LABoratorio di DISEGNO

Guida al toolbox anmglib_4.1

ottobre 2024

Giulio Casciola

Indice

Capitolo 1

Ambiente MATLAB

1.1 ▪ L'ambiente MATLAB

MATLAB è un software commerciale sviluppato dalla *MathWorks*. Si tratta di un ambiente per il calcolo numerico, dove è possibile anche il calcolo simbolico, ma soprattutto è un potente linguaggio di programmazione.

La piattaforma è molto semplice da usare, ma molto prestante allo stesso tempo; la GUI è intuitiva e permette di ambientarsi velocemente. L'interfaccia grafica è composta da:

1. la *Command Window*, per inserire i comandi e interagire con MATLAB;
2. il *Workspace*, dove vengono elencate le variabili dichiarate e i loro valori;
3. l' *Editor*, per creare e modificare gli script in linguaggio MATLAB;
4. la finestra *Current Folder*, cioè il percorso file corrente, cioè quello nel quale stiamo lavorando.

1.1.1 ▪ MATLAB come strumento di calcolo

L'utilizzo di MATLAB come strumento per la computazione è, tra i principali impieghi di questo software, quello prescelto; infatti tramite questa piattaforma è possibile svolgere calcoli ed elaborazioni numeriche attraverso comandi in-line. MATLAB esegue operazioni aritmetiche, potenze, logaritmi ed esponenziali, funzioni trigonometriche oltre che numerose funzioni predefinite richiamabili attraverso il command prompt integrato. Oltre questo, MATLAB offre la possibilità di implementare algoritmi attraverso un linguaggio di programmazione ad alto livello e di testarne l'efficienza all'interno dello stesso ambiente. Grazie a questa caratteristica è stato possibile creare il toolbox `anmglib_4.1` e fare in modo che il suo utilizzo sia accessibile a tutti. Infatti, una volta aggiunto il toolbox ai path di MATLAB, tutte le funzioni sono disponibili e consultabili direttamente dall'utente, cosicché sia ridotta al minimo la distanza tra le nozioni teoriche e il loro utilizzo nella pratica.

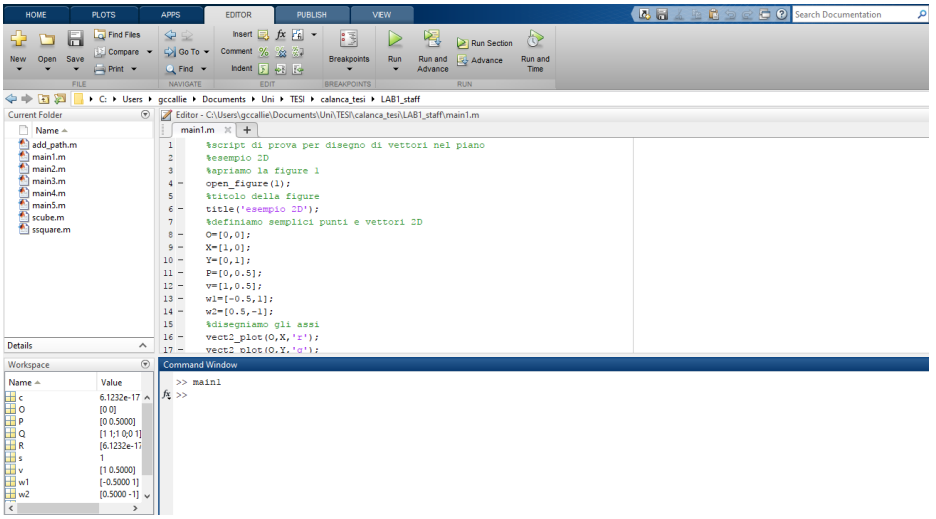

Figura 1.1. *L' interfaccia grafica di MATLAB*

1.1.2 ▪ Potenzialitè grafiche di MATLAB

Tra le molteplici applicazioni possibili di questo software è importante mettere in evidenza la predisposizione al disegno in genere e alla rappresentazione di figure geometriche in particolare. MATLAB infatti, dispone di un ambiente grafico integrato, essenziale allo svolgimento delle esercitazioni in modo pratico ed efficace. La maggior parte delle funzionalità del toolbox anmglib_4.1 si identificano infatti nella rappresentazione grafica di elementi geometrici, pertanto, è indispensabile la semplicità di gestione delle finestre di disegno, create con l'ausilio del comando `figure`. Questa peculiarità permette di rappresentare all'interno delle *figure window* qualsiasi disegno o grafico tramite l'uso opportuno delle funzioni `plot` e `mesh` di MATLAB. Queste sono, infatti, alla base di tutte le altre function di disegno: `plot` è legata al disegno di punti e linee (vettori, curve 2D e 3D), mentre `mesh` alla rappresentazione di superfici. A differenza di altri linguaggi di programmazione ad alto livello, (ad esempio Java, C, Python, Javascript, etc.) per i quali sono necessarie librerie grafiche aggiuntive, oltre che accortezze sintattiche non indifferenti, MATLAB offre un'integrazione predefinita e immediata per la creazione e modifica di funzioni, dati ed elementi geometrici. La modifica del disegno può avvenire in modo *dinamico*, nel senso che non c'è necessità, come nel caso di altri linguaggi, di modificare il codice, ricompilarlo ed eseguirlo, bensì MATLAB offre la possibilità di correggere "live" l'output del nostro script. Specifici tool all'interno della finestra grafica permettono, tra le varie cose, di estrarre informazioni dal disegno, cambiare le modalità di visualizzazione, controllandone gli effetti in tempo reale.

Capitolo 2

Toolbox anmglib_4.1

2.0.1 ▪ Struttura del toolbox

La struttura di questo toolbox può essere analizzata in vari modi. Un modo interessante di vederla è nella sua suddivisione per livelli di *profondità* delle funzioni: partendo dal livello più esterno di quelle subito utilizzabili dagli utenti, si raggiunge quello più interno di function richiamabili solo da utenti più esperti.

1. Le funzioni del *primo* livello sono quelle usate da tutti gli utenti, anche quelli meno esperti: è sufficiente effettuare una chiamata di funzione specificandone correttamente gli argomenti e questa produrrà un output e/o un disegno.
   Un esempio è la funzione `curv2_bezier_plot` che è facilmente richiamabile definendo la struttura di una curva di Bézier, il numero di punti di valutazione, i parametri di disegno e che restituirà in output le coordinate dei punti valutati e disegnerà la curva specificata dai dati di input.

2. Nel *secondo* livello, le funzioni sono richiamate all'interno di quelle del primo e possono quindi essere usate direttamente solo dagli utenti che hanno più dimestichezza con la libreria; per esempio la `curv2_bezier_plot` utilizza la `decast_val` che può essere richiamata direttamente solo quando l'utente è capace di sfruttarne l'output: la function restituisce le coordinate della curva valutate nei punti specificati che serviranno a disegnarla. Nella `curv2_bezier_plot` questi valori di output vengono passati alla `mesh_curv2_plot`, che disegna effettivamente i punti della curva.

3. Per quanto riguarda il *terzo* livello, le funzioni di questa sezione sono quelle più "nascoste" e possono non essere mai considerate da un utente inesperto. All'interno di queste non sono chiamate altre funzioni della libreria, ma solamente funzioni base di MATLAB. Un esempio di questa categoria sono le fuction per disegnare le funzioni base di Bernstein, spline e NURBS.

