Spline e curve di Bèzier

Riccardo Colavita

Indice

Capitolo 1

Introduzione

Scopo di questa relazione è di trattare l'interpolazione polinomiale a tratti, spiegandone le motivazioni e i principali modi per affrontare i problemi che man mano si presenteranno.
Ci soffermeremo in particolar modo su strumenti quali le spline e le curve di Bezier. Delle prime studieremo la proprietà di minima variazione della curvatura in due casi particolari (le naturali e le complete); delle seconde studieremo un metodo per l'approssimazione grafica di funzioni.

1.1 Perchè l'interpolazione polinomiale a tratti

È noto che data una funzione $f(x)$ di cui sono noti i valori in $n+1$ nodi distinti $x_i \in [a,b], i = 0, ..., n$, esiste ed è unico il polinomio di interpolazione $p_n(x)$ di grado al più n tale che $p_n(x_i) = f(x_i), i = 0, ..., n$.
In generale, non è detto che aumentando n l'approssimazione di $f(x)$ mediante $p_n(x)$ migliori. Infatti, è possibile dimostrare che per ogni successione N_n di sottoinsiemi finiti dell'intervallo $[a,b]$, ognuno di cardinalità $n+1$, esiste una funzione $f(x)$ continua in $[a,b]$ tale che per $p_n(x)$ non converga a $f(x)$ anche se,

$$\lim_{n\to\infty} \max_i |x_{i+1} - x_i| = 0$$

Per questa ragione ci soffermeremo sulle funzioni polinomiali a tratti.

Definizione 1.1. Supponendo che i nodi da interpolare siano ordinati in $[a,b]$, cioè

$$a = x_0 < x_1 < ... < x_n = b$$

si definisce polinomiale a tratti su $[a,b]$ una funzione $t(x)$ che sull'i-esimo intervallo $[x_i, x_{i+1}]$ coincide con il polinomio $t_i(x)$ di grado prefissato k.

La $t(x)$ può essere rappresentata mediante una matrice A di ordine n la cui i-esima riga

$$[a_{i,k}, a_{i,k-1}, \ldots, a_{i,0}]$$

contiene i coefficienti di $t_i(x)$ con la variabile traslata rispetto al punto x_i , cioè

$$t_i(x) = a_{i,k}(x - x_i) + a_{i,k-1}(x - x_i) + \ldots + a_{i,0}$$

Vediamo adesso alcuni esempi di funzioni polinomiali a tratti.

1.2 Esempi di interpolazione polinomiale

1.2.1 Interpolazione a trattti lineare

In questo caso la $t(x)$ è una *funzione lineare* a tratti che coincide con $f(x)$ sui nodi x_i, cioè $t(x) = t_i(x)$ per $x \in [x_i, x_{i+1}]$, dove

$$t_i(x) = \frac{f_{i+1} - f_i}{h_i}(x - x_i) + f_i, \qquad h_i = x_{i+1} - x_i$$

Questa funzione polinomiale viene usata spesso nella pratica ma non fornisce una buona rappresentazione grafica, anche perchè non vi è alcuna condizione sulle derivate dei singoli polinomi, per cui nei nodi x_i il raccordo fra due polinomi lineari presenta un punto angoloso anche se $f \in C^1[a, b]$.

Osservazione 1.1. $t(x)$ è $C^0[a, b]$ ma in generale non è $C^1[a, b]$.

1.2.2 Interpolazione cubica di Hermite

Il metodo dell'interpolazione cubica di Hermite consiste nell'approssimazione di una $f \in C^1[a, b]$, i polinomi $t_i(x)$ hanno tutti al più grado 3, $t(x)$ coincide con $f(x)$ sui nodi x_i, $t(x)$ è derivabile e $t'(x)$ coincide con $f'(x)$ sui nodi x_i quindi

$$t_i(x) = (2(f_i - f_{i+1}) + h_i(f'_i - f'_{i+1}))\frac{(x-x_i)^3}{h_i^3} - (3(f_i - f_{i+1}) + h_i(2f'_i - f'_{i+1}))\frac{(x-x_i)^2}{h_i^2} + \\ + f'_i(x - x_i) + f_i, \qquad \text{dove } h_i = x_{i+1} - x_i$$

Questa funzione polinomiale fornisce una migliore rappresentazione grafica rispetto a quella di una funzione lineare, poichè non da luogo a punti angolosi nei nodi di raccordo, pertanto, anche se nei punti di raccordo i polinomi hanno la stessa pendenza, non è detto che abbiano la stessa concavità.

Osservazione 1.2. $t(x)$ è $C^1[a,b]$ ma in generale non è $C^2[a,b]$.

Osservazione 1.3. Per l'interpolazione cubica di Hermite è evidente, dalla formula che la definisce, che serve conoscere il valore f' nei nodi x_i, se tali valori non sono noti allora il metodo non è applicabile.

1.2.3 Spline

Le *spline* sono una forma particolare di *interpolazione polinomiale a tratti* sufficientemente regolari per la quale non serve conoscere i valori di f_i'.

Definizione 1.2. Data $f(x) : [a,b] \to \mathbb{R}$,una *funzione* spline per f di *ordine* $k \geq 2$ su un insieme di nodi distinti $a = x_0 < x_1 < \ldots < x_n = b$, *è una funzione* $s(x) : [a,b] \to \mathbb{R}$ *che verifica le seguenti proprietà:*

1. $s(x_i) = f(x_i), \quad i = 0, \ldots, n;$
2. $s(x)$ *coincide con il polinomio* $s_i(x)$ *di grado minore di* k *in ciascun intervallo* $[x_i, x_{i+1}]$;
3. $s(x) \in C^{k-2}[a,b]$

Nel caso $k = 2$ la *spline* coincide con la funzione ottenuta con l'interpolazione *lineare* a tratti. Se $k > 2$ allora la *spline* è almeno di classe C^1 , anche se non si conosce la derivata di f.

Osservazione 1.4. Consideriamo lo spazio vettoriale dei polinomi di grado al più 4:

$$\mathbb{R}[x]_{\leq 4} = \left\{ p(x) \in \mathbb{R}[x] \;\middle|\; \deg(p) \leq 4 \right\}$$

Per individuare punti (polinomi) nel suddetto spazio si deve risolvere un sistema lineare di questa forma:

$$Ax = b$$

dove la matrice A è data dalle condizioni *1.*,*2.*,*3.* della definizione 1.2. In particolare la matrice A è *singolare* in quanto ha *rango* $n-2$. Per renderla *invertibile* bisogna aggiungere, opportunamente a seconda del caso, altre due condizioni. In seguito tratteremo un criterio di scelta per le condizioni mancanti.

Capitolo 2

Spline

Ci soffermeremo sulle *funzioni spline* di ordine 4 (spline cubiche) che vedremo avere interessanti proprietà.

Osservazione 2.1. Entrano in gioco n polinomi $(s_0, \ldots, s_{n-1})$ di grado 3 quindi per la *osservazione* 1.4 si devono calcolare $4n$ coefficienti.

2.1 Spline Cubiche

Fra le funzioni spline quelle più usate nella pratica, anche perchè consentono di ottenere ottimi risultati dal punto di vista grafico, sono le *spline cubiche*, in cui ciascun $s_i(x)$ è un polinomio di grado al più 3.

2.1.1 Definizioni e prime proprietà

Definizione 2.1. Data $f(x) : [a, b] \to \mathbb{R}$,una *funzione* spline per f di *ordine* $k = 4$ su un insieme di nodi distinti $a = x_0 < x_1 < \ldots < x_n = b$, *è una funzione* $s(x) : [a, b] \to \mathbb{R}$ *che verifica le seguenti proprietà:*

1. $s_i(x_i) = f(x_i), \quad s_i(x_{i+1}) = f(x_{i+1}), \quad i = 0, \ldots, n-1;$
2. $s_i'(x_i) = s_{i+1}'(x_i), \quad i = 0, \ldots, n-1;$
3. $s_i''(x_i) = s_{i+1}''(x_i), \quad i = 0, \ldots, n-1;$

Poichè i coefficienti dei polinomi sono $4n$ il sistema lineare di riferimento ha $4n-2$ equazioni *linearmente indipendenti*, quindi per determinare i polinomi $s_i(x)$ occorrerà imporre due condizioni aggiuntive, di cui parleremo successivamente.

Si definiscono momenti

$$\mu_i = s_i''(x_i) \quad e \quad \mu_n = s_{n-1}''(x_n) \quad per \quad i = 0, \ldots, n-1$$

che permettono di dare una caratterizzazione delle spline cubiche attraverso questo teorema.

Teorema 2.1. *Una spline cubica coincide in ciascun intervallo* $[x_i, x_{i+1}]$ *con il polinomio di grado al più 3.*

$$s_i(x) = \mu_{i+1}\frac{(x-x_i)^3}{6h_i} - \mu_i\frac{(x-x_{i+1})^3}{6h_i} + \alpha_i(x-x_i) + \beta_i, \quad i = 0, \ldots, n-1,$$

dove

$$\beta_i = f_i - \mu_i\frac{h_i^2}{6}, \quad \alpha_i = \frac{f_{i+1}-f_i}{h_i} - \frac{h_i}{6}(\mu_{i+1}-\mu_i), \quad i = 0, \ldots, n-1,$$

$h_i = x_{i+1} - x_i$ *e i momenti* μ_i $i = 0, \ldots, n$, *risolvono le* $n-1$ *equazioni*

$$h_{i-1}\mu_{i-1} + 2(h_{i-1}+h_i)\mu_i + h_i\mu_{i+1} = 6f_{i-1,i,i+1} \quad i = 1, \ldots, n-1,$$

dove

$$f_{i-1,i,i+1} = \frac{f_{i+1}-f_i}{h_i} - \frac{f_i - f_{i-1}}{h_{i-1}}$$

Come avevamo preannunciato, occorrerà imporre due condizioni aggiuntive affinchè i polinomi $s_i(x)$ definiscano univocamente una spline.
Alcune condizioni possibili sono le seguenti:

$c1$) Si può imporre alla spline un andamento lineare agli estremi, ponendo $s_0''(x_0) = s_{n-1}''(x_n) = 0$ (spline naturale).

$c2$) Se sono noti $f'(a)$ e $f'(b)$,$s_0'(x_0) = f'(a)$, $s_{n-1}'(x_n) = f'(b)$ *(spline completa)* che impone alla spline la tangenza alla $f(x)$ negli estremi. Se i valori di $f'(a)$ e $f'(b)$ non fossero disponibili, si potrebbero sostituire con delle approssimazioni.

