
>>> p1 not in e1
False
```

Infine, il metodo `__getitem__` ci permette di accedere alle coordinate del punto come se fosse una lista o una tupla:

```
>>> p3 = e1.point(3)
>>> p3
ECPoint(3.0, 4.89898, EC(-1, 0))
>>> p3[0]
Fraction(3, 1)
>>> p3[1]
Fraction(244949, 50000)
>>> float(p2[1])
4.89898
```

Notiamo che le coordinate sono memorizzate come frazioni (questo ha a che fare con il problema dell'approssimazione dei numeri decimali, di cui parleremo più avanti).

4.2 La classe `EC`: gruppo ellittico su $\mathbb{R}$

La classe `EC` rappresenta una curva ellittica su $\mathbb{R}$. Ci permette di creare curve ellittiche con diversi parametri a e b, valutare la funzione della curva in dati punti, calcolare somme di punti come nel gruppo ellittico, tracciare grafici della curva e persino rappresentazioni grafiche della somma di due punti.

Come in tutte le classi, il primo metodo che definiamo è `__init__`:

```
def __init__(self, a, b):
    if 4*a**3 + 27*b**2 == 0:
        raise ValueError("Parametri a e b invalidi: 4*a**3 + 27*b**2
            = 0!")

    self.a = a
    self.b = b
```

In realtà questo metodo diventerà più complicato a causa del problema dell'approssimazione e dei calcoli preliminari per la rappresentazione grafica, ma tutto questo lo vedremo in seguito.

Il primo metodo importante è `f`, che permette di valutare il/i valore/i della curva per una data ascissa.

```
def f(self, x):
    if not isinstance(x, Fraction):
        x = Fraction(x).limit_denominator()
    y2 = self.D(x)
    if y2 > 0:
        y = Fraction(sqrt(y2)).limit_denominator()
        # Radice quadrata, quindi due valori!
        return [y, -y]
    elif y2 == 0:
        # sqrt(0) = 0
        return Fraction(0)
    else:
        # La radice quarata di un numero negativo è indefinita
        return False

def D(self, x):
    # Questo metodo viene usato anche per il metodo di Newton (
        trattato in seguito), per questo è separato da self.f
    return Fraction(x**3 + self.a*x + self.b)

def __call__(self, x):
    return self.f(x)
```

Il metodo `__call__`, che fa riferimento a `f`, ci permette di 'chiamare' un oggetto `EC` come se fosse una funzione per valutarne il/i valore/i:

```
>>> e = EC(-2, 1)
>>> e.f(-1)
[Fraction(141421, 100000), Fraction(-141421, 100000)]
>>> e(-1)
[Fraction(141421, 100000), Fraction(-141421, 100000)]
>>> e(1)
Fraction(0, 1)
```

4.2.1 Somma di punti

Il metodo `EC.sum` permette di calcolare la somma di due punti appartenenti alla curva ellittica, come descritto nel capitolo 3.2.

```
def sum(self, p1, p2):
    # 0 é il punto all'infinito (l'elemento neutro)
    if p1 == 0:
        return p2
    elif p2 == 0:
        return p1
    elif (not self.P_in_EC(p1)) or (not self.P_in_EC(p2)):
        return False

```

```
        x1, y1 = p1[0], p1[1]
        x2, y2 = p2[0], p2[1]
        fx1 = Fraction(x1).limit_denominator()
        fx2 = Fraction(x2).limit_denominator()
        fy1 = Fraction(y1).limit_denominator()
        fy2 = Fraction(y2).limit_denominator()
        a = Fraction(self.a).limit_denominator()

        if fx1 == fx2 and fy1 == Fraction(-y2).limit_denominator():
            # retta verticale
            return 0
        elif fx1 == fx2:
            # retta tangente
            m = (3*x1**2 + float(self.a)) / (2*y1)
        else:
            m = (fy1 - fy2) / (fx1 - fx2)
        x3 = m**2 - fx1 - fx2
        y3 = m*(x3 - fx1) + fy1
        x3 = Fraction(x3).limit_denominator()
        y3 = Fraction(y3).limit_denominator()
        return ECPoint(x3, -y3, self)
```

Ho usato l'intero 0 per rappresentare il punto all'infinito (ossia l'elemento neutro della somma su curve ellittiche). Di conseguenza, se uno dei due punti è 0, ritorniamo semplicemente l'altro. Se questo non è il caso, ci assicuriamo dapprima che entrambi i punti appartengano alla curva ellittica, in seguito procediamo a svolgere i calcoli come esposto nel paragrafo 3.1. Infine, il risultato è ritornato come oggetto `ECPoint`.

4.2.2 Il problema dell'approssimazione

Come si può vedere dal metodo `sum`, tutti i calcoli sono effettuati usando la classe `Fraction`, ossia con le frazioni. Ciò per ovviare ad un problema classico del calcolo decimale con il computer: l'approssimazione dei `float` (i numeri decimali). I computer funzionano in base 2 (binario) e questo significa che, per esempio, il numero 0.125, che in frazioni decimali si può esprimere come $\frac{1}{10}+\frac{2}{100}+\frac{5}{1000}$, viene in realtà memorizzato dal computer nella sua forma binaria $(0.001)_2$ ossia $\frac{0}{2}+\frac{0}{4}+\frac{1}{8}$. In questo caso, la forma binaria corrisponde esattamente a quella decimale, come si può controllare anche in Python:

```
>>> "%.50f"%(1/8)
'0.12500000000000000000000000000000000000000000000000'
```

Il problema è che la maggior parte dei numeri decimali non può essere espressa in modo esatto da somme di frazioni i cui denominatori sono potenze di 2. Ad esempio:

```
>>> from decimal import Decimal
>>> Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Infatti 0.1 non è memorizzato come $\frac{1}{10}$, ma come $\frac{1}{16}+\frac{1}{32}+\frac{1}{256}+\frac{1}{512}+\frac{1}{4096}+\ldots$

```
>>> 0/2 + 0/4 + 0/8 + 1/16 + 1/32 + 0/64 + 0/128 + 1/256 + 1/512 +
    0/1024 + 0/2048 + 1/4096
0.099853515625
```

È un problema analogo a quello della rappresentazione decimale di $\frac{1}{3}$:

$$\begin{aligned}
\frac{3}{10} &= 0.3 \\
\frac{3}{10} + \frac{3}{100} &= 0.33 \\
\frac{3}{10} + \frac{3}{100} + \frac{3}{1000} &= 0.333 \\
\frac{3}{10} + \frac{3}{100} + \frac{3}{1000} + \frac{3}{10000} &= 0.3333
\end{aligned}$$

Indipendentemente da quanti termini si considerano, sarà sempre un'approssimazione.

La soluzione che ho adottato è quella di lavorare con le frazioni, quindi in $\mathbb{Q}$, con la classe `Fraction` di Python. Il metodo `Fraction.limit_denominator` ci permette di usare denominatori 'contenuti', e quindi di avere rappresentazioni esatte dei numeri decimali:

```
from fractions import Fraction
>>> a = Fraction(0.1)
>>> print(a)
3602879701896397/36028797018963968
>>> b = a.limit_denominator()
>>> print(b)
1/10
```

Purtroppo neanche questa soluzione è priva di problemi: ad esempio, $\sqrt{2}$ non è un numero razionale, e quindi non può essere rappresentato esattamente da una frazione. Inoltre, mentre tutte le altre operazioni fra oggetti `Fraction` danno risultati in forma di `Fraction`, la radice quadrata ritorna sempre un `float`, e quindi riemerge ancora un'approssimazione.

```
>>> sqrt(Fraction(9, 1))
3.0
>>> Fraction(9, 1)**0.5
3.0
```

Questa incertezza si presenta nella nostra classe `EC` soprattutto nel metodo di somma:

```
>>> e = EC(-2, 1)
>>> e.sum(e.point(-1), e.point(0))
[...]
builtins.ValueError: Il punto (1.171570, -0.514720) non appartiene
    alla curva EC(-2, 1)
```

Questo errore è causato nel metodo `EC.P_in_EC`, che viene chiamato durante l'inizializzazione di `ECPoint` per controllare se il punto appartiene alla curva ellittica.

Infatti `EC.P_in_EC` e `EC.sum` calcolano l'immagine diversamente: uno valuta semplicemente la funzione per quell'ascissa, l'altro invece usa la pendenza della retta passante per i primi due punti della somma. Nell'esempio precedente, `EC.P_in_EC` ottiene un risultato di -0.51471, che è diverso da quello di `EC.sum` (-0.51472), e quindi causa un errore:

```
>>> float(e(1.171570)[1])
-0.51471
>>> e.P_in_EC((1.171570, -0.514720))
False
```

La soluzione più semplice, a questo punto, è di modificare il metodo `EC.sum` in modo che valuti la y del risultato allo stesso modo di `EC.P_in_EC`:

```
[...]
x3 = m**2 - fx1 - fx2
y3 = m*(x3 - fx1) + fy1
x3 = Fraction(x3).limit_denominator()
#y3 = Fraction(y3).limit_denominator()
if y3 < 0:
    y3 = e.f(x3)[1]
elif y3 > 0:
    y3 = e.f(x3)[0]
# else y3 == 0
return ECPoint(x3, -y3, self)
```

4.2.3 Multipli di un punto

Dato che moltiplicare un punto P di una curva ellittica per l'intero n equivale alla somma $P \oplus P \oplus \cdots \oplus P$ con n termini, il modo più semplice in assoluto per svolgere questo calcolo in Python consiste nell'utilizzare un ciclo:

```
if not self.P_in_EC(p):
    return False
elif n == 0:
    return (0, 0)
elif n == 1:
    return p

curp = p

for _ in range(n-1):
    curp = e.sum(p, curp)

return curp
```

Questo metodo funziona benissimo per piccoli multipli:

```
>>> e = EC(-2, 1)
>>> p = e.point(3)
>>> e.mult(p, 4)
ECPoint(2.8171216729089736, -4.209861042837524, EC(-2, 1))
```

Ma dato che la 'semplice' operazione di somma è parecchio laboriosa, se dovessimo aver bisogno di multipli molto grandi, ad esempio $20000P$, questa funzione comincia ad impiegarci veramente troppo tempo (dato che deve fare 20000 operazioni di somma):

```
>>> from time import time
>>> s = time(); e.mult(p, 20000); print(time() - s, 'secondi')
ECPoint(1.7848311545154478, 1.765259187682675, EC(-2, 1))
35.90005302429199 secondi
```

Dobbiamo quindi cercare un modo più rapido per fare questi calcoli. Il metodo che utilizzeremo è in realtà molto simile a quello per le esponenziazioni modulari rapide, esposto nella sezione 2.2.3.

