Università degli Studi di Parma

DIPARTIMENTO DI MATEMATICA E INFORMATICA
Corso di Laurea in Informatica

Studio e Sperimentazione dell'Uso della Forma di Bernstein per Problemi di Soddisfacimento di Vincoli

Study and Experimentation
of the Bernstein Form
for Constraint Satisfaction Problems

Candidato:
Amerigo Mancino

Relatore:
Prof. Federico Bergenti

Anno Accademico 2014–2015

A nonno Ettore,
fonte d'ispirazione e
maestro di vita.

Ringraziamenti

I ringraziamenti sono una cosa facile solo apparentemente. Sicuramente questo lavoro non sarebbe stato possibile senza la guida del mio relatore, *Federico Bergenti*, che ha seguito da vicino la mia attività. Un sincero ringraziamento va poi anche alle professoresse *Alessandra Aimi*, *Chiara Guardasoni*, *Stefania Monica* e al professor *Francesco Di Renzo* per i preziosi spunti e consigli dati in corso d'opera oltre che, in generale, a tutti i professori del Dipartimento di Matematica e Informatica per essere una inesauribile fonte d'ispirazione. Grazie ai *miei genitori* per avermi supportato economicamente e spiritualmente lungo questo percorso, senza aver mai perso la speranza o la fiducia, anche in situazioni inaspettate o difficoltose. Grazie a mio fratello *Walter* perché forse il fatto di essere stati separati in una città nuova ci ha reso più uniti che mai. Grazie agli *aquilotti* per essere stati interminabile sostegno e forza. Grazie alle mie amiche del *Pam Pam Pam*, *Silvia* e *Andia*, per essere le migliori complici che si possano desiderare. Grazie ad *Andrea Segalini*, *Chesterino*, *Ayoub (il Signore Oscuro)*, *Marco Grillo*, *Parra* lo spammer, *Benini*, *Schianchi* e, in generale, grazie a tutti i *compagni di corso* di questi anni per essere stati gli amici di studio e di vita che questo piccolo frammento di tempo ha scelto, in un crocevia di esistenze che non potranno mai essere davvero separate. Grazie a *Gloria Mastrodomenico*, la più simpatica non-informatica che ho conosciuto in questo arco parmense e che è stata capace di distrarmi e farmi sorridere anche durante la stesura della tesi. Grazie a *Valeria Savazzi*, *Giulia Menegardi*, *Elisa Violante* e a tutti gli amici del gruppo *Uno come noi, noi come tutti* e dell'associazione *Ottavo Colore* per le magnifiche esperienze che mi hanno permesso di vivere e per lottare con forza a favore dei diritti per le persone LGBT. Grazie ad *Alessandro Deminath*, per la vita condivisa. Grazie a *Carmen* e *Samira* per essere un pilastro fondamentale della mia esistenza. Grazie ai miei rismondini *Sapiens*, *Assal* e *Gelsomina* per essere stati, in un qualche senso, una famiglia indimenticabile. E grazie infine alla mia determinazione e alla mia forza di volontà, che mi porteranno (ne sono sicuro) verso traguardi sempre maggiori.

Indice

Introduzione

I problemi di soddisfacimento di vincoli, o CSP (Constraint Satisfaction Problem), sono un'idea risalente agli anni '80. Alla fine degli anni '90 si pensava potessero addirittura rivoluzionare il mondo della programmazione. La storia ci racconta che questo non è avvenuto, ma ciononostante i CSP hanno trovato numerose applicazioni pratiche, in particolar modo nel mondo dell'Intelligenza Artificiale e della Ricerca Operativa, reclamando di conseguenza la propria fetta di ricercatori. Si tratta, infatti, di problemi abbastanza generali da risultare interessanti, ma anche abbastanza specifici da ammettere algoritmi efficaci.

Questa trattazione si pone come obiettivo la ricerca di algoritmi di consistenza per la riduzione di domini in vincoli polinomiali. Nel Capitolo 1 formalizzeremo la definizione di CSP, mostrando esempi pratici classici di applicazione, come il problema delle 8 regine o semplici problemi di criptoaritmetica, quale SEND + MORE = MONEY, per la cui risoluzione viene proposta una procedura sviluppata in SWI-Prolog [7]. Nel Capitolo 2 introdurremo i polinomi nella base di Bernstein, una particolare classe di polinomi sul campo reale che risulterà d'aiuto nella formulazione di alcuni algoritmi di consistenza. Saranno presentate e dimostrate le principali proprietà e verranno descritti e implementati alcuni metodi di conversione da forma di potenze a forma di Bernstein. Nel Capitolo 3 esibiremo i diversi algoritmi di riduzione dei domini che sono stati studiati e implementati in linguaggio MATLAB, a partire dall'idea di sfruttare le trasformazioni affini per i polinomi, fino ad arrivare a tecniche più complesse, ma decisamente valide. Concluderemo la trattazione designando le conclusioni ottenute a fine lavoro e i possibili sviluppi futuri di questo progetto. Infine, esporremo nell'Appendice A un'implementazione Java di diversi algoritmi realizzati nel Capitolo 3, sfruttando JAMA (JAva MAtrix package) [4], un package contenete la messa a punto delle funzioni dell'algebra lineare di base.

Capitolo 1

Problemi di soddisfacimento di vincoli

Molti problemi nell'ambito dell'Intelligenza Artificiale e della Ricerca Operativa sono classificabili come problemi di soddisfacimento di vincoli (Constraint Satisfaction Problems o CSP) e vengono usati per risolvere casi di allocazione di risorse, realizzazione di negozi virtuali, gestione e configurazione di reti di telecomunicazioni, problemi di complessità combinatoria, ragionamento temporale, ma anche progettazione di circuiti integrati digitali. I CSP infatti risultano:

- abbastanza generali da essere interessanti;
- abbastanza specifici da ammettere algoritmi efficaci.

In questo capitolo presenteremo una completa descrizione dei CSP e delle loro caratteristiche, nonché alcuni esempi d'uso.

1.1 Definizione formale

Introduciamo la trattazione presentando la seguente

Definizione 1.1 (CSP)**.** Un problema di soddisfacimento di vincoli o CSP è definito come:

1. un insieme (finito) di variabili $V = \{x_i\}$, con $i = 0, \ldots, n$;
2. un insieme (finito) di domini $dom(x_i)$, uno per ogni variabile;
3. un insieme (finito) di vincoli, relazioni definite tra variabili.

Un vincolo può essere visto quindi come una restrizione dei valori che le variabili possono assumere simultaneamente e può essere classificato in tre tipologie differenti:

- vincolo unario, coinvolge una sola variabile;
- vincolo binario, coinvolge due variabili;
- vincolo globale, coinvolge più di due variabili.

Ogni vincolo globale può essere ricondotto ad un insieme di vincoli binari, aumentando tuttavia in questo modo il numero di vincoli del problema. La scelta se compiere o meno questa trasformazione dipende dai casi.

Definizione 1.2 (Soluzione di un CSP)**.** Una soluzione (o goal) di un CSP è un assegnamento completo (le variabili hanno tutte un valore) e consistente (tutti i vincoli sono rispettati).

Un CSP può avere nessuna soluzione, una soluzione o infinite soluzioni. In questa trattazione non ci interesseremo alle soluzioni in sé ma ad opportune restrizioni dei domini delle variabili in modo tale che tali restrizioni contengano almeno una soluzione. Per poter raggiungere e identificare una soluzione devono essere compiute una serie di azioni, ovvero assegnamenti di valori a variabili.

Definizione 1.3 (Stato)**.** Uno stato in un CSP è un assegnamento (potenzialmente parziale e/o inconsistente) di valori v_i alle variabili x_i, dove ogni v_i appartiene al dominio di ogni rispettiva x_i:

$$\{x_1 = v_1,\ x_2 = v_2,\ \ldots,\ x_k = v_k\}$$

I CSP sono classificabili ulteriormente in due categorie:

- A domini discreti

  Domini finiti, naturali, interi, stringhe, etc.

- A domini continui

  Reali, complessi, insiemi di funzioni polinomiali, etc.

In questa trattazione ci concentreremo principalmente su CSP a domini continui.

Per un CSP a soli vincoli binari (o unari) è possibile definire un Grafo dei Vincoli, ossia un grafo non orientato in cui:

- Le variabili costituiscono i nodi del grafo;
- Gli archi del grafo raffigurano i vincoli.

Esiste un arco fra due variabili se e solo se esiste un vincolo che le coinvolge entrambe. Esiste un arco in sé stesso se il vincolo è unario.

1.2 Il problema delle 8 regine

Forniamo in questo paragrafo un caso di formalizzazione di un CSP usando un esempio classico per il genere, ossia il problema delle 8 regine. Detto in termini semplici, occorre posizionare su una scacchiera di dimensioni 8×8 tutte le 8 regine facendo in modo che nessuna di esse possa catturarne un'altra.

Per formalizzare il problema dobbiamo identificare un insieme di variabili, un insieme di domini e un insieme di vincoli. Poiché su ogni riga e su ogni colonna della scacchiera dovrà esserci esattamente una regina, una possibile formalizzazione del problema è quella di considerare come variabili ciascuna delle 8 righe sulla scacchiera. L'insieme delle variabili dunque sarà:

$$V = \{R_1, R_2, \ldots, R_8\}$$

Ciascuna di queste variabili può assumere un valore che rappresenta la colonna su cui si trova la regina corrispondente, per cui i domini della variabili sono le 8 possibili colonne:

$$dom(R_1) = dom(R_2) = \cdots = dom(R_8) = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

Abbiamo già inserito in questa formalizzazione il fatto che due regine non possono trovarsi sulla stessa riga; resta da imporre che non possono trovarsi sulla stessa colonna o diagonale. L'insieme dei vincoli allora sarà:

$$\begin{cases} \forall\ i, j\ R_i \neq R_j \\ \forall\ i, j \text{ se } R_i = a \text{ e } R_j = b \text{ allora } i - j \neq a - b \text{ e } i - j \neq b - a \end{cases}$$

Con la prima equazione imponiamo, infatti, a due regine di non poter occupare la stessa colonna mentre con la seconda equazione imponiamo loro di non potersi trovare sulla stessa diagonale.

1.3 Tecniche per la risoluzione di CSP

Nonostante in questo testo ci dedicheremo principalmente ad algoritmi di consistenza e quindi al restringimento dei domini delle variabili coinvolte in vincoli polinomiali, è opportuno citare anche le numerose tecniche che permettono di risolvere problemi di soddisfacimento di vincoli. Fra queste ricordiamo algoritmi che valutano i vincoli a posteriori (backtracking, backjumping, backmarking), algoritmi che valutano i vincoli a priori (forward checking, partial look-ahead, full look-ahead) e algoritmi che permettono di semplificare il problema (arc-consistenza, path-consistenza).

Un caso interessante è poi la presenza di diversi linguaggi atti a descrivere e poi risolvere CSP. Uno di questi è MiniZinc [8], che introduce un formalismo svincolato dal risolutore e abbastanza generale da essere utilizzabile. MiniZinc di per sé quindi non risolve CSP ma si appoggia ad un solver (di default o implementato dall'utente) che compie il lavoro di risoluzione.

Un'altra alternativa valida è la Constraint Logic Programming (CLP), ossia un'estensione della programmazione logica, parametrica rispetto al dominio delle variabili. Un linguaggio che permette di realizzare CLP è SWI-Prolog. Presentiamo di seguito un esempio di applicazione. Consideriamo il classico problema di criptoaritmetica:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Per risolvere questo problema è necessario assegnare ad ogni variabile un valore compreso fra 0 e 9 in maniera tale che la somma risulti corretta. Per far questo possiamo sfruttare, ad esempio, CLP/bounds, un Integer Bound Constraint Solver:

```
:- use_module(library('clp/bounds')).

puzzle(S,E,N,D,M,O,R,Y):-
    Variabili = [S,E,N,D,M,O,R,Y],
    Variabili in 0..9,
    all_different(Variabili),
    S #>= 1,
    M #>= 1,
              S*1000 + E*100 + N*10 + D +
              M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
    label(Variabili).
```

Capitolo 2

Polinomi di Bernstein

Gli algoritmi che presentermo nel seguito per il restringimento di domini delle variabili coinvolte in vincoli polinomiali poggiano su una forte base matematica. Per questo motivo, in questo capitolo verranno introdotti e descritti i polinomi nella base di Bernstein, una particolare classe di polinomi sul campo reale, utilizzati principalmente nell'ambito dell'analisi numerica e dovuti al matematico russo Sergej Natanovič Bernštejn, che risulteranno di fondamentale importanza nei prossimi capitoli.

2.1 Una nuova base

Un polinomio può essere espresso per mezzo della base canonica, detta anche *base di potenze*, come segue:

$$p(x) = \sum_{k=0}^{n} d_k x^k$$

Questa forma, come è noto, non è tuttavia l'unica possibile. La base di Bernstein rappresenta infatti una valida alternativa per esprimere un polinomio. Diamo quindi la seguente

Definizione 2.1 (Polinomio di Bernstein)**.** Un polinomio di Bernstein [3] (o polinomio nella base di Bernstein) $p(x)$ di grado n è dato dalla formula:

$$p(x) = \sum_{k=0}^{n} c_k B_k^{(n)}(x)$$

dove i $B_k^{(n)}(\cdot)$ sono gli elementi costituenti la base di Bernstein, in particolare:

$$B_k^{(n)}(x) = \binom{n}{k} x^k (1-x)^{n-k} \text{ con } x \in [0,1]$$

Per convenzione poniamo $B_k^{(n)}(x) \equiv 0$ se $k < 0$ o $k > n$.

Osservazione. Dato che i polinomi godono della proprietà di essere invarianti per traslazione e scala dell'intervallo di definizione o cambio di variabile, la definizione può essere generalizzata a polinomi di Bernstein rispetto ad un intervallo $[a, b]$ qualsiasi, considerando la trasformazione affine seguente:

$$[a, b] \to [0, 1]$$

$$x \to t = X(x) := \frac{x - a}{b - a}$$

Da cui si ricava la definizione del k-esimo polinomio di Bernstein nell'intervallo $[a, b]$:

$$B_k^{(n)}(t) = \binom{n}{k} \frac{(b - t)^{n-k}(t - a)^k}{(b - a)^n} \text{ con } t \in [a, b]$$

Esempio. Proviamo ad esprime un polinomio di grado 2 in base di Bernstein in $[0, 1]$. Avremo:

$$\begin{aligned} p(x) &= \sum_{k=0}^{2} c_k B_k^{(2)}(x) \\ &= c_0 B_0^{(2)}(x) + c_1 B_1^{(2)}(x) + c_2 B_2^{(2)}(x) \end{aligned}$$

dove rispettivamente:

- $B_0^{(2)}(x) = \binom{2}{0} x^0 (1 - x)^{2-0} = (1 - x)^2$
- $B_1^{(2)}(x) = \binom{2}{1} x^1 (1 - x)^{2-1} = 2x(1 - x)$
- $B_2^{(2)}(x) = \binom{2}{2} x^2 (1 - x)^{2-2} = x^2$

Quindi il polinomio potrà essere scritto come:

$$p(x) = c_0(1 - x)^2 + c_1 2x(1 - x) + c_2 x^2$$

Ora siamo pronti per dimostrare che i polinomi di Bernstein formano davvero una base:

Teorema 2.1.1. *I polinomi di Bernstein formano una base dello spazio vettoriale* $\Pi_n(\mathbb{R})$ *dei polinomi a coefficienti reali di grado al più* n.