$c3$) Se le derivate agli estremi non sono note, può essere conveniente richiedere che $s'''(x)$ sia continua nel secondo e nel penultimo nodo, cioè $s_0'''(x_1) = s_1'''(x_1)$ e $s_{n-2}'''(x_{n-1}) = s_{n-1}'''(x_{n-1})$ e la corrispondente spline è chiamata *not-a-knot spline*. Poichè gli $s_i(x)$ sono polinomi di grado al più 3, le condizioni sono equivalenti a:

$$s_0'''(x) = s_1'''(x), \quad s_{n-2}'''(x) = s_{n-1}'''(x)$$

.

*c*4) Nel caso di una funzione $f(x)$ periodica di periodo $b-a$, si definisce la *spline periodica* tramite la condizione $s_0'(x_0) = s_{n-1}'(x_n)$ e $s_0''(x_0) = s_{n-1}''(x_n)$.

2.1.2 Proprietà di minima variazione della curvatura

Le spline cubiche sono molto usate nella grafica perché fra le funzioni con derivata seconda continua che interpolano la funzione $f(x)$ nei nodi $x_i,\ i = 0,\dots,n$, sono quelle che hanno minima curvatura, cioè che oscillano meno, come risulta dal seguente teorema.

Teorema 2.2. Fra tutte le funzioni $g \in C^2[a,b]$ tali che $g(x_i) = f_i,\ i = 0,\dots,n$, la spline cubica naturale $s(x)$ è quella che minimizza l'integrale

$$\int_a^b [g''(x)]^2 dx$$

Un teorema analogo al teorema 2.2 vale per la spline completa.

Teorema 2.3. Fra tutte le funzioni $g \in C^2[a,b]$ tali che $g(x_i) = f_i,\ i = 0,\dots,n$, e $g'(a) = f'(a)$ e $g'(b) = f'(b)$ la spline cubica completa $s(x)$ è quella che minimizza l'integrale

$$\int_a^b [g''(x)]^2 dx$$

2.1.3 Condizionamento

Studiamo ora il condizionamento del calcolo di $s(x)$ per un x diverso dai nodi. Consideriamo il caso della spline naturale e supponiamo per semplicità che i nodi siano equidistanti e non affetti da errore. Scriviamo il sistema per il calcolo della spline del teorema 2.1 nella forma

$$\mathcal{M}\mu = \mathbf{b}$$

dove μ è il vettore dei μ_i e $\mathbf{b}$ è il vettore di componenti $b_i = 6(f_{i+1} - 2f_i + f_{i-1})/h^2$. Supponendo di perturbare i dati del problema da f_i a $\tilde{f}_i = f_i + \delta_i$, con $\left|\delta_i\right| \le \delta$, la corrispondente variazione $\tilde{\mathbf{b}} - \mathbf{b}$ del termine noto del sistema risulta maggiorata in norma infinito da

$$\|\tilde{\mathbf{b}} - \mathbf{b}\|_\infty = \frac{6}{h^2} \min_{i=1,n-1} \left|\delta_{i+1} - 2\delta_i + \delta_{i-1}\right| \le \frac{24\delta}{h^2}$$

Quindi

$$\frac{\|\tilde{\mu} - \mu\|_\infty}{\|\mu\|_\infty} \;\leq\; K(\mathcal{M})\, \frac{\|\tilde{\mathbf{b}} - \mathbf{b}\|_\infty}{\|\mathbf{b}\|_\infty}$$

dove $\tilde{\mu}$ è la soluzione del sistema il cui termine noto è $\tilde{\mathbf{b}}$ e $K(\mathcal{M})$ è il *numero di condizionamento di* $\mathcal{M}$ *in norma infinito*[1]. La matrice $\mathcal{M}$ è ben condizionata perchè $K(\mathcal{M}) \leq 6$ per ogni n. Quindi il problema del calcolo dei μ_i è ben condizionato. Lo stesso si può dire per il calcolo dei coefficienti α_i e β_i e per il calcolo di $s_i(x)$ del teorema 2.1 per $x \in [x_i, x_{i+1}]$.

[1] Cioè $K(\mathcal{M}) = \|\mathcal{M}\|\, \|\mathcal{M}^{-1}\|$.

Capitolo 3

Curve di Bèzier

Se invece di interpolare o approssimare i dati vogliamo "modellare" una curva sui dati, costruiamo una curva di Bèzier. In questo capitolo tratteremo lo studio della rappresentazione delle *curve di Bèzier* per l'approssimazione di funzioni.
Ricordiamo ora la definizione di curva parametrica prima di andare a vedere come sono fatte le curve di Bèzier e quali proprietà hanno.

Definizione 3.1. Una curva parametrica piana di classe C^k è una $\gamma \in C^k(I, \mathbb{R}^2)$ con $I \subseteq R$ intervallo tale che:

$$\gamma \colon I \to \mathbb{R}^2$$

$$t \mapsto (p(t), q(t))$$

dove $p, q \in \mathbb{R}[t]$. Diremo che γ ha grado n se $n = \max\{ \deg(p), \deg(q) \}$

Esempio 3.1.

$\gamma(t) = (t^2, t^3)$	cuspide	$y^2 = x^3$
$\gamma(t) = (t^2, t^3 - t)$	cubiche nodali	$y^2 = x^3 - x$

3.1 Polinomi di Bèzier costruiti mediante le basi di Bernstein

Definizione 3.2. Chiameremo n-esima Base di Bernstein il sottoinseme di $\mathbb{R}[t]_{\leq n}$ data da $\mathcal{B}_n = \{ B_{0,n}, ..., B_{n,n} \}$ dove

$$B_{k,n}(t) = \binom{n}{k} t^k (1-t)^{n-k}$$

Osservazione 3.1. Per i polinomi $B_{k,n}$ vale la seguente relazione a tre termini:

$$B_{k,n}(t) = (1-t)B_{k,n-1}(t) + tB_{k-1,n-1}(t)$$

Osservazione 3.2. Vale $\deg(B_{i,n}) = n$. Inoltre l'n-esima base di Bernstein è una base di $\mathbb{R}[t]_{\leq n}$.

Definizione 3.3. Sapendo che, data una base $\mathcal{B}_n = \{ B_{0,n}, \ldots, B_{n,n} \}$ per $\mathbb{R}[x]_{\leq n}$, per ogni curva polinomiale γ di grado n $\exists (p_i, q_i) = c_i$ tali che

$$\gamma(t) = \big(\sum_{i=0}^{n} p_i B_{i,n}(t), \sum_{i=0}^{n} q_i B_{i,n}(t) \big) = \sum_{i=0}^{n} c_i B_{i,n}(t)$$

chiameremo i c_i **punti di controllo** di γ rispetto a $B_{i,n}$.
In particolare diremo che i c_i sono i **punti di controllo di Bèzier** se la base utilizzata è quella di Bernstein.

Definizione 3.4. Una curva γ è detta di Bezier, o in forma di Bèzier, se è rappresentata come combinazione lineare dei polinomi nella base di Bernstein.

Osservazione 3.3. I punti di controllo rappresentano i vertici di un poligono che al suo interno contiene la *curva di Bèzier*. In particolare la curva risulta tangente al poligono nel primo e nell'ultimo punto di controllo, che sono gli unici punti che hanno in comune la curva e il poligono.

Definizione 3.5. Dati $n + 1$ punti P_i, denominati punti di controllo. La curva di Bèzier definita dai punti P_i è descritta dalla formula parametrica:

$$\mathbf{b}(u) = \sum_{i=0}^{n} P_i B_{i,n}(u) \qquad u \in [0, 1], P_i \in \mathbb{R}^2$$

Qualsiasi trasformazione affine parametrica della forma:

$$u = a(1 - t) + bt \qquad a \neq b$$

lascia il grado della curva $\mathbf{b}$ inalterato. Di conseguenza anche $\mathbf{b}(u(t))$ ha una rappresentazione di Bèzier di grado n

$$\mathbf{b}(u(t)) = \sum_{i=0}^{n} \mathbf{b}_i B_{i,n}(t)$$

i coefficienti $\mathbf{b}_i$ sono elementi di $\mathbb{R}^d$ e vengono chiamati punti di Bèzier che non sono altro che i vertici del poligono di Bèzier rispetto a $\mathbf{b}_i$ sull'intervallo $[a, b]$.

3.2 Polinomi di Bèzier costruiti mediante l'algoritmo di de Casteljau

L'algoritmo di de Casteljau è un algoritmo ricorsivo che permette di valutare in un punto il polinomio di Bèzier con un procedimento ricorsivo che utilizza la parametrizzazione segmentaria della retta, ovvero la parametrizzazione $R(t) = a + t(b - a)$ della retta r passante per due punti a e b che fornisce a quando $t = 0$ e b quando $t = 1$. Di seguito per non appesantire la notazione avremo sempre $a = 0$, $b = 1$ in modo da non dover riparametrizzare .
Dati gli $n + 1$ punti di controllo della curva $\mathbf{b}(t)$ $P_0, \ldots, P_n \in \mathbb{R}^2$ e $t \in [0, 1]$, l'algoritmo è così definito:
Usando ripetutamente la relazione di ricorrenza a tre termini, e raccogliendo opportunamente i termini, otteniamo:

$$\mathbf{b}(t) = \sum_{i=0}^{n} b_i^0 B_{i,n}(t) = \sum_{i=0}^{n-1} b_i^1 B_{i,n-1}(t) = \cdots = \sum_{i=0}^{0} b_i^n B_{i,0}(t) = b_0^n$$

In particolare risulta:

$$b_i^0 = P_i$$

$$b_i^{k+1} = (1 - t)b_i^k(t) + tb_{i+1}^k(t)$$

con $k = 1, \ldots, n$ e $i = 0, \ldots, n - k$. Allora il valore della curva di Bèzier nel punto $t_0 \in [0, 1]$ è $\mathbf{b}(t_0) = b_0^n(t_0) = b_0^n$.
Per il calcolo manuale del valore della curva di Bèzier $\mathbf{b}(t)$ in un punto $t_0 \in [0, 1]$, è conveniente costruire il cosidetto **"Schema di De Casteljau"**:

$$\begin{array}{ccccccc}
P_0 & & & & & & \\
P_1 & b_0^1 & & & & & \\
P_2 & b_1^1 & b_0^2 & & & & \\
\vdots & \vdots & \vdots & \ddots & & & \\
P_{n-1} & b_{n-2}^1 & b_{n-3}^2 & \cdots & b_0^{n-1} & & \\
P_n & b_{n-1}^1 & b_{n-2}^2 & \ldots & b_1^{n-1} & b_0^n = \mathbf{b}(t_0)
\end{array}$$

Sebbene l'algoritmo sia più lento per la maggior parte delle architetture se comparato all'approccio diretto, è numericamente più stabile.

