Treball Fi de Grau – Curs 2025/2026

Animació procedimental per a cossos tous

Autor: **Sau Felguera Martínez**

Tutor: ENRIC COSME LLÓPEZ

Facultat de Matemàtiques

Grau en Matemàtiques

Índex

1 Introducció **1**

2 Splines **3**

2.1 Corbes de Bézier 4

2.1.1 Algoritme de De Casteljau 5

2.1.2 Polinomis de Bernstein 7

2.1.3 Ajust de punts 13

2.2 Corbes de Catmull-Rom 25

2.3 Comparació entre tipus de corbes 35

3 Integració de Verlet **37**

4 Implementació **39**

4.1 Polígon i ratolí 39

4.2 Gravetat 40

4.3 Fregament i ratolí 41

4.4 Molls 42

4.5 Conservació de l'àrea 42

4.6 Integració de Verlet 43

4.7 Revisió i ajust de paràmetres 45

4.8 Color 46

4.9 Bézier 46

4.10 Revisió de Bézier 48

4.11 Treball a Matlab 48

4.12 Canvi de Bézier a Catmull-Rom 50

4.13 Pèndol 51

4.14 Revisió de pèndol 52

4.15 Restriccions del pèndol 53

4.16 Granota 54

5 Conclusions **57**
5.1 Valoració 57
5.2 Treball futur 58
5.2.1 Obstacles amb col·lisions 58
5.2.2 Més d'un cos 58
5.2.3 Cossos diferents 59

Bibliografia **59**

Capítol 1

Introducció

Les matemàtiques s'han utilitzat com a eina en l'art infinitats de voltes, des del dibuix tècnic fins a l'animació per ordinador, dels grafs i fractals a la il·lustració digital. Aquesta relació entre matemàtiques i art ha existit des de l'antiguitat i només s'ha reforçat al llarg de la història, sobretot amb el creixement del mon digital en els últims anys. Tot i això, pareix que malauradament no tothom s'adona de l'existència de la connexió art-matemàtiques, i les dues disciplines se solen observar com incompatibles o difícils de relacionar entre si. Amb aquest treball es busca difondre i donar a conèixer com es pot fer art i matemàtiques al mateix temps, i com es poden aprofitar els punts forts de l'una per a potenciar l'altra.

L'animació procedimental és un procés utilitzat en animació digital mitjançant el qual es poden obtindre animacions molt complexes a través d'algoritmes matemàtics. Aquests algoritmes marquen les regles sota les quals els elements animats s'han de comportar i aquests actuen en conseqüència. Aquest procediment és molt comú als videojocs, sobretot en motors de física, animacions de partícules i comportaments de bandada, ja que ajuden a simplificar la resta del codi. A més, també podem dividir l'animació procedimental en 3 tipus, segons el cos físic que estiguem intentant simular: cossos rígids, cossos tous i fluids. Finalment, es fa ús de les eines d'animació procedimental tant en 2D com en 3D.

Així doncs, en aquest treball hem utilitzat animació procedimental per a simular les físiques d'un cos tou en 2 dimensions. Els cossos tous (o *soft bodies* en anglès) s'utilitzen molt en modelitzacions físiques, ja que permeten simular experiments, realitzar observacions i obtindre resultats de forma virtual. Aquestes simulacions permeten predir els comportaments físics de distints cossos front a diverses forces que actuen contra ells. El benefici principal de l'ús de cossos tous front a cossos rígids en aquest tipus d'aplicacions és el fet que els primers, a diferència dels cossos rígids, admeten ser deformats de moltes formes distintes: es poden estirar, tòrcer, comprimir i molt més.

Per poder fer una animació digital calen diverses eines. Entre aquestes trobem tant un maquinari que puga suportar el processament requerit com un programari que permeta per una banda programar com també mostrar l'animació. Per a aquest treball, el programari

d'animació que s'ha utilitzat és *Processing*, que és al mateix temps un programari de dibuix i un entorn de programació amb el seu propi llenguatge, que està basat en *Java*. Aquest llenguatge està dissenyat per a facilitar la representació gràfica al quadern que habilita el programa. La filosofia de disseny del programa busca la simplicitat, facilitant així la tasca de programació que hi ha darrere de les animacions i dissenys que el programa permet fer.

L'objectiu final d'aquest treball és fer una simulació d'un cos tou en forma de granota. La programació s'ha fet en *Processing* i s'ha utilitzat el vídeo *Simulating soft body animals* [1], publicat a *YouTube* per el canal *Argonaut* en 2025, com a inspiració.

Al llarg d'aquest document estudiarem primerament els conceptes matemàtics que han sigut necessaris per a crear l'animació, començant pels distints tipus de corbes *Splines* al Capítol 2. En aquest punt explorarem primer les corbes de Bézier (2.1) i després les corbes de Catmull-Rom (2.2). Finalment compararem els dos tipus de corba (2.3). Seguint amb els conceptes matemàtics, continuarem per la integració de Verlet al Capítol 3. Més endavant quedarà detallat el procés pel qual s'ha obtés la implementació resultat final (al Capítol 4), aprofundint en cadascun dels passos i implementacions prèvies que s'han realitzat. Finalment, conclourem al Capítol 5 el treball amb les conclusions extretes després de realitzar l'animació, així com diverses formes d'aprofundir sobre la matèria en un futur.

Tot el codi final que s'ha fet en aquest treball es pot trobar a GitHub. A més, també està en els annexos II i III.

Capítol 2

Splines

En línia amb el producte final que volem obtindre, un dels recursos més importants que caldrà utilitzar són interpolacions de punts (en el nostre cas, punts al pla). Aquestes s'utilitzaran per a automatitzar el dibuix del cos tou per pantalla, sense necessitat de considerar constantment el moviment de cada punt del perímetre del cos. Així, s'aconsegueix reduir la càrrega de processament necessària per a que la simulació funcione.

L'eina seleccionada per a realitzar aquestes interpolacions ha sigut els *splines*. La informació general sobre aquests ha sigut obtinguda del document *Numerik I* del professor Christian Lubich [8]. Així doncs, partim de la definició general d'una corba *spline*.

Definició 2.1. [Corba spline] Donat un conjunt finit i ordenat de punts $P = \{x_i \in \mathbb{R}^n | 0 \leq i \leq k\}$, una corba spline que els interpola és una funció real

$$s : [a, b] \longrightarrow \mathbb{R}^n$$

que satisfà les següents propietats:

1. $\forall 0 \leq i \leq k \quad \exists t_i \in [a, b]$ tal que $s(t_i) = x_i$. A més, $t_i < t_j \; \forall \; 0 \leq i < j \leq k$.
2. Les restriccions de la funció $s(t)$ als diferents intervals $[t_{i-1}, t_i] \; \forall \; 1 \leq i \leq k$ són funcions polinòmiques $s_i(t)$ de grau δ_i. A més, el màxim dels graus de les restriccions $\delta_s := \max\{\delta_i | 1 \leq i \leq k\}$ s'anomena ordre de l'spline.

Com que l'animació final que volem fer és en 2 dimensions, per a la resta del treball només ens interessaran les corbes sobre el pla. Així, considerem una definició més concreta.

Definició 2.2 (Corba spline al pla)**.** Donat un conjunt finit de punts $P = \{(x_i, y_i) \in \mathbb{R}^2 | 0 \leq i \leq k\}$, amb $x_0 \leq x_1 \leq \cdots \leq x_k$ una corba spline que els interpola és una funció real

$$s : [x_0, x_n] \longrightarrow \mathbb{R}^2$$

que satisfà les següents propietats:

1. $\forall 0 \leq i \leq k \quad s(x_i) = y_i$.

2. Les restriccions de la funció $s(x)$ als intervals $[x_{i-1}, x_i]$ $\forall$ $1 \leq i \leq k$ són funcions polinòmiques $s_i(x)$ de grau δ_i. A més, el màxim dels graus de les restriccions $\delta_s := \max\{\delta_i | 1 \leq i \leq k\}$ s'anomena ordre de l'spline.

Addicionalment, es poden afegir restriccions per obtindre corbes spline de tipus específics, com és, per exemple, el cas dels splines cúbics. Aquests afigen una restricció sobre els graus dels polinomis que composen la corba i sobre la suavitat de la corba.

Definició 2.3 (Spline cúbic)**.** Donat un conjunt finit de punts $P = \{(x_i, y_i) \in \mathbb{R}^2 | 0 \leq i \leq k\}$, amb $x_0 \leq x_1 \leq \cdots \leq x_k$ una corba spline cúbic que els interpola és una funció real

$$s : [x_0, x_n] \longrightarrow \mathbb{R}^2$$

que satisfà les següents propietats:

1. $\forall 0 \leq i \leq k \quad s(x_i) = y_i$.

2. Les restriccions de la funció $s(x)$ als intervals $[x_{i-1}, x_i]$ $\forall$ $1 \leq i \leq k$ són funcions polinòmiques $s_i(x)$ de grau $\delta_i \leq 3$. L'ordre de l'spline és $\delta_s := \max\{\delta_i | 1 \leq i \leq k\} \leq 3$.

3. $s \in C^2([x_0, x_n])$.

Considerem un exemple.

Exemple 2.1. Considerem els punts $P = \{(0,1), (1,3), (2,1), (3,-1)\}$ i notem que la funció

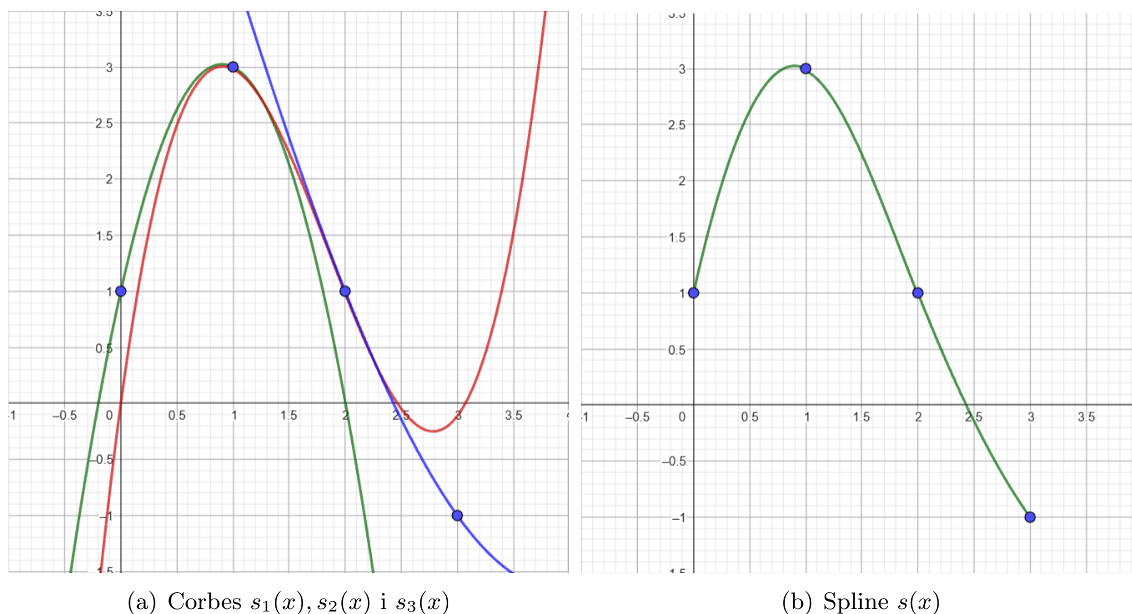
$$s(x) = \begin{cases} s_1(x) = -\frac{132}{53}x^2 + \frac{238}{53}x + 1 & x \in [0,1) \\ s_2(x) = \frac{52}{53}x^3 - \frac{288}{53}x^2 + \frac{394}{53}x & x \in [1,2) \\ s_3(x) = \frac{4}{53}x^3 - \frac{182}{53}x + \frac{385}{53} & x \in [2,3] \end{cases}$$

forma un spline cúbic que interpola els punts. L'spline apareix representat a la Figura 2.1.

Conegut doncs el concepte d'spline, anem a estudiar 2 tipus particulars d'aquests que seran rellevants per a la resta del treball: les corbes de Bézier i les corbes de Catmull-Rom.

2.1 Corbes de Bézier

Per a l'estudi de les corbes de Bézier, s'ha utilitzat com a referència el document *Fundamentos Geométricos del Diseño por Ordenador* de J.V. Beltran i J. Monterde [2].



(a) Corbes $s_1(x), s_2(x)$ i $s_3(x)$

(b) Spline $s(x)$

Figura 2.1: Càlcul gràfic de l'spline que interpola els punts en P en l'Exemple 2.1.

2.1.1 Algoritme de De Casteljau

El primer tipus d'splines concret que estudiarem són les corbes de Bézier. Per començar, estudiarem la seua construcció.

Definició 2.4 (Algoritme de De Casteljau)**.** Donats els punts $P_0, ..., P_n \in \mathbb{R}^2$, aquests defineixen una corba de Bézier mitjançant el següent algoritme recursiu:

$$P_i^0(t) = P_i \quad \forall 0 \leq i \leq n$$

$$P_i^r(t) = (1-t)P_i^{r-1}(t) + tP_{i+1}^{r-1}(t) \quad \forall 1 \leq r \leq n \quad \forall 0 \leq i \leq n-r$$

D'aquesta forma, definim la corba de Bézier $c : [0,1] \longrightarrow \mathbb{R}^2$ com $c(t) = P_0^n(t)$.

D'aquest procediment podem notar com, a diferència de les definicions anteriors, en el cas de les corbes de Bézier calculades amb l'algoritme de De Casteljau, els punts que es consideren no son punts pels quals passa la corba (a excepció del primer i l'últim), sinó que són el que es coneix com a **punts de control**.

Per definició, tot conjunt ordenat de punts de control al pla defineix una única corba de Bézier en aquest. Observem alguns exemples.

Exemple 2.2. Considerem els punts:

$$P_0 = (2,3) \quad P_1 = (-1,8) \quad P_2 = (-5,-3) \quad P_3 = (3,0) \quad P_4 = (-7,-2)$$

i construïm, mitjançant l'algoritme de De Casteljau la corba de Bézier que defineixen.

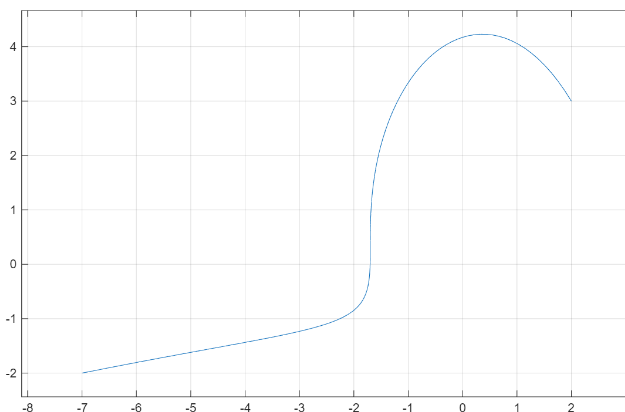


Figura 2.2: Corba de Bézier de l'Exemple 2.2.

	$P_i^0(t)$	$P_i^1(t)$	$P_i^2(t)$
$i=0$	$(2,3)$	$(2-3t, 3+5t)$	$(2-6t-t^2, 3+10t-16t^2)$
$i=1$	$(-1,8)$	$(-1-4t, 8-11t)$	$(-1-8t+12t^2, 8-22t+14t^2)$
$i=2$	$(-5,-3)$	$(-5+8t, -3+3t)$	$(-5+16t-18t^2, -3+6t-5t^2)$
$i=3$	$(3,0)$	$(3-10t, -2t)$	
$i=4$	$(-7,-2)$		

	$P_i^3(t)$
$i=0$	$(2-9t-3t^2+13t^3, 3+15t-48t^2+30t^3)$
$i=1$	$(-1-12t+36t^2-30t^3, 8-33t+42t^2-19t^3)$
$i=2$	
$i=3$	
$i=4$	

$$P_0^4(t) = (2-12t-6t^2+52t^3-43t^4, 3+20t-96t^2+120t^3-49t^4).$$

Concloem doncs que la corba de Bézier de punts de control P_0, P_1, P_2, P_3 i P_4 és

$$c(t) = (2-12t-6t^2+52t^3-43t^4, 3+20t-96t^2+120t^3-49t^4).$$

Notem que $c(0) = (2,3) = P_0$ i $c(1) = (-7,-2) = P_4$. La corba apareix representada a la Figura 2.2.

Exemple 2.3. Considerem els punts:

$$P_0 = (1,7) \quad P_1 = (0,-3) \quad P_2 = (-4,-2) \quad P_3 = (8,0) \quad P_4 = (1,7)$$

i construïm, mitjançant l'algoritme de De Casteljau la corba de Bézier que defineixen.

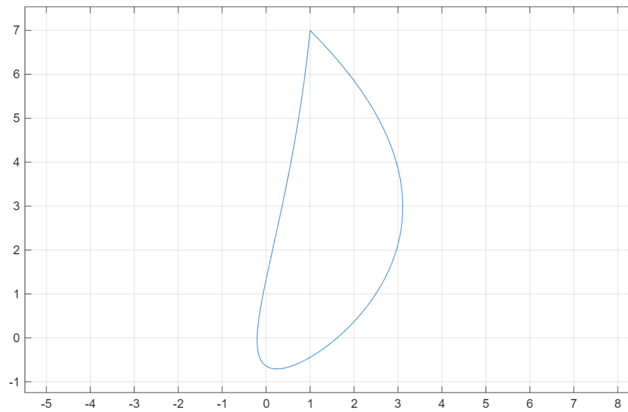


Figura 2.3: Corba de Bézier de l'Exemple 2.3.

	$P_i^0(t)$	$P_i^1(t)$	$P_i^2(t)$
$i=0$	$(1,7)$	$(1-t,7-10t)$	$(1-2t-3t^2,7-20t+11t^2)$
$i=1$	$(0,-3)$	$(-4t,-3+t)$	$(-8t+16t^2,-3+2t+t^2)$
$i=2$	$(-4,-2)$	$(-4+12t,-2+2t)$	$(-4+24t-19t^2,-2+4t+5t^2)$
$i=3$	$(8,0)$	$(8-7t,7t)$	
$i=4$	$(1,7)$		

	$P_i^3(t)$
$i=0$	$(1-3t-9t^2+19t^3,7-30t+33t^2-10t^3)$
$i=1$	$(-12t+48t^2-35t^3,-3+3t+3t^2+4t^3)$
$i=2$	
$i=3$	
$i=4$	

$$P_0^4(t)=(1-4t-18t^2+76t^3-54t^4,7-40t+66t^2-40t^3+14t^4).$$

Concloem doncs que la corba de Bézier de punts de control P_0, P_1, P_2, P_3 i P_4 és

$$c(t)=(1-4t-18t^2+76t^3-54t^4,7-40t+66t^2-40t^3+14t^4).$$

Notem que $c(0)=(1,7)=P_0$ i $c(1)=(1,7)=P_4$.

La corba apareix representada a la Figura 2.3.

2.1.2 Polinomis de Bernstein

L'algoritme de De Casteljau és molt útil per a obtindre corbes de Bézier, però no és l'únic mètode. Estudiem també els polinomis de Bernstein.

Definició 2.5 (Polinomi de Bernstein)**.** Definim l'i-èssim polinomi de Bernstein de grau n de la següent forma:

$$B_i^n(t) = \left(\begin{array}{c} n \\ i \end{array} \right) t^i (1-t)^{n-i} = \begin{cases} \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} & si\ 0 \leq i \leq n \\ 0 & en\ altre\ cas \end{cases}$$

Exemple 2.4. Considerem alguns polinomis de Bernstein:

- $n = 0$

$$B_0^0(t) = 1$$

- $n = 1$

$$B_0^1(t) = 1 - t$$
$$B_1^1(t) = t$$

- $n = 2$

$$B_0^2(t) = 1 - 2t + t^2$$
$$B_1^2(t) = 2t - 2t^2$$
$$B_2^2(t) = t^2$$

- $n = 3$

$$B_0^3(t) = 1 - 3t + 3t^2 - t^3$$
$$B_1^3(t) = 3t - 6t^2 + 3t^3$$
$$B_2^3(t) = 3t^2 - 3t^3$$
$$B_3^3(t) = t^3$$

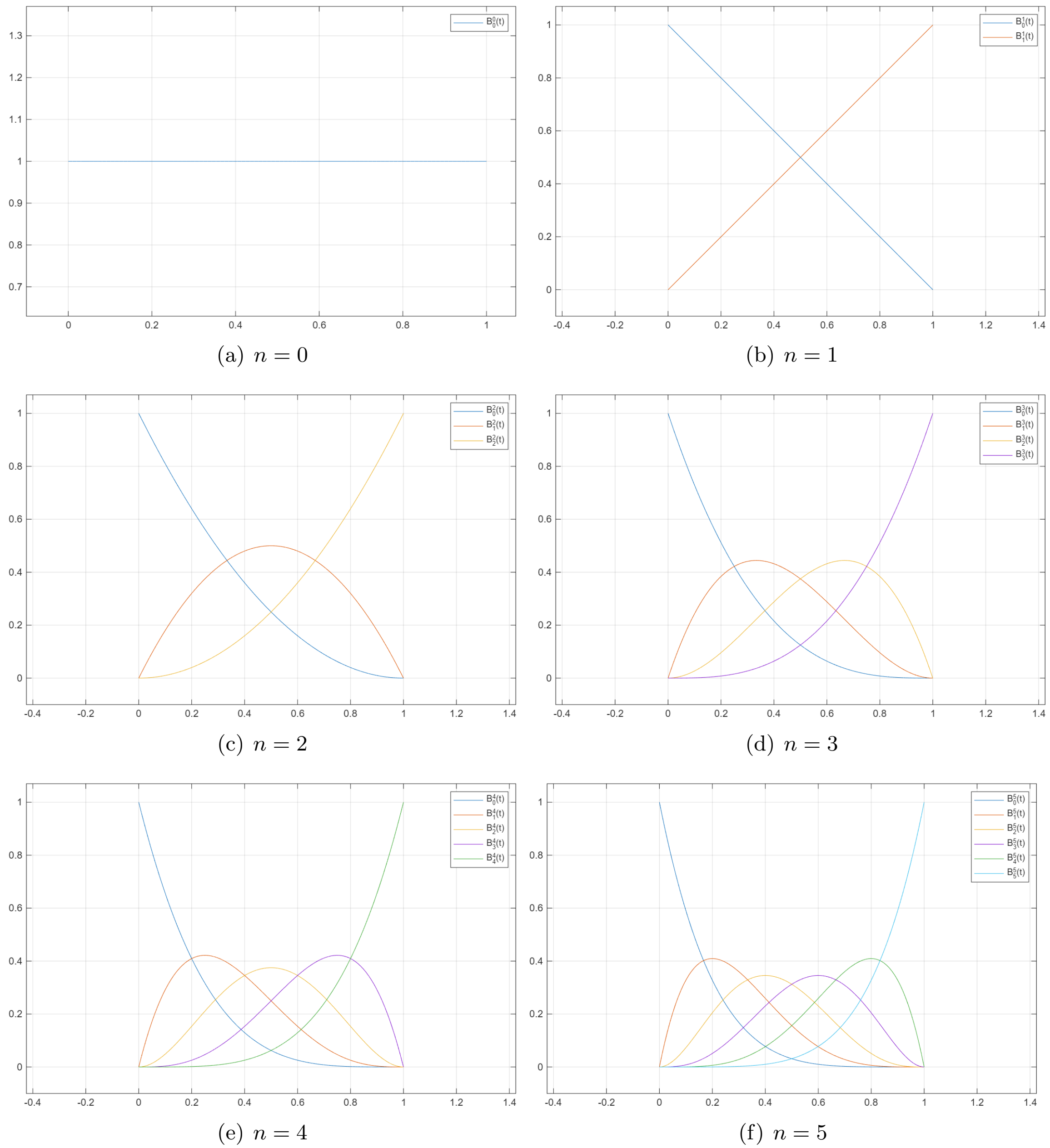
- $n = 4$

$$B_0^4(t) = 1 - 4t + 6t^2 - 4t^3 + t^4$$
$$B_1^4(t) = 4t - 12t^2 + 12t^3 - 4t^4$$
$$B_2^4(t) = 6t^2 - 12t^3 + 6t^4$$
$$B_3^4(t) = 4t^3 - 4t^4$$
$$B_4^4(t) = t^4$$

- $n = 5$

$$B_0^5(t) = 1 - 5t + 10t^2 - 10t^3 + 5t^4 - t^5$$
$$B_1^5(t) = 5t - 20t^2 + 30t^3 - 20t^4 + 5t^5$$
$$B_2^4(t) = 10t^2 - 30t^3 + 30t^4 - 10t^5$$
$$B_3^4(t) = 10t^3 - 20t^4 + 10t^5$$
$$B_4^4(t) = 5t^4 - 5t^5$$
$$B_5^5(t) = t^5$$

Els diferents polinomis apareixen representats a la Figura 2.4.



