

# Costruzione di una base per $\mathbb{S}_{3,1}(\Delta_{PS})$ : note teoriche e di implementazione

Andrea Favilli

29 Luglio 2020

## Sommario

Questo documento è da leggersi in parallelo alla presentazione "Superfici B-spline cubiche di Powell-Sabin". Nella prima parte presentiamo dimostrazioni o verifiche di alcuni fatti teorici citati nella trattazione, indispensabili ad una comprensione organica di teoria e procedimento ma pedanti ad una prima lettura, capaci di allontanare dal filone principale del discorso. La seconda parte è invece dedicata all'illustrazione del codice Python impiegato per la realizzazione delle immagini, allo scopo di dare un risvolto pratico all'intero lavoro.

## 1 Complementi alla trattazione

### 1.1 Note su polinomi bivariati e superfici di Bézier triangolari

**Nota 1.1.** Mostriamo l'uguaglianza

$$\mathbf{P} = \sum_{i+j+k=d} \mathbf{V}_{ijk} B_{ijk}^d(\mathbf{P}),$$

con  $\mathbf{V}_{ijk} := \frac{i}{d} \mathbf{V}_1 + \frac{j}{d} \mathbf{V}_2 + \frac{k}{d} \mathbf{V}_3$ .

Si tratta di iniziare dal termine a destra, esplicitare  $\mathbf{V}_{ijk}$  e effettuare un opportuno cambio di variabili.

$$\begin{aligned} \sum_{i+j+k=d} \mathbf{V}_{ijk} B_{ijk}^d(\mathbf{P}) &= \sum_{i+j+k=d} \mathbf{V}_{ijk} \frac{d!}{i!j!k!} \tau_1^i \tau_2^j \tau_3^k = \\ &= \sum_{i+j+k=d} \frac{i}{d} \frac{d!}{i!j!k!} \tau_1^i \tau_2^j \tau_3^k \mathbf{V}_1 + \sum_{i+j+k=d} \frac{j}{d} \frac{d!}{i!j!k!} \tau_1^i \tau_2^j \tau_3^k \mathbf{V}_2 + \sum_{i+j+k=d} \frac{k}{d} \frac{d!}{i!j!k!} \tau_1^i \tau_2^j \tau_3^k \mathbf{V}_3 = \\ &= \sum_{i+j+k=d} \frac{(d-1)!}{(i-1)!j!k!} \tau_1^i \tau_2^j \tau_3^k \mathbf{V}_1 + \sum_{i+j+k=d} \frac{(d-1)!}{i!(j-1)!k!} \tau_1^i \tau_2^j \tau_3^k \mathbf{V}_2 + \sum_{i+j+k=d} \frac{(d-1)!}{i!j!(k-1)!} \tau_1^i \tau_2^j \tau_3^k \mathbf{V}_3 = \end{aligned}$$

Operiamo le sostituzioni  $\iota := i - 1$ ,  $\chi := j - 1$ ,  $\lambda := k - 1$ .

$$= \tau_1 \mathbf{V}_1 \left( \sum_{\iota+\chi+\lambda=d-1} \frac{(d-1)!}{\iota!\chi!\lambda!} \tau_1^\iota \tau_2^\chi \tau_3^\lambda \right) + \tau_2 \mathbf{V}_2 \left( \sum_{\iota+\chi+\lambda=d-1} \frac{(d-1)!}{\iota!\chi!\lambda!} \tau_1^\iota \tau_2^\chi \tau_3^\lambda \right) +$$

$$\begin{aligned}
& + \tau_3 \mathbf{V}_3 \left( \sum_{i+j+\lambda=d-1} \frac{(d-1)!}{i!j!\lambda!} \tau_1^i \tau_2^j \tau_3^\lambda \mathbf{V}_3 \right) = \\
& = (\tau_1 \mathbf{V}_1 + \tau_2 \mathbf{V}_2 + \tau_3 \mathbf{V}_3) \underbrace{\left( \sum_{i+j+k=d-1} B_{ijk}^{d-1} \right)}_1 = \tau_1 \mathbf{V}_1 + \tau_2 \mathbf{V}_2 + \tau_3 \mathbf{V}_3 = \mathbf{P}.
\end{aligned}$$

**Nota 1.2.** Dobbiamo far vedere come si estende l'usuale condizione di raccordo  $C^1$  per superfici di Bézier triangolari al caso dei polinomi.