3.3 Condizionamento delle basi

Le curve di Bèzier sono maggiormente utilizzate in grafica per modellare curve smussate. Dato che la curva è contenuta completamente nell'inviluppo

convesso dei suoi punti di controllo, i punti possono essere visualizzati graficamente ed usati per manipolare la curva intuitivamente.
Inoltre i punti di controllo sono importanti per quel che riguarda il condizionamento del problema nel calcolo delle loro radici. Vediamo più in dettaglio a cosa ci riferiamo:
Come visto nell'osservazione 3.2 ogni famiglia della forma:

$$\mathcal{B} = \{\ p_0(t), \ldots, p_n(t) \in \mathbb{R}[t] \ \big|\ \deg(p_i) = i\ \}$$

è una base di $\mathbb{R}[t]_{\leq n}$. Pertanto ogni polinomio $p(t)$ di grado n si può rappresentare come

$$p(t) = \sum_{i=0}^{n} c_i p_i(t)$$

con $c_i \in \mathbb{R}\ \forall i$.
Nelle sezioni che seguiranno ci occuperemo della stabilità di tali rappresentazioni e di come vengano influenzati il calcolo dei valori assunti da p e il calcolo delle sue radici, sotto perturbazioni arbitrarie dei coefficienti $c_0, \ldots, c_n$ corrispondenti a una base $\mathcal{B}$.

3.3.1 Numero di condizionamento del calcolo dei valori di un polinomio

Definizione 3.6. Definiamo il numero di condizionamento $C_{\mathcal{B}}(p(t))$ per il calcolo del valore di $p(t)$ nella rappresentazione data dalla base $\mathcal{B}$ come:

$$C_{\mathcal{B}}(p(t)) = \sum_{i=0}^{n} |c_i p_i(t)|$$

Osservazione 3.4. Data una perturbazione di modulo ϵ, con ϵ costante positiva, un forte limite sul $\delta p(t)$, ottenuto sotto l'effetto di tale perturbazione, può essere espresso in termini di $C_{\mathcal{B}}(p(t))$ dalla formula:

$$|\delta p(t)| \leq C_{\mathcal{B}}(p(t))\epsilon$$

Osservazione 3.5. Il limite sopra citato vale con ϵ arbitrario e non necessariamente infinitesimale. Inoltre il valore di $C_{\mathcal{B}}(p(t))$ dipende sia dagli elementi della base $\mathcal{B}$ sia dal polinomio p considerato.

3.3.2 Numero di condizionamento del calcolo delle radici di un polinomio

Definizione 3.7. Sia τ una *radice semplice*[1] del polinomio p, definiamo il numero di condizionamento della radice τ, $C_{\mathcal{B}}(\tau)$ nella rappresentazione data dalla base $\mathcal{B}$ come:

$$C_{\mathcal{B}}(\tau) = \frac{1}{|p'(\tau)|} \sum_{i=0}^{n} |c_i p_i(\tau)|$$

Osservazione 3.6. Data una perturbazione di modulo ϵ, con ϵ costante positiva, un forte limite sul $\delta\tau$, ottenuto sotto l'effetto di tale perturbazione, può essere espresso in termini di $C_{\mathcal{B}}(\tau)$ dalla formula:

$$|\delta\tau| \leq C_{\mathcal{B}}(\tau)\epsilon$$

Osservazione 3.7. La maggiorazione sopra citata potrebbe non aver senso per tutti gli ϵ positivi, per tanto la disuguaglianza sopra è da intendersi per ϵ infinitesimale.

3.3.3 Confronto tra i numeri di condizionamento associati a basi diverse

Per ottenere un risultato accurato, quando vengono utilizzati numeri in virgola mobile nel calcolo di un polinomio, è preferibile utilizzare una base per la quale i numeri di condizionamento definiti sopra siano più piccoli possibile.
Confrontando i numeri di condizionamento di due basi $\mathcal{B}$ e $\mathcal{C}$ scopriremo, in generale, che ad alcuni polinomi p e valori di $t \in [a, b]$ è associato un numero di condizionamento basso nella base $\mathcal{B}$, mentre altri son meglio condizionati nella rappresentazione in base $\mathcal{C}$. In altre parole non c'è nessuna disuguaglianza tra $C_{\mathcal{B}}(p(t))$ e $C_{\mathcal{C}}(p(t))$ valida per ogni polinomio p e per tutti i $t \in [a, b]$.
Questo problema può essere aggirato imponendo restrizioni adeguate sulle basi da noi considerate. In particolare ci concentreremo sulle basi non-negative:

Definizione 3.8. Data una base $\mathcal{B} = \{ p_0, \ldots, p_n \}$ diremo che è una base non-negativa su l'intervallo $[a, b]$ se per $i = 0, \ldots, n$ vale:

$$p_i(t) \geq 0 \qquad \forall t \in [a, b]$$

[1] Ossia $p(\tau) = 0 \neq p'(\tau)$.

Osservazione 3.8. Se la base è una *partizione dell'unità*,[2] a partire dalla rappresentazione di un polinomio p di grado $\leq n$ come combinazione lineare degli elementi di $\mathcal{B}$ con coefficienti $c_0, \ldots, c_n$, si ottiene la seguente proprietà:

$$\min_k(c_k) \leq p(t) \leq \max_k(c_k) \quad \forall t \in [a, b]$$

In termini di numero di condizionamento le basi non-negative sono di particolare interesse per il seguente risultato.

Proposizione 3.1. Siano $\mathcal{B} = \{ p_0(t), \ldots, p_n(t) \}$ $\mathcal{C} = \{ q_0(t), \ldots, q_n(t) \}$ due basi non-negative su $[a, b]$ per $\mathbb{R}[t]_{\leq n}$ tali che la seconda possa essere espressa come combinazione lineare non-negativa della prima ossia:

$$q_j(t) = \sum_{i=0}^{n} M_{j,i} p_i(t), \quad j = 0, \ldots, n$$

dove $M_{j,i} \geq 0$ per ogni $0 \leq j, i \leq n$.[3] Allora i numeri di condizionamento del calcolo del valore $p(t)$ di ogni polinomio p di grado n in ogni punto $t \in [a, b]$ in queste due basi soddisfa la disuguaglianza:

$$C_{\mathcal{B}}(p(t)) \leq C_{\mathcal{C}}(p(t))$$

Osservazione 3.9. La disuguaglianza del teorema precedente vale chiaramente anche tra i numeri di condizionamento del calcolo di una radice τ, in qunato essi differiscono rispettivamente da $C_{\mathcal{B}}(p(t))$ e da $C_{\mathcal{C}}(p(t))$ della stessa quantità $\frac{1}{|p'(\tau)|}$.

3.3.4 Ordine parziale sulle basi non-negative

In questa sezione ci concentreremo sulla valutazione dei polinomi su un generico intervallo $I = [a, b]$ con $a < b \in \mathbb{R}$.

Definizione 3.9. Sia:

$$\mathcal{P}_n = \left\{ \mathcal{B} \;\middle|\; \mathcal{B} \text{ base non-negativa di } \mathbb{R}[t]_{\leq n} \right\}$$

l'insieme delle basi non-negative dello spazio dei polinomi a coefficienti reali in una variabile di grado $\leq n$. Definiamo su $\mathcal{P}_n$ una relazione di ordine parziale:
Siano $\mathcal{B}, \mathcal{C} \in \mathcal{P}_n$ diremo che $\mathcal{B} \prec \mathcal{C}$ se, detta $M = (M_{ji})_{j,i=0}^{n}$ la matrice di cambiamento di base da $\mathcal{B}$ a $\mathcal{C}$, essa ha tutte le entrate positive.

[2] Ossia: $(p_0(t) + \ldots + p_n(t) \equiv 1)$. Per esempio i polinomi di Bernstein lo sono.

[3] In altre parole, detta $M = (M_{ji})_{j,i=0}^{n}$ la matrice di cambiamento di base da $\mathcal{B}$ a $\mathcal{C}$, essa ha tutte le entrate positive.

Osservazione 3.10. La relazione $\prec$ è una relazione di ordine parziale.

Teorema 3.1. Date due basi $\mathcal{B}, \mathcal{C} \in \mathcal{P}_n$ vale:

$$\mathcal{B} \prec \mathcal{C} \quad \Leftrightarrow \quad C_{\mathcal{B}}(p(t)) \leq C_{\mathcal{C}}(p(t))$$

dove la disuguaglianza a destra vale $\forall p \in \mathbb{R}[t]_{\leq n}$ e $\forall t \in I$.

Finalmente arriviamo al risultato che ci mostra l'utilità delle basi di Bernstein.

Teorema 3.2. Le basi di Bernstein sono elementi di $\mathcal{P}_n$ minimali rispetto alla relazione d'ordine parziale $\prec$.

Osservazione 3.11. Combinando gli ultimi due teoremi risulta che i numeri di condizionamento associati alla base di Bernstein sono i minimi possibili. In particolare rappresentare un polinomio p in forma di Bèzier minimizza l'errore della macchina nel calcolo dei valori di p in ogni $t \in [0, 1]$ e nel calcolo delle sue radici.

3.4 Curve di Bèzier per l'interpolazione

Come abbiamo descritto in precedenza le curve di Bèzier possono essere modellate, mediante i punti di controllo, con lo scopo, per esempio, di riuscire ad approssimare il grafico di una funzione.
In questo sezione rappresenteremo l'unico polinomio di grado n che interpola la funzione in $n + 1$ nodi fissati in forma di Bèzier, ottenendone così una rappresentazione con numero di condizionamento più basso possibile.

3.4.1 I punti di controllo relativi all'interpolazione

Consideriamo una funzione $f(x)$ di cui sono noti i valori in $n+1$ nodi distinti $x_i \in [a, b],\ i = 0, \ldots, n$ e supponiamo di volerla interpolare su tali nodi con l'unico polinomio di grado n. Quindi le incognite del problema sono i punti di controllo della curva di Bèzier di grado n $\mathbf{b}$ tale che $\mathbf{b}(x_i(t_i)) = f(x_i),\ i = 0, \ldots, n$.
Supponiamo che i nodi da interpolare siano ordinati in $[a, b]$ cioè:

$$a = x_0 < x_1 < \ldots < x_n = b$$

e consideriamo il sistema

$$\left(\begin{array}{c|c} \mathrm{M} & 0 \\ \hline 0 & \mathrm{M} \end{array}\right) \begin{pmatrix} p_0 \\ \vdots \\ p_n \\ q_0 \\ \vdots \\ q_n \end{pmatrix} = \begin{pmatrix} x_0 \\ \vdots \\ x_n \\ f(x_0) \\ \vdots \\ f(x_n) \end{pmatrix}$$

dove $c_i = (p_i, q_i)$ sono i punti di controllo della curva di Bézier e la matrice M è data da:

$$M_{i,j} = \begin{cases} 1 & \text{se } i = j = 0 \vee \text{se } i = j = n, \\ 0 & \text{se } i = 0 \wedge j \neq 0 \vee \text{se } i = n \wedge j \neq n \\ \binom{n}{j} (\frac{i}{n})^j (1 - \frac{i}{n})^{n-j} & \text{altrimenti} \end{cases}$$

Una volta trovati i punti di controllo risolvendo il sistema lineare, siamo in grado di disegnare la curva di Bèzier che interpola nei punti $(x_i, f(x_i)),\ i = 0, \ldots, n$.