2.0.2 ▪ Tipi di dati utilizzati

Un utente, usando la libreria, dovrà lavorare su diversi tipi di dato, per la maggior parte utilizzerà dati di tipo numerico, per il calcolo e la definizione dei costrutti

geometrici, oltre a questi però si servirà anche di dati di tipo char e di strutture. Le function, come dicevamo, operano principalmente su input numerici, che possono essere coordinate di punti, intervalli ecc. MATLAB memorizza tutti i valori numerici come floating point a doppia precisione. E' comunque possibile memorizzare numeri e array di numeri come interi o a singola precisione, questo comporta un più efficiente uso della memoria rispetto alla doppia precisione. Per quanto riguarda i dati di tipo testo, MATLAB fornisce due diversi modi per memorizzarli: *character array* e *string array*. Il primo costrutto memorizza il testo come una sequenza di caratteri (per es. `chr='testo'`), allo stesso modo di una sequenza numerica e con gli stessi metodi di accesso. Uno *string array* funziona come un contenitore per la stringa (per es. `str="testo"`) e fornisce alcuni metodi per utilizzarli e lavorarci. Questi generi di dato sono utilizzati, all'interno delle function della libreria, in special modo per specificare le opzioni di disegno. Per ogni categoria geometrica, tipo di calcolo e classe, le funzioni di disegno hanno infatti alcuni parametri facoltativi che è possibile impostare per modificare l'aspetto del disegno. Alcune tra queste opzioni sono appunto di tipo *char* e in particolare vengono memorizzate come *character array*, dato che non c'è necessità di effettuare operazioni come stringhe. L'ultimo argomento di queste funzioni è `varargin` che rende possibile la specifica di alcuni parametri, non obbligatori, per personalizzare il disegno. Essi sono i seguenti:

- *LineSpecification*, definisce lo stile della linea, il simbolo usato come marker e il colore
- *LineWidth*, specifica la larghezza della linea
- *MarkerEdgeColor*, indica il colore esterno del marker
- *MarkerFaceColor*, indica il colore interno del marker
- *MarkerSize*, stabilisce la misura del marker

Le Structure Array permettono l'aggregazione di più variabili in modo similare a quello degli array, ma a differenza di questi, possono contenere variabili non omogenee. Esse vengono memorizzate in campi, ai quali si può accedere attraverso l'operatore "punto", in questo modo: nomeStruct.nomeCampo. All'interno della libreria sono presenti due macro tipi di strutture, uno raggruppa quelle specifiche per le curve e uno quelle per le superfici. Per entrambi i tipi sono presenti "sottostrutture" proprie di ogni classe geometrica. Per le curve (2D e 3D) avremo le strutture:

- bezier, utilizzata per contenere le informazioni relative ad una curva di Bézier. I campi della struttura interessano il grado: bezier.deg, i punti di controllo: bezier.cp e l'intervallo di definizione: bezier.ab.
- ppbezier contiene i dati che definiscono una curva di Bézier a tratti, i campi sono: ppbezier.deg, ppbezier.cp e ppbezier.ab, che memorizza la partizione dell'intervallo di definizione di tanti elementi quanti sono i tratti della curva.
- mdppbezier contiene le informazioni di definizione di una curva di Bézier a tratti multi-degree. I campi sono gli stessi di una ppbezier con la differenza che mdppbezier.deg contiene un'array di gradi per ogni tratto della curva.

- spline, struttura di una curva spline composta da 3 campi: spline.deg contenente il grado, spline.cp i punti di controllo, spline.knot il vettore dei nodi.
- nurbs raggruppa le variabili che definiscono una curva NURBS, i campi sono i medesimi della struttura spline con l'aggiunta di un ulteriore campo, nurbs.w contenente il vettore dei pesi della curva.

Le strutture utilizzate per la memorizzazione delle informazioni relative alle superfici sono molto simili alle precedenti, esse sono le seguenti:

- surfbezier contiene i campi relativi ad una superficie di Bézier, i campi sono: surfbezier.deguv che contiene i gradi della superficie in u e in v, parametri di valutazione della curva, surfbezier.cp per i punti di controllo, surfbezier.ab e surfbezier.cd al cui interno è memorizzato, rispettivamente, l'intervallo di definizione in u e in v.
- surfspline per le superfici spline ha campi: surfbezier.deguv definito come sopra, il campo surfbezier.cp per i punti di controllo, surfbezier.ku e surfbezier.kv che contengono, rispettivamente il vettore dei knot in u e in v.
- surfnurbs è usata per rappresentare una superficie NURBS, i campi sono gli stessi della surfspline con l'aggiunta del campo surfnurbs.w che contiene la griglia dei pesi.

```
bezier.cp=[0,2;1,1;2,1;3,2];
bezier.deg=length(bezier.cp(:,1))-1;
bezier.ab=[0 1];
curv2_bezier_plot(bezier,40,'-md',2,'y','k',8);
```

E' interessante approfondire le motivazioni di alcune scelte fatte a proposito della forma della libreria o toolbox. Il toolbox anmglib_4.1 è composto da un numero abbastanza elevato di funzioni (circa 250) che possono, a prima vista, sembrare ridondanti per via dei nomi ripetuti e della somiglianza tra le varie funzioni. Innanzitutto, la libreria è composta da molte funzioni per tenere separati argomenti e sottoargomenti, in modo che agli utenti risulti più chiara la divisione delle diverse funzionalità e il loro uso. Quindi per ogni categoria geometrica si avranno tante funzioni quante le operazioni e i calcoli che si possono effettuare su quel tipo di curva, superficie, solido. Come esempio vediamo alcune funzioni per curve 2D in forma parametrica:

```
curv2_plot.m
curv2_trans_plot.m
curv2_tan_plot.m
curv2_vel_plot.m
curv2_comb_plot.m
curv2_kur_plot.m
```

Per quanto riguarda il namespace utilizzato nel toolbox anmglib_4.1, si è seguita una logica basata fondamentalmente sull'immediatezza di comprensione dei nomi delle function; questi infatti sono autoesplicativi e hanno una struttura ripetitiva. Per questo motivo, tutti i nomi delle funzioni del primo livello sono stati scelti seguendo lo standard seguente:

1. La prima parte del nome indica il tipo di geometria sulla quale agisce la function. In particolare sarà una tra queste: *point* per i punti, *vect* per i vettori, *plane* per superfici piane, *curv2* e *curv3* se si tratta, rispettivamente, di curve 2D e curve 3D, *surf* per le superfici e *solid* per quanto riguarda i solidi.

2. La seconda componente del nome invece, identifica la classe sulla quale stiamo lavorando. Per punti, vettori e superfici piane non c'è necessità di specificarla, altrimenti le classi sono: *bezier*, *ppbezier*, *mdppbezier*, *spline* e *nurbs* per curve 2D e 3D, *bezier*, *spline*, *nurbs* per superfici, *box*, *sphere*, *cone* e *cylinder* per i solidi.

3. La terza parte, specificata quando necessario, indica il tipo di operazione o calcolo che vogliamo effettuare. Per quanto riguarda le operazioni su file con estensione db sarà del tipo *save* o *load*, per traslazioni e rotazioni *trans*, per il calcolo della curva di interpolazione *interp*, se si tratta di operazioni su geometria differenziale avremo *tan* per il vettore tangente, *vel* per il vettore velocità, *comb* per la rappresentazione a pettine, *kur* per la funzione curvatura.

4. Il quarto suffisso, anche questo opzionale, è *plot* e viene utilizzato quando la funzione, oltre al calcolo, disegna i valori ottenuti.

Lo standard di naming delle function è sicuramente un modo efficace per semplificare l'approccio iniziale all'utilizzo del toolbox anmglib_4.1. In secondo luogo è utile al fine di acquistare dimestichezza e rendere scorrevole la creazione di script per le diverse categorie e classi geometriche. All'interno dell'ambiente MATLAB però, è la funzione help lo strumento vero e proprio di documentazione e supporto all'utilizzo di questo toolbox e, in generale, di tutte le funzioni della piattaforma. Questo mezzo permette di stampare in output i commenti sottostanti la dichiarazione della funzione. Le righe commentate contengono:

- La definizione completa della chiamata della funzione
- Una breve spiegazione del suo utilizzo
- La descrizione degli argomenti in input
- La descrizione degli argomenti in output

Grazie alla function help è possibile sapere cosa fa ogni funzione e avere chiarimenti sulla sua chiamata, direttamente all'interno dell'ambiente MATLAB.

