Università degli Studi di Padova
Corso di Laurea Triennale in Ingegneria Informatica

Metodo dell'Elissoide in Programmazione Lineare

Relatore: prof. Matteo Fischetti
Studente: Nicola Zago
Matricola: 539993
Anno Accademico: 2007/2008

Indice

Introduzione

In Ricerca Operativa, la Teoria della Complessità Computazionale svolge un ruolo importante nel tentativo di comprendere il livello di difficoltà intrinseco dei problemi. Uno dei capisaldi di questa materia è la divisione dei problemi nelle classi $\mathcal{P}$ e $\mathcal{NP}$.
A grandi linee la classe $\mathcal{P}$ comprende i problemi che anche nel caso peggiore (nel *worst case*) possono essere risolti tramite un algoritmo con un numero di operazioni polinomialmente proporzionale alla dimensione del problema stesso. La classe $\mathcal{NP}$ invece comprende quei problemi che, nonostante non sia necessariamente noto un algoritmo polinomiale per risolverli, è possibile comunque verificare in tempo polinomiale nelle dimensioni del problema se una soluzione fornita è corretta.
Il tempo di esecuzione polinomiale è stato proposto come criterio per caratterizzare l'efficienza degli algoritmi, anche se per polinomi di grado elevato questa rimane solo teorica.
Inoltre, si può aggiungere che non è detto che gli algoritmi non polinomiali siano inefficienti in pratica; ci possono essere algoritmi che si rivelano non efficienti sono in casi particolari o patologici. Ciononostante, i problemi risolvibili solo con algoritmi di questo tipo sono da considerare più difficili dei problemi della classe $\mathcal{P}$, in quanto alcune loro istanze hanno tempi di risoluzione proibitivi anche se caratterizzate da dati apparentemente contenuti.

Un esempio pratico di questa situazione si può vedere nella Programmazione Lineare, uno dei più importanti campi di studio della Ricerca Operativa. Fino ad alcuni decenni fa, per la risoluzione dei problemi di Programmazione Lineare esistevano solo algoritmi con un'ottima efficienza pratica, che però non presentavano un limite polinomiale nel caso peggiore, come per esempio il famoso algoritmo del simplesso. Per questo la Programmazione Lineare andava considerata appartenere alla classe $\mathcal{NP}$.
Tuttavia, nel febbraio 1979, il matematico russo Leonid Khachiyan dimostrò che apportando opportune modifiche all'algoritmo dell'elissoide, nato originariamente per l'ottimizzazione convessa, e applicandolo a problemi di programmazione lineare, erano garantite prestazioni polinomiali in ogni caso. Studi successivi mostrarono che in realtà questo algoritmo ha un'efficienza più teorica che pratica, e nelle applicazioni reali si rivelano più efficienti gli algoritmi tradizionali. Quindi questo è un ottimo esempio di come gli algoritmi polinomiali non sono

necessariamente migliori a priori degli algoritmi non polinomiali, dato che i vantaggi potrebbero essere apprezzabili solo per istanze molto grandi o addirittura solo per istanze patologiche.

Un'interessante curiosità sull'algoritmo dell'elissoide è il modo in cui è stato accolto dall'opinione pubblica. Alla fine degli anni Settanta era ben noto come la Ricerca Operativa fosse applicata con successo in attività decisionali di governi e industrie; inoltre era anche già famosa la Teoria della Complessità Computazionale e i suoi concetti principali, come l'efficenza degli algoritmi polinomiali.
Dopo la comparsa dell'articolo di Khachiyan, la scoperta fu divulgata da alcune riviste scientifiche, in un modo a volte fraintendibile. Infatti molti giornalisti di quotidiani ne travisarono completamente il significato, descrivendo questo nuovo algoritmo polinomiale come un punto di rottura con i metodi di risoluzione del passato, che si sarebbe tradotto in miglioramenti nei vari campi in cui si utilizzava la Ricerca Operativa. Alcuni giunsero addirittura a trarre conclusioni riguardanti la possibilità di creare nuovi algoritmi efficienti per famosi problemi come quello del commesso viaggiatore, tanto che un lettore informato avrebbe potuto chiedersi se si riferissero veramente all'articolo di Khachiyan. Nonostante queste malcomprensioni e al fatto che l'algoritmo non sia di utilità pratica, rimane il fatto che esso sia molto importante, in quanto rappresenta la prova che i problemi di Programmazione Lineare appartengono alla classe $\mathcal{P}$.

Questa tesina ha lo scopo di descrivere il metodo dell'elissoide, mostrando quali siano le sue principali caratteristiche e gli approcci per sfruttarlo in applicazione pratiche, mettendone in luce anche i problemi. Più in dettaglio, si tratterà nel primo Capitolo l'algoritmo di base, usato per trovare in tempo polinomiale una soluzione per un sistema di disequazioni lineari. Nel Capitolo 2 saranno presentati alcuni accorgimenti per accelerarne la convergenza, per poi mostrare nel Capitolo 3 come si può impiegare per creare algoritmi polinomiali per la Programmazione Lineare. Infine nel Capitolo 4 si commenterà un'implementazione Java dell'algoritmo per la soluzione di sistemi di disequazioni, i cui sorgenti saranno disponibili nell'Appendice. Durante l'esposizione e soprattutto nella Bibliografia si farà riferimento ai vari testi consigliati per approfondire gli argomenti.
Molti concetti di base della Ricerca Operativa, come le varie nozioni di programmazione convessa e lineare, saranno dati per scontati, si consiglia quindi la consultazione di un qualsiasi testo riguardante l'argomento in caso il lettore manchi di queste conoscenze, per esempio [8].

Capitolo 1

Metodo dell'Elissoide per sistemi di disequazioni lineari

In questo capitolo si mostrerà il metodo dell'elissoide adattato per dimostrare la determinatezza di un sistema di disequazioni lineare, cioè l'algoritmo troverà una soluzione $x \in \Re^n$, $x \geq 0$, o dimostrerà che non ne esistono, per m vincoli del tipo:

$$a_i^T x \leq \beta_i \qquad i = 1, 2, \ldots, m$$

Questi possono anche essere scritti con la notazione più compatta:

$$A^T x \leq b \tag{1.1}$$

dove A^T è una matrice $m \times n$ e $b \in \Re^m$.
Si forniranno poi gli elementi che permettono di concludere che l'algoritmo è polinomiale. Prima di introdurre il metodo è opportuno definire brevemente alcuni concetti che saranno usati durante la rimanente esposizione.

Definizione 1 *Si dice* sfera euclidea *(Euclidean ball), o semplicemente sfera, il sottoinsieme di* $\Re^n$*:*

$$B(x_0, r) = \{x \in \Re^n : \sqrt{\|x - x_0\|} \leqslant r\} = \{x \in \Re^n : (x - x_0)^T (x - x_0) \leqslant r^2\}$$

dove il vettore $x_0 \in \Re^n$ *è detto* centro *della sfera e* $r \in \Re$, $r \geq 0$ *è detto* raggio. $B(x_0, r)$ *consiste di tutti i punti con distanza minore o uguale a* r *dal centro* x_0 *e si può dimostrare che è un insieme convesso (si veda [6]).*
La sfera euclidea in $\Re^2$ *corrisponde a un cerchio di raggio* r*, mentre in* $\Re^3$ *corrisponde a una sfera sempre di raggio* r*.*

Definizione 2 *Si dice* elissoide *(Ellipsoid) il sottoinsieme di* $\Re^n$*:*

$$E(x_0, B) = \{x \in \Re^n : (x - x_0)^T B^{-1} (x - x_0) \leqslant 1\} \tag{1.2}$$

dove il vettore $x_0 \in \Re^n$ è detto centro *dell'elissoide e B è una matrice simmetrica e positivamente definita (*positive definite[1]*) che definisce fino a che distanza dal centro si estende l'elissoide. Anche questa è, come la sfera euclidea, un insieme convesso.*
Sarà utile conoscere in seguito il volume in $\Re^n$ di un'elissoide $E(x_0, B)$. Esso è dato da:

$$vol(E(x_0, B)) = vol(E(0, I)) \times \sqrt{\det(B)} \tag{1.3}$$

dove $E(0, I)$ è la sfera unitaria di $\Re^n$. La sfera è un caso particolare di elissoide con $B = r^2 I$. In $\Re^2$ l'elissoide corrisponde ad un'ellisse.

1.1 Algoritmo di base

Il metodo dell'elissoide considera un'elissoide iniziale E_0, che si suppone contenga l'insieme delle soluzioni di (1.1), $P = \{x \in \Re^n : A^T x \leq b, x \geq 0\}$, e da questa genera una sequenza di elissoidi E_0, E_1, ... , E_k, ... All'iterazione $(k+1)$ considera il centro x_k dell'elissoide corrente E_k e verifica se $x_k \in P$.
In caso positivo l'algoritmo ha trovato una soluzione di (1.1), quindi termina presentando come soluzione x_k. Altrimenti considera uno dei vincoli violati da x_k, per esempio:

$$a^T x \leq \beta \tag{1.4}$$

e genera un taglio parallelo a questo e passante per x_k che individua la semi-elissoide contenente la soluzione:

$$\{x \in E_k : a^T x \leq a^T x_k\} \tag{1.5}$$

La semi-elissoide viene racchiusa in una nuova elissoide E_{k+1}, con centro x_{k+1}, che viene usata per ripetere la verifica precedente. Nella figura (1.1) si vede un esempio in $\Re^2$, dove il centro di E_k viola un vincolo e viene generata una nuova ellisse contenente la semicirconferenza individuata da (1.5). Per creare la nuova elissoide si usano le seguenti trasformazioni:

$$x_{k+1} = x_k - \tau(B_k a / \sqrt{a^T B_k a}) \tag{1.6}$$

$$B_{k+1} = \delta\Big(B_k - \sigma(B_k a a^T B_k)/(a^T B_k a)\Big) \tag{1.7}$$

dove

$$\tau = 1/(n+1) \qquad \sigma = 2/(n+1) \qquad \delta = n^2/(n^2-1) \tag{1.8}$$

Queste generano $E_{k+1} = \mathrm{E}(x_{k+1},\ B_{k+1})$, che è l'elissoide di volume minimo contenente la semi-elissoide (1.5). Per una dimostrazione di ciò si veda l'Appendice B di [2]. Questo costituisce il nucleo dell'algoritmo e grazie ad alcuni accorgimenti può essere terminato in un numero polinomiale di passi.

[1] Una matrice M si dice positivamente definita se $x^T M x \geq 0 \quad \forall x \in \Re^n$

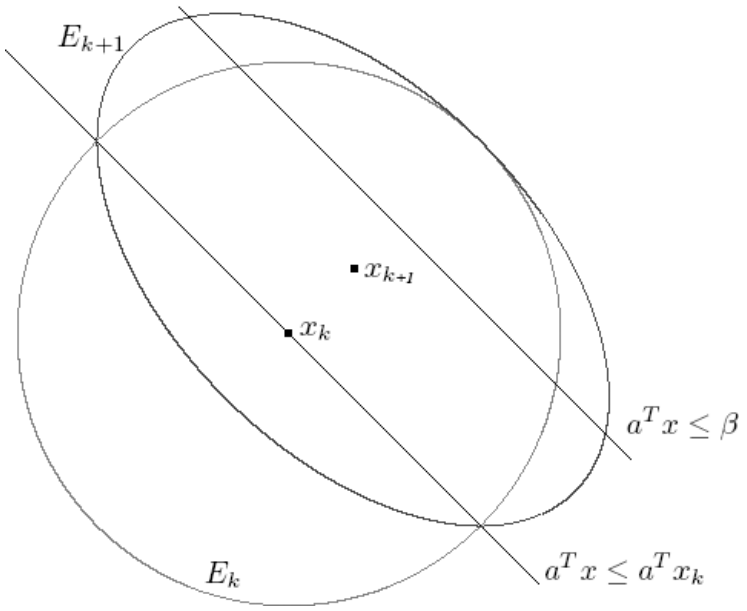


Figura 1.1: Esempio di un'iterazione dell'algoritmo

1.2 Polimonialità dell'algoritmo

Questa sezione ha lo scopo di esporre le argomentazioni che consentono di dimostrare la polinomialità dell'algoritmo dell'elissoide, evidenziando i concetti fondamentali, ma rimandando a letture più approfondite per una dimostrazione completa.

1.2.1 Dimensioni del problema e precisione richiesta

Per dimostrare la polinomialità di un algoritmo bisogna innanzitutto considerare le dimensioni di una generica istanza del problema da risolvere. Parlando di un sistema di m disequazioni lineari a n incognite $A^T x \leq b, x \geq 0$, si ha la necessità di memorizzare gli interi n e m, gli elementi a_{ij} di A e gli elementi β_i di b. Un intero positivo p può essere salvato in $1 + \lfloor \log p \rfloor$ bit, o in 1 bit se è uguale a zero, dove il logaritmo è in base 2 e $\lfloor x \rfloor$ indica il maggior numero intero minore o uguale ad x. Inoltre ogni elemento ha bisogno di un bit aggiuntivo per il segno.

Si ha quindi che il sistema ha la seguente dimensione in bit:

$$\begin{aligned} L &= (2 + \lfloor \log n \rfloor) + (2 + \lfloor \log m \rfloor) \\ &\quad + (2mn + \sum_{1 \le i \le m, 1 \le j \le n, a_{ij} \neq 0} \lfloor \log |a_{ij}| \rfloor) \\ &\quad + (2m + \sum_{1 \le i \le m, \beta_i \neq 0} \lfloor \log |\beta_i| \rfloor) \end{aligned} \tag{1.9}$$

Bisogna dimostrare che l'algoritmo converge in un numero di passi polinomiale in L.

Le formule (1.6)-(1.8) implicano aritmetica esatta, ma nei calcolatori si deve usare aritmetica a precisione finita. Khachiyan (vedi [1]) indicò che $23L$ bit di precisione prima della virgola e $38nL$ bit dopo la virgola sono sufficienti, a patto di moltiplicare B_{k+1} per un fattore leggermente maggiore di uno, altrimenti non è detto che E_{k+1} contenga la semi-elissoide desiderata. Khachiyan usò il fattore $2^{1/4n^2}$; altri studiosi come Grötschel (vedi [3]) anzichè usare un fattore moltiplicativo, hanno variato il valore di δ nella (1.8) in $\delta = (n^2+3)/n^2$ per ottenere lo stesso effetto. Ad ogni modo, il concetto importante consiste nel fatto che è sufficiente considerare numeri con dimensione polinomiale in L.

1.2.2 Rapporto tra elissoidi consecutive

Si cercherà ora di mostrare quanto diminuisce il volume dell'elissoide contenente l'insieme delle soluzioni durante un'iterazione. Innanzitutto si noti che una generica elissoide $E(x_k, B)$ in $\Re^n$ può essere ottenuta dalla sfera unitaria di $\Re^n$ $E(0, I)$, tramite una trasformazione $y = Px$, dove $B = P^T P$, e una successiva traslazione di vettore x_k. Infatti:

$$\begin{aligned} x^T x &\le 1 \\ y^T (P^{-1})^T P^{-1} y &= x^T x \qquad \text{dove} \quad x = P^{-1} y \\ y^T (P^{-1})^T P^{-1} y &\le 1 \\ y^T B^{-1} y &\le 1 \end{aligned}$$

dove la prima equazione è quella della sfera unitaria e l'ultima una generica elissoide centrata nell'origine. Una trasformazione di questo tipo genera da una sfera unitaria un'elissoide di $\det(P)$ volte il volume originario; in generale ogni insieme viene scalato di un fattore $\det(P)$, inoltre vale $\det(P) = \sqrt{\det(B)}$. La traslazione non fa invece variare il volume degli insiemi coinvolti.