Osservazione 3.12. Osserviamo che, nel caso in cui fossimo partiti dalla n-sima base canonica: $\mathcal{E} = \{\ 1, t, \ldots, t^n\ \}$ la matrice M sarebbe stata una matrice di Vandermonde la quale sappiamo avere un alto numero di condizionamento.

Capitolo 4

Descrizione del problema a livello algoritmico

4.1 Calcolo numerico in Python

In questo capitolo tratteremo le principali scelte implementative utilizzate per il calcolo delle *spline* e delle *curve di bèzier* utilizzando come linguaggio di programmazione Python.
Python è un linguaggio di scripting ad oggetti. Come proprietà fondamentale, consente la *"tipizzazione dinamica"*(dynamic typing), cioè:

1. riconosce automaticamente oggetti quali numeri, stringhe, liste, ..., e quindi non richiede di dichiararne il tipo e la dimensione prima dell'utilizzo;

2. effettua in modo automatico l'allocazione e la gestione della memoria.

Queste caratteristiche contribuiscono in modo sostanziale a velocizzare la prototipazione di algoritmi di calcolo numerico. L'utilizzo di Python per l'implementazione di algoritmi numerici è possibile principalmente grazie all'utilizzo di alcune librerie, tra cui segnaliamo:

1. **NumPy**: È uno dei package fondamentali per il calcolo numerico, permette la creazione di array N-dimensionali utilizzati come contenitori di dati generici per la risoluzione di sistemi lineari.

2. **MathPlotlib**: È una libreria di plotting 2D che produce figure di qualità e un ambiente interattivo per una valutazione accurata di ciò che viene plottato.

3. **Scipy**: Fornisce molte routine numeriche amichevoli ed efficienti, come le routine per l'integrazione numerica e l'ottimizzazione.

4. **bèzier**: È un modulo per la creazione di curve di Bèzier a partire da punti di controllo utilizzando l'algoritmo di DeCasteljau.

Questi ultimi due moduli verranno presi come pietra di paragone per i risultati prodotti dagli algoritmi che verranno descritti in questa sessione.

4.2 Scelta dei principali algoritmi

In questa sezione ci soffermeremo sull'analisi e l'implementazione degli algoritmi per la generazione di *spline cubiche* e *curve di bèzier*.

4.2.1 La classe Spline

Utilizzando i teoremi e le definizioni viste nel capitolo 2, andremo ad analizzare i principali algoritmi utilizzati per il calcolo delle spline cubiche, che sono tra le funzioni spline quelle più usate nella pratica.
Lo stato della classe spline viene così definito:

```
import numpy as np
from scipy import linalg
import math

class Spline():
  def __init__(self,interval , n_Knot , function ,target=('
    natural',None)):

    self.dist = n_Knot # distanza nodi
    self.f = function # funzione da approssimare
    self.partition , self.index_n_Knot = self.partition_interval
       (interval,self.dist)
    self.y_newData = [ self.f(x) for x in self.partition] #y_knot
    self.h = self.h_i(self.partition) # distanze
    self.n_knot = len(self.partition)-1 # numero di nodi
    if target[0] == 'natural' : # spline naturale
      self.m_i = self.momenti_naturalSpline(self.n_knot,self.
       y_newData,self.h)
    elif target[0] == 'complete': #spline completa
      if target[1] == None:
        raise ReferenceError
      else:
        f1_a,f1_b = target[1][0][1] , target[1][1][1] #valore
    derivata agli estremi
```

```
21          self.m_i = self.momenti_splineCompleta(self.n_knot+1,self
    .           y_newData,self.h,f1_a,f1_b)
22      elif target[0] == 'not-and-knot': # spline not-and-knot
23        self.m_i = self.momenti_splineNot_and_knot(self.n_knot,
            self.y_newData,self.h)
24      self.beta , self.alfa = self.beta_i_alfa_i(self.m_i,self.
          y_newData,self.n_knot,self.h)
25      self.result_x , self.result_y = self.interpola_nodi(interval)
        #calcolo s(x) perogni x in [a,b]
```

Come noto dalla definizione 2.1 la prima cosa è quella di andare a definire una partizione dell'intervallo $[a, b]$ in $n + 1$ nodi distinti che verranno utilizzati come nodi di interpolazione per il calcolo delle spline. Partiamo con l'utilizzo della seguente funzione che consente di creare una partizione di nodi equidistanti a partire da un intervallo:

```
1   def partition_interval (self,interval,n):
2   '''Partiziona l'intevallo in n+1 nodi equidistanti
3       Args: interval = intervallo [a,b] da partizionare
4             n = distanza tra i nodi
5       Return: tavola dei nodi di interpolazione e indici dei nodi
      '''
6       i = 0
7       part = []
8       idx = []
9       while i < len(interval):
10        part.append(float(interval[i]))
11        idx.append(i)
12        i += n
13      if part.count(interval[len(interval)-1]) == 0: # aggiungo l'
      estremo dell'intervallo se non
14        part.append(float(interval[len(interval)-1])) # e' stato
      incluso nella tavola
15        idx.append(len(interval)-1)
16      return part , idx
```

La funzione restituisce la partizione che è l'insieme dei nodi da interpolare e un array di indici che servirà per la valutazione delle spline nei nodi. Inoltre alla fine della chiamata si controlla se l'estremo destro dell'intervallo deve essere incluso nella partizione.
Calcoliamo le distanze tra i nodi di interpolazione attraverso la seguente funzione:

```
1 def h_i (x):
2 '''Calcola il vettore che contiene le distanze tra i nodi x_i ,
    x_i+1
3   Args: x = vettore delle ascisse dei nodi di interpolazione
4   Return: vettore delle distanze'''
```

```
  h_i = []
  for i in range(len(x)-1):
    h_i.append(x[i+1]-x[i])
  return h_i
```

Quindi data la tavola dei nodi da interpolare e il tipo di spline andiamo a costruire i **momenti** che ci aiutano a risolvere l'equazione della spline descritta nel teorema 2.1.
Per la spline *naturale*, dalla $c1$) si ha:

$$\mu_0 = 0 \quad \mu_n = 0$$

Dunque bisogna risolvere il sistema:

$$\mathcal{M} \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{n-1} \end{bmatrix} = 6 \begin{bmatrix} f_{0,1,2} \\ f_{1,2,3} \\ \vdots \\ f_{n-2,n-1,n} \end{bmatrix}$$

dove

$$\mathcal{M} = \begin{pmatrix} 2(h_0+h_1) & h_1 & & \\ h_1 & 2(h_1+h_2) & h_2 & \\ & \ddots & \ddots & \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{pmatrix}$$

Il sistema viene risolto attraverso la seguente funzione

```
def momenti_naturalSpline(n,ydata,x,h):
  '''Calcola i momenti relativi alle spline naturali
    Args: n = numero di nodi
          ydata = vettore delle ordinate dei nodi di
            interpolazione
          h = vettore delle distanze tra nodi
  Return: vettore dei momenti per le spline naturali'''
  M = [[0.0 for x in range(n-1)] for y in range(n-1)]
  for i in range(n-1):
    for j in range(n-1):
      if i == j :
        M[i][j] = 2*(h[i]+h[i+1])
      elif i+1 == j :
        M[i][j] = h[j]
      elif i == j+1 :
        M[i][j] = h[i]
  vect_f = []
  for i in range(1,n):
    vect_f.append(6*(((ydata[i+1]-ydata[i])/h[i])-((ydata[i]-
    ydata[i-1])/h[i-1])))
```

```
19    solution = linalg.solve(M, vect_f)
20    momenti = []
21    for i in solution:
22      momenti.append(i)
23    momenti.insert(0,0.0) #aggiungo mu_0 , mu_n
24    momenti.insert(n,0.0)
25
26    return momenti
```

Dove le righe da 1 a 13 corrispondono all'inizializzazione di M e del vettore delle $f_{i-1,i,i+1}$, alla riga 14 viene risolto il sistema attraverso la chiamata di routin linalg.solve(A,b) della libreria **NumPy**, infine vengono aggiunti alla soluzione $\mu_0 = 0, \mu_n = 0$.
Per la spline *completa*, dalla $c2$ si ha:

$$s'_0 = -\mu_0 \frac{h_0}{3} - \mu_1 \frac{h_0}{6} + \frac{f_1 - f_0}{h_0} = f'_0$$

$$s'_{n-1} = \mu_{n-1} \frac{h_{n-1}}{6} + \mu_n \frac{h_{n-1}}{3} + \frac{f_n - f_{n-1}}{h_{n-1}} = f'_n$$

dove $f'_0 = f'(a)$ e $f'_n = f'(b)$ sono assegnati. Si ottiene

$$h_0(2\mu_0 + \mu_1) = 6f_{0,0,1}, \quad h_{n-1}(\mu n - 1 + 2\mu_n) = 6f_{n-1,n,n}$$

dove

$$f_{0,0,1} = \frac{f_1 - f_0}{h_0} - f'_0, \quad f_{n-1,n,n} = f'_n - \frac{f_n - f_{n-1}}{h_{n-1}}$$

Dunque bisogna risolvere il sistema:

$$\mathcal{M} \begin{bmatrix} \mu_0 \\ \mu_1 \\ \vdots \\ \mu_n \end{bmatrix} = 6 \begin{bmatrix} f_{0,0,1} \\ f_{0,1,2} \\ \vdots \\ f_{n-2,n-1,n} \\ f_{n-1,n,n} \end{bmatrix}$$

dove

$$\mathcal{M} = \begin{pmatrix} 2h_0 & h_0 & & & \\ h_0 & 2(h_0 + h_1) & h_1 & & \\ & \ddots & \ddots & \ddots & \\ & & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ & & & h_{n-1} & 2h_{n-1} \end{pmatrix}$$