Come esempio, prendiamo un calcolo relativamente semplice: $23P$. Il numero 23 può essere scritto in forma binaria come $(10111)_2$, cioé

$$23P = (2^0 + 2^1 + 2^2 + 2^4)P = 2^0P \oplus 2^1P \oplus 2^2P \oplus 2^4P$$

Un'idea a questo punto potrebbe essere di calcolare $2P$, 2^2P, 2^3P e 2^4P per poi sommare le componenti necessarie: questo sarebbe già un miglioramento, dato che $2^2P = 2 \cdot 2P$, $2^3P = 2 \cdot 2^2P$ eccetera. Ma possiamo fare di meglio mettendo in evidenza qualche termine:

$$\begin{aligned}23P &= 2^0P \oplus 2^1P \oplus 2^2P \oplus 2^4P \\ &= P \oplus 2(P \oplus 2^1P \oplus 2^3P) = P \oplus 2(P \oplus 2(P \oplus 2^2P)) \\ &= P \oplus 2(P \oplus 2(P \oplus 2(2P))).\end{aligned}$$

Possiamo raggiungere questa forma con un procedimento iterativo come il seguente:

bit	multipli di P
1:	P
0:	$2P$
1:	$2(2P) \oplus P$
1:	$2(2(2P \oplus P) \oplus P) \oplus P$
1:	$2(2(2(2P) \oplus P) \oplus P) \oplus P$

Compiamo un'iterazione per ciascun bit che compone $23 = (10111)_2$, partendo dal primo a sinistra (che indica 2^4). A ciascuna iterazione raddoppiamo il risultato, inoltre aggiungiamo P se il bit corrispondente è pari a 1. In questo modo abbiamo drasticamente ridotto il numero di addizioni necessarie.

```
def mult(self, P, n):
        if not self.P_in_EC(P):
            return False
        elif n <= 0 or not isinstance(n, int):
            raise ValueError("Si puo' solo moltiplicare per un intero
                n > 0!")
        elif n == 1:
```

```
            return P

        a, b = float(self.a), float(self.b)

        # forma binaria di n
        binar = bin(n)[2:]
        # inizializzazione
        c = (float(P[0]), float(P[1]))
        for e in binar[1:]:
            m1 = (3*c[0]**2 + a)
            m2 = (2*c[1])
            M = (m1 / m2)
            xc = (M**2 - 2*c[0])
            yc = (M*(c[0] - xc) - c[1])
            c = (xc, yc) # c := 2c, raddoppio

            if e == '1':
                # c := c + P, somma
                m1 = (c[1] - P[1])
                m2 = (c[0] - P[0])
                M = (m1 / m2)
                xc = (M**2 - P[0] - c[0])
                yc = (M*(P[0] - xc) - P[1])
                c = (xc, yc)

        x = c[0]
        fx = self.f(x)
        if fx == 0:
            y = 0
        else:
            y = fx[0] * sgn(c[1])
        return ECPoint(x, y, self)
```

Listato 1: Moltiplicazione rapida in PYTHON.

Possiamo subito vedere che il tempo d'esecuzione è ridotto drasticamente:

```
>>> e = EC(-2, 1)
>>> p = e.point(3)
>>> from time import time
>>> s = time(); e.mult(p, 20000); print(time() - s, 'secondi')
ECPoint(1.7456544840945956, 1.6817371970685284, EC(-2, 1))
0.0019998550415039062 secondi
```

Notiamo pure che i risultati differiscono parecchio (ad esempio la x del risultato varia da 1.78 a 1.74): infatti, dato che vengono eseguite molte meno operazioni, anche l'errore dovuto ai float è ridotto.

4.3 Tracciare le curve ellittiche su $\mathbb{R}$

Al fine di tracciare le curve ellittiche con PYTHON abbiamo deciso di usare il modulo `matplotlib`, che offre un metodo semplice ed efficace per rappresentare funzioni matematiche. Ecco un esempio basilare:

```
import numpy as np
```

```
import matplotlib.pyplot as pl

def plot(self, xmin, xmax, numpts=500):
    """ Metodo della classe EC per tracciare la curva"""
    X = np.linspace(xmin, xmax, numpts, endpoint=True)
    Y1, Y2 = [], []
    k = 0

    for _ in range(0, X.size):
        y = self.f(X[k])
        if y == False:
            X = np.delete(X, k)
            k -= 1
        elif y == 0:
            Y1.append(y)
            Y2.append(y)
        else:
            Y1.append(y[0])
            Y2.append(y[1])
        k += 1

    pl.figure(figsize=(8,6), dpi=80)
    pl.subplot(1,1,1)

    pl.plot(X, Y1, color="blue", linewidth=1.0, linestyle="-")
    pl.plot(X, Y2, color="blue", linewidth=1.0, linestyle="-")

    pl.xlim(xmin,xmax)
    pl.ylim(xmin,xmax)

    pl.show()
```

Listato 2: Disegno basilare con PYTHON.

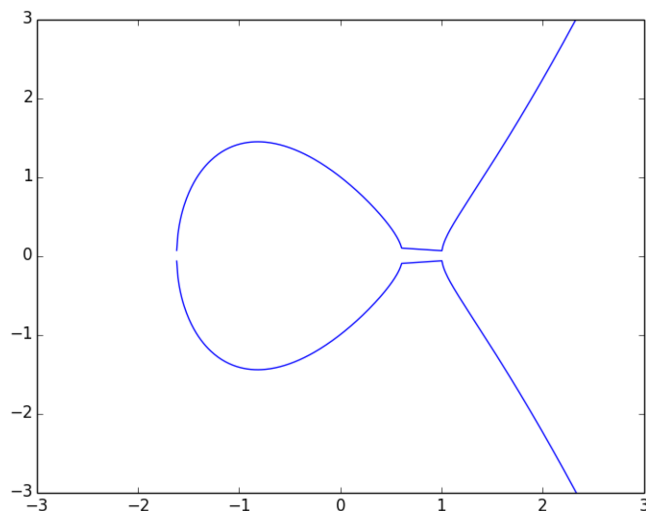


Figura 4: La curva $y^2 = x^3 - 2x + 1$ tracciata con PYTHON.