Per usare il comando help bisogna preliminarmente conoscere il nome della funzione di cui si vogliono le informazioni. A tal fine il comando "help_anmglib" seguito da un suffisso ritorna tutti i nomi delle function della libreria con quel suffisso. Per esempio per conoscere il nome completo di tutte le funzioni che nel nome hanno il suffisso "tan", si può dare il comando:

```
>> help_anmglib tan
curv2_bezier_tan_plot.m curv3_from_surf_tan_plot.m
curv2_nurbs_tan_plot.m curv3_nurbs_tan_plot.m
curv2_spline_tan_plot.m curv3_spline_tan_plot.m
curv2_tan_plot.m          curv3_tan_plot.m
curv3_bezier_tan_plot.m
```

Capitolo 3

Utilizzi del toolbox

Di seguito verranno illustrati alcuni possibili casi d'uso del toolbox. Dato che l'obiettivo primario dell' `anmglib_4.1` è di offrire un supporto didattico, verranno analizzati più approfonditamente gli esempi relativi a questo tipo di utilizzo. D'altra parte però, non sono da tralasciare le possibilità che offre la libreria di staccarsi dal suo lato più scolastico.

Ricordiamo che all'inizio di ogni sessione di lavoro deve essere eseguito lo script `add_path.m` che aggiunge il percorso del toolbox ai path di MATLAB; questo permette a MATLAB di sapere dove sono le funzioni da utilizzare.

3.1 ▪ Plotting di elementi geometrici

Vediamo innanzitutto alcuni esempi di function per il disegno di elementi geometrici. Solitamente la loro rappresentazione grafica viene indicata con il termine inglese *plot*, che significa letteralmente tracciare, disegnare. Utilizzando il toolbox `anmglib_4.1` è possibile definire in modo semplice, attraverso strutture o array, anche oggetti complessi, passabili poi come argomenti alle funzioni per il disegno. Queste function sono utili per comprendere tutte quelle del toolbox, infatti una volta compreso il meccanismo di funzionamento di queste, provandole e variandone i parametri in input, si è in grado di usare tutte le altre.

3.1.1 ▪ Plotting di punti e vettori

Consideriamo lo script 4.1; la funzione `open_figure(id)` permette l'apertura di una finestra grafica con un suo id (identificatore). La funzione `axis_plot(lax)` disegna gli assi cartesiani di lunghezza `lax`. Ora definiamo un quadrato definendo i suoi vertici 2D in un array `P` di dimensione 2×5 e invochiamo la `point_plot(P,varargin)` per fare il plot del quadrato (la poligonale chiusa i cui vertici sono memorizzati in `P`). Utilizziamo la function `vect2_plot(P,v,varargin)` per disegnare il vettore 2D definito dall'array `v2` di componenti `(1,0)` applicato al punto medio del secondo lato del quadrato e denominato `E`; questo per avere un versore ortogonale ad un lato del quadrato. Analogamente poi per un versore ortogonale al terzo lato del quadrato.

```
1 %apertura di una finestra
```

```
2  open_figure(1);
3  %plot degli assi cartesiani
4  axis_plot(4);
5  %definizione punti 2D nella matrice P
6  P=[1,1; 3,1; 3,3; 1,3; 1,1];
7  point_plot(P,'-ms',1,'m','b',8);
8  % disegna dei versori ortogonali a due lati
9  E=0.5.*(P(2,:)+P(3,:));
10 v2=[1,0];
11 vect2_plot(E,v2,'b-',1,'m','c');
12 N=0.5.*(P(3,:)+P(4,:));
13 v3=[0,1];
14 vect2_plot(N,v3,'b-',1,'m','c');
```

Listing 3.1. *Disegno di punti e vettori*

L'output grafico relativo al codice precedente si può vedere nella Figura 3.1. I punti definiti nella matrice `P` sono evidenziati da quadratini blu mentre le linee che li collegano in magenta grazie alla *LineSpecification* definita da `'-ms'`, cioè tratto continuo, colore magenta e marker square come vertici. I restanti parametri sia della function `point_plot` che `vect2_plot` definiscono rispettivamente i parametri di "LineWidth", "MarkerEdge", "MarkerFace" e "MarkerSize".

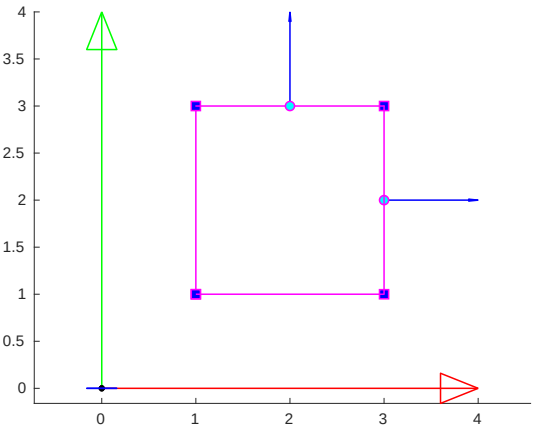

Figura 3.1. *Plotting di punti, e vettori*

3.1.2 ▪ Plotting di curve parametriche e di Bézier

Nel prossimo esempio vediamo la definizione e il plotting di una curva in forma parametrica e di una curva di Bézier. Grazie a questi esempi possiamo mettere in evidenza le singole caratteristiche e proprietà e studiarne le differenze. Per quanto riguarda le curve in forma parametrica è necessario scrivere la sua espressione parametrica in una function che memorizzeremo con un nome ed estensione `.m`. Nel nostro esempio la curva è una circonferenza e la sua espressione parametrica è scritta nella function `circle(t)` il cui corpo è mostrato nel listato 3.2: essa calcola x e y in funzione del parametro t.

```
%circonferenza di centro l'origine e raggio unitario
function [ x,y ] = c2_circle( t)
x = cos(t);
y = sin(t);
end
```

Listing 3.2. *Funzione per la circonferenza*

Per la definizione di una curva di Bézier, invece, è necessario creare la struttura che la contiene definendo i campi per i punti di controllo, il grado e l'intervallo di definizione; vedi lo script 3.3.

```
%creazione della struttura
bez.cp=[0,2;1,1;2,1;3,2];
bez.deg=length(bez.cp(:,1))-1;
bez.ab = [0  1];
```

Listing 3.3. *Struttura di una curva di Bézier*

Per disegnare la curva parametrica definita in precedenza (circonferenza) occorre invocare la `curv2_plot(curvname,a,b,np,varargin)` alla quale si passano come argomenti: il nome della function con l'espressione parametrica della curva, gli estremi `a` e `b` dell'intervallo di definizione, il numero di punti di valutazione e i parametri di disegno. La curva di Bézier invece è disegnata richiamando la funzione `curv2_bezier_plot(bezier,np,varargin)` alla quale si passano come argomenti: la struttura della curva, il numero di punti in cui valutarla e le opzioni di disegno. La chiamata alla function `point_plot` permette di disegnare un array di punti, per esempio i punti 2D contenuti nell'array `bez.cp`, ossia i punti 2D di controllo della curva di Bézier. Si veda lo script 3.4.

```
%disegno della circonferenza
curv2_plot('c2_circle',0,2*pi,40,'m');
%disegno della curva di Bezier e dei cp
curv2_bezier_plot(bez,40,'r',2);
point_plot(bez.cp,'g-o',1,'k','b');
```

Listing 3.4. *Disegno curva parametrica e curva di Bézier*

Dopo aver aperto una finestra grafica e disegnato gli assi, le linee di codice descritte in: 3.2, 3.3 e 3.4 disegneranno le curve di Figura 3.2.

Si noti che la libreria mette a disposizione alcune function per disegnare curve elementari come un segmento retto (funzione `line_plot`) o una circonferenza 2D (funzione `circle2_plot`).

Si noti inoltre che tutte le funzioni di disegno (suffisso `plot`) prevedono come output, se richiesto, le coordinate dei punti che vengono disegnate
`P=curv2_plot(curvname,a,b,np,varargin);`
`P=curv2_bezier_plot(bezier,np,varargin);`
dove `P` è un array np x 2. Ancora, passando come `np` un numero negativo si richiede alla funzione di plotting di valutare i punti da disegnare, ma di non disegnarli.