(a) $n = 0$ (b) $n = 1$

(c) $n = 2$ (d) $n = 3$

(e) $n = 4$ (f) $n = 5$

Figura 2.4: Polinomis de Bernstein de grau n de l'Exemple 2.4.

Per a poder construir les corbes de Bézier a partir dels polinomis de Bernstein, volem trobar alguna relació entre aquests i els polinomis de l'algoritme de De Casteljau.

Lema 2.1. *Els polinomis de Bernstein verifiquen la fórmula recursiva:*

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t).$$

Demostració. Considerem $n, i \in \mathbb{N}$ i fem càlculs:

$$(1-t)B_i^{n-1}(t) = (1-t)\binom{n-1}{i} t^i(1-t)^{n-1-i} = \binom{n-1}{i} t^i(1-t)^{n-i}.$$

$$tB_{i-1}^{n-1}(t) = t\binom{n-1}{i-1} t^{i-1}(1-t)^{n-1-(i-1)} = \binom{n-1}{i-1} t^i(1-t)^{n-i}.$$

Si ho ajuntem, obtenim:

$$\begin{aligned}
(1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) &= \left(\binom{n-1}{i} + \binom{n-1}{i-1}\right) t^i(1-t)^{n-i} \\
&= \left(\frac{(n-1)!}{i!(n-1-i)!} + \frac{(n-1)!}{(i-1)!(n-1-(i-1))!}\right) t^i(1-t)^{n-i} \\
&= \left(\frac{(n-1)!}{i!(n-1-i)!} + \frac{(n-1)!}{(i-1)!(n-i)!}\right) t^i(1-t)^{n-i} \\
&= \left(\frac{(n-1)!(n-i)}{i!(n-i)!} + \frac{(n-1)!i}{i!(n-i)!}\right) t^i(1-t)^{n-i} \\
&= \left(\frac{(n-1)!(n-i) + (n-1)!i}{i!(n-i)!}\right) t^i(1-t)^{n-i} \\
&= \left(\frac{(n-1)!((n-i)+i)}{i!(n-i)!}\right) t^i(1-t)^{n-i} \\
&= \left(\frac{(n-1)!n}{i!(n-i)!}\right) t^i(1-t)^{n-i} \\
&= \left(\frac{n!}{i!(n-i)!}\right) t^i(1-t)^{n-i} \\
&= \binom{n}{i} t^i(1-t)^{n-i} \\
&= B_i^n(t).
\end{aligned}$$

□

La següent proposició ens relaciona els polinomis de De Casteljau amb els polinomis de Bernstein.

Proposició 2.1. *Donats els punts* $\{Q_i\}_{i=0}^n$ *i siguen* $P_l^k(t)$ *els polinomis que s'obtenen amb la construcció de l'algoritme de De Casteljau. Aleshores es compleix que:*

$$P_l^r(t) = \sum_{k=0}^{r} B_k^r(t) Q_{l+k}.$$

Demostració. Fixem un $n \in \mathbb{N}$ i ho demostrem per inducció sobre $0 \leq r \leq n$.

- Cas base $r = 0$.

  Donat $0 \leq l \leq n - r = n$, sabem que $P_l^0(t) = Q_l$. Per altra banda notem que

$$\sum_{k=0}^{r} B_k^r(t) Q_{l+k} = B_0^0(t) Q_{l+0} = 1 \cdot Q_l = Q_l = P_l^0(t).$$

- Cas inductiu.

  Suposem l'enunciat cert $\forall\, 0 \leq h \leq r-1$, ho demostrem per a r. Fixem $0 \leq l \leq n-r$. Per construcció de l'algoritme de De Casteljau, sabem que

$$P_l^r(t) = (1-t) P_l^{r-1}(t) + t P_{l+1}^{r-1}(t).$$

  Ara, per hipòtesi inductiva sabem també que

$$P_l^{r-1}(t) = \sum_{k=0}^{r-1} B_k^{r-1}(t) Q_{l+k}. \qquad P_{l+1}^{r-1}(t) = \sum_{k=0}^{r-1} B_k^{r-1}(t) Q_{l+1+k}.$$

  Així:

$$(1-t) P_l^{r-1}(t) + t P_{l+1}^{r-1}(t) = (1-t) \sum_{k=0}^{r-1} B_k^{r-1}(t) Q_{l+k} + t \sum_{k=0}^{r-1} B_k^{r-1}(t) Q_{l+1+k}.$$

  Notem també que per definició dels polinomis de Bernstein, tenim que $B_{-1}^{r-1}(t) = 0$ i $B_r^{r-1}(t) = 0$. Per tant:

$$\begin{aligned}
(1-t) \sum_{k=0}^{r-1} B_k^{r-1}(t) Q_{l+k} &+ t \sum_{k=0}^{r-1} B_k^{r-1}(t) Q_{l+1+k} \\
&= (1-t) \sum_{k=0}^{r} B_k^{r-1}(t) Q_{l+k} + t \sum_{k=0}^{r} B_{k-1}^{r-1}(t) Q_{l+k} \\
&= \sum_{k=0}^{r} \left((1-t) B_k^{r-1}(t) + t B_{k-1}^{r-1}(t) \right) Q_{l+k} \\
&= \sum_{k=0}^{r} B_k^r(t) Q_{l+k}.
\end{aligned}$$

  Aquest últim pas segueix del Lema 2.1. Finalment, si ajuntem totes les igualtats, obtenim:

$$P_l^r(t) = \sum_{k=0}^{r} B_k^r(t) Q_{l+k}. \qquad \square$$

Definició 2.6 (Corba de Bézier mitjançant polinomis de Bernstein)**.** Seguint sobre la proposició anterior, notem que donats els punts $\{Q_i\}_{i=0}^n$, la corba de Bézier que els utilitza com a punts de control és

$$\alpha(t) = P_0^n(t) = \sum_{k=0}^{n} B_k^n(t) Q_k.$$

Revisitem els exemples anteriors amb el nou mètode.

Exemple 2.5. Considerem els punts:

$$P_0 = (2,3) \quad P_1 = (-1,8) \quad P_2 = (-5,-3) \quad P_3 = (3,0) \quad P_4 = (-7,-2)$$

i construïm, mitjançant polinomis de Bernstein la corba de Bézier que defineixen. Sabem que:

$$\begin{aligned}\alpha(t) &= \sum_{k=0}^{4} B_k^4(t) P_k \\ &= B_0^4(t)P_0 + B_1^4(t)P_1 + B_2^4(t)P_2 + B_3^4(t)P_3 + B_4^4(t)P_4.\end{aligned}$$

$$\begin{aligned}B_0^4(t)P_0 &= (1 - 4t + 6t^2 - 4t^3 + t^4) \cdot (2,3) \\ &= (2 - 8t + 12t^2 - 8t^3 + 2t^4, 3 - 12t + 18t^2 - 12t^3 + 3t^4).\end{aligned}$$

$$\begin{aligned}B_1^4(t)P_1 &= (4t - 12t^2 + 12t^3 - 4t^4) \cdot (-1,8) \\ &= (-4t + 12t^2 - 12t^3 + 4t^4, 32t - 96t^2 + 96t^3 - 32t^4).\end{aligned}$$

$$\begin{aligned}B_2^4(t)P_2 &= (6t^2 - 12t^3 + 6t^4) \cdot (-5,-3) \\ &= (-30t^2 + 60t^3 - 30t^4, -18t^2 + 36t^3 - 18t^4).\end{aligned}$$

$$\begin{aligned}B_3^4(t)P_3 &= (4t^3 - 4t^4) \cdot (3,0) \\ &= (12t^3 - 12t^4, 0).\end{aligned}$$

$$\begin{aligned}B_4^4(t)P_4 &= (t^4) \cdot (-7,-2) \\ &= (-7t^4, -2t^4).\end{aligned}$$

$$\begin{aligned}\alpha(t) &= B_0^4(t)P_0 + B_1^4(t)P_1 + B_2^4(t)P_2 + B_3^4(t)P_3 + B_4^4(t)P_4 \\ &= (2 - 12t - 6t^2 + 52t^3 - 43t^4, 3 + 20t - 96t^2 + 120t^3 - 49t^4).\end{aligned}$$

Notem que el resultat és el mateix que ja havíem obtingut a l'Exemple 2.1.

Exemple 2.6. Considerem els punts:

$$P_0 = (1,7) \quad P_1 = (0,-3) \quad P_2 = (-4,-2) \quad P_3 = (8,0) \quad P_4 = (1,7)$$

i construïm, mitjançant polinomis de Bernstein la corba de Bézier que defineixen. Sabem que:

$$\begin{aligned}\alpha(t) &= \sum_{k=0}^{4} B_k^4(t)P_k \\ &= B_0^4(t)P_0 + B_1^4(t)P_1 + B_2^4(t)P_2 + B_3^4(t)P_3 + B_4^4(t)P_4.\end{aligned}$$

$$\begin{aligned}B_0^4(t)P_0 &= (1-4t+6t^2-4t^3+t^4)\cdot(1,7) \\ &= (1-4t+6t^2-4t^3+t^4, 7-28t+42t^2-28t^3+7t^4).\end{aligned}$$

$$\begin{aligned}B_1^4(t)P_1 &= (4t-12t^2+12t^3-4t^4)\cdot(0,-3) \\ &= (0, -12t+36t^2-36t^3+12t^4).\end{aligned}$$

$$\begin{aligned}B_2^4(t)P_2 &= (6t^2-12t^3+6t^4)\cdot(-4,-2) \\ &= (-24t^2+48t^3-24t^4, -12t^2+24t^3-12t^4).\end{aligned}$$

$$\begin{aligned}B_3^4(t)P_3 &= (4t^3-4t^4)\cdot(8,0) \\ &= (32t^3-32t^4, 0).\end{aligned}$$

$$\begin{aligned}B_4^4(t)P_4 &= (t^4)\cdot(1,7) \\ &= (t^4, 7t^4).\end{aligned}$$

$$\begin{aligned}\alpha(t) &= B_0^4(t)P_0 + B_1^4(t)P_1 + B_2^4(t)P_2 + B_3^4(t)P_3 + B_4^4(t)P_4 \\ &= (1-4t-18t^2+76t^3-54t^4, 7-40t+66t^2-40t^3+14t^4).\end{aligned}$$

Notem que el resultat és el mateix que ja havíem obtingut a l'Exemple 2.2.

2.1.3 Ajust de punts

Fins ara, tot el que hem estudiat sobre les corbes de Bézier ha sigut amb punts de control. Volem prosseguir ara a estudiar les mateixes corbes però amb punts pels quals passen. En altres paraules, donats uns punts $\{Q_i\}_{i=0}^n$, volem trobar uns punts $\{P_j\}_{i=0}^n$ de forma que,

donada $\alpha(t)$ la corba de Bézier definida com:

$$\alpha(t) = \sum_{k=0}^{n} B_k^n(t) P_k.$$

Aleshores aquesta corba travessa les punts $\{Q_i\}_{i=0}^n$ en els instants de temps $\{t_i\}_{i=0}^n$. És a dir, $\alpha(t_i) = Q_i$ amb $0 \leq i \leq n$.

D'aquestes igualtats podem escriure el sistema d'equacions següent:

$$\begin{cases} \alpha(t_0) = \sum_{k=0}^{n} B_k^n(t_0) P_k & = Q_0 \\ \alpha(t_1) = \sum_{k=0}^{n} B_k^n(t_1) P_k & = Q_1 \\ \vdots \qquad\quad \vdots & \quad \vdots \\ \alpha(t_k) = \sum_{k=0}^{n} B_k^n(t_1) P_k & = Q_k \end{cases}$$

Si l'escrivim de forma matricial, obtenim l'expressió $MP = Q$ amb les matrius definides de la següent forma:

$$M = \begin{bmatrix} B_0^n(t_0) & B_1^n(t_0) & \cdots & B_n^n(t_0) \\ B_0^n(t_1) & B_1^n(t_1) & \cdots & B_n^n(t_1) \\ \vdots & \vdots & \ddots & \vdots \\ B_0^n(t_k) & B_1^n(t_k) & \cdots & B_n^n(t_k) \end{bmatrix} \in \mathbb{R}^{(k+1)\times(n+1)}$$

$$Q = \begin{bmatrix} Q_0 \\ Q_1 \\ \vdots \\ Q_k \end{bmatrix} \in \mathbb{R}^{(k+1)\times 1} \qquad P = \begin{bmatrix} P_0 \\ P_1 \\ \vdots \\ P_n \end{bmatrix} \in \mathbb{R}^{(n+1)\times 1}$$

Nota 2.1. Generalment considerarem $n < k$, és a dir, treballarem amb menys punts de control que punts pels quals passa la corba. Assumirem aquesta condició certa per a la resta de la secció.

A l'hora de resoldre el sistema i obtindre el vector de punts de control P sorgeix un problema. Com que M en general no és una matriu quadrada, no existeix la seua inversa. Per a evitar aquest problema, fem us de la matriu pseudoinversa.

Definició 2.7 (Matriu inversa de Moore-Penrose)**.** Donada una matriu real $A \in \mathbb{R}^{n\times m}$, la seua matriu inversa de Moore-Penrose (també coneguda com pseudoinversa o inversa generalitzada) és la única matriu A^+ que satisfà les següents propietats:

1. $AA^+A = A$;
2. $A^+AA^+ = A^+$;
3. $(AA^+)^* = AA^+$;
4. $(A^+A)^* = A^+A$.

On $B^* = \overline{B}^T$ representa la conjugada transposada de la matriu B. En cas que el rang de la matriu A és màxim, es té que $A^+ = (A^TA)^{-1}A^T$.

Amb aquestes matrius podem reescriure el sistema matricial de la següent forma:

$$MP = Q.$$

Podem doncs, multiplicar l'expressió per la matriu transposada de M a l'esquerra per obtindre un nou sistema

$$M^TMP = M^TQ.$$

Arribat a aquest punt sorgeix un nou problema, la invertibilitat de la matriu M^TM.

Lema 2.2. *La matriu M, definida a l'inici de la secció, té rang màxim i la matriu M^TM és invertible (utilitzant la matriu M definida anteriorment).*

Demostració. El primer que haurem de comprovar és que la matriu M té rang màxim. Per veure-ho observem primer la construcció de la matriu.

$$M = \begin{bmatrix} B_0^n(t_0) & B_1^n(t_0) & \cdots & B_n^n(t_0) \\ B_0^n(t_1) & B_1^n(t_1) & \cdots & B_n^n(t_1) \\ \vdots & \vdots & \ddots & \vdots \\ B_0^n(t_k) & B_1^n(t_k) & \cdots & B_n^n(t_k) \end{bmatrix}$$

Com que estem assumint $n < k$, volem comprovar que $\text{rang}(M) = n + 1$. Així, anem a veure que les columnes de la matriu són linealment independents.

Per reducció a l'absurd, suposem que no ho son. Aleshores, $\exists \lambda_0, \lambda_1, \cdots, \lambda_n \in \mathbb{R}$ tals que el polinomi de grau $\leq n$ definit com $P(t) = \sum_{i=0}^{n} \lambda_i B_i^n(t)$ s'anul·la als punts $t_0, t_1, \cdots, t_k$ distints dos a dos. Sabem que el polinomi $P(t)$ tindrà com a màxim n arrels reals, però tenim ja k arrels diferents i $n < k$, per tant arribem a contradicció. L'única opció que ens queda és que $P(t) = 0$, és a dir, que P siga el polinomi nul.

D'aquesta conclusió sorgeix un nou problema, com que hem trobat valors $\lambda_i \in \mathbb{R}$ amb $0 \leq i \leq n$ tals que $\lambda_0 B_0^n(t) + \lambda_1 B_1^n + \cdots + \lambda_n B_n^n(t) = 0$, però sabem que $\{B_i^n(t)\}_{i=0}^n$ forma una base de polinomis de grau $\leq n$. Per tant, concloem que $\lambda_0 = \lambda_1 = \cdots = \lambda_n = 0$.

Així doncs, les columnes de M són linealment independents i per tant el seu rang és màxim, $\text{rang}(M) = n + 1$. Ara queda comprovar que M^TM és invertible.

Per reducció a l'absurd suposem que $M^T M$ no és invertible. Aleshores, $\exists x \in \mathbb{R}^{n+1}$ distint del zero, tal que $(M^T M)x = 0$. Aleshores,

$$0 = \langle 0, x \rangle = \langle (M^T M)x, x \rangle = \langle Mx, Mx \rangle = ||Mx||^2.$$

Per tant podem afirmar que $||Mx||^2 = 0$ i en conseqüència es té que $||Mx|| = 0$. Així, concloem que $Mx = 0$, però com sabem que $\text{rang}(M) = n + 1$, que és rang màxim, açò només pot passar si $x = 0$. Entrem en contradicció i per tant podem concloure que $M^T M$ és una matriu invertible. □

Gràcies a aquest resultat, podem continuar arreglant el sistema matricial anterior. Partim doncs del sistema

$$M^T M P = M^T Q.$$

Ara sabem, gràcies al Lema 2.2, que la matriu $M^T M$ és invertible. Aleshores, podem reescriure el sistema per obtindre

$$P = (M^T M)^{-1} M^T Q.$$

Obtenim així una expressió del vector P que busquem.

Nota 2.2. Podem notar que, seguint el Lema 2.2, la matriu M té rang màxim. Per tant, per la definició de la matriu inversa de Moore-Penrose per a matrius amb rang màxim, tenim $M^+ = (M^T M)^{-1} M^T$. Notem doncs que: $P = M^+ Q$.

Amb tot açò, el que quedaria per trobar per a poder calcular els punts de control de la corba de Bézier que busquem seran els valors $t_0, ..., t_k$ tals que $c(t_i) = Q_i$ per a $0 \leq i \leq k$, ja que amb aquests valors triats, obtindrem una expressió numèrica de M i podrem resoldre numèricament el vector $P = (M^T M)^{-1} M^T Q$.

Hi ha diversos criteris que es poden seguir per escollir els t_i.

1. El primer criteri i el més senzill és fixar els t_i com a valors equidistants en l'interval $[0, 1]$, amb $t_i = \frac{i}{k} \in [0, 1]$ per a $0 \leq i \leq k$.

2. El segon criteri augmenta un poc la complexitat dels càlculs, però sense excedir-se. Consisteix a fer les distàncies entre els valors t_i proporcionals a les distàncies euclidianes entre els punts Q_i. Per fer-ho, definim primer:

   $$d_i = \text{dist}(Q_{i-1}, Q_i) \quad \forall\, 1 \leq i \leq k.$$

   Amb açò, podem considerar

   $$t_i = \frac{\displaystyle\sum_{j=1}^{i} d_j}{\displaystyle\sum_{l=1}^{k} d_l}$$

   per a $1 \leq i \leq k$, fixant $t_0 = 0$.

Els dos criteris considerats són només algunes de les opcions més senzilles, però no són els únics criteris que es poden utilitzar.

Finalment, amb els punts de control obtinguts, es podrà construir la corba de Bézier que s'ajusta als punts Q, utilitzant la formula inicial.

$$\alpha(t) = \sum_{k=0}^{n} B_k^n(t) P_k.$$

Nota 2.3. Aquesta corba en general no travessarà els punts $\{Q_i\}_{i=0}^k$, però serà la millor aproximació que es pot obtindre amb les dades inicials escollides.

Proposició 2.2. *Donats els punts $\{Q_i\}_{i=0}^k$, el valor $n \in \mathbb{N}$ i els paràmetres $\{t_i\}_{i=0}^k$, la corba de Bézier obtinguda pel procediment anterior és la que millor s'ajusta als punts $\{Q_i\}_{i=0}^k$. És a dir, és la corba que minimitza l'error mitjà*

$$\sum_{i=0}^{k} ||Q_i - \alpha(t_i)||^2.$$

Demostració. Sabem que la corba de Bézier $\alpha(t)$ depén dels punts de control $\{Q_i\}_{i=0}^n$, per tant anem a considerar l'expressió general obtinguda pels polinomis de Bernstein que depén dels punts de control:

$$\alpha(t)[P_0, \cdots, P_n] = \sum_{i=0}^{n} B_i^n(t) P_i.$$

Per trobar el mínim de l'error mitjà, fem la derivada parcial respecte al punt P_l amb $0 \leq l \leq n$ i la igualem a 0.

$$\begin{aligned}
0 &= \frac{\partial}{\partial P_l} \left(\sum_{i=0}^{k} ||Q_i - \alpha(t_i)[P_0, \cdots, P_n]||^2 \right) \\
&= \frac{\partial}{\partial P_l} \left(\sum_{i=0}^{k} ||Q_i - \sum_{j=0}^{n} B_j^n(t_i) P_j||^2 \right) \\
&= \sum_{i=0}^{k} 2\langle Q_i - \sum_{j=0}^{n} B_j^n(t_i) P_j, -B_l^n(t_i) \rangle \\
&= -2 \sum_{i=0}^{k} \langle Q_i - \sum_{j=0}^{n} B_j^n(t_i) P_j, B_l^n(t_i) \rangle \\
&= -2 \left(\sum_{i=0}^{k} B_l^n(t_i) Q_i - \sum_{i=0}^{k} B_l^n(t_i) \sum_{j=0}^{n} B_j^n(t_i) P_j \right).
\end{aligned}$$

Aleshores:

$$0 = \sum_{i=0}^{k} B_l^n(t_i)Q_i - \sum_{i=0}^{k} B_l^n(t_i) \sum_{j=0}^{n} B_j^n(t_i)P_j.$$

Si ho escrivim en forma matricial obtenim la següent igualtat.

$$0 = \begin{bmatrix} B_l^n(t_0) & \cdots & B_l^n(t_k) \end{bmatrix} \begin{bmatrix} Q_0 \\ \vdots \\ Q_k \end{bmatrix} - \begin{bmatrix} B_l^n(t_0) & \cdots & B_l^n(t_k) \end{bmatrix} \begin{bmatrix} \sum_{j=0}^n B_j^n(t_0)P_j \\ \vdots \\ \sum_{j=0}^n B_j^n(t_k)P_j \end{bmatrix}.$$

Notem a més, que aquesta igualtat volem que es satisfaça per a tot $0 \leq l \leq n$. Així doncs obtenim la següent expressió matricial.

$$0 = \begin{bmatrix} B_0^n(t_0) & \cdots & B_0^n(t_k) \\ \vdots & \ddots & \vdots \\ B_l^n(t_0) & \cdots & B_l^n(t_k) \\ \vdots & \ddots & \vdots \\ B_n^n(t_0) & \cdots & B_n^n(t_k) \end{bmatrix} \begin{bmatrix} Q_0 \\ \vdots \\ Q_k \end{bmatrix} - \begin{bmatrix} B_0^n(t_0) & \cdots & B_0^n(t_k) \\ \vdots & \ddots & \vdots \\ B_l^n(t_0) & \cdots & B_l^n(t_k) \\ \vdots & \ddots & \vdots \\ B_n^n(t_0) & \cdots & B_n^n(t_k) \end{bmatrix} \begin{bmatrix} \sum_{j=0}^n B_j^n(t_0)P_j \\ \vdots \\ \sum_{j=0}^n B_j^n(t_k)P_j \end{bmatrix}.$$

Notem també que:

$$\begin{bmatrix} \sum_{j=0}^n B_j^n(t_0)P_j \\ \vdots \\ \sum_{j=0}^n B_j^n(t_k)P_j \end{bmatrix} = \begin{bmatrix} B_0^n(t_0) & \cdots & B_l^n(t_0) & \cdots & B_n^n(t_0) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ B_0^n(t_k) & \cdots & B_l^n(t_k) & \cdots & B_n^n(t_k) \end{bmatrix} \begin{bmatrix} P_0 \\ \vdots \\ P_n \end{bmatrix}.$$

Recordem ara la construcció de las matrius i vectors M, Q i P. Amb aquestes reescrivim l'equació anterior.

$$0 = M^T Q - M^T M P.$$

A continuació, com pel Lema 2.1 sabem que la matriu $M^T M$ té inversa, podem escriure que el punt crític de l'error mitjà s'assoleix quan

$$P = (M^T M)^{-1} M^T Q.$$

Finalment queda comprovar que aquest punt crític és un mínim. Per veure-ho, comprovarem que tots els valors propis del Hessià són estrictament positius. Primer, caldrà construir el Hessià H, que sabem que és la matriu formada per les segones derivades parcials. Així doncs, les hem de calcular. Considerem $0 \leq l, s \leq n$

$$H_{l,j} = \frac{\partial^2}{\partial P_l \partial P_s} \left(\sum_{i=0}^{k} ||Q_i - \alpha(t_i)[P_0, \cdots, P_n]||^2 \right)$$

$$
\begin{aligned}
&= \frac{\partial^2}{\partial P_l \partial P_s}\left(\sum_{i=0}^{k}||Q_i - \sum_{j=0}^{n} B_j^n(t_i)P_j||^2\right)\\
&= -2\frac{\partial}{\partial P_s}\left(\sum_{i=0}^{k} B_l^n(t_i)Q_i - \sum_{i=0}^{k} B_l^n(t_i)\sum_{j=0}^{n} B_j^n(t_i)P_j\right)\\
&= 2\sum_{i=0}^{k} B_l^n(t_i)B_s^n(t_i).
\end{aligned}
$$

Notem doncs que el Hessià és exactament $H = 2M^TM$. Siga x un vector propi unitari i λ el seu valor propi de H associat. Per tant $\lambda x = Hx = 2(M^TM)x$ aleshores $\frac{1}{2}\lambda x = (M^TM)x$. Amb aquest resultat, comprovem que:

$$
\frac{1}{2}\lambda = \frac{1}{2}\lambda||x||^2 = \langle\frac{1}{2}\lambda x, x\rangle = \langle(M^TM)x, x\rangle = \langle Mx, Mx\rangle = ||Mx||^2.
$$

Per tant $\lambda = 2||Mx||^2$. Pel Lema 2.1 sabem que la matriu M té rang màxim, i per tant, per a tot $v \neq 0$ es té que $Mv \neq 0$. Així, com que x és vector propi unitari, es té que $x \neq 0$ i per tant, $Mx \neq 0$ fet que implica que $||Mx||^2 > 0$ i aleshores tenim

$$
\lambda = 2||Mx||^2 > 0.
$$

Així $\lambda > 0$. Concloem que tots els valors propis del Hessià H són positius i per tant el punt crític que s'assoleix en $P = (M^TM)^{-1}M^TQ$ és un mínim de l'error mitjà. És a dir, per als valors $\{t_i\}_{i=0}^k$ i n donats, la corba de Bézier que millor s'ajusta als punts $\{Q_i\}_{i=0}^k$ és $\alpha(t)[P] = \sum_{i=0}^n B_i^n(t)P_i$ on $P = (M^TM)^{-1}M^TQ$. □

Per a veure ara les diferències entre les corbes que podem obtindre utilitzant un criteri o un altre per escollir els valors dels t_i i valors distints de n, estudiarem un mateix exemple de diverses formes distintes.