Ricordiamo la regola generale: se ho  $\sum_{i+j+k=d} \mathbf{P}_{ijk} B_{ijk}^d(\cdot)$  superficie di Bézier su  $\mathcal{T}_1 = \langle \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3 \rangle$  e  $\sum_{i+j+k=d} \widetilde{\mathbf{P}}_{ijk} B_{ijk}^d(\cdot)$  superficie di Bézier su  $\mathcal{T}_2 = \langle \mathbf{V}_2, \mathbf{V}_3, \mathbf{V}_4 \rangle$  allora le due superfici si raccordano in maniera  $C^1$  lungo  $\overline{\mathbf{V}_2 \mathbf{V}_3}$  se

$$\begin{cases} \widetilde{\mathbf{P}}_{jk0} = \mathbf{P}_{0jk} & \forall j, k \text{ t.c. } j+k = d, \\ \widetilde{\mathbf{P}}_{jk1} = \tau_1 \mathbf{P}_{1jk} + \tau_2 \mathbf{P}_{0(j+1)k} + \tau_3 \mathbf{P}_{0j(k+1)} & j+k = d-1, \end{cases}$$

dette  $(\tau_1, \tau_2, \tau_3)$  le coordinate baricentriche di  $\mathbf{V}_4$  rispetto a  $\mathcal{T}_1$ .

Nel caso dei polinomi ho punti di controllo della forma  $\mathbf{P}_{ijk} = (\mathbf{V}_{ijk}, b_{ijk})$ , dunque se mostro che comunque

$$\begin{cases} \widetilde{\mathbf{V}}_{jk0} = \mathbf{V}_{0jk} & \forall j, k \text{ t.c. } j+k = d, \\ \widetilde{\mathbf{V}}_{jk1} = \tau_1 \mathbf{V}_{1jk} + \tau_2 \mathbf{V}_{0(j+1)k} + \tau_3 \mathbf{V}_{0j(k+1)} & j+k = d-1, \end{cases}$$

posso ridurre la verifica delle condizioni di raccordo  $C^1$  soltanto alla componente delle altezze:

$$\begin{cases} \widetilde{b}_{jk0} = b_{0jk} & \forall j, k \text{ t.c. } j+k = d, \\ \widetilde{b}_{jk1} = \tau_1 b_{1jk} + \tau_2 b_{0(j+1)k} + \tau_3 b_{0j(k+1)} & j+k = d-1. \end{cases}$$

- $\widetilde{\mathbf{V}}_{jk0} = \mathbf{V}_{0jk}$  risulta ovvio, visto che altrimenti i triangoli  $\mathcal{T}_1$  e  $\mathcal{T}_2$  non condividerebbero il lato  $\overline{\mathbf{V}_2 \mathbf{V}_3}$ ;

- mostriamo l'altra relazione,

$$\begin{aligned}
& \tau_1 \overbrace{\left( \frac{1}{d} \mathbf{V}_1 + \frac{j}{d} \mathbf{V}_2 + \frac{k}{d} \mathbf{V}_3 \right)}^{\mathbf{V}_{1jk}} + \tau_2 \overbrace{\left( \frac{j+1}{d} \mathbf{V}_2 + \frac{k}{d} \mathbf{V}_3 \right)}^{\mathbf{V}_{0(j+1)k}} + \tau_1 \overbrace{\left( \frac{j}{d} \mathbf{V}_2 + \frac{k+1}{d} \mathbf{V}_3 \right)}^{\mathbf{V}_{0j(k+1)}} = \\
& = \frac{1}{d} \overbrace{(\tau_1 \mathbf{V}_1 + \tau_2 \mathbf{V}_2 + \tau_3 \mathbf{V}_3)}^{\mathbf{V}_4} + \frac{j}{d} \mathbf{V}_2 \underbrace{(\tau_1 + \tau_2 + \tau_3)}_{=1} + \frac{k}{d} \mathbf{V}_3 \underbrace{(\tau_1 + \tau_2 + \tau_3)}_{=1} = \widetilde{\mathbf{V}}_{jk1}.
\end{aligned}$$