Dimostrazione. Mediante lo sviluppo di Newton, abbiamo che:

$$B_k^{(n)}(x) = \binom{n}{k} x^k (1-x)^{n-k} = \binom{n}{k} x^k \sum_{j=0}^{n-k} \binom{n-k}{j} (-1)^{n-k-j} x^{n-k-j}$$

Quindi, usando una notazione matriciale, è possibile esprimere i polinomi di Bernstein in termini della base canonica come segue:

$$\begin{pmatrix} B_0^{(n)}(x) \\ \\ \vdots \\ \\ B_n^{(n)}(x) \end{pmatrix} = \begin{pmatrix} 1 & * & * & * & * \\ & \ddots & * & * & * \\ & & \binom{n}{k} & * & * \\ & & & \ddots & * \\ & & & & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^n \end{pmatrix}$$

dove i termini indicati con l'asterisco sono non nulli. Affinché i $B_k^{(n)}$ formino una base, la matrice rappresentata deve essere non singolare, ma la matrice è triangolare superiore, quindi il determinante è dato dalla produttoria degli elementi sulla diagonale principale, tutti diversi da 0, ragion per cui i $B_k^{(n)}$ formano una base di $\Pi_n(\mathbb{R})$. □

2.2 Principali proprietà

Illustriamo in questo paragrafo le principali proprietà, che ci saranno di aiuto nel seguito, di cui gode la base di Bernstein.

Proposizione 2.2.1. *I polinomi di Bernstein possono essere definiti per ricorsione dalla formula:*

$$B_k^{(n)}(x) = (1-x)B_k^{(n-1)}(x) + xB_{k-1}^{(n-1)}(x)$$

Dimostrazione. Dimostriamo la formula per induzione. Proviamo con n fissato che agli estremi

$$\begin{cases} B_0^{(n)}(x) = & (1-x)^n \\ B_n^{(n)}(x) = & x^n \end{cases}$$

Si ha per la prima equazione:

$$B_0^{(n)}(x) = \binom{n}{0} x^0 (1-x)^n = (1-x)(1-x)^{n-1}$$

Mentre per la seconda equazione:

$$B_n^{(n)}(x) = \binom{n}{n} x^n (1-t)^0 = x \cdot x^{n-1}$$

Dimostriamo ora il caso induttivo. Ricordando la proprietà del coefficiente binomiale:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Dalla definizione:

$$B_k^{(n-1)}(x) = \binom{n-1}{k} x^k (1-x)^{n-1-k}$$
$$B_{k-1}^{(n-1)}(x) = \binom{n-1}{k-1} x^{k-1} (1-x)^{n-1-k+1}$$

Possiamo quindi scrivere:

$$\begin{aligned} B_k^{(n)}(x) &= (1-x) B_k^{(n-1)}(x) + x B_{k-1}^{(n-1)}(x) \\ &= \binom{n-1}{k} x^k (1-x)^{n-k} + \binom{n-1}{k-1} x^k (1-x)^{n-k} \end{aligned}$$

A questo punto il passaggio per ottenere la tesi diventa immediato. □

Proposizione 2.2.2. *I polinomi di Bernstein partizionano l'unità, ossia:*

$$\sum_{k=0}^{n} B_k^{(n)}(x) = 1$$

Dimostrazione. Possiamo facilmente ricavare la tesi dallo sviluppo di Newton:

$$1 = 1^n = (x + 1 - x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k (1-x)^{n-k}$$

□

Un altro risultato interessante che sarà ampiamente usato nel seguito è rappresentato dalla seguente

Proposizione 2.2.3. *I polinomi di Bernstein sono simmetrici sull'intervallo* $[0,1]$*, cioè per ogni* $x \in [0,1]$ *e per ogni* $k \in \{0 \dots n\}$ *si ha:*

$$B_k^{(n)}(x) = B_{n-k}^{(n)}(1-x)$$

Dimostrazione. Dal momento che:

$$B_k^{(n)}(x) = \binom{n}{k} x^k (1-x)^{n-k}$$
$$B_{n-k}^{(n)}(x) = \binom{n}{n-k} (1-x)^{n-k} x^k$$

e dato che per i coefficienti binomiali vale che $\binom{n}{k} = \binom{n}{n-k}$ si ha la tesi. □

Calcoliamo adesso la derivata di $B_k^{(n)}(x)$ e determiniamo i punti di massimo:

$$\begin{aligned}
\frac{\mathrm{d}B_k^{(n)}(x)}{\mathrm{d}x} &= \binom{n}{k} \left[k x^{k-1} (1-x)^{n-k} + (-1)(n-k) x^k (1-x)^{n-k-1} \right] \\
&= \binom{n}{k} x^{k-1} (1-x)^{n-k-1} \left[k(1-x) - (n-k)x \right] \\
&= \binom{n}{k} x^{k-1} (1-x)^{n-k-1} \left[k - nx \right]
\end{aligned}$$

Quindi i $B_k^{(n)}(x)$ hanno massimi locali nei punti $\tilde{x} = \frac{k}{n}$, per $k = 0, \ldots, n$. Osserviamo inoltre che la derivata può essere espressa in forma ricorsiva:

$$\begin{aligned}
\frac{\mathrm{d}B_k^{(n)}(x)}{\mathrm{d}x} &= \frac{\mathrm{d}}{\mathrm{d}x} \binom{n}{k} x^k (1-x)^{n-k} \\
&= \frac{k\ n!}{k!(n-k)!} x^{k-1} (1-x)^{n-k} - \frac{(n-k)n!}{k!(n-k)!} x^k (1-x)^{n-k-1} \\
&= n \frac{(n-1)!}{(k-1)!(n-k)!} x^{k-1} (1-x)^{n-k} - n \frac{(n-1)!}{k!(n-k-1)!} x^k (1-x)^{n-k-1} \\
&= n \left(\binom{n-1}{k-1} x^{k-1} (1-x)^{n-k} - \binom{n-1}{k} x^k (1-x)^{n-k-1} \right) \\
&= n \left(B_{k-1}^{(n-1)}(x) - B_k^{(n-1)}(x) \right)
\end{aligned}$$

Proposizione 2.2.4. *I polinomi di Bernstein formano una combinazione convessa (ossia una combinazione lineare di elementi fatti da coefficienti non negativi a somma 1), ovvero:*

- $\forall k \in \mathbb{N}, B_k^{(n)}(x) \geq 0$
- $\sum_{k=0}^{n} B_k^{(n)}(x) = 1$

2.3 L'Algoritmo di De Casteljau

L'algoritmo di De Casteljau, che prende il nome da Paul de Casteljau, fisico e matematico francese, è un metodo ricorsivo per valutare polinomi nella forma di Bernstein. Nonostante il metodo sia generalmente più lento per la maggior parte delle architetture, se comparato all'approccio tradizionale, è numericamente più stabile.

Sia $p(x) = \sum_{k=0}^{n} c_k B_k^{(n)}(x)$ un polinomio in forma di Bernstein di grado n e supponiamo di volerlo valutare nel punto $x = x_0$. Allora per $i = 0, \ldots, (n-j)$ e per $j = 1, \ldots, n$ vale la seguente relazione di ricorrenza:

$$\begin{cases} c_i^{(0)} & = c_i \\ c_i^{(j)} & = c_i^{(j-1)}(1 - x_0) + c_{i+1}^{(j-1)} x_0 \end{cases} \tag{2.1}$$

con $p(x_0) = c_0^{(n)}$.

Esempio. Si voglia calcolare il valore del polinomio di grado 2 di coefficienti:

$$c_0^{(0)} = c_0 \qquad c_1^{(0)} = c_1 \qquad c_2^{(0)} = c_2$$

nel punto x_0. Quanto voluto è il risultato del seguente calcolo:

$$p(x_0) = c_0^{(2)} = c_0^{(1)}(1 - x_0) + c_1^{(1)} x_0$$

Allora, applicando l'algoritmo di De Casteljau, per ricorsione si produce:

$$\begin{aligned} c_0^{(1)} &= c_0^{(0)}(1 - x_0) + c_1^{(0)} x_0 = c_0(1 - x_0) + c_1 x_0 \\ c_1^{(1)} &= c_1^{(0)}(1 - x_0) + c_2^{(0)} x_0 = c_1(1 - x_0) + c_2 x_0 \end{aligned}$$

Infine sostituendo si ottiene:

$$\begin{aligned} p(x_0) = c_0^{(2)} &= c_0^{(1)}(1 - x_0) + c_1^{(1)}(x_0) \\ &= (c_0(1 - x_0) + c_1 x_0)(1 - x_0) + (c_1(1 - x_0) + c_2 x_0) x_0 \\ &= c_0(1 - x_0)^2 + 2c_1 x_0(1 - x_0) + c_2 x_0^2 \end{aligned}$$

abbiamo così quanto voluto.

L'algoritmo è stato implementato in linguaggio MATLAB per permettere una valutazione efficace dei polinomi di Bernstein. Prende come input il polinomio in forma di Bernstein `pBernstein` e il punto `point` in cui deve avvenire la valutazione e restituisce il risultato di detta valutazione nella variabile `res`. All'avvio dell'algoritmo, vengono create due variabili: `lung`, che rappresenta la dimensione della matrice delle soluzioni parziali e `M`, che identifica la suddetta matrice. In seguito si procede con un ciclo ad applicare la prima equazione della formula 2.1:

```
for k = 1 : lung
    M(1,k) = pBernstein(k);
end
```

A questo punto è possibile applicare la seconda relazione della formula 2.1 con un doppio ciclo:

```
index = 1;
for i = 2 : lung
    for j = 1 : lung - index
        M(i,j) = M(i-1,j)*(1-point) + M(i-1,j+1)*point;
    end
    index = index + 1;
end
```

ed estrarre quindi la soluzione finale:

```
res = M(lung,1);
```

Ad esempio, se si volesse valutare il polinomio

$$p(x) = -8B_0^{(3)} + 13,6667B_1^{(3)} - 14,6667B_2^{(3)} - 3B_3^{(3)}$$

in $\frac{1}{2}$ basterebbe digitare nel workspace:

```
>> polB = [-8 13,6667 -14,6667 -3];
>> point = 0.5;
>> DeCasteljau(polB,point)

ans =

   -1.7500
```

2.4 Passaggio in Forma di Bernstein

Dato il polinomio di grado n a coefficienti reali $d_i \in \mathbb{R}$ espresso in forma di potenze:

$$p(x) = \sum_{k=0}^{n} d_k x^k$$

ci si pone il problema di scrivere lo stesso polinomio nella forma di Bernstein:

$$p(x) = \sum_{k=0}^{n} c_k B_k^{(n)}(x) = \sum_{k=0}^{n} c_k \binom{n}{k} x^k (1-x)^{n-k}$$

Occorre cioè calcolare i coefficienti di Bernstein c_k a partire dai noti d_k. Procediamo sviluppando con Taylor il polinomio $p(x)$ intorno al punto $x = 0$,

ossia operiamo uno Sviluppo di Maclaurin su $p(x)$:

$$p(x) = p(0) + p'(0)x + \frac{p''(0)x^2}{2!} + \cdots + \frac{p^{(k)}(0)x^k}{k!} + \dots$$

Affinché le due rappresentazioni di $p(x)$ coincidano, occorre che anche i due sviluppi coincidano, ovvero che coincidano i coefficienti corrispondenti per ogni potenza di x. Imponiamo quindi che per $k = 0, \dots, n$:

$$\frac{\mathrm{D}^{(\mathrm{k})}(d_k x^k)(0)x^k}{k!} = \frac{\mathrm{D}^{(\mathrm{k})}(c_k B_k^{(n)}(x))(0)x^k}{k!}$$

Ossia:

$$\frac{\mathrm{D}^{(\mathrm{k})}(d_k x^k)(0)}{k!} = \frac{\mathrm{D}^{(\mathrm{k})}(c_k B_k^{(n)}(x))(0)}{k!}$$

Semplificando il primo termine ci si riconduce all'espressione:

$$d_k = \frac{\mathrm{D}^{(\mathrm{k})}(c_k B_k^{(n)}(x))(0)}{k!}$$

Infatti, ad esempio, si può agilmente verificare che:

$$\begin{aligned}\frac{\mathrm{D}^{(3)}(2x^3)(0)}{3!} &= \frac{3 \cdot \mathrm{D}^{(2)}(2x^2)(0)}{3!} = \frac{3 \cdot 2 \cdot \mathrm{D}^{(1)}(2x)(0)}{3!} \\ &= \frac{3 \cdot 2 \cdot 1 \cdot (2)}{3!} = \frac{3! \cdot (2)}{3!}\end{aligned}$$

Ora, ricordando la forma della derivata dei polinomi di Bernstein mostrata precedentemente, i coefficienti dei polinomi derivati in forma di Bernstein sono $n(c_1 - c_0), n(c_2 - c_1), \dots, n(c_n - c_{n-1})$ per la derivata prima e $n(n-1)(c_2 - 2c_1 + c_0), \dots, n(n-1)(c_n - 2c_{n-1} + c_{n-2})$ per la derivata seconda, e così via. Allora si può partire da:

$$p(0) = d_0 = \sum_{k=0}^{n} c_k B_k^{(n)}(0) = \sum_{k=0}^{n} c_k \binom{n}{k} 0^k (1-0)^{n-k} = c_0$$

Quest'ultima uguaglianza è giustificata dal fatto che nella sommatoria tutti i termini si annullano eccetto il primo, dal momento che:

$$\begin{aligned}&\forall\ k > 0,\ 0^k = 0 \\ &\forall\ n \geq 0,\ n^0 = 1\end{aligned}$$

Per cui $0^0 = 1$ senza ambiguità. Per le derivate successive allora si ha:

$$\begin{aligned} p'(0) &= n(c_1 - c_0) \\ p''(0) &= n(n-1)(c_2 - 2c_1 + c_0) \\ &\vdots \\ p^{(k)}(0) &= n(n-1)\dots(n-k+1)\sum_{j=0}^{k}(-1)^{k-j+1}\binom{k}{j}c_j \end{aligned}$$

È possibile introdurre, a questo punto, la formula ricorsiva:

$$c_k^{[j]} = c_{k+1}^{[j-1]} - c_k^{[j-1]}$$

dove i termini: $c_k^{[0]} = c_k$ sono i coefficienti che dobbiamo determinare. Possiamo dunque scrivere in forma compatta, usando la notazione appena introdotta:

$$p^{(k)}(0) = n(n-1)\dots(n-k+1)c_0^{[k]} = \frac{n!}{(n-k)!}c_0^{[k]}$$

Quindi, riassumendo, dallo sviluppo di Taylor eseguito prima, abbiamo che:

$$d_k = \frac{1}{k!}\frac{n!}{(n-k)!}c_0^{[k]} = \binom{n}{k}c_0^{[k]}$$

In questo modo abbiamo ottenuto un algoritmo di conversione efficace (molto simile all'algoritmo di De Casteljau) che può essere applicato nel modo seguente:

1. Si calcolino i valori $c_0^{[k]}$ per mezzo della formula $d_k = \binom{n}{k}c_0^{[k]}$, cioè

$$c_0^{[k]} = \frac{d_k}{\binom{n}{k}} \tag{2.2}$$

2. Si calcolino i successivi valori per mezzo della formula ricorsiva:

$$c_k^{[j]} = c_{k+1}^{[j-1]} - c_k^{[j-1]}$$

per la cui applicazione osserviamo che:

$$c_{k-1}^{[j+1]} = c_k^{[j]} - c_{k-1}^{[j]}$$

da cui si ricava la formula effettivamente usata per il calcolo dei coefficienti:

$$c_k^{[j]} = c_{k-1}^{[j+1]} + c_{k-1}^{[j]} \tag{2.3}$$

3. Si selezionino i $c_k^{[0]}$, effettivi coefficienti di Bernstein.

Esempio. Proviamo a calcolare i coefficienti di Bernstein per il polinomio di grado $n = 3$:

$$p(x) = 4 - 3x + 6x^2 + 2x^3$$

Dalla formula 2.2 si ricava:

$$c_0^{[0]} = \frac{4}{\binom{3}{0}} = 4, \qquad c_0^{[1]} = \frac{-3}{\binom{3}{1}} = -1$$
$$c_0^{[2]} = \frac{1}{\binom{3}{2}} = \frac{1}{3}, \qquad c_0^{[3]} = \frac{2}{\binom{3}{3}} = 2$$