Exemple 2.7. Considerem els punts

$$
Q_0 = (0,0) \quad Q_1 = (1,3) \quad Q_2 = (4,2) \quad Q_3 = (8,-2) \quad Q_4 = (1,5)
$$

Per al primer estudi, considerarem $n = 2$ i $t_i = \frac{i}{k}$ amb $0 \leq i \leq k$. Per tant tenim $t_0 = 0,\ t_1 = \frac{1}{4},\ t_2 = \frac{1}{2},\ t_3 = \frac{3}{4},\ t_4 = 1$. Construïm la matriu M:

$$
M = \begin{bmatrix} B_0^2(t_0) & B_1^2(t_0) & B_2^2(t_0)\\ B_0^2(t_1) & B_1^2(t_1) & B_2^2(t_1)\\ B_0^2(t_2) & B_1^2(t_2) & B_2^2(t_2)\\ B_0^2(t_3) & B_1^2(t_3) & B_2^2(t_3)\\ B_0^2(t_4) & B_1^2(t_4) & B_2^2(t_4)\end{bmatrix} = \begin{bmatrix} 1 & 0 & 0\\ 9/16 & 3/8 & 1/16\\ 1/4 & 1/2 & 1/4\\ 1/16 & 3/8 & 9/16\\ 0 & 0 & 1\end{bmatrix}.
$$

Així, obtenim també M^T, M^TM, $(M^TM)^{-1}$ i $M^+ = (M^TM)^{-1}M^T$.

$$M^T = \begin{bmatrix} 1 & 9/16 & 1/4 & 1/16 & 0 \\ 0 & 3/8 & 1/2 & 3/8 & 0 \\ 0 & 1/16 & 1/4 & 9/16 & 1 \end{bmatrix}.$$

$$M^TM = \begin{bmatrix} 177/128 & 23/64 & 17/128 \\ 23/64 & 17/32 & 23/64 \\ 17/128 & 23/64 & 177/128 \end{bmatrix}.$$

$$(M^TM)^{-1} = \begin{bmatrix} 31/35 & -23/35 & 3/35 \\ -23/35 & 97/35 & -23/35 \\ 3/35 & -23/35 & 31/35 \end{bmatrix}.$$

$$M^+ = (M^TM)^{-1}M^T = \begin{bmatrix} 31/35 & 9/35 & -3/35 & -1/7 & 3/35 \\ -23/35 & 22/35 & 37/35 & 22/35 & -23/35 \\ 3/35 & -1/7 & -3/35 & 9/35 & 31/35 \end{bmatrix}.$$

Queda fer el càlcul dels punts de control. Els obtenim de resoldre $P = M^+Q$.

$$\begin{aligned} P &= M^+Q \\ &= \begin{bmatrix} 31/35 & 9/35 & -3/35 & -1/7 & 3/35 \\ -23/35 & 22/35 & 37/35 & 22/35 & -23/35 \\ 3/35 & -1/7 & -3/35 & 9/35 & 31/35 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 3 \\ 4 & 2 \\ 8 & -2 \\ 1 & 5 \end{bmatrix} \\ &= \begin{bmatrix} -8/7 & 46/35 \\ 323/35 & -19/35 \\ 86/35 & 116/35 \end{bmatrix}. \end{aligned}$$

Finalment, amb aquests punts de control ja podem construir la corba de Bézier que millor s'ajusta als punts Q sota les condicions imposades.

$$\begin{aligned} \alpha(t) &= \sum_{k=0}^{4} B_k^2(t)P_k \\ &= B_0^2(t)P_0 + B_1^2(t)P_1 + B_2^2(t)P_2 \\ &= \left(-\frac{8}{7} + \frac{726}{35}t - \frac{600}{35}t^2, \frac{46}{35} - \frac{130}{35}t + \frac{200}{35}t^2\right). \end{aligned}$$

La corba apareix representada a la Figura 2.5.

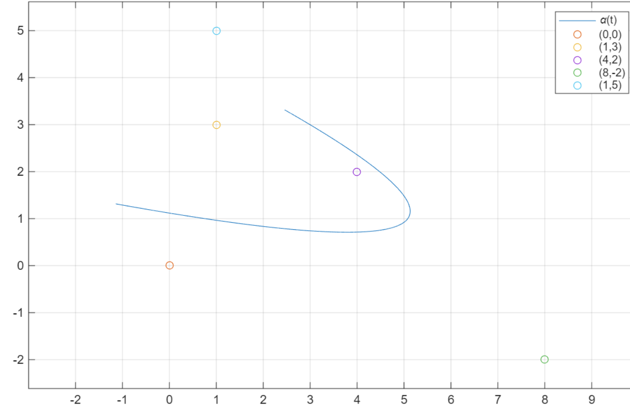


Figura 2.5: Corba de Bézier de l'Exemple 2.7 amb $n = 2$ i $t_i = \frac{i}{k}$.

Exemple 2.8. Considerem els punts

$$Q_0 = (0,0) \quad Q_1 = (1,3) \quad Q_2 = (4,2) \quad Q_3 = (8,-2) \quad Q_4 = (1,5)$$

Per al segon estudi, considerarem $n = 3$ i $t_i = \frac{i}{k}$ amb $0 \leq i \leq k$. Per tant tenim $t_0 = 0,\ t_1 = \frac{1}{4},\ t_2 = \frac{1}{2},\ t_3 = \frac{3}{4},\ t_4 = 1$. Construïm la matriu M:

$$M = \begin{bmatrix} B_0^3(t_0) & B_1^3(t_0) & B_2^3(t_0) & B_3^3(t_0) \\ B_0^3(t_1) & B_1^3(t_1) & B_2^3(t_1) & B_3^3(t_1) \\ B_0^3(t_2) & B_1^3(t_2) & B_2^3(t_2) & B_3^3(t_2) \\ B_0^3(t_3) & B_1^3(t_3) & B_2^3(t_3) & B_3^3(t_3) \\ B_0^3(t_4) & B_1^3(t_4) & B_2^3(t_4) & B_3^3(t_4) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 27/64 & 27/64 & 9/64 & 1/64 \\ 1/8 & 3/8 & 3/8 & 1/8 \\ 1/64 & 9/64 & 27/64 & 27/64 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Així, obtenim també M^T, M^TM, $(M^TM)^{-1}$ i $M^+ = (M^TM)^{-1}M^T$.

$$M^T = \begin{bmatrix} 1 & 27/64 & 1/8 & 1/64 & 0 \\ 0 & 27/64 & 3/8 & 9/64 & 0 \\ 0 & 9/64 & 3/8 & 27/64 & 0 \\ 0 & 1/64 & 1/8 & 27/64 & 1 \end{bmatrix}.$$

$$M^TM = \begin{bmatrix} 2445/2048 & 465/2048 & 231/2048 & 59/2048 \\ 465/2048 & 693/2048 & 531/2048 & 231/2048 \\ 231/2048 & 531/2048 & 693/2048 & 465/2048 \\ 59/2048 & 231/2048 & 465/2048 & 2445/2048 \end{bmatrix}.$$

$$(M^TM)^{-1} = \begin{bmatrix} 69/70 & -629/630 & 281/630 & -1/70 \\ -629/630 & 47389/5670 & -36121/5670 & 281/630 \\ 281/630 & -36121/5670 & 47389/5670 & -629/630 \\ -1/70 & 281/630 & -629/630 & 69/70 \end{bmatrix}.$$

$$M^+ = (M^TM)^{-1}M^T = \begin{bmatrix} 69/70 & 2/35 & -3/35 & 2/35 & -1/70 \\ -629/630 & 698/315 & 71/105 & -422/315 & 281/630 \\ 281/630 & -422/315 & 71/105 & 698/315 & -629/630 \\ -1/70 & 2/35 & -3/35 & 2/35 & 69/70 \end{bmatrix}.$$

Queda fer el càlcul dels punts de control. Els obtenim de resoldre $P = M^+Q$.

$$\begin{aligned} P &= M^+Q \\ &= \begin{bmatrix} 69/70 & 2/35 & -3/35 & 2/35 & -1/70 \\ -629/630 & 698/315 & 71/105 & -422/315 & 281/630 \\ 281/630 & -422/315 & 71/105 & 698/315 & -629/630 \\ -1/70 & 2/35 & -3/35 & 2/35 & 69/70 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 3 \\ 4 & 2 \\ 8 & -2 \\ 1 & 5 \end{bmatrix} \\ &= \begin{bmatrix} 11/70 & -13/70 \\ -3371/630 & 2711/210 \\ 11399/630 & -2539/210 \\ 81/70 & 337/70 \end{bmatrix}. \end{aligned}$$

Finalment, amb aquests punts de control ja podem construir la corba de Bézier que millor s'ajusta als punts Q sota les condicions imposades.

$$\begin{aligned} \alpha(t) &= \sum_{k=0}^{4} B_k^2(t)P_k \\ &= B_0^3(t)P_0 + B_1^3(t)P_1 + B_2^3(t)P_2 + B_3^3(t)P_3 \\ &= \left(\frac{11}{70} - \frac{1636}{105}t + \frac{608}{7}t^2 - \frac{208}{3}t^3, -\frac{13}{70} + \frac{275}{7}t - \frac{800}{7}t^2 + 80t^3\right). \end{aligned}$$

La corba apareix representada a la Figura 2.6.

Exemple 2.9. Considerem els punts

$$Q_0 = (0,0) \quad Q_1 = (1,3) \quad Q_2 = (4,2) \quad Q_3 = (8,-2) \quad Q_4 = (1,5)$$

Per a aquest estudi, considerarem $n = 2$ i t_i proporcionals a les distàncies. Calculem primer les distàncies entre els punts.

$$\begin{aligned} d_1 &= \mathrm{dist}(Q_0, Q_1) = \sqrt{1^2 + 3^2} = \sqrt{10}. \\ d_2 &= \mathrm{dist}(Q_1, Q_2) = \sqrt{(4-1)^2 + (2-3)^2} = \sqrt{9+1} = \sqrt{10}. \\ d_3 &= \mathrm{dist}(Q_2, Q_3) = \sqrt{(8-4)^2 + (-2-2)^2} = \sqrt{16+16} = 4\sqrt{2}. \\ d_4 &= \mathrm{dist}(Q_3, Q_4) = \sqrt{(1-7)^2 + (5-(-2))^2} = \sqrt{36+49} = \sqrt{85}. \\ d_T &= d_1 + d_2 + d_3 + d_4 = 2\sqrt{10} + 4\sqrt{2} + \sqrt{85}. \end{aligned}$$

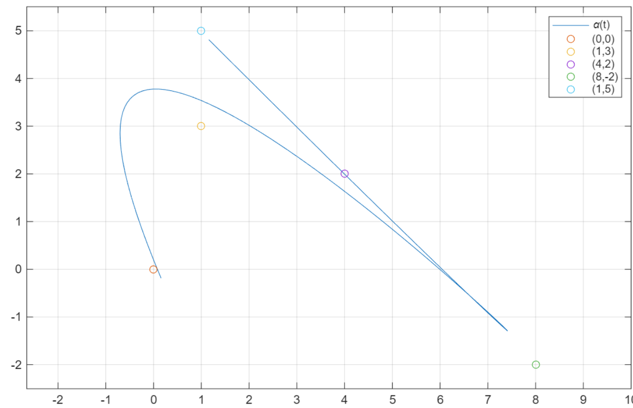


Figura 2.6: Corba de Bézier de l'Exemple 2.8 amb $n = 3$ i $t_i = \frac{i}{k}$.

$$
\begin{aligned}
t_0 &= 0.\\
t_1 &= \frac{\sqrt{10}}{2\sqrt{10}+4\sqrt{2}+\sqrt{85}} = \frac{1540}{4951} + \frac{744\sqrt{5}}{4951} - \frac{65\sqrt{34}}{4951} - \frac{160\sqrt{170}}{4951}.\\
t_2 &= \frac{2\sqrt{10}}{2\sqrt{10}+4\sqrt{2}+\sqrt{85}} = \frac{3080}{4951} + \frac{1488\sqrt{5}}{4951} - \frac{130\sqrt{34}}{4951} - \frac{320\sqrt{170}}{4951}.\\
t_3 &= \frac{2\sqrt{10}+4\sqrt{2}}{2\sqrt{10}+4\sqrt{2}+\sqrt{85}} = \frac{6056}{4951} + \frac{2720\sqrt{5}}{4951} - \frac{770\sqrt{34}}{4951} - \frac{372\sqrt{170}}{4951}.\\
t_4 &= 1.
\end{aligned}
$$

Fem ara els càlculs per construir la matriu M:

$$
M = \begin{bmatrix} B_0^2(t_0) & B_1^2(t_0) & B_2^2(t_0) \\ B_0^2(t_1) & B_1^2(t_1) & B_2^2(t_1) \\ B_0^2(t_2) & B_1^2(t_2) & B_2^2(t_2) \\ B_0^2(t_3) & B_1^2(t_3) & B_2^2(t_3) \\ B_0^2(t_4) & B_1^2(t_4) & B_2^2(t_4) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ B_0^2(t_1) & B_1^2(t_1) & B_2^2(t_1) \\ B_0^2(t_2) & B_1^2(t_2) & B_2^2(t_2) \\ B_0^2(t_3) & B_1^2(t_3) & B_2^2(t_3) \\ 0 & 0 & 1 \end{bmatrix}.
$$

Treballem en `Matlab` per obtindre la gràfica pertinent a aquestes dades.

La corba apareix representada a la Figura 2.7.

Exemple 2.10. Considerem els punts

$$Q_0 = (0,0) \quad Q_1 = (1,3) \quad Q_2 = (4,2) \quad Q_3 = (8,-2) \quad Q_4 = (1,5)$$

Per a l'últim estudi, considerarem $n = 3$ i t_i proporcionals a les distàncies. Aprofitem els

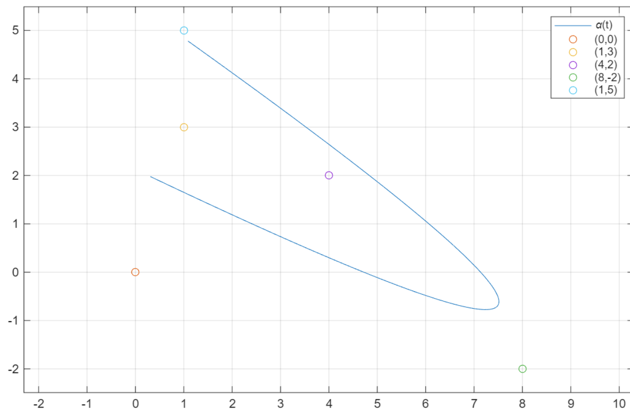


Figura 2.7: Corba de Bézier de l'Exemple 2.9 amb $n = 3$ i t_i proporcionals a les distàncies.

càlculs anteriors per obtindre:

$$t_0 = 0.$$
$$t_1 = \frac{\sqrt{10}}{2\sqrt{10}+4\sqrt{2}+\sqrt{85}} = \frac{1540}{4951} + \frac{744\sqrt{5}}{4951} - \frac{65\sqrt{34}}{4951} - \frac{160\sqrt{170}}{4951}.$$
$$t_2 = \frac{2\sqrt{10}}{2\sqrt{10}+4\sqrt{2}+\sqrt{85}} = \frac{3080}{4951} + \frac{1488\sqrt{5}}{4951} - \frac{130\sqrt{34}}{4951} - \frac{320\sqrt{170}}{4951}.$$
$$t_3 = \frac{2\sqrt{10}+4\sqrt{2}}{2\sqrt{10}+4\sqrt{2}+\sqrt{85}} = \frac{6056}{4951} + \frac{2720\sqrt{5}}{4951} - \frac{770\sqrt{34}}{4951} - \frac{372\sqrt{170}}{4951}.$$
$$t_4 = 1.$$

Construïm la matriu M:

$$M = \begin{bmatrix} B_0^3(t_0) & B_1^3(t_0) & B_2^3(t_0) & B_3^3(t_0) \\ B_0^3(t_1) & B_1^3(t_1) & B_2^3(t_1) & B_3^3(t_1) \\ B_0^3(t_2) & B_1^3(t_2) & B_2^3(t_2) & B_3^3(t_2) \\ B_0^3(t_3) & B_1^3(t_3) & B_2^3(t_3) & B_3^3(t_3) \\ B_0^3(t_4) & B_1^3(t_4) & B_2^3(t_4) & B_3^3(t_4) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ B_0^3(t_1) & B_1^3(t_1) & B_2^3(t_1) & B_3^3(t_1) \\ B_0^3(t_2) & B_1^3(t_2) & B_2^3(t_2) & B_3^3(t_2) \\ B_0^3(t_3) & B_1^3(t_3) & B_2^3(t_3) & B_3^3(t_3) \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Treballem en `Matlab` per obtindre la gràfica pertinent a aquestes dades.

La corba apareix representada a la Figura 2.8.

Dels exemples podem observar, tal i com ja s'havia esmentat, com en general les corbes no passen exactament pels punts considerats, sinó que s'ajusten el màxim possible donades les condicions escollides. D'aquests també podem observar com incrementar la quantitat de punts de control calculats sembla millorar l'ajust de la corba. També, en prendre valors de t_i proporcionals en lloc d'equidistants, les corbes han millorat en el seu ajust.

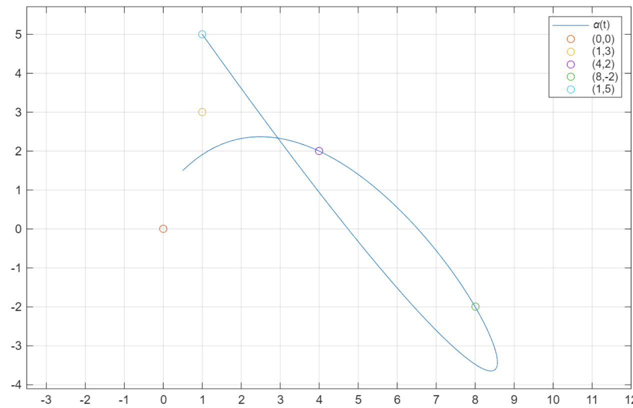


Figura 2.8: Corba de Bézier de l'Exemple 2.10 amb $n = 3$ i t_i proporcionals a les distàncies.

2.2 Corbes de Catmull-Rom

Un cop estudiades les corbes de Bézier, transladem la nostra atenció a un segon tipus de corbes *spline*: les corbes de Catmull-Rom. Per a l'estudi d'aquestes noves corbes s'ha utilitzat com a referència el document *Catmull-Rom splines* de Christopher Twigg [14]. Comencem veient una definició formal d'un segment d'una corba de Catmull-Rom.

Definició 2.8 (Segment de Catmull-Rom)**.** Un segment de Catmull-Rom és un tipus de spline cúbic que, donats 4 punts de control ordenats $\{a, b, c, d\}$, interpola els punts b i c mitjançant un polinomi cúbic $p(t)$ en $t \in [0, 1]$. Aquesta corba satisfà les següents propietats:

1. $p(0) = b$.
2. $p(1) = c$.
3. $p'(0) = \tau(c - a)$.
4. $p'(1) = \tau(d - b)$.

El paràmetre τ s'anomena tensió. Aquest controla la brusquedat de la corba d'interpolació resultant. Habitualment, s'utilitza el valor $\tau = \frac{1}{2}$.

Lema 2.3 (Construcció d'un segment de Catmull-Rom)**.** *Donats els punts de control p_0, p_1, p_2 i p_3, podem definir matricialment el segment de Catmull-Rom que interpola els punts p_1 i p_2 de la següent manera:*

$$p(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau - 3 & 3 - 2\tau & -\tau \\ -\tau & 2 - \tau & \tau - 2 & \tau \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}.$$

Demostració. Per començar, sabem que, per definició, el segment de Catmull-Rom que busquem és un polinomi cúbic, per tant, en general tindrà la forma:

$$p(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}.$$

A aquesta expressió li imposem les restriccions donades per definició d'un segment de Catmull-Rom:

$$\begin{aligned} p_1 &= p(0) = c_0. \\ p_2 &= p(1) = c_0 + c_1 + c_2 + c_3. \\ \tau(p_2 - p_0) &= p'(0) = c_1. \\ \tau(p_3 - p_1) &= p'(1) = c_1 + 2c_2 + 3c_3. \end{aligned}$$

D'aquestes equacions podem obtindre'n de noves:

$$\begin{aligned} c_2 + c_3 &= (c_0 + c_1 + c_2 + c_3) - c_0 - c_1 \\ &= p(1) - p(0) - p'(0) \\ &= p_2 - p_1 - \tau(p_2 - p_0). \end{aligned}$$

$$\begin{aligned} 2c_2 + 3c_3 &= (c_1 + 2c_2 + 3c_3) - c_1 \\ &= p'(1) - p'(0) \\ &= \tau(p_3 - p_1) - \tau(p_2 - p_0). \end{aligned}$$

Així, formem un sistema d'equacions lineals de 2 incògnites (c_2 i c_3) i 2 equacions. En resoldre'l, obtenim les següents conclusions:

$$\begin{aligned} &c_0 = p_1 = 0p_0 + 1p_1 + 0p_2 + 0p_3. \\ &c_1 = \tau(p_2 - p_0) = -\tau p_0 + 0p_1 + \tau p_2 + 0p_3. \\ &c_2 = 3(p_2 - p_1) - 2\tau(p_2 - p_0) - \tau(p_3 - p_1) = -2\tau p_0 + (\tau - 3)p_1 + (3 - 2\tau)p_2 - \tau p_3. \\ &c_3 = \tau(p_3 - p_1) - 2(p_2 - p_1) + \tau(p_2 - p_0) = -\tau p_0 + (2 - \tau)p_1 + (\tau - 2)p_2 + \tau p_3. \end{aligned}$$

Escrivim ara aquestes equacions en forma matricial.