Ora si considerino le formule (1.6)-(1.8) nel caso particolare in cui l'elissoide E_k sia la sfera unitaria $E(z, D)$, quindi z è l'origine e $D = I$. Esse divengono:

$$\begin{aligned} z' &= z - \tau \left(\frac{Ia}{\sqrt{a^T I a}} \right) = -\frac{1}{n+1} \left(\frac{a}{\sqrt{a^T a}} \right) \\ D' &= \delta \left(I - \sigma \frac{I a a^T I}{a^T I a} \right) = \frac{n^2}{n^2 - 1} \left(I - \frac{2}{n+1} \frac{a a^T}{a^T a} \right) \end{aligned} \tag{1.10}$$

Dal momento che si sta lavorando con una sfera, anche se si ruota il vettore a, che determina la direzione del taglio, le dimensioni della nuova elissoide che conterrà la semisfera non cambieranno, così per comodità di calcolo si consideri $a = [1, 0, \ldots, 0]^T$. Sviluppando i calcoli si ottiene:

$$\begin{aligned} z' &= \left[\frac{-1}{n+1}, 0, \ldots, 0\right]^T \\ D' &= \frac{n^2}{n^2-1}\left(I - \frac{2}{n+1}\begin{bmatrix} 1 & & & & \\ & 0 & & & \\ & & 0 & & \\ & & & \ddots & \\ & & & & 0 \end{bmatrix}\right) \\ &= \frac{n^2}{n^2-1}\begin{bmatrix} 1-\frac{2}{n+1} & 0 & 0 & \ldots & 0 \\ 0 & 1 & & & \\ 0 & & 1 & & \\ \vdots & & & \ddots & \\ 0 & & & & 1 \end{bmatrix} \end{aligned} \tag{1.11}$$

Dopo le precedenti osservazioni e ricordando la (1.3), nell'algoritmo dell'elissoide il rapporto di due elissoidi consecutive E_k e E_{k+1} è quindi:

$$\frac{vol\big(E_{k+1}\big)}{vol\big(E_k\big)} = \frac{vol\big(E(x_{k+1}, B_{k+1})\big)}{vol\big(E(x_k, B_k)\big)} = \frac{vol\big(E(0, B_{k+1})\big)}{vol\big(E(0, B_k)\big)} = \frac{vol\big(E(0, D')\big) \times \sqrt{\det(B)}}{vol\big(E(0, D)\big) \times \sqrt{\det(B)}} = \frac{vol\big(E(0, I)\big) \times \sqrt{\det(D')}}{vol\big(E(0, I)\big) \times \sqrt{\det(D)}}$$

Dal momento che $D = I$, $\det(D) = 1$, e considerando la (1.11) si ha:

$$\begin{aligned} \frac{vol\big(E_{k+1}\big)}{vol\big(E_k\big)} &= \sqrt{\det D'} = \sqrt{\left(\frac{n^2}{n^2-1}\right)^n \left(1 - \frac{2}{n+1}\right)} \\ &= \left(\frac{n^2}{n^2-1}\right)^{\frac{n}{2}} \left(\frac{n-1}{n+1}\right)^{\frac{1}{2}} = \left(\frac{n^2}{n^2-1}\right)^{\frac{n-1}{2}} \frac{n}{\sqrt{(n+1)(n-1)}} \frac{(n-1)^{1/2}}{(n+1)^{1/2}} \\ &= \left(1 + \frac{1}{n^2-1}\right)^{\frac{n-1}{2}} \left(1 - \frac{1}{n+1}\right) \leq e^{\frac{1}{(n+1)(n-1)}\frac{n-1}{2}} e^{-\frac{1}{n+1}} = e^{-\frac{1}{2(n+1)}} \end{aligned}$$

dove nell'ultimo passaggio si è usato $e^x > 1 + x$. Si è ottenuta in questo modo l'importante disuguaglianza:

$$\frac{vol\big(E_{k+1}\big)}{vol\big(E_k\big)} \leq e^{-\frac{1}{2(n+1)}} \tag{1.12}$$

Da questa si può notare che il rapporto massimo tra volumi di elissoidi successive aumenta al crescere di n, cioè l'efficienza dell'algoritmo degrada al crescere del numero di incognite.

1.2.3 Assunzioni e numero di passi

Per procedere con la dimostrazione si fanno ora due assunzioni, che verranno giustificate nella prossima sezione. Si suppone che:

1. L'insieme delle soluzioni P del sistema (1.1) sia limitato, esiste quindi una sfera centrata nell'origine che lo contiene completamente. Sia questa $B(0, R)$, allora vale $B(0, R) \supseteq P$.
2. L'insieme P contenga una sfera $B(x_r, r)$; questo garantisce che P abbia lo stesso numero di dimensioni di $B(0, R)$, altrimenti potrebbe anche avere un numero di dimensioni minore (come una retta in $\Re^2$ o una superficie in $\Re^3$), e quindi avrebbe volume nullo in $\Re^n$.

Sotto la validità di queste, si ha:

$$B(0, R) \supseteq P \supseteq B(x_r, r) \tag{1.13}$$

Si noti che l'elissoide E_k al passo k deve avere volume maggiore rispetto a $B(x_r, r)$, altrimenti non potrebbe contenerlo. Al passaggio finale, quando nel caso peggiore E_k diventa $B(x_r, r)$, x_k, che coinciderebbe con x_r, appartiene a P.
Sia $E_0 = B(0, R)$ l'elissoide iniziale dell'algoritmo. Si cercherà ora di stabilire quanti passi sono necessari per passare dall'elissoide iniziale E_0 a quella finale E_k, sottostimando il volume di questa con la sfera $B(x_r, r)$ in essa contenuta.

$$\begin{aligned}
\frac{vol(E_k)}{vol(E_0)} &\leq e^{-\frac{k}{2(n+1)}} \\
\frac{vol\big(B(x_r, r)\big)}{vol\big(B(0, R)\big)} = \frac{vol\big(E(x_r, r^2 I)\big)}{vol\big(E(0, R^2 I)\big)} &= \frac{vol\big(E(0, I)\big) \times \sqrt{\det\,(r^2 I)}}{vol\big(E(0, I)\big) \times \sqrt{\det\,(R^2 I)}} = \\
&= \left(\frac{r}{R}\right)^n \leq e^{-\frac{k}{2(n+1)}}
\end{aligned}$$

Da questa si ricava:

$$\begin{aligned}
\left(\frac{R}{r}\right)^n &\geq e^{\frac{k}{2(n+1)}} \\
n \log \frac{R}{r} &\geq \frac{k}{2(n+1)} \\
k &\leq 2n(n+1) \log \frac{R}{r}
\end{aligned} \tag{1.14}$$

Quindi si ha che se $\log(R/r)$ è polinomiale in L, il numero di iterazioni per arrivare da E_0 a E_k è ancora polinomiale in L. Se si superano k passi significa che il sistema di disequazioni considerato è impossibile (in quanto E_k non può contenere la sfera $B(x_r, r) \subseteq P$).

1.2.4 Conclusione

Si vuole ora mostrare come sia possibile rendere vere le assunzioni fatte nella sezione precedente.

Spesso nelle applicazioni pratiche sono disponibili dei limiti noti a priori per i valori delle soluzioni, e potrebbero essere usati per inizializzare l'algoritmo. Nel caso non ve ne siano, Khachiyan ha dimostrato che se $P \neq \emptyset$ allora $P \cap S(0, 2^L) \neq \emptyset$, quindi si può porre $E_0 = B(0, 2^L)$. Si noti che la prima assunzione era più stringente del necessario, è sufficente che E_0 contenga anche solo una parte di P perchè l'algoritmo ne possa individuare un punto (a patto che la parte di P compresa in E_0 abbia volume non nullo).

Per quanto riguarda la seconda assunzione, si procede perturbando le disequazioni originali in disequazioni della forma:

$$2^L {a_i}^T x \leq 2^L \beta_i + 1 \qquad i = 1, 2, \ldots, m \tag{1.15}$$

al quale è associato l'insieme di soluzioni P'. Khachiyan ha dimostrato che P è non vuoto se e solo se lo è anche P'. Inoltre si può ottenere una soluzione del sistema originale a partire da una del sistema perturbato in un numero polinomiale di passi (si veda anche (3.4)) e per codificare il sistema perturbato sono necessari al più $(m(n+1)+1)L$ bit, che è anch'esso polinomiale in L.

Vale la pena considerare questo sistema in quanto si può dimostrare che se il sistema (1.1) ammette una certa soluzione $\widehat{x}$, allora $P' \supseteq S(\widehat{x}, 1/\max_i \|2^L a_i\|)$. Dal momento che $\|a_i\| \leq 2^L$ si ha allora:

$$S(\widehat{x}, 2^{-2L}) \subseteq P' \tag{1.16}$$

se $\widehat{x} \in P$.

Abbiamo quindi ottenuto, per il sistema perturbato nella forma (1.15), $R = 2^L$ e $r = 2^{-2L}$, che se sostituite nella quantità $\log(R/r)$ precedendemente considerata danno $\log(2^L/2^{-2L}) = 3L$. Sostituendo questo nella (1.14) si ottiene $k \leq 6Ln(n+1)$, che è polinomiale in L e mostra come siano necessarie solo un numero polinomiale di iterazioni per trovare una soluzione, superate le quali si può considerare il sistema impossibile.

Capitolo 2

Miglioramenti dell'algoritmo

In questo capitolo si descriveranno alcuni accorgimenti che permetto di migliorare il tasso di convergenza dell'algoritmo. È da sottolineare il fatto che anche con questi miglioramenti l'algoritmo risulta essere meno efficiente di altri metodi usati in programmazione lineare. La loro importanza è dovuta all'applicazione dell'algoritmo dell'elissoide nel suo campo di nascita, cioè in programmazione convessa, dove permette di ottimizzare anche funzioni non derivabili.

2.1 Tagli profondi

L'algoritmo di base prevede, in presenza di un vincolo violato $a^T x \leq \beta$, di considerare la semi-elissoide (1.5). In realtà la violazione del vincolo porta con sè un'informazione maggiore di quella usata nell'iterazione, infatti si potrebbe usare come taglio direttamente il vincolo stesso, e cercare l'elissoide di minimo volume contenente:

$$\{x \in E_k : a^T x \leq \beta\} \tag{2.1}$$

Come si vede in figura (2.1), il taglio generato con questo metodo può essere molto più efficente di quello eseguito dall'algoritmo di base. Questo è indicato col nome di *taglio profondo* (deep cut), e si differenzia da un taglio normale, che è detto taglio leggero (shallow cut). I tagli profondi furono proposti per la prima volta da Shor e Gershovich.

Per generare l'elissoide richiesta si usano le (1.6)-(1.7), ma anzichè usare i parametri (1.8) si usano i seguenti:

$$\tau = \frac{1+n\alpha}{1+n} \quad \sigma = \frac{2(1+n\alpha)}{(n+1)(1+\alpha)} \quad \delta = \frac{n^2}{n^2-1}(1-\alpha^2) \tag{2.2}$$

$$\text{dove} \quad \alpha = \frac{a^T x_k - \beta}{\sqrt{a^T B_k a}} \tag{2.3}$$

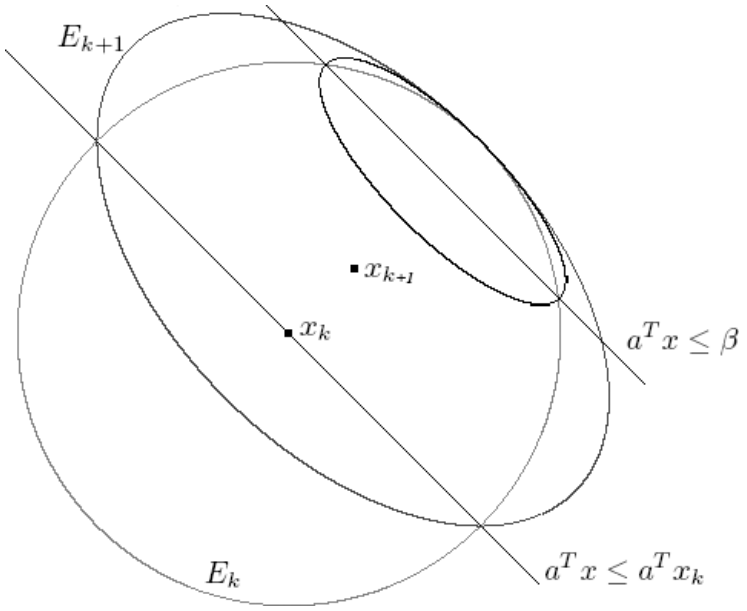


Figura 2.1: Esempio di taglio profondo

il parametro α corrisponde alla distanza di x_k dallo semispazio $H = \{x \in \Re^n : a^T x \leq \beta\}$ nella metrica corrispondente alla matrice B_k (cioè definendo la norma di un vettore come $\|y\|_B = (y^T B y)^{1/2}$). Per dettagli su questi concetti si veda [6], per l'interpretazione di α e per la dimostrazione della validità dei parametri (2.2) si veda [2]. Ora verranno citati alcuni risultati, giustificati anch'essi in [2]. Il parametro α genera tagli validi solo se $-1/n \leq \alpha \leq 1$. In particolare si distinguono i seguenti casi:

$\alpha < -1/n$ Si ha che l'elissoide di volume minimo contenente $E_k \cap H$ è ancora E_k.

$-1/n \leq \alpha < 0$ Si ha un taglio leggero, che contiene più di metà elissoide E_k, compreso x_k.

$0 \leq \alpha \leq \mathbf{1}$ Si ha un taglio profondo.

$\alpha > 1$ L'insieme $E_k \cap H$ è vuoto, quindi il sistema è impossibile.

Se si considera il rapporto tra i volumi di E_{k+1} e E_k, si ottiene, con un procedimento analogo a quello presentato nel capitolo precedente, la seguente relazione in funzione di α:

$$r(\alpha) = \frac{vol(E_{k+1})}{vol(E_k)} = \left(\frac{n^2(1-\alpha^2)}{n^2-1}\right)^{\frac{n-1}{2}} \frac{n(1-\alpha)}{n+1} \tag{2.4}$$

Questa funzione decresce monotonamente da 1, per $\alpha = -1/n$, a 0 per $\alpha = 1$. Applicando il metodo dei tagli profondi all'algoritmo dell'elissoide, si calcola il valore di α per tutti i vincoli violati da x_k, e si genera il taglio usando il vincolo con α più prossimo possibile a 1, cioè quello che garantisce il rapporto tra volumi $r(\alpha)$ minore. Se uno o più tra gli α calcolati è maggiore di 1, il sistema risulta impossibile.

2.2 Tagli surrogati

Combinando due o più vincoli, si possono generare tagli molto più profondi rispetto a quelli generari da un singolo vincolo. Come si vede in figura, combinando ad esempio due vincoli violati dal centro dell'elissoide E_k si può considerare un'area minore da richiudere con l'elissoide E_{k+1}. I tagli generati con questa tecnica si dicono *tagli surrogati* (surrogate cuts), in quanto erano stati proposti per sostituire i tagli normali e i tagli profondi nel metodo dell'elissoide.

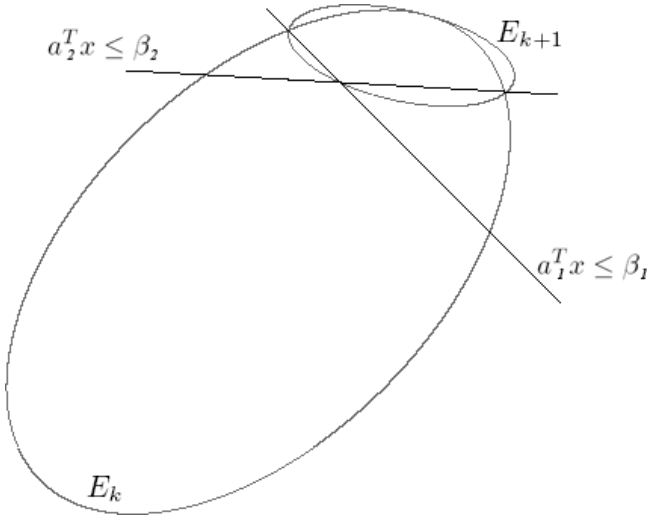


Figura 2.2: Esempio di taglio surrogato

Combinando più vincoli, si ottieme un taglio della forma $u^T A^T x \leq u^T b$, che equivale alla forma $a^T x \leq \beta$, posti $a = Au$ e $\beta = u^T b$. Si hanno tagli validi solo per $u \geq 0$. Il migliore di questi tagli si otterrebbe risolvendo

$$\max_{u \geq 0} u^T (A^T x_k - b)/(u^T A^T B A u)^{1/2} \tag{2.5}$$

che equivale a risolvere un problema di programmazione quadratica.

In alternativa si può considerare il sistema di disequazioni:

$$\bar{A}^T x \leq \bar{b} \tag{2.6}$$

dove $\bar{A}$ è un sottoinsieme di vincoli di (1.1) linearmente indipendenti e almeno uno di essi è violato da x_k. Goldfarb e Todd hanno dimostrato in [4] che se il vettore

$$\bar{u} = (\bar{A}^T B_k \bar{A})^{-1}(\bar{A}x_k - \bar{b}) \tag{2.7}$$

è non negativo, allora il taglio surrogato $\bar{u}^T \bar{A}^T x \leq \bar{u}^T \bar{b}$ è il più profondo per quel sottoinsieme di vincoli. Hanno dimostrato inoltre che se $\bar{A}^T B_k \bar{A}$ non ha elementi positivi al di fuori della diagonale e x_k viola tutti i vincoli allora $\bar{u}$ è non negativo.

Purtroppo risolvere la (2.5), o calcolare $\bar{u}$ per un gran numero di sottoinsiemi di vincoli può essere computazionalmente troppo costoso, per questo nel metodo dei tagli surrogati si considerano solo tagli generari da due vincoli. Krol e Mirman, in [5], forniscono condizioni necessarie e sufficienti per generare tagli surrogati a partire dal vincolo più violato tra quelli del sistema e uno meno violato o addirittura soddisfatto. Queste condizioni indicano, specificati i due vincoli, se $\bar{u}$ è non negativo.

Il metodo dei tagli surrogati prevede di usare iterativamente il taglio surrogato appena generato, che è più profondo di qualsiasi altro generabile da un vincolo singolo, per formare il taglio dell'iterazione successiva, combinandolo con un nuovo vincolo del sistema. In realtà questo procedimento può essere visto anche come l'applicazione del *relaxation method* per risolvere la determinatezza di (1.1), quindi non verrà ulteriormente approfondito in questa tesina e si rimanda ai testi già citati in precedenza per ulteriori dettagli.