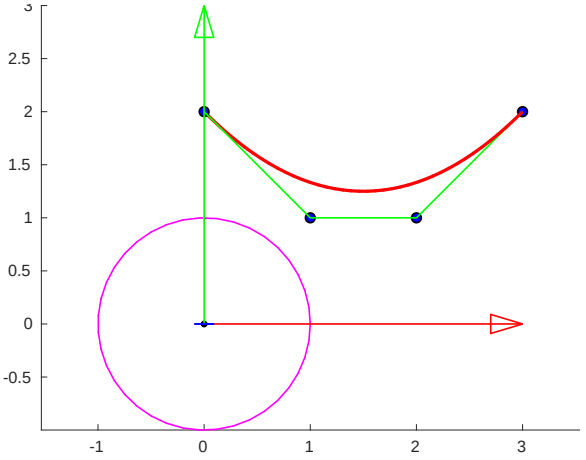

Figura 3.2. *Circonferenza e curva di Bézier*

3.1.3 ▪ Plotting di curve spline e NURBS e confronto

Vediamo ora un caso d'utilizzo del toolbox per curve spline e curve NURBS. Come si sa dalla teoria, le curve NURBS sono una generalizzazione delle spline, infatti una curva spline non è altro che una NURBS con tutti i coefficienti pesi settati al valore 1. Ma quali sono i vantaggi di utilizzare curve NURBS al posto delle spline? Rispondiamo a questa domanda con un'altra. Come possiamo definire una spline in modo che disegni una circonferenza? Fissando 8 control-point, 12 nodi e provando man mano ad alzarne il grado non riusciamo a disegnarla. Questo succede perché l'espressione parametrica della circonferenza, $x = 2t/(1+t^2)$ e $y = (1-t^2)/(1+t^2)$, è razionale anziché polinomiale. La necessità di rappresentare questo tipo di curve ha indotto l'introduzione delle NURBS, ottenute "sollevando" i punti di controllo $3D$ ($2D$) a $4D$ ($3D$) grazie ai coefficienti pesi, costruendo la spline $4D$ ($3D$) e proiettandola infine di nuovo in $3D$ ($2D$) attraverso una proiezione centrale. Per aiutarci a vedere graficamente questo concetto definiamo in un file, che chiameremo `c2_nurbs_circle`, di estensione `db`, una curva NURBS a due dimensioni, con gli stessi punti di controllo e nodi della spline. In questo file saranno contenute le informazioni sui campi: grado, punti di controllo e nodi. I punti di controllo avranno ora 3 coordinate poiché consideriamo un sistema di coordinate omogenee, ovvero definite a meno di un coefficiente di proporzionalità. Il vettore dei pesi sarà quindi ricavabile grazie a queste e sarà formato dall'ultima coordinata di ogni punto. Ora non resta che estrarre le informazioni contenute in `c2_nurbs_circle.db`, caricarle all'interno di una struttura NURBS e disegnare la circonferenza.

```
1 %carico il file db nella struttura nurbsCirc
2 nurbsCirc=curv2_nurbs_load('c2_nurbs_circle.db');
3 %disegno la curva definita dal file db
4 open_figure(1);
```

```
curv2_nurbs_plot(nurbsCirc,10,'b-',2);
%metto in evidenza i control point
point_plot(nurbsCirc.cp,'ko',1,'y','k');
```

Listing 3.5. *Circonferenza NURBS*

Possiamo notare come, attraverso semplici chiamate di funzione, si sia riusciti a verificare una supposizione altrimenti difficile da comprendere appieno. Questa è una delle caratteristiche che rendono `anmglib_4.1` un ottimo strumento di supporto allo studio di chi si avvicina per la prima volta a concetti di modellazione geometrica.

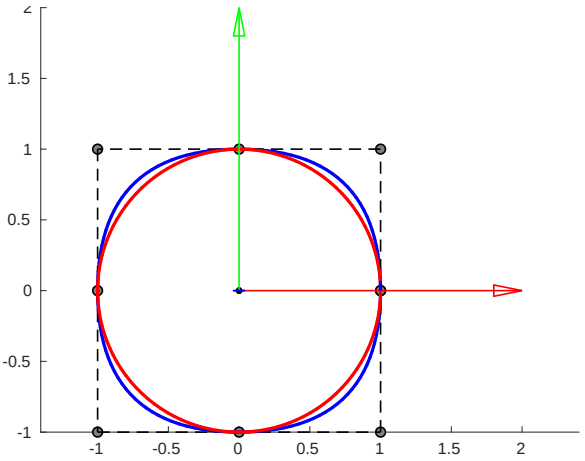

Figura 3.3. *Confronto tra spline e NURBS*

3.1.4 ▪ Plotting di solidi

Per quanto riguarda i solidi, è possibile disegnare alcune forme semplici come cubo, sfera, cono e cilindro. Vediamo nello script 3.6 un esempio di rappresentazione di un cubo, disegnato attraverso la funzione `solid_box_plot(a,b,nu,nv,type,varargin)` che prende in input due vertici opposti del cubo, `a=[2,0.5,0]` e `b=[3,1.5,1]`, il numero di punti di valutazione per faccia `nu=5`, `nv=5`, il tipo di disegno `type=3` e le proprietà del disegno. Possiamo osservare l'output grafico dei solidi nella Figura 3.4.

```
%setting della colormap
colormap(spring);
%disegno di un cubo
solid_box_plot([2,0.5,0],[3,1.5,1],5,5,3,'c','b');
%disegno di un cilindro
solid_cylinder_plot([1,2,0],[1,2,2],0.5,30,10,3.);
```

Listing 3.6. *Codice per il plot di un cubo e un cilindro*

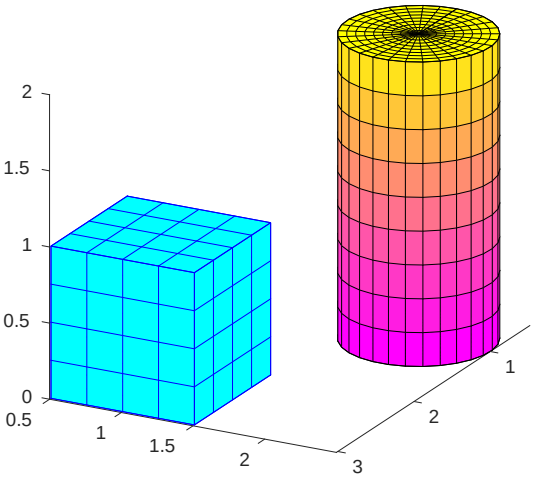

Figura 3.4. *Plot di cubo e cilindro*

3.2 ▪ Geometria differenziale

Come la quasi totalità delle function del toolbox, quelle per il calcolo della geometria differenziale di curve e superfici, disegnano ciò che viene richiesto e restituiscono in output i valori calcolati. Per calcolare e disegnare ad esempio le funzioni velocità, curvatura e torsione di una curva di Bézier $3D$ il codice utilizzato sarà il seguente:

```
1  %definizione curva 3D di Bezier
2  bezQ.cp=[0,2,0;1,1,1;2,1,1;3,2,2];
3  bezQ.deg=length(bezQ.cp(:,1))-1;
4  bezQ.ab=[0  1];
5  %disegno della curva e relativi punti di controllo
6  open_figure(1)
7  curv3_bezier_plot(bezQ,40,'b',2);
8  point_plot(bezQ.cp,'k-o',1);;
9  %funzione velocita'
10 open_figure(3)
11 vel = curv3_bezier_vel_plot(bezQ,20,'m');
12 %funzione curvatura
13 open_figure(2)
14 kur = curv3_bezier_kur_plot(bezQ,20,'b');
15 %funzione torsione
16 open_figure(4)
17 tor = curv3_bezier_tor_plot(bezQ,20,'y');
```