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau - 3 & 3 - 2\tau & -\tau \\ -\tau & 2 - \tau & \tau - 2 & \tau \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}.$$

Així doncs, podem concloure que:

$$p(t) = \begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau-3 & 3-2\tau & -\tau \\ -\tau & 2-\tau & \tau-2 & \tau \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}. \qquad \square$$

Exemple 2.11. Anem a veure a continuació com canvia un mateix segment de Catmull-Rom segons el valor de τ. Per fer-ho, considerem els punts de control

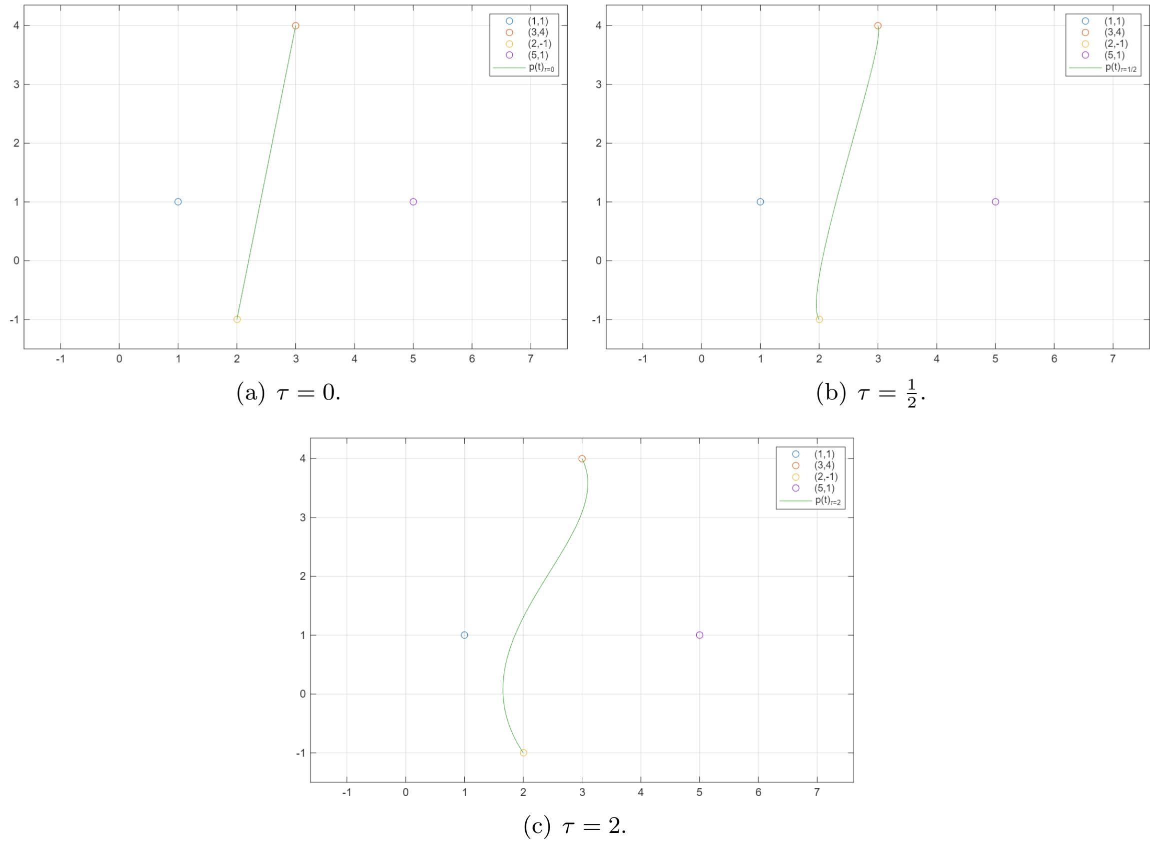
$$p_0 = (1,1) \quad p_1 = (3,4) \quad p_2 = (2,-1) \quad p_3 = (5,1)$$

També considerem $\tau_0 = 0, \tau_1 = \frac{1}{2}, \tau_2 = 2$. Utilitzem la forma matricial anterior per a construir les corbes.

$$\begin{aligned} p(t)_{\tau=0} &= \begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -3 & 3 & 0 \\ 0 & 2 & -2 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 3 & 4 \\ 2 & -1 \\ 5 & 1 \end{bmatrix} \\ &= \begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix} \begin{bmatrix} 3 & 4 \\ 0 & 0 \\ -3 & -15 \\ 2 & 10 \end{bmatrix} \\ &= \left(2t^3 - 3t^2 + 3, 10t^3 - 15t^2 + 4\right). \end{aligned}$$

$$\begin{aligned} p(t)_{\tau=1/2} &= \begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1 & -5/2 & 2 & -1/2 \\ -1/2 & 3/2 & -3/2 & 1/2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 3 & 4 \\ 2 & -1 \\ 5 & 1 \end{bmatrix} \\ &= \begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix} \begin{bmatrix} 3 & 4 \\ 1/2 & -1 \\ -5 & -23/2 \\ 7/2 & 15/2 \end{bmatrix} \\ &= \left(\frac{7}{2}t^3 - 5t^2 + \frac{1}{2}t + 3, \frac{15}{2}t^3 - \frac{23}{2}t^2 - t + 4\right). \end{aligned}$$

$$p(t)_{\tau=2} = \begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -2 & 0 & 2 & 0 \\ 4 & -1 & -1 & -2 \\ -2 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 3 & 4 \\ 2 & -1 \\ 5 & 1 \end{bmatrix}$$



(a) $\tau = 0$. (b) $\tau = \frac{1}{2}$. (c) $\tau = 2$.

Figura 2.9: Segments de Catmull-Rom de l'Exemple 2.11 segons el valor de τ.

$$
\begin{aligned}
&= \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 2 & -4 \\ -11 & -1 \\ 8 & 0 \end{bmatrix} \\
&= \left(8t^3 - 11t^2 + 2t + 3, -t^2 - 4t + 4\right).
\end{aligned}
$$

Les corbes apareixen representades a la Figura 2.9.

Definició 2.9 (Funcions de mescla)**.** Arrel de l'expressió matricial, podem definir unes funcions $w_j(t)$ continues i derivables en $[0, 1]$ anomenades funcions de mescla que satisfan les següents condicions. Donats els punts de control $\{p_0, p_1, p_2, p_3\}$ i la tensió τ, podem escriure el segment de Catmull-Rom que defineixen com

$$p(t) = \sum_{j=0}^{3} w_j(t) p_j.$$

Corol·lari 2.1. *Donada una tensió τ, les funcions de mescla són:*

$$\begin{aligned} w_0(t) &= -\tau t + 2\tau t^2 - \tau t^3. \\ w_1(t) &= 1 + (\tau - 3)t^2 + (2 - \tau)t^3. \\ w_2(t) &= \tau t + (3 - 2\tau)t^2 + (\tau - 2)t^3. \\ w_3(t) &= -\tau t^2 + \tau t^3. \end{aligned}$$

Demostració. Considerem τ una tensió qualsevol. Partim de l'expressió matricial demostrada al Lema 2.3 i fem càlculs.

$$\begin{aligned} p(t) &= \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau - 3 & 3 - 2\tau & -\tau \\ -\tau & 2 - \tau & \tau - 2 & \tau \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \\ &= \left(-\tau t + 2\tau t^2 - \tau t^3\right) p_0 + \left(1 + (\tau - 3)t^2 + (2 - \tau)t^3\right) p_1 + \\ &\quad + \left(\tau t + (3 - 2\tau)t^2 + (\tau - 2)t^3\right) p_2 + \left(-\tau t^2 + \tau t^3\right) p_3. \end{aligned}$$

D'aquesta expressió podem concloure que les funcions de mescla són:

$$\begin{aligned} w_0(t) &= -\tau t + 2\tau t^2 - \tau t^3. \\ w_1(t) &= 1 + (\tau - 3)t^2 + (2 - \tau)t^3. \\ w_2(t) &= \tau t + (3 - 2\tau)t^2 + (\tau - 2)t^3. \\ w_3(t) &= -\tau t^2 + \tau t^3. \end{aligned}$$

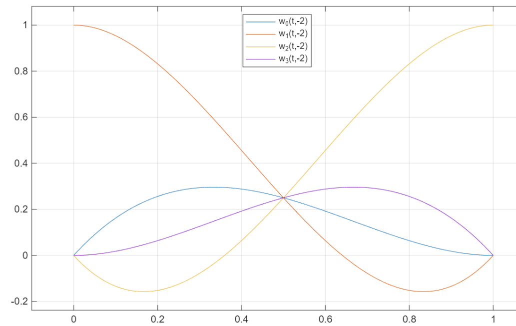
□

Nota 2.4. Com que les funcions de mescla depenen de la tensió τ, anem a incorporar aquest fet en la definició d'aquestes. Per a la resta del document, escriurem les funcions de mescla com funcions de dues variables $w_j(t, \tau)$ amb $j \in \{0, 1, 2, 3\}$ i $t \in [0, 1]$.

Exemple 2.12. Anem a veure diversos exemples de funcions de mescla segons el valor de la tensió τ.

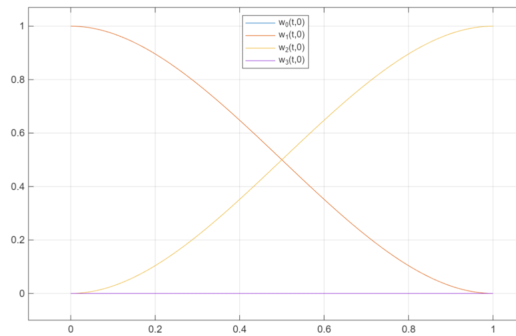
- $\tau = -2$

$$w_0(t) = 2t - 4t^2 + 2t^3$$
$$w_1(t) = 1 + -5t^2 + 4t^3$$
$$w_2(t) = -2t + 7t^2 - 4t^3$$
$$w_3(t) = 2t^2 + -2t^3$$



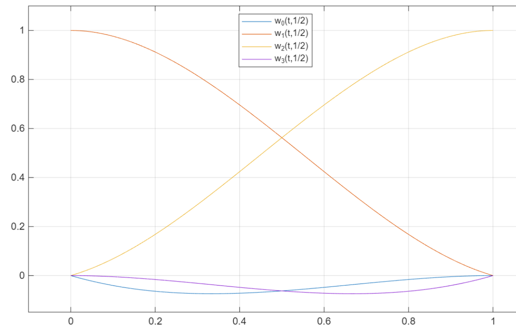
- $\tau = 0$

$$w_0(t, 0) = 0$$
$$w_1(t, 0) = 1 + -3t^2 + 2t^3$$
$$w_2(t, 0) = 3t^2 + -2t^3$$
$$w_3(t, 0) = 0$$



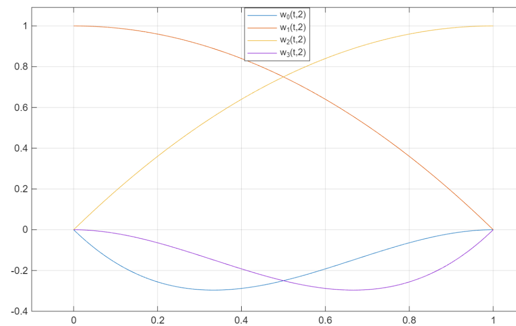
- $\tau = 1/2$

$$w_0(t) = -\frac{1}{2}t + t^2 - \frac{1}{2}t^3$$
$$w_1(t) = 1 - \frac{5}{2}t^2 + \frac{3}{2}t^3$$
$$w_2(t) = \frac{1}{2}t + 2t^2 - \frac{3}{2}t^3$$
$$w_3(t) = -\frac{1}{2}t^2 + \frac{1}{2}t^3$$



- $\tau = 2$

$$w_0(t) = -2t + 4t^2 - 2t^3$$
$$w_1(t) = 1 - t^2$$
$$w_2(t) = 2t - t^2$$
$$w_3(t) = -2t^2 + 2t^3$$



Definició 2.10 (Corba de Catmull-Rom)**.** Una corba o spline de Catmull-Rom és un tipus de spline cúbic que, donats $n+1$ punts de control ordenats $\{q_0, \cdots, q_n\}$, i una tensió τ, interpola els punts consecutius dos a dos mitjançant splines cúbics. Aquesta corba es defineix generalment per parts de la següent manera:

$$p(t) = \begin{cases} p_1(t) & t \in [0,1) \\ p_2(t-1) & t \in [1,2) \\ \vdots & \vdots \\ p_n(t-(n-1)) & t \in [n-1,n] \end{cases}$$

On

$$p_i(s) = \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau-3 & 3-2\tau & -\tau \\ -\tau & 2-\tau & \tau-2 & \tau \end{bmatrix} \begin{bmatrix} q_{i-2} \\ q_{i-1} \\ q_i \\ q_{i+1} \end{bmatrix} \quad s \in [0,1]$$

és un segment de Catmull-Rom per a tot $2 \leq i \leq n-1$.

Els casos $p_1(s)$ i $p_n(s)$ són especials, ja que no podem utilitzar la mateixa definició, degut a que els punts q_{-1} i q_{n+1} no estan definits. Per obtindre les funcions que volem tenim dues opcions:

1. Podem optar per fer la nostra corba cíclica. Així, fixem $q_{-1} = q_n$ i $q_{n+1} = q_0$. D'aquesta forma podem utilitzar la mateixa fórmula per a calcular $p_1(s)$ i $p_n(s)$ que hem gastat per a calcular la resta de $p_i(s)$. Així, obtindrem una corba que és quasi cíclica, a la qual si li afegim un segment de Catmull-Rom que interpole l'últim i el primer punts, prenent com a anterior el penúltim i com a següent el segon, formaríem un cicle.

2. Si no volem que la corba siga cíclica, el que haurem de fer serà fixar quins valors volem que prenguen $p_1'(0)$ i $p_n'(1)$. D'aquesta forma es suprimeix la necessitat de tindre els punts que no estan en la definició del segment de Catmull-Rom. Sovint es fixen a $p_1'(0) = \tau(q_1 - q_0)$ i $p_n'(1) = \tau(q_n - q_{n-1})$. Així, obtenim les funcions següents:

$$p_1(s) = \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau-3 & 3-2\tau & -\tau \\ -\tau & 2-\tau & \tau-2 & \tau \end{bmatrix} \begin{bmatrix} q_0 \\ q_0 \\ q_1 \\ q_2 \end{bmatrix} \quad s \in [0,1].$$

$$p_n(s) = \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau-3 & 3-2\tau & -\tau \\ -\tau & 2-\tau & \tau-2 & \tau \end{bmatrix} \begin{bmatrix} q_{n-2} \\ q_{n-1} \\ q_n \\ q_n \end{bmatrix} \quad s \in [0,1].$$

Exemple 2.13. Anem a considerar un mateix exemple però resolt utilitzant les dues versions que s'han descrit a la definició de les corbes. Considerem la tensió $\tau = \frac{1}{2}$ i els punts de control:

$$q_0 = (3,5) \quad q_1 = (2,2) \quad q_2 = (6,0) \quad q_3 = (8,3) \quad q_4 = (7,5)$$

Notem primer que la nostra corba tindrà la forma:

$$p(t) = \begin{cases} p_1(t) & t \in [0,1) \\ p_2(t-1) & t \in [1,2) \\ p_3(t-2) & t \in [2,3) \\ p_4(t-3) & t \in [3,4] \end{cases}$$

Utilitzem la forma matricial ja estudiada per obtindre $p_2(s)$ i $p_3(s)$.

$$\begin{aligned} p_2(s) &= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1 & -5/2 & 2 & -1/2 \\ -1/2 & 3/2 & -3/2 & 1/2 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 2 & 2 \\ 6 & 0 \\ 8 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 3/2 & -5/2 \\ 6 & -3/2 \\ -7/2 & 2 \end{bmatrix} \\ &= \left(-\frac{7}{2}s^3 + 6s^2 + \frac{3}{2}s + 2,\ 2s^3 - \frac{3}{2}s^2 - \frac{5}{2}s + 2 \right). \end{aligned}$$

$$\begin{aligned} p_3(s) &= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1 & -5/2 & 2 & -1/2 \\ -1/2 & 3/2 & -3/2 & 1/2 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 6 & 0 \\ 8 & 3 \\ 7 & 5 \end{bmatrix} \\ &= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 6 & 0 \\ 3 & 1/2 \\ -1/2 & 11/2 \\ -1/2 & -3 \end{bmatrix} \\ &= \left(-\frac{1}{2}s^3 - \frac{1}{2}s^2 + 3s + 6,\ -3s^3 + \frac{11}{2}s^2 + \frac{1}{2}s \right). \end{aligned}$$

Per a les corbes $p_1(s)$ i $p_4(s)$ fem una distinció segons quin procediment utilitzem. Comen-

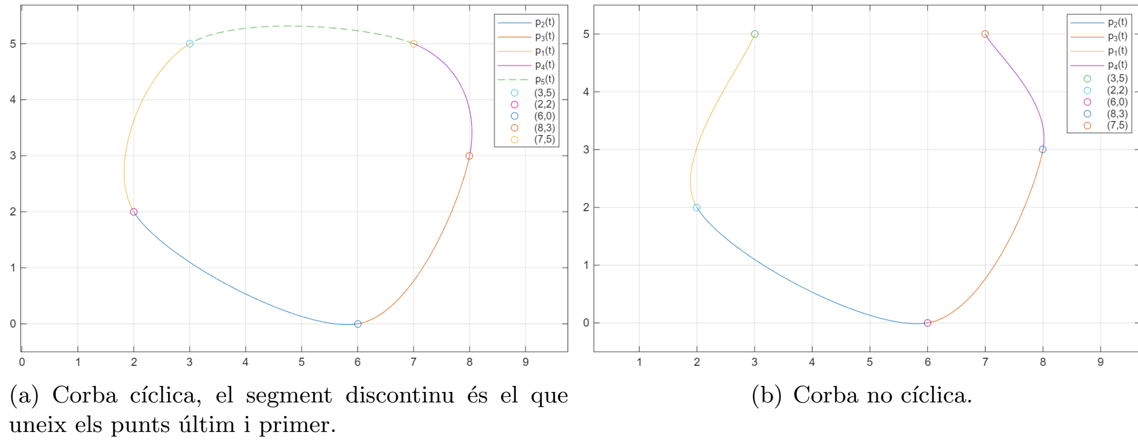
cem amb la corba cíclica.

$$
\begin{aligned}
p_1^c(s) &= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1 & -5/2 & 2 & -1/2 \\ -1/2 & 3/2 & -3/2 & 1/2 \end{bmatrix} \begin{bmatrix} 7 & 5 \\ 3 & 5 \\ 2 & 2 \\ 6 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ -5/2 & -3/2 \\ 1/2 & -7/2 \\ 1 & 2 \end{bmatrix} \\
&= \left(s^3 + \frac{1}{2}s^2 - \frac{5}{2}s + 3,\ 2s^3 - \frac{7}{2}s^2 - \frac{3}{2}s + 5 \right).
\end{aligned}
$$

$$
\begin{aligned}
p_4^c(s) &= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1 & -5/2 & 2 & -1/2 \\ -1/2 & 3/2 & -3/2 & 1/2 \end{bmatrix} \begin{bmatrix} 6 & 0 \\ 8 & 3 \\ 7 & 5 \\ 3 & 5 \end{bmatrix} \\
&= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 1/2 & 5/2 \\ -3/2 & 0 \\ 0 & -1/2 \end{bmatrix} \\
&= \left(-\frac{3}{2}s^2 + \frac{1}{2}s + 8,\ -\frac{1}{2}s^3 + \frac{5}{2}s + 3 \right).
\end{aligned}
$$

Ara fem els càlculs per a la corba no cíclica.

$$
\begin{aligned}
p_1^n(s) &= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1 & -5/2 & 2 & -1/2 \\ -1/2 & 3/2 & -3/2 & 1/2 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 3 & 5 \\ 2 & 2 \\ 6 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ -1/2 & -3/2 \\ -7/2 & -7/2 \\ 3 & 2 \end{bmatrix} \\
&= \left(3s^3 - \frac{7}{2}s^2 - \frac{1}{2}s + 3,\ 2s^3 - \frac{7}{2}s^2 - \frac{3}{2}s + 5 \right).
\end{aligned}
$$



(a) Corba cíclica, el segment discontinu és el que uneix els punts últim i primer.

(b) Corba no cíclica.

Figura 2.10: Corbes de Catmull-Rom de l'Exemple 2.13 amb tensió $\tau = \frac{1}{2}$.

$$
\begin{aligned}
p_4^n(s) &= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1 & -5/2 & 2 & -1/2 \\ -1/2 & 3/2 & -3/2 & 1/2 \end{bmatrix} \begin{bmatrix} 6 & 0 \\ 8 & 3 \\ 7 & 5 \\ 7 & 5 \end{bmatrix} \\
&= \begin{bmatrix} 1 & s & s^2 & s^3 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 1/2 & 5/2 \\ -7/2 & 0 \\ 2 & -1/2 \end{bmatrix} \\
&= \left(2s^3 - \frac{7}{2}s^2 + \frac{1}{2}s + 8,\ -\frac{1}{2}s^3 + \frac{5}{2}s + 3 \right).
\end{aligned}
$$

Amb tots els càlculs fets, podem representar les corbes al pla per veure les diferències de forma més clara. Les corbes apareixen representades a la Figura 2.10. A les gràfiques les diferències entre les dues corbes són clares.

Proposició 2.3. *Les corbes de Catmull-Rom són C^1 en el seu domini.*

Demostració. Considerem una corba de Catmull-Rom $c(t)$ amb tensió τ i punts de control $\{q_0, \cdots, q_n\}$ Recordem primer que les corbes estan definides per parts i que cadascuna de les parts és un polinomi de grau 3, per tant, la corba és contínua i derivable contínua en totes les seues parts. Els únics punts a considerar, doncs, són els punts on la corba passa d'estar definida per una funció a estar definida per una altra. Per definició de la corba, aquests punts són els números naturals entre 1 i $n-1$ ambdós inclosos. No incloem el 0 i el n ja que aquests són els extrems.

Així doncs, donat un valor $m \in \mathbb{N}$ tal que $1 \leq m \leq n-1$, volem veure si la corba $c(t)$ és contínua i derivable contínua en $t = m$. Notem primer que en aquest punt, la corba passa d'avaluar-se com $p_m(t-(m-1))$ a $p_{m+1}(t-m)$. Notem doncs que:

$$\lim_{t \to m^-} c(t) = \lim_{t \to m^-} p_m(t-(m-1)) = p_m(1) = q_m.$$

$$\lim_{t \to m^+} c(t) = \lim_{t \to m^+} p_{m+1}(t-m) = p_{m+1}(0) = q_m.$$

$$c(m) = p_{m+1}(t-m) = p(1) = q_m.$$

Per tant concloem que la corba $c(t)$ és contínua en m i en conseqüència en tots els punts del seu domini. Resta comprovar que és derivable. Notem que:

$$\begin{aligned}
\lim_{t \to m^-} \frac{c(t)-c(m)}{t-m} &= \lim_{t \to m^-} \frac{p_m(t-(m-1)) - p_{m+1}(0)}{t-m} = \lim_{t \to m^-} \frac{p_m(t-m+1) - q_m}{t-m} \\
&= \lim_{s \to 0} \frac{p_m(1+s) - p_m(1)}{s} = p'_m(1) = \tau(q_{m+1} - q_{m-1}). \\
\lim_{t \to m^+} \frac{c(t)-c(m)}{t-m} &= \lim_{t \to m^+} \frac{p_{m+1}(t-m) - p_{m+1}(0)}{t-m} = \lim_{s \to 0} \frac{p_{m+1}(s) - p_{m+1}(0)}{s} \\
&= p'_{m+1}(0) = \tau(q_{m+1} - q_{m-1}).
\end{aligned}$$

Per tant concloem que la corba $c(t)$ és derivable contínua en m i en conseqüència en tots els punts del seu domini. Concloem així que les corbes de Catmull-Rom són C^1 en el seu domini. □

2.3 Comparació entre tipus de corbes

Amb tots dos tipus de corbes definits, podem procedir a comparar-les. El primer que cal notar és que les corbes de Catmull-Rom són corbes que naturalment travessen els seus punts de control, mentre que per a aconseguir un efecte similar amb corbes de Bézier la computació addicional necessària és costosa i el resultat no té per què travessar els punts de control. Aquest és el major punt a favor de l'ús de corbes de Catmull-Rom per a aquest treball, ja que per a l'objectiu final és necessari que les corbes passen per un conjunt de punts concret. Tot i així, cal remarcar els punts a favor de les corbes de Bézier sobre les de Catmull-Rom.