2.3 Tagli paralleli

L'ultima variazione apportabile all'algoritmo di base riguarda la possibilità di considerare contemporaneamente, nel caso siano presenti, due vincoli paralleli, per generare la nuova elissoide. I tagli generati con questa tecnica prendono il

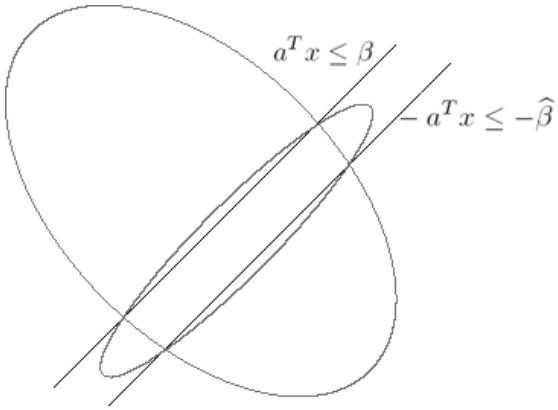


Figura 2.3: Esempio di tagli paralleli

nome di *tagli paralleli* (parallel cuts) e, come i tagli profondi, furono proposti da Shor e Gershovich.

Due vincoli paralleli sono nella forma

$$a^T x \le \beta \qquad \text{e} \qquad -a^T x \le -\widehat{\beta}$$

ed è possibile considerarli in contemporanea per generare la nuova elissoide se valgono le seguenti:

$$\begin{array}{rcccl} \alpha & \le & -\widehat{\alpha} & \le & 1 \\ \alpha\widehat{\alpha} & < & 1/n & & \end{array}$$

dove

$$\begin{aligned} \alpha &= (a^T x_k - \beta)/\sqrt{a^T B_k a} \\ \widehat{\alpha} &= (\widehat{\beta} - a^T x_k)/\sqrt{a^T B_k a} \end{aligned}$$

In questo caso si usano le formule (1.6)-(1.7) con i parametri

$$\begin{aligned} \sigma &= \big(1/(n+1)\big)\Big(n + \big(2/(\alpha-\widehat{\alpha})^2\big)\big(1 - \alpha\widehat{\alpha} - \rho/2\big)\Big) \\ \tau &= ((\alpha-\widehat{\alpha})/2)\sigma \\ \delta &= \big(n^2/(n^2-1)\big)\big(1 - (\alpha^2 + \widehat{\alpha}^2 - \rho/n)/2\big) \\ &\quad \text{dove} \quad \rho = \sqrt{4(1-\alpha^2)(1-\widehat{\alpha}^2) + n^2(\widehat{\alpha}^2 - \alpha^2)^2} \end{aligned}$$

Questi generano l'elissoide contenente la porzione $\{x \in E_k : \widehat{\beta} \le a^T x \le \beta\}$ di E_k. Nel caso $\beta = \widehat{\beta}$, cioè nel caso tutte le soluzioni giacciano nel piano $a^T x = \beta$, si pongono $\tau = \alpha$, $\sigma = 1$ e $\delta = (n^2/(n^2-1))(1-\alpha^2)$, che permettono di ottenere E_k piatta nella direzione di a. Per maggiori dettagli sulla nascita di queste formule e la dimostrazione della loro validità si vedano i riferimenti riportati nella sezione 4 di [2].

Capitolo 3

Metodo dell'Elissoide in Programmazione Lineare

In questo capitolo si mostrerà come l'algoritmo di base può essere applicato a problemi di programmazione lineare. Si considera cioè il problema di massimizzare il valore di una funzione, detta *funzione obiettivo*, su un insieme convesso in $\Re^n$ definito attraverso un sistema di disequazioni lineare come il (1.1):

$$\begin{cases} \max c^T x \\ A^T x \leq b \\ x \geq 0 \end{cases} \tag{3.1}$$

Alcuni metodi producono in tempo polinomiale delle soluzione approssimate, vicine a piacere a una soluzione ottima; si mostrerà nell'ultima sezione un metodo per trovare, ancora in tempo polinomiale, una soluzione ottima da una approssimata.

3.1 Soluzione simultanea dei problemi primale e duale in $\Re^{m+n}$

Ad ogni problema (3.1), detto problema *primale*, è associato un problema *duale*

$$\begin{cases} \min b^T y \\ Ay \geq c \\ y \geq 0 \end{cases} \tag{3.2}$$

Dalla *dualità forte*, si sa che (3.1) ha una soluzione finita se e solo se anche (3.2) ne possiede una, e in questo caso i rispettivi valori delle funzioni obiettivo coincidono ($c^T x = b^T y$). Inoltre dalla *dualità debole* è noto che per ogni soluzione finita x del primale e y del duale, vale $c^T x \leq b^T y$. Così si ha che x^* e y^* sono ottime rispettivamente per (3.1) e (3.2) se e solo se risolvono il sistema

di disequazioni lineari

$$\begin{cases} A^T x & \leq & b \\ -x & \leq & 0 \\ -Ay & \leq & -c \\ -y & \leq & 0 \\ -c^T x + b^T y & \leq & 0 \end{cases} \tag{3.3}$$

Questo sistema può essere risolto tramite il metodo dell'elissoide esposto nel Capitolo 1, che produce in questo caso contemporaneamente una soluzione per entrambi i problemi.

Questo approccio presenta molti svantaggi pratici. Innanzitutto si opera in $\Re^{m+n}$ e l'elevato numero di dimensioni rallenta la convegenza (si ricorda che sono richiesti fino a $6Ln(n+1)$ passi e in questo caso il numero di variabili n è $n+m$). Inoltre spesso sono noti dei limiti allo spazio delle soluzioni di (3.1) e quindi un algoritmo che opera solo sul primale potrebbe essere inizializzato con un'elissoide iniziale di dimensioni molto inferiori, accelerando la convergenza. Invece i limiti per il duale non sempre sono disponibili e ricavarli può essere computazionalmente dispendioso, quindi bisogna inizializzare l'algoritmo per risolvere il (3.3) con un'elissoide più grande del necessario.

Un altro svantaggio consiste nel fatto che tutte le soluzioni giacciono sul piano $c^T x = b^T y$, quindi l'insieme delle soluzioni, se esiste, ha volume nullo, e bisogna necessariamente perturbare il sistema. Anche se il sistema perturbato è determinato, il suo insieme delle soluzioni ha comunque un volume molto piccolo rispetto al volume dell'elissoide iniziale, fatto che fa prevede un gran numero di iterazioni dell'algoritmo prima di trovare una soluzione.
Per ultimo, si noti che se l'algoritmo determina che il (3.3) è impossibile, non è specificato se il (3.1) in particolare sia impossibile o indeterminato.

Alcuni di questi inconvenienti possono essere aggirati se si scelgono opportunamente i vincoli con cui generare i tagli. Infatti, eccetto per l'ultimo vincolo, i vincoli di (3.3) si possono dividere in due sottosistemi indipendenti. Se non si basa alcun taglio sull'ultimo vincolo la matrice B_k dell'elissoide E_k rimane *block diagonal* [1] con due parti corrispondenti alle variabili x e y. Quando si aggiorna

[1] Una *block matrix* (matrice a blocchi) è una matrice partizionata in sottomatrici chiamate *blocks* (blocchi), per esempio

$$P = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} = \begin{bmatrix} P_1 & P_2 \\ P_3 & P_4 \end{bmatrix} \quad \text{dove} \quad P_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, P_2 = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \dots$$

Una matrice quadrata si dice *block diagonal* se tutti i *blocks* della diagonale sono matrici quadrate, e tutti quelli al di fuori della diagonale hanno tutti elementi nulli, come per esempio:

$$P = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 4 & 4 \end{bmatrix} = \begin{bmatrix} P_1 & 0 \\ 0 & P_4 \end{bmatrix}$$

Si noti che i *blocks* fuori della diagonale possono essere anche rettangolari. Si veda un testo sulla teoria delle matrici per dettagli. Queste definizioni sono state trovate nel sito *http://www.mathreference.com/* e sono a cura di Karl Dahlke.

B_k, viene modificato solo il blocco relativo alle variabili usate nel taglio, quindi si aggiorna solo uno tra x_k e y_k. Si potrebbe procedere eseguendo tagli basati solo sui vincoli del problema primale fino a generare una sua soluzione, poi eseguendo tagli basati sui vincoli del duale fino a trovare una soluzione anche per questo e sfruttare solo a questo punto l'ultimo vincolo.
In questo modo il gran numero di dimensioni non è un problema troppo drammatico. Si supponga che prima di perturbare il (3.3), esso richieda L bit per essere codificato. Si può dimostrare che se non si genera una soluzione primale per il sistema perturbato in $6n(m+n+1)L$ passi allora (3.1) è impossibile. Se si genera una soluzione primale in k passi ma in $\min\{k+6m(m+n+1)L, 6(m+n)(m+n+1)L-k\}$ passi non si genera una soluzione duale, allora (3.1) è illimitato. Per ulteriori dettagli si veda [2].

3.2 Metodo della bisezione

Si è visto che considerando assieme il problema primale e duale si ha un rallentamento della convergenza dell'algoritmo dovuto all'elevato numero di dimensioni in cui si opera. Il metodo della bisezione cerca di evitare questo problema utilizzando un sistema della forma

$$\begin{cases} A^T x & \leq & b \\ -x & \leq & 0 \\ -c^T x & \leq & -\zeta \end{cases} \tag{3.4}$$

con vari valori di ζ.
Per prima cosa il metodo prevede di trovare con l'algoritmo dell'elissoide una soluzione primale x_-, se esiste, altrimenti si ha che (3.1) è impossibile. A questo punto è noto che $\zeta_- = c^T x_-$ è un lower bound per di (3.1), e si è ora interessati a cercare un upper bound.

Se l'insieme delle soluzioni di (3.1) è limitato e contenuto nell'elissoide corrente E_k, si può usare come upper bound $\zeta^+ = c^T x_k + (c^T B_k c)^{1/2}$, altrimenti si ricerca una soluzione y^+ del problema duale (3.2) e si pone $\zeta^+ = b^T y^+$. Se il duale è impossibile l'algoritmo termina (e il primale è illimitato).

Ora cominciano le iterazioni vere e proprie dell'algoritmo. È noto che il valore ottimo della funzione si trova nell'intervallo $[\zeta_-, \zeta^+]$. Si pone $\zeta = (\zeta_- + \zeta^+)/2$ nel sistema (3.4) e lo si risolve col metodo dell'elissoide. Se viene generata una soluzione x_k, si pone $x_- \leftarrow x_k$, $\zeta_- \leftarrow c^T x_k$ e si itera l'algoritmo. Altrimenti se risulta impossibile, si pone $\zeta^+ \leftarrow \zeta$ e si itera. Si esegue il procedimento finchè la quantità $\zeta^+ - \zeta_-$ diventa sufficientemente piccola.

Questo algoritmo che combina la bisezione con il metodo dell'elissoide per problemi di programmazione lineare è stato proposto da Padberg e Rao. Ha il vantaggio di operare in $\Re^n$ (eccetto eventualmente per risolvere una volta il duale in $\Re^m$). Presenta anche alcuni svantaggi, per esempio tutte le volte che il sistema definito dal ζ corrente è impossibile, porterà l'algoritmo a compiere tutti i $k = 6Ln(n+1)$ passi per mostrarne l'impossibilità. Si consiglia quindi di usare tecniche basate sui tagli profondi o surrogati per accelerare il più possibile

la convergenza.
Si noti anche che quando ζ diventa maggiore del precedente (cioè è stato generato un nuovo x_-), si può usare l'ultima elissoide generata nell'iterazione, centrandola in x_-, come elissoide iniziale per l'iterazione successiva. Se invece ζ decresce, cioè il problema (3.4) era impossibile, è necessario tornare all'ultima elissoide che aveva un centro valido, si ha quindi che l'algoritmo deve tornare a configurazioni precedenti (backtrack). Il prossimo algoritmo cercherà di superare questo inconveniente.

3.3 Metodo della funzione obiettivo scorrevole

Questo metodo prevede, come il precedente, di trovare un lower bound iniziale risolvendo il (3.1), trovando così un x_- e risolvere il (3.4) con $\zeta \leftarrow \zeta_- = c^T x_-$ con il metodo dell'elissoide. Si noti che nonostante x_- sia una soluzione ammissibile del problema, viene usato per generare il taglio $c^T x \geq c^T x_- = \zeta$. Quando si trova un nuovo x_k, se soddisfa $c^T x_k > c^T x_- = \zeta_-$, si pone $\zeta \leftarrow \zeta_- \leftarrow c^T x_k$, e si procede come sopra. L'ultima elissoide dell'iterazione precedente può sempre essere usata come elissoide iniziale della nuova iterazione, non si deve quindi mai tornare a considerare configurazioni precedenti (l'algoritmo non ha backtrack). Inoltre si considerano sempre problemi determinati, in quanto il taglio è passante per la soluzione ammissibile x_-, e tutte le operazioni avvengono in $\Re^n$. Per questi motivi è da considerare il più efficiente tra quelli esposti in questa tesina.
Se l'insieme delle soluzioni è limitato ed è contenuto in E_k, si possono trovare degli upper bound ${\zeta_k}^+$:

$$ {\zeta_k}^+ = \min\{{\zeta_{k-1}}^+, c^T x_k + (c^T B_k c)^{1/2}\} $$

Si terminano le iterazioni quando ${\zeta_k}^+ - \zeta_-$ è sufficientemente piccolo.

In fase di inizializzazione, una volta che si è verificato che il (3.1) è possibile con il metodo dell'elissoide, bisogna verificare che non sia illimitato. Per fare ciò, si può applicare il metodo dell'elissoide ai vincoli del duale (ma si opererebbe in $\Re^m$), oppure si cerca di risolvere il seguente sistema:

$$ \begin{cases} A^T x & \leq & 0 \\ -x & \leq & 0 \\ -c^T & x & \leq -1. \end{cases} $$

Ogni soluzione a queste disuguaglianze implica che (3.1) è illimitato. Dato che richiedono molte nozioni aggiuntive, per ulteriori dettagli sulla polinomialità del resto del metodo si rimanda a [2]. Il metodo fu proposto da Yudin, Nemirovskii e Shor; viene utilizzato anche da Grötschel e altri in [3] per dimostrare alcune equivalenze polinomiali di certi problemi di ottimizzazione combinatoria.

3.4 Soluzioni esatte da soluzioni approssimate

Gli ultimi due metodi trattati forniscono solo soluzioni approssimate a problemi di programmazione lineare. Per completare la soluzione di questi problemi in tempo polinomiale bisogna poter ricavare una soluzione ottima da una approssimata in tempo ancora polinomiale. Innanzitutto è necessario introdurre la seguente definizione:

Definizione 3 *Per ogni $\epsilon > 0$, si dice che x è una* soluzione approssimata ϵ *(ϵ-approximate solution) di (3.1), se esistono una soluzione ammissibile y e una soluzione ottima x^* di (3.1) tali che:*

$$\|y - x\| \leq \epsilon \quad e \quad c^T(x^* - x) \leq \epsilon$$

In [3], Grötschel, Lavosz e Schrijver discutono la seguente tecnica per ricavare una soluzione ottima da una soluzione approssimata ϵ.

Si supponga di sapere che (3.1) ha una soluzione ottima. Sia Δ un intero positivo e x^* un vettore razionale della forma

$$\left(\frac{p_1}{q_1}, \ldots, \frac{p_n}{q_n}\right)^T \qquad p_i, q_i \text{ interi e } |q_i| \leq \Delta \quad \forall i = 1, \ldots, n \tag{3.5}$$

Dato $x \in \Re^n$ tale che x^* è all'interno della sfera $S(x, 1/(2\Delta^2))$, allora x^* è l'unico vettore razionale di forma (3.5) in tale sfera. A questo punto, se x e x^* sono come quelli descritti, e x è noto, si può ricavare x^* arrotondando ogni componente di x al più vicino p/q con $|q| \leq \Delta$ tramite il metodo delle frazioni continue[2].

Si è interessati al caso in cui x^* è un estremo ottimo di un problema come il (3.1) e x è stato trovato col metodo dell'elissoide. In [3] si propone una perturbazione della funzione obiettivo, rimpiazzando il vettore c col vettore perturbato $d = \gamma^n c + (\gamma^0, \ldots, \gamma^{n-1})$, in modo che il problema

$$\begin{cases} \max d^T x \\ A^T x \leq b \\ x \geq 0 \end{cases} \tag{3.6}$$

[2]Si dice *frazione continua* una frazione scritta nella forma:

$$a_1 + \cfrac{b_1}{a_2 + \cfrac{b_2}{a_3 + \cfrac{b_3}{a_4 + \ddots}}}$$

Per esempio il numero 67/29 (=2,3103448...) può essere scritto come:

$$\frac{67}{29} = 2 + \frac{9}{29} = 2 + \frac{1}{\frac{29}{9}} = 2 + \frac{1}{3 + \frac{2}{9}} = \cdots = 2 + \frac{1}{3 + \frac{1}{4 + 1/2}}$$

Da cui si ottengono le approssimazioni 2, 2+1/3 = 2,3333..., 2+1/(3+1/4) = 2+4/13 = 2,3076..., e per ultimo il valore originale dell'espressione: 2+1/(3+1/(4+1/2)) = 2 + 9/29. Definizione ed esempio tratti da *Continued Fractions* di C. D. Olds, 1963, Yale University (si è fatto riferimento alla traduzione italiana *Frazioni Continue*, Zanichelli Bologna, 1968).

abbia un'unica soluzione ottima x^* che risolve anche (3.1). Nel testo citato si discute un opportuno valore di γ che consenta di mantenere le dimensioni del problema (3.6) polinomiali rispetto a (3.1) e un valore di Δ adatto a x^*. Si discute inoltre di come scegliere ϵ tale che la soluzione approssimata ϵ sia ottenibile in tempo polinomiale in L dall'algoritmo dell'elissoide. Si dimostra che una volta trovata una soluzione approssimata, sono sufficienti $O[n(p+L)]$ operazione elementari per ottenere una soluzione ottima, dove p è la precisione in bit dei numeri gestiti dal metodo dell'elissoide. Per dettagli si vedano [2] e soprattutto [3].