Il risultato è però insoddisfacente: il problema è che `matplotlib` si aspetta una funzione *continua*, mentre una curva ellittica non è nemmeno una funzione (per questo bisogna disegnare separatamente $y = \sqrt{x^3 + ax + b}$ e $y = -\sqrt{x^3 + ax + b}$, e queste non sono funzioni continue se c'è la 'bolla'). Inoltre, visto che l'intervallo dato viene suddiviso in intervalli di uguale ampiezza, non necessariamente verranno disegnati i punti di intersezione della curva ellittica con l'asse x, ossia gli zeri della funzione $y = \sqrt{x^3 + ax + b}$.

Entrambi questi problemi possono essere risolti facilmente: bisogna includere gli zeri nella lista di dati per chiudere gli spazi e, se è presente una bolla, tracciarla separatamente dal resto della funzione, in modo che `matplotlib` non disegni il collegamento.

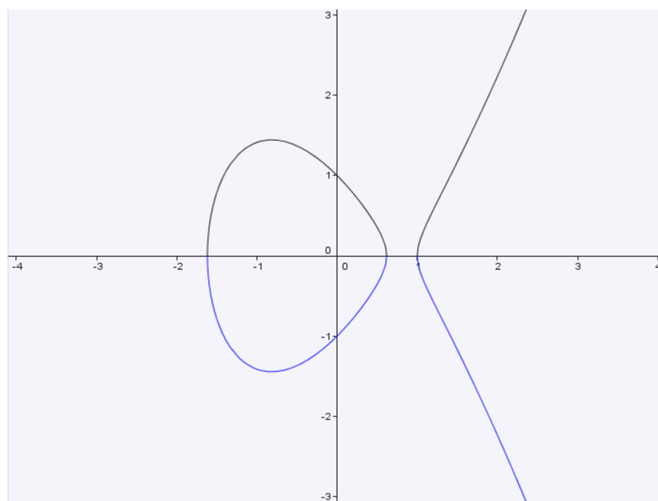


Figura 5: La curva $y^2 = x^3 - 2x + 1$ (GeoGebra).

4.3.1 Ottenere gli zeri

Usiamo il metodo algebrico classico, ponendo l'equazione della curva uguale a zero:

$$\sqrt{x^3 + ax + b} = 0,$$

ossia

$$x^3 + ax + b = 0.$$

A questo punto siamo però bloccati: bisogna ripiegare su un metodo approssimativo. Fortunatamente $\Delta(x) = x^3 + ax + b$ è una funzione polinomiale, dunque continua e (infinitamente) derivabile su tutto $\mathbb{R}$: basterà quindi utilizzare il metodo di Newton per trovare gli zeri di $\Delta(x)$ e dunque anche i punti di intersezione della curva ellittica con l'asse x.

4.3.2 Il metodo di Newton con Python

Ricordiamo che per trovare uno zero della funzione $f\colon [a, b] \longrightarrow \mathbb{R}$ con il metodo di Newton devono essere soddisfatte le ipotesi di Fourier:

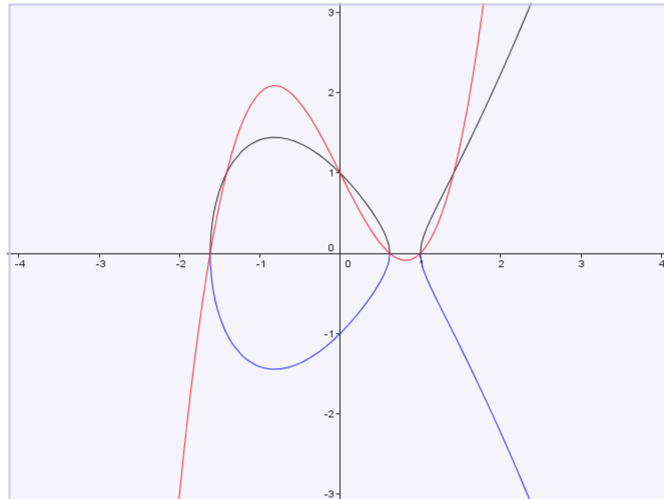


Figura 6: La curva $y^2 = x^3 - 2x + 1$ e, in rosso, il relativo $\Delta(x)$.

- f è derivabile due volte con derivate continue su $[a, b]$;
- $f(a) \cdot f(b) < 0$;
- f' ha segno costante non nullo in $[a, b]$;
- f'' ha segno costante in $[a, b]$).

Scelto $x_0 \in [a, b]$ qualsiasi, la *successione di Newton* definita da

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad (n \in \mathbb{N})}$$

convergerà verso uno zero di f. Fissata una tolleranza $\varepsilon > 0$ è ragionevole continuare i calcoli fintanto che

$$|x_{n+1} - x_n| > \varepsilon.$$

Definiamo quindi un metodo della nostra classe `EC` per approssimare uno zero di $\Delta(x)$, dato un x_0 iniziale.

```
1  def getZero(self, x0, e=0.001):
2      # Metodo di Newton per ottenere zero di una funzione
3      # self.D e' la funzione, self.D_1 e' la prima derivata
4      oldx, curx = x0, 0
5  
6      while True:
7          # Calcoliamo il prossimo valore della sucessione:
8          curx = oldx - self.D(oldx) / self.D_1(oldx)
9  
10         if abs(curx - oldx) < e:
11             # E' stato raggiunto il limite di tolleranza:
12             # curx e' il valore cercato
13             break
14 
15         oldx = curx
```

```
16
17      return curx
```

Listato 3: Il metodo di Newton con Python

4.3.3 Presenza della 'bolla'

Un'altra informazione importante per il nostro programma è la presenza o meno della bolla: se c'è, bisogna tracciarla separatamente dal resto della curva.