Listing 3.7. *Codice per il plot di funzione velocità e curvatura*

Nello script 3.7 le tre function
`curv3_bezier_vel_plot`,
`curv3_bezier_kur_plot`,
`curv3_bezier_tor_plot`
calcolano i valori delle funzioni, rispettivamente, velocità, curvatura e torsione,

in una discretizzazione dell'intervallo di definizione della curva, specificato dal secondo parametro delle function (`20` in questo caso). All'interno delle variabili `kur,vel` e `tor` vengono salvate le coordinate dei valori calcolati, che permetteranno successivamente il plot della funzione. Possiamo in alternativa calcolare il valore di queste funzioni in punti specifici. Nello script 3.8, è presentato un esempio di quest'operazione sulla curva di Bézier $3D$ definita in precedenza: viene definito in t una lista di punti, questa viene poi passata come parametro alla function `decast_valder` insieme alla struttura della curva di Bézier e all'ordine di derivazione (in questo caso il grado stesso della curva). Successivamente, tra i valori della curva nei parametri t ne scelgo tre su cui applicare le formule per velocità, curvatura e torsione in un punto.

```
%definizione parametro in cui valutare e valutazione
t=bezQ.ab(1)+0.25*(bezQ.ab(2)-bezQ.ab(1));
C=decast_valder(bezQ,bezQ.deg,t);
%estraggo i valori di derivata prima nel punto
C1=[C(2,1,1),C(2,1,2),C(2,1,3)];
%estraggo i valori di derivata seconda nel punto
C2=[C(3,1,1),C(3,1,2),C(3,1,3)];
%estraggo i valori di derivata terza nel punto
C3=[C(4,1,1),C(4,1,2),C(4,1,3)];
%calcolo di velocita', curvatura e torsione
vel=norm(C1,2);
kur=norm(cross(C1,C2),2)/vel^3;
M=[C1',C2',C3'];
tor=det(M)/norm(cross(C1,C2),2)^2;
```

Listing 3.8. *Codice per il plot di funzione velocità curvatura e torsione*

Un'altra interessante applicazione del toolbox sull'argomento geometria differenziale è il disegno del piano tangente ad una superficie. Consideriamo un paraboloide; i passaggi per disegnare il piano tangente in un punto scelto `P`, definito in termini dei valori parametrici $u0, v0$, sono espressi nello script 3.9.

```
%plot della superficie
surf_plot('s_parabola',0,2,20,0,2*pi,20,3,'flat');
%scelgo un punto P in termini dei parametri u e v
u0=1; v0=pi/2;
[xp,yp,zp]=s_parabola(u0,v0);
P=[xp,yp,zp];
%disegno la isocurva per u
curv3_from_surf_plot('s_parabola',0,2*pi,20,2,u0,'m');
%determino il vettore tangente all'isocurva per u in v
[xvt,yvt,zvt]=curv3_from_surf_tan_plot('s_parabola',v0,v0
    ,1,2,u0,'m',2);
%disegno la isocurva per v
curv3_from_surf_plot('s_parabola',0,2,20,1,v0,'b');
%determino il vettore tangente all'isocurva per v in u
[xut,yut,zut]=curv3_from_surf_tan_plot('s_parabola',u0,u0
    ,1,1,v0,'b',2);
%disegno il piano tangente in P
n=cross([xut,yut,zut],[xvt,yvt,zvt]);
```

```
17 plane_plot(n,P,-1.5,1,-1.5,1,[0.5  0.5  0.5]);
```

Listing 3.9. *Codice per disegnare il piano tangente ad un paraboloide*

La `surf_plot` disegna la superficie paraboloide, definita in forma parametrica nel file `s_parabola.m`. Il punto P appartiene alla superficie e viene calcolato ponendo `u0=1` e `v0`=$\pi/2$. Successivamente occorre estrarre l'isocurva per `u0` e determinare il vettore tangente a questa in `v` tramite le function `curv3_from_surf_plot` e la `curv3_from_surf_tan_plot` con opportuni argomenti. Lo stesso passaggio viene eseguito per l'isocurva per `v0`, per la quale viene determinato il vettore tangente in `u`. Per il plot del piano tangente viene infine chiamata la `plane_plot` che esegue il plot a partire dal punto `P` e la normale al piano data dal prodotto vettoriale tra i vettori tangente trovati. Il risultato del codice appena descritto è rappresentato in Figura 3.5 da due differenti punti di vista.

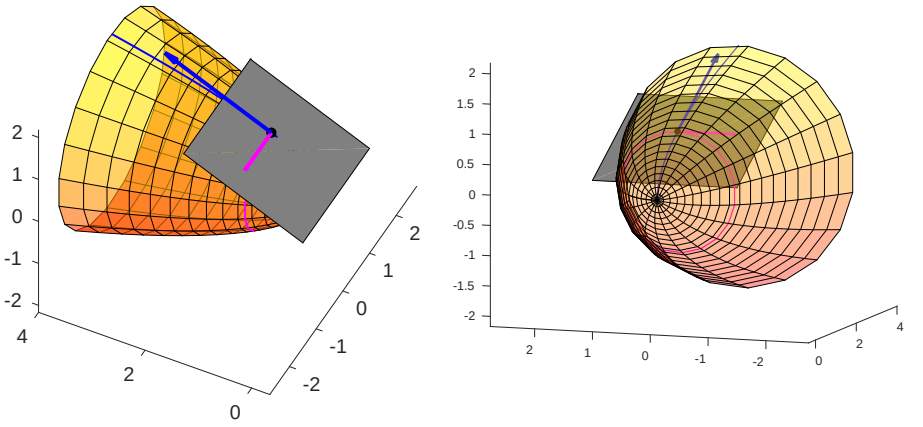

Figura 3.5. *Piano tangente ad un paraboloide*

3.3 ▪ Algoritmi di calcolo

All'interno del toolbox sono contenute funzioni per valutare curve e superfici. Come si sa esistono più modi per effettuare tale valutazione. Se consideriamo ad esempio le curve di Bézier è possibile sperimentare due di questi metodi attraverso le function del toolbox `anmglib_4.1`. Il primo consiste nella valutazione della curva al variare del parametro t nella base di Bernstein, moltiplicato poi per i corrispondenti punti di controllo, eseguendo la combinazione lineare. Il primo step è eseguito dalla function `bs=bernst(g,x)`, che calcola le funzioni base di Bernstein di grado `g` nei punti `x` in `[0 1]`, il passo successivo è moltiplicare i valori ottenuti per i punti di controllo, ricavando così in `x` e `y` le coordinate dei punti valutati.

```
1 %Genero np punti tra a e b
2 mesh = linspace(bezier.ab(1),bezier.ab(2),np);
3 %Valutzione della curva
4 bas_bernst = bernst(bezier.deg,mesh);
```

```
qx=bezier.cp(:,1);
qy=bezier.cp(:,2);
x=bas_bernst*qx;
y=bas_bernst*qy;
%Chiamata di disegno
mesh_curv2_plot(x,y,varargin{:});
```

Listing 3.10. *Algoritmo base per valutare curve di Bézier*

Il secondo metodo consiste nell'applicare l'algoritmo di de Casteljau. Esso consiste nel definire la curva attraverso interpolazioni lineari successive ed è usato all'interno di tutte le funzioni dove è necessario valutare curve di Bézier. L'algoritmo è implementato all'interno della function `Px=decast_val(bezier,t)` può calcolare il valore di una curva nD nella base di Bernstein in [0,1] definita nella struttura `bezier` passata come input, nei punti t. E' possibile inoltre, sempre grazie a questo algoritmo, mediante la function `Px=decast_valder(bezier,k,t)`, calcolare le derivate fino all'ordine `k<=bezier.deg`.