Capitolo 4

Implementazione Java dell'algoritmo

In questo capitolo si mostrerà un'implementazione Java semplificata dell'algoritmo dell'elissoide usato per dimostrare la determinatezza di un sistema di disequazioni lineari. Essa opererà in $\Re^2$ e permetterà di visualizzare le varie elissoidi (in questo caso ellissi) generate durante l'esecuzione del metodo.
Si descriveranno le varie classi che compongono il software, cercando di mostrare quali possono essere i problemi nella realizzazione di versioni più robuste e generali. Alla fine del capitolo si mostreranno alcuni output significativi del programma.

4.1 Programma d'esempio e possibili miglioramenti

Il programma è composto da tre classi che si dividono le funzionalità richieste. Una prima classe, *PianoCartesiano*, rappresenta, come dice il nome, un piano cartesiano, e ha il compito di gestire gli output, una seconda classe, *Interfaccia*, si occupa della lettura gli input e dell'inizializzazione del programma e infine una terza classe, *MetodoElissoide*, che è quella che implementa l'algoritmo vero e proprio. In seguito si illustreranno gli scopi e le strutture di ciascuna classe, per i sorgenti completi si veda l'Appendice A.

4.1.1 Classe PianoCartesiano

Questa classe gestisce un piano cartesiano a due dimensioni, e permette di disegnare punti, rette e coniche. Il metodo dell'elissoide, se non viene inizializzato con un insieme delle soluzioni predefinito, considera la sfera $B(0, 2^L)$, che può essere un cerchio di raggio dell'ordine di 10^{15}-10^{20} unità anche per sistemi di sole 3-5 disequazioni, per poi arrivare dopo diversi passaggi a considerare ellissi

con assi di molti ordini di grandezza minori. Quindi il programma deve poter in qualche modo gestire scale di visualizzazione diverse.
Dato che lo scopo che si cerca di raggiungere per questa tesina è la semplice visualizzazione delle due ellissi E_k e E_{k+1} generate in un passaggio dell'algoritmo, per poi salvarle su un'immagine, si risolve il problema di gestire scale diverse creando una nuova istanza della classe con una scala minore quando serve.
I coefficienti delle equazioni e dei punti del piano cartesiano sono memorizzati in numeri a precisione doppia a 64 bit (double), che permettono di gestire numeri approssimati con circa 15 cifre decimali significative (52 cifre binarie) tra circa 10^{-324} e 10^{308} ($(2^{-52})2^{-1022} = 2^{-1074}$ e $(2 - 2^{-52})2^{1023} \simeq 2^{1024}$). Si noti che la precisione di visualizzazione di questa classe non influisce con la convergenza dell'algoritmo. La classe permette di salvare il piano cartesiano in un'immagine bitmap a 256 colori (si possono assegnare dei colori alle varie ellissi).

In seguito si riporta l'interfaccia pubblica della classe.

```
/* costruttore della classe, imposta le dimensioni totali del
   piano, da quanti punti e' costituita un'unita' e le dimensioni
   del primo quadrante */
public PianoCartesiano(double h, double w, double ris,
     double h1, double w1)

// disegna il punto (x,y)
public void disegnaPunto(double x, double y, int col)

// disegna ax + by + c = 0
public void disegnaRetta(double a, double b, double c, int col)

// disegna la conica ax^2 + bxy + cy^2 + dx + ey + f = 0
public void disegnaConica(double a, double b, double c,
     double d, double e, double f, int col)

// salva in un'immagine il contenuto del piano
// unitaMis serve per definire le unita' di misura degli assi
public void salva8bit(String nomeFile, double unitaMis)
```

4.1.2 Classe Interfaccia

Questa classe ha lo scopo di leggere dal file di testo 'input.txt' un sistema di disequazioni, con un'opportuna formattazione, da usare per creare un'istanza di MetodoElissoide. Il formato del file prevede l'inserimento del numero m di vincoli che costituiscono il sistema, e poi dei coefficienti degli m vincoli. Può gestire solo i casi a due incognite. Nella tabella (4.1) si mostra un esempio di come trasformare un sistema nel formato adatto alla lettura per questa classe.

Per lanciare il programma è sufficiente invocare `java Interfaccia` da riga di comando. In alternativa, è possibile passare un numero come parametro, che viene interpretato dal programma come raggio del cerchio E_0 per inizializzare l'algoritmo, anzichè usare il valore 2^L.

		`4`
$-x + y \leq -10$		`-1 1 -10`
$x - y \leq 14$	$\Rightarrow$	`1 -1 14`
$-x - y \leq -20$		`-1 -1 -20`
$x + y \leq 24$		`1 1 24`

Tabella 4.1: Esempio di formattazione del file 'input.txt'

4.1.3 Classe MetodoElissoide

L'interfaccia pubblica della classe è molto semplice, prevede semplicemente il costruttore e il metodo solve(), per avviare l'algoritmo e generare una soluzione se possibile. Quest'ultimo metodo utilizza comunque molti altri metodi privati che gli permettono di eseguire le computazioni.

La classe ha le seguenti variabili d'esemplare:

- `double[][] At, B, Binv`: rispettivamente A^T di (1.1), B_k usata nella formula (1.7) e ${B_k}^{-1}$, che permette di scrivere l'elissoide come conica tramite la (1.2).
- `double[] b, xk, Ek`: il vettore b di (1.1), il centro dell'elissoide x_k e i coefficienti della conica corrispondente all'elissoide corrente.
- `int n, L, maxIteration`: rispettivamente il numero di incognite n, la dimensione del problema L e il numero massimo di iterazioni che l'algoritmo deve fare prima di considerare il problema impossibile.
- `double tau, sigma, delta`: i parametri (1.8).

Nelle prossime sottosezioni verranno analizzati più nel dettaglio i metodi della classe.

Costruttore

Il costruttore accetta come parametri due array `double[][] At` e `double[] b` che rappresentano A^T e b del sistema (1.1). Controlla che il numero di incognite n sia 2, in caso contrario esce. Invece il numero di vincoli m può essere arbitrario e viene configurato in base agli array dati in input. Da questi può calcolare le dimensioni del problema, usando la formula (1.9):

```
int L = 0; // determino dimensioni problema
double log2 = Math.log(2);
for (int i=0; i<At.length; i++) {
    for (int j=0; j<At[i].length; j++) {
        if (Math.abs(At[i][j])!=0)
            L += (int)(Math.log(Math.abs(At[i][j]))/log2);
    }
}
```

```
for (int i=0; i<b.length; i++) {
    if (Math.abs(b[i])!=0)
        L += (int)(Math.log(Math.abs(b[i]))/log2);
}

L += (int)(Math.log(n)/log2 + Math.log(b.length)/log2);
L += 2*b.length*(n+1)+4; // + 2mn + 2m + 4
```

Inoltre vengono inizializzati i parametri tau, sigma, delta in funzione di n. Avendo a disposizione L, l'algoritmo può anche inizializzare l'elissoide iniziale E_0, che è una sfera di raggio $r = 2^L$, e quindi un'elissoide con $B_0 = r^2 I = 2^{2L} I$.

```
// E0 e' una ball con r=2^L, xk=(0,0)
B = new double[2][2];
B[0][0] = B[1][1] = Math.pow(2,2*L);
xk = new double[2];
xk[0]=xk[1]=0;
```

L'utilizzo di un E_0 diverso non è previsto da questa classe, e il suo sviluppo viene delegato a Interfaccia. Una versione completa dell'algortimo permetterebbe di gestire anche configurazioni con un numero n generico di incognite, quindi B_k sarebbe una matrice $n \times n$ inizializzata a $r^2 I$.

Per ultimo, si imposta il numero massimo di iterazione che farà l'algoritmo

```
maxIteration = 6*n*(n+1)*L;
```

Metodo solve()

Questo metodo implementa l'algoritmo dell'elissoide vero e proprio, con l'aiuto di alcuni metodi privati della classe. Lo schema del metodo in pseudocodice è il seguente:

```
public double[] solve() {
   // calcola l'ellisse corrispondente all'elissoide corrente
   Ek = getEllisse(Bk,xk);

   for (int a=0; a<maxIteration; a++) {
    // ogni 15 iterazioni ridimensiona il piano in base alle
    // dimensioni dell'elissoide attuale
      if (a%15==0) ridimensiona piano;
      piano.disegnaConica(Ek);

      if (ci sono vincoli violati) {
         scegli un vincolo 'a' violato;
         aggiorna xk e Bk;

    // disegna la nuova ellisse e il taglio che l'ha generata
         Ek = getEllisse(Bk,xk);
         piano.disegna(Ek);
         piano.disegna(a x = a xk);
      }
```

```
      else {
    // ritorna la soluzione trovata
        return xk;
      }

    // salva l'immagine corrente e pulisce il piano
      piano.salvaImmagine();
      piano.clear();
   }

   return impossibile;
}
```

Nell'implementazione sono presenti i seguenti metodi di supporto a solve(); si ricorda che tutti i metodi implementati lavorano con numeri o array di opportune dimensioni di double:

matriceInversa(M) Calcola la matrice inversa di M. In questo programma si gestisce solo il caso in cui M è 2×2.

getEllisse(B^{-1}, x) Ricava i coefficienti a, b, c, d, e e f della conica corrispondente all'elissoide E(x, B), per permettere alla classe PianoCartesiano di disegnarla.

int[] vincoliViolati() Verifica se x_k corrente viola qualche vincolo. In caso affermativo ritorna un array con gli indici di tutti i vincoli violati.

informazioni() Stampa su standard output la dimensione attuale del piano cartesiano e le caratteristiche dell'elissoide corrente (x_k e B_k).

aggiornaEllisse() Aggiorna x_k e B_k usando le (1.6)-(1.8).

int[] getParametriPiano() Calcola i parametri da passare a PianoCartesiano per permettere una visualizzazione in scala corretta dell ellissi. Se si mantenesse sempre la scala iniziale in 20-25 passaggi circa le ellissi si ridurrebbero a dei punti. Viene richiamato ogni 15 iterazioni.

In quest'implementazione viene scelto un vincolo violato a caso, senza applicare criteri di scelta particolari. Come si vedrà in seguito, una modifica che si potrebbe apportare a questo criterio per sfruttare meglio la precisione dei double, consiste nel scegliere il vincolo violato più prossimo a formare un angolo retto con l'asse maggiore dell'ellisse corrente.

Se si volesse estendere il programma per gestire il caso con n generico, sarebbe sufficiente modificare i metodi richiamati da solve(), e in particolare il metodo per calcolare la matrice inversa e le formule di aggiornamento, che in quest'implementazione usano un algoritmo valido solo per problemi a due incognite. Ovviamente se si avessero più di due dimensioni non si avrebbe più a che fare con ellissi, quindi non si potrebbe effettuare la rappresentazione delle iterazioni su un piano cartesiano.

Al di là del numero di incognite gestite, bisogna notare che in realtà il grande limite in quest'implementazione è l'utilizzo di numeri a precisione doppia. Come si è specificato durante la dimostrazione di polinomialità dell'algoritmo, ogni numero usato ha bisogno di $23L$ cifre binarie prima della virgola e $38nL$ dopo, oltre ad opportuni coefficienti moltiplicativi nelle formule, per garantire di poter rappresentare adeguatamente E_{k+1}.
Per esempio si consideri il sistema

$$\begin{array}{rcrcr} -x & & +y & \leq & -10 \\ x & & -y & \leq & +14 \\ -x & & -y & \leq & -20 \\ x & & +y & \leq & +24 \end{array}$$

Questo ha $L = 45$ bit, il che significa che per essere risolto correttamente ha bisogno, in base a quanto dimostrato da Khachiyan, di 1035 bit prima e 3420 bit dopo la virgola, per una precisione totale di 4455 bit per ogni numero utilizzato durante l'esecuzione dell'algoritmo. Si può facilmente intuire che i 64 bit utilizzati dai double sono insufficienti per un programma che voglia gestire ogni possibile sistema. Per risolvere il problema in un'implementazione Java si può ricorre alla classe java.math.BigDecimal, che permette di gestire numeri della precisione voluta, a costo di rallentare pesantemente l'esecuzione del programma, in quanto i double sono un tipo numerico nativo, elaborato direttamente dai processori, mentre BigDecimal è una classe vera e propria.
Nella prossima sezione si vedrà un esempio che illustra i motivi della necessità di una precisione così elevata.

Altro fattore da tenere in considerazione in un'implementazione reale, è l'utilizzo delle tecniche esposte nel Capitolo 2, come i tagli profondi, che permettono di sfruttare con più efficienza le informazioni portate dai vincoli violati. Per la loro implementazione si andrebbe a modificare il metodo di aggiornamento dei parametri, che anzichè applicare semplicemente le (1.6)-(1.8), dovrebbe considerare il parametro α definito in (2.3) per ogni vincolo violato, e scegliere il vincolo con questo parametro più prossimo a uno.

Per ultimo, bisogna ricordare che nel caso in cui l'insieme delle soluzioni non abbia effettivamente n dimensioni bisognerebbe perturbare il sistema come spiegato nella sottosezione 1.2.3, per poi trovare una soluzione per il problema originale da un'eventuale soluzione di quello perturbato, come mostrato nella sezione 3.4. Non si è preso in considerazione lo sviluppo di questa tecnica nel programma a causa della bassa precisione utilizzata per i dati numerici, che non si presta né a dare garanzie di convergenza, né a supportare le operazioni di arrotondamento necessarie.

4.1.4 Implementazioni ottimizzate

In [2] sono riportate alcune utili informazioni riguardo le implementazioni dell'algoritmo sviluppate in ambito di ricerca. Si propongono due approcci, che mirano a ridurre il numero di operazioni richieste nelle computazioni e la memoria utilizzata.

Il primo sfrutta il fatto che $B_k = {J_k}^T J_K$ (come si può vedere nella sottosezione 1.2.2). In questo caso J_k non è unica e può essere scelta in modo tale da essere triangolare. Con questo stratagemma si risparmia quasi metà dello spazio necessario a memorizzarla, e un terzo delle operazioni necessarie all'aggiornamento.

Un secondo approccio considera B_k nella forma $L_k D_k {L_k}^T$, dove L_k è triangolare e D_k è positivamente definita. Se si sfrutta questa notazione, si possono aggiornare x_k, L_k e D_k in $2n^2 + O(n)$ operazioni, sacrificando però l'efficienza in termini di memoria utilizzata (si ha necessità di memorizza una matrice $n \times n$ in più).

Per dettagli riguardo a queste implementazioni si veda il testo citato e i relativi riferimenti.

4.2 Output significativi

Ora si esamineranno alcuni output ottenuti dal programma precedentemente descritto, che permettono di capire il comportamento del metodo dell'elissoide nelle applicazioni reali e i problemi a cui è soggetto.

4.2.1 Sistema 1 - Funzionamento

Nel primo esempio si userà l'algoritmo dell'elissoide nella risoluzione del sistema

$$\begin{cases} +2y \leq 15 \\ -2x + y \leq -10 \\ x - y \leq 10 \\ -x - 2y \leq -10 \\ 2x + y \leq 30 \end{cases} \tag{4.1}$$

in due casi. Nel primo si suppone di sapere che la soluzione appartenga a un cerchio di raggio 80 e si visualizzeranno i vari passaggi compiuti dal metodo per trovare una soluzione; nel secondo caso si applicherà l'algoritmo generale, cioè il cerchio iniziale avrà raggio 2^L.
Questo problema ha dimensione $L = 57$, cioè richiederà al massimo 2052 iterazioni nel caso dell'algoritmo generale. L'insieme delle soluzioni è l'area evidenziata nella figura (4.1).