Osserviamo la Figura 6: dato che c'è la bolla, ci sono tre zeri; se non ci fosse, si avrebbe soltanto uno zero.

Possiamo quindi usare il numero di zeri come 'indicatore' della presenza o meno della bolla.

Per capire come dedurre il numero di zeri dobbiamo prima dare un'occhiata agli effetti dei parametri a e b sul polinomio $\Delta(x)$.

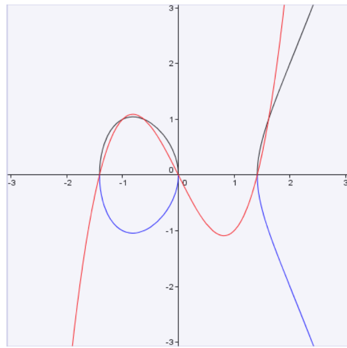


Figura 7: $y^2 = x^3 - 2x$.

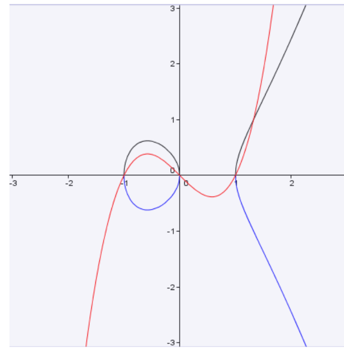


Figura 8: $y^2 = x^3 - x$.

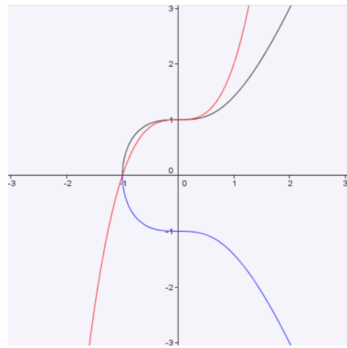


Figura 9: $y^2 = x^3 + 1$.

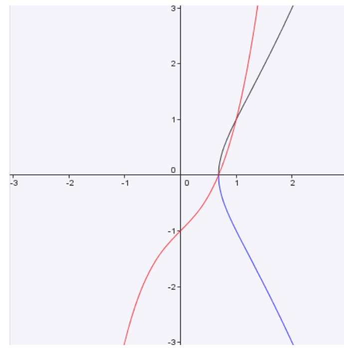


Figura 10: $y^2 = x^3 + x - 1$.

Notiamo che il parametro a influenza la *forma* della curva: la sua 'ampiezza' A e 'lunghezza d'onda' 2λ. Da notare che se $a \geq 0$ non vi è nessuna inflessione, e di conseguenza ci sarà un solo zero. Il parametro b, invece, è lo spostamento verticale della curva.

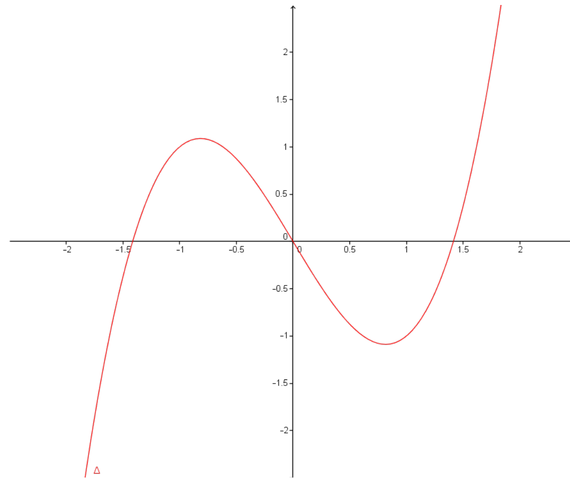


Figura 11: Un polinomio $\Delta(x)$ (GeoGebra).

Ricaviamo la 'lunghezza d'onda' calcolando le ascisse dei punti di inflessione, che corrispondono a $\pm\lambda$:

$$\Delta'(x) = 3x_\lambda^2 + a = 0,$$
$$x_\lambda = \pm\sqrt{-a/3},$$

da cui

$$\boxed{\lambda = \sqrt{-a/3}}$$

Infine calcoliamo l''ampiezza' A come l'immagine di λ a cui si sottrae b:

$$\boxed{A = |\lambda^3 + a\lambda|}$$

Per determinare il numero di zeri del polinomio $\Delta(x)$ e quindi la presenza della 'bolla' nella nostra curva ellittica dobbiamo tener conto di quanto segue:

- Se $a < 0$ e $b \in]-A; A[$: tre zeri, bolla presente.
- Se $a < 0$ e $b = +A$: due zeri, bolla a contatto con il corpo della curva.
- Se $a < 0$ e $b = -A$: due zeri, nessuna bolla.
- Altrimenti, uno zero solo e nessuna bolla.