Il sistema viene risolto attraverso la seguente funzione

```
1  def momenti_splineCompleta(self,n,ydata,h,f1a,f1b):
2  '''Calcola i momenti relativi alle spline complete
3    Args: n = numero di nodi
4          ydata = vettore delle ordinate dei nodi di interpolazione
5          h = vettore delle distanze tra nodi
6          f1a,f1b = valori della derivata agli estremi
7    Return: vettore dei momenti per le spline complete'''
8    M = [[0.0 for x in range(n)] for y in range(n)]
9    for i in range(n):
10     for j in range(n):
11       if i == j :
12         if i != 0 and i!= n-1:
13           M[i][j] = 2*(float(h[i]+h[i-1]))
14         elif i == 0:
15           M[i][j] = 2*(float(h[i]))
16         else:
17           M[i][j] = 2*(float(h[i-1]))
18       elif i+1 == j :
19         M[i][j] = float(h[j-1])
20       elif i == j+1 :
21         M[i][j] = float(h[i-1])
22   vect_f = []
23   vect_f.append(6*(((ydata[1]-ydata[0])/h[0])-f1a))
24   for i in range (1,n-1):
25     vect_f.append(6*(((ydata[i+1]-ydata[i])/h[i])-((ydata[i]-
       ydata[i-1])/h[i-1])))
26   vect_f.append(6*(f1b-((ydata[n-1]-ydata[n-2])/h[n-2])))
27   solution = linalg.solve(M, vect_f)
28   return solution
```

Dove le righe da 1 a 13 corrispondono all'inizializzazione di M le righe da 14 a 18 corrispondono all'inizializzazione del vettore delle $f_{i-1,i,i+1}$, alla riga 19 viene risolto il sistema.
Per la spline *not-and-knot*, dalla $c3$) si ha: Si ha

$$\mu_0 = \mu_1 \left(1 + \frac{h_o}{h_1}\right) - \mu_2 \frac{h_0}{h_1}$$

e

$$\mu_n = \mu_{n-1} \left(1 + \frac{h_{n-2}}{h_{n-1}}\right) - \mu_{n-2} \frac{h_{n-2}}{h_{n-1}}$$

Dunque bisogna risolve il sistema:

$$\mathcal{M} \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{n-1} \end{bmatrix} = 6 \begin{bmatrix} f_{0,1,2} \\ f_{1,2,3} \\ \vdots \\ f_{n-2,n-1,n} \end{bmatrix}$$

dove

$$\mathcal{M} = \begin{pmatrix} 3h_0 + 2h_1 + \frac{h_0^2}{h_1} & h_1 - \frac{h_0^2}{h_1} & & \\ h_1 & 2(h_1 + h_2) & h_2 & \\ & \ddots & \ddots & \\ & & h_{n-2} - \frac{h_{n-1}^2}{h_{n-2}} & 3h_{n-1} + 2h_{n-2} + \frac{h_{n-1}^2}{h_{n-2}} \end{pmatrix}$$

Il sistema viene risolto attraverso la seguente funzione:

```
def momenti_splineNot_and_knot(self,n,ydata,h):
'''Calcola i momenti relativi alle spline not-and-knot
    Args: n = numero di nodi
        ydata = vettore delle ordinate dei nodi di interpolazione
        h = vettore delle distanze tra nodi
    Return: vettore dei momenti per le spline not-and-knot'''
  M = [[0.0 for x in range(n-1)] for y in range(n-1)]
  for i in range(n-1):
    for j in range(n-1):
      if i == j :
        if i != 0 and i!= n-1:
          M[i][j] = 2*(float(h[i]+h[i+1]))
        elif i == 0:
          M[i][j] = 3*(float(h[i])) + 2*(float(h[i+1])) + (float(
            pow(h[i],2)/h[i+1]))
        else:
          M[i][j] = 3*(float(h[i])) + 2*(float(h[i-1])) + (float(
            pow(h[i],2)/h[i-1]))
      elif i+1 == j :
        if i == 0:
          M[i][j] = float(h[j]) - float(pow(h[i],2)/h[j])
        else:
          M[i][j] = float(h[j])
      elif i == j+1 :
        if i == n-1:
          M[i][j] = float(h[j]) - float(pow(h[i],2)/h[j])
        else:
          M[i][j] = float(h[i])
  vect_f = []
  for i in range(1,n):
    vect_f.append(6*(((ydata[i+1]-ydata[i])/h[i])-((ydata[i]-
     ydata[i-1])/h[i-1])))
  solution = linalg.solve(M, vect_f)
  momenti = []
  mu_0 = solution[0]*(1+(float(h[0]/h[1]))) - solution[1]*(float(
    h[0]/h[1]))
  momenti.append(mu_0)
  lenght = len(solution)-1
  for i in solution:
```

```
    momenti.append(i)
  mu_n = solution[lenght]*(1-(float(h[lenght-1]/h[lenght])))-
    solution[lenght-1]*(float(h[lenght-1]/h[lenght]))
  momenti.append(mu_n)
  return momenti
```

Non resta che calcolare gli α_i , i β_i attraverso le funzioni qui descritte.

```
def beta_i_alfa_i(mu_i,y,n,h):
'''Calcola il vettore degli alpha_i e beta_i
  Args: mu_i = momenti che definiscono la spline
      y = vettore delle ordinate dei nodi di intrpolazione
      n = numero di nodi di interpolazione
      h = distanza tra x_i e x_i+1
  Return: vettore degli alpha_i beta_i'''
  beta_i , alfa_i = [] , []
  for i in range(n):
    beta_i.append(y[i]-(mu_i[i]*(h[i]**2)/6))
    alfa_i.append(((y[i+1]-y[i])/h[i]) -((h[i]/6)*(mu_i[i+1]-mu_i
    [i])))
  return beta_i ,alfa_i
```

Si hanno ora a disposizione tutti i parametri per il calcolo della spline.

```
def index_(xdata , x):
'''Trova l'indice dell' i-esima spline s_i(x)
  Args: xdata = tupla dei nodi di interpolazione in cui il
          secondo elemento serve per la valutazione nei nodi di
          raccordo e per stabilire a quale intervallo
          appartiene il punto x
        x = punto in [a,b]
  Return: indice della spline che servira' per la valutazione in
    quell'intervallo'''
  trovato = -1
  i = 0
  while trovato == -1 and i < len(xdata)-1:
    if x == xdata[i][0] and i == 0 :
      trovato = i
    elif x == xdata[i][0] and i != 0 :
      if xdata[i][1] and i != len(xdata)-1:
        trovato = i+1
      else:
        xdata[i] = [xdata[i][0],True]
        trovato = i
    elif xdata[i][0] < x < xdata[i+1][0]:
      trovato = i
    i+=1
  if i == len(xdata)-1:
    trovato = i

```

```
23    return trovato
24
25  def interp_tratti(mu_i,alfa,beta,xdata,x,n,lenght,h) :
26  '''Trova l'indice dell'i-esima spline e ne restituisce il valore
      calcolato nel punto x
27      Args: mu_i = momenti
28            alfa , beta = coefficienti dell'interpolazione
29            xdata = tuple di nodi di interpolazione
30            x = punto in [a,b]
31            n = distanza tra nodi
32            lenght = lunghezza intervallo
33            h = distanza tra x_i e x_i+1
34      Returns: y = s(x)'''
35    idx  = self.index_(xdata , x )
36    if idx == -1:#primo nodo
37      return eval_spline(mu_i,alfa,beta,x,xdata,n,0,h)
38    else:
39      return eval_spline(mu_i,alfa,beta,x,xdata,n,idx,h)
40
41  def eval_spline(mu_i , alpha , beta , x , n_x , n , idx , h):
42  '''Calcola la spline s_i(x)'''
43    if idx == len(n_x) -1: #ultimo nodo
44      g = mu_i[idx]*((x-n_x[idx-1][0])**3)/(6*h[idx-1])
45      g1 = mu_i[idx-1] *((x-n_x[idx][0])**3)/(6*h[idx-1])
46      a = alpha[idx-1]*(x-n_x[idx-1][0])
47      b = beta[idx-1]
48    else:
49      g = mu_i[idx+1]*((x-n_x[idx][0])**3)/(6*h[idx])
50      g1 = mu_i[idx] *((x-n_x[idx+1][0])**3)/(6*h[idx])
51      a = alpha[idx]*(x-n_x[idx][0])
52      b = beta[idx]
53
54    return g - g1 + a + b
55
56  def interpola_nodi(self,interval):
57  '''Calcolo di s(x) perogni x in [a,b]'''
58    y = [] #ordinate interpolazione
59    data = [] # ascisse interpolazione
60    for i in range(len(interval)):
61      if self.partition.count(float(interval[i])) > 0:#aggiungo
62        if i != 0 and i != len(interval)-1:#2 volte il nodo
63          data.append(float(interval[i]))#i-esimo per farlo
64          data.append(float(interval[i]))#calcolare a due spline
65        else:
66          data.append(float(interval[i]))
67      else:
68        data.append(float(interval[i]))
69    k_not = []#ascisse
70    for i in self.partition:
```

```
71      k_not.append((i,False))
72    for i  in data:
73      y.append(interp_tratti(self.m_i,self.alfa,self.beta,k_not,i,
        self.n,len(data),self.h))
74    for i in self.index_n_Knot:
75      if i != len(interval) -1 and i != 0:
76      data.pop(i+1)#estraggo dalle ascisse e dalle ordinate
77      y.pop(i+1)#i nodi che snon stati ripetuti due volte
78    return data , y
```

Dove la funzione *index* viene utilizzata per trovare l'indice dell'i-esima spline s_i e le funzioni *interptratti*, *evalspline* e *interpolanodi* servono per il calcolo delle spline definita nel teorema 2.1.