## 1.2 Note sulla determinazione degli elementi di base

**Nota 1.3.** Si tratta semplicemente di imporre le condizioni ed eseguire i calcoli. Ricordiamo, come riportato ad esempio in [1], che se  $p(\mathbf{P}) = \sum_{i+j+k=d} b_{ijk} B_{ijk}^d(\mathbf{P})$  è la locale parametrizzazione come superficie di Bézier di un polinomio, per le derivate direzionali vale la seguente regola:

$$D_{\mathbf{v}} p(\mathbf{P}) = \sum_{i+j+k=d-1} d [a_1 b_{(i+1)jk} + a_2 b_{ij(k+1)} + a_3 b_{ij(k+1)}] B_{ijk}^{d-1}(\mathbf{P}), \quad (1)$$

dette  $(a_1, a_2, a_3)$  le coordinate direzionali del vettore  $\mathbf{v}$  rispetto al triangolo sul quale la superficie di Bézier è localmente parametrizzata.

Scegliamo, senza perdita di generalità, una locale parametrizzazione per  $B_{i,r}^v$  su un qualsiasi triangolo  $\mathcal{T}'_m = \langle \mathbf{V}_i, \mathbf{Z}_n, \mathbf{R}_{i,j} \rangle \in \Delta_{PS}$ : dobbiamo dunque ricavare le coordinate di  $\mathbf{e}_1$  ed  $\mathbf{e}_2$  rispetto a  $\mathcal{T}'_m$  al fine di poter utilizzare (1).

Possiamo passare dalle coordinate cartesiane alle baricentriche agevolmente mediante l'applicazione lineare:

$$\underbrace{\begin{bmatrix} \mathbf{V}_i^x & \mathbf{Z}_n^x & \mathbf{R}_{i,j}^x \\ \mathbf{V}_i^y & \mathbf{Z}_n^y & \mathbf{R}_{i,j}^y \\ 1 & 1 & 1 \end{bmatrix}}_M \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ 0 \end{bmatrix}, \quad (2)$$

dette  $(v_1, v_2)$  le coordinate cartesiane del vettore del quale stiamo ricercando le baricentriche. Nel nostro caso, per avere le coordinate di  $\mathbf{e}_i \in \mathbb{R}^2$ , ci serve calcolare  $M^{-1}\mathbf{e}_i$ .

Invertendo  $M$ , ad esempio mediante lo sviluppo di Laplace, otteniamo:

$$M^{-1} = \frac{1}{\gamma} \begin{bmatrix} \mathbf{Z}_n^y - \mathbf{R}_{i,j}^y & \mathbf{R}_{i,j}^x - \mathbf{Z}_n^x & \mathbf{Z}_n^x \mathbf{R}_{i,j}^y - \mathbf{Z}_n^y \mathbf{R}_{i,j}^x \\ \mathbf{R}_{i,j}^y - \mathbf{V}_i^y & \mathbf{V}_i^x - \mathbf{R}_{i,j}^x & \mathbf{V}_i^y \mathbf{R}_{i,j}^x - \mathbf{V}_i^x \mathbf{R}_{i,j}^y \\ \mathbf{V}_i^y - \mathbf{Z}_n^y & \mathbf{Z}_n^x - \mathbf{V}_i^x & \mathbf{V}_i^x \mathbf{Z}_n^y - \mathbf{V}_i^y \mathbf{Z}_n^x \end{bmatrix},$$

con

$$\gamma = \det M = \mathbf{V}_i^x \mathbf{Z}_n^y + \mathbf{Z}_n^x \mathbf{R}_{i,j}^y + \mathbf{R}_{i,j}^x \mathbf{V}_i^y - \mathbf{V}_i^x \mathbf{R}_{i,j}^y - \mathbf{Z}_n^x \mathbf{V}_i^y - \mathbf{R}_{i,j}^x \mathbf{Z}_n^y.$$