Dalla formula 2.3, a questo punto si ottiene:

$$c_1^{[0]} = c_0^{[1]} + c_0^{[0]} = -1 + 4 = 3, \qquad c_1^{[1]} = c_0^{[2]} + c_0^{[1]} = -1 + \frac{1}{3} = -\frac{2}{3}$$
$$c_1^{[2]} = c_0^{[3]} + c_0^{[2]} = 2 + \frac{1}{3} = \frac{7}{3}, \qquad c_2^{[0]} = c_1^{[1]} + c_1^{[0]} = -\frac{2}{3} + 3 = \frac{7}{3}$$
$$c_2^{[1]} = c_1^{[2]} + c_1^{[1]} = -\frac{2}{3} + \frac{7}{3} = \frac{5}{3}, \qquad c_3^{[0]} = c_2^{[1]} + c_2^{[0]} = \frac{5}{3} + \frac{7}{3} = 4$$

I coefficienti del polinomio risultante sono dati dai $c_k^{[0]}$, per cui si ottiene infine:

$$p(x) = 4B_0^{(3)}(x) + 3B_1^{(3)}(x) + \frac{7}{3}B_2^{(3)}(x) + 4B_3^{(3)}(x)$$

L'algoritmo è stato correttamente implementato in linguaggio MATLAB. Viene preso come input il solo vettore di coefficienti del polinomio in forma di potenze `C`, dichiarato a partire dal termine meno significativo. Si assume inoltre che il dominio della variabile indipendente sia $[0, 1]$. All'avvio dell'algoritmo, viene dedotto il grado del polinomio e salvato in una variabile `degree`, dopodiché viene applicata la formula 2.2 tramite un semplice ciclo:

```
for k = 1 : length(C)
    B(1,k) = C(k)/nchoosek(degree, k-1);
end
```

I risultati sono memorizzati sulla prima riga di una matrice `B`. A questo punto è possibile applicare la formula ricorsiva 2.3, inserendo le restanti soluzioni nella matrice `B`:

```
index = 1;
for k = 2 : length(C)
    for j = 1 : length(C) - index
        B(k,j) = B(k-1,j) + B(k-1,j+1);
    end
    index = index + 1;
end
```

Infine vengono estratti dalla matrice e ritornati dalla function gli effettivi coefficienti di Bernstein:

```
for k = 1 : length(B)
    res(k) = B(k,1);
end
```

Di seguito viene fornito un esempio in cui viene portato in base di Bernstein il polinomio

$$p(x) = -8 + 65x - 150x^2 + 90x^3$$

con $x \in [0, 1]$:

```
>> pol = [-8 65 -150 90];
>> toBernsteinMono(pol)

ans =

   -8.0000   13.6667  -14.6667   -3.0000
```

2.5 Polinomi di Bernstein in più variabili

Diamo nel seguito una trattazione coerente per descrivere, per polinomi in più variabili, tutti i concetti esposti nei paragrafi precedenti. Sia $n > 0$ e sia $x = (x_1, x_2, \ldots, x_n)$ un vettore di variabili reali.

Definizione 2.2 (Multi-indice)**.** Si definisce multi-indice un vettore $I = (i_1, i_2, \ldots, i_n)$ con $I \in \mathbb{N}^n$.

I non è altro quindi che un vettore a sole coordinate naturali, che, per convenzione, siamo soliti chiamare indici.

Definizione 2.3 (Multi-potenza)**.** Siano dati $n > 0$, $x = (x_1, x_2, \ldots, x_n)$ e $I = (i_1, i_2, \ldots, i_n)$. Si definisce multi-potenza di x:

$$x^I = \prod_{k=0}^{n} x_k^{i_k} = x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n}$$

Si definiscono inoltre le seguenti regole per $I, J \in \mathbb{N}^n$ e $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$:

- $I \pm J = (i_1 \pm j_1, i_2 \pm j_2, \ldots i_n \pm j_n)$
- $I \leq J \Leftrightarrow \forall k \in \{1, \ldots, n\}, i_k \leq j_k$
- $I_{r,k} = (i_1, \ldots, i_{r-1}, i_r + k, i_{r+1}, \ldots, i_n)$

- $\binom{I}{N} = \prod_{k=1}^{n} \binom{i_k}{n_k}$

Date queste semplici premesse, possiamo definire il polinomio nella variabile n-dimensionale $x = (x_1, \ldots, x_n)$ di grado $N = (n_1, \ldots, n_n)$ sul dominio $\mathbf{x} = [\underline{x}_1, \overline{x}_1] \times [\underline{x}_2, \overline{x}_2] \times \ldots \times [\underline{x}_n, \overline{x}_n]$ in forma di potenze come:

$$p(x) = \sum_{I \leq N} d_I x^I = \sum_{i_1=0}^{n_1} \left(\sum_{i_2=0}^{n_2} \left(\ldots \left(\sum_{i_n=0}^{n_n} d_{i_1 i_2 \ldots i_n} x_1^{i_1} x_2^{i_2} \cdot \ldots \cdot x_n^{i_n} \right) \ldots \right) \right)$$

Un polinomio di questo tipo può essere rappresentato in una multi-matrice in cui l'elemento in posizione $(i_1, i_2, \ldots, i_n)$ è il coefficiente del termine $(x_1^{i_1} x_2^{i_2} \cdot \ldots \cdot x_n^{i_n})$.

A sua volta, possiamo definire l'I-esimo polinomio di Bernstein di grado N sull'intervallo $[0, 1]^n$ come:

$$B_I^{(N)}(x) = B_{i_1}^{(n_1)}(x_1) \cdot \ldots \cdot B_{i_n}^{(n_n)}(x_n)$$

dove:

$$B_{i_j}^{(n_j)}(x_j) = \binom{n_j}{i_j} x_j^{i_j} (1 - x_j)^{n_j - i_j}$$

E dunque il polinomio $p(x)$ può essere rappresentato in Forma di Bernstein come:

$$p(x) = \sum_{I \leq N} c_I B_I^{(N)}(x)$$

dove i coefficienti nella base di Bernstein di grado N sono dati da:

$$c_I = \sum_{J \leq I} \frac{\binom{I}{J}}{\binom{N}{J}} d_J, I \leq N$$

Estendendo la definizione su un generico dominio $\mathbf{x}$ possiamo rappresentare l'I-esimo polinomio di Bernstein di grado N come:

$$B_I^{(N)}(x) = B_{i_1}^{(n_1)}(x_1) \cdot \ldots \cdot B_{i_n}^{(n_n)}(x_n)$$

dove per $i_j = 0, \ldots, n_j$ e per $j = 1, \ldots, n$ e $x_j \in [\underline{x}_j, \overline{x}_j]$:

$$B_{i_j}^{(n_j)}(x_j) = \frac{1}{(\overline{x}_j - \underline{x}_j)^n} \binom{n_j}{i_j} (x_j - \underline{x}_j)^{i_j} (\overline{x}_j - x_j)^{n_j - i_j}$$

Quindi rappresentando il polinomio $p(x)$ in forma di Bernstein di grado N come $p(x) = \sum_{I \leq N} c_I B_I^{(N)}(x)$ si ha che i coefficienti nella base di Bernstein sul generico dominio $\mathbf{x}$ sono dati da:

$$c_I = \sum_{J \leq I} \frac{\binom{I}{J}}{\binom{N}{J}} (\underline{x} - \overline{x})^J \sum_{J \leq K \leq N} \binom{K}{J} \underline{x}^{K-J} d_K \tag{2.4}$$

Un metodo diretto per calcolare i coefficienti di Bernstein a partire dai coefficienti del polinomio in forma di potenze consiste quindi nell'applicare la formula 2.4. Grazie ad alcune brillanti osservazioni, tuttavia, l'informatico e matematico britannico Andrew Paul Smith ha generato un procedimento più efficiente rispetto a quello appena illustrato. In questo modo, Smith ha migliorato in termini di costo e tempo il calcolo dei coefficienti di Bernstein, non essendo più necessario memorizzare tutti i coefficienti, ma solo alcuni, quando vengono richiesti[1]. L'algoritmo consiste nel calcolare in primo luogo i coefficienti dei monomi in una variabile, per poi ottenere i coefficienti del monomio in più variabili e infine giungere al polinomio in più variabili desiderato.

2.6 L'Algoritmo di Smith

Come anticipato nel paragrafo precedente, illustriamo adesso un procedimento efficace per ottenere i coefficienti di Bernstein a partire dal polinomio in forma di potenze. Per dare un'idea del funzionamento di questa procedura, dato un polinomio multi-variabile, vengono dapprima considerati singolarmente i monomi in più variabili da cui è costituito il polinomio, per poi essere smembrati in monomi ad una sola variabile. Di questi si calcolano i coefficienti di Bernstein. Questi risultati parziali concorrono ad ottenere prima i coefficienti del monomio multi-variabile e poi quelli del polinomio multi-variabile.

2.6.1 Monomi in una variabile

I coefficienti di Bernstein c_{i_m} di grado n_m del monomio in una variabile $x_m^{k_m}$ calcolati sull'intervallo mono-dimensionale $[\overline{x}_m, \underline{x}_m]$ sono dati, sfruttando la formula 2.4 presentata nel paragrafo precedente, da:

$$c_{i_m} = \sum_{j=0}^{\min(i_m, k_m)} \frac{\binom{i_m}{j}}{\binom{n_m}{j}} (\overline{x}_m - \underline{x}_m)^j \binom{k_m}{j} \underline{x}_m^{k_m - j}$$

Questo calcolo può essere semplificato analizzando il grado relativo e assoluto della variabile x_m. Per grado relativo si intende il grado di x_m in un monomio estrapolato dal polinomio, mentre per grado assoluto si intende il grado massimo della variabile nel polinomio in più variabili.

Esempio. Sia dato il polinomio $p(x) = x_1^3 x_2^2 - 6x_1 x_2$. Il grado assoluto della variabile x_1 è 3, mentre il suo grado relativo è 3 nel primo monomio e 1 nel secondo monomio. Il grado assoluto della variabile x_2 è 2, mentre il suo grado relativo è 2 nel primo monomio e 1 nel secondo monomio.

Indicando con k_m il grado relativo e con n_m il grado assoluto, si possono distinguere due casi:

1. Il grado assoluto è uguale al grado relativo, ossia $k_m = n_m$.
   In questo caso si trova che:

$$\begin{aligned} c_{i_m} &= \sum_{j=0}^{\min(i_m,k_m)} \frac{\binom{i_m}{j}}{\binom{n_m}{j}} (\overline{x}_m - \underline{x}_m)^j \binom{k_m}{j} \underline{x}_m^{k_m-j} \\ &= \sum_{j=0}^{i_m} \binom{i_m}{j} (\overline{x}_m - \underline{x}_m)^j \underline{x}_m^{k_m-j} \\ &= {\underline{x}_m}^{k_m-i_m} \sum_{j=0}^{i_m} \binom{i_m}{j} (\overline{x}_m - \underline{x}_m)^j \underline{x}_m^{i_m-j} \\ &= {\underline{x}_m}^{k_m-i_m} (\overline{x}_m - \underline{x}_m + \underline{x}_m)^{i_m} \\ &= {\underline{x}_m}^{k_m-i_m} {\overline{x}_m}^{i_m} \end{aligned}$$

   Quindi se il grado relativo e il grado assoluto coincidono vale che:

$$c_{i_m} = {\underline{x}_m}^{k_m-i_m} {\overline{x}_m}^{i_m} \tag{2.5}$$

2. Il grado assoluto è maggiore del grado relativo, ossia $n_m > k_m$.
   In questo caso, senza entrare eccessivamente nel dettaglio, sfruttando la tecnica del degree elevation (che consente di aumentare il grado della curva che stiamo rappresentando senza per questo cambiarne la forma) possiamo giungere alla seguente formula per il coefficiente c_{i_m} di grado n_m:

$$c_{i_m}^{(n_m)} = \sum_{j=\max(0,i_m-r)}^{\min(i_m,k_m)} \frac{\binom{r}{i_m-j}\binom{k_m}{j}}{\binom{k_m+r}{i_m}} c_j^{(k_m)} \tag{2.6}$$

$$= \sum_{j=\max(0,i_m-r)}^{\min(i_m,k_m)} \frac{\binom{r}{i_m-j}\binom{k_m}{j}}{\binom{k_m+r}{i_m}} \underline{x}^{k_m-j} \overline{x}^j \tag{2.7}$$

   dove con r indichiamo la differenza fra grado assoluto e grado relativo, ossia $r = n_m - k_m$.

2.6.2 Monomi in più variabili

Una volta ottenuti tutti i coefficienti di Bernstein dei monomi in una variabile, si ottengono i coefficienti del monomio in più variabili moltiplicando tramite prodotto tensoriale gli oggetti ottenuti dal calcolo.

Definizione 2.4 (Tensore)**.** Un tensore è una più generale tabella di numeri n-dimensionale, che generalizza i casi $n = 1$ (un vettore) e $n = 2$ (una matrice). La dimensione n del tensore è detta ordine.

Dato un monomio multi-variabile $q(x) = x^K$, con $0 \leq K \leq N$, distinguiamo due casi:

- Il monomio è composto da due sole variabili.

  In questo caso si ottengono i coefficienti di Bernstein del polinomio multi-variabile moltiplicando il vettore dei coefficienti ottenuti dal primo monomio uni-variabile con il trasposto del vettore di coefficienti ottenuti dal secondo monomio uni-variabile. Il tutto deve essere poi moltiplicato per il coefficiente d_i del monomio multi-variabile. Quindi se $\mathbf{a}$ e $\mathbf{b}$ sono i vettori dei coefficienti di Bernstein dei monomi uni-variabile e d_i è il coefficiente del monomio multi-variabile, allora i coefficienti di Bernstein del monomio multi-variabile sono dati da:

  $$d_I = d_i(\mathbf{a} \otimes \mathbf{b}) = d_i(\mathbf{a}\mathbf{b}^T)$$

  dove con $\otimes$ indichiamo l'operazione di prodotto tensoriale, ossia:

  $$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \end{bmatrix} \otimes \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 & \cdots \end{bmatrix} = \begin{bmatrix} a_1b_1 & a_2b_2 & a_3b_3 & \cdots \\ a_2b_1 & a_2b_2 & a_2b_3 & \cdots \\ a_3b_1 & a_3b_2 & a_3b_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

- Il monomio è composto da più di due variabili.

  In questo caso, senza addentrarci nello specifico nell'algebra tensoriale e senza introdurre eccessivi formalismi, ci limiteremo a definire il prodotto tensoriale fra due tensori di ordine rispettivamente h e k come una matrice multi-dimensionale di ordine $h + k$ in cui tutti i possibili elementi del primo tensore sono stati moltiplicati per tutti i possibili elementi del secondo tensore. Si rimanda per una trattazione più dettagliata a testi specifici.

Esempio. Proviamo a moltiplicare un tensore di ordine 1 con un tensore di ordine 2. Il risultato sarà un tensore di ordine 3 (ossia una matrice tridimensionale), calcolato come segue:

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \otimes \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \left\{ \begin{bmatrix} a_1 b_1 & a_1 b_2 \\ a_1 b_3 & a_1 b_4 \end{bmatrix}, \begin{bmatrix} a_2 b_1 & a_2 b_2 \\ a_2 b_3 & a_2 b_4 \end{bmatrix} \right\}$$

2.6.3 Polinomio in più variabili

Una volta ottenuti tutti i coefficienti di Bernstein dei monomi in più variabili si ricavano i coefficienti del polinomio multi-variabile sommando tutti gli oggetti ottenuti dal calcolo dei monomi multi-variabile. Questo vuol dire che ogni coefficiente di Bernstein è uguale alla somma dei corrispondenti coefficienti di Bernstein di ogni termine.