4.3.4 Disegno della curva in Python

Ora che possiamo ricavare gli zeri necessari, procediamo a modificare la classe `EC` per tracciare adeguatamente la curva ellittica che rappresenta. Prima di tutto, in `__init__`, valutiamo il numero di zeri. Teniamo presente che se $b = -A$ ci sono due zeri di $\Delta(x)$, ma solo un punto di intersezione della curva ellittica con l'asse x (non c'è la bolla).

```python
def __init__(self, a, b, t=5):
    if not isinstance(a, Fraction):
        a = Fraction(a).limit_denominator()
    if not isinstance(b, Fraction):
        b = Fraction(b).limit_denominator()

    if round(4*a**3 + 27*b**2, t) == 0:
        raise ValueError("Parametri a e b invalidi: 4*a**3 + 27*b**2
            = 0!")

    self.a = a
    self.b = b
    self.zeros = None

    if a >= 0:
        self.hasBubble = False
        self.numZeros = 1
    else:
        self.wl = Fraction(sqrt(-a/3)).limit_denominator()
        wl = self.wl
        self.A = Fraction(abs(wl**3 + a*wl)).limit_denominator()

        if b < - self.A or b > self.A:
            self.hasBubble = False
            self.numZeros = 1
        elif b == self.A or b == - self.A:
            # Bolla 'attaccata' al corpo
            self.hasBubble = True
            self.numZeros = 2
        else:
            self.hasBubble = True
            self.numZeros = 3
```

In seguito definiamo i metodi `plot` e `findZeros`, che ci permetteranno di tracciare la curva.

```python
def plot(self, xmin, xmax, numpts=250):
    """Permette di plottare la curva ellittica"""
    # Se non sono ancora stati trovati gli zeri, trovali
    if self.zeros is None:
        self.findZeros()

    if not self.hasBubble:
        # Tracciatura continua, senza bolla
        self.contPlot(xmin, xmax, numpts)
    else:
        # Tracciatura con bolla separata
        self.bubblePlot(xmin, xmax, numpts)

def findZeros(self):
    """ Trova gli zeri della curva ellittica e mette i valori nella
        lista self.zeros"""

    if self.a >= 0:
        # Un solo zero: se b e' <= 0, esso si trovera' a destra dello
            zero, altrimenti a sinistra.
```

```
        if self.b > 0:
            self.zeros = [self.getZero(-1)]
        else:
            self.zeros = [self.getZero(1)]

    elif self.numZeros == 1 and self.b > self.A:
        # a < 0 e b > A
        # Lo zero sara' a sinistra del primo punto di inflessione
        self.zeros = [self.getZero(-self.wl - 1)]

    elif self.numZeros == 1:
        # a < 0 e b <= -A
        # Lo zero sara' a destra del secondo punto di inflessione
        self.zeros = [self.getZero(self.wl + 1)]

    elif self.numZeros == 3:
        # Tre zeri: uno a sinistra del primo punto di inflessione,
        # uno a destra del secondo punto di inflessione e il terzo
        # fra i due punti di inflessione
        self.zeros = [self.getZero(-self.wl - 1),
                      self.getZero(0),
                      self.getZero(self.wl + 1)]

    elif self.numZeros == 2:
        rightZero = self.getZero(self.wl + 1)
        self.zeros = [self.getZero(-self.wl - 1),
                      rightZero,
                      rightZero]

    else:
        return False

    return self.numZeros
```

I due metodi `contPlot` e `bubblePlot` usati da `plot` sono stati tralasciati per brevità, ma sono molto simili alla funzione basilare del listato 2. Possiamo ora facilmente disegnare una curva ellittica con Python, ad esempio:

```
>>> e = EC(-2, 1)
>>> e.plot(-3, 3)
```

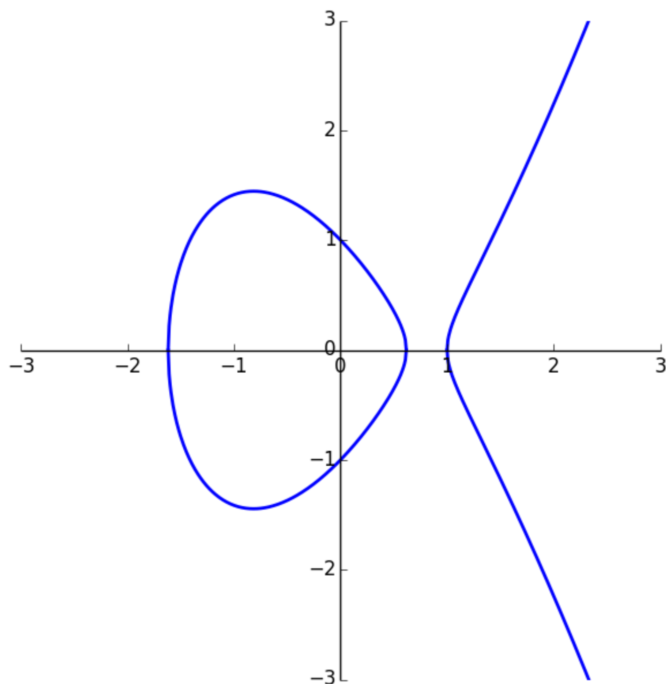


Figura 12: La curva $y^2 = x^3 - 2x + 1$ tracciata con PYTHON.