Caso 1

Una volta inizializzato il programma con $E_0 = B(0, 80)$, si lavora in un piano cartesiano dove l'insieme delle soluzioni ha le dimensioni di figura (4.2). In seguito si vedranno i passaggi delle varie iterazioni, con specificati i vari x_k, B_k e il vincolo considerato per generare il taglio. Nelle figure, in rosso è rappresentata l'elissoide E_k e in blu l'elissoide E_{k+1} generata durante l'iterazione.
Quest'esempio mostra chiaramente come l'algoritmo consideri il taglio parallelo a un vincolo violato e passante per il centro dell'elissoide corrente, che

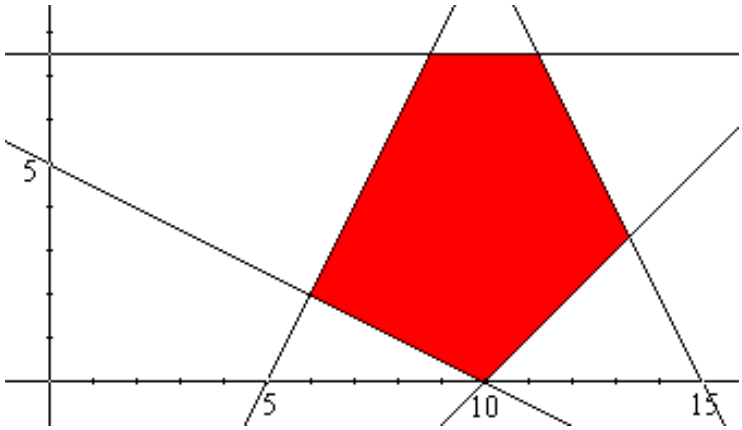


Figura 4.1: Insieme delle soluzioni del sistema (4.1)

va a definire una semi-elissoide candidata a contenere una soluzione. Con le formule note viene calcolata l'elissoide di volume minimo che contiene questa semi-elissoide.

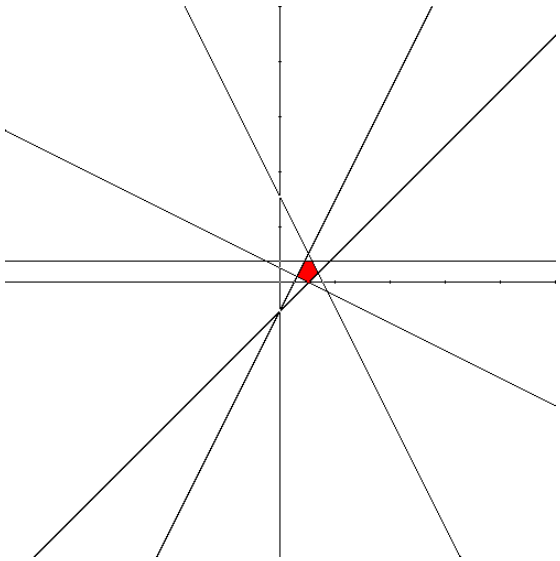
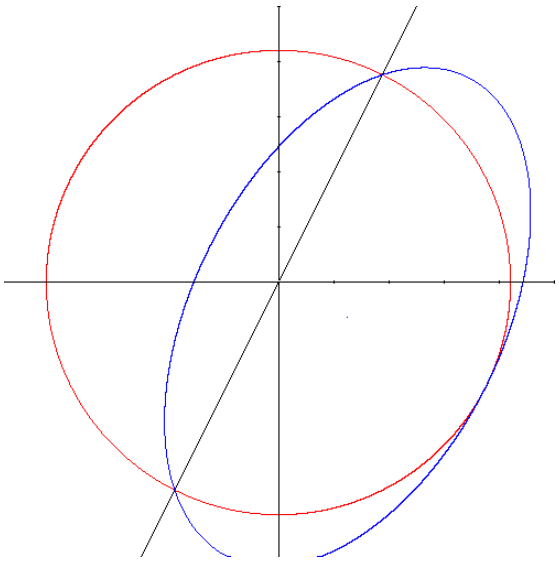


Figura 4.2: Insieme delle soluzioni del sistema (4.1), caso 1



Iterazione 1

$$B_0 = \begin{bmatrix} 6400 & 0 \\ 0 & 6400 \end{bmatrix}$$

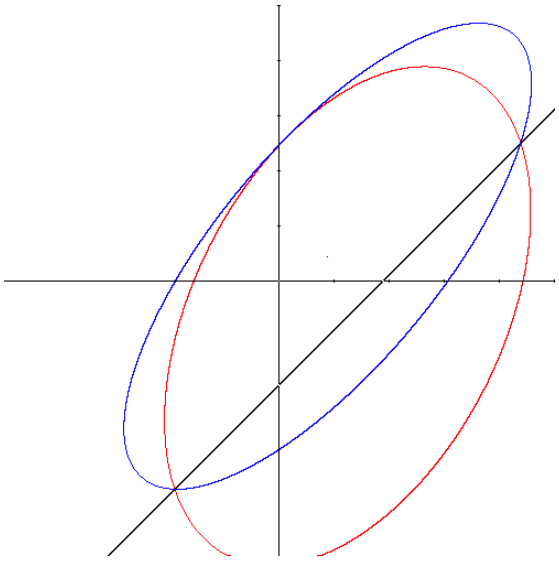
$x_0 = [0, 0]^T$

vincolo violato:

$-2x + y \leq -10$

$$B_1 = \begin{bmatrix} 3982.2 & 2275.5 \\ 2275.5 & 7395.5 \end{bmatrix}$$

$x_1 = [23.85, -11.92]^T$



Iterazione 2

$$B_1 = \begin{bmatrix} 3982.2 & 2275.5 \\ 2275.5 & 7395.5 \end{bmatrix}$$

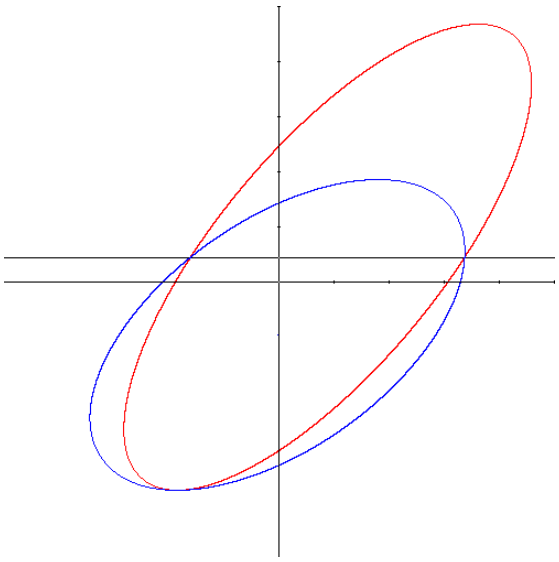
$x_1 = [23.85, -11.92]^T$

vincolo violato:

$x - y \leq 10$

$$B_2 = \begin{bmatrix} 4930.4 & 4171.9 \\ 4171.9 & 6447.4 \end{bmatrix}$$

$x_2 = [16.97, 8.73]^T$



Iterazione 3

$$B_2 = \begin{bmatrix} 4930.4 & 4171.9 \\ 4171.9 & 6447.4 \end{bmatrix}$$

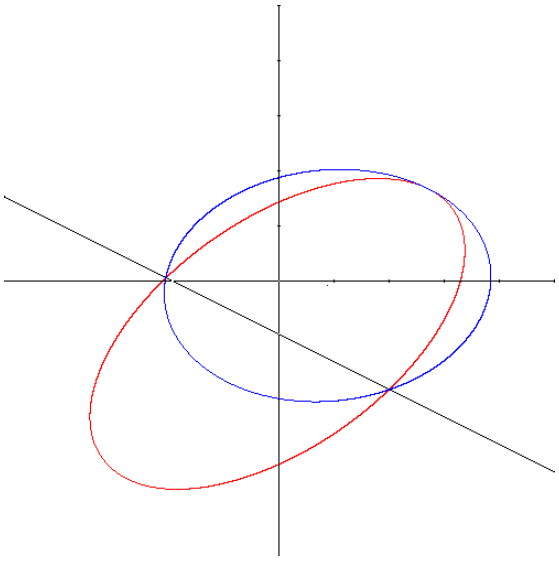
$x_2 = [16.97, 8.73]^T$

vincolo violato:

$2y \leq 15$

$$B_3 = \begin{bmatrix} 4174.3 & 1854.1 \\ 1854.1 & 2865.5 \end{bmatrix}$$

$x_3 = [-0.35, -18.03]^T$



Iterazione 4

$$B_3 = \begin{bmatrix} 4174.3 & 1854.1 \\ 1854.1 & 2865.5 \end{bmatrix}$$

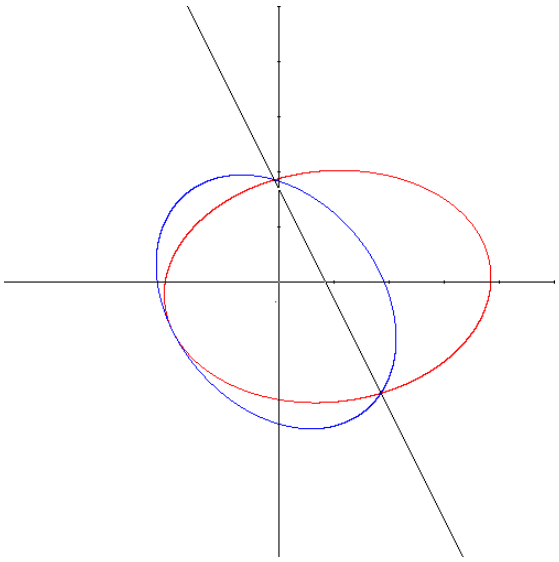
$x_3 = [-0.35, -18.03]^T$

vincolo violato:

$-x - 2y \leq -10$

$$B_4 = \begin{bmatrix} 3169.9 & 166.74 \\ 166.74 & 1602.2 \end{bmatrix}$$

$x_4 = [16.95, -1.38]^T$



Iterazione 5

$$B_4 = \begin{bmatrix} 3169.9 & 166.74 \\ 166.74 & 1602.2 \end{bmatrix}$$

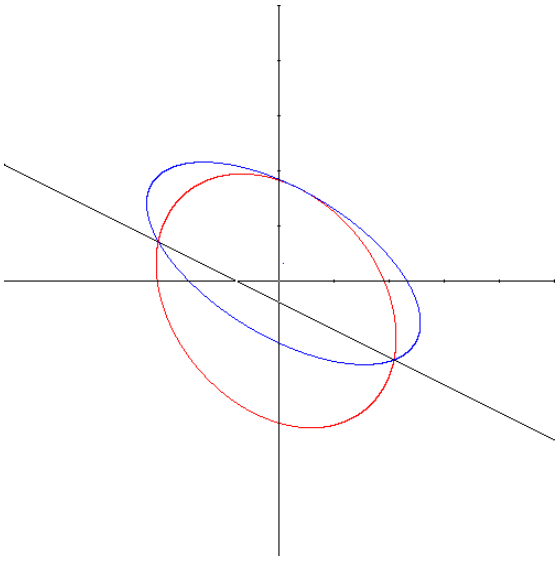
$x_4 = [16.95, -1.38]^T$

vincolo violato:

$2x + y \leq 30$

$$B_5 = \begin{bmatrix} 1709.2 & -526.6 \\ -526.6 & 1913.5 \end{bmatrix}$$

$x_5 = [-0.76, -6.65]^T$



Iterazione 6

$$B_5 = \begin{bmatrix} 1709.2 & -526.6 \\ -526.6 & 1913.5 \end{bmatrix}$$

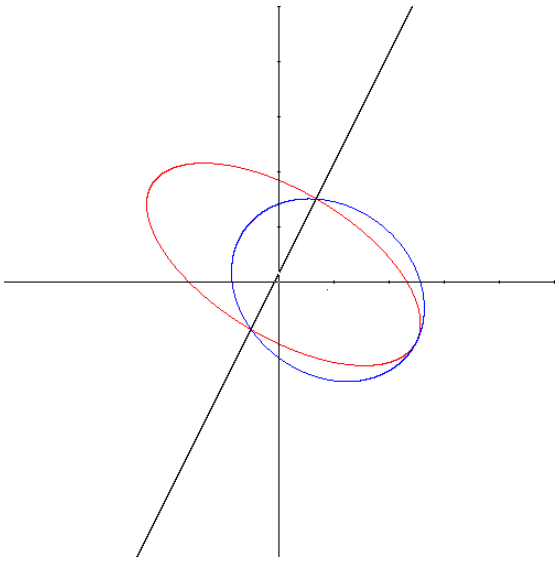
$x_5 = [-0.76, -6.65]^T$

vincolo violato:

$-x - 2y \leq -10$

$$B_6 = \begin{bmatrix} 2226.2 & -967.3 \\ -967.3 & 1217.1 \end{bmatrix}$$

$x_6 = [1.78, 6.25]^T$



Iterazione 7

$$B_6 = \begin{bmatrix} 2226.2 & -967.3 \\ -967.3 & 1217.1 \end{bmatrix}$$

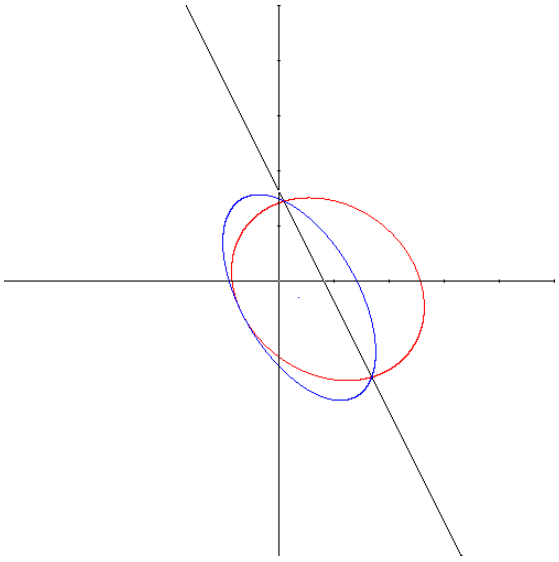
$x_6 = [1.78, 6.25]^T$

vincolo violato:

$-2x + y \leq -10$

$$B_7 = \begin{bmatrix} 1102.1 & -204.54 \\ -204.54 & 991.69 \end{bmatrix}$$

$x_7 = [17.05, -2.63]^T$



Iterazione 8

$$B_7 = \begin{bmatrix} 1102.1 & -204.54 \\ -204.54 & 991.69 \end{bmatrix}$$

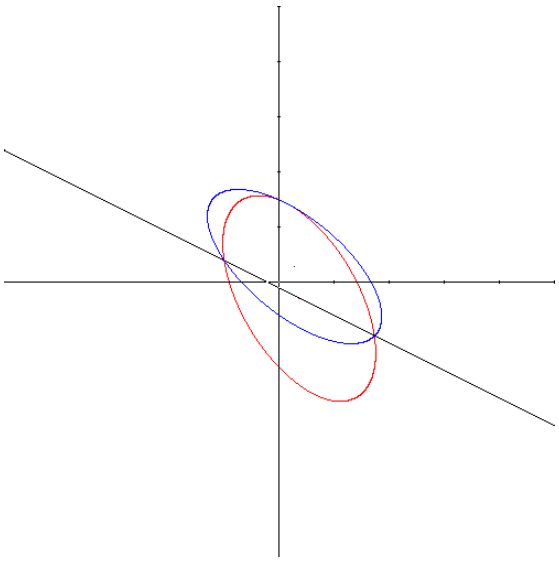
$x_7 = [17.05, -2.63]^T$

vincolo violato:

$2x + y \leq 30$

$$B_8 = \begin{bmatrix} 693.74 & -498.73 \\ -498.73 & 1256.4 \end{bmatrix}$$

$x_8 = [7.21, -5.50]^T$



Iterazione 9

$$B_8 = \begin{bmatrix} 693.74 & -498.73 \\ -498.73 & 1256.4 \end{bmatrix}$$

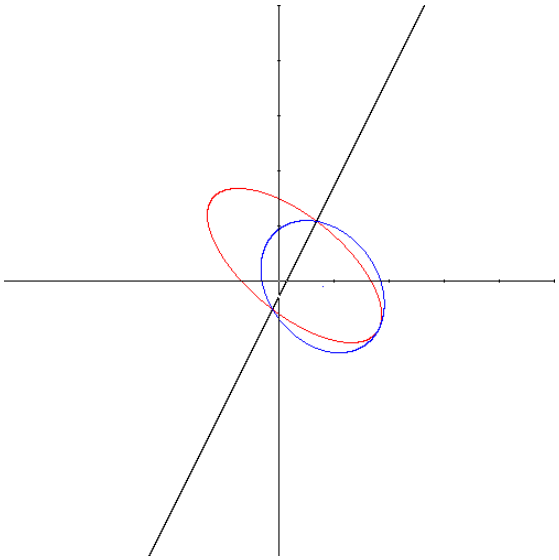
$x_8 = [7.21, -5.50]^T$

vincolo violato:

$-x - 2y \leq -10$

$$B_9 = \begin{bmatrix} 902.97 & -518.98 \\ -518.98 & 707.06 \end{bmatrix}$$

$x_9 = [5.55, 5.50]^T$



Iterazione 10

$$B_9 = \begin{bmatrix} 902.97 & -518.98 \\ -518.98 & 707.06 \end{bmatrix}$$

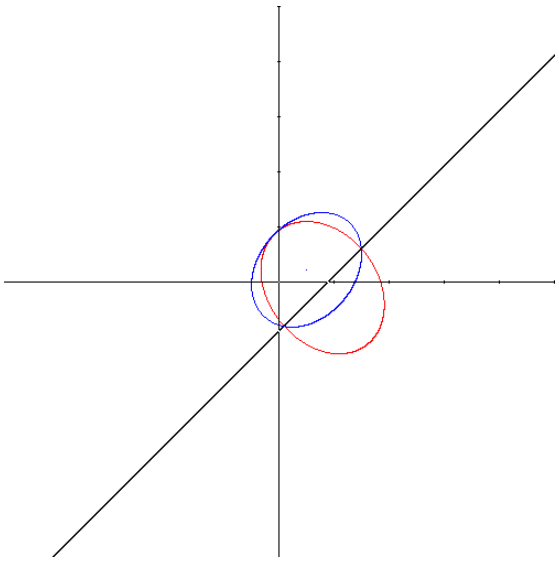
$x_9 = [5.55, 5.50]^T$

vincolo violato:

$-2x + y \leq -10$

$$B_{10} = \begin{bmatrix} 452.62 & -128.04 \\ -128.04 & 519.48 \end{bmatrix}$$

$x_{10} = [15.24, -1.77]^T$



Iterazione 11

$$B_{10} = \begin{bmatrix} 452.62 & -128.04 \\ -128.04 & 519.48 \end{bmatrix}$$

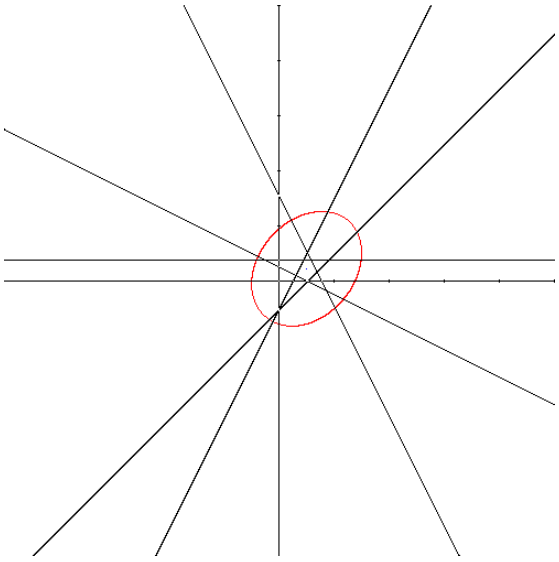
$x_{10} = [15.24, -1.77]^T$

vincolo violato:

$x - y \leq 10$

$$B_{11} = \begin{bmatrix} 359.48 & 101.40 \\ 101.40 & 389.19 \end{bmatrix}$$

$x_{11} = [9.71, 4.39]^T$



Iterazione 12

$$B_{11} = \begin{bmatrix} 359.48 & 101.40 \\ 101.40 & 389.19 \end{bmatrix}$$

$x_{11} = [9.71, 4.39]^T$

$x_{11} \in P \rightarrow$ FINE

Caso 2

In questo secondo caso si è inizializzato l'algoritmo con il cerchio $B(0, 2^L)$. Non si riporteranno i grafici ottenuti durante l'esecuzione, ma semplicemente alcune statistiche e considerazioni.
Come si è già detto, viene scelto un vincolo a caso tra quelli violati, quindi l'esecuzione del programma da risultati di volta in volta diversi. La classe è stata leggermente modificata per ritornare solo gli output testuali ed è stata eseguita in un ciclo for di una classe appositamente scritta per memorizzare tutti i risultati.

L'algoritmo è stato fatto eseguire 10000 volte in cui si sono ottenuti risultati validi in tutti i casi, con un numero di passaggi compreso tra 246 e 285, con una media di 278 iterazioni per esecuzione, circa il 13,3% del numero massimo di iterazioni previste. Il volume medio dell'elissoide finale, misurato come $\sqrt{\det(B_k)}$, è risultato 2310, mentre quello di E_0 è $2^{114} \simeq 2 \cdot 10^{34}$, quindi si misura un rapporto tra volumi di elissoidi consecutive medio di $\sqrt[278]{2310/2^{114}} \simeq 0.774$, migliore del massimo teorico calcolato con la (1.12), che vale circa 0.846.

Nella figura (4.3) viene visualizzata la distribuzione dei risultati. I prossimi

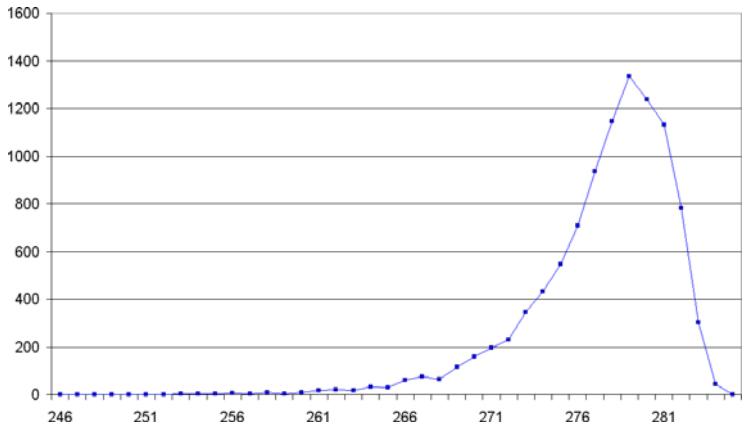


Figura 4.3: Numero di iterazioni per risolvere il problema (4.1)

esempi cercheranno di evidenziare gli effetti dell'utilizzo di numeri a precisione insufficiente e della mancata implementazione della perturbazione del sistema.

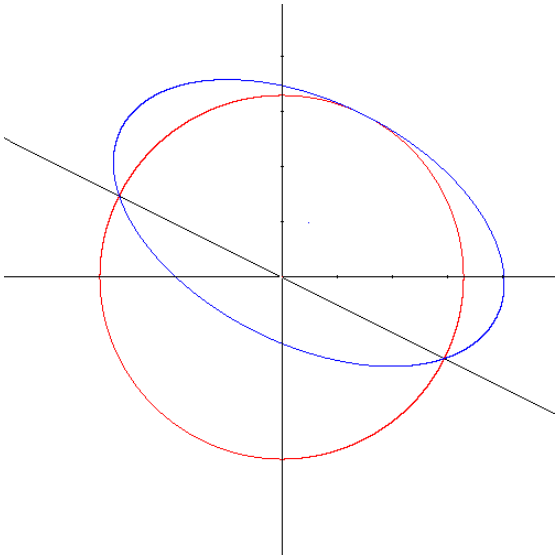
4.2.2 Sistema 2 - Limiti dei double

Si prende ora in considerazione il semplice sistema:

$$\begin{cases} x + 2y \leq 10 \\ -x - 2y \leq -10 \end{cases} \tag{4.2}$$

che ha come insieme di soluzioni la retta $x + 2y = 10$, per valore positivi di x e y. Il sistema ha dimensione $L = 26$, richiede quindi al più 936 passi per

trovare una soluzione. Si riutilizza la classe precedente per eseguire il metodo dell'elissoide 10000 volte con il (4.2) in input. Dal momento che il programma non implementa la perturbazione del sistema ci si aspetta che dopo 936 iterazioni esso esca dicendo che il sistema è impossibile (l'insieme delle soluzioni è una retta, quindi ha volume nullo).
In tutti 10000 i tentativi invece il programma è uscito ben prima, all'iterazione 40, presentando numeri non validi per la matrice e per il centro dell'elisoide (i valori double `Infinity` e `NaN`, infinito e Not a Number). Si riportano qui solo alcuni passaggi per questo problema.



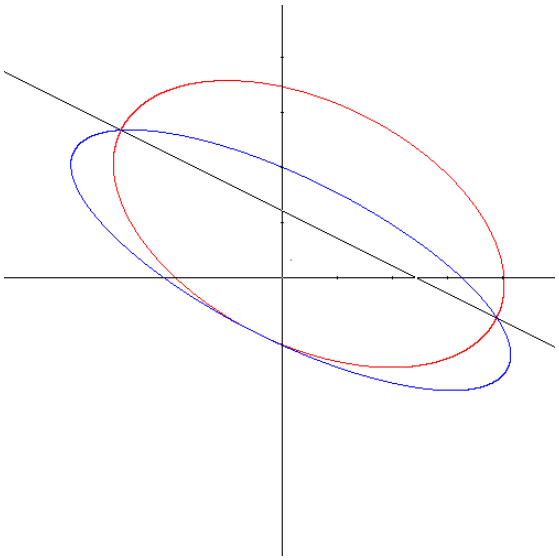
Iterazione 1

$$B_0 = \begin{bmatrix} 4.5 \cdot 10^{15} & 0 \\ 0.0 & 4.5 \cdot 10^{15} \end{bmatrix}$$

$$x_0 = [0, 0]^T$$

$$B_1 = \begin{bmatrix} 5.2 \cdot 10^{15} & -1.6 \cdot 10^{15} \\ -1.6 \cdot 10^{15} & 2.8 \cdot 10^{15} \end{bmatrix}$$

$$x_1 = [10^7, 2 \cdot 10^7]^T$$



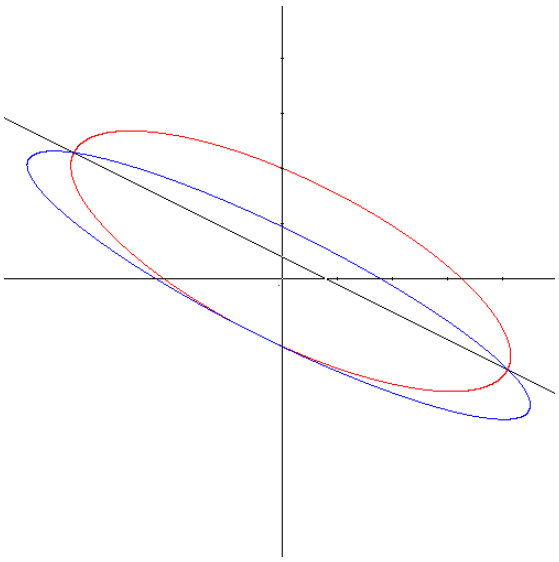
Iterazione 2

$$B_1 = \begin{bmatrix} 5.2 \cdot 10^{15} & -1.6 \cdot 10^{15} \\ -1.6 \cdot 10^{15} & 2.8 \cdot 10^{15} \end{bmatrix}$$

$$x_1 = [10^7, 2 \cdot 10^7]^T$$

$$B_2 = \begin{bmatrix} 6.6 \cdot 10^{15} & -2.8 \cdot 10^{15} \\ -2.8 \cdot 10^{15} & 2.3 \cdot 10^{15} \end{bmatrix}$$

$$x_2 = [3.33 \cdot 10^6, 6.66 \cdot 10^6]^T$$



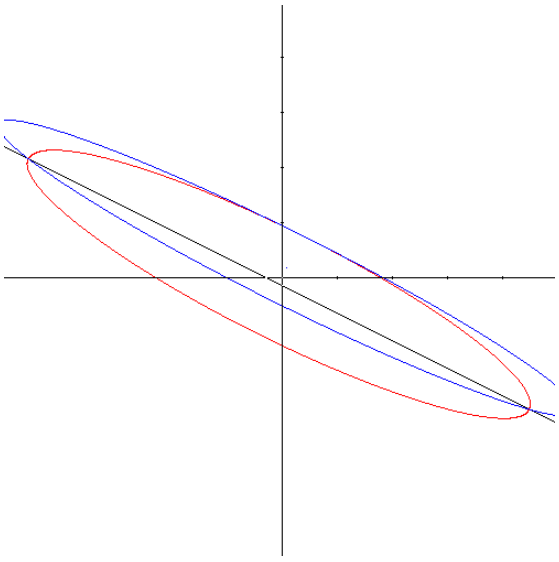
Iterazione 3

$$B_2 = \begin{bmatrix} 6.6 \cdot 10^{15} & -2.8 \cdot 10^{15} \\ -2.8 \cdot 10^{15} & 2.3 \cdot 10^{15} \end{bmatrix}$$

$x_2 = [3.33 \cdot 10^6, 6.66 \cdot 10^6]^T$

$$B_3 = \begin{bmatrix} 8.6 \cdot 10^{15} & -4.1 \cdot 10^{15} \\ -4.1 \cdot 10^{15} & 2.4 \cdot 10^{15} \end{bmatrix}$$

$x_3 = [-1.1 \cdot 10^6, -2.2 \cdot 10^6]^T$



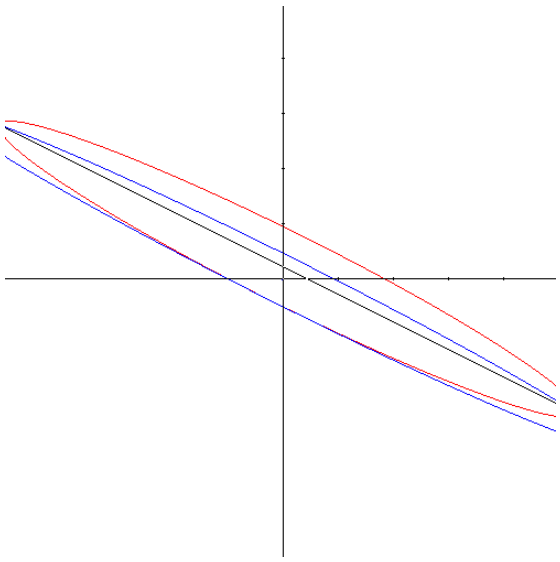
Iterazione 4

$$B_3 = \begin{bmatrix} 8.6 \cdot 10^{15} & -4.1 \cdot 10^{15} \\ -4.1 \cdot 10^{15} & 2.4 \cdot 10^{15} \end{bmatrix}$$

$x_3 = [-1.1 \cdot 10^6, -2.2 \cdot 10^6]^T$

$$B_4 = \begin{bmatrix} 1.1 \cdot 10^{16} & -5.6 \cdot 10^{15} \\ -5.6 \cdot 10^{15} & 3.0 \cdot 10^{15} \end{bmatrix}$$

$x_4 = [1.8 \cdot 10^6, 3.7 \cdot 10^6]^T$



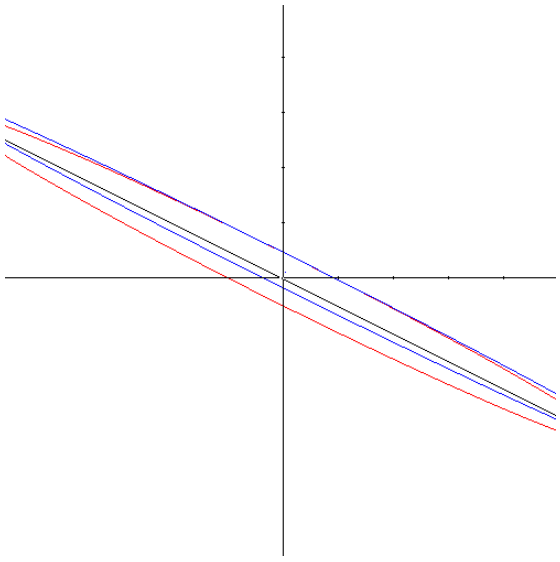
Iterazione 5

$$B_4 = \begin{bmatrix} 1.1 \cdot 10^{16} & -5.6 \cdot 10^{15} \\ -5.6 \cdot 10^{15} & 3.0 \cdot 10^{15} \end{bmatrix}$$

$x_4 = [1.8 \cdot 10^6, 3.7 \cdot 10^6]^T$

$$B_5 = \begin{bmatrix} 1.5 \cdot 10^{16} & -7.5 \cdot 10^{15} \\ -7.5 \cdot 10^{15} & 3.8 \cdot 10^{15} \end{bmatrix}$$

$x_5 = [-1.2 \cdot 10^5, -2.4 \cdot 10^5]^T$



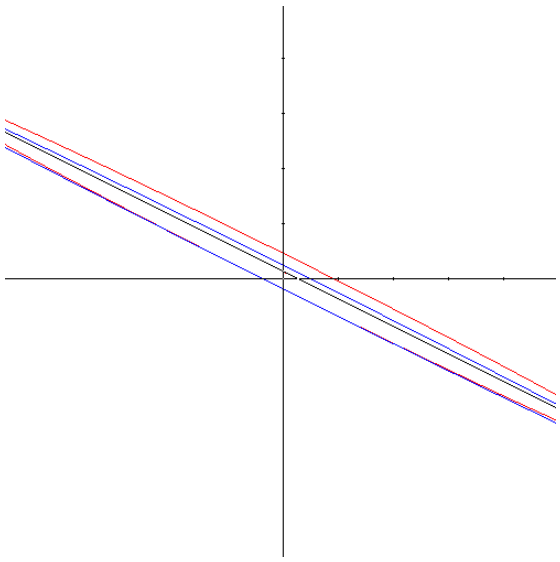
Iterazione 6

$$B_5 = \begin{bmatrix} 1.5 \cdot 10^{16} & -7.5 \cdot 10^{15} \\ -7.5 \cdot 10^{15} & 3.8 \cdot 10^{15} \end{bmatrix}$$

$x_5 = [-1.2 \cdot 10^5, -2.4 \cdot 10^5]^T$

$$B_6 = \begin{bmatrix} 2.0 \cdot 10^{16} & -1.0 \cdot 10^{16} \\ -1.0 \cdot 10^{16} & 5 \cdot 10^{15} \end{bmatrix}$$