Esempio. Si vogliano calcolare i coefficienti di Bernstein del polinomio $p(x) = x_1^3 x_2^2 - 6x_1 x_2$ di grado $N = (3, 2)$ sul dominio $\mathbf{x} = [1, 2] \times [2, 4]$ con variabile bi-dimensionale $x = (x_1, x_2)$. Il polinomio p è formato da due monomi in più variabili:

$$q_1(x) = x_1^3 x_2^2 \qquad q_2(x) = -6x_1 x_2$$

Analizziamo il monomio q_1. Esso è formato, a sua volta, da due monomi in una variabile:

$$q_1'(x) = x_1^3 \qquad q_1''(x) = x_2^2$$

Consideriamo q_1'. Il suo grado relativo coincide con il suo grado assoluto e vale 3. Allora, applicando la formula 2.5 si ha:

$$c_0 = 1^{3-0} \cdot 2^0 = 1 \qquad c_1 = 1^{3-1} \cdot 2^1 = 2$$
$$c_2 = 1^{3-2} \cdot 2^2 = 4 \qquad c_3 = 1^{3-3} \cdot 2^3 = 8$$

Consideriamo q_1''. Il suo grado relativo coincide con il suo grado assoluto e vale 2. Allora, applicando la formula 2.5 si ha:

$$c_0 = 2^{2-0} \cdot 4^0 = 4 \qquad c_1 = 2^{2-1} \cdot 4^1 = 8 \qquad c_2 = 2^{2-2} \cdot 4^2 = 16$$

A partire da questi coefficienti possiamo ottenere i coefficienti del primo monomio in due variabili:

$$1 \left(\begin{bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{bmatrix} \otimes \begin{bmatrix} 4 \\ 8 \\ 16 \end{bmatrix} \right) = 1 \begin{bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{bmatrix} \begin{bmatrix} 4 & 8 & 16 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 16 \\ 8 & 16 & 32 \\ 16 & 32 & 64 \\ 32 & 64 & 128 \end{bmatrix}$$

Analizziamo ora il monomio q_2. Esso è formato, a sua volta, da due monomi in una variabile:

$$q_2'(x) = x_1 \qquad q_2'' = x^2$$

Consideriamo q_2'. Applicando la formula 2.7 si ha:

$$c_0^{(3)} = \sum_{j=\max(0,0-2)}^{\min(0,1)} \frac{\binom{2}{0-j}\binom{1}{j}}{\binom{1+2}{0}} 1^{1-j}2^j = 1, \qquad c_1^{(3)} = \sum_{j=\max(0,1-2)}^{\min(1,1)} \frac{\binom{2}{1-j}\binom{1}{j}}{\binom{1+2}{1}} 1^{1-j}2^j = \frac{4}{3}$$

$$c_2^{(3)} = \sum_{j=\max(0,2-2)}^{\min(2,1)} \frac{\binom{2}{2-j}\binom{1}{j}}{\binom{1+2}{2}} 1^{1-j}2^j = \frac{5}{3}, \qquad c_3^{(3)} = \sum_{j=\max(0,3-2)}^{\min(3,1)} \frac{\binom{2}{3-j}\binom{1}{j}}{\binom{1+2}{3}} 1^{1-j}2^j = 2$$

Consideriamo q_2''. Applicando la formula 2.7 si ha:

$$c_0^{(2)} = \sum_{j=\max(0,0-1)}^{\min(0,1)} \frac{\binom{1}{0-j}\binom{1}{j}}{\binom{1+1}{0}} 2^{1-j}4^j = 2 \qquad c_1^{(2)} = \sum_{j=\max(0,1-1)}^{\min(1,1)} \frac{\binom{1}{1-j}\binom{1}{j}}{\binom{1+1}{1}} 2^{1-j}4^j = 3$$

$$c_2^{(2)} = \sum_{j=\max(0,2-1)}^{\min(2,1)} \frac{\binom{1}{2-j}\binom{1}{j}}{\binom{1+1}{2}} 2^{1-j}4^j = 4$$

A partire da questi coefficienti possiamo ottenere i coefficienti del secondo monomio in due variabili:

$$-6\left(\begin{bmatrix}1\\ 4/3\\ 5/3\\ 2\end{bmatrix} \otimes \begin{bmatrix}2\\3\\4\end{bmatrix}\right) = -6\begin{bmatrix}1\\ 4/3\\ 5/3\\ 2\end{bmatrix}\begin{bmatrix}2 & 3 & 4\end{bmatrix} = \begin{bmatrix}-12 & -18 & -24\\ -16 & -24 & -32\\ -20 & -30 & -40\\ -24 & -36 & -48\end{bmatrix}$$

A questo punto per avere i coefficienti del polinomio in più variabili è sufficiente sommare gli oggetti ottenuti:

$$\begin{bmatrix}4 & 8 & 16\\ 8 & 16 & 32\\ 16 & 32 & 64\\ 32 & 64 & 128\end{bmatrix} + \begin{bmatrix}-12 & -18 & -24\\ -16 & -24 & -32\\ -20 & -30 & -40\\ -24 & -36 & -48\end{bmatrix} = \begin{bmatrix}-8 & -10 & -8\\ -8 & -8 & 0\\ -4 & 22 & 24\\ -8 & 28 & 80\end{bmatrix}$$

L'algoritmo è stato implementato per polinomi in due variabili in linguaggio MATLAB. Si è scelto di rappresentare il polinomio in forma di potenze come una matrice i cui due indici indicano i gradi delle rispettive variabili e l'elemento in posizione (i, j) indica il coefficiente di $x^{i-1}y^{j-1}$ (dal momento

che MATLAB conta gli indici partendo da 1). Dando quindi in input al programma la suddetta matrice di coefficienti `C` e una matrice $2 \times N$ `dom` in cui gli elementi in posizione $(1, k)$ e $(2, k)$ rappresentano gli estremi del dominio della k-esima variabile, si ottiene in output la matrice di coefficienti in forma di Bernstein. All'avvio viene inizializzato il vettore dei gradi assoluti e vengono ricavate le dimensioni della matrice di input in una variabile `dimM`:

```
g_ass = [0 0];
dimM = size(C);
```

Sono quindi calcolati gli effettivi gradi assoluti delle singole variabili con un doppio ciclo:

```
for i = 1 : dimM(1)
    for j = 1 : dimM(2)
        if(g_ass(1) < i && C(i,j) ~= 0)
            g_ass(1) = i-1;
        end

        if(g_ass(2) < j && C(i,j) ~= 0)
            g_ass(2) = j-1;
        end
    end
end
```

A questo punto, dopo aver inizializzato la matrice finale dei coefficienti di Bernstein del polinomio in due variabili, si inizia a scandire la matrice `C` con un doppio ciclo innestato. Quando viene identificato un coefficiente diverso da 0 si applica la function `monomio` due, uno o zero volte, a seconda se sono presenti entrambe le variabili, solo una delle due o nessuna (termine noto). Infatti una variabile non presente farà pervenire un vettore di coefficienti unitario. Una volta applicata il numero di volte necessario la function `monomio`, si moltiplicano i vettori risultati tramite prodotto tensoriale e si somma quanto ottenuto alla matrice soluzione `B_f`, inizializzata a 0 all'inizio del doppio ciclo.

La function `monomio` calcola i coefficienti di Bernstein di un monomio con una variabile, prendendo in input il grado assoluto `g_ass`, il grado relativo `g_rel` di tale variabile, oltre che gli estremi del dominio `massimo` e `minimo`. Se i due gradi coincidono, viene applicata la formula 2.5:

```
if(g_rel == g_ass)
    for k = 0 : g_ass
        b(k+1) = minimo^(g_rel - k) * massimo^(k);
    end
```

Altrimenti viene applicata la formula 2.7.

Di seguito ne diamo un esempio di applicazione, usando come polinomio $p(x) = x_1^3 x_2^2 - 6x_1 x_2$ con $x = [1, 2] \times [2, 4]$:

```
>> C = zeros(4,4);
>> C(2,2) = -6;
>> C(4,3) = 1;
>> dom(1,1) = 1;
>> dom(2,1) = 2;
>> dom(1,2) = 2;
>> dom(2,2) = 4;
>> toBernsteinDuo(C,dom)

ans =

    -8   -10    -8
    -8    -8     0
    -4     2    24
     8    28    80
```

Ovviamente il codice risponde correttamente anche nel caso particolare in cui il polinomio possieda una sola variabile. Usiamo come esempio il solito polinomio $p(x) = -8 + 65x - 150x^2 + 90x^3$ con $x \in [0, 1]$:

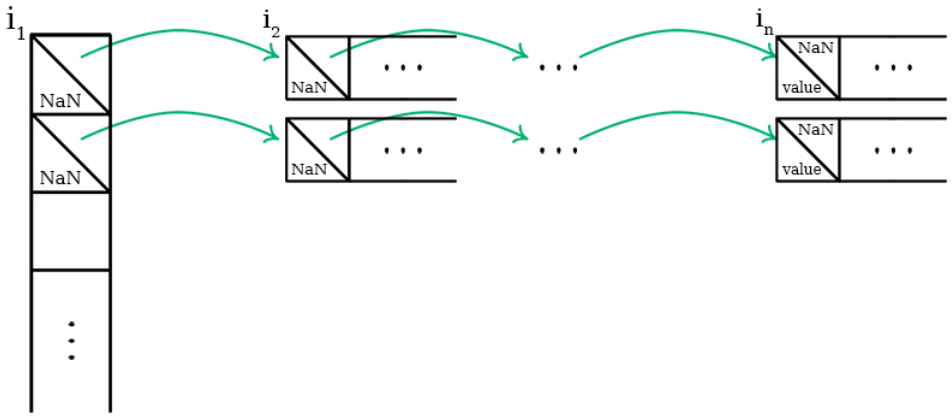
```
>> pol = zeros(4,4);
>> pol(1,1) = -8;
>> pol(2,1) = 65;
>> pol(3,1) = -150;
>> pol(4,1) = 90;
>> dom = zeros(2);
>> dom(2,1) = 1;
>> toBernsteinDuo(pol,dom)

ans =

   -8.0000
   13.6667
  -14.6667
   -3.0000
```

2.6.4 Generalizzazione dell'algoritmo

L'algoritmo di Smith è stato generalizzato per polinomi a più di due variabili. Per realizzare ciò non è agevole, come in precedenza, rappresentare i coefficienti del polinomio per mezzo di un vettore, di una matrice o, in questo caso, di una multi-matrice, dal momento che non è noto a priori l'ordine della matrice multi-dimensionale stessa. Si è pensato di usare allora un algoritmo ricorsivo ed è stata definita una struttura come segue:

$$tensor := \begin{cases} \text{campo } value \\ \text{campo } next \end{cases}$$

In MATLAB le strutture non si possono definire ma sono create dinamicamente aggiungendo un campo ad un oggetto tramite l'operatore punto, quindi il nome *tensor* è puramente indicativo. Non essendoci neanche distinzione tra tipi in MATLAB, specifichiamo che il campo *value* è inteso come un valore, mentre il campo *next* richiama il concetto di "puntatore ad un elemento" (in particolare è pensato come puntatore ad un altro oggetto "di tipo *tensor*"). In questo modo possiamo definire un vettore di oggetti *tensor* in cui il campo *next* di ogni cella "punta" ad un altro vettore di oggetti *tensor* e così via. Se il campo *next* è impostato a NaN, si è giunti in fondo alla struttura e l'elemento *value* corrispondente rappresenta il coefficiente di $x_1^{i_1-1} \cdot x_2^{i_2-1} \cdot \ldots \cdot x_n^{i_n-1}$. Ricordiamo infatti che MATLAB conta gli indici partendo da 1. Ovviamente le celle in cui il valore *next* non è impostato a NaN avranno invece il campo *value* a NaN, proprio ad indicare che la struttura prosegue. Rappresentiamo di seguito graficamente quanto appena descritto:



L'algoritmo prende in input una variabile `tensore`, contente la struttura rappresentata in figura, una matrice $2 \times N$ `dom` dei domini delle singole variabili e, per semplicità, il vettore dei gradi assoluti `ass`. Nel caso di un polinomio mono-dimensionale, viene chiamata la function `toBernsteinMono`, usata in precedenza, altrimenti viene chiamata la function ricorsiva `monomioN`. Questa controlla se esiste un'ulteriore dimensione (ossia se esiste un'ulteriore variabile) e, in quel caso, richiama se stessa su di essa, altrimenti, se si è giunti in fondo alla struttura, chiama la function `monomio`, vista in precedenza, un certo numero di volte e salva quanto ottiene, insieme al coefficiente di quel termine, in una struttura finale.

Nonostante sia stato testato un Tensor Toolbox [5] per l'ambiente MATLAB, sviluppato dalla SANDIA Corporation nei Sandia National Laboratories, i test fatti non hanno fornito i risultati sperati per poter implementare un pro-

dotto tensoriale efficace in questo algoritmo di conversione, mostrando alcune lacune e miglioramenti applicabili al Toolbox. Si è ritenuto sufficiente, allora, fare in modo che la function `toBernsteinN`, che compie questa conversione di base, restituisse la forma implicita del polinomio di Bernstein, formata dall'insieme dei vettori dei singoli monomi mono-variabile con i loro coefficienti. Per ottenere la forma finale effettiva del polinomio in base di Bernstein occorre, dunque, preso l'output dall'algoritmo, moltiplicare singolarmente i vettori ricavati dai monomi mono-variabile (con il rispettivo coefficiente) e sommare poi gli oggetti ricavati in questo modo.

Mostriamo di seguito un esempio di applicazione dell'algoritmo. Si voglia portare in base di Bernstein il polinomio di grado $N = (3, 2)$:

$$p(x) = x_1^3 x_2^3$$

con $\mathbf{x} = [1, 2] \times [2, 4]$ Creiamo, in prima istanza, la struttura tensoriale:

```
>> for k = 1 : 4
      tensoreA(k).value = NaN;
      tensoreA(k).next = tensoreA;
   end
>> for k = 1 : 4
       for kk = 1 : 4
          tensoreA(k).next(kk).value = 0;
          tensoreA(k).next(kk).next = NaN;
       end
   end
>> tensoreA(4).next(3).value = 1;
```

Creiamo poi la matrice di domini delle singole variabili e il vettore dei gradi assoluti:

```
>> dom(1,1) = 1;
>> dom(2,1) = 2;
>> dom(1,2) = 2;
>> dom(2,2) = 4;
>> ass(1) = 3;
>> ass(2) = 2;
```

Siamo quindi pronti per chiamare l'algoritmo principale:

```
>> toBernsteinN(tensoreA,dom,ass)

ans =

    coeff1: 1
        f1: [1 2 4 8]
        f2: [4 8 16]
```

I coefficienti di Bernstein voluti sono dati quindi dal prodotto tensoriale:

$$1 \left(\begin{bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{bmatrix} \otimes \begin{bmatrix} 4 \\ 8 \\ 16 \end{bmatrix} \right) = 1 \begin{bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{bmatrix} \begin{bmatrix} 4 & 8 & 16 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 16 \\ 8 & 16 & 32 \\ 16 & 32 & 64 \\ 32 & 64 & 128 \end{bmatrix}$$

E si ottiene in questo modo quanto richiesto.

2.7 Inviluppo convesso

In questo paragrafo si andrà ad analizzare una interessante proprietà dei polinomi di Bernstein che sarà particolarmente d'aiuto nei prossimi capitoli.

Definizione 2.5 (Inviluppo convesso)**.** Si definisce inviluppo convesso o involucro convesso di un qualsiasi insieme I di uno spazio vettoriale $\subseteq \mathbb{R}^n$ l'intersezione di tutti gli insiemi convessi che contengono I, dove per insieme convesso si intende un insieme nel quale, per ogni coppia di punti, il segmento che li congiunge è interamente contenuto nell'insieme.

Ovviamente se l'insieme I è convesso, il suo inviluppo è I stesso.