4.2.2 Test spline naturale/completa

Consideriamo la funzione $f(x) = e^{-x}\cos(2\pi x), x \in [a,b]$ con $a = 0$ e $b = 5$ partizionato in 500 punti. Scegliamo nodi equidistanti con $n = 3$.
Il seguente codice Python crea una spline completa, mettendola a paragone con la spline completa della libreria **Scipy**

```
1  import numpy as np
2  from scipy import interpolate
3  import matplotlib.pyplot as plt
4  import Spline_Bezier as s_b
5
6  def f(t):
7  #funzione
8    return np.exp(-t) * np.cos(2*np.pi*t)
9
10 def f_prime(t): #derivata
11   return -2*np.pi*(np.exp(-t))*np.sin(2*np.pi*t) -((np.exp(-t))*(
       np.cos(2*np.pi*t)))
12
13 x = np.arange(0.0, 5.0, 0.01) # [a,b] = [0,0.01,0.02,...,5]
14 n = 3 # numero nodi di interpolazione
15 dist = (int(len(x)/n)) # distanza tra i nodi
16 y = []
17 for i in x:
18   y.append(f(i)) # f(x) per ogni x in [a,b]
19 f1_a , f1_b = f_prime(x[0]) , f_prime(x[len(x)-1]) # valori della
       derivata prima agli estremi
20 spline_completa = s_b.Spline(x,dist,f,target=['complete',
       [(1,f1_a),(1,f1_b)]]) # creazione della spline completa
21 x_knots = [k for k in spline_completa.partition] # ascisse dei
       nodi di interpolazione
```

```
y_knots = [f(k) for k in spline_completa.partition] # ordinate
    dei nodi di interpolazione
tck = interpolate.CubicSpline(x_knots, y_knots,bc_type=((1,f1_a)
    ,(1,f1_b))) # rappresentazione Scipyspline completa s(x)
y_scipy = tck(x) # calcolo s(x) per ogni x in [a,b]
plt.plot(x,y,'k',color='b') #real function
plt.plot(x_knots,y_knots,'o',color='y') # k_not
plt.plot(x,y_scipy,'k—',color='g') # spline scipy
plt.plot(spline_completa.result_x,spline_completa.result_y,'k:',
    color='r') #Myspline
plt.legend(['real','knot','ScipyCompl','MysplineCompl'],loc='best
    ')
plt.show()
```

La figura 4.1 mostra il grafico di $f(x)$, il grafico della funzione spline cubica della libreria **Scipy** e il grafico della spline cubica implementata.

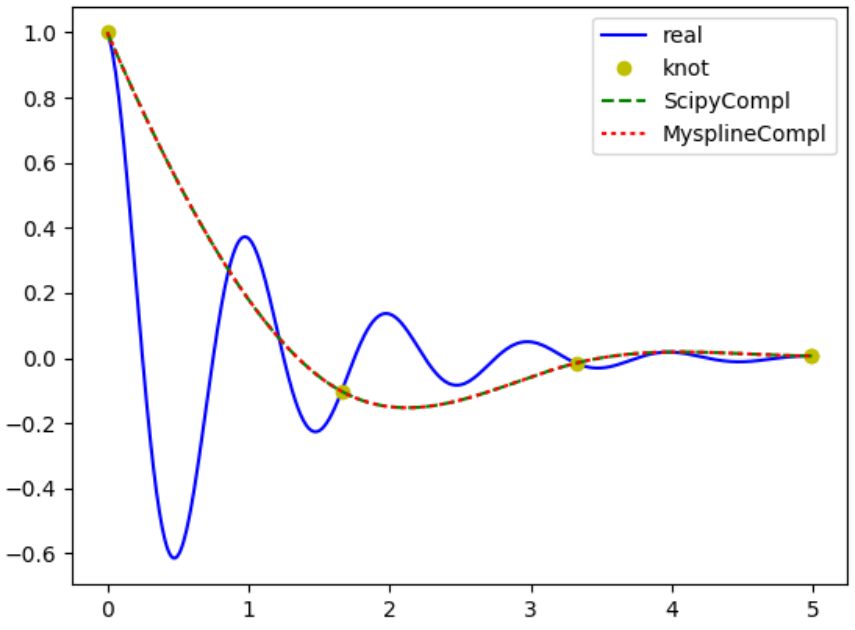
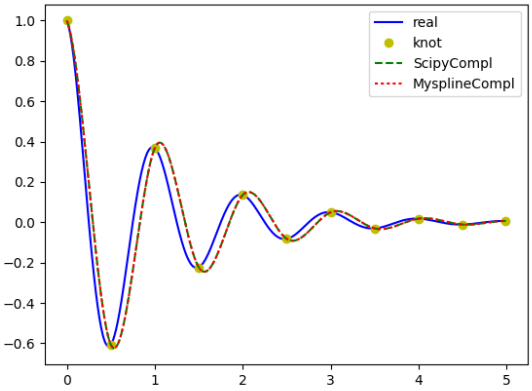
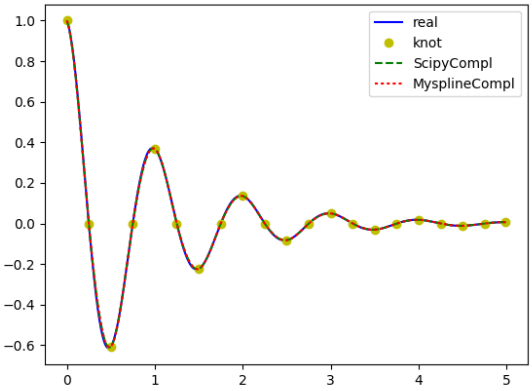


Figura 4.1

Aumentando il numero dei nodi si hanno i seguenti risultati:
Le figure 4.2 a) e b) mostrano l'andamento delle spline complete con 11 e 21 nodi di interpolazione.



(a) Spline completa 11 nodi



(b) spline completa 21 nodi

Figura 4.2: Spline complete

Il test per la spline naturale paragona la spline cubica naturale della libreria **Scipy** e la spline naturale implementata.

```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt
import Spline_Bezier as s_b
  def f(t):
  #funzione da interpolare
     return np.exp(-t) * np.cos(2*np.pi*t)

x = np.arange(0.0, 5.0, 0.01) # [a,b] = [0,0.01,0.02,...,5]
n = 3 # numero nodi di interpolazione
```

```
dist = (int(len(x)/n)) # distanza tra i nodi
y = []
for i in x:
  y.append(f(i)) # f(x) per ogni x in [a,b]
spline_natural = s_b.Spline(x,dist,f) # creazione della spline
    naturale
x_knots = [k for k in spline_natural.partition] # ascisse dei
    nodi di interpolazione
y_knots = [f(k) for k in spline_natural.partition] # ordinate dei
     nodi di interpolazione
tck = interpolate.CubicSpline(x_knots, y_knots,bc_type='natural')
    # Scipyspline naturale s(x)
y_scipy = tck(x) # calcolo s(x) per ogni x in [a,b]
plt.plot(x,y,'k',color='b') #real function
plt.plot(x_knots,y_knots,'o',color='y') # k_not
plt.plot(x,y_scipy,'k--',color='g') # spline scipy
plt.plot(spline_natural.result_x,spline_natural.result_y,'k:',
    color='r') #Myspline
plt.legend(['real','knot','ScipyNatural','MysplineNatural'],loc='
    best')
plt.show()
```

La figura 4.3 mostra il grafico di $f(x)$, il grafico della funzione spline cubica naturale della libreria **Scipy** e il grafico della spline naturale implementata.

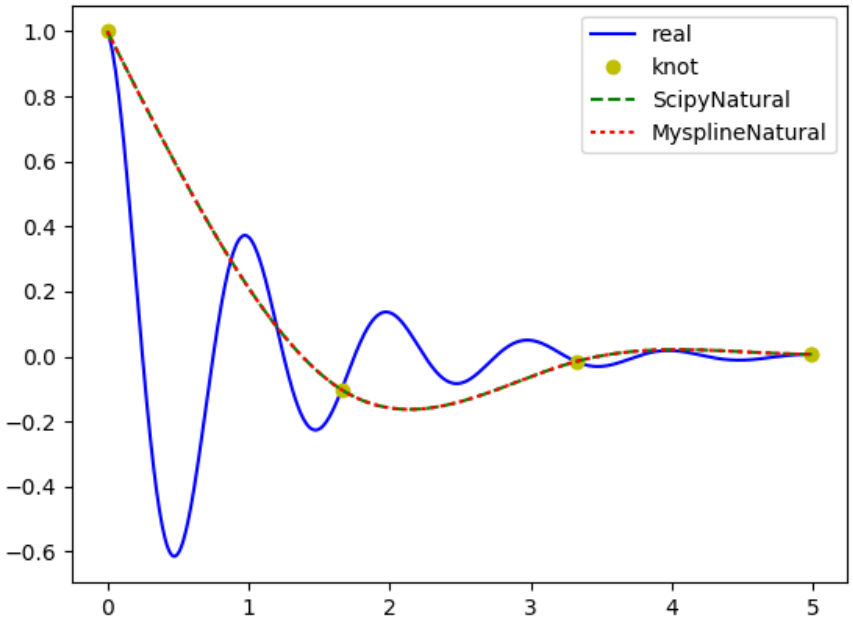
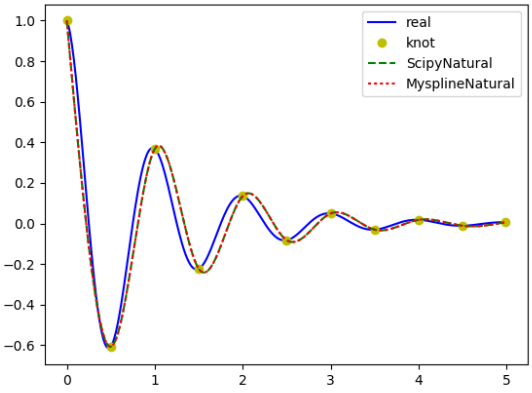
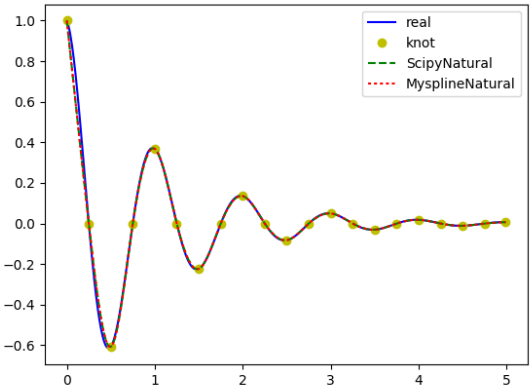


Figura 4.3

Aumentando il numero dei nodi si hanno i seguenti risultati:
Le figure 4.4 a) e b) mostrano l'andamento delle spline naturale con 11 e 21 nodi di interpolazione.