4.3.5 Disegno della somma di punti

Un altro metodo utile potrebbe essere la rappresentazione grafica della somma di due punti su una curva ellittica: è a questo che serve il metodo `EC.plotSum`:

```
1  def plotSum(self, p1, p2, xmin=-3, xmax=3, numpts=250):
2      accTypes = (tuple, list) # tipi di oggetto accettati oltre a
           ECPoint
3      tp1, tp2 = type(p1), type(p2)
4  
5      if tp1 == ECPoint:
6          if not p1 in self:
7              raise ValueError("Il primo punto non appartiene alla
                   curva {}".format(self))
8      elif tp1 in accTypes:
9          # se non è ECPoint ma un altro tipo accetato,
10         # creare un nuovo ECPoint
11         p1 = ECPoint(p1[0], p1[1], self)
12     else:
13         raise TypeError("I punti devono essere di tipo ECPoint, list
               o tuple!")
14 
15     if tp2 == ECPoint:
16         if not p2 in self:
17             raise ValueError("Il secondo punto non appartiene alla
```

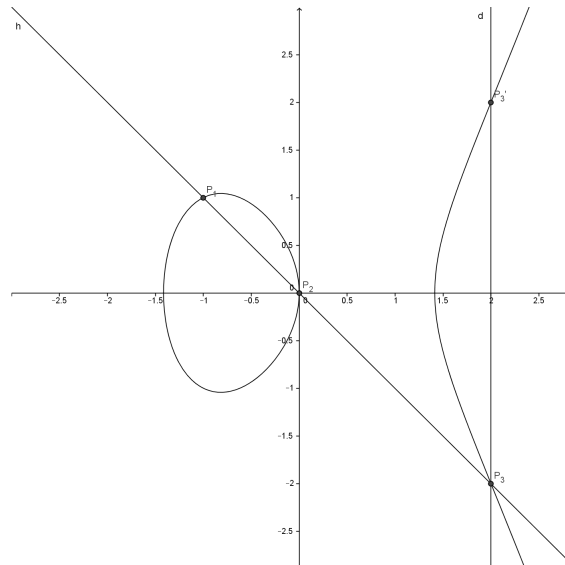


Figura 13: La somma di punti (GeoGebra).

```
                curva {}".format(self))
        elif tp2 in accTypes:
            p2 = ECPoint(p2[0], p2[1], self)
        else:
            raise TypeError("I punti devono essere di tipo ECPoint, list
                o tuple!")

        # calcolare il terzo punto d'intersezione
        p3 = self.sum(p1, p2)

        if self.zeros is None:
            self.findZeros()

        # tracciare la curva
        if not self.hasBubble:
            pl = self.contPlot(xmin, xmax, numpts)
        else:
            pl = self.bubblePlot(xmin, xmax, numpts)

        pl.xlim(xmin, xmax)
        pl.ylim(xmin, xmax)

        ax = pl.gca()
        ax.spines['right'].set_color('none')
        ax.spines['top'].set_color('none')
        ax.xaxis.set_ticks_position('bottom')
        ax.spines['bottom'].set_position(('data',0))
        ax.yaxis.set_ticks_position('left')
        ax.spines['left'].set_position(('data',0))

        # Tracciare i segmenti fra i punti
        pl.plot([p1[0], p2[0], p3[0]],
                [p1[1], p2[1], -p3[1]], color='black')
        pl.plot([p3[0], p3[0]],
                [p3[0], -p3[0]], color="black", aa=False)
```

```
    # Segnare i punti
    pl.plot([p1[0], p2[0], p3[0], p3[0]],
            [p1[1], p2[1], -p3[1], p3[1]], 'ro', zorder=1000)

    # Segnare le annotazioni dei punti
    D = (xmax-xmin)/100
    bbox = dict(color='white', alpha=0.75)
    pl.text(p1[0]+D, p1[1]+D, '$P_1$', bbox=bbox) # P1
    pl.text(p2[0]+D, p2[1]+D, '$P_2$', bbox=bbox) # P2
    pl.text(p3[0]+D, p3[1]+D, '$P_1 \oplus P_2$', bbox=bbox) # P1+P2
    pl.text(p3[0]+D, -p3[1]+D, '$P_3$', bbox=bbox) # P3

    pl.show() # visualizzare il grafico

    return p3 # ritorniamo la somma dei punti
```

Le righe 47–48 tracciano il segmento passante per P_1, P_2 e P_3, le righe 49–50 il segmento verticale fra P_3 ed il suo simmetrico lungo l'asse x (ossia il risultato della somma $P_1 \oplus P_2$) e le righe 52–53 segnano i punti effettivi. Le righe 58–61 aggiungono a quei punti le annotazioni (notiamo che `matplotlib` ci permette di usare stringhe con formattazione LaTeX: possiamo quindi scrivere `$P_1 \oplus P_2$` e verrà formattato nell'output come $P_1 \oplus P_2$). Infine, il metodo ritorna semplicemente il punto risultante dalla somma.

```
>>> e = EC(-2, 0)
>>> e.plotSum(e.point(-1), e.point(0))
ECPoint(2.0, 2.0, EC(-2, 0))
```

4.4 La classe `EC_modp`: gruppo ellittico su $\mathbb{Z}_p$

La classe `EC_modp` è molto simile alla classe `EC`, con la differenza che permette di creare un gruppo ellittico $(E(a, b)/p, \oplus)$ su $\mathbb{Z}_p$. Di conseguenza, tutti i calcoli sono effettuati con gli interi (il che significa che non abbiamo il problema dell'imprecisione dei decimali!) e non ci sono i metodi per tracciare le curve (poiché avremmo un insieme di punti sparsi).