$x_6 = [1.2 \cdot 10^6, 2.4 \cdot 10^6]^T$



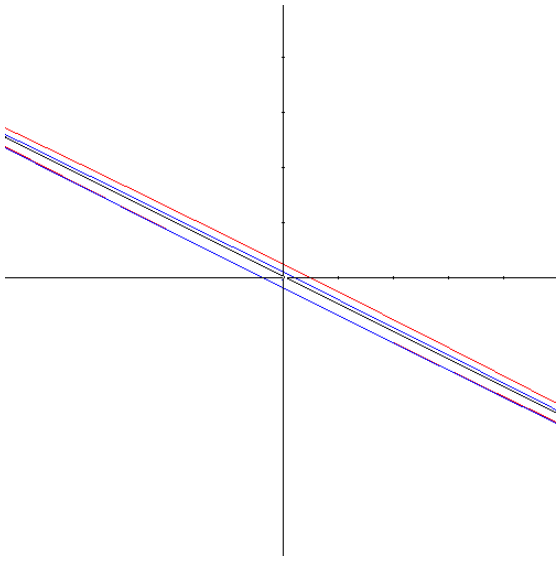
Iterazione 7

$$B_6 = \begin{bmatrix} 2.0 \cdot 10^{16} & -1.0 \cdot 10^{16} \\ -1.0 \cdot 10^{16} & 5 \cdot 10^{15} \end{bmatrix}$$

$x_6 = [1.2 \cdot 10^6, 2.4 \cdot 10^6]^T$

$$B_7 = \begin{bmatrix} 2.7 \cdot 10^{16} & -1.3 \cdot 10^{16} \\ -1.3 \cdot 10^{16} & 6.8 \cdot 10^{15} \end{bmatrix}$$

$x_7 = [3.1 \cdot 10^5, 6.3 \cdot 10^5]^T$



Iterazione 8

$$B_7 = \begin{bmatrix} 2.7 \cdot 10^{16} & -1.3 \cdot 10^{16} \\ -1.3 \cdot 10^{16} & 6.8 \cdot 10^{15} \end{bmatrix}$$

$x_7 = [3.1 \cdot 10^5, 6.3 \cdot 10^5]^T$

$$B_8 = \begin{bmatrix} 3.6 \cdot 10^{16} & -1.8 \cdot 10^{16} \\ -1.8 \cdot 10^{16} & 9 \cdot 10^{15} \end{bmatrix}$$

$x_8 = [-2.7 \cdot 10^5, -5.4 \cdot 10^5]^T$

Questi grafici mostrano come l'eccentricità delle ellissi aumenta ad ogni iterazione, così come il loro asse maggiore. Se un'ellisse giunge ad avere eccentricità $e = 1$, diventa degenere (coincide con due rette parallele) e il determinante della sua B_k è nullo. Questo caso va escluso poichè l'algoritmo genera sempre ellissi valide (ognuna contenente almeno metà della precedente), quindi il fattore critico non può che essere la scarsa precisione dei dati numerici. Se si considerano le ultime matrici stampate in output dal programma prima dell'errore

```
iterazione 32:
   matrice B:
      | 4.782203683053314E19 -2.3911018415266562E19 |
      | -2.3911018415266562E19 1.1955509207633293E19 |
   det:   7.555786372591432E23

iterazione 33:
   matrice B:
      | 6.3762715774044185E19 -3.188135788702209E19 |
      | -3.188135788702209E19 1.5940678943511048E19 |
   det:   4.5334718235548594E23

iterazione 34:
   matrice B:
      | 8.501695436539224E19 -4.250847718269611E19 |
      | -4.250847718269611E19 2.125423859134806E19 |
   det:   9.066943647109719E23

iterazione 35:
   matrice B:
      | 1.1335593915385632E20 -5.667796957692815E19 |
      | -5.667796957692815E19 2.833898478846408E19 |
   det:   1.2089258196146292E24

iterazione 36:
   matrice B:
      | 1.5114125220514175E20 -7.557062610257086E19 |
      | -7.557062610257086E19 3.778531305128544E19 |
   det:   2.4178516392292583E24

iterazione 37:
   matrice B:
      | 2.0152166960685567E20 -1.007608348034278E20 |
      | -1.007608348034278E20 5.038041740171392E19 |
   det:   6.044629098073146E24

iterazione 38:
   matrice B:
      | 2.6869555947580752E20 -1.3434777973790376E20 |
      | -1.3434777973790376E20 6.717388986895188E19 |
   det:   0.0
```

si vede che nelle ultime iterazioni il determinante addirittura cresce anzichè

diminuire e questo è dovuto ai limiti di precisione dei double. Il determinante della matrice B_k è dato da $b_{11}b_{22} - b_{12}b_{21}$; i singoli termini $b_{11}b_{22}$ e $b_{12}b_{21}$ sono dell'ordine di 10^{40}, mentre la loro differenza, e quindi il determinante, è circa dell'ordine di 10^{24}, cioè i termini precedenti hanno le prime 15-16 cifre decimali uguali. Dato che i numeri double mantengono solo circa 15 cifre decimali significative, a causa degli errori di arrotondamento il dato risulta falsato, tanto da risultare addirittura nullo nell'ultima iterazione, quando tutte le cifre significative risultano uguali. Si noti che in base a quanto esposto nella sezione 1.2.1, il semplice sistema preso in esame richiederebbe almeno 2376 bit di precisione per essere al sicuro da questo genere di errori.

Per rendere il programma meno esposto a questo tipo di problemi, si può notare che le ellissi con bassa eccentricità hanno la matrice B_k con il prodotto $b_{12}b_{21}$ molto minore di $b_{11}b_{22}$ (tanto che al limite, nelle circonferenze vale 0, in quanto B_k diventa una matrice diagonale), quindi il determinante soffre meno della scarsa precisione dei dati numerici. Si potrebbe procedere, al momento della scelta di un vincolo violato, con la selezione del vincolo che è più prossimo a essere perpendicolare all'asse maggiore, in modo che l'ellisse generata abbia un'eccentricità preferibilmente minore della precedente. Questa strategia non è comunque sufficiente per dare garanzie di arrivare a una soluzione o a stabilire che il sistema è impossibile. Al sistema preso in considerazione in questa sezione per esempio non si potrebbe applicare e inoltre sarebbe difficilmente estendibile al caso a n dimensioni.

Un altro problema che insorge con l'utilizzo dei double si ha per sistemi che abbiano una dimensione $L \geq 1024$. Infatti questo tipo numerico può gestire numeri fino a circa 2^{1024}, quindi al momento dell'inizializzazione dell'algoritmo, quando si cerca di creare l'elissoide iniziale E_0, il raggio $r = 2^{1024}$ verrebbe interpretato come infinito e non si potrebbe neanche avviare la prima iterazione.

4.2.3 Sistema 3

Il sistema di prima aveva l'insieme delle soluzioni con volume nullo in R^2. Lo stesso errore dovuto a scarsa precisione dei dati numerici si verifica però anche con sistemi che hanno l'insieme delle soluzioni a 2 dimensioni. Si consideri per esempio il sistema

$$\begin{cases} -x + y \leq -10 \\ x - y \leq 14 \\ -x - y \leq -20 \\ x + y \leq 24 \end{cases} \tag{4.3}$$

che ha un insieme di soluzioni evidenziato in figura (4.4). Questo problema ha dimensione $L = 45$, e richiede al più 1620 passi per trovare la soluzione. A questo sistema si applica l'algoritmo generale, con la stessa classe usata prima, per eseguire 10000 tentativi di soluzione. L'esecuzione di questa ha dato 9932 risultati validi, che hanno richiesto tra 167 e 225 iterazioni, con una media di 206. Considerando solo i casi validi, il volume medio dell'elissoide finale è risultato 366631 e il volume tra elissoidi consecutive medio è risultato circa 0.787, ancora

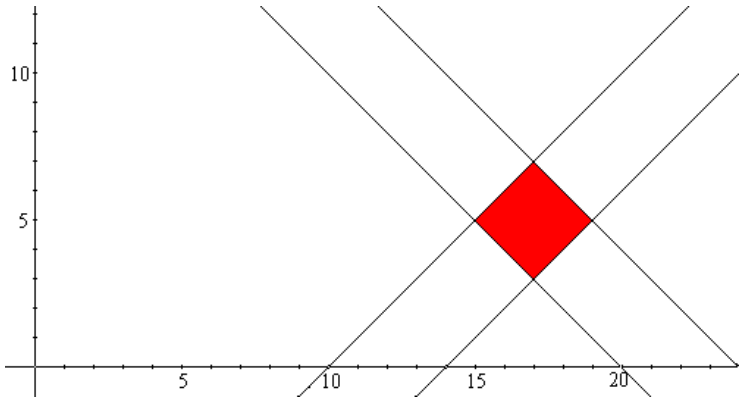


Figura 4.4: Insieme delle soluzioni del sistema (4.3)

una volta migliore del massimo teorico previsto in (1.12). Nella figura (4.5) viene visualizzata la distribuzione dei risultati. In 68 casi l'algoritmo è terminato con un risultato non valido, dovuto sempre ai motivi esposti in precedenza. A

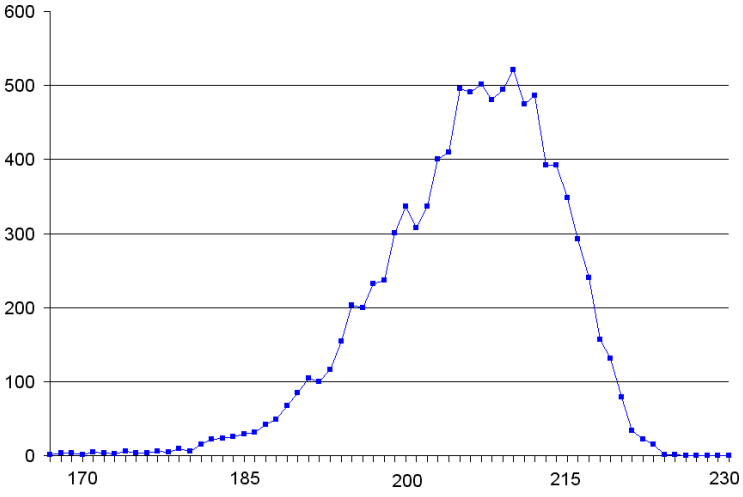


Figura 4.5: Numero di iterazioni per risolvere il sistema (4.3)

differenza dell'esempio precedente se si applica una miglior strategia di scelta dei vincoli, per esempio quella proposta alla fine della sezione precedente, si riuscirebbe a risolvere il sistema nella totalità dei casi.

Appendice A

Sorgenti del programma

Quest'appendice riporta i sorgenti del programma esposto nel Capitolo precedente.

A.1 Classe PianoCartesiano

```
import java.io.FileOutputStream;

public class PianoCartesiano {
   byte[][] p;  // piano cartesiano (fino a 256 colori)
   int xc,yc;   // x centro, y centro
   double ris;

   /** Crea un piano cartesiano di altezza totale h, larghezza w,
     * con ris punti per ogni unita', il primo quadrante ha in
     * tutto altezza h1 e larghezza w1 */

   /* Rappresentazione del piano nell'array p[][].
    *             ^
    *   [h-1,0]   |            [h-1,w-1] = [h1,w1]
    *             |
    *             |
    *             |[xc,yc] = [0,0]
    *   ----------+------------------------------>
    *             |
    *             |
    *             |
    *             |
    *  [0,0]      |            [0,w-1]
    */
   public PianoCartesiano(double h, double w, double ris,
      double h1, double w1) {
      p = new byte[(int)(h*ris)][(int)(w*ris)];
      for (int a=0; a<p.length; a++)   // fondo bianco
```

```
      for (int b=0; b<p[0].length; b++)
         p[a][b] = (byte) 255;

   xc = (int)((h-1-h1)*ris);
   yc = (int)((w-1-w1)*ris);
   this.ris = ris;
}

/** Disegna il pixel corrispondente al punto del piano carte-
  * siano (x,y). Se (x,y)  maggiore delle dimensioni del qua-
  * dro verra' ignorato. */
public void disegnaPunto(double x, double y, int col) {
   int coordx = (int)(xc+x*ris);
   int coordy = (int)(yc+y*ris);

   if (coordx>=0 && coordx<p[0].length && coordy >=0 &&
   coordy<p.length)
      p[coordy][coordx] = (byte)col;
}


/** Disegna una retta ax + by + c = 0. */
public void disegnaRetta(double a, double b, double c,
      int col) {
   double x = 1.0*(-xc)/ris;
   for (int xi=0; xi<p[0].length; xi++, x=x+1.0/ris) {
      disegnaPunto(x,(-a*x-c)/b,col);
   }

   double y = 1.0*(-yc)/ris;
   for (int yi=0; yi<p[0].length; yi++, y=y+1.0/ris) {
      disegnaPunto((-b*y-c)/a,y,col);
   }
}

/** Accetta anche array di coeff per disegnare ellisse. */
public void disegnaConica(double[] C, int col) {
   if (C.length!=6)
      throw new IllegalArgumentException("Una conica ha 6 " +
                "coefficienti.");
   disegnaConica(C[0],C[1],C[2],C[3],C[4],C[5],col);
}

/** Disegna l'ellisse ax^2 + bxy + cy^2 + dx + ey + f = 0. */
public void disegnaConica(double a, double b, double c,
   double d, double e, double f, int col) {
   // brute force sull'asse x di tutte le eq di 2 grado
   double x = 1.0*(-xc)/ris;
   for (int xi=0; xi<p[0].length; xi++, x=x+1.0/ris) {
```

```
      double[] pti = risolviEq2gr(c, b*x+e, a*x*x+d*x+f);

      if (pti==null) continue;
      else {
         disegnaPunto(x,pti[0],col);
         disegnaPunto(x,pti[1],col);
      }
   }

   // brute force lungo asse y (per definire anche gli estremi)
   double y = 1.0*(-yc)/ris;
   for (int yi=0; yi<p[0].length; yi++, y=y+1.0/ris) {

      double[] pti = risolviEq2gr(a, b*y+d, c*y*y+e*y+f);

      if (pti==null) continue;
      else {
         disegnaPunto(pti[0],y,col);
         disegnaPunto(pti[1],y,col);
      }
   }
}

/** Risolve un'eq di 2 grado (null se impossibile). */
private static double[] risolviEq2gr(double a,double b,
    double c) {
   double delta = b*b-4*a*c;
   if (delta<0) return null;

   delta = Math.sqrt(delta);
   double[] ris = new double[2];
   ris[0] = (-b + delta)/(2*a);
   ris[1] = (-b - delta)/(2*a);

   return ris;
}

/** Salva il piano in una bitmap a 8 bit per pixel (256 colori).
  * Visualizza sugli assi le unita' di misura ogni [unitaMis].*/
public void salva8bit(String file, double unitaMis) {
   try {
      FileOutputStream fo = new FileOutputStream(file);
      fo.write(getHeader(8));      // salva su file l'header

      // disegna assi/ unita' di misura
      drawRemoveAxis(true, unitaMis);

      int a,b,c,d;
      for (a=0; a<p.length; a++) {   // salva su file il piano
         for (b=0; b<p[a].length; b++)
```

```
            fo.write(p[a][b]);

         c = p[a].length%4;
         if (c==0) c+=4;
         for (d=4; d>c; d--)
            fo.write(0);
      }

      // tolgp assi/ unita' di misura
      drawRemoveAxis(true, unitaMis);

   }
   catch (Exception e) {
      System.err.println("Errore in salvataggio: "+e);
   }
}

public void cleanPlane() {
   p = new byte[p.length][p[0].length];

   for (int i=0; i<p.length; i++) {
      for (int j=0; j<p[0].length; j++) {
         p[i][j] = (byte)255;
      }
   }
}

/** Aggiunge/rimuove gli assi e le unita' di misura */
private void drawRemoveAxis(boolean assi, double unMis) {
   if (assi)  {
      for (int a=0; a<p[0].length; a++)
         p[yc][a] = (byte)(255 - p[yc][a]);
      for (int a=0; a<p.length; a++)
         p[a][xc] = (byte)(255 - p[a][xc]);
   }

   if (unMis>0) {
      int t = (int)(ris*unMis);
      for (int a=t+xc; a<p[0].length; a+=t) {
         p[yc+1][a] = (byte)(255 - p[yc+1][a]);
         p[yc-1][a] = (byte)(255 - p[yc-1][a]);
      }
      for (int a=t+yc; a<p.length; a+=t) {
         p[a][xc+1] = (byte)(255 - p[a][xc+1]);
         p[a][xc-1] = (byte)(255 - p[a][xc-1]);
      }
   }
}

/** Ritorna l'header della bitmap a nbits bit; gestisce solo il
```

```
     * caso a 8 bit */
   private byte[] getHeader(int nbits) {
      byte[] h = null;
      int dim=0,dimdata=0;

      if (nbits == 8) {
         h = new byte[54+1024];

         dim = p.length*p[0].length+54+1024;
         if (p[0].length%4!=0) dim+=(4-p[0].length%4)*p.length;
         dimdata = dim-54-1024;
         h[11] = 4;   // data inizia a fine palette, 1024+54

         //un po' di colori
         h[54]=h[55]=h[56]=0;   // 0 - nero
         h[58]=h[59]=0; h[60]=(byte)255;   // 1 - rosso
         h[62]=(byte)255; h[63]=h[64]=0;   // 2 - blu
         h[67]=(byte)255; h[66]=h[68]=0;   // 3 - verde
         h[1074]=h[1075]=h[1076]=(byte)255;   // 255 - bianco
      }

      h[0]='B'; h[1]='M';
      h[2]= (byte)(dim%256);
      h[3]= (byte)((dim/256)%256);
      h[4]= (byte)(dim/65536);
      h[10] = (byte)54; h[14] = (byte)40;
      h[18] = (byte)(p[0].length%256);
      h[19] = (byte)(p[0].length/256);
      h[22] = (byte)(p.length%256);
      h[23] = (byte)(p.length/256);
      h[26]=1; h[28]=(byte)nbits;
      h[34]= (byte)(dim%256);
      h[35]= (byte)((dim/256)%256);
      h[36]= (byte)(dim/65536);
      return h;
   }
}
```