Proposizione 2.7.1. *Un polinomio $p(x)$ di grado N, con $N > 0$, sul dominio $\boldsymbol{x}$ è contenuto nell'inviluppo convesso dell'insieme delle coppie formate dai punti di controllo $^I/_N$ e dai suoi coefficienti di Bernstein.*

Esempio. Sia dato il polinomio $p(x) = -8 + 65x - 150x^2 + 90x^3$, con $x \in [0, 1]$. La sua forma di Bernstein, ricavata con l'algoritmo mostrato in questa sezione, è la seguente:

$$p(t) = -8B_0^{(3)} + 13,6667B_1^{(3)} - 14,6667B_2^{(3)} - 3B_3^{(3)}$$

I punti di controllo entro cui è contenuto l'intero polinomio sono:

$$P_1(^0/_3, -8) \qquad P_2(^1/_3, 13,6667)$$
$$P_3(^2/_3, -14,6667) \qquad P_4(^3/_3, -3)$$

Notiamo che il primo e l'ultimo punto di controllo sono sempre interpolanti il polinomio, e questo accade perché:

$$p(0) = \sum_{i=0}^{n} c_i B_i^{(n)}(0) = c_0, \qquad p(1) = \sum_{i=0}^{n} c_i B_i^{(n)}(1) = c_1$$

Questa proprietà è del tutto generale, ovvero vale che:

Proposizione 2.7.2. *Sia S_0 il sottoinsieme degli indici $S = \{I : I \leq N\}$ comprendenti gli indici corrispondenti al primo e all'ultimo dei coefficienti c_I, allora si ha $\forall I \in S_0 : c_I = p(\frac{I}{N})$.*

Banale conseguenza dell'inviluppo convesso, a questo punto, diviene la seguente proprietà, principale motivo per l'introduzione della rappresentazione in base di Bernstein.

Proposizione 2.7.3 (Proprietà di limitatezza)**.** *Sia $p(x)$ un polinomio di grado N nella variabile reale $x = (x_1, \ldots, x_n) \subseteq \mathbb{R}^n$, con $x \in \mathbf{x}$ e sia $p(t) = \sum_{I \leq N} c_I B_I^{(N)}(t)$ la sua rappresentazione in base di Bernstein. Allora $p(x)$ è limitato superiormente (risp. inferiormente) dal massimo (risp. minimo) coefficiente di Bernstein della sua rappresentazione, ovvero:*

$$\min_I(c_I) \leq p(x) \leq \max_I(c_I)$$

Concludiamo quindi questo capitolo con la seguente

Osservazione. La limitazione inferiore (risp. superiore) è esatta se e solo se il minimo (risp. il massimo) dei coefficienti di Bernstein viene raggiunto in corrispondenza di un vertice del dominio.

Questo vale in generale, a prescindere dal numero di variabili del polinomio.

Capitolo 3

Algoritmi di Consistenza

In questo capitolo verrà descritto il modo in cui la forma di Bernstein, presentata nel precedente capitolo, possa risultare utile per la risoluzione di vincoli polinomiali multi-variabile. Verranno descritti gli approcci utilizzati per risolvere il problema e saranno presentati i diversi algoritmi implementati. Come linguaggio di programmazione è stato usato MATLAB.

3.1 Uno sguardo al problema

Apriamo la trattazione presentando di seguito la definizione formale di vincolo polinomiale:

Definizione 3.1 (Vincolo polinomiale)**.** Sia $p(x) = \sum_{I \leq N} d_I x^I$ un polinomio nella variabile n-dimensionale $x = (x_1, \ldots, x_n)$ di grado $N = (n_1, \ldots, n_n)$ sul dominio $\mathbf{x} = [\underline{x}_1, \overline{x}_1] \times [\underline{x}_2, \overline{x}_2] \times \ldots \times [\underline{x}_n, \overline{x}_n]$. Si definisce vincolo polinomiale (o semplicemente vincolo) una relazione polinomiale del tipo $p(x) < 0$ oppure $p(x) = 0$.

La definizione 3.1 è completamente esaustiva, in quanto tutte le altre relazioni sono ad essa riconducibili. Infatti:

- $p(x) > 0$ può essere scritta come $-p(x) < 0$;
- $p(x) \leq 0$ si riconduce a $p(x) < 0 \wedge p(x) = 0$;
- $p(x) \geq 0$ si riconduce invece a $-p(x) < 0 \wedge p(x) = 0$;
- $p(x) \neq 0$, infine, si riconduce a $p(x) < 0 \vee -p(x) < 0$.

Osservazione. Il vincolo $p(x) \neq 0$ ha un comportamento esplosivo, nel senso che viene sostituito da due relazioni diverse. Se avessimo allora un sistema di

r vincoli e, di questi, k fossero del tipo $p(x) \neq 0$, allora, se lo si volesse portare nella forma mostrata poc'anzi, si genererebbero 2^k sistemi di r vincoli.

Un insieme di vincoli polinomiali costituisce un sistema di vincoli polinomiali. Formalmente:

Definizione 3.2. Siano $\{p_1(x), p_2(x), \ldots, p_n(x)\}$ un insieme di polinomi definiti sullo stesso insieme di variabili sul dominio $\mathbf{x}$ e siano $v_1(x), v_2(x), \ldots, v_n(x)$ vincoli polinomiali formati dai polinomi dell'insieme. Si definisce sistema di vincoli polinomiali il sistema di equazioni e/o disequazioni formato come segue:

$$\begin{cases} v_1(x) \\ v_2(x) \\ \vdots \\ v_n(x) \end{cases}$$

Dato, allora, un vincolo polinomiale $v(x)$ con $x \in [\underline{x}, \overline{x}]$, questa capitolo si propone di progettare algoritmi efficaci per restringere il dominio della variabile x, mantenendo consistente il vincolo stesso. Questi approcci vanno sotto il nome di *algoritmi di consistenza*, tecniche per propagare i vincoli prima dell'inizio della ricerca: riducono le dimensioni del problema originario eliminando dai domini delle variabili i valori esclusi dalle possibili soluzioni.

3.2 Un approccio tradizionale

Per codificare un algoritmo di riduzione del dominio per vincoli polinomiali si è pensato, in primo luogo, di sfruttare le trasformazioni affini dei polinomi e la proprietà di limitatezza della forma di Bernstein. Per codificare il tutto ci si è avvalsi del calcolo simbolico dell'ambiente MATLAB, considerando solamente vincoli mono-dimensionali, ossia in una sola variabile, imposti con il segno di $<$ (dato che come abbiamo appena visto, gli altri tipi di vincoli sono ad esso riconducibili). L'algoritmo prende in input il polinomio che costituisce il vincolo (in forma di potenze) `polinomio`, un valore che costituisce il vincolo stesso `val` e una tolleranza assegnata `toll` (si assume infatti che il dominio della variabile, salvato in un vettore `dom`, sia $[0, 1]$). Quindi si procede nel modo seguente:

1. Si porti in forma di Bernstein il vincolo polinomiale che si sta considerando.

2. Si identifichino il massimo e minimo coefficiente di Bernstein.

3. Si verifichi che il massimo e il minimo coefficiente di Bernstein siano entrambi minori o maggiori del valore imposto dal vincolo. Se entrambi sono minori (primo caso) si accetti l'intervallo considerato in questo frangente, mentre se sono entrambi maggiori (secondo caso) si rifiuti. Se non si può ancora dedurre nulla, invece, si proceda con il punto 4.

4. Si divida l'intervallo in due metà esatte, calcolando il punto medio.

5. Si applichi la trasformazione affine sul polinomio per il primo e il secondo intervallo ricavati dal punto 4.

6. Si iteri il procedimento per entrambi i polinomi ricavati a partire dal punto 2.

Per dividere l'intervallo viene usata una function ricorsiva `biseziona`. Prende in input il polinomio in forma di potenze `polinomio`, quello in base di Bernstein `pBernstein`, oltre che le variabili `val` e `toll` il cui significato è lo stesso descritto in precedenza. L'algoritmo controlla se il vincolo è verificato:

```
massimo = max(pBernstein);
minimo  = min(pBernstein);
if(massimo > val && minimo > val)
    D_res(1,1) = NaN; % Vincolo mai verificato
    D_res(2,1) = NaN;
elseif(massimo < val+toll && minimo < val+toll)
    D_res(1,1) = dom(1,1); % Vincolo sempre verificato
    D_res(2,1) = dom(2,1);
```

Se non lo è, divide il dominio in due metà, ricalcola i coefficienti sui nuovi domini e richiama la function `biseziona`. Quando arriva alla fine unisce i risultati ottenuti in una grande matrice `D_res` $2 \times N$ in cui gli elementi $(1, k)$ e $(2, k)$ sono gli estremi di una parte di dominio accettata:

```
middle = (dom(1,1) + dom(2,1))/2;

dom1(1,1) = dom(1,1);
dom1(2,1) = middle;
[~, polB1] = ricalcolaBernstein(pol, dom1(1,1), dom1(2,1));
D1 = biseziona(pol, polB1, dom1, val, toll);

dom2(1,1) = middle;
dom2(2,1) = dom(2,1);
[~, polB2] = ricalcolaBernstein(pol, dom2(1,1), dom2(2,1));
D2 = biseziona(pol, polB2, dom2, val, toll);

D_res = cat(2, D1, D2);
% Unifico i risultati in una matrice 2xN
```

La function `ricalcolaBernstein` è quella che si occupa di effettuare la trasformazione affine tramite il calcolo simbolico di MATLAB. Dato un polinomio `pol` definito in [`min`, `max`], restituisce i coefficienti del polinomio in forma di potenze e in base di Bernstein in $[0, 1]$. Per farlo viene risolto il sistema lineare che permette di compiere la trasformazione affine:

```
A = [0 1; 1 1];
b = [min, max];
sol = b/A;
```

Viene creato il polinomio simbolico:

```
symPol = sym(pol(1));
for i = 1 : grado
    symPol = symPol + sym(pol(i+1))*'x'^sym(i);
end
symPol = simplify(symPol);
```

Viene compiuta la sostituzione $x = \alpha\tilde{x} + \beta$ nel polinomio:

```
toChange = sym(sol(1))*'x' + sym(sol(2));
symPol = subs(symPol, 'x', toChange);
symPol = simplify(symPol);
polR = sym2poly(symPol);
```

Fatta qualche semplificazione e restituiti i due risultati `polRes` e `polResB`. Quest'ultimo in particolare contiene i coefficienti di Bernstein ricalcolati:

```
for k = 1 : length(polR)
   polRes(k) = polR(length(polR)-(k-1));
end
polResB = toBernsteinMono(polRes);
```

Esempio. Consideriamo il vincolo:

$$p(x) = -8 + 65x - 150x^2 + 90x^3 < 0$$

con $x \in [0, 1]$. Soddisfare il vincolo vuol dire determinare gli intervalli contenuti in $[0, 1]$ in cui la variabile indipendente verifica l'equazione.

Calcoliamo i coefficienti di Bernstein del polinomio applicando l'algoritmo visto nel capitolo precedente. Si ottengono i seguenti:

$$c_0 = -8 \qquad c_1 = 13,6667$$
$$c_2 = -14,6667 \qquad c_3 = -3$$

Osserviamo dunque che:

$$c_{\min} = -14,6667 \qquad c_{\max} = 13,6667$$

Non possiamo fare ancora alcuna deduzione, per cui dividiamo l'intervallo in due sottointervalli:

$$I_1 = [0, 1/2] \qquad\qquad I_2 = [1/2, 1]$$

Consideriamo il primo sottointervallo e applichiamo la trasformazione affine su $p(x) \to p(\tilde{x})$ con $x \in [0, 1/2]$ e $\tilde{x} \in [0, 1]$, usando la relazione $x = \alpha\tilde{x} + \beta$. Dobbiamo quindi risolvere il seguente sistema lineare:

$$\begin{cases} 0 = \alpha \cdot 0 + \beta \\ \frac{1}{2} = \alpha \cdot 1 + \beta \end{cases} \Rightarrow \begin{cases} \beta = 0 \\ \alpha = \frac{1}{2} \end{cases}$$

Quindi ricaviamo che $x = \frac{1}{2}\tilde{x}$, da cui:

$$\begin{aligned} p(\tilde{x}) &= 90\left(\frac{1}{2}\tilde{x}\right)^3 - 150\left(\frac{1}{2}\tilde{x}\right)^2 + 65\left(\frac{1}{2}\tilde{x}\right) - 8 \\ &= \frac{90}{8}\tilde{x}^3 - \frac{150}{4}\tilde{x}^2 + \frac{65}{2}\tilde{x} - 8 \end{aligned}$$

Sfruttando l'algoritmo di conversione in base di Bernstein su questo nuovo polinomio otteniamo i coefficienti di Bernstein su $I_1 = [0, 1/2]$:

$$\begin{aligned} c_0 &= -8 & c_1 &= 2,8333 \\ c_2 &= 1,1666 & c_3 &= -1,75 \end{aligned}$$

Osserviamo dunque che:

$$c_{\min} = -1,75 \qquad\qquad c_{\max} = 2,8333$$

Dal momento che non si può ancora dedurre nulla, è necessario dividere ancora l'intervallo I_1 nei due intervalli $I_3 = [0, 1/4]$ e $I_4 = [1/4, 1/2]$, ripetendo la trasformazione affine per il primo e il secondo.

Consideriamo ora $I_2 = [1/2, 1]$. Applichiamo la trasformazione affine su $p(x) \to p(\tilde{x})$, con $x \in [1/2, 1]$ e $\tilde{x} \in [0, 1]$, usando la relazione $x = \alpha\tilde{x} + \beta$. Dobbiamo quindi risolvere il sistema lineare:

$$\begin{cases} \frac{1}{2} = \alpha \cdot 0 + \beta \\ 1 = \alpha \cdot 1 + \beta \end{cases} \Rightarrow \begin{cases} \beta = \frac{1}{2} \\ \alpha = \frac{1}{2} \end{cases}$$

Quindi ricaviamo che $x = \frac{1}{2}\tilde{x} + \frac{1}{2}$, da cui:

$$\begin{aligned} p(\tilde{x}) &= 90\left(\frac{1}{2}\tilde{x} + \frac{1}{2}\right)^3 - 150\left(\frac{1}{2}\tilde{x} + \frac{1}{2}\right)^2 65\left(\frac{1}{2}\tilde{x} + \frac{1}{2}\right) - 8 \\ &= \frac{90}{8}\tilde{x}^3 - \frac{30}{8}\tilde{x}^2 - \frac{70}{8}\tilde{x} - \frac{14}{8} \end{aligned}$$

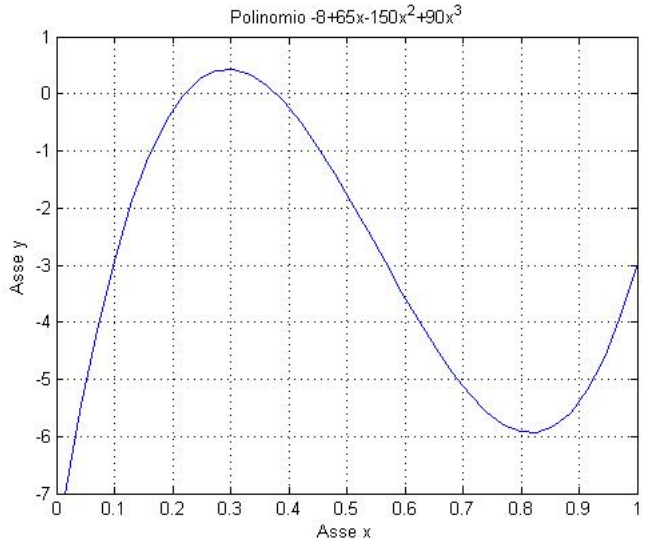
Sfruttando l'algoritmo di conversione in base di Bernstein su questo nuovo polinomio otteniamo i coefficienti di Bernstein su $I_1 = [1/2, 1]$:

$$c_0 = -1,75 \qquad c_1 = -4,6667$$
$$c_2 = -8,8333 \qquad c_3 = -3$$

Osserviamo dunque che:

$$c_{\min} = -8,8333 \qquad c_{\max} = -1,75$$

Quindi, applicando la proprietà di limitatezza, l'intervallo I_1 è interamente soluzione del vincolo. Per convincersene, è sufficiente confrontare i risultati ottenuti in questo esempio e quelli presentati di seguito ottenuti al calcolatore con il grafico rappresentate l'andamento di $p(x)$:



Di seguito presentiamo un esempio di applicazione dell'algoritmo in ambiente MATLAB:

```
>> pol = [-8 65 -150 90];
>> toll = 10^-1;
>> checkConstrain(pol,0,toll)
ans =

         0    0.1250    0.1875    0.2188    0.3750    0.5000
    0.1250    0.1875    0.2188    0.2266    0.5000    1.0000
```

In questo modo stiamo studiando il vincolo

$$p(x) = -8 + 65x - 150x^2 + 90x^3 < 0$$

con $x \in [0, 1]$. L'algoritmo ci restituisce gli intervalli accettati che soddisfano il vincolo e, come si vede confrontandoli con il grafico del polinomio, sono corretti a margine di un errore dato dalla tolleranza assegnata. Semplificando le cose, gli intervalli accettati, quindi, sono:

$$[0, 0.2266] \bigcup [0.3750, 1]$$

Abbiamo quindi con successo ridotto il dominio iniziale del polinomio.