Podem notar que la construcció d'una corba de Catmull-Rom no és massa costosa computacionalment parlant, ja que només requereix de dos productes de matrius, mentre que les corbes de Bézier tenen una complexitat més elevada en utilitzar l'algoritme de De Casteljau. Però, si realitzem les construccions mitjançant polinomis de Bernstein i funcions de mescla respectivament, el cost computacional s'equipara. Açò es deu a que en ambdós casos, es requereix el càlcul d'unes funcions polinòmiques i després una suma de productes de funció per punt de control.

Finalment, el major benefici en l'ús de les corbes de Bézier sobre les de Catmull-Rom és que les primeres són C^∞ en el seu domini per construcció (ja que són polinomis). Mentre que les corbes de Catmull-Rom, en estar definides a trossos, només podem assegurar que són C^1. Qualsevol grau major de suavitat de la corba dependrà dels punts de control i la tensió escollits, i per tant no es pot assegurar que la corba tinga un major grau de suavitat que C^1. En conclusió, mentre que per a altre tipus de treballs o projectes, les corbes de Bézier són idònies gràcies a la seua suavitat, per als objectius que busca satisfer aquest treball, les corbes de Catmull-Rom funcionaran millor. Açò es deu a que aquestes passen pels seus punts de control naturalment, fet necessari per mantindre la consistència en tamany del cos tou que volem animar.

Capítol 3

Integració de Verlet

La integració de Verlet és un mètode numèric utilitzat en física, principalment en l'estudi de la cinemàtica. És un algoritme senzill que serveix per calcular les posicions futures d'un punt o un cos en un espai amb forces que li donen acceleració. Així, la informació exposada en aquesta secció s'ha extret del document *Métodos numéricos* de Patricio Cordero S [4].

Partim de les fórmules de la posició i la velocitat, assumint que el pas entre un punt temporal i el següent és d'una unitat de temps:

$$\Delta t = 1.$$
$$x_{t+1} = x_t + v_t \Delta t = x_t + v_t.$$
$$v_{t+1} = v_t + a_t \Delta t = v_t + a_t.$$

Apliquem a aquestes equacions el mètode d'Euler.

$$x_{t+1} = x_t + v_{t+1}.$$
$$v_{t+1} = v_t + a_t.$$

Juntem les dues expressions anteriors per obtindre una única equació:

$$x_{t+1} = x_t + v_t + a_t.$$

Notem ara també que, de la fórmula de la posició podem aïllar el terme de la velocitat en funció de dues posicions:

$$x_t = x_{t-1} + v_t.$$

Així,

$$v_t = x_t - x_{t-1}.$$

Si substituïm aquesta expressió en la fórmula de la posició següent x_{t+1}, podem obtindre una expressió per a aquesta que no depén de cap velocitat.

$$x_{t+1} = x_t + v_t + a_t = x_t + x_t - x_{t-1} + a_t = 2x_t - x_{t-1} + a_t.$$

D'aquesta forma, arribem a l'expressió de la integració de Verlet que buscàvem.

Definició 3.1 (Fórmula de la integració de Verlet)**.** La integració de Verlet ens proporciona la següent fórmula numèrica per a calcular la següent posició d'un punt en funció de la seua posició actual, la seua posició anterior i la seua acceleració actual.

$$x_{t+1} = 2x_t - x_{t-1} + a_t.$$

Aquesta fórmula resulta molt útil a l'hora de fer la implementació en codi que volem, ja que permet alliberar càrrega computacional en el càlcul de les posicions dels punts. Aquest fet permet donar més potència a la resta de càlculs necessaris per fer que el cos que estem animant tinga un comportament tant pròxim a la realitat com siga possible, mantenint una fidelitat elevada al moviment real que hauria de tindre el cos en la situació establerta.

Capítol 4

Implementació

Per a poder arribar a la implementació final desitjada en aquest treball han fet falta moltes implementacions intermèdies. En aquest apartat es mostra el procés que s'ha seguit de forma cronològica amb explicacions de les implementacions realitzades a cada pas.

4.1 Polígon i ratolí

El primer objectiu a programar per a fer el personatge final és el seu cos, és a dir, el cos tou en si que es vol programar. Així doncs, per començar cal una figura sobre la que treballar. La més senzilla i la que es considerà òptima per a treballar fou un polígon regular, degut a la seua simplicitat.

Per crear un polígon cal denotar les posicions dels seus vèrtexs. El procediment implementat ha sigut el següent: Primer cal escollir el centre de la figura (designant les seues coordenades com (c_1, c_2)), la distància del centre als vèrtexs r i el número de vèrtexs n. Amb tota aquesta informació podem dividir 2π entre el número de vèrtexs que volem, obtenint així l'angle $\alpha = \frac{2\pi}{n}$.

Cal remarcar abans de continuar amb la construcció del polígon que en *Processing* l'origen de coordenades és l'extrem superior esquerre del llenç. A més, els valors de les coordenades horitzontals incrementen en desplaçar-se cap al costat dret del llenç i els valors de les coordenades verticals incrementen en desplaçar-se cap a l'extrem inferior del llenç. Així doncs, construïm cada vèrtex de la següent forma:

$$v_i = \left(c_1 + d \cdot \cos\left(\frac{\pi}{2} + i \cdot \alpha\right), c_2 + d \cdot \sin\left(\frac{\pi}{2} + i \cdot \alpha\right)\right) \text{ amb } 0 \leq i < n.$$

Aleshores, els vèrtexs es van creant des de l'extrem superior en ordre antihorari.

A banda de crear el polígon, aquest primer programa també té una segona funció: provar les entrades del ratolí. Aquesta funció és secundària i no es manté en la resta de programes, però és necessària de primeres per entendre vertaderament el funcionament dels paràmetres `mouseX` i `mouseY` de *Processing*.

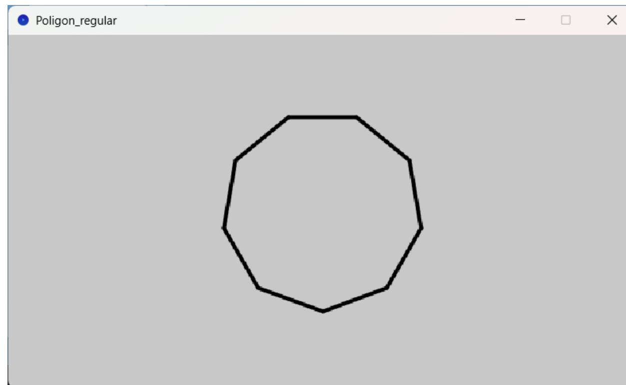


Figura 4.1: Resultat per pantalla del programa `Poligon_regular`.

La forma escollida per a observar el funcionament d'aquests paràmetres és fer el valor de r variable en funció de la posició vertical del cursor del ratolí en pantalla i de l'altura del marc h, seguint la següent fórmula:

$$r = \left| \texttt{mouseY} - \frac{h}{2} \right|.$$

Finalment, per poder visualitzar correctament tot el que està passant, cal dibuixar la figura. Per no complicar el codi de primeres i per provar diverses funcionalitats de *Processing*, la representació inicial de la figura es realitza amb línies rectes unint punts contigus. Aquestes línies les dibuixa el programa en el llenç gràcies a la funció `line`.

4.2 Gravetat

El següent pas és crear un nou paràmetre que represente la gravetat. Aquest es representa com un vector de 2 dimensions, de forma que tenim una acceleració de la gravetat tant en l'eix X com en l'eix Y, ho denotem per $g = (g_1, g_2)$. També ens cal una velocitat inicial $v^0 = (v_1^0, v_2^0)$.

Amb tots aquests paràmetres, podem aplicar la fórmula de la posició i la velocitat en un moviment rectilini uniformement accelerat (MRUA) a cada instant de temps t:

$$v(t) = (v_1(t-1) + g_1, v_2(t-1) + g_2).$$

$$X(t) = (x_1(t-1) + v_1(t), x_2(t-1) + v_2(t)).$$

Amb $v(0)$ i $X(0)$ sent la velocitat i la posició inicial que s'han establert respectivament, s'obtenen les formules necessàries per a poder marcar la posició dels vèrtexs a cada instant de temps.

Amb aquestes fórmules tenim una primera aproximació al comportament que busquem per al cos, però també sorgeix un primer problema. Sense cap tipus de restricció addicional, la figura caurà indefinidament i desapareixerà del llenç, volem que la figura quede emmarcada en tot moment, que no isca de les vores. Per poder resoldre aquest nou problema simplement cal fer una comprovació addicional del lateral del marc (dalt, baix, esquerra i dreta) cada instant de temps.

Així, cal utilitzar les funcions `min` i `max`, que reben 2 valors i retornen el menor o el major dels dos respectivament. Aleshores, si al primer valor se li assigna la posició calculada en l'instant de temps i al segon valor el valor límit, el que s'aconsegueix és que la figura no isca mai del marc, que és exactament el que buscàvem.

4.3 Fregament i ratolí

Amb les implementacions anteriors, la caiguda de la figura funciona, però queda massa rígida: busquem animar un cos tou, no un cos rígid. Així doncs, cal implementar un paràmetre que permeta que la figura rebote en arribar a terra. Anomenem fregament f a aquest paràmetre i per implementar-lo cal modificar prèviament el codi anterior, en concret el que paralitza el cos en terra quan impacta. Enlloc d'utilitzar les funcions `min` i `max`, les canviem per una comprovació per veure si el punt queda fora del marc després del moviment calculat. En cas afirmatiu, el que caldrà fer és primer fixar la posició a l'extrem pertinent (tal com ja es feia abans), i a més, invertir el sentit de la component de la velocitat referent a l'eix en el qual s'ha arribat a l'extrem i multiplicar-la per el nou paràmetre f per simular la pèrdua d'energia deguda a l'impacte.

La importància d'aquest paràmetre ve donada per la llei de conservació d'energia. Com que l'objectiu és fer una simulació realista, cal tindre en compte les lleis físiques que actuarien en una situació similar real. Així doncs, cal tindre en compte que, en rebotar contra el terra, el cos tou no pot pujar amb la mateixa potència amb la que baixa, ja que part d'aquesta quedaria transformada en calor per l'impacte. Així doncs, per la implementació seguida, cal remarcar que el paràmetre fregament ha de satisfer que $f \in]0, 1[$. Així, com més prop del 0 s'establisca aquest valor, més ràpid perdrà velocitat el cos i abans es quedarà en repòs. En canvi, com més prop de l'1 s'establisca aquest valor, més tardarà el cos en quedar en repòs.

Aquest programa també implementa una funció secundària. Fins ara, tots els programes creaven un cos centrat en el llenç en inicialitzar-se i aquest s'anima segons el programa, el segon canvi implementat és la capacitat de reiniciar aquest procés sense necessitat de tancar i tornar a iniciar el programa. Prenent la posició del ratolí amb els paràmetres `mouseX` i `mouseY` podem crear una nova instància del polígon del tamany fixat a l'inici i centrat en la posició del cursor del ratolí cada cop que es prem el botó esquerre d'aquest. Amb l'objectiu d'evitar possibles sobrecàrregues del sistema, aquesta acció crea un nou polígon al mateix temps que deixa de mostrar el polígon anterior.

4.4 Molls

Amb el programa anterior s'aconsegueix una primera aproximació a l'objectiu final, però la figura encara no es comporta com caldria. Un dels punts problemàtics són les distàncies que es formen entre vèrtexs, augmenten i disminueixen indefinidament, fet problemàtic ja que en la realitat, els cossos tous no es comporten així. El que hauria de passar és que les distàncies entre vèrtexs estiguen delimitades: tenen marge d'estirar-se i contraure's, però sempre intenten tornar a la seua posició de repòs.

Per solucionar aquest problema, creem el concepte de molls dins el programa. Els molls son variables que emmagatzemen la distància a la que han d'estar 2 vèrtexs puntuals. Aleshores, quan els punts estan massa prop o massa lluny entre ells, després de calcular la nova posició, s'afegirà una correcció en forma de vector velocitat que s'aplicarà als punts per tractar de corregir les distàncies. Cal notar que, com la quantitat de vèrtexs del polígon no és fixa, sinó que podem modificar-la des del codi, els molls s'emmagatzemen en un vector de longitud igual al nombre de vèrtexs escollits.

Per construcció, aquesta correcció no és immediata, aquest és l'objectiu. En la realitat, quan un cos tou es deforma, quan la força que l'està deformant desapareix, aquest recupera redistribueix la tensió produïda al llarg de la seva superfície continuadament, no immediatament. Aquest és el comportament que tractem de recrear.

4.5 Conservació de l'àrea

Igual que abans, amb el nou programa anterior desenvolupat, cada cop estem més prop del comportament desitjat, però encara falta altre paràmetre a conservar. En la realitat, quan un cos es deforma, no perd part del seu volum, sinó que aquest es trasllada a altres parts del cos. Per exemple, si pressionem una pilota de goma des de dalt contra una taula, aquesta es xafarà, reduint la seua altura, però també s'eixamplarà, conservant així el seu volum total.

En el cas del nostre programa, treballem en un entorn bidimensional, no tridimensional. Així doncs, el comportament observat es traduirà en una conservació de l'àrea total del polígon o cos tou que estem animant. Per aconseguir-ho, calen 2 coses: primer una funció que calcule l'àrea actual d'un polígon irregular, ja que un cop comencen a rebotar els vèrtexs i la figura es deforme, el polígon ja no serà regular, i segon cal un algoritme que allunye o aprope els vèrtexs per igual segons calga.

- **Regla del trapezoide.**

  Per calcular l'àrea d'un polígon irregular hem programat la regla del trapezoide. Aquesta consisteix en el següent: Partim d'un vèrtex qualsevol del polígon i escollim una direcció (sentit horari o antihorari). Així doncs, calculem l'àrea del trapezi format pel vector del què partim, el següent i les seues projeccions ortogonals a l'eix X. Si el segon vèrtex té un valor x major que l'original, considerem aquesta àrea positiva,

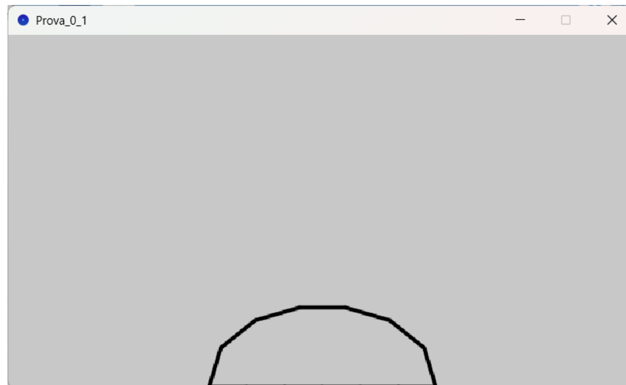


Figura 4.2: Resultat per pantalla del programa `Area`.

en cas contrari, la considerem negativa. Tot seguit prenem el segon vèrtex com al nou original i continuem al següent. Continuem aquest procediment fins tornar al vèrtex original, en aquest moment, sumem totes les àrees obtingudes i prenem el valor absolut (per a evitar problemes).

- **Vectors perpendiculars.**

  Un cop l'àrea actual està calculada, es compara amb l'àrea desitjada o inicial per saber si cal expandir el cos o contraure'l. En qualsevol dels casos, ens cal obtindre primer vectors perpendiculars als vectors formats entre dos vèrtexs consecutius, prenent el sentit apropiat segons si volem augmentar o reduir l'àrea actual. Els vectors obtinguts s'han de normalitzar, multiplicar per un factor depenent de la diferència entre l'àrea calculada i l'àrea desitjada, i aplicar als vèrtexs dels quals s'han obtingut. Així, a cada vèrtex se li aplicaran 2 vectors distints per a modificar la seua posició.

Amb aquestes modificacions es pot corregir l'àrea deformada continuadament, igual que les distàncies entre vèrtexs. El motiu de que aquest procés siga lent i no immediat és el mateix que abans: per a aproximar-se millor a la realitat amb la simulació.

També cal remarcar que, amb el mateix objectiu d'aproximar-se a la realitat, en lloc d'aplicar tots els vectors de correccions un darrere d'altre, per a cada vèrtex sumem tots els vectors de correccions i dividim les seues components entre el número total de correccions fetes en l'instant de temps. Aquest nou vector és el que realment apliquem al vèrtex.

4.6 Integració de Verlet

El següent pas que cal prendre és millorar un poc el rendiment del programa i fer-lo més llegible. Així, en lloc de les fórmules explícites que s'estaven utilitzant abans per a calcular

la posició, procedim a utilitzar la integració de Verlet, que com ja s'ha explicat abans en la Secció 3 parteix d'un mètode implícit.

Per realitzar aquest canvi, programem dintre de la classe `Vertex` la funció `Verlet` que amb els paràmetres de les posicions actual i anterior i la gravetat calcula la posició següent utilitzant la integració de Verlet.

Així doncs, amb aquesta funció creada, només cal reescriure la funció que actualitza les posicions dels vèrtexs cada instant de temps dintre de la classe `Poligon`. El primer que haurà de fer, doncs, és obtindre les noves posicions dels vèrtexs del polígon utilitzant la funció `Verlet` que acabem de crear. Un cop obtingudes, cal calcular totes les correccions necessàries a les posicions com s'ha discutit a les Seccions 4.4 i 4.5. Amb totes les correccions necessàries calculades, obtenim els vectors de correccions totals de cada vèrtex i els apliquem. Donem així per finalitzada l'actualització de les posicions. Resta només dibuixar els costats de la figura.

A banda de la implementació de la integració de Verlet dins el codi, també s'ha fet una segona implementació addicional en aquest programa. Ací per primera vegada es permet agafar el cos i desplaçar-lo per la pantalla. Per aconseguir aquest efecte han fet falta diverses implementacions.

Primer cal una variable *booleana* nova per a la classe `Poligon` que indique si el cos està sent subjectat per l'usuari o no, ja que si està agafat, no volem que es puga tornar a agafar fins a que no es solte. També cal una funció que detecte si el cursor del ratolí està situat dintre de la figura o no. Per fer aquesta funció seguim un procediment similar al que hem realitzat per a fer el càlcul de l'àrea.

Partim de que el cursor està fora, i escollim un vèrtex inicial. Recorrem els vèrtexs un a un en ordre (tant horari com antihorari són vàlids, al programa s'ha utilitzat un ordre antihorari) i comprovem primer si la posició horitzontal del ratolí està entre les posicions horitzontals dels dos vèrtexs considerats. En cas negatiu, passem als següents vèrtexs. En cas afirmatiu, comprovem si el cursor està situat per damunt del segment de recta format unint els dos vèrtexs sobre els que estem treballant. Si aquesta segona condició es compleix, aleshores invertim la situació, és a dir, si estàvem considerant que el cursor estava dins, ara considerem que està fora, i a la inversa. Si la segona condició no es compleix, aleshores no passa res. En qualsevol dels casos, passem al següent parell de vèrtexs i repetim la comprovació. Seguim així fins a recòrrer tots els vèrtexs i el resultat final indicarà si el cursor està realment dins o fora de la figura.

Una vegada tenim tots els ingredients preparats, ja podem programar l'acció d'agafar el cos. Aleshores, si el cursor del ratolí està dins de la figura (comprovació que es fa amb la funció anteriorment descrita) i es prem el botó esquerre d'aquest, la figura passa a estar en estat `agafada`. Quan es prem el botó, el programa registra les distàncies del cursor als distints vèrtexs i així, mentre la figura estiga agafada, es calcula un vector nou de correccions per a cada vèrtex. Aquest nou vèrtex es calcula simulant l'existència d'un moll addicional que connecta el cursor amb cada vèrtex i que mesura la distància calculada a l'inici. Aquesta nova correcció es suma a les anteriors per calcular el vector de correccions

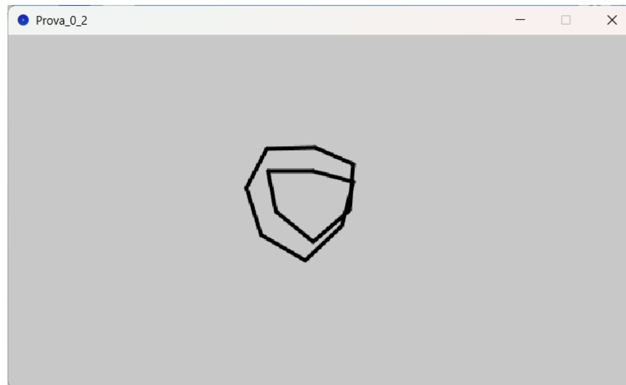


Figura 4.3: Resultat per pantalla del programa `Verlet`.

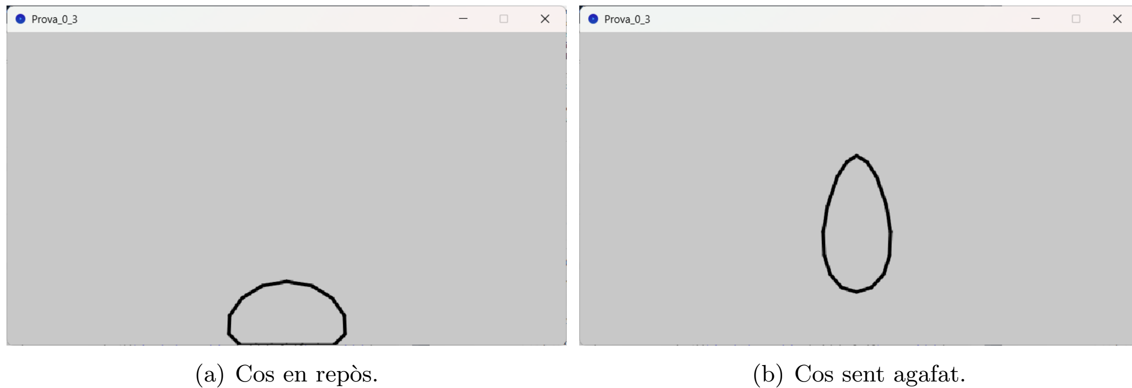
total. La resta funciona igual que abans. D'aquesta forma, es simula l'acció d'agafar el cos tou.

4.7 Revisió i ajust de paràmetres

En executar el programa anterior sorgeix un problema evident: degut al codi escrit, en agafar el cos, aquest s'entrecreua entre si, deformant-lo de formes que deurien ser impossibles. Aquest següent programa tracta de solucionar aquest problema.

Cal reescriure la funció que s'encarrega de subjectar el polígon, ja que aquesta és la causa del problema. Així, la primera correcció serà la quantitat de vèrtexs que es fixen. Podem notar com el fet de fixar distàncies en tots els vèrtexs, en afegir la gravetat i el fet de que les correccions no ocorren de forma immediata, fa que la figura s'entrecreue de formes poc realistes, aleshores la primera solució trobada és limitar els vèrtexs que es fixen a només uns pocs. El criteri escollit ha sigut prendre els vèrtexs situats per damunt del punt del que s'agafa la figura i tractar aquests com a àncores, mentre que els altres ja no tenen les restriccions addicionals per estar sent agafats.

D'aquesta correcció sorgeix un nou problema, ara quan s'agafa la figura sembla més que queda penjant d'un fil i no que estiga sent agafada en si. També hi ha un segon problema, en modificar el programa, ara el cos té una aparença massa líquida i pareix que té poca consistència. La causa són els paràmetres escollits com a valors d'àrea desitjada i longitud dels molls, que donen massa marge de deformació a la figura. Aquests problemes perduraran diverses iteracions més del programa degut a que no es trobava cap solució satisfactòria, però finalment quedaran solucionats.



(a) Cos en repòs.

(b) Cos sent agafat.