In Python, l'operatore per il modulo è `%`.

```
def sum(self, p1, p2):
    if not (self.P_in_EC(p1) and self.P_in_EC(p2)):
        return False

    x1, y1 = p1[0], p1[1]
    x2, y2 = p2[0], p2[1]

    a, b, p = self.getParams()
    if (x1 % p) == (x2 % p) and (y1 % p) == (-y2 % p):
        return 0
    elif p1 == p2:
        m = (3*x1**2 + a) * inv_modp(2*y1, p)
    else:
        m = (y2 - y1) * inv_modp(x2 - x1, p)

    m = m % p
```

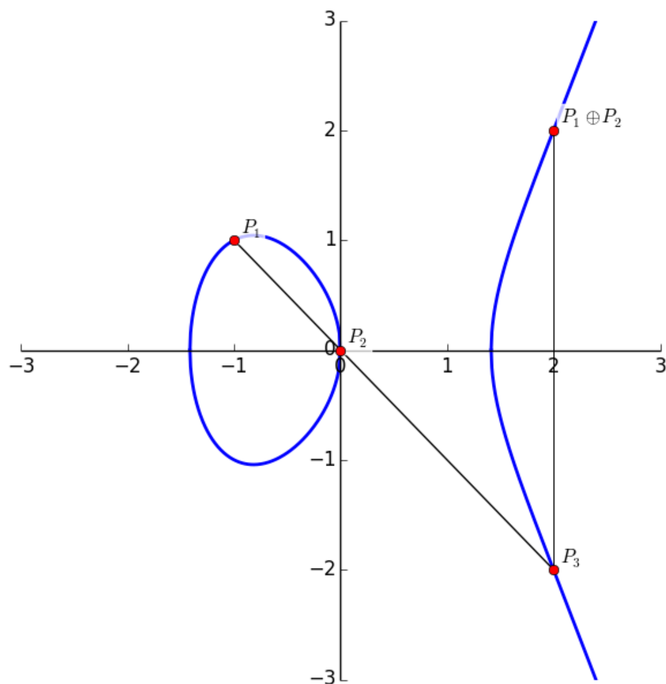


Figura 14: La somma di punti in PYTHON.

```
17        x3 = m**2 - x1 - x2
18        return ECPoint(x3 % p, (m*(x1 - x3) - y1) % p, self)
```

Vale la pena sottolinare il fatto che, operando in modulo p, invece di dividere moltiplichiamo per l'inverso in modulo p: per questo alle righe 12 e 14 usiamo la funzione `inv_modp`, definita così:

```
1  def inv_modp(b, p):
2      """inv(b, p): Calcolare l'inverso moltiplicativo di b in un
           gruppo modulo p (dove p e' primo)"""
3      # Grazie al Piccolo teorema di Fermat:
4      return (b**(p-2)) % p
```

Questa funzione dà per scontato che p sia un numero primo per poter usare il teorema 2.1 (piccolo teorema di Fermat).

Facciamo un esempio pratico. Per rappresentare la curva $y^2 = x^3 - x - 1$ su $\mathbb{Z}_{17}$, che corrisponde al gruppo ellittico $(E(16, 16)/17, \oplus)$ $(-1 \mod 17 = 16)$, possiamo usare la classe `EC_modp` come segue:

```
1  >>> e = EC_modp(-1, -1, 17)
2  >>> print(e)
3  EC(16, 16)/17
```

Le immagini di 0, ad esempio, sono $y = 4$ e $y = -4 \equiv 13 \pmod{17}$:

```
>>> e(0)
[4, 13]
>>> e.f(0)
[4, 13]
```

Come possiamo vedere, 'chiamare' direttamente l'oggetto `EC_modp` o utilizzare il metodo `EC_modp.f` è equivalente. Creiamo ora alcuni oggetti `ECPoint`, che rappresentano punti appartenenti alla nostra curva ellittica:

```
>>> e(1)
[4, 13]
>>> p1 = ECPoint(1, 13, e)
ECPoint(14, 3, EC(16, 16)/17)
>>> p2 = e.point(0, 13)
ECPoint(0, 13, EC(16, 16)/17)
>>> p3 = e.point(0)
ECPoint(0, 4, EC(16, 16)/17)
```

Disponiamo di tre modi diversi per creare un oggetto `ECPoint`. Possiamo utilizzare direttamente la classe `ECPoint` passando come argomenti le coordinate del punto e la curva ellittica a cui appartiene (`e`, nel nostro caso). In alternativa, possiamo usare il metodo `.point` della curva stessa, passandogli le coordinate, e questo creerà un punto sulla curva cui appartiene. È possibile in questo caso omettere il valore y: verrà automaticamente calcolato e, se vi sono due immagini, verrà utilizzata la prima (quella positiva).

Naturalmente, se non vi sono immagini per la x data, ci sarà un errore:

```
>>> e(2)
False
>>> e.point(2)
[...]
builtins.ValueError: La curva EC(16, 16)/17 è indefinita per x = 2!
```

Abbiamo ora tre punti $P_1(1, 13)$, $P_2(0, 13)$ e $P_3(0, 4)$ sulla curva ellittica `e`. P_2 e P_3 sono inversi ($x_2 = x_3$ e $y_2 = -y_3$), quindi $P_2 \oplus P_3 = \infty$, che il programma indica con l'intero 0:

```
>>> p2 + p3
0
>>> p1 + p2
ECPoint(16, 4, EC(16, 16)/17)
```

Infine possiamo 'moltiplicare' i punti per degli interi:

```
>>> p1 + p1
ECPoint(14, 3, EC(16, 16)/17)
>>> p1*2
ECPoint(14, 3, EC(16, 16)/17)
>>> 3*p1
0
```

5 Bibliografia e sitografia

[1] SONG Y. YAN. *Number Theory for Computing (2nd edition)*. Springer, 2010.

[2] DAVID M. BRESSOUD. *Factorization and Primality Testing*. Springer, 1989.

[3] `http://docs.python.org/3.3/`. Documentazione *online* di PYTHON.