3.3 Un approccio recente

In questa sezione verrà analizzato un algoritmo alternativo a quello presentato nel precedente paragrafo per la riduzione di domini per vincoli polinomiali in una variabile. Questo algoritmo è stato ricavato come conseguenza degli studi di P. S. V. Nataraj e M. Arounassalame e della proprietà di limitatezza dei polinomi di Bernstein e differisce principalmente dall'approccio con le trasformazioni affini nel metodo di ricalcolo dei coefficienti di Bernstein.

Si consideri quindi un polinomio in una variabile $p(x) = \sum_{k=0}^{n} d_k x^k$, definito sul dominio $\mathbf{x} = [\underline{x}, \overline{x}]$ e si supponga che la sua forma di Bernstein sia $p(x) = \sum_{k=0}^{n} c_k B_i^{(n)}(x)$. Si considerino inoltre i due domini:

$$\mathbf{x}_A = [\underline{x}, x_M] \qquad \mathbf{x}_B = [x_M, \overline{x}]$$

dove con x_M indichiamo il punto medio fra $\underline{x}$ e $\overline{x}$. Come nel precedente paragrafo, siamo interessati a calcolare i coefficienti di Bernstein sui due nuovi intervalli. Per farlo, posto per $j = 0, \ldots, n$:

$$c_j^{[0]} = c_j$$

ovvero posti come $c_j^{[0]}$ i coefficienti di Bernstein del polinomio di partenza, è possibile definire per $k = 1, \ldots, n$ e per $i = 0, \ldots, n$ la seguente relazione matematica:

$$c_i^{[k]} = \begin{cases} c_i^{[k-1]}, & \text{se } i < k \\ (1-\lambda)c_{i-1}^{[k-1]} + \lambda c_i^{[k-1]}, & \text{se } i \geq k \end{cases} \tag{3.1}$$

dove λ è detto parametro di suddivisione. Dal momento che l'intervallo viene sempre diviso in due metà uguali è possibile, senza perdita di generalità, porre $\lambda = \frac{1}{2}$. L'insieme dei nuovi coefficienti di Bernstein $\tilde{c}_i$ su $\mathbf{x}_A$ è dato per $i = 0, \ldots, n$ da:

$$\tilde{c}_i(\mathbf{x}_A) = c_i^{[n]}$$

mentre l'insieme dei coefficienti $\tilde{c}_i$ su $\mathbf{x}_B$ è possibile ottenerlo per simmetria per $k = 0, \ldots, n$ usando la relazione:

$$\tilde{c}_{n-k}(\mathbf{x}_B) = c_n^{[n-k]}$$

A questo punto, l'algoritmo usato dovrebbe risultare chiaro:

1. Si porti in forma di Bernstein il vincolo polinomiale che si sta considerando.
2. Si identifichino il massimo e minimo coefficiente di Bernstein.
3. Si verifichi che il massimo e il minimo coefficiente di Bernstein siano entrambi minori o maggiori del valore imposto dal vincolo. Se entrambi sono minori (primo caso) si accetti l'intervallo considerato in questo frangente, mentre se sono entrambi maggiori (secondo caso) si rifiuti. Se non si può ancora dedurre nulla, invece, si proceda con il punto 4.
4. Si divida l'intervallo in due metà esatte, calcolando il punto medio.
5. Si applichi la formula 3.1 sul polinomio ricavando i coefficienti di Bernstein sui due nuovi intervalli calcolati al punto 4.
6. Si iteri il procedimento per entrambi i polinomi ricavati a partire dal punto 2.

Esempio. Consideriamo il vincolo:

$$p(x) = -8 + 65x - 150x^2 + 90x^3 < 0$$

con $x \in [0, 1]$. Calcoliamo i coefficienti di Bernstein del polinomio:

$$c_0 = -8 \qquad c_1 = 13,6667 \qquad c_2 \quad = -14,6667 \qquad c_3 = -3$$

Come già osservato in precedenza $c_{\min} = -1,75$ e $c_{\max} = 2,8333$ per cui, per la proprietà di limitatezza, non si può ancora dedurre nulla. Dividiamo quindi l'intervallo $[0, 1]$ in due sottointervalli. Si ottengono:

$$I_1 = [0, 1/2] \qquad\qquad I_2 = [1/2, 1]$$

Applichiamo a questo punto la formula 3.1. Nel nostro caso $n = 3$, per cui si ricavano i valori:

$$c_0^{[1]} = -8 \qquad c_1^{[1]} = 2,8333$$
$$c_2^{[1]} = -0,5 \qquad c_3^{[1]} = -8,8333$$
$$c_0^{[2]} = -8 \qquad c_1^{[2]} = 2,8333$$
$$c_2^{[2]} = \frac{1}{2}c_1^{[1]} + \frac{1}{2}c_1^{[2]} = 1,1666 \qquad c_3^{[2]} = \frac{1}{2}c_2^{[1]} + \frac{1}{2}c_3^{[1]} = -4,6667$$
$$c_0^{[3]} = -8 \qquad c_1^{[3]} = 2,8333$$
$$c_2^{[3]} = 1,1666 \qquad c_3^{[3]} = \frac{1}{2}c_2^{[2]} + \frac{1}{2}c_3^{[2]} = -1,75$$

I nuovi coefficienti di Bernstein sull'intervallo $I_1 = [0, 1/2]$ dunque sono:

$$c_0 = -8 \qquad c_1 = 2,8333 \qquad c_2 = 1,1666 \qquad c_3 = -1,75$$

Mentre i nuovi coefficienti di Bernstein sull'intervallo $I_2 = [1/2, 1]$ sono:

$$c_{3-0} = c_3^{[3]} = -1,75 \qquad c_{3-1} = c_3^{[3-1]} = -4,6667$$
$$c_{3-2} = c_3^{[3-2]} = -8,8333 \qquad c_{3-3} = c_3^{[3-3]} = -3$$

A questo punto bisogna sfruttare la proprietà di limitatezza e, nel caso non riesca a dare una risposta definitiva, riapplicare la formula 3.1 dividendo ancora gli intervalli su cui non è possibile prendere una decisione.

L'algoritmo è stato implementato in ambiente MATLAB. La function `NatarajCheckConstraint` è il programma principale, mentre la function `NatarajBiseziona` compie fondamentalmente la stessa procedura che faceva nel paragrafo precedente la function `biseziona` (con le dovute piccole modifiche). Ciò che davvero differisce con l'algoritmo precedente è la function `NatarajRicalcolaBernstein`. Questa prende in input il polinomio in forma di Bernstein `pBernstein` e il grado del polinomio `degree`. Quando viene chiamata, inserisce i coefficienti di Bernstein noti (che gli abbiamo passato) sulla prima riga di una matrice:

```
for i = 1 : degree + 1
    M(1,i) = pBernstein(i);
    % Matrice delle soluzioni parziali
end
```

Dopodiché applica la formula 3.1:

```
for k = 2 : degree + 1
    for i = 1 : degree + 1
        if(k > i) % Formula di Nataraj
            M(k,i) = M(k-1,i);
        else % k <= i
            M(k,i) = (0.5*M(k-1,i-1)) + (0.5*M(k-1,i));
        end
    end
end
```

Infine vengono estratti gli effettivi coefficienti di Bernstein sulla prima metà dell'intervallo e sulla seconda metà e salvati e restituiti dalla function rispettivamente in una variabile `polB1` e `polB2`.

Di seguito forniamo un esempio di applicazione usando come test il solito vincolo

$$-8 + 65x - 150x^2 + 90x^3 < 0$$

con $x \in [0, 1]$:

```
>> pol = [-8 65 -150 90];
>> toll = 10^-1;
>> NatarajCheckConstrain(pol,0,toll)
ans =

         0    0.1250    0.1875    0.2188    0.3750    0.5000
    0.1250    0.1875    0.2188    0.2266    0.5000    1.0000
```

3.4 Risultati a confronto

I due algoritmi visti nei due paragrafi precedenti sono corretti e gli esperimenti fatti al calcolatore hanno dato risultati simili per entrambi in ogni test eseguito. Si è pensato di confrontare, a questo punto, il tempo d'esecuzione delle due procedure, essendo emerso il sospetto che l'implementazione della procedura tradizionale, usando pesantemente il calcolo simbolico di MATLAB, potesse ottenere risultati peggiori. È stato dunque considerato un vincolo campione:

$$p(x) = -8 + 65x - 150x^2 + 90x^3 < 0$$

Si è eseguito un esperimento di prove ripetute per $p(x)$, lanciando 1000 volte i due algoritmi e misurando il tempo impiegato ad ogni step del ciclo tramite le primitive `tic` e `toc` di MATLAB. Raccolti i dati, si è calcolata la media:

Algoritmo Classico	0.778605 ± 0.005414 sec
Algoritmo di Nataraj	0.001474 ± 0.000002 sec

L'errore sulla media è stato calcolato dividendo la standard deviation delle misure fatte con la radice quadrata del numero di misure, ossia è stata dapprima calcolata la std:

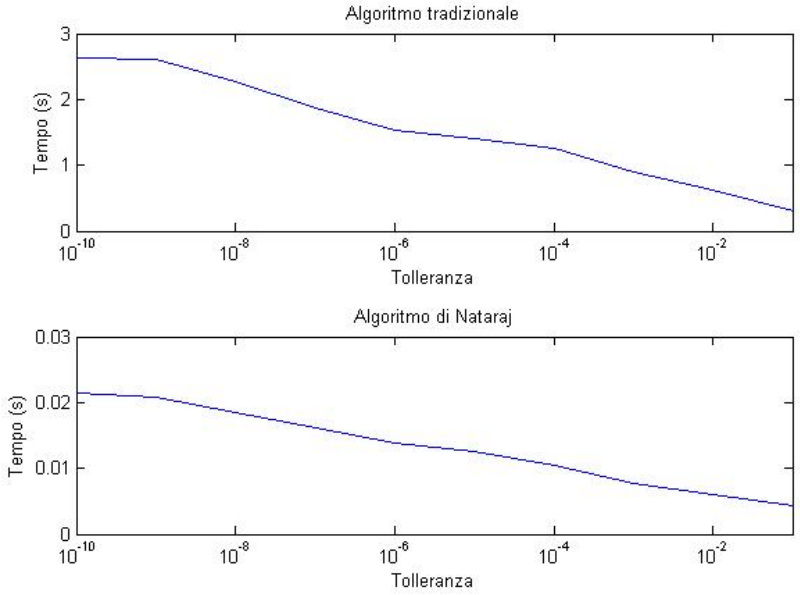
$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}}$$

dove gli x_i sono i campioni misurati, $\overline{x}$ è la media, e n il numero di misure. In seguito si è ricavato l'errore effettivo sulla media applicando la seguente relazione:

$$err = \frac{\sigma}{\sqrt{n}}$$

Come si era immaginato, l'algoritmo che sfrutta le trasformazioni affini ottiene risultati decisamente peggiori. Ovviamente questi risultati sono da considerare solo dei campioni. Affinché la bontà di un algoritmo venga effettivamente provata, occorrerebbe ripetere l'esperimento molte volte e per un campione significativo di dati di input.

A questo punto, si è pensato di confrontare anche l'andamento del tempo di esecuzione dei due algoritmi al variare della tolleranza assegnata come input (usando sempre lo stesso polinomio d'esempio), ricavando il seguente andamento su scala semi-logaritmica:



Anche in questo caso all'aumentare della tolleranza il tempo di esecuzione si abbassa per entrambe le procedure, ma l'approccio tradizionale con una tolleranza pari a 0.01 sfiora il secondo, mentre l'approccio che sfrutta la formula di Nataraj in quel caso impiega meno di 0.01 secondi per restituire l'output.

I risultati sono stati ottenuti su un calcolatore a 2.5 Ghz usando come sistema operativo Microsoft Windows 7.

3.5 Vincoli in due variabili

L'algoritmo realizzato implementando la formula 3.1 è in realtà molto più generale di quanto si pensi e la formula stessa è valida non solo per polinomi ad una dimensione ma per polinomi a più dimensioni. In questo paragrafo, analizzeremo un algoritmo per restringere i domini di vincoli polinomiali bi-dimensionali. L'algoritmo è esattamente quello presentato nel paragrafo 3.3. Ciò che cambia, ancora una volta, è il metodo per ricalcolare i coefficienti di Bernstein.

Consideriamo allora un polinomio in due variabili $p(x) = \sum_{I \leq N} d_I x^I$ con $N = (n_1, n_2)$, $x = (x_1, x_2)$ e $\mathbf{x} = [\underline{x}_1, \overline{x}_1] \times [\underline{x}_2, \overline{x}_2]$. Possiamo dividere questo dominio in due modi:

$$\mathbf{x}_A = [\underline{x}_1, m(x_1)] \times [\underline{x}_2, \overline{x}_2]$$
$$\mathbf{x}_B = [m(x_1), \overline{x}_1] \times [\underline{x}_2, \overline{x}_2]$$

Oppure come:

$$\mathbf{x}_A = [\underline{x}_1, \overline{x}_1] \times [\underline{x}_2, m(x_2)]$$
$$\mathbf{x}_B = [\underline{x}_1, \overline{x}_1] \times [m(x_2), \overline{x}_2]$$

dove $m(x_i)$ indica il punto medio dell'intervallo i-esimo. Allora, portato in base di Bernstein il polinomio $p(x)$, si ottengono i coefficienti $B(x)$ (salvabili ad esempio in una matrice bi-dimensionale). Partendo dunque con $B^0(x) = B(x)$ per $k = 1, \ldots, n_r$, per $i_1 = 0, \ldots, n_1$ e $i_2 = 0, \ldots, n_2$, se scegliamo di dividere l'intervallo nel primo modo vale:

$$c^{[k]}_{(i_1,i_2)} = \begin{cases} c^{[k-1]}_{(i_1,i_2)}, & \text{se } i_1 < k \\ (1-\lambda)c^{[k-1]}_{(i_1-1,i_2)} + \lambda c^{[k-1]}_{(i_1,i_2)}, & \text{se } i_1 \geq k \end{cases} \tag{3.2}$$

E quindi i nuovi coefficienti di Bernstein su $\mathbf{x}_A$ sono $B(\mathbf{x}_A) = B^{(n_1)}(\mathbf{x})$ mentre quelli su $\mathbf{x}_B$ sono $c_{(n_1-k,i_2)}(\mathbf{x}_B) = c^{(n_1-k)}_{(n_1,i_2)}$ per simmetria. Se scegliamo di

dividere l'intervallo nel secondo modo, invece, vale:

$$c_{(i_1,i_2)}^{[k]} = \begin{cases} c_{(i_1,i_2)}^{[k-1]}, & \text{se } i_2 < k \\ (1-\lambda)c_{(i_1,i_2-1)}^{[k-1]} + \lambda c_{(i_1,i_2)}^{[k-1]}, & \text{se } i_2 \geq k \end{cases} \tag{3.3}$$

E quindi questa volta i nuovi coefficienti di Bernstein su $\mathbf{x}_A$ sono $B(\mathbf{x}_A) = B^{(n_2)}(\mathbf{x})$ mentre quelli su $\mathbf{x}_B$ sono $c_{(i_1,n_2-k)}(\mathbf{x}_B) = c_{(i_1,n_2)}^{(n_2-k)}$ per simmetria.