Figura 4.4: Resultats per pantalla del programa `Ajust_Verlet`.

4.8 Color

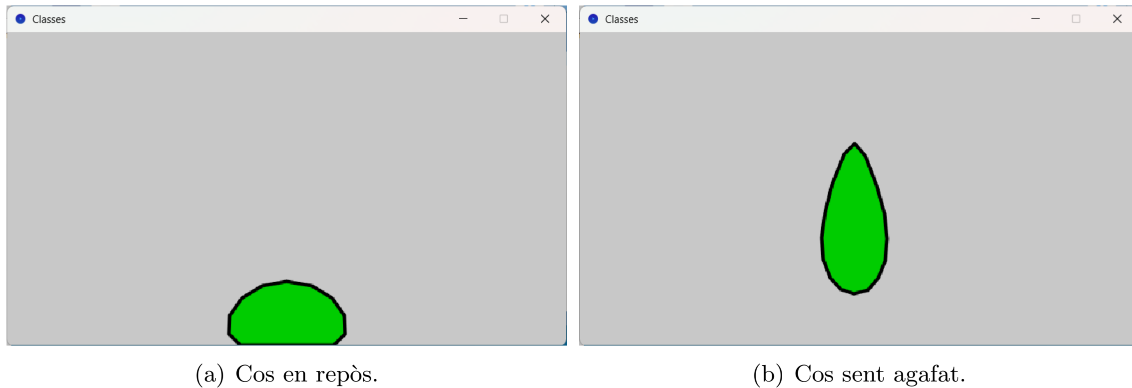
Amb els procediments bàsics del funcionament del programa fets, arriba el moment de començar a tractar la part estètica del programa. Fins ara, el cos sobre el que s'estava treballant era un polígon transparent, però l'objectiu final és obtindre un cos similar a una granota. Així doncs, cal corregir la transparència i que l'interior de la figura tinga color.

Per aconseguir aquest efecte cal modificar lleugerament la forma de dibuixar la figura en pantalla, ja que fer càlculs constantment per calcular l'àrea exacta que s'ha de cobrir de color és poc òptim i podria causar problemes de rendiment. Així doncs, en lloc de dibuixar les rectes manualment, procedim a utilitzar l'entorn `Shape` que proporciona el mateix programari que s'està utilitzant. Aquest entorn permet dibuixar una figura de costats rectes denotant els seus vèrtexs de forma ordenada, que a efectes pràctics és el que ja s'estava fent. La diferència, és que aquest entorn admet la funció `fill()` que internament ompli l'àrea interior de forma que es pinte amb un color llis que se li assigne en valors del sistema de color *RGB*. Així és com obtenim la figura que teníem, però ara plena de color.

4.9 Bézier

Un cop aconseguit el color, el següent objectiu estètic a aconseguir és suavitzar les vores del cos. És a dir, fins ara s'està treballant amb polígons de costats rectes, ara el que busquem és canviar les rectes per corbes i obtindre una silueta suau, sense pics abruptes ni talls de continuïtat. La primera opció que es considerà fou utilitzar corbes de Bézier per a la silueta del cos. Aquest programa tracta d'implementar-les.

Pel funcionament intern de *Processing*, no permet dibuixar segments corbs directament, així que el que cal fer és calcular punts que estiguen en la corba que volem dibuixar i després unir-los amb rectes. Ací trobem la primera problemàtica: per a obtindre un resultat que



(a) Cos en repòs. (b) Cos sent agafat.

Figura 4.5: Resultats per pantalla del programa `Color`.

realment simule una corba, cal una quantitat de punts intermedis calculats molt elevada, si hi ha massa pocs, la il·lusió desapareix i es veuen les línies rectes. Aquest fet ens dona dos opcions per a procedir: podem calcular inicialment molts punts i després deixar de calcular els punts de les corbes i simplement tractar-los tots com a vèrtexs, o podem mantindre pocs vèrtexs que tinguen els càlculs de posicions i en cada instant, una volta estiguen les posicions calculades, calcular la resta de punts per formar la corba i dibuixar-la així.

Considerem primer la primera opció. Podem fer un càlcul inicial d'una corba de Bézier amb els punts del polígon com a punts de control utilitzant l'algoritme de De Casteljau (ja descrit a la Secció 2.1). Fixem una precisió que corresponga al nombre de vèrtexs que volem per al cos i tots els càlculs posteriors els fem per a aquests vèrtexs. En executar aquest nou programa notem immediatament un problema seriós. Si augmentem molt la precisió, el cos queda massa poc consistent i acaba convergint a un punt. En canvi, si reduïm la precisió, ens quedem amb un cos equivalent al que ja teníem inicialment, però més menut. Així doncs, aquesta opció no és útil.

Considerem doncs la segona opció. Ací sorgeix un nou problema, que és el motiu pel qual aquesta opció ni tan sols ha sigut implementada en codi. Com bé ja hem vist, les corbes de Bézier generalment es defineixen amb punts de control, no amb punts pels quals passa la corba. Açò és especialment important amb aquesta opció a diferència de l'anterior ja que si només es calcula la corba primerament amb els vèrtexs com a punts de control, simplement obtindrem un cos més menut que el polígon original. En canvi, si volem fer aquest càlcul cada instant de temps, hem de considerar els vèrtexs com a punts pels quals passa la corba, ja que si els considerem punts de control, el cos es faria cada cop més menut, causant problemes majors. Així, hem de calcular a cada instant de temps la corba de Bézier que passa pels vèrtexs. Com ja hem vist a la Secció 2.1, aquest càlcul requereix d'un procés d'optimització o com a mínim de càlculs matricials amb productes i càlculs de matrius inverses. El programari utilitzat és un *software* preparat per a dibuix i animació,

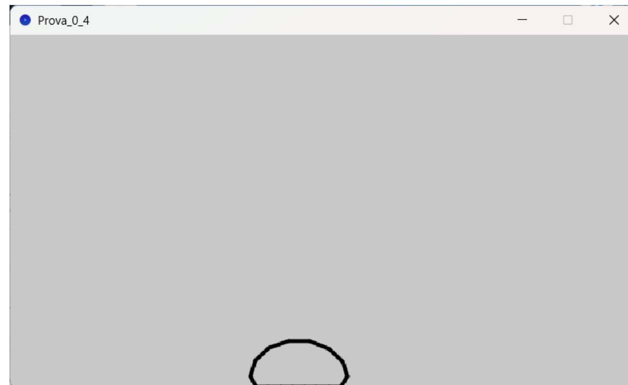


Figura 4.6: Resultat per pantalla del programa `Bezier`.

no per a càlculs matemàtics, així doncs, aquests càlculs necessaris són molt tediosos i carreguen molt al programa, causant potencials problemes de rendiment o directament errors d'execució. Per aquests motius s'ha valorat que no era un camí viable a seguir.

4.10 Revisió de Bézier

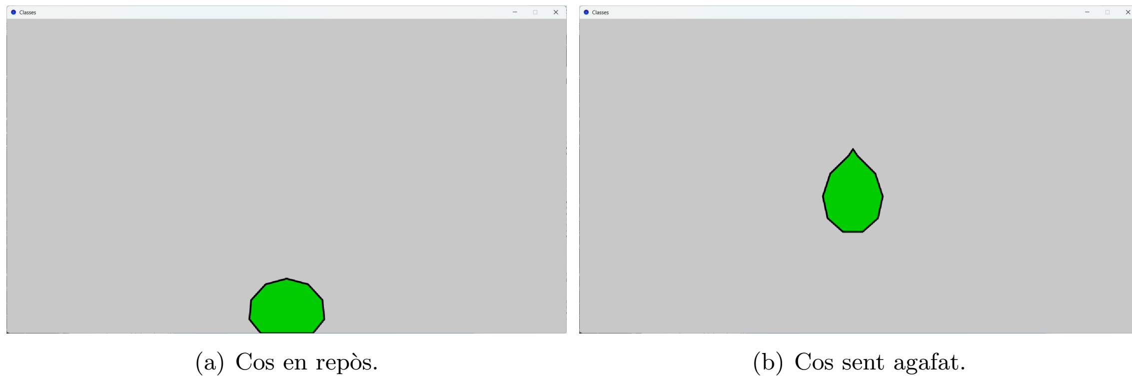
Procedint amb els raonaments anteriors, la decisió final pel que fa a la implementació de corbes de Bézier ha sigut mantindre el càlcul inicial d'una corba de Bézier amb els vèrtexs del polígon com a punts de control i operant tota la resta del programa amb els nous punts de la corba.

La resta del codi no ha sigut modificat en gran mesura, el que s'ha editat amb aquesta nova versió del programa són les constants i els paràmetres, amb l'objectiu d'obtindre una animació més realista com a producte final: el valor de la gravetat, del fregament, la velocitat inicial, el nombre de vèrtexs... L'objectiu d'aquest programa és millorar el realisme de l'animació mitjançant un ajust de paràmetres. Ací s'aconsegueix que el cos deixe de parèixer tant líquid.

A banda d'aquest ajust, també s'ha modificat lleugerament la funció que s'encarrega de la lògica darrere d'agafar el cos. En lloc de subjectar els vèrtexs que estan situats sobre el cursor a l'hora d'agafar el cos, tal com s'estava fent anteriorment, ara fixem 3 vèrtexs decidits prèviament. Dona igual des d'on s'agafe la figura, sempre seran els mateixos 3 vèrtexs els que es fixen.

4.11 Treball a Matlab

Vistes les limitacions per part del programari utilitzat, finalment no s'han utilitzat les corbes de Bézier al codi. Tot i així, assumint l'existència d'altres programaris suficientment



(a) Cos en repòs. (b) Cos sent agafat.

Figura 4.7: Resultats per pantalla del programa `Revisio_Bezier`.

potents com per a suportar la càrrega de càlculs, s'han fet funcions de `Matlab` que duen a terme els càlculs que caldrien en un instant de temps, mostrant les corbes resultants per pantalla.

La primera de les funcions, `pol_Bernstein`, pren els valors i i n i retorna el polinomi de Bernstein $B_n^i(x)$ com a funció anònima de x.

La segona de les funcions, `De_Casteljau`, és una implementació directa del mètode que li dona nom. Així doncs, pren com a paràmetres primer un llistat de valors de control (correspondran posteriorment la llista formada per una de les components dels vectors del llistat de punts en el pla) i la precisió. La precisió ve donada per un nombre natural i fa referència a la quantitat de punts intermedis equidistants que es calcularan al codi i que per tant apareixeran com a resultat de la funció. Així doncs, el que retorna aquesta funció és un llistat de valors pels quals la corba passa de forma equidistant en el temps.

Amb aquestes dues funcions bàsiques programades, es pot procedir ja a la creació de funcions més complexes que comencen a fer el que realment busquem. Aleshores, la següent funció que s'ha programat ha sigut `Rep_Bezier`. Aquesta funció té el paper de traure per pantalla la corba de Bézier obtinguda dels punts de control marcats. Així doncs, pren d'entrada un llistat de punts en l'espai, un valor de precisió i un paràmetre *booleà* per indicar si es vol la representació desglossada o no. Amb aquestes dades, el codi utilitza la funció `De_Casteljau` per a calcular amb la precisió donada les coordenades dels punts pels quals passa la corba de Bézier, calculant per una banda les primeres coordenades i per altra les segones, tot per a ajuntar-ho en un gràfic bidimensional que es mostra en pantalla on es pot veure la corba final de Bézier. En el cas en que el valor del paràmetre `desglossar` siga vertader, aleshores aquest procés es fa diverses vegades, primer amb només el primer punt de la llista i cada cop afegint-ne un més fins arribar a tots. Així, s'obté una evolució de la corba de Bézier representant com es va modificant a mesura que li afegim punts de control nous.

Les següents funcions creades són les que s'encarreguen de calcular els punts de control

de corbes de Bézier que passen per un llistat de punts ordenats. Per simplificar el codi, s'ha evitat programar el problema d'optimització dels paràmetres t_i, enlloc d'això, s'han considerat les dues suposicions més comunes a l'hora de fer aquest càlcul més simple i menys pesat a nivell de processament (aquestes ja estan explicades a la Secció 2.1), evitant així el problema d'optimització. La primera funció, `BezParamEqui`, assumeix els punts equidistants en el temps i fa servir el procediment ja vist per calcular tants punts de control com s'indique en cridar a la funció en un programa. En canvi, la funció `BezParamProp` pren les distàncies espacials entre els punts que rep i calcula les distàncies temporals que hauran de tindre de forma proporcional. Amb els punts en el temps calculats, ja pot fer servir el mateix procediment que abans per a calcular tants punts de control com s'indiquen.

Finalment, les últimes dues funcions implementades, anomenades `Rep_BezParamEqui` i `Rep_BezParamProp`, s'encarreguen de mostrar i calcular punts de corbes de Bézier que passen per un llistat de punts a l'espai donats, indicant quants punts de control haurà de tindre la corba mostrada, la precisió que es vol tindre en la representació i si es volen desglossar els passos o no. Així, primer aprofiten les funcions `BezParamEqui` i `BezParamProp` respectivament per a calcular tants punts de control com s'indiquen per a la corba de Bézier que passa per un llistat de punts ordenats donada. Un cop es tenen aquests punts de control, s'utilitza la funció ja descrita `Rep_Bezier` per a representar la corba de Bézier amb els punts de control calculats, la precisió demanada i desglossada en cas de que així s'indique. Addicionalment, a la gràfica final apareixeran també marcats els punts inicials per els quals se suposa que hauria de passar la corba. Gràcies a aquesta adició final, podem comprovar com, segons els punts donats, per a poder obtindre una corba de Bézier que passe per tots caldran a voltes més o menys punts de control.

4.12 Canvi de Bézier a Catmull-Rom

Com ja s'ha dit diverses voltes al llarg del treball, la idea original era treballar amb corbes de Bézier. Però finalment s'han utilitzat corbes de Catmull-Rom. La decisió de fer aquest canvi s'ha pres en base a 2 factors determinants.

Primerament, el cos s'ha definit utilitzant punts sobre els que passa la seua silueta. Açò presenta un clar problema si s'utilitzen corbes de Bézier per a dibuixar-lo, ja que aquestes es defineixen utilitzant punts de control pels quals la corba generalment no passa. Tot i així, a la Secció 2.1 hem vist un procediment pel qual es pot obtindre una corba de Bézier que s'ajuste a uns punts donats. El problema ve del fet que, com ja hem remarcat a la Nota 2.3, la corba ajustada no té perquè passar pels punts. Així doncs, si s'utilitzaren corbes de Bézier per dibuixar el cos, el tamany podria resultar massa inconsistent degut als errors acumulats.

Addicionalment, el programari *Processing* és bastant potent a l'hora de fer animacions per ordinador. En canvi, a l'hora de fer càlculs matemàtics no ho és tant. Així, sorgeix el problema de la capacitat de computació. Com hem vist a la Secció 2.1, per obtindre

Figura 4.8: Resultat per pantalla del programa `Pendol`.

una corba de Bézier que passe per uns punts concrets, calen fer diversos càlculs matricials. Entre els quals es troba el càlcul d'una inversa també. *Processing* no està preparat per a realitzar aquests càlculs diverses voltes per segon. A més, el llenguatge de programació utilitzat no té suport per a treballar amb matrius, fet que complica encara més els càlculs.

Per aquests motius determinants, s'ha decidit prescindir de l'ús de corbes de Bézier en el treball final. Així, cal trobar un nou tipus de corba *spline* per a utilitzar en l'animació per a suavitzar la forma del cos. Es busca un tipus d'*spline* que per una banda es definisca amb els punts que travessa i que no tinga una alta necessitat computacional a l'hora d'obtindre la seva expressió numèrica. Les corbes de Catmull-Rom satisfan aquests dos requeriments. Per aquest motiu, han sigut les escollides per a continuar amb el treball.

Així doncs, s'han modificat totes les funcions encarregades de dibuixar per pantalla les parts del cos. Ara totes aquestes utilitzen corbes de Catmull-Rom per dibuixar el cos.

4.13 Pèndol

Amb el cos en funcionament, és moment de programar les potes de la granota. Per fer-ho, primer hem fet un programa que simule el funcionament d'un pèndol. Aquest és realment un pèndol doble, ja que farà el paper de les potes i aquestes tenen dues seccions rectes amb una articulació en mig.

Aquest programa es basa en el funcionament físic d'un pèndol doble, les cordes del qual no es poden allargar o acurtar en cap moment. Així doncs, si degut a la gravetat aquestes distàncies es modificaren, el programa fa una correcció de la posició per a fixar així el radi. A diferència del cos de la granota que és un cos tou, les potes són rígides, així, a diferència de les correccions lentes fetes en programes anteriors, si cal corregir la posició de les articulacions de les potes, aquesta correcció es fa de forma immediata fixant els punts on haurien d'estar.

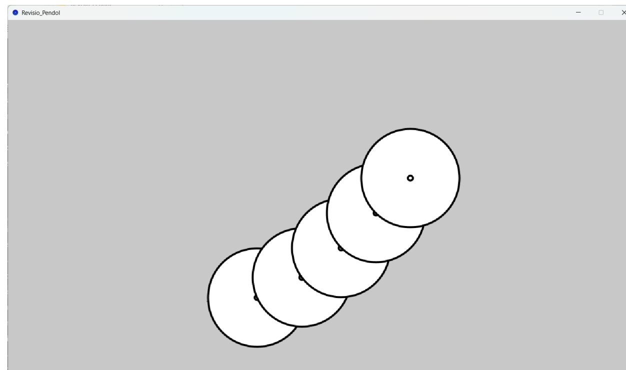


Figura 4.9: Resultat per pantalla del programa `Revisio_Pendol`.

4.14 Revisió de pèndol

El resultat obtingut al primer programa del pèndol no ha resultat massa satisfactori, per tant s'ha valorat fer una revisió per enfocar el problema des d'altre punt de vista. Així comença aquest nou programa.

Ací es crea una nova classe per als pèndols. Aquesta emmagatzema tant el nombre d'articulacions que té la figura (contant tant la primera com la última), com les seues posicions i les distàncies que hi ha entre cada parell d'articulacions consecutives. La primera articulació actua com a clau fixat que delimita les posicions de les posteriors articulacions.

Aleshores, cada instant de temps s'actualitza la posició del clau al lloc desitjat i una a una s'actualitzen les posicions de la resta d'articulacions de la següent manera, començant per l'articulació contigua al clau. Primer es comprova si la distància actual entre l'articulació que estem actualitzant i l'anterior (ja corregida) és la que hauria de ser. En cas afirmatiu, no cal canviar la posició i podem procedir a la següent articulació. En cas negatiu, prenem el vector que defineixen aquestes dues articulacions en sentit i direcció cap a la que volem corregir. Normalitzem aquest vector i el multipliquem per l'escalar corresponent a la distància que ha d'haver entre les articulacions. Finalment apliquem aquest vector obtingut a l'articulació anterior per obtindre així la nova posició a la que ha d'estar l'articulació que estem actualitzant. Corregim la posició i podem passar ja a la següent articulació. Un cop estiguen totes actualitzades, ja podem dibuixar en pantalla la posició del pèndol actualitzada.

A diferència del programa anterior, aquesta implementació sí que ha resultat satisfactòria per a l'objectiu.

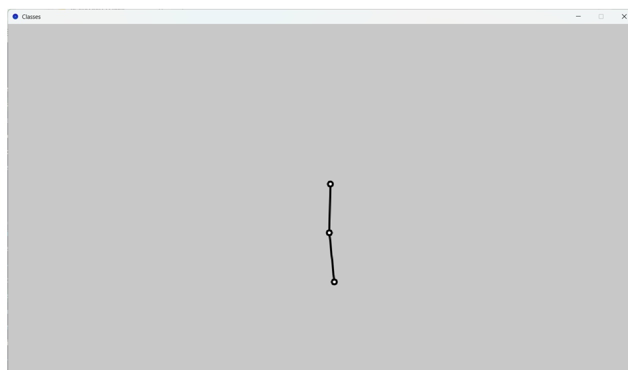


Figura 4.10: Resultat per pantalla del programa `Restriccions_Pendol`.

4.15 Restriccions del pèndol

Abans de poder ajuntar tot el treball fet, queden unes xicotetes correccions per al funcionament dels pèndols.

Primer fixem el fet de que tots els pèndols a partir d'ara tindran exactament 3 articulacions, ja que van a fer el paper de les potes. Així, el codi resultant podrà quedar més net i simplificar la seua lectura.

També, cal implementar l'impacte de la gravetat en les posicions de les articulacions, ja que aquesta també estira d'aquestes. Així doncs, aprofitem la funció que implementa la integració de Verlet per a aquest procediment, simplificant així la tasca de programació. Només cal tindre en compte que aquest càlcul cal fer-lo abans de les correccions i a més, que s'han de revisar les col·lisions amb els extrems del marc per a que les futures potes no se n'isquen d'aquest.

Finalment, la implementació clau que quedava per fer abans de poder ajuntar tot el treball en un únic programa: les restriccions angulars. Si observem un braç, per exemple, podem veure com les articulacions no tenen un marge de rotació de 360 graus, sinó que tenen límits de fins a on es poden doblar. Aquest fet és el que faltava per implementar. Així doncs, només cal fer una comprovació addicional mentre s'està corregint la posició de les articulacions, abans de passar d'una a la següent. Un cop obtinguda la posició, caldrà comprovar l'angle que forma amb l'articulació anterior. Si aquest angle supera els límits establerts, caldrà corregir la posició un cop més, per a que l'angle que quede al final siga el màxim i no el supere. Si l'angle format es troba dins dels límits permesos, no caldrà fer cap modificació extra. Amb aquesta comprovació completa, ja es podrà passar a la següent articulació i tornar a repetir el procediment.

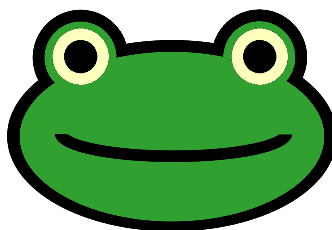


Figura 4.11: Imatge utilitzada per al cap de la granota.

4.16 Granota

Arribat aquest punt, totes les peces que conformen el producte final que busquem estan programades, només ens cal ajuntar-les en un únic programa per a poder donar la simulació per programada i així poder provar-la pròpiament.

El primer que cal és formar una nova classe que anomenarem `Granota`. Aquesta cohesionarà en una única entitat totes les diferents parts que conformen el cos final. Aquest haurà de contindre 1 cos, 4 potes i 1 cap. El cos estarà format per 1 polígon, cada pota serà un pèndol i el cap serà una imatge en format *.png*. Pel que fa a les potes hi haurà de 2 tipus: les 2 potes de davant seran més curtes i tindran més restricció de moviment, així com les 2 potes de darrere seran més llargues i tindran més llibertat de moviment.

La primera de les components que caldrà crear serà el cos, ja que és la figura central de la composició i la resta de parts van adherides a aquest. Així, amb el cos format, podem fixar 4 posicions relatives als distints vèrtexs del polígon per a que cada una actue com a clau per a una de les potes. Finalment, cal fixar 1 posició més dins del cos per a que actue de clau per al cap. Les posicions per a les potes de davant s'han fixat prenent dos vèrtexs de la part inicialment superior del cos de la granota, dividint el segment que els uneix en 5 parts iguals i prenent la primera i la última partició com a posicions per a les potes. Per altra banda, les posicions de les potes de darrere s'obtenen des de dos vèrtexs de la part inicialment inferior del cos, desviant-los lleugerament cap al mig del cos. A més, la posició del cap es fixa al vèrtex inicialment superior, que correspon al primer i últim del polígon.

Per a actualitzar la posició en cada instant de temps es fa us de les funcions prèviament creades per a cada classe. Així, el primer que s'actualitza ha de ser la posició del cos, ja que d'aquesta depenen totes les altres. Un cop actualitzat el cos, es recalculen les posicions dels claus seguint el mateix procediment que en els seus respectius càlculs inicials. Amb els claus calculats, es pot cridar a la funció que actualitza la posició dels pèndols fixant la nova posició que té cada clau. Després, es reposiciona la imatge del cap sobre el clau pertinent. Per acabar, es dibuixen les distintes components de la granota en pantalla seguint aquest ordre: primer les potes de darrere, després el cos, a continuació les potes de davant i finalment el cap.