3.4 ▪ Operazioni su file di tipo database

Fino ad ora quasi tutti gli elementi geometrici che abbiamo rappresentato sono stati definiti tramite strutture e variabili, passate poi come argomenti alle function appropriate. E' tuttavia possibile definire curve, superfici e solidi, attraverso un particolare formato di file, non MATLAB, con estensione *.db*, come fatto nella sezione 3.1.3. I file con questo formato sono di tipo database e immagazzinano informazioni in righe e colonne di tabelle. Possiamo sfruttare questo tipo di file per memorizzare i dati necessari ad effettuare calcoli e disegnare i costrutti geometrici di nostra scelta. Vediamo un esempio di file *.db* nel listato 3.11. In questo caso vogliamo definire una curva spline 2D di grado 2, con 5 punti di controllo e 8 nodi.

```
FILENAME: c2_spline.db
DEGREE
2
N.C.P.
5
N.KNOTS
8
COORD.C.P.(X,Y)
1.0  0.0
1.0  1.0
1.5  2.0
2.0  2.0
2.0  0.0
KNOTS
0.0
0.0
0.0
0.3
0.6
1.0
```

```
1.0
1.0
```

Listing 3.11. *Esempio di file db*

Dato un file *.db* definito in modo corretto, possiamo estrarre tutte le informazioni che necessitiamo, memorizzarle in variabili o strutture e utilizzarle come siamo soliti, cioè dandole in input a function di disegno/calcolo. Possiamo inoltre fare anche l'operazione inversa, cioè salvare all'interno di un file *.db* le informazioni descritte all'interno di strutture `bezier, spline, nurbs`. Le function che permettono questo tipo di operazioni sono descritte nello script 3.12.

```
%Definizion dei campi della struttura spline
open_figure(1);
splineQ.deg=3;
splineQ.cp=[0,2,0;1,1,1;2,1,1;3,2,2];
splineQ.knot=[0,0,0,0,1,1,1,1];
%Salvataggio della splineQ nel file new_spline
curv2_spline_save('new_spline.db',splineQ);
%Caricamento della spline da file nella struttura splineP
splineP=curv2_spline_load('spline_file.db');
```

Listing 3.12. *Operazioni su file db*

E' possibile quindi creare manualmente questo tipo di file, cosa che può risultare utile in quanto permette di condividere la stessa curva o superficie tra più utenti. La peculiarità di questo tipo di file non è però questa, infatti, oltre alla creazione manuale è possibile generare questo tipo di file in modo "automatico". Questo è possibile grazie alle function di tipo `save`.

3.5 ▪ Rappresentazione grafica interattiva

Grazie alla versatilità di MATLAB è possibile creare alcune semplici applicazioni e animazioni grafiche, costruite con l'obiettivo di rendere l'insegnamento più interattivo e stimolante, oltre che intuitivo.

3.5.1 ▪ Un' applicazione MATLAB

Approfondiamo ora la possibilità, menzionata nella sezione precedente, di generare automaticamente file *.db* che contengano informazioni su curve e superfici. All'interno del toolbox `anmglib_4.1` vi è un esempio di un'applicazione per la creazione di curve 2D di Bézier: attraverso una semplice interfaccia, composta da una finestra grafica e alcuni pulsanti, si possono disegnare i punti di controllo (tramite click) e fare il plot della curva così definita. Oltre a ciò vi è la possibilità di aggiungere, dopo la creazione della curva, altri control-point, generando così una curva di Bézier a tratti. Attraverso i pulsanti inoltre si può scegliere di disegnare la tangente o la normale in un punto selezionato della curva. Posso quindi creare curve 2D di Bézier, di Bézier a tratti e anche di Bézier a tratti multi-degree, per queste ultime basterà aggiungere un tratto di curva con grado diverso da quello precedente. Nella Figura 3.6 vediamo una curva di Bézier a tratti multi-degree, essa è composta da 2 tratti rispettivamente di grado 3 e 2. Utilizzando questa applicazione è possibile dunque studiare "dal vivo" i com-

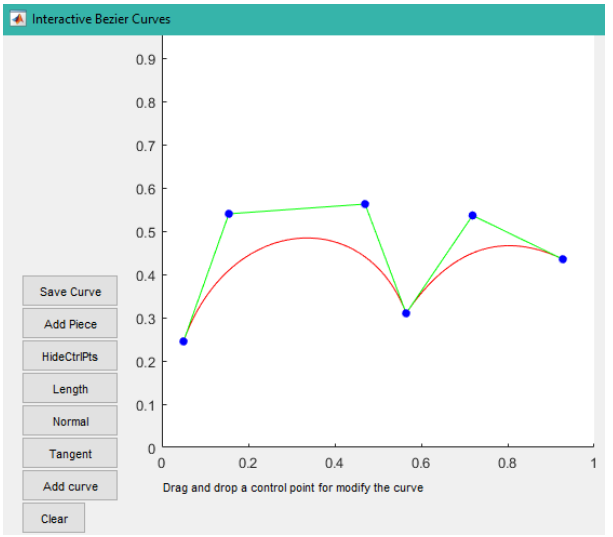

Figura 3.6. *Curva creata attraverso l'applicazione ppbez*

portamenti di una curva 2D di Bézier al variare dei punti di controllo che la definiscono. La curva risultante, oltre che essere modificabile interattivamente, può essere salvata su file. Dal disegno verranno estratte le informazioni che la definiscono, quindi queste saranno trascritte come dati in un file di estensione *.db*. A seconda di come vengono settati i punti di controllo si avranno file che descrivono tipi differenti di curva, distinguibili da numero e definizione dei campi. Questa applicazione è sicuramente di grande utilità, infatti offre agli studenti uno strumento interattivo che funge da tramite tra la definizione formale di una curva e il suo disegno.

3.5.2 ▪ Un'animazione per il disegno

Un bell'esempio di come sia possibile rendere l'uso del toolbox meno "statico" è rappresentato dalla function MATLAB `movie(M,n)`. Essa riproduce `n` volte l'animazione definita dalla matrice `M`, le cui colonne sono frame. Consideriamo l'esempio dello script 3.13: si tratta di uno esempio per rendere animato il plotting del vettore tangente, in 10 punti, di una curva di Bézier $2D$.

```
open_figure(1);
np=10;
bezier=curv2_bezier_load('my_bez.db');
for i=1:np
  cla
  axis([-0.2,1.5,-0.8,1.5]);
  axis_plot(1);
  curv2_bezier_plot(bezier,np,'b');
  point_plot(bezier.cp,'k--o',1,'k','k',4);
```

```
10    curv2_bezier_tan_plot(bezier,i,'r-',1,'k','r',5);
11    %cattura frame in una struttura di np matrici
12    m(:,i)=getframe;
13  end
14  %visualizza l'animazione 3 volte
15  movie(m,3);
```

Listing 3.13. *Esempio di animazione tramite movie*

L'animazione è molto semplice, ma fa capire come con qualche linea di codice sia possibile rendere più interessanti e comprensibili costrutti matematici e geometrici che altrimenti risulterebbero tediosi.

3.6 ▪ Funzioni come curve

Fino ad ora abbiamo visto come definire curve in forma parametrica attraverso apposite strutture. Utilizzando le function del toolbox `anmglib_4.1` vogliamo rappresentare e disegnare classiche funzioni scalari. Una funzione matematica può essere vista come una particolare curva in forma parametrica. Sia data la funzione $y = f(x)$, con $x \in [a, b]$, allora considerando $x = t$ e $y = f(t)$ possiamo definire la curva come:

$$C(t) = \begin{pmatrix} t \\ f(t) \end{pmatrix} \tag{3.1}$$

con $t \in [a, b]$. Abbiamo dunque trovato un modo per rappresentare il grafico di qualsiasi funzione matematica utilizzando il toolbox `anmglib_4.1`. Come esempio vediamo la funzione di Runge, funzione nota come test di interpolazione polinomiale che su punti equispaziati all'aumentare del grado mostra l'inflessibilità della classe dei polinomi evidenziata dalle grosse oscillazioni presenti agli estremi dell'intervallo di interpolazione. Consideriamo dunque la funzione:

$$y = \frac{1}{1 + 25x^2} \quad x \in [-1, 1] \tag{3.2}$$

La definiamo quindi nel modo spiegato precedentemente (Vedi lo script 3.14).

```
1  function [x,y]=runge(t)
2  %funzione di Runge
3  x = t;
4  y = 1./(1+25.*t.^2);
5  return
```

Listing 3.14. *Definizione della funzione di Runge*

Per disegnarla effettueremo una chiamata alla `[x,y]=curv2_plot(curvname,a,b,np,varargin)` nel modo indicato nello script 3.15.

```
1  open_figure(1);
2  axis_plot(1)
3  a=-1;
4  b=1;
5  curv2_plot('runge',a,b,100,'b-',1);
```

Listing 3.15. *Chiamata di disegno della funzione di Runge*

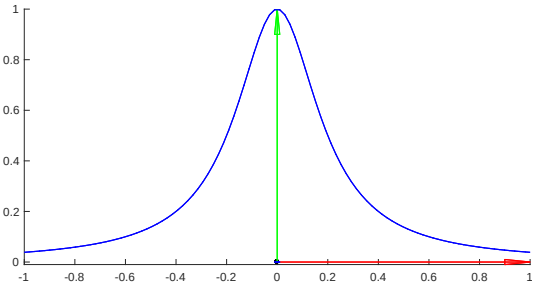

Figura 3.7. *Grafico della funzione di Runge*

Valutata in 100 punti, il grafico risulta quello in Figura 3.7. Questo "nuovo" modo di vedere una funzione come curva parametrica, può essere sfruttato anche per rappresentare polinomi, ad esempio quelli nella base di Bernstein. Infatti se consideriamo la funzione polinomiale

$$p(x) = \sum_{i=0}^{n} c_i B_i^n(x) \quad x \in [0,1]$$

allora l'espressione parametrica della curva sarà:

$$C(t) = \begin{pmatrix} t \\ \sum_{i=0}^{n} c_i B_i^n(t) \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{n} \frac{i}{n} B_i^n(t) \\ \sum_{i=0}^{n} c_i B_i^n(t) \end{pmatrix} \tag{3.3}$$

infatti è possibile dimostrare che vale $\sum_{i=0}^{n} \frac{i}{n} B_i^n(t) = t$ per $t \in [0,1]$.