L'algoritmo implementato prende in input una matrice `polinomio` dei coefficienti del polinomio in forma di potenze, due vettori dei domini `dom1` e `dom2` delle singole variabili, un valore che costituisce il vincolo `val` e una tolleranza `toll` e restituisce una matrice $4 \times N$ `M_res`, in cui ogni colonna rappresenta una porzione del dominio della superficie tridimensionale. I primi due valori si riferiscono alla prima variabile mentre gli altri due riguardano la seconda variabile. L'euristica implementata per scegliere in quale direzione eseguire il taglio del dominio sceglie quello più grande (ossia con più valori possibili). All'avvio della procedura, viene portato il polinomio in forma di Bernstein e si controlla se è possibile una deduzione immediata (dominio interamente accettato o rifiutato). Nel caso in cui questo non sia possibile, viene chiamata la function `NatarajBisezionaDuo`. Questa compie la bisezione del dominio applicando l'euristica definita poco prima e chiama due volte (su entrambi gli intervalli generati) la function `NatarajRicalcolaBernsteinDuo`, la quale controlla in che direzione si è scelto di tagliare il dominio e applica, a seconda della scelta fatta, la formula 3.2 oppure la 3.3. Infine estrae i risultati effettivi dalle soluzioni parziali calcolate e li inserisce in due matrici `B1` e `B2` che vengono poi restituite.

Mostriamo di seguito un esempio di applicazione dell'algoritmo. Consideriamo il vincolo:

$$p(x) = x_1 - x_1 x_2 < 0.5$$

con $\mathbf{x} = [0,1] \times [0,1]$. Il polinomio $p(x)$ possiede l'andamento mostrato in figura 3.1.

Proviamo a restringerne il dominio, applicando l'algoritmo appena descritto. Definiamo la matrice di coefficienti:

```
>> C(1,1) = 0;
>> C(2,1) = 1;
>> C(1,2) = 0;
>> C(2,2) = -1
```

Definiamo poi i domini e gli altri parametri d'uso:

```
>> domA(1,1) = 0;
>> domA(2,1) = 1;
>> domB = dom1;
```

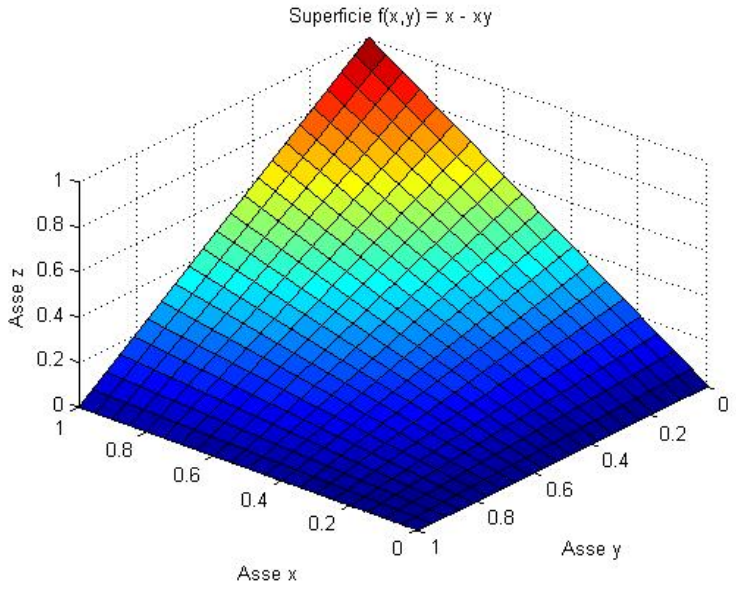


Figura 3.1: Andamento del polinomio $p(x) = x_1 - x_1x_2 < 0.5$

```
>> val = 0.5;
>> toll = 10^-1;
```

Adesso siamo pronti per chiamare la function principale:

```
>> NatarajCheckConstraintDuo(C, domA, domB, val, toll)

ans =

  Columns 1 through 6

         0    0.5000    0.5625    0.5000    0.6250    0.5000
    0.5000    0.5625    0.6250    0.6250    0.6875    0.7500
         0         0    0.0625    0.1250    0.1875    0.2500
    1.0000    0.1250    0.1250    0.2500    0.2500    0.5000

  Columns 7 through 11

    0.7500    0.7500    0.8750    0.9375    0.5000
    0.8125    0.8750    0.9375    1.0000    1.0000
    0.3125    0.3750    0.3750    0.4375    0.5000
    0.3750    0.5000    0.5000    0.5000    1.0000
```

Le prime suddivisioni dei domini fatte dall'algoritmo sono mostrate nella figura 3.2. Per essere sicuri della correttezza dell'algoritmo è stato implementato un semplice algoritmo di valutazione per polinomi di Bernstein in

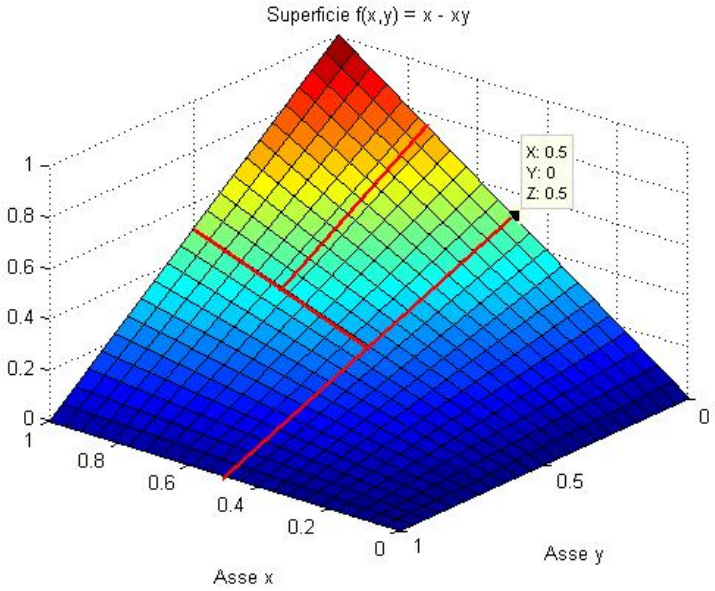


Figura 3.2: Prime suddivisioni dei domini compiute dall'algoritmo

due variabili, che procede ad effettuare il seguente calcolo:

$$B_I^N(x) = B_{i_1,i_2}^{(n_1,n_2)}(x,y) = \binom{n_1}{i} \frac{(b-x)^{n_1-i}(x-a)^i}{(b-a)^{n_1}} \binom{n_2}{j} \frac{(d-y)^{n_2-j}(y-x)^j}{(d-c)^{n_2}}$$

dopodiché si è controllato se nei punti di tangenza (le linee rosse in figura) di due porzioni differenti della superficie (ovviamente tangenti) i polinomi calcolati assumessero lo stesso valore.

3.6 Generalizzazione della procedura

Come anticipato nel precedente paragrafo, la formula 3.1 ha validità generale e può essere applicata a polinomi ad una o più dimensioni. Usando la notazione multi-indice introdotta nel Capitolo 3, possiamo allora dedurre la formula per il ricalcolo dei coefficienti di Bernstein per polinomi a più variabili.

Sia l il numero di variabili, $x = (x_1, \ldots, x_l) \in \mathbb{R}^l$ e $p(x) = \sum_{I \leq N} d_I x^I$ un polinomio in forma di potenze. Inoltre sia $\mathbf{x} = [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_l, \overline{x}_l]$ il dominio della multi-variabile x. Il polinomio $p(x)$ in base di Bernstein ha forma $p(x) = \sum_{I \leq N} c_I B_I^N(x)$. Una bisezione nella direzione r-esima, con $1 \leq r \leq l$ è una bisezione perpendicolare lungo quella direzione. Ossia dato:

$$\mathbf{x} = [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_r, \overline{x}_r] \times \cdots \times [\underline{x}_l, \overline{x}_l]$$

Si supponga di applicare la bisezione lungo la direzione r-esima. Si ottengono due nuovi intervalli:

$$\mathbf{x}_A = [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_r, m(x_r)] \times \cdots \times [\underline{x}_l, \overline{x}_l]$$
$$\mathbf{x}_B = [\underline{x}_1, \overline{x}_1] \times \cdots \times [m(x_r), \overline{x}_r] \times \cdots \times [\underline{x}_l, \overline{x}_l]$$

dove $m(x_r)$ indica il punto medio dell'intervallo r-esimo. Allora, partendo da $B^0(x) = B(x)$, ossia considerando come c_I^0 i coefficienti di Bernstein iniziali, possiamo ricalcolare i nuovi coefficienti su $\mathbf{x}_A$ e $\mathbf{x}_B$ applicando la formula seguente per $k = 1, 2, \ldots, n_r$:

$$c_I^{[k]} = \begin{cases} c_I^{[k-1]}, & \text{se } i_r < k \\ (1-\lambda)c_I^{[k-1]} + \lambda c_I^{[k-1]}, & \text{se } i_r \geq k \end{cases} \tag{3.4}$$

dove λ è il solito parametro di suddivisione. Per ottenere i nuovi coefficienti la formula deve essere applicata per $i_j = 0, \ldots, n_j$ e per $j = 1, \ldots, r-1, r+1, \ldots, l$. I nuovi coefficienti su $\mathbf{x}_A$ saranno allora:

$$B(\mathbf{x}_A) = B^{n_r}(\mathbf{x})$$

Mentre i nuovi coefficienti su $\mathbf{x}_B$ sono calcolati per simmetria:

$$c_{i_1,\ldots,n_r-k,\ldots,i_l}(\mathbf{x}_B) = c^{(n_r-k)}_{i_1,\ldots,n_r,\ldots,i_l}$$

3.7 Selezione della direzione di suddivisione

La suddivisione della matrice dei coefficienti di Bernstein può essere eseguita lungo qualsiasi direzione. Possiamo definire allora una funzione di merito avente la seguente forma:

$$k := \min\{j \ : \ j \in \{1, 2, \ldots, l\},\ y(j) = \max y(r),\ r = 1, 2, \ldots, l\}$$

dove $y(r)$ rappresenta il *fattore di direzione* ed è determinato da una regola stabilita. La direzione r per cui $y(r)$ è massimo è scelta per la suddivisione. Se più direzioni ottengono lo stesso valore di $y(r)$ viene convenzionalmente presa quella più bassa.

Una regola collaudata usata per l'analisi di intervalli è la cosiddetta *maximum width*, in cui il fattore di direzione è posto a:

$$y(r) = \texttt{wid}\ \mathbf{x}_r$$

dove $\texttt{wid}\ \mathbf{x}_r$ indica la larghezza del dominio nella direzione r. La direzione di suddivisione scelta è quella per cui $y(r)$ risulta massimo.

3.8 Bernstein Box Consistency

In questa sezione illustreremo un algoritmo di consistenza, sperimentato in questi ultimi anni, chiamato *Bernstein Box Consistency (BBC)* [2]. Prima di illustrare, tuttavia, la procedura, è necessario introdurre preliminarmente alcune nozioni sull'aritmetica di intervalli.

3.8.1 Aritmetica di Intervalli

L'aritmetica di intervalli, conosciuta anche come analisi di intervalli o matematica di intervalli o anche computazione di intervalli, è un metodo sviluppato fra il 1950 e il 1960 come approccio per porre limiti a errori di arrotondamento e errori di misura nel calcolo matematico, e quindi per giungere a risultati più affidabili. In termini semplici, nell'aritmetica di intervalli ogni valore rappresenta in realtà un range di possibilità. Per esempio, invece di stimare la lunghezza di un oggetto utilizzando l'aritmetica standard a 1.0 metro, possiamo essere certi nell'aritmetica di intervallo che la misura si attesta fra 0.97 e 1.3 metri. Mentre l'aritmetica standard definisce operazioni su numeri individuali, quindi, l'aritmetica di intervalli definisce operazione su intervalli. Le operazioni di base dell'analisi di intervalli sono, dati due intervalli $[a, b]$ e $[c, d]$, che sono un sottoinsieme della retta reale $(-\infty, \infty)$, le seguenti:

- $[a, b] + [c, d] = [a + c, b + d]$
- $[a, b] - [c, d] = [a - d, b - c]$
- $[a, b] \cdot [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$
- $\frac{[a,b]}{[c,d]} = \left[\min\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right), \max\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right)\right]$ dove $0 \notin [c, d]$

L'operazione di divisione di un intervallo in cui il divisore contiene contiene lo zero non è definita nell'aritmetica di intervalli di base. Per gli scopi di questa trattazione non estenderemo questa nozione. L'addizione e il prodotto sono operazioni che godono delle proprietà commutativa e associativa. Inoltre godono della proprietà sub-distributiva, ossia dati tre intervalli X, Y e Z l'insieme $X(Y + Z)$ è un sottoinsieme di $XY + XZ$. Invece di lavorare con un valore reale x incerto, l'aritmetica di intervalli opera con i due estremi di un intervallo $[a, b]$ che contiene x: x giace fra a e b oppure è uno di questi due valori. Allo stesso modo, l'applicazione di una funzione f ad x fornisce un risultato incerto. Nell'aritmetica degli intervalli f produce un intervallo $[c, d]$ che corrisponde a tutti i valori di $f(x)$ per $x \in [a, b]$. L'uso più comune dell'aritmetica di intervallo è quello di tener traccia e gestire gli errori di

arrotondamento e le incertezze direttamente durante il calcolo, con lo scopo di conoscere i valori esatti di parametri fisici e tecnici. Questi ultimi spesso derivano da errori di misura, tolleranze o a causa di limiti alla precisione di calcolo. L'aritmetica intervallo aiuta anche a trovare soluzioni affidabili a equazioni e problemi di ottimizzazione.

3.8.2 Algoritmo BBC

L'Algoritmo BBC è un algoritmo che permette di ridurre il dominio di ricerca di una variabile in un CSP. Di fatto, al contrario delle procedure esposte nei paragrafi precedenti, non frammenta potenzialmente il dominio in tanti pezzi, ma ne contrae gli estremi. Descriviamo di seguito la procedura per vincoli in due dimensioni (facilmente estendibile a vincoli globali).

Consideriamo un vincolo di diseguaglianza del tipo $g(x_1, x_2) \leq 0$, e siano $B(\mathbf{x}_1, \mathbf{x}_2)$ i coefficienti di Bernstein di g calcolati sul dominio $\mathbf{x} = \mathbf{x}_1 \times \mathbf{x}_2$. L'algoritmo va iterato per i domini di ogni variabile. Sia allora $\mathbf{x}_1 = [a, b]$ il dominio della prima variabile. Con il metodo BBC si cerca di incrementare il valore di a e di decrementare il valore di b, per ridurre la larghezza di $\mathbf{x}_1$. Per aumentare il valore di a, occorre calcolare prima la funzione $g(a, x_2)$, in questo modo si ottiene una funzione ad una sola variabile. Dopodiché $g(a, x_2)$ viene portata in forma di Bernstein $B(x_2)$ e si ricava l'intervallo che denominiamo $\mathbf{g}(a)$ come segue:

$$\mathbf{g}(a) = [\min B(x_2), \max B(x_2)]$$

Ossia $\mathbf{g}(a)$ è l'intervallo formato dalla coppia costituita dal più piccolo e dal più grande coefficiente di Bernstein di $g(a, x_2)$. A questo punto si distinguono due casi:

- Se $\mathbf{g}(a)$ è un intervallo non completamente positivo, non possiamo incrementare a.
- Se $\mathbf{g}(a)$ è un intervallo completamente positivo, vuol dire che il vincolo $g(x_1, x_2)$ è insoddisfatto all'end-point a. Quindi, a partire da a lungo $\mathbf{x}_1 = [a, b]$ possiamo cercare il primo punto in cui $g(x_1, x_2)$ è soddisfatto, ossia cerchiamo uno zero di $\mathbf{g}$. Denotiamo questo zero con a'. Chiaramente $g(x_1, x_2)$ è insoddisfatto su $[a, a')$ e quindi possiamo scartarlo per ottenere la contrazione $[a', b]$. Lo stesso discorso può essere fatto al viceversa, considerando l'end-point b e tentando di decrementarlo.