Una xicoteta modificació que s'ha fet a les funcions d'actualitzar les distintes figures ha

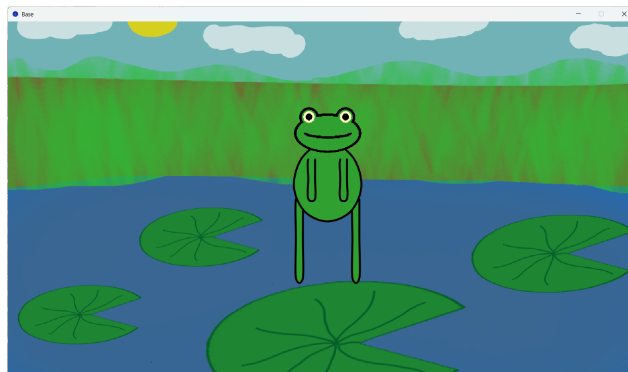
Figura 4.12: Imatge utilitzada per al fons.

sigut suprimir la crida a la funció que les dibuixa en pantalla. L'objectiu és poder controlar millor l'ordre en que es dibuixa cada part, per poder predir què apareixerà per sobre. A més, aquesta forma d'organització queda més neta al codi, facilita la seua comprensió i evita possibles redundàncies.

Seguint amb el tema de les funcions que s'encarreguen de dibuixar, fins ara els pèndols es dibuixaven unint les articulacions amb línies rectes, però en el context de la granota açò suposa un problema, ja que les potes haurien de tindre un mínim de grossor i volum. Així, la solució emprada ha sigut utilitzar de nou corbes de *Catmull-Rom*. Per fer-ho, es defineixen punts auxiliars fent un desplaçament perpendicular als segments originals des de les articulacions i unint aquests nous punts amb una corba. A diferència de la corba utilitzada per al cos, aquesta s'ha decidit fer oberta, és a dir, que no comença i acaba en el mateix punt, creant així la il·lusió que la pota està adherida al cos i no és un bloc posat a sobre d'aquest. Els punts inicial i final son els punts auxiliars obtinguts des del clau. Per acabar amb la part estètica del programa, també s'ha il·lustrat un fons per substituir el fons gris per defecte que utilitza *Processing*.

A més, s'ha aprofitat aquest codi definitiu per a homogeneïtzar i comentar amb més profunditat el codi. Açò es fa amb objectiu de facilitar possibles treballs futurs, tant propis com d'altres desenvolupadors. A més, en cas de necessitar fer cap tipus de correcció posterior o implementació alternativa, un codi ben estructurat i comentat facilita la feina.

(a) Granota en repòs.



(b) Granota sent agafada.

Figura 4.13: Resultats per pantalla del programa `Granota`.

Capítol 5

Conclusions

Amb totes les implementacions realitzades poc a poc s'ha aconseguit arribar a un producte molt pròxim a allò que s'havia visualitzat originalment. Alguns canvis s'han hagut de realitzar pels imprevistos que han sorgit durant el procés de programació, però cap ha sigut fatal.

5.1 Valoració

Tenint la realització d'una animació procedimental per a un cos tou en 2 dimensions com a objectiu, hem pogut estudiar diversos conceptes.

Les corbes *splines* són tot un món. Hi ha de molts tipus amb aplicacions molt diferents. Al llarg del treball hem centrat l'atenció en 2 d'aquests tipus: les corbes de Bézier i les corbes de Catmull-Rom. S'ha dedicat temps a comparar-les, veure els punts forts i febles de cada tipus per a la feina que volíem fer. Tot i que la conclusió final ha sigut decantar-se per les corbes de Catmull-Rom, també hem explorat alguns escenaris per als quals les corbes de Bézier podrien ser útils o inclús millors. Més endavant hem treballat amb un mètode numèric basat en el mètode d'Euler per a descriure un moviment al pla. La integració de Verlet, tot i ser un mètode bastant específic, en el seu àmbit fa molt bé el seu paper i permet fer càlculs de posicions i moviments amb facilitat.

A través de la implementació en codi *Processing* s'han trobat tant oportunitats d'aprofundir coneixements com obstacles a superar. Sense tots dos no s'hauria arribat al resultat final al que hem arribat. La realització d'aquest treball ens ha permés aprofundir en temes d'interès, com les corbes al pla. A més, també ens ha permés treballar i millorar les habilitats de programació tant en `Matlab` com en un llenguatge completament nou, *Processing*. També ens obri possibilitats d'expansió del treball de cara al futur.

5.2 Treball futur

Per les limitacions temporals del treball, no es pot fer tot. Encara així, havent aconseguit programar una animació procedimental d'un cos tou tal i com volíem, s'han pogut veure a través del desenvolupament del treball diverses possibilitats d'expansió del codi. Aquestes podrien ajudar a simular més situacions diferents però similars al que s'ha realitzat, ja que la nostra simulació només considera un únic cos tou sense massa interferències. Anem a veure alguns exemples.

5.2.1 Obstacles amb col·lisions

Una primera implementació addicional a considerar és la implementació d'un sistema de col·lisions amb obstacles fixes en l'espai. Rectes que tallen el llenç o polígons fixats. L'objectiu seria que el cos els tracte de parets fixes, com ocorre amb els extrems del llenç. Per realitzar aquesta tasca primer caldrà considerar les col·lisions del cos a rectes horitzontals i verticals. A continuació es faran simplificacions dels distints objectes que apareguen pel llenç utilitzant quadrats i abans de comprovar si el cos impacta amb la figura en si, es vorà si el cos xoca amb aquestes simplificacions.

El motiu d'aquest càlcul previ és el següent. Calcular col·lisions ortogonals és molt més lleuger de processament que calcular col·lisions diagonals. Per tant, fer el càlcul previ permet fer el segon només quan realment és necessari, reduint així el requeriment de processament del programa, intentant optimitzar-lo tot el possible.

5.2.2 Més d'un cos

Un cop les col·lisions a obstacles fixes estiguen ben programades, el següent pas lògic seria la implementació d'un segon cos tou, implementant interaccions entre els dos. En aquest cas, però, els cossos que interactuen son dos corbes. Aquest fet intensifica el problema explicat anteriorment. Així doncs, el que caldrà fer serà una implementació per parts: fent comprovacions amb quadrícules cada cop més fines, acabant finalment amb la comprovació de col·lisions de dos corbes. Així, només caldrà fer la comprovació de corbes, que és notablement més costosa a nivell de processament, quan els dos cossos estiguen vertaderament prop. En qualsevol altre cas, les comprovacions pararan quan en alguna de les quadrícules ja no es superposen els cossos. El nombre total de quadrícules a fer s'hauria d'obtindre segons el que es vullga, augmentant o reduint-lo segons convinga.

Un cop hi haja una implementació programada per a 2 cossos, augmentar el nombre d'aquests ja no serà problema de programació, sinó qüestió de la capacitat de processament del dispositiu utilitzat per a la simulació. Tot i així, és un pas molt important, ja que permetrà fer simulacions de cossos molt més complexes i fidels a la realitat, ja que els objectes d'estudi no viuen en aïllament, sinó que interactuen en el seu medi.

5.2.3 Cossos diferents

Per altra banda, les col·lisions no són l'únic que es podria implementar en un treball futur, sinó que també es podria considerar l'ús de cossos diferents.

En el funcionament actual del programa, el cos es construeix a partir d'un polígon regular, però, en la realitat no tots els cossos tous tenen forma d'esfera. Al contrari, hi ha una gran varietat de formes que prenen. Així doncs, si es modifiquen les posicions inicials dels punts que formen l'esquelet del cos, modificant les longituds dels molls, i inclús afegint un nou tipus de restricció angular que faça referència als angles que poden arribar a formar dos vèrtexs consecutius, es podrien obtindre formes noves. Amb aquestes es podrien obtindre simulacions més properes a la realitat, ja que es podria obtindre una silueta inicial del cos més fidel a l'objecte d'estudi.

Bibliografia

[1] Argonaut, *Simulating soft body animals*, Vídeo, YouTube, 2025.

[2] J.V. Beltran i J. Monterde, *Fundamentos Geométricos del Diseño por Ordenador*, Universitat de València, 2024.

[3] S.L. Campbell i C.D. Meyer Jr., *Generalized Inverses of Linear Transformations*, New York: Dover, 1921.

[4] P. Cordero S., *Métodos numéricos*, Universidad de Chile, 2013.

[5] B. Flückiger, *Computer-Animation II: Verfahren*, Lèxic cinematogràfic, Universitats de Hagen, Kiel, Luxemburg, Münster i Zürich.

[6] Godot Engine, *docs.godotengine.org*, Documentació Godot Engine, 2014.

[7] F. Holmér, *The beauty of Bézier curves*, Vídeo, YouTube, 2021.

[8] C. Lubich, *Numerik I*, Eberhard-Karls-Universtität Tübingen, 2006.

[9] B. K. Nikolic, *Euler-Cromer Method*, University of Delaware, 2021.

[10] ORamaVR, *docs.oramavr.com*, Documentació MAGES SDK, 2021.

[11] Processing, *processing.org*, Documentació Processing, 2001.

[12] Python, *Splines in Euclidean Space and Beyond*, Documentació mòdul splines, Python, 2024.

[13] D. Shiffman, *The nature of code: simulating natural systems with javascript*, No Starch Press, 2024.

[14] C. Twigg, *Catmull-Rom splines*, Carnegie Mellon University, 2003.

Annexos

Annex I. Glossari

Software:

El *software* o programari és el conjunt de procediments lògics que realitza un ordinador o sistema informàtic per a complir una tasca. Generalment, aquests processos s'emmagatzemen en programes.

Hardware:

El *hardware* o maquinari d'un ordinador o sistema informàtic és la part física tangible del sistema, les components mecàniques, elèctriques i electròniques.

Motor de física:

S'utilitzen principalment en desenvolupament de videojocs. Un motor de física (o *physics engine* en anglès) és un tipus de programari que permet realitzar simulacions d'algun sistema físic. Per exemple, la deformació d'un cos tou.

Sistema de partícules:

Un sistema de partícules és un programari dissenyat per a poder animar una gran quantitat d'objectes. Generalment s'utilitzen per animar comportaments de llum, foc, fum o similars.

Comportament de bandada:

Un comportament de bandada és el que tenen sovint grups grans animals que es desplacen junts, per exemple una bandada d'ocells o un banc de peixos.

Java:

És un llenguatge de programació basat en *C* disenyat per James Gosling i Sun Microsystems en 1995. És un llenguatge compilat orientat a objectes.

Input/Output:

Un *input* o entrada és una peça d'informació que es transmet de fora a un programari. Pot ser un *click* d'una tecla en un teclat o d'un ratolí, una imatge, un valor d'una variable o inclús una petjada dactilar. En contraposició, un *output* o sortida és una peça d'informació que es transmet de dins d'un programari a fora.

Annex II. Codi *Processing*

El codi d'aquest annex es pot trobar a GitHub.