Conoscendo l'espressione della base di Bernstein di grado n, possiamo rappresentare la curva $C(t)$ e quindi la funzione polinomiale al variare di $t \in [1,0]$.

Capitolo 4

Esercitazioni per lo studente

Vediamo in quest'appendice una diversa chiave di lettura del toolbox. Si sono esaminati, nel capitolo 3 i possibili utilizzi del toolbox, da quelli basici ad altri un po' più avanzati. In questi, per i vari temi trattati, il procedimento è stato pressoché lo stesso:

1. Si è stabilito che cosa si aveva intenzione di disegnare e/o approfondire.
2. Si sono scelte le function appropriate e si è scritto il codice MATLAB appropriato.
3. Si è verificato graficamente se il risultato fosse concorde con le nostre aspettative.

Un approccio interessante sarebbe invece l'inverso di quello appena discusso: partendo da una rappresentazione grafica ed eventualmente qualche altra informazione sul disegno, arrivare a scrivere il codice che l'ha generata. Questa modalità di utilizzo del toolbox è sicuramente molto utile al fine dell'apprendimento. Infatti, se non si sono fatti propri i concetti presentati nella teoria, non sarà possibile risolvere un esercizio "al contrario". Un possibile testo di esercizio potrebbe essere il seguente:

4.1 ▪ Esercizi guidati

Esercizio 1 *Utilizzando curve di Bézier di interpolazione e curve a tratti di Bézier riprodurre il disegno di fig. 4.1. Vengono assegnate due sequenze di punti 2D da interpolare con curve di Bézier e che rappresentano due delle quattro parti curve del disegno:*

```
>primo tratto
218.2 385.4
206.3 445.0
238.9 520.4
285.8 538.8
330.5 515.6
354.8 453.8
348.7 403.68
```

```
>secondo tratto
348.7 403.68
443.0 412.2
528.4 349.1
504.0 255.8
375.0 220.5
321.4 241
```

si sfrutti poi la simmetria del disegno. Per riprodurre il disegno si devono fare una serie di passi e per la precisione: interpolare il primo set di punti con una curva di Bézier di grado 6, quindi interpolare il secondo set con una curva di grado 5, infine per unire le due curve in un'unica curva a tratti di Bézier sarà necessario uniformare prima i gradi delle curve elevando il grado della seconda in quanto una curva a tratti di Bézier deve avere ogni tratto dello stesso grado. Per sfruttare la simmetria del disegno è sufficiente applicare una riflessione della curva appena detreminata rispetto alla retta passante per i suoi estremi. Una delle due antenne gialle può essere modellata con un triangolo lungo e stretto o meglio come una lineare a tratti composta da tre tratti; la seconda antenna può essere ottenuta per rotazione della prima. Il seguente codice implementa quanto detto utilizzando alcune funzioni della libreria `anmglib_4.1`.

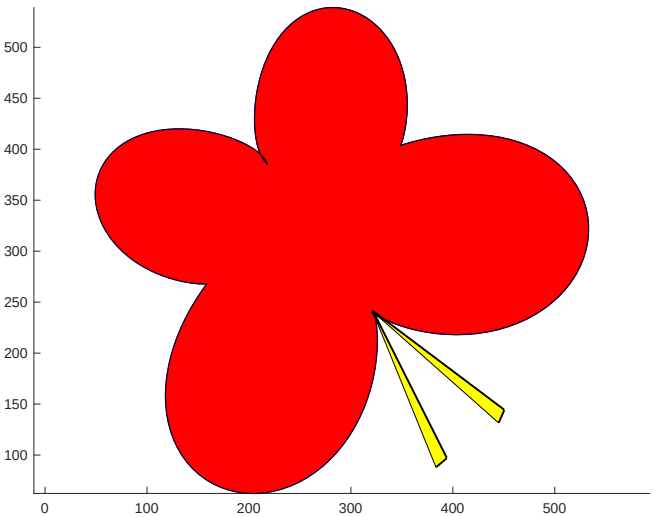

Figura 4.1. *Disegno di una farfalla stilizzata*

Listing 4.1. *script per riprodurre il disegno di Figura 4.1. captionpos*

```
1 %script per riprodurre il disegno di fig.
2 clear all
3 close all
4
5 col=['r','g','b','k'];
6 open_figure(1);
7
```

```
8  %primo set di punti da interpolare
9  Q1=[218.2 385.4
10     206.3 445.0
11     238.9 520.4
12     285.8 538.8
13     330.5 515.6
14     354.8 453.8
15     348.7 403.68];
16 %secondo set di punti da interpolare
17 Q2=[348.7 403.68
18     443.0 412.2
19     528.4 349.1
20     504.0 255.8
21     375.0 220.5
22     321.4 241];
23
24 %disegno punti di interpolazione
25 % point_plot(Q1,'ko',1,'k');
26 % point_plot(Q2,'ko',1,'k');
27
28 %intervallo parametrico di definizione
29 a=0;
30 b=1;
31
32 %numero punti di plotting
33 np=60;
34
35 %scelta dei parametri per i punti di interpolazione (
       param = 0, 1 o 2)
36 param=1;
37 %chiamata a funzione di interpolazione della libreria
       anmglib_4.1
38 bP1=curv2_bezier_interp(Q1,a,b,param);
39 bP2=curv2_bezier_interp(Q2,a,b,param);
40
41 %uniformiamo i gradi per definire una curva di BÃ©zier a
       tratti
42 [bP3.cp(:,1),bP3.cp(:,2)]=gc_pol_de2d(bP2.deg,bP2.cp(:,1)
       ,bP2.cp(:,2));
43 bP3.deg=bP2.deg+1;
44 bP3.ab=bP2.ab;
45
46 %join delle due curve di BÃ©zier
47 ppbP=curv2_ppbezier_join(bP1,bP3,1.0e-4);
48 curv2_ppbezier_plot(ppbP,np,'r-');
49 % point_plot(ppbP.cp,'bo-')
50
51 %simmetrica della curva a tratti rispetto alla retta che
52 %passa per gli estremi
53 [ppbQ,T,R]=align_curve(ppbP);
```

```
54 ppbQ.cp(:,2)=-ppbQ.cp(:,2);
55 Minv=inv(R*T);
56 ppbQ.cp=point_trans(ppbQ.cp,Minv);
57 % curv2_ppbezier_plot(ppbQ,np,'b-');
58 % point_plot(ppbQ.cp(1,:),'bo-')
59
60 %join delle due curve di BÃ©zier a tratti
61 ppbR=curv2_ppbezier_join(ppbP,ppbQ,1.0e-4);
62 xy=curv2_ppbezier_plot(ppbR,np,'b-');
63 % point_plot(ppbR.cp(1,:),'bo-')
64 fill(xy(:,1),xy(:,2),'r');
65
66 %definiamo una delle antenne direttamente come
67 %punti di controllo di una lineare a tratti
68 bpS.deg=1;
69 bpS.ab=[0,1,2];
70 bpS.cp=[321.3  241;
71         450  144.2;
72         444.6  131.8;
73         321.3  241];
74 xy=curv2_ppbezier_plot(bpS,2,'k-',2);
75 % point_plot(bpS.cp(1,:),'bo-')
76 fill(xy(:,1),xy(:,2),'y');
77 % curv2_ppbezier_plot(bpS,2,'k-',2);
78
79 %ruotiamo la curva intorno al primo punto di controllo in
       senso orario
80 C=[321.3,241];
81 T=get_mat_trasl(-C);
82 alfa=-0.46;
83 R=get_mat2_rot(alfa);
84 Tinv=get_mat_trasl(C);
85 M=Tinv*R*T;
86 bpT=bpS;
87 bpT.cp=point_trans(bpS.cp,M);
88 xy=curv2_ppbezier_plot(bpT,2,'k-',2);
89 % point_plot(ppbT.cp(1,:),'bo-')
90 fill(xy(:,1),xy(:,2),'y');
91 % curv2_ppbezier_plot(bpT,3,'k-',2);
92
93
94 function [ppbezQ,T,R]=align_curve(ppbezP)
95  ncp=length(ppbezP.cp(:,1));
96  ppbezQ=ppbezP;
97  T=get_mat_trasl(-ppbezP.cp(1,:));
98  tol=0.01;
99  alfa = -atan2(ppbezP.cp(ncp,2) - ppbezP.cp(1,2), ppbezP.
       cp(ncp,1) - ppbezP.cp(1,1));
100 R=get_mat2_rot(alfa);
101 M=R*T;
```

```
102   ppbezQ.cp=point_trans(ppbezP.cp,M);
103 end
```