Per trovare uno zero di $\mathbf{g}$ in $[a, b]$, possiamo applicare una iterazione (una sola per risparmiare tempo di calcolo, ma volendo anche più di una) della

versione mono-variabile del metodo di Bernstein-Newton, ossia una variante del Metodo di Newton che opera su intervalli:

$$\mathbf{N}(\mathbf{x}_1) = a - \frac{\mathbf{g}(a)}{\mathbf{g}'_{x_1}} \tag{3.5}$$

dove rispettivamente a indica l'intervallo $[a, a]$, $g(a)$ denota invece l'intervallo $[\min B(x_2), \max B(x_2)]$ ricavato prima, mentre $\mathbf{g}'_{x_1}$ è l'intervallo ricavato come segue: si calcola la derivata parziale lungo x_1 (la variabile di cui si sta considerando il dominio) di $g(x_1, x_2)$, dopodiché si computa $g'_{x_1}(a, x_2)$. Di questa nuova funzione mono-variabile si calcolano i coefficienti di Bernstein $\overline{B}(x_2)$. L'intervallo $\mathbf{g}'_{x_1}$ è costituito quindi da:

$$\mathbf{g}'_{x_1} = [\min \overline{B}(x_2), \max \overline{B}(x_2)]$$

Il calcolo della formula 3.5 va fatto, come anticipato prima, in aritmetica di intervalli. Una volta ricavato $\mathbf{N}(\mathbf{x}_1)$ il nuovo dominio $\mathbf{x}'_1$ viene ottenuto computando:

$$\mathbf{x}'_1 = \mathbf{x}_1 \bigcap \mathbf{N}(\mathbf{x}_1)$$

Da questo calcolo due panorami sono possibili:

- $\mathbf{x}'_1$ coincide con l'intervallo di partenza $\mathbf{x}_1$.
- $\mathbf{x}'_1$ è un dominio contratto rispetto a $\mathbf{x}_1$.

Esempio. Consideriamo il seguente vincolo di diseguaglianza:

$$g(x) = x_2 - 2 - 2x_1^4 + 8x_1^3 - 8x_1^2 \leq 0$$

su $\mathbf{x}_1 = [1.5, 2.25]$ e $\mathbf{x}_2 = [3, 4]$. I coefficienti di Bernstein di $g(x)$ sono:

$$B(\mathbf{x}) = \begin{pmatrix} -0.1250 & 0.8750 \\ 0.4375 & 1.4375 \\ 1.0938 & 2.0938 \\ 1.4219 & 2.4219 \\ 0.3672 & 1.3672 \end{pmatrix}$$

Ricordiamo che i coefficienti di Bernstein di $g(a, x_2)$ si trovano sulla prima riga, i coefficienti di $g(b, x_2)$ si trovano sull'ultima riga, i coefficienti di $g(x_1, a)$ si trovano sulla prima colonna, mentre i coefficienti di $g(x_1, b)$ si trovano sulla seconda colonna. Applichiamo l'algoritmo BBC lungo $\mathbf{x}_2$. Cerchiamo di incrementare l'end-point $a = 3$. Da quanto appena detto, il minimo e il massimo coefficiente di Bernstein sulla prima colonna sono rispettivamente

-0.1250 e 1.4219, dandoci quindi $\mathbf{g}(a) = [-0.1250, 1.4219]$, che non è un intervallo completamente positivo. Quindi non possiamo incrementare $a = 3$.

Cerchiamo di decrementare allora l'end-point $b = 4$. Il minimo e il massimo coefficiente di Bernstein sulla seconda colonna sono rispettivamente 0.8750 e 2.4219, dandoci quindi $\mathbf{g}(a) = [0.8750, 2.4219]$, che è un intervallo completamente positivo. Quindi possiamo decrementare b usando la tecnica del BBC. Per farlo dobbiamo calcolare $\mathbf{g}'_{x_2}$. Ora la derivata parziale di g lungo x_2 vale:

$$g'_{x_2} = 1$$

e di conseguenza:

$$g'_{x_2}(x_1, b) = 1$$

e portare $g'_{x_2}(x_1, b)$ in forma di Bernstein ci fornisce come risultato sempre 1. Quindi l'intervallo che ci interessa è $\mathbf{g}'_{x_2} = [1, 1]$. Una iterazione della Formula 3.5 di Bernstein-Newton ci dà:

$$\mathbf{N}(\mathbf{x}_2) = b - \frac{\mathbf{g}(b)}{g'_{x_2}} = [4, 4] - \frac{[0.8750, 2.4219]}{[1, 1]} = [1.5781, 3.1251]$$

operando in aritmetica di intervalli. Il nuovo dominio $\mathbf{x}_2$ sarà allora:

$$\mathbf{x}'_2 = \mathbf{N}(\mathbf{x}_2) \bigcap \mathbf{x}_2 = [1.5781, 3.1251] \bigcap [3, 4] = [3, 3.1251]$$

Siamo riusciti quindi a restringere l'intervallo iniziale per la variabile x_2 contraendo l'end-point b.

L'algoritmo è stato implementato in linguaggio MATLAB per vincoli in due variabili. Viene preso come input la matrice dei coefficienti del polinomio, il polinomio simbolico e la solita matrice 2×2 contenente i domini delle variabili (espressi per colonne). Mostriamo di seguito unicamente quello che succede per l'analisi del secondo estremo della seconda variabile. Gli altri casi sono analoghi, fatte le dovute modifiche. Detta `polB` la matrice dei coefficienti di Bernstein del polinomio, viene calcolato innanzi tutto l'intervallo $\mathbf{g}(b)$:

```
temp = polB(:,dim(2));
g_b2 = [min(temp), max(temp)];
```

Se il dominio ottenuto è completamente positivo, viene calcolata la derivata parziale mediante il calcolo simbolico di MATLAB, fatta la sostituzione di x_2 con b e portato ciò che resta in forma di Bernstein. Da questa sono estratti il minimo e il massimo coefficiente e, quindi, può essere applicata la formula 3.5 di Bernstein-Newton. Infine, viene intersecato l'intervallo ottenuto da questo calcolo con il dominio iniziale e il risultato è sostituito al dominio di partenza.

Proviamo ad applicare la function MATLAB per il vincolo:

$$g(x) = x_2 - 2 - 2x_1^4 + 8x_1^3 - 8x_1^2 \leq 0$$

con $\mathbf{x}_1 = [1.5, 2.25]$ e $\mathbf{x}_2 = [3, 4]$. Si ha:

```
>> pol = [-2 1; 0 0; -8 0; 8 0; -2 0];
>> dom = [1.5 3; 2.25 4];
>> BBC(pol,'y-2*x^4 +8*x^3-8*x^2',dom)

ans =

    1.5653    3.0000
    2.2500    3.1250
```

Notiamo che il dominio è stato effettivamente contratto e, in particolare, il secondo estremo dell'intervallo della seconda variabile coincide con quello presentato nell'esempio precedente, quando si era eseguito il calcolo a mano.

Conclusioni e Sviluppi futuri

Il lavoro svolto ha dimostrato come i polinomi nella base di Bernstein trovino molte applicazioni ai problemi di soddisfacimento di vincoli polinomiali. I risultati ottenuti hanno permesso, attraverso lo studio della suddetta base algebrica, di costruire e implementare alcuni efficaci algoritmi di consistenza per la riduzione dei domini delle variabili in CSP a vincoli polinomiali, passando attraverso soluzioni meno valide ma comunque degne di nota. Tutto questo è stato reso possibile anche grazie alla disamina riservata alle proprietà dei polinomi di Bernstein e alla codifica dell'Algoritmo di Smith per polinomi ad 1, 2 ed n dimensioni. Da sottolineare è anche l'implementazione in linguaggio Java dell'algoritmo sopracitato per la riduzione dei domini, realizzato sfruttando JAMA [4], un package prodotto da MathWorks e NIST contenente la messa a punto delle operazioni dell'algebra lineare di base.

Il lavoro compiuto, quindi, ha chiaramente gettato una solida base su cui ampliare questi orizzonti, producendo nuovi algoritmi che sfruttino i punti di forza che la base di Bernstein può offrire. L'algoritmo di Bernstein Box Consistency (BBC), ad esempio, proposto in questa tesi, può essere semplificato sostituendo la codifica dell'aritmetica degli intervalli proposta, introducendo le primitive di INTLAB [6], un package prodotto dal professor Siegfried M. Rump, dell'Università di Amburgo, per MATLAB e Octave che implementa direttamente le operazioni nell'ottica del calcolo di intervallo.

Inoltre, il modello di CSP qui presentato è, di fatto, un modello statico, formato da vincoli inflessibili. Questa rigidità, a volte, è un difetto che impedisce di rappresentare facilmente alcuni problemi. Sono state quindi proposte diverse varianti alla definizione per adattare il modello ad una più grande varietà di casi. Questo ha portato alla definizione di CSP distribuiti, in cui variabili e i vincoli sono, per l'appunto, distribuiti in una rete di nodi di esecuzione, CSP flessibili, in cui una soluzione non deve necessariamente soddisfare tutti i vincoli, ma un numero minimo di essi, o ancora CSP dinamici, utilizzati quando i vincoli cambiano a causa dell'ambiente. La ricerca di algoritmi efficaci per risolvere questi nuovi problemi è tutt'ora in corso e potrà essere senz'altro oggetto di studio in lavori futuri su tali argomenti.

Appendice A

Implementazione in Java

In questa appendice, per fornire maggiore consistenza alla trattazione, si presentano alcuni algoritmi analizzati nei precedenti capitoli, riproposti questa volta in linguaggio Java.

A.1 JAMA

Per la realizzazione degli algoritmi di riduzione del dominio è stato usato JAMA 1.0.3 (JAva MAtrix package) [4], un package contenente una codifica delle operazioni dell'algebra lineare di base, destinato a divenire la classe standard per le matrici in Java. L'implementazione di riferimento, di pubblico dominio, è stata sviluppata da MathWorks e NIST (National Institute of Standards and Technology) e verrà proposta al Java Grande Forum prima e a Sun poi, per divenire la classe standard per l'abstract data type "matrice" in Java. JAMA è composto da sei classi:

- classe `Matrix`
- classe `CholeskyDecomposition`
- classe `LUDecomposition`
- classe `QRDecomposition`
- classe `SingularValueDecomposition`
- classe `EigenvalueDecomposition`

La classe principale usata per i nostri scopi è la classe `Matrix`, la quale provvede ad implementare le operazioni fondamentali fra matrici, mentre le altre permettono, ad esempio, la fattorizzazione o la ricerca degli autovalori.

Nel Java project realizzato sono presenti tre sole classi: la prima, `Binomial`, offre un'implementazione del coefficiente binomiale matematico, mentre le altre due saranno analizzate nel dettaglio nei prossimi paragrafi.

A.2 La classe BernsteinTools

La classe `BernsteinTools` è il cuore dell'intero programma, poiché racchiude l'implementazione delle funzioni per la riduzione dei domini della variabili nei vincoli, realizzata applicando la formula 3.4. Sono presenti le seguenti funzioni:

- La procedura `toBernsteinMono` non è altro che una trasposizione dell'equivalente versione MATLAB. Prende in input un vettore di coefficienti realizzato utilizzando il tipo `Matrix` di JAMA e restituisce un oggetto `Matrix` contenente i relativi coefficienti di Bernstein della rappresentazione.

- La procedura `toBernstienDuo`, esattamente come la precedente, è una trasposizione della funzione di conversione in forma di Bernstein per polinomi in due dimensioni. Prende in input una `Matrix` contenente i coefficienti del polinomio bi-dimensionale e una `Matrix` con i domini delle singole variabili e restituisce un oggetto `Matrix` con i coefficienti di Bernstein che sono stati calcolati. Utilizza la function ausiliaria `monomio`.

- La funzione `NatarajCheckConstraint` compie la riduzione del dominio per vincoli in una variabile. Prende in input un vettore di coefficienti, un valore che rappresenta il vincolo e una tolleranza e restituisce una `Matrix` $2 \times N$ contenente i valori dei domini accettati con la stessa logica usata nel programma realizzato in MATLAB.

- Le funzioni `NatarajRicalcolaBernstein` e `NatarajBiseziona` sono le procedure ausiliarie della `NatarajCheckConstraint` che implementano la bisezione del dominio e il ricalcolo dei coefficienti di Bernstein, effettuato applicando la formula 3.4.

- Le funzioni `maximum` e `minimum` ricercano il massimo (risp. minimo) elemento all'interno di un oggetto `Matrix`. Sono state utilizzare per facilitare la ricerca del più piccolo e più grande coefficiente di Bernstein negli algoritmi di consistenza.

JAMA non permette di realizzare, al momento, matrici multi-dimensionali e per questo risulta difficile realizzare l'algoritmo per più dimensioni, a meno di non realizzare una propria implementazione dell'ADT "matrice multi-dimensionale".

A.3 La classe MainTest

La classe `MainTest` contiene una serie di esperimenti adeguatamente commentati sulle funzioni implementate nella classe `BernsteinTools`. Per la procedura `toBernsteinMono` utilizziamo il solito polinomio $p(x) = -8 + 63x - 150x^2 + 90x^3$ con $x \in [0, 1]$:

```
Matrix prova = new Matrix(1,4);
prova.set(0, 0, -8);
prova.set(0, 1, 65);
prova.set(0, 2, -150);
prova.set(0, 3, 90);
System.out.println("Dato il polinomio di coefficienti:");
prova.print(2, 0);

Matrix result = BernsteinTools.toBernsteinMono(prova);
System.out.println("Si ottiene la sua forma di Bernstein:");
result.print(2, 3);
```

Per la procedura `NatarajCheckConstraint` sfruttiamo il polinomio $p(x)$ definito precedentemente, vincolandolo ad essere $p(x) < 0$ con $x \in [0, 1]$ e usando una tolleranza di 10^{-1}:

```
Matrix vincolo =
  BernsteinTools.NatarajCheckConstraint(prova, 0, 0.3);
vincolo.print(2, 4);
```

Infine, mostriamo uno dei test effettuati sulla funzione `toBernsteinDuo`. Il polinomio in due dimensioni considerato in questo caso è $q(x) = x_1^3 x_2^2 - 6x_1 x_2$ di dominio $x = [0, 1] \times [0, 1]$. Inizializziamo quindi la matrice dei coefficienti che rappresenta il polinomio:

```
Matrix dueVar = new Matrix(4,3);
dueVar.set(0, 0, 0);
dueVar.set(0, 1, 0);
dueVar.set(0, 2, 0);
dueVar.set(1, 0, 0);
dueVar.set(1, 1, -6);
dueVar.set(1, 2, 0);
dueVar.set(2, 0, 0);
dueVar.set(2, 1, 0);
dueVar.set(2, 2, 0);
```

```
dueVar.set(3, 0, 0);
dueVar.set(3, 1, 0);
dueVar.set(3, 2, 1);
```

Definiamo poi la matrice contenente i domini delle singole variabili:

```
Matrix dominion = new Matrix(2, 2);
dominion.set(0, 0, 0);
dominion.set(0, 1, 0);
dominion.set(1, 0, 1);
dominion.set(1, 1, 1);
```

Infine chiamiamo la funzione, passandogli i paramentri richiesti:

```
Matrix dueVarBernstein =
  BernsteinTools.toBernsteinDuo(dueVar, dominion);
```

Bibliografia

[1] Smith, Andrew. *Fast construction of constant bound functions for sparse polynomials.* Journal of global optimization 43.2-3 (2009):445-458.

[2] Nataraj, P. *Constrained global optimization of multivariate polynomials using Bernstein branch and prune algorithm.* Journal of global optimization 49.2 (2011):185-212.

[3] Farouki, Rida. *The Bernstein polynomial basis: a centennial retrospective.* Computer aided geometric design 29.6 (2012):379-419.

[4] JAMA: A JAva MAtrix Package,
`http://math.nist.gov/javanumerics/jama`

[5] MATLAB Tensor Toolbox,
`http://www.sandia.gov/~tgkolda/TensorToolbox`

[6] INTLAB The Matlab/Octave toolbox for Reliable Computing
`http://www.ti3.tu-harburg.de/rump/intlab`

[7] SWI-Prolog
`http://www.swi-prolog.org`

[8] MiniZinc
`http://www.minizinc.org`