Codi del programa Granota

```
//Definim les constants
PVector g = new PVector(0, 0.75); //Gravetat
PVector f = new PVector(0.95, 0.95); //Fregament
PVector v0 = new PVector(0,-5); //Velocitat inicial
float reescala = 0.05;
int costats = 10;
PVector[] anglesc = {new PVector(0, PI), new PVector(-PI, 0), new
    PVector(0, PI), new PVector(-PI, 0)}; //Angles minims i maxims
    formats per els muscles i colzes de cada pota
PVector[] angles = {new PVector(-PI/12, PI/12), new PVector(-PI/12,
    PI/12), new PVector(-PI/3, PI/3), new PVector(-PI/3, PI/3)}; //
    Angles minims i maxims formats per els colzes i canells de cada
    pota
PVector[] longit = {new PVector(40, 40), new PVector(80, 80)}; //
    Longituds dels segments de les potes, primer les de les potes de
    davant, segon les de les potes de darrere
boolean ocupat = false; //true si la granota esta sent agafada,
    false sino
PImage fons; //Imatge de fons
float velmax = 20; //Moviment maxim que pot fer un cos cada vegada
    que es mou

//Inicialitzem la granota
Granota figura = new Granota(costats, g, f, reescala, anglesc,
    angles, longit, new PVector(width/2,height/2), v0, velmax);

//Definim el marc i pujem les imatges necessaries
void setup() {
  size(640*2, 360*2);
  fons = loadImage("panta.png");
  PImage icona = loadImage("cap_granota.png");
  surface.setIcon(icona);
  surface.setTitle("Granota");
  strokeWeight(4);
  figura.inicialitza();
}

//Cada instant de temps s'executa aquest bloc de codi
void draw() {
  image(fons,0,0);
```

```
31    figura.actualitza();
32  }
33
34  //Agafa la granota si el cursor esta dintre del seu cos
35  void mousePressed() {
36    if (!ocupat){
37      if (figura.cos.estadins(mouseX, mouseY)){
38        if (mouseButton == LEFT) {
39          ocupat = true;
40          figura.cos.mantindreagafat(new PVector(mouseX,mouseY));
41        }
42      } else {
43      }
44    }
45  }
46
47  //Allibera la granota si estava sent agafada
48  void mouseReleased() {
49    if (mouseButton == LEFT && ocupat) {
50      ocupat = false;
51      figura.cos.allibera();
52    }
53  }
54
55  //Crea una nova granota suprimint la que ja existia i mantin-la
      agafada
56  void keyPressed() {
57    if (key == ' ' && !ocupat) {
58      figura.cos.inicialitza(new PVector(mouseX,mouseY), new PVector
          (0,0));
59      ocupat = true;
60      figura.cos.mantindreagafat(new PVector(mouseX,mouseY));
61    }
62  }
63
64  //Allibera la nova granota
65  void keyReleased() {
66    if (key == ' ' && ocupat) {
67      ocupat = false;
68      figura.cos.allibera();
69    }
70  }
71
72  //La classe vertex guarda un vector de posicio i un vector de
      posicio anterior
73  //Aixi podem aplicar la integracio de Verlet internament
```

```
74 class Vertex{
75   PVector posicio, posicioant; //Les posicions actual i anterior del
        vertex respectivament
76   float velmax; //Espai maxim que es pot moure el vertex en una
       iteracio de la integracio de Verlet
77
78   //Inicialitzem un vertex assignant valors als distints parametres
79   Vertex(PVector s0, PVector s1, float v){
80     posicio = s1;
81     posicioant = s0;
82     velmax = v;
83   }
84
85   //Fitem la posicio actual dins un marc escollit
86   void fita (float xmin, float xmax, float ymin, float ymax) {
87     if (posicio.x < xmin) {
88       posicio.x = xmin;
89     } else if (posicio.x > xmax) {
90       posicio.x = xmax;
91     }
92     if (posicio.y < ymin) {
93       posicio.y = ymin;
94     }else if (posicio.y > ymax) {
95       posicio.y = ymax;
96     }
97   }
98
99   //Fitem la posicio anterior dins un marc escollit
100  void fitaant (float xmin, float xmax, float ymin, float ymax) {
101    if (posicioant.x < xmin) {
102      posicioant.x = xmin;
103    } else if (posicioant.x > xmax) {
104      posicioant.x = xmax;
105    }
106    if (posicioant.y < ymin) {
107      posicioant.y = ymin;
108    }else if (posicioant.y > ymax) {
109      posicioant.y = ymax;
110    }
111  }
112
113  //Calcul de la nova posicio d'un vertex utilitzant integracio de
       Verlet
114  void Verlet (PVector fr, PVector gr) {
115    PVector pos_nou = new PVector(posicio.x + fr.x*(posicio.x -
         posicioant.x) + gr.x, posicio.y + fr.y*(posicio.y -
```

```
        posicioant.y) + gr.y);
116     //Abans d'aplicar el canvi, restringim la velocitat maxima per a
          que no divergisca
117     float xmax=posicio.x+velmax;
118     float xmin=posicio.x-velmax;
119     float ymax=posicio.y+velmax;
120     float ymin=posicio.y-velmax;
121     if (pos_nou.x < xmin) {
122       pos_nou.x = xmin;
123     } else if (pos_nou.x > xmax) {
124       pos_nou.x = xmax;
125     }
126     if (pos_nou.y < ymin) {
127       pos_nou.y = ymin;
128     }else if (pos_nou.y > ymax) {
129       pos_nou.y = ymax;
130     }
131     posicioant = new PVector(posicio.x, posicio.y);
132     posicio = new PVector(pos_nou.x, pos_nou.y);
133   }
134 
135   //Aplica un vector a un punt del pla per obtindre un nou punt
136   void Desp (PVector mov) {
137     posicioant = new PVector(posicioant.x + mov.x, posicioant.y +
        mov.y);
138     posicio = new PVector(posicio.x + mov.x, posicio.y+mov.y);
139   }
140 }
141 
142 class Poligon{
143   int r, N; //Distancia del centre del poligon als vertexs i nombre
        total de vertexs
144   float x, y, Aloc, Areal; //Coordenades horiztontal i vertical del
        centre, area actual del poligon, area que volem que tinga el
        poligon
145   float ang, C; //Variable auxiliar de l'angle per a crear els
        vertexs, constant auxiliar per a les correccions de l'area
146   PVector grav, freg; //Gravetat i fregament
147   Vertex[] vert; //Llistat de vertexs del poligon
148   Vertex centre; //Centre del poligon
149   float[] molls; //Llistat de distancies desitjades entre els
        vertexs consecutius del poligon
150   float[] estruc; //Llistat de distancies desitjades del punt des d'
        on esta agafat el poligon fins als vertexs ancores (actiu quan
        el poligon esta sent agafat)
```

```
151   int[] ancores; //Llistat de vertexs que actuen com a ancores (
        actiu quan el poligon esta sent agafat)
152   boolean agafat; //true si el poligon esta sent agafat, false sino
153   float velmax; //Espai maxim que es pot moure cada vertex del
        poligon per a corregir la seua posicio
154
155   //Quan creem el poligon cal guardar les constants: costats,
        gravetat, fregament
156   Poligon (int s, PVector g, PVector f, float c, float v){
157     N = s;
158     grav = g;
159     freg = f;
160     C = c;
161     velmax=v;
162   }
163
164   //Inicialitzem el poligon
165   void inicialitza(PVector center, PVector v_ini){
166     //Inicialitzem el vertex central i els vectors de vertexs i
          molls
167     centre = new Vertex(center, v_ini, velmax);
168     vert = new Vertex[N+1];
169     molls = new float[N+1];
170     agafat = false;
171
172     //Construim els vertexs del perimetre del poligon anant en
          sentit antihorari
173     x = center.x;
174     y = center.y;
175     r = abs(height/6);
176     ang = -HALF_PI;
177     for (int i = 0; i < N; i++){
178       vert[i] = new Vertex(new PVector(x + r*cos(ang)-v_ini.x, y + r
            *sin(ang)-v_ini.y),new PVector(x + r*cos(ang), y + r*sin(
            ang)), velmax);
179       ang = ang + TWO_PI/N;
180     }
181     vert[N] = new Vertex(new PVector(x + r*cos(-HALF_PI)-v_ini.x, y
          + r*sin(-HALF_PI)-v_ini.y),new PVector(x + r*cos(-HALF_PI), y
           + r*sin(-HALF_PI)), velmax);
182
183     //Creem els molls entre els vertexs del poligon
184     for (int i = 0; i < N; i++){
185       molls[i]=1;
186     }
187
```

```
188     //Comprovem que tots els punts estiguen dins la pantalla
189     for (int i=0; i<N+1; i++){
190       vert[i].fita(0, width-1, 0, height-1);
191       vert[i].fitaant(0, width-1, 0, height-1);
192     }
193 
194     //Valor de l'area que volem fixar
195     Areal = 20000;
196   }
197 
198   //Necessitem una funcio per calcular l'area en tot moment
199   float calculaarea(Vertex[] ver, int n){
200     //Seguim la formula del trapezoide
201     float L,W;
202     float area = 0;
203     for (int i=0; i<n; i++){
204       L=ver[i].posicio.x-ver[i+1].posicio.x;
205       W=(ver[i].posicio.y+ver[i+1].posicio.y)/2;
206       area = area + L*W;
207     }
208     return area;
209   }
210 
211   //Dibuixem el cos utilitzant una corba de Rom-Catmull
212   void dibuixar(){
213     beginShape();
214     fill(48, 161, 48);
215     curveVertex(vert[0].posicio.x, vert[0].posicio.y);
216     for (int i = 0; i <= N ; i++){
217       curveVertex(vert[i].posicio.x, vert[i].posicio.y);
218     }
219     curveVertex(vert[N].posicio.x, vert[N].posicio.y);
220     endShape();
221   }
222 
223   //Aquest procediment es repeteix cada instant de temps
224   void actualitza() {
225     float F; //Factor de correccio de l'area
226     PVector[] correccions = new PVector[N+1]; //Vector de
           correccions de cada vertex
227     int[] num_correccions = new int[N+1]; //Nombre de correccions
           aplicades a cada vertex en l'instant
228 
229     PVector[] punt_a = new PVector[N+1]; //Vector de posicions
           teoriques segons el tamany dels molls calculats en sentit
           antihorari
```

```
230     PVector[] punt_b = new PVector[N+1]; //Vector de posicions
          teoriques segons el tamany dels molls calculats en sentit
          horari
231
232     //Actualitzem primer la posicio amb integracio de verlet
233     for (int i = 0; i < N+1; i++){
234       vert[i].Verlet(freg, grav);
235       correccions[i] = new PVector(0,0);
236       num_correccions[i] = 0;
237     }
238
239     //Calculem les correccions pels molls
240     for (int i=0; i<N; i++){
241       PVector puntmig = new PVector((vert[i].posicio.x+vert[i+1].
            posicio.x)/2, (vert[i].posicio.y+vert[i+1].posicio.y)/2);
242       PVector direc = new PVector(vert[i].posicio.x-vert[i+1].
            posicio.x, vert[i].posicio.y-vert[i+1].posicio.y);
243       direc.normalize();
244       punt_a[i] = new PVector(puntmig.x + molls[i]*0.5*direc.x,
            puntmig.y + molls[i]*0.5*direc.y);
245       punt_b[i+1] = new PVector(puntmig.x - molls[i]*0.5*direc.x,
            puntmig.y - molls[i]*0.5*direc.y);
246     }
247     punt_a[N] = new PVector(punt_a[0].x, punt_a[0].y);
248     punt_b[0]= new PVector(punt_b[N].x, punt_b[N].y);
249
250     //Apliquem les correccions 5 voltes per a que convergeixca mes
          rapid
251     for (int j=0; j<5; j++) {
252       for (int i=0; i<N+1; i++){
253         if (punt_a[i] != vert[i].posicio){
254           PVector correc = new PVector(punt_a[i].x-vert[i].posicio.x
                , punt_a[i].y-vert[i].posicio.y);
255           correccions[i] = new PVector(correccions[i].x + correc.x,
                correccions[i].y + correc.y);
256           num_correccions[i] = num_correccions[i]+1;
257         }
258         if (punt_b[i] != vert[i].posicio){
259           PVector correc = new PVector(punt_b[i].x-vert[i].posicio.x
                , punt_b[i].y-vert[i].posicio.y);
260           correccions[i] = new PVector(correccions[i].x + correc.x,
                correccions[i].y + correc.y);
261           num_correccions[i] = num_correccions[i]+1;
262         }
263       }
264     }
```

```
265
266     //Si esta agafat calculem les correccions per l'estructura
267     if (agafat) {
268       PVector[] punt_c = new PVector[N+1];
269       for (int i : ancores){
270       PVector direc = new PVector(vert[i].posicio.x-mouseX, vert[i].
            posicio.y-mouseY);
271       direc.normalize();
272       punt_c[i] = new PVector(mouseX + estruc[i]*direc.x, mouseY +
            estruc[i]*direc.y);
273       }
274       for (int i : ancores){
275         if (punt_c[i] != vert[i].posicio){
276           PVector correc = new PVector(punt_c[i].x-vert[i].posicio.x
                , punt_c[i].y-vert[i].posicio.y);
277           correccions[i] = new PVector(correccions[i].x + correc.x,
                correccions[i].y + correc.y);
278           num_correccions[i] = num_correccions[i]+1;
279         }
280       }
281     }
282
283
284     //Calculem l'area actual
285     Aloc = calculaarea(vert,N);
286     //Comparem l'area teorica en l'actual i procedim segons el
          resultat
287     if (Areal != Aloc){
288       F = C*(Areal - Aloc)/N;
289       for (int j=0; j<5; j++){
290         for (int i = 1; i < N; i++){
291           PVector prov = new PVector(vert[i+1].posicio.y-vert[i-1].
                posicio.y, -vert[i+1].posicio.x+vert[i-1].posicio.x);
292           prov.normalize();
293           correccions[i] = new PVector(correccions[i].x + F*prov.x,
                correccions[i].y + F*prov.y);
294           num_correccions[i] = num_correccions[i]+1;
295         }
296         PVector prov = new PVector(vert[1].posicio.y-vert[N-1].
              posicio.y, -vert[1].posicio.x+vert[N-1].posicio.x);
297         prov.normalize();
298         correccions[0] = new PVector(correccions[0].x + F*prov.x,
              correccions[0].y + F*prov.y);
299         num_correccions[0] = num_correccions[0]+1;
300         correccions[N] = new PVector(correccions[N].x + F*prov.x,
              correccions[N].y + F*prov.y);
```

```
301          num_correccions[N] = num_correccions[N]+1;
302        }
303      }
304  
305      //Corregim la posicio de les ancores si estan massa lluny del
           punt des d'on s'esta agafant la figura
306      if (agafat){
307        for (int i:ancores) {
308          if (i>0 && i<N){
309            if (vert[i].posicio.x < mouseX-10) {
310              correccions[i] = new PVector(correccions[i].x + (mouseX
                   -10)-vert[i].posicio.x, correccions[i].y);
311            } else if (vert[i].posicio.x > mouseX+10) {
312              correccions[i] = new PVector(correccions[i].x + (mouseX
                   +10)-vert[i].posicio.x, correccions[i].y);
313            }
314            if (vert[i].posicio.y < 0) {
315              correccions[i] = new PVector(correccions[i].x,
                   correccions[i].y - vert[i].posicio.y);
316            } else if (vert[i].posicio.y > mouseY-1) {
317              correccions[i] = new PVector(correccions[i].x,
                   correccions[i].y + (mouseY-1) - vert[i].posicio.y);
318            }
319          } else {
320            if (vert[i].posicio.x < 0) {
321              correccions[i] = new PVector(correccions[i].x -vert[i].
                   posicio.x, correccions[i].y);
322            } else if (vert[i].posicio.x > width-1) {
323              correccions[i] = new PVector(correccions[i].x + (width
                   -1)-vert[i].posicio.x, correccions[i].y);
324            }
325            if (vert[i].posicio.y < 0) {
326              correccions[i] = new PVector(correccions[i].x,
                   correccions[i].y - vert[i].posicio.y);
327            } else if (vert[i].posicio.y > mouseY-10) {
328              correccions[i] = new PVector(correccions[i].x,
                   correccions[i].y + (mouseY-10) - vert[i].posicio.y);
329            }
330          }
331        }
332      }
333  
334      //Sumem totes les correccions a la posicio (Aquest bloc fa que
           en si comen a en alguns punts de la pantalla la figura
           desaparega immediatament?)
335      for (int i = 0; i < N+1; i++){
```

```
336       if (num_correccions[i] >=1){
337         //Restringim el maxim que es pot moure el cos per unitat de
               temps
338         PVector modificacions = new PVector(correccions[i].x/
               num_correccions[i], correccions[i].y/num_correccions[i]);
339         if (modificacions.x < -velmax) {
340           modificacions.x = -velmax;
341         } else if (modificacions.x > velmax) {
342           modificacions.x = velmax;
343         }
344         if (modificacions.y < -velmax) {
345           modificacions.y = -velmax;
346         } else if (modificacions.y > velmax) {
347           modificacions.y = velmax;
348         }
349         vert[i].posicio = new PVector(vert[i].posicio.x +
               modificacions.x, vert[i].posicio.y + modificacions.y);
350         vert[i] = new Vertex(vert[i].posicioant, vert[i].posicio,
               velmax);
351       }
352     }
353
354     //Corregim la posicio per si la figura se n'ix del marc
355     for (int i=0; i<N+1; i++){
356       vert[i].fita(0, width-1, 0, height-1);
357     }
358   }
359
360   //Fixa les ancores desde les quals es mante agafat el poligon,
         aixi com el clau des del que es mante
361   void mantindreagafat (PVector clau) {
362     agafat = true;
363     ancores = new int[4];
364     ancores[0] = 0;
365     ancores[1] = 1;
366     ancores[2] = N-1;
367     ancores[3] = N;
368     estruc = new float[N+1];
369     for (int i : ancores){
370       estruc[i] = dist(vert[i].posicio.x, vert[i].posicio.y, clau.x,
             clau.y);
371     }
372   }
373
374   //Deixa de mantindre subjectat el poligon
375   void allibera () {
```

```
376      agafat = false;
377    }
378
379    //Comprovem si les coordenades estan dintre del poligon utilitzant
         un metode similar a la regla del trapezoide per a l'area
380    boolean estadins (float posx, float posy){
381      boolean dins = false;
382      for (int i=0; i<N; i++){
383        if ( ((vert[i].posicio.x <= posx)&&(vert[i+1].posicio.x >=
             posx)) || ((vert[i].posicio.x >= posx)&&(vert[i+1].posicio.
             x <= posx)) ){
384          if (vert[i].posicio.x != vert[i+1].posicio.x) {
385             float t = (posx-vert[i].posicio.x)/(vert[i+1].posicio.x -
                    vert[i].posicio.x);
386             float y = vert[i].posicio.y + t*(vert[i+1].posicio.y -
                   vert[i].posicio.y);
387             if (posy <= y) {
388               dins = !dins;
389             }
390          } else {
391            if ( posy <= vert[i].posicio.y && posy <= vert[i+1].
                 posicio.y ){
392              dins = !dins;
393            }
394          }
395        }
396      }
397      return dins;
398    }
399  }
400
401  class Pota{
402    PVector muscle; //Aquest vector correspon al clau
403    Vertex colze, canell; //Les altres articulacions
404    PVector grav, freg; //Gravetat i fregament
405    float moll1, moll2; //Longituts entre muscle i colze, i colze i
         canell
406    float angmax1, angmin1; //Angles maxim i minim entre el muscle i
         el colze
407    float angmax2, angmin2; //Angles maxim i minim entre el colze i el
           canell
408    float factor; //Factor que indica el grossor de les potes
409
410    //Inicialitzem una pota assignant valors als distints parametres
411    Pota (PVector[] punts, PVector ang1, PVector ang2, PVector g,
         PVector f, float v){
```

```
412     muscle = new PVector(punts[0].x, punts[0].y);
413     colze = new Vertex(new PVector(punts[1].x, punts[1].y), new
          PVector(punts[1].x, punts[1].y), v);
414     canell = new Vertex(new PVector(punts[2].x, punts[2].y), new
          PVector(punts[2].x, punts[2].y), v);
415     angmin1 = ang1.x;
416     angmax1 = ang1.y;
417     angmin2 = ang2.x;
418     angmax2 = ang2.y;
419     moll1 = dist(punts[0].x, punts[0].y, punts[1].x, punts[1].y);
420     moll2 = dist(punts[1].x, punts[1].y, punts[2].x, punts[2].y);
421     grav = new PVector(g.x, g.y);
422     freg = new PVector(f.x, f.y);
423     factor = 7;
424   }
425
426   //Dibuixem la pota
427   void dibuixar(){
428
429     //Obtenim els vectors perpendiculars a cada segment de les potes
430     PVector direc1 = new PVector(colze.posicio.x-muscle.x, colze.
          posicio.y-muscle.y);
431     direc1.normalize();
432     PVector direc2 = new PVector(canell.posicio.x-colze.posicio.x,
          canell.posicio.y-colze.posicio.y);
433     direc2.normalize();
434
435     //Utilitzant les corbes de Rom-Catmull integrades ja al
          programari, dibuixem les vores de la pota
436     beginShape();
437     fill(48, 161, 48);
438     curveVertex(muscle.x+direc1.y*factor, muscle.y-direc1.x*factor);
439     curveVertex(muscle.x+direc1.y*factor, muscle.y-direc1.x*factor);
440     curveVertex(colze.posicio.x+(direc1.y+direc2.y)*factor/2, colze.
          posicio.y-(direc1.x+direc2.x)*factor/2);
441     curveVertex(canell.posicio.x+direc2.y*factor, canell.posicio.y-
          direc2.x*factor);
442     curveVertex(canell.posicio.x-direc2.y*factor, canell.posicio.y+
          direc2.x*factor);
443     curveVertex(colze.posicio.x-(direc1.y+direc2.y)*factor/2, colze.
          posicio.y+(direc1.x+direc2.x)*factor/2);
444     curveVertex(muscle.x-direc1.y*factor, muscle.y+direc1.x*factor);
445     curveVertex(muscle.x+direc1.y*factor, muscle.y-direc1.x*factor);
446     endShape();
447   }
448
```

```
449   //Aquest procediment es repeteix cada instant de temps
450   void actualitza(PVector mov){
451     muscle = new PVector(mov.x, mov.y);
452     colze.Verlet(freg,grav);
453     canell.Verlet(freg,grav);
454 
455     //Primer reajustem distancies des del muscle fins al colze
456     if (dist(muscle.x, muscle.y, colze.posicio.x, colze.posicio.y)
          != moll1) {
457       PVector direc = new PVector(colze.posicio.x-muscle.x, colze.
            posicio.y-muscle.y);
458       direc.normalize();
459       PVector posreal = new PVector(muscle.x + direc.x*moll1, muscle
            .y + direc.y*moll1);
460       PVector movi = new PVector(posreal.x-colze.posicio.x, posreal.
            y-colze.posicio.y);
461       colze.Desp(movi);
462     }
463 
464     //A continuacio reajustem l'angle que formen el muscle i el
          colze
465     if (muscle.x-colze.posicio.x >= 0) {
466       if (acos((colze.posicio.y-muscle.y)/(dist(muscle.x, muscle.y,
            colze.posicio.x, colze.posicio.y))) > angmax1){
467         colze.posicio = new PVector(muscle.x-sin(angmax1)*moll1,
              muscle.y+cos(angmax1)*moll1);
468       }
469     } else {
470       if (-acos((colze.posicio.y-muscle.y)/(dist(muscle.x, muscle.y,
             colze.posicio.x, colze.posicio.y))) < angmin1){
471         colze.posicio = new PVector(muscle.x-sin(angmin1)*moll1,
              muscle.y+cos(angmin1)*moll1);
472       }
473     }
474 
475     //Ara reajustem distancies des del colze fins al canell
476     if (dist(colze.posicio.x, colze.posicio.y, canell.posicio.x,
          canell.posicio.y) != moll2) {
477       PVector direc = new PVector(canell.posicio.x-colze.posicio.x,
            canell.posicio.y-colze.posicio.y);
478       direc.normalize();
479       PVector posreal = new PVector(colze.posicio.x + direc.x*moll2,
             colze.posicio.y + direc.y*moll2);
480       PVector movi = new PVector(posreal.x-canell.posicio.x, posreal
            .y-canell.posicio.y);
481       canell.Desp(movi);
```

```
482     }
483
484     //A continuacio reajustem l'angle que formen el colze i el
          canell
485     if (colze.posicio.x-canell.posicio.x >= 0) {
486       if (acos((canell.posicio.y-colze.posicio.y)/(dist(colze.
            posicio.x, colze.posicio.y, canell.posicio.x, canell.
            posicio.y))) > angmax2){
487         canell.posicio = new PVector(colze.posicio.x-sin(angmax2)*
              moll2, colze.posicio.y+cos(angmax2)*moll2);
488       }
489     } else {
490       if (-acos((canell.posicio.y-colze.posicio.y)/(dist(colze.
            posicio.x, colze.posicio.y, canell.posicio.x, canell.
            posicio.y))) < angmin2){
491         canell.posicio = new PVector(colze.posicio.x-sin(angmin2)*
              moll2, colze.posicio.y+cos(angmin2)*moll2);
492       }
493     }
494
495     //Corregim la posicio del canell si se n'ix del marc
496     if (canell.posicio.y > height-1) {
497       float noux;
498       float dy = dist(canell.posicio.x, canell.posicio.y, canell.
            posicio.x, height-1);
499       float a = 1;
500       float b = -2*(muscle.x);
501       float c = (muscle.x)*(muscle.x) + (muscle.y-(colze.posicio.y-
            dy))*(muscle.y-(colze.posicio.y-dy)) -moll1*moll1;
502       if (colze.posicio.x >= muscle.x) {
503         noux = (-b+sqrt(b*b-4*a*c))/(2*a);
504       } else {
505         noux = (-b-sqrt(b*b-4*a*c))/(2*a);
506       }
507       float dx = noux - colze.posicio.x;
508       canell.posicio = new PVector (canell.posicio.x + dx, canell.
            posicio.y - dy);
509       colze.posicio = new PVector (colze.posicio.x+dx, colze.posicio
            .y - dy);
510     }
511   }
512 }
513
514 class Granota{
515   Pota dretadavant, dretadarrere, esquerradavant, esquerradarrere;
        //Les 4 potes
```

```
516   Poligon cos; //El cos de la granota
517   PImage cap; //La imatge del cap
518
519   //Inicialitzem una granota assignant valors als distints
        parametres i inicialitzant cadascuna de les parts que la
        composen
520   Granota (int s, PVector g, PVector f, float c, PVector[] angles1,
        PVector[] angles2, PVector[] longit, PVector centre, PVector
        v_ini, float v_max){
521     cos = new Poligon(s,g,f,c,v_max);
522     cos.inicialitza(centre, v_ini);
523     PVector[] punts = {centre, new PVector(centre.x+longit[0].x,
          centre.y), new PVector(centre.x+longit[0].x+longit[0].y,
          centre.y)};
524     dretadavant = new Pota(punts, angles1[0], angles2[0],g,f,v_max);
525     esquerradavant = new Pota(punts, angles1[1], angles2[1],g,f,
          v_max);
526     PVector[] punts2 = {centre, new PVector(centre.x+longit[1].x,
          centre.y), new PVector(centre.x+longit[1].x+longit[1].y,
          centre.y)};
527     dretadarrere = new Pota(punts2, angles1[2], angles2[2],g,f,v_max
          );
528     esquerradarrere = new Pota(punts2, angles1[3], angles2[3],g,f,
          v_max);
529   }
530
531   //Carreguem la imatge del cap de la granota en el programa
532   void inicialitza(){
533     cap = loadImage("cap_granota.png");
534   }
535
536   //Dibuixem la granota dibuixant les seues parts en l'ordre
        apropiat
537   void dibuixar(){
538     dretadarrere.dibuixar();
539     esquerradarrere.dibuixar();
540     cos.dibuixar();
541     dretadavant.dibuixar();
542     esquerradavant.dibuixar();
543     image(cap,cos.vert[0].posicio.x-(1109/16),cos.vert[0].posicio.y
          -(747/16),1109/8,747/8); //El tamany ha estat obtingut a
          traves de prova i error
544   }
545
546   //Aquest procediment es repeteix cada instant de temps
547   void actualitza(){
```

```
548     //Primer actualitzem el cos
549     cos.actualitza();
550
551     //Podem asumir ara ja que el poligon del cos te 10 vertex,
           calculem les posicions de les potes de davant
552     PVector posDreDav, posEsDav, posDreDar, posEsDar;
553     float desx = (cos.vert[8].posicio.x-cos.vert[2].posicio.x)/5;
554     float desy = (cos.vert[8].posicio.y-cos.vert[2].posicio.y)/5;
555     posEsDav = new PVector(cos.vert[2].posicio.x+desx, cos.vert[2].
           posicio.y+desy);
556     posDreDav = new PVector(cos.vert[8].posicio.x-desx, cos.vert[8].
           posicio.y-desy);
557     //Ara calculem les posicions de les potes de darrere
558     float desyd = (cos.vert[3].posicio.y-cos.vert[4].posicio.y)/2;
559     float desxd = (cos.vert[3].posicio.x-cos.vert[4].posicio.x)/2;
560     float desye = (cos.vert[7].posicio.y-cos.vert[6].posicio.y)/2;
561     float desxe = (cos.vert[7].posicio.x-cos.vert[6].posicio.x)/2;
562     posEsDar = new PVector(cos.vert[4].posicio.x+desxd, cos.vert[4].
           posicio.y+desyd);
563     posDreDar = new PVector(cos.vert[6].posicio.x+desxe, cos.vert
           [6].posicio.y+desye);
564     //Finalment actualitzem les potes
565     dretadavant.actualitza(posDreDav);
566     dretadarrere.actualitza(posDreDar);
567     esquerradavant.actualitza(posEsDav);
568     esquerradarrere.actualitza(posEsDar);
569
570     //Un cop tot esta al seu lloc, dibuixem la granota
571     dibuixar();
572   }
573 }
```

Annex III. Codi *Matlab*

El codi d'aquest annex es pot trobar a GitHub.

`Pol_Bernstein`

```
function [B] = Pol_Bernstein(i, n)
    coef = factorial(n)/(factorial(i)*factorial(n-i));
    if i==0
        B = @(t) coef*((1-t).^(n));
    elseif i==n
            B = @(t) coef*(t.^n);
    else
        B = @(t) coef*(t.^i).*((1-t).^(n-i));
    end
end
```

`Rep_Pol_Bernstein`

```
function [] = Rep_Pol_Bernstein(n,tots)

t=linspace(0,1,200);
if tots
    for m=0:n
        figure
        for i=0:m
            B=Pol_Bernstein(i,m);
            plot(t,B(t))
            hold on
        end
        axis equal
        axis padded
    end
else
    figure
    for i=0:n
        B=Pol_Bernstein(i,n);
        plot(t,B(t))
        hold on
    end
    axis equal
    axis padded
end
```

De_Casteljau

```
function [alpha] = De_Casteljau(P,part)
    n = length(P);
    alpha=zeros(part+1,1);
    for k=1:(part+1)
        t=(k-1)/part;
        inter=zeros(n);
        for i=1:n
            inter(1,i)= P(i);
        end
        for r=2:n
            for j=1:(n-r+1)
                inter(r,j) = (1-t)*inter(r-1,j) + t*inter(r-1,j+1);
            end
        end
        alpha(k) = inter(n,1);
    end
end
```

Rep_Bezier

```
function [coordY,coordX] = Rep_Bezier(P, part, desglossar)
    figure
    if desglossar
        for i=1:length(P(:,1))
            coordX=De_Casteljau(P(1:i,1), part);
            coordY=De_Casteljau(P(1:i,2), part);
            plot(coordX, coordY)
            hold on
        end
    else
        coordX=De_Casteljau(P(:,1), part);
        coordY=De_Casteljau(P(:,2), part);
        plot(coordX, coordY)
        hold on
    end
    axis equal
    axis padded
end
```

BezParamEqui

```
function [P] = BezParamEqui(Q,n)
    k=length(Q);
    M=zeros(k,n);
    for i=1:k
        t=(i-1)/(k-1);
        for j=1:n
            B=Pol_Bernstein(j-1,n-1);
            M(i,j)=B(t);
        end
    end
    Mt=M.';
    A=Mt*M;
    B=Mt*Q;
    P=A\B;
end
```

Rep_BezParamEqui

```
function [coordY,coordX] = Rep_BezParamEqui(Q,n,part,desglossar)
    Px=BezParamEqui(Q(:,1),n);
    Py=BezParamEqui(Q(:,2),n);
    P=[Px,Py];
    [coordY,coordX] = Rep_Bezier(P,part,desglossar);
    for i=1:length(Q(:,1))
        hold on
        plot(Q(i,1),Q(i,2),"o")
    end
end
```

BezParamProp

```
function [P] = BezParamProp(Q,n)
    k=length(Q);
    M=zeros(k,n);
    dist=zeros(k-1,1);
    disttot=zeros(k-1,1);
    for i=1:k-1
        dist(i)=sqrt((Q(i,1)^2+(Q(i,2)^2)));
        if i==1
            disttot(i)=dist(i);
        else
            disttot(i)=disttot(i-1)+dist(i);
        end
```

```
13      end
14      for i=1:k
15          if i==1
16              t=0;
17          else
18              t=disttot(i-1)/disttot(k-1);
19          end
20          for j=1:n
21              B=Pol_Bernstein(j-1,n-1);
22              M(i,j)=B(t);
23          end
24      end
25      Mt=M.';
26      A=Mt*M;
27      B1=Mt*Q(:,1);
28      P1=A\B1;
29      B2=Mt*Q(:,2);
30      P2=A\B2;
31      P=[P1,P2];
32  end
```

Rep_BezParamProp

```
1  function [coordY,coordX] = Rep_BezParamProp(Q,n,part,desglossar)
2      P=BezParamProp(Q,n);
3      [coordY,coordX] = Rep_Bezier(P,part,desglossar);
4      for i=1:length(Q(:,1))
5          hold on
6          plot(Q(i,1),Q(i,2),"o")
7      end
8  end
```

SegmentCatmullRom

```
1  function [alpha] = SegmentCatmullRom(p0,p1,p2,p3,tau,part)
2      alpha=zeros(part+1,1);
3      c0=p1;
4      c1=-tau*p0+tau*p2;
5      c2=2*tau*p0+(tau-3)*p1+(3-2*tau)*p2-tau*p3;
6      c3=-tau*p0+(2-tau)*p1+(tau-2)*p2+tau*p3;
7      for k=1:(part+1)
8          t=(k-1)/part;
9          alpha(k) = c0+c1*t+c2*t^2+c3*t^3;
```

```
        end
end
```

Rep_SegmentCatmullRom

```
function [coordY, coordX] = Rep_SegmentCatmullRom(p0,p1,p2,p3,tau,
    part,puntsvis)
    figure
    if puntsvis
        plot(p0(1),p0(2),"o")
        hold on
        plot(p1(1),p1(2),"o")
        hold on
        plot(p2(1),p2(2),"o")
        hold on
        plot(p3(1),p3(2),"o")
        hold on
    end
    coordX=SegmentCatmullRom(p0(1), p1(1), p2(1), p3(1), tau, part);
    coordY=SegmentCatmullRom(p0(2), p1(2), p2(2), p3(2), tau, part);
    plot(coordX, coordY)
    hold on
    axis equal
    axis padded
end
```

Rep_SegmentCatmullRom_nodib

```
function [coordY, coordX] = Rep_SegmentCatmullRom_nodib(p0,p1,p2,p3,
    tau,part)
    coordX=SegmentCatmullRom(p0(1), p1(1), p2(1), p3(1), tau, part);
    coordY=SegmentCatmullRom(p0(2), p1(2), p2(2), p3(2), tau, part);
    plot(coordX, coordY)
    hold on
end
```

Rep_funcionsmescla

```
function [w0,w1,w2,w3] = Rep_funcionsmescla(tensio)
    w0 = @(t) -tensio.*t+2*tensio.*t.^2-tensio.*t.^3;
    w1 = @(t) 1+(tensio-3).*t.^2+(2-tensio).*t.^3;
    w2 = @(t) tensio.*t+(3-2*tensio).*t.^2+(tensio-2).*t.^3;
```

```
    w3 = @(t) -tensio.*t.^2+tensio.*t.^3;
    t=linspace(0,1);

    figure
    plot(t,w0(t));
    hold on
    plot(t,w1(t));
    hold on
    plot(t,w2(t));
    hold on
    plot(t,w3(t));
    hold on
    axis padded
end
```

Rep_CatmullRom

```
function [coordY, coordX] = Rep_CatmullRom(P,tau,part,puntvis,ciclic
    ,tancar)
    n=length(P(:,1));
    figure
    for i=2:(n-2)
        Rep_SegmentCatmullRom_nodib(P(i-1,:),P(i,:),P(i+1,:),P(i
            +2,:),tau,part);
    end
    if ciclic
        Rep_SegmentCatmullRom_nodib(P(n,:),P(1,:),P(2,:),P(3,:),tau,
            part);
        Rep_SegmentCatmullRom_nodib(P(n-2,:),P(n-1,:),P(n,:),P(1,:),
            tau,part);
        if tancar
            Rep_SegmentCatmullRom_nodib(P(n-1,:),P(n,:),P(1,:),P
                (2,:),tau,part);
        end
    else
        Rep_SegmentCatmullRom_nodib(P(1,:),P(1,:),P(2,:),P(3,:),tau,
            part);
        Rep_SegmentCatmullRom_nodib(P(n-2,:),P(n-1,:),P(n,:),P(n,:),
            tau,part);
    end
    if puntvis
        for i=1:n
        plot(P(i,1),P(i,2),"o")
        hold on
        end
```

```
22      end
23      axis equal
24      axis padded
25  end
```