(a) Spline naturale 11 nodi



(b) spline naturale 21 nodi

Figura 4.4: Spline naturale

Il test per la spline not-and-knot paragona la spline cubica not-and-knot della libreria **Scipy** e la spline not-and-knot implementata.

```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt
import Spline_Bezier as s_b

def f(t):
  #funzione da interpolare
   return np.exp(-t) * np.cos(2*np.pi*t)

x = np.arange(0.0, 5.0, 0.01) # [a,b] = [0,0.01,0.02,...,5]
```

```
n = 3 # numero nodi di interpolazione
dist = (int(len(x)/n)) # distanza tra i nodi
y = []
for i in x:
y.append(f(i)) # f(x) per ogni x in [a,b]
spline_not_and_knot = s_b.Spline(x,dist,f,target=['not-and-knot'])# spline not-and-knot
x_knots = [k for k in spline_not_and_knot.partition] # ascisse dei nodi di interpolazione
y_knots = [f(k) for k in spline_not_and_knot.partition] # ordinate dei nodi di interpolazione
tck = interpolate.CubicSpline(x_knots, y_knots) # Scipyspline not-and-not s(x)
y_scipy = tck(x) # calcolo s(x) per ogni x in [a,b]
plt.plot(x,y,'k',color='b') #real function
plt.plot(x_knots,y_knots,'o',color='y') # k_not
plt.plot(x,y_scipy,'k--',color='g') # spline scipy
plt.plot(spline_not_and_knot.result_x,spline_not_and_knot.result_y,'k:',color='r') #Myspline
plt.legend(['real','knot','ScipyNot-knot','MysplineNot-knot'],loc='best')
plt.show()
```

La figura 4.5 mostra il grafico di $f(x)$, il grafico della funzione spline cubica not-and-knot della libreria **Scipy** e il grafico della spline cubica not-and-knot implementata.

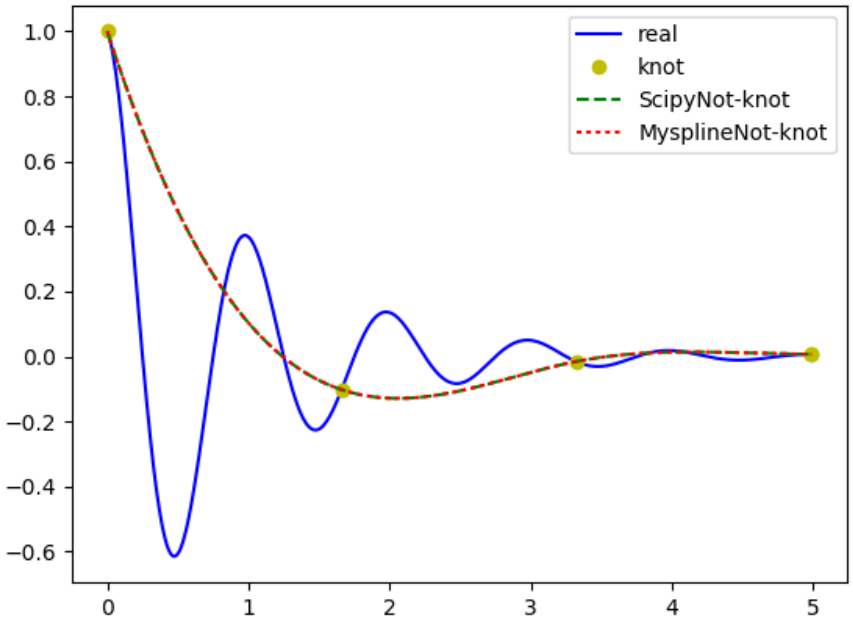
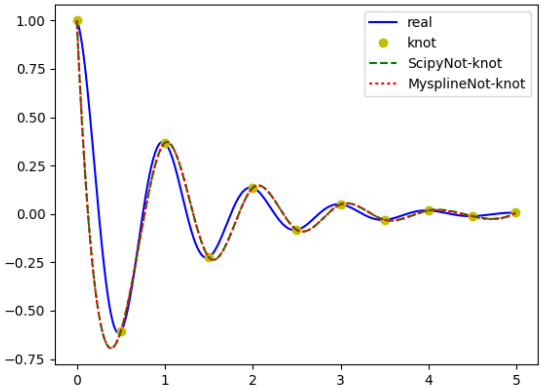
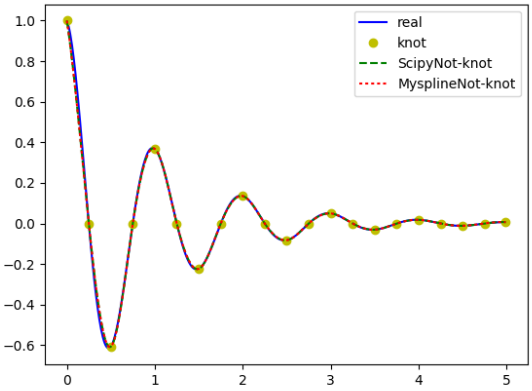


Figura 4.5

Aumentando il numero dei nodi si hanno i seguenti risultati:
Le figure 4.6 a) e b) mostrano l'andamento delle spline not-and-knot con 11 e 21 nodi di interpolazione.



(a) Spline not-and-knot 11 nodi



(b) Spline not-and-knot 21 nodi

Figura 4.6: Spline not-and-knot

4.2.3 La classe Bèzier

Analizziamo ora i principali algoritmi utilizzati per modellare le curve di Bèzier, mediante i loro punti di controllo, in modo da riuscire ad approssimare il grafico di una funzione.
La classe delle curve di Bèzier viene così definita:

```
1  class Bezier():
2    def __init__(self,interval , n_Knot  , function  , time):
3      self.n = n_Knot
4      self.time = time
5      self.partition , self.index_n_Knot =
               self.partition_interval(interval,self.n)
```

```
        self.partition.append(interval[len(interval)-1])
        self.f = function
        self.Matrix = self.init_system()
        self.controlPoint_x , self.controlPoint_y =
                    self.find_controlPoint()
        self.result_x , self.result_y = self.bezierCurve()
```

Così come è stato fatto per le spline bisogna andare a definire la tavola dei nodi partizionando l'intervallo $[a, b]$ in $n + 1$ nodi distinti che verranno utilizzati come nodi di interpolazione per il calcolo della curva di Bèzier. Riutilizziamo la funzione *partitionInterval* definita nell'implementazione delle spline.

```
def partition_interval (self,interval,n):
'''Partiziona l'intevallo in n+1 nodi equidistanti
  Args: interval = intervallo [a,b] da partizionare
      n = distanza tra i nodi
  Return: tavola dei nodi di interpolazione e indici dei nodi'''
  i = 0
  part = []
  idx = []
  while i < len(interval):
    part.append(float(interval[i]))
    idx.append(i)
    i += n
  if part.count(interval[len(interval)-1]) == 0: # aggiungo l'
   estremo dell'intervallo se non
  part.append(float(interval[len(interval)-1])) # e' stato
   incluso nella tavola
  idx.append(len(interval)-1)
  return part , idx
```

Andiamo a costruire il sistema definito nella sezione 3.4.1 utilizzando la seguente funzione

```
def init_system (self):
  M = [[0.0 for x in range(len(self.partition))]
       for y in range(len(self.partition))]

  for i in range(len(self.partition)):
    for j in range(len(self.partition)):

      if (i == 0 and j == 0) or (i == len(self.partition)-1
        and j == len(self.partition)-1):
          M[i][j] = 1.0
      elif i == 0 or i == len(self.partition)-1:
          M[i][j] = 0.0
      else:
          M[i][j] = self.binomial(len(self.partition)-1,j,i)
  return M
```

Dove le righe 2 e 3 inizializzano una matrice $(n+1)$ x $(n+1)$, le righe da 5-14 riempiono la matrice come è stato definito nel sistema della sezione 3.4.1 e la funzione *binomial* svolge il calcolo della formula

$$\mathcal{M}_{i,j} = \binom{n}{j} \left(\frac{i}{n}\right)^j \left(1 - \frac{i}{n}\right)^{n-j}$$

```
1  def binomial (self, n_b , j , i ):
2    binomial = (self.fattoriale(n_b)) / (self.fattoriale(j) *
       self.fattoriale(n_b-j))
3    k = i
4    p = j
5    l = (float(k) / float(n_b))
6    c = pow(l,p)
7    d =  pow((1-l),(n_b-j))
8
9    return binomial * c * d
10
11 ################ Fattoriale ################
12
13 def fattoriale(self,n,limit=1):
14   if n == 0 or n == 1:
15     return 1
16   result = 1
17   while n > limit:
18     result *= n
19     n -= 1
20   return result
```

Adesso per trovare l'unico polinomio di grado n che interpola la funzione nei nodi della partizione bisogna trovare i punti di controllo della curva di Bèzier. Non ci resta che risolvere il sistema 3.4.1 attraverso la seguente funzione

```
1  def find_controlPoint(self):
2    x_b , y_b = [] , []
3    for x in self.partition:
4      x_b.append(float(x))
5      y_b.append(float(self.f(x)))
6    x_i = linalg.solve(self.Matrix,x_b)
7    y_i = linalg.solve(self.Matrix,y_b)
8
9    return x_i , y_i
```

dove la chiamata linalg.solve(A,b) della libreria **NumPy** risolve il sistema lineare.
Una volta trovati i punti di controllo siamo ora in grado di calcolare la curva di Bèzier che interpola la funzione nei punti $(x_i, f(x_i)), i = 0, \ldots, n$ utilizzando la rappresentazione nella base di Bernstein con l'ausilio delle seguenti funzioni.

```
  def bernstain_polynomial (self, n_b , k , t ) :
  #Forma di Bernstein
    binomiale = (float(self.fattoriale(n_b)) /
    (float(self.fattoriale(k)) * float(self.fattoriale(n_b-k))))

    return ((binomiale * pow(t,n_b-k)) * pow((1-t),(k)))
  def bezier (self,x_b , n_b  , y_b , t):
    somma_x = 0
    somma_y = 0
    for i in range(n_b):#calcolo del polinomio con Bernstein
      somma_x += (x_b[i] * self.bernstain_polynomial(n_b-1,i,t))
      somma_y += (y_b[i] * self.bernstain_polynomial(n_b-1,i,t))
    return somma_x , somma_y

  def bezierCurve(self):
    y_bezier , r_x , r_y = [] , [] , []
    for i in range(len(self.time)):
      y_bezier.append(self.bezier(self.controlPoint_x,
      len(self.partition),self.controlPoint_y,self.time[i]))
    for i , j in y_bezier:
      r_x.append(float(i))#ascisse
      r_y.append(float(j))#ordinate

    return r_x , r_y
```

Dove la funzione *bernstainpolynomial* insieme alla funzione *bèzier* calcolano il polinomio utilizzando la base di Bernstein e la funzione *bezierCurve* calcola tutti i punti della curva in funzione dei punti di controllo e del tempo.