Esercizio 2 *Data la curva 2D NURBS "aperta" (curva blu in Figura 4.2) di grado 3, definita nell'intervallo* $[0,3]$*, con nodi* $[0,0,0,0,1,2,3,3,3,3]$*, pesi* $[1,1,5,5,1,1]$ *e punti di controllo* $[-1,2;1,0;1,-1;-1,-1;-1,0;1,2]$*, realizzare uno script che disegni la curva NURBS "chiusa" e periodica (in verde in fig. 4.2), definita dagli stessi punti di controllo.*

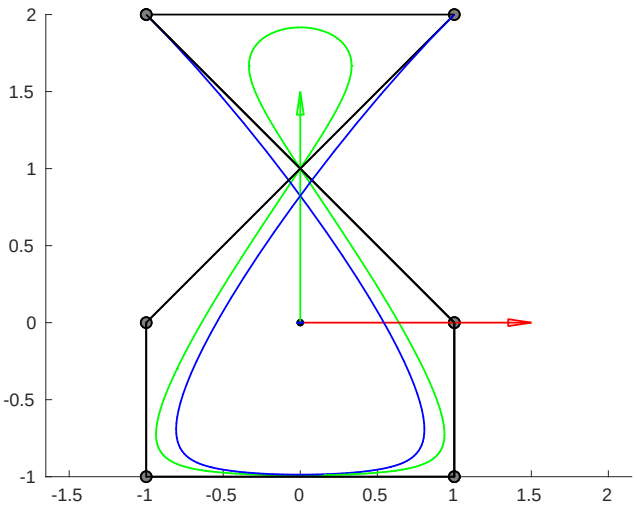

Figura 4.2. *Curva NURBS aperta (blu) e curva NURBS chiusa periodica (verde)*

Vediamo un possibile procedimento risolutivo per questo esercizio. Per prima cosa, perché la curva sia chiusa è necessario che i primi n punti di controllo coincidano con gli ultimi n, dove n è il grado della curva. Aggiungiamo quindi tre punti in coda al vettore, uguali ai primi tre.

$$nurbs.cp = [-1,2;1,0;1,-1;-1,-1;-1,0;1,2;-1,2;1,0;1,-1]$$

Avendo aggiunto dei punti di controllo occorre aggiungere anche i relativi pesi, che coincideranno con quelli dei primi tre punti di controllo.

$$nurbs.w = [1,1,5,5,1,1,1,1,5]$$

A questo punto ci torna utile la teoria sulle partizioni nodali delle curve spline. La partizione estesa si costruisce per definire le funzioni B-Spline (e quindi anche le B-Spline razionali per le NURBS). Il vettore dei nodi della NURBS di partenza è una partizione estesa con nodi aggiuntivi coincidenti (curva *aperta*) ed ha una struttura del tipo:

- $t_1 \leq t_2 \leq ... \leq t_{2n+K+2}$ con k numero dei nodi interni.
- $t_{n+1} = a; \quad t_{n+K+2} = b$ con $[a,b]$ intervallo di definizione della curva.

- $t_{n+2} \leq ... \leq t_{n+K+1} \equiv (x_1 = ... = x_1 < ... < x_k = ... = x_k)$ con ogni x_i ripetuto $n - \mu_i$ volte.
- $t_1 = ... = t_n \equiv a$ con a ripetuto alla fine $n + 1$ volte.
- $t_{n+K+3} = ... = t_{2n+K+2} \equiv b$ con b ripetuto alla fine $n + 1$ volte.

Perché la curva sia chiusa si deve definire una partizione nodale estesa *periodica* in cui i primi e gli ultimi n nodi aggiuntivi siano una replica rispettivamente degli ultimi e dei primi n nodi interni ad $[a, b]$. Per semplicità consideriamo nodi equispaziati e per farlo possiamo utilizzare la function `linspace`; ma quanti nodi? avendo $n + k + 1 = 9$ punti di controllo e grado $n = 3$ dovremo definire $n + k + 1 + n + 1$ nodi ossia $nnodi = 9 + 3 + 1 = 13$:

$$nurbs.knot = linspace(0, 3, nnodi);$$

Con queste informazioni, inserite all'interno della struttura adeguata, siamo in grado di disegnare la curva NURBS "chiusa e periodica" come richiesto.

4.2 ▪ Esempi di Esercizi

Esercizio 1 *Riprodurre la curva a tratti di Bézier in figura 4.3 sapendo che il numero di punti di controllo è 25 e che la curva è di grado 3. Per ogni tratto disegnare i vettori tangenti nei punti di raccordo. Esaminare graficamente quali continuità (C^0, G^1 o C^1) si hanno nei punti di raccordo tra i tratti.*

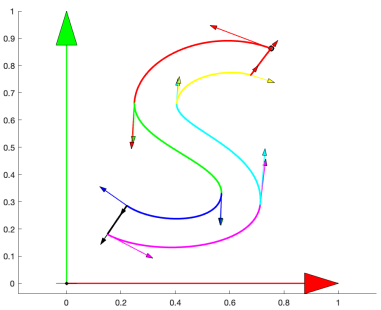

Figura 4.3. *Curva di Bézier a tratti e vettori tangenti nei punti di raccordo*

Esercizio 2 *Definita una curva di Bézier nel piano determinare il suo bounding-box, cioè il piiù piccolo rettangolo con lati paralleli agli assi che lo contiene. Estendere poi quanto fatto al caso di una curva di Bézier a tratti nello spazio.*

Esercizio 3 *Definita una curva di Bézier nel piano determinare e disegnare una curva offset, cioè una curva che disti una costante assegnata dalla curva data, lungo la normale alla curva.*

Esercizio 4 *Determinare quali operazioni sono necessarie per disegnare ed effettuare la trasformazione del cubo in figura 4.4. I vertici opposti del cubo di partenza sono* $[0, 0, 0]$ *e* $[2, 2, 2]$.

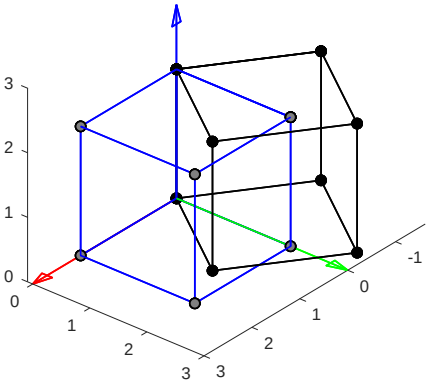

Figura 4.4. *Trasformazione di un cubo*