A.2 Classe Interfaccia

```
import java.util.Scanner;
import java.io.FileInputStream;

public class Interfaccia {
   public static void main(String[] args) {
      double[][] At;
      double[] b;
      try {
         Scanner s = new Scanner(new FileInputStream("input.txt"));
```

```
         int t = s.nextInt();
         //At x <= b
         At = new double[t][2];
         b = new double[t];

         for (int a=0; a<t; a++) {
            At[a][0] = s.nextDouble();
            At[a][1] = s.nextDouble();
            b[a] = s.nextDouble();
         }

     // visualizzazione del grafico dell'insieme delle soluzioni
     // (va riconfigurata manualmente ad ogni utilizzo)
         PianoCartesiano pc = new PianoCartesiano(25,25,20,23,23);

         for (int a=0; a<t; a++)
            pc.disegnaRetta(At[a][0],At[a][1],-b[a],0);
         pc.salva8bit("InsiemeSoluzioni.bmp",1);

         EllipsoidMethod em = new EllipsoidMethod(At,b);
     // cambia raggio iniziale se ne viene specificato uno
         double[][] newB = new double[2][2];
         if (args.length == 1) {
            try {
               double d = Double.parseDouble(args[0]);
               newB[0][0] = newB[1][1] = d*d;
               em.B = newB;
            } catch (Exception exc) {
               System.out.println("Raggio non valido, verra' ignorato.");
            }
         }
         em.solve();
      }
      catch(Exception e) { }
   }
}
```

A.3 Classe MetodoElissoide

```
/** Esempio di implementazione dell'algoritmo dell'elissoide per
  * la risoluzione di sistemi di disequazioni lineari.
  * Risolve solo sistemi a due incognite. */

public class EllipsoidMethod {
   double[][] At;   // i vettori del sistema At x <= b
   double[] b;
   double[][] B, Binv;   // matrice dell'elissoide
   double[] Ek;   // conica corrispondente all'elissoide da disegnare
```

```
double[] xk;   // centro dell'elissoide

int n;      // numero di dimensioni
int maxIteration;
int L;      // dimensioni del problema.in bit
double tau, omega, delta;   // parametri delle trasformazioni

public EllipsoidMethod(double[][] At, double[] b) {
   if (At[0].length!=2 || At.length!=b.length) {
      System.err.println("Array non validi");
      System.exit(1);
   }

   this.At = At;
   this.b = b;

   n=At[0].length;      // deve essere a 2 variabili...
   tau = 1.0/(n+1);
   omega = 2.0/(n+1);
   delta = 1.0*n*n/(n*n-1);
   System.out.println("n="+n+"\ntau="+tau+"\nomega="+omega+
       "\ndelta="+delta);

   L = 0;   // determino dimensioni problema
   double log2 = Math.log(2);
   for (int i=0; i<At.length; i++) {
      for (int j=0; j<At[i].length; j++) {
         if (Math.abs(At[i][j])!=0)
            L += (int)(Math.log(Math.abs(At[i][j]))/log2);
      }
   }

   for (int i=0; i<b.length; i++) {
      if (Math.abs(b[i])!=0)
         L += (int)(Math.log(Math.abs(b[i]))/log2);
   }

   L += (int)(Math.log(n)/log2 + Math.log(b.length)/log2);
   L += 2*b.length*(n+1)+4;
   System.out.println("L = "+L);

   B = new double[2][2];
   // E0  una ball con r=2^L, xk=(0,0)
   B[0][0] = B[1][1] = Math.pow(2,2*L);
   xk = new double[2];
   xk[0]=xk[1]=0;


   maxIteration = 6*n*(n+1)*L;
   System.out.println("Massimo numero di passi:"+maxIteration);
```

```
}

/** Ritorna una soluzione di At x <= b, null se impossibile */
public double[] solve() {
   PianoCartesianoDouble pc = null;
   double[] par = null;
   boolean end = false;

   Binv = matriceInversa(B);
   Ek = getEllisse(Binv,xk);

   for (int a=0; a<maxIteration && !end; a++) {
    // reinquadra il piano cartesiano ogni 15 passaggi
      if (a%15==0) {
         if (a==0) {
            par = getParametriPiano();
            System.out.println("====>Reinquadra piano: " +
            par[0]+" "+par[1]+" "+par[2]+" "+par[3]+" "+par[4]);
            pc = new PianoCartesianoDouble(par[0],par[1],par[2],
                 par[3],par[4]);
         }
   // evita problemi quando centro ha valori negativi
         else { // mediamente funziona
            for (int i=0; i<5; i++) par[i]/=7;
            par[2] = 500/par[0];
            pc = new PianoCartesianoDouble(par[0],par[1],
                 par[2],par[3],par[4]);
         }
      }

      informazioni(a,Math.max(par[0],par[1])/10);

      pc.disegnaConica(Ek,1);
      pc.disegnaPunto(xk[0],xk[1],1);

      int[] vv;
      if ((vv = vincoliViolati())!=null) {
         double x1=xk[0], x2=xk[1];
         int v = aggiornaEllisse(vv);
         double axk = At[vv[v]][0]*x1+At[vv[v]][1]*x2;
         Binv = matriceInversa(B);
         Ek = getEllisse(Binv,xk);

         informazioni(a,Math.max(par[0],par[1])/10);
         pc.disegnaRetta(At[vv[v]][0],At[vv[v]][1],-axk,0);
         pc.disegnaConica(Ek,2);
         pc.disegnaPunto(xk[0],xk[1],2);
      }
      else {
         System.out.println("SOLUZIONE: x = ("+xk[0]+", "+
```

```
             xk[1]+")");
         for (int i =0; i<At.length; i++)
            pc.disegnaRetta(At[i][0],At[i][1],-b[i],0);
         pc.disegnaPunto(xk[0],xk[1],2);

         end = true;
      }

      pc.salva8bit("immagine"+a+".bmp",Math.max(par[0],
             par[1])/10);
      pc.cleanPlane();
   }

   return null;
}

/** Ritorna un array col numero dei vincoli violati da xk,
  * centro dell'ellisse */
private int[] vincoliViolati() {
   boolean[] v = new boolean[b.length];
   for (int i=0; i<b.length; i++)
      if (At[i][0]*xk[0]+At[i][1]*xk[1]>b[i]) v[i]=true;

   int count = 0;
   for (int a=0; a<b.length; a++)
      if (v[a]) count++;

   if (count==0) return null;     // trovata una soluzione

   int[] ris = new int[count];
   count=0;

   for (int a=0; a<b.length; a++)
      if (v[a]) ris[count++]=a;

   return ris;
}

/** Passa da E(k) a E(k+1) - ritorna vincolo usato per taglio.
  * - sceglie un vincolo violato a caso
  * - usa il taglio a x < a xk (no deep cut) */
private int aggiornaEllisse(int[] vv) {
   int vvs = (int)(Math.random()*vv.length);
   double[] a = At[vv[vvs]];
   System.out.println("vv:"+vvs);

   // calcolo Bk a at Bk
   double[] Ba = new double[2];
   Ba[0] =B[0][0]*a[0] + B[0][1]*a[1];
   Ba[1] =B[1][0]*a[0] + B[1][1]*a[1];
```

```
    double[] aB = new double[2];
    aB[0] =B[0][0]*a[0] + B[1][0]*a[1];
    aB[1] =B[0][1]*a[0] + B[1][1]*a[1];

    double[][] BaaB = new double[2][2];
    BaaB[0][0] = Ba[0]*aB[0];
    BaaB[0][1] = Ba[0]*aB[1];
    BaaB[1][0] = Ba[1]*aB[0];
    BaaB[1][1] = Ba[1]*aB[1];

    // calcolo at Bk a
    double aBa=a[0]*(a[0]*B[0][0]+a[1]*B[1][0]) +
         a[1]*(a[0]*B[0][1]+a[1]*B[1][1]);

    // aggiorno le xk,yk
    for (int i = 0; i<2; i++) {
       xk[i] = xk[i] - tau*(Ba[i])/Math.sqrt(aBa);
    }

    // aggiorno B
    for (int i=0; i<2; i++) {
       for (int j=0; j<2; j++) {
          B[i][j] = delta*(B[i][j] - omega*BaaB[i][j]/aBa);
       }
    }

    return vvs;
 }

 /** Ritorna i parametri che deve avere il piano cartesiano */
 // raddoppia la distanza dal centro finche' non arriva in un
 // punto dove non c'e' piu' la conica, quindi usa una
 // ricerca dicotomica per individuare di preciso dov'e' il
 // limite di estensione della stessa.
 // funziona per la prima iterazione, con le altre ha problemi,
 // e' meglio riscriverlo
 private double[] getParametriPiano() {
    double[] ris = new double[5];

    double prevx=0,prevy=0,temp;
    int count=0;
    double[] rad;
    double x,y;

// trovo il punto piu' lontano dove si estende la conica lungo x
    for (x=xk[0]; ; count++) {
       rad = PianoCartesianoDouble.risolviEq2gr(Ek[2],
             Ek[1]*x+Ek[4], Ek[0]*x*x+Ek[3]*x+Ek[5]);
```

```
      if (rad==null) {
         temp = x;
         x = (prevx+x)/2;
         prevx = x;
      }
      else {
         prevx=x;
         x=(x+1)*2;
      }

      if (count>100 || (rad!=null && rad[0]==rad[1]) ||
          (count>5 && Math.abs(x-prevx)<0.5))
         break;
   }

   count=0;
// trovo il punto piu' lontano dove si estende la conica lungo y
   for (y=xk[1]; ; count++) {
      rad = PianoCartesianoDouble.risolviEq2gr(Ek[0],
                Ek[1]*y+Ek[3], Ek[2]*y*y+Ek[4]*y+Ek[5]);

      if (rad==null) {
         temp = y;
         y = (prevy+y)/2;
         prevy = y;
      }
      else {
         prevy=y;
         y=(y+1)*2;
      }

      if (count>100 || (rad!=null && rad[0]==rad[1]) ||
          (count>5 && Math.abs(y-prevy)<0.5))
         break;
   }

   ris[0] = 2*(x-xk[0])*1.01;   // doppio di larghezza/altezza
   ris[1] = 2*(y-xk[1])*1.01;
   ris[2] = 500/Math.max(ris[0],ris[1]); // immagini 500 pixel
   ris[3] = (x-xk[0]);          // immagine centrata
   ris[4] = (x-xk[0]);
   return ris;
}

private void informazioni(int iterazione, double ris) {
   System.out.println("iterazione "+iterazione+":");
   System.out.println("\tcentro ellisse: xk=("+xk[0]+", "+
           xk[1]+")\n\tmatrice B:");
   printMatrix(B);
   System.out.println("\tdelta: "+
```

```
            (B[0][0]*B[1][1]-B[0][1]*B[1][0]) );
   System.out.println("matrice B inversa:");
   printMatrix(Binv);
   System.out.println("\tconica: "+Ek[0]+" "+Ek[1]+" "+Ek[2]+
           " "+Ek[3]+" "+Ek[4]+" "+Ek[5]);
   System.out.println("\tunita' misura grafico: "+ris);
}

/** Ritorna una conica nella forma ax^2 + ... + f = 0 a partire
  * da un'elissoide.
  *  (x-xc)T Minv (x-xc) <= 1 */
private double[] getEllisse(double[][] Minv, double[] xc) {
   double[] ris = new double[6];

   ris[0] = Minv[0][0];      // x^2
   ris[1] = Minv[1][0]+Minv[0][1];   // xy
   ris[2] = Minv[1][1];   // y^2
   ris[3] = -2*Minv[0][0]*xc[0] - ris[1]*xc[1];   // x
   ris[4] = -ris[1]*xc[0] - 2*Minv[1][1]*xc[1];    // y
   ris[5] = Minv[0][0]*xc[0]*xc[0] + ris[1]*xc[0]*xc[1] +
         Minv[1][1]*xc[1]*xc[1] - 1.0; // f

   return ris;
}

/** Inverte la matrice M */
private double[][] matriceInversa(double[][] M) {
   if (M.length!=2 || M[0].length!=2)
      throw new IllegalArgumentException("Inverte solo" +
               " matrici 2x2.");
   double delta = M[0][0]*M[1][1] - M[0][1]*M[1][0];

   double[][] ris = new double[2][2];
   ris[0][0] = M[1][1]/delta;
   ris[0][1] = -M[1][0]/delta;
   ris[1][0] = -M[0][1]/delta;
   ris[1][1] = M[0][0]/delta;

   return ris;
}

/** Stampa una matrice */
private void printMatrix(double[][] m) {
   for (int i=0; i<m.length; i++) {
      System.out.print("|");
      for (int j=0; j<m[i].length; j++) {
         System.out.print(" "+m[i][j]);
      }
      System.out.print(" |\n");
   }
```

```
    }
}
```

Bibliografia

[1] L. G. Khachiyan: *A Polynomial Algorithm in Linear Programming*
Il paper originale di Khachiyan sulla polinomialità del metodo dell'elissoide in Programmazione Lineare. Non è stato possibile consultarlo direttamente.

[2] R. G. Bland, D. Goldfarb, M. J. Todd: *The Ellipsoid Method: A Survey*, Operations Research, Vol. 29, No. 6, (Nov. - Dec. , 1981), pp. 1039-1091
Ottimo paper dell'Agosto '80; fornisce una panoramica completa dell'algoritmo dell'elissoide. Tratta dell'argomento da molti punti di vista: da quello storico (l'evoluzione di altri algoritmi che hanno portato alla nascita di questo), a quello matematico, dai problemi delle implementazioni reali, ai legami che presenta con i metodi classici. Il livello di approfondimento matematico nel testo è medio-basso, si tende a citare i risultati e darne un'interpretazione intuitiva più che a giustificarli rigorosamente; questo stile rende il testo ben comprensibile anche disponendo solo di conoscenze di base riguardanti l'argomento.
Comunque nelle appendici finali vengono riportate alcune dimostrazioni importanti, con un livello di rigorosità matematica maggiore rispetto al resto del paper.

[3] M. Grötschel, L. Lovasz, A. Schrijer: *The Ellipsoid Method and its Consequences in Combinatorial Optimization*, Combinatorica, No. 2, (1981), pp. 169-197
Questo paper tratta del metodo dell'elissoide e della sua applicazione in problemi di ottimizzazione combinatoria. Il livello di approfondimento richiede di avere conoscenze vaste e solide sull'argomento per poter comprendere i contenuti. Non ha uno stile discorsivo, eccetto per le introduzioni dei vari capitoli, che contestualizzano le dimostrazioni riportate poi nel corpo. Queste dimostrazioni sono molto rigorose e richiedono conoscenze matematiche anche in altri campi per poter seguire tutti i passaggi.
La lettura di un paper di questo tipo può essere utile come approfondimento sull'argomento, una volta che si hanno già conoscenze adeguate.

[4] D. Goldfarb, M. J. Todd: *Modifications and Implementation of the Shor-Khachiyan Algorithm for Linear Programming*, Technical Report 406, Department of Computer Science, Cornell University, Ithaca, N.Y.

Paper consigliato da [2] per approfondimenti su tagli surrogati generati combinando vincoli linearmente indipendenti.

[5] Y. Krol, B. Mirman: *Some Practical Modifications Of Ellipsoid Method for LP Problems* Arcon, Inc., Boston.
Paper consigliato da [2] per approfondimenti su tagli surrogati generati da due vincoli.

[6] S. Boyd, L. Vandenberghe: *Convex Optimization*, Cambridge University Press
Testo riguardante l'ottimizzazione convessa utilizzato presso l'Università inglese di Stanford. Fornisce una trattazione completa dell'argomento, dalle basi alle tecniche di risoluzione oggi note. Non descrive il metodo dell'elissoide direttamente, ma i metodi da esso derivati. Una nota molto importante riguardo questo libro è la sua disponibilità in internet per scopi didattici.

[7] S. Vempala: *Lecture The Ellipsoid Algorithm*, Internet
Questa lettura riporta una dimostrazione semplificata della polinomialità dell'algoritmo dell'elissoide. Nonostante non sia molto rigorosa, ha il grande pregio di esplicitare tutti i passaggi, ciò permette di comprendere chiaramente la tesi che vuole dimostrare.

[8] M. Fischetti: *Lezioni di Ricerca Operativa*, Edizioni Libreria Progetto Padova
Testo di riferimento per le nozioni di base richieste dalla tesina sugli argomenti di programmazione convessa e lineare.

Elenco delle tabelle

Elenco delle figure