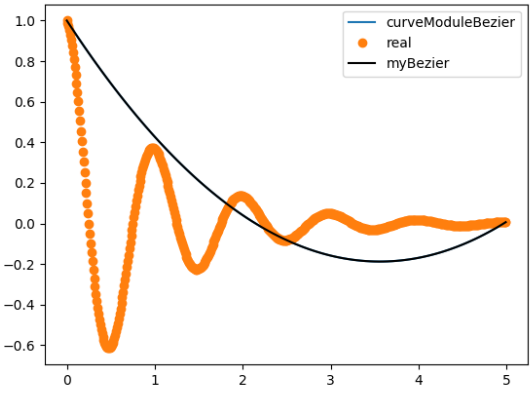
4.2.4 Test Curve di Bèzier

Consideriamo sempre la funzione $f(x) = e^{-x}\cos(2\pi x), x \in [a,b]$ con $a = 0$ e $b = 5$ partizionato in 500 punti. Scegliamo nodi equidistanti con $n = 250$ (abbiamo quindi $\frac{\#partizione[a,b]}{n} + 1 = 3$ punti di controllo).
Il seguente codice Python crea una curva di Bèzier, mettendola a paragone con la curva della libreria **bèzier** che calcola in modo efficiente la curva usando l'algoritmo di De Casteljau descritto nella sezione 3.2.

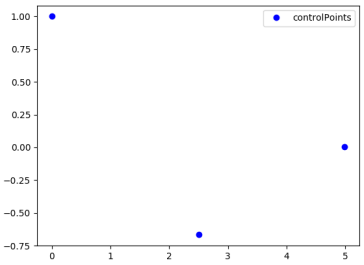
```
  import numpy as np
  import matplotlib.pyplot as plt
  import Spline_Bezier as s_b
  import bezier

  def f(t):
  #funzione
    return np.exp(-t) * np.cos(2*np.pi*t)
```

```
9
10  n = 250 # n nodi intervallo diviso in lunghezza intervallo
11  t1 = np.arange(0.0, 5.0, 0.01) # [a,b] = [0,0.01,0.02,...,5]
12  t2 = [] # y(x) con x in [a,b]
13  for i in t1:
14    t2.append(f(i))
15
16  time = np.arange(0.0,1.01,0.01) #tempo
17  Bezier_curva = s_b.Bezier(t1,n,f,time)
18  nodes2 = np.asfortranarray([ [0.0,0.0] for x in range(len(
      Bezier_curva.controlPoint_x))])
19  for i in range(len(nodes2)):
20    nodes2[i] = [Bezier_curva.controlPoint_x[i],Bezier_curva.
      controlPoint_y[i]]
21  curve2 = bezier.Curve.from_nodes(nodes2)
22  curve2.plot(len(t1))
23  plt.plot(t1,f(t1),'o',Bezier_curva.result_x,Bezier_curva.
      result_y,'k')
24  plt.legend(['curveModuleBezier','real','myBezier'],loc='best')
25  plt.show()
```

La figura 4.7 mostra il grafico di $f(x)$, il grafico della curva di bèzier della libreria **bèzier** e il grafico della curva implementata.



(a) grafico delle funzioni

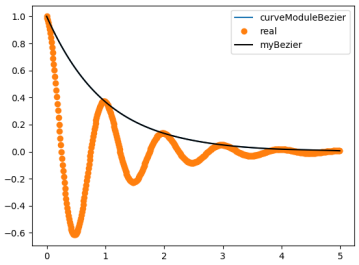


(b) Punti di controllo

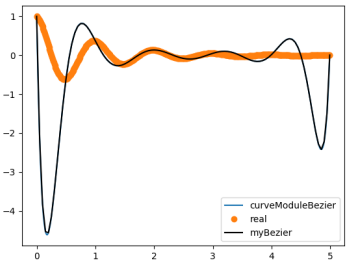
Figura 4.7: Curva di bèzier e i relativi punti di controllo

Incrementando il numero dei punti di controllo[1] che definiscono la curva di Bèzier, si riesce ad aumentare il grado di accuratezza per l'approssimazione delle funzione presa in esame. Le seguenti figure mostrano i grafici relativi alle curve di Bèzier aumentando i punti di controllo.

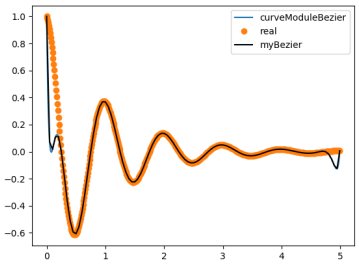
[1] Questo processo prende il nome di *"Degree elevation"*.



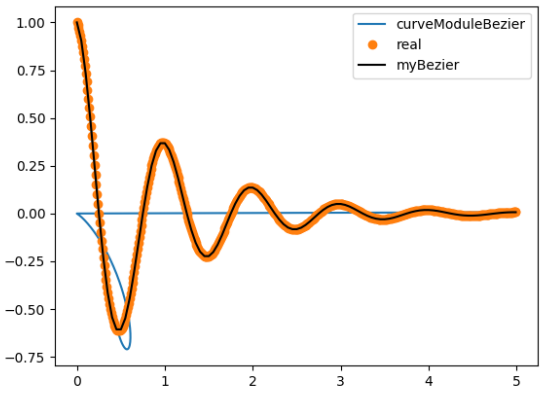
(a) 5 punti di controllo



(b) 10 punti di controllo



(c) 20 punti di controllo



(d) 50 punti di controllo

Figura 4.8: Curve di bèzier e i relativi punti di controllo

Nella figura 4.8 (*d*) si può notare che la curva della libreria **bèzier** non riesce ad approssimare il grafico della funzione in quanto per un numero di punti di controllo molto alto l'algoritmo di DeCasteljau risulta instabile, quindi possiamo concludere che l'algoritmo che utilizza la base di Bernstein è molto più stabile di quello di DeCasteljau.

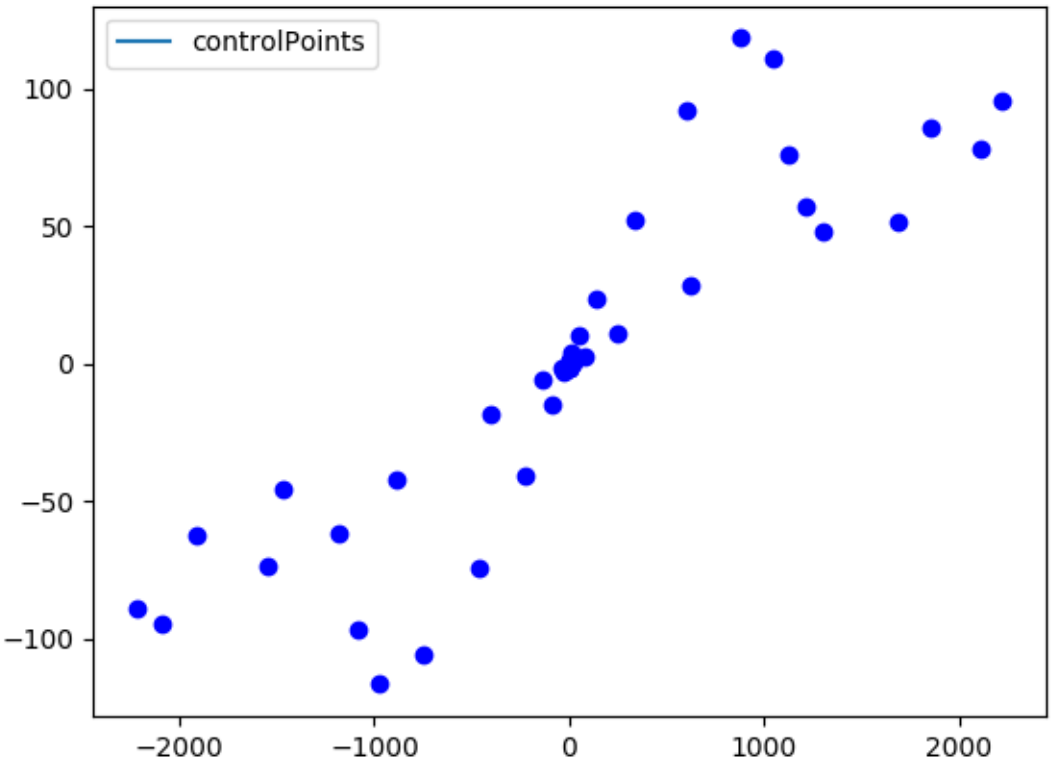


Figura 4.9: Disposizione dei punti di controllo della figura 4.8(d)

Conclusioni

Questa relazione descrive il procedimento seguito per la creazione di Spline e curve di Bèzier utilizzate per l'interpolazione polinomiale.
Il lavoro è stato svolto analizzando le principali tecniche matematiche per la costruzione di entrambi gli oggetti, in particolare abbiamo affrontato le tecniche di interpolazione polinomiale a tratti per quel che riguarda la creazione delle Spline di ordine 4. Si è passati poi allo studio del condizionamento associato a basi diverse, in particolare nel calcolo dei polinomi di Bèzier.
Si è proceduto ad un analisi dei requisiti per la creazione del codice Python, in particolare sono state studiate alcune librerie per velocizzare la prototipazione di algoritmi numerici.
Per le funzioni Spline, grazie alla teoria, l'implementazione del codice non ha presentato problemi e i risultati ottenuti, confrontati con quelli della libreria Scipy, non presentano molte differenze a parte per il costo in tempo che risulta migliore nelle Spline implementate della libreria presa come oracolo.
Mentre per quel che riguarda le curve di Bèzier, in fase di progettazione, sono stati riscontrati alcuni problemi nella costruzione del sistema lineare utilizzato per trovare i punti di controllo della curva. I problemi erano legati ad un errore nell'inizializzazione della matrice del sistema, la cui soluzione prevedeva che gli elementi della prima e dell'ultima riga fossero tutti 0 tranne uno, quello relativo all'estremo dell'intervallo che doveva essere uguale a 1 per far sì che la curva passasse per i due estremi come da definizione. I risultati ottenuti possono essere considerati ottimi se paragonati a quelli della libreria bèzier che risulta instabile per un numero di punti arbitrariamente alto.

Bibliografia

[1] R. Bevilacqua e O. Menchi. *Appunti di Calcolo Numerico* (2011-2012)

[2] R. T. Farouki and T. N. T. Goodman. *On the Optimal Stability of the Bernstein Basis* Published by: American Mathematical Society (1996)

[3] Hartmut Prautzsch , Wolfgang Boehm, Marco Paluszny *Bézier and B-Spline Techniques*, Springer Verlag Berlin Heidelberg (2002)

[4] Helmuth Spath , *Two Dimensional Spline Interpolation Algorithms*, University Oldenburg, Germany (1995)

[5] http://pagine.dm.unipi.it/bini/Didattica/IAN/appunti/appunti.html *Interpolazione polinomiale a tratti e funzioni Spline*