Alla fine otteniamo che  $\mathbf{e}_1$  ha coordinate rispetto a  $\mathcal{T}'_m$ :

$$\left( \frac{\mathbf{Z}_n^y - \mathbf{R}_{i,j}^y}{\gamma}, \frac{\mathbf{R}_{i,j}^y - \mathbf{V}_i^y}{\gamma}, \frac{\mathbf{V}_i^y - \mathbf{Z}_n^y}{\gamma} \right); \quad (3)$$

mentre  $\mathbf{e}_2$  ha coordinate:

$$\left( \frac{\mathbf{R}_{i,j}^x - \mathbf{Z}_n^x}{\gamma}, \frac{\mathbf{V}_i^x - \mathbf{R}_{i,j}^x}{\gamma}, \frac{\mathbf{V}_i^y - \mathbf{Z}_n^y}{\gamma} \right). \quad (4)$$

Passiamo ora ai calcoli di interpolazione veri e propri.

- Imponendo la condizione  $B_{i,r}^v(\mathbf{V}_i) = \alpha_{i,r}$ , e scegliendo la locale parametrizzazione di  $B_{i,r}^v$  in un qualsiasi triangolo  $\mathcal{T}'_m = \langle \mathbf{V}_i, \mathbf{Z}_n, \mathbf{R}_{i,j} \rangle \in \Delta_{PS}$ :

$$B_{i,r}^v(\mathbf{V}_i) = \sum_{i+j+k=3} b_{ijk} B_{ijk}^3(\mathbf{V}_i) = \underbrace{b_{300}}_{b_{1,r}^v} \underbrace{B_{300}^3(\mathbf{V}_i)}_1 = b_{1,r}^v,$$

pertanto  $B_{i,r}^v(\mathbf{V}_i) = \alpha_{i,r}$  vuol dire  $b_{1,r}^v = \alpha_{i,r}$ .

- Veniamo alle condizioni  $D_{\mathbf{e}_1}B_{i,r}^{\nu}(\mathbf{V}_i) = \alpha_{i,r}^x$ ,  $D_{\mathbf{e}_2}B_{i,r}^{\nu}(\mathbf{V}_i) = \alpha_{i,r}^y$  per un  $r \in \{1, 2, 3\}$  fissato.

Scrivendo (1) otteniamo, visto che l'unico multi-indice in  $\mathbf{V}_i$  a valori non nulli è  $(2, 0, 0)$ :

$$D_{\mathbf{e}_i}B_{i,r}^{\nu}(\mathbf{V}_i) = 3 \left[ a_1 \underbrace{b_{300}}_{b_{1,r}^{\nu}} + a_2 \underbrace{b_{210}}_{b_{2,r}^{\nu}} + a_3 \underbrace{b_{201}}_{b_{302,r}^{\nu}} \right] \underbrace{B_{200}^{d-1}}_{=1}.$$

Alla luce delle coordinate (3),(4) individuate ed imponendo le condizioni richieste otteniamo equazioni accoppiate della forma:

$$\begin{cases} 3 \left[ \frac{\mathbf{Z}_n^y - \mathbf{R}_{i,j}^y}{\gamma} \underbrace{\alpha_{i,r}}_{b_{1,r}^{\nu}} + \frac{\mathbf{V}_i^y - \mathbf{Z}_n^y}{\gamma} b_{2,r}^{\nu} + \frac{\mathbf{R}_{i,j}^y - \mathbf{V}_i^y}{\gamma} b_{3,r}^{\nu} \right] = \alpha_{i,r}^x \\ 3 \left[ \frac{\mathbf{R}_{i,j}^x - \mathbf{Z}_n^x}{\gamma} \underbrace{\alpha_{i,r}}_{b_{1,r}^{\nu}} + \frac{\mathbf{Z}_n^x - \mathbf{V}_i^x}{\gamma} b_{2,r}^{\nu} + \frac{\mathbf{V}_i^x - \mathbf{R}_{i,j}^x}{\gamma} b_{3,r}^{\nu} \right] = \alpha_{i,r}^y. \end{cases} \quad (5)$$

Ricavo  $b_{2,r}^{\nu}$  dalla prima equazione in (5)

$$b_{2,r}^{\nu} = \frac{1}{3} \frac{\gamma}{\mathbf{V}_i^y - \mathbf{Z}_n^y} \alpha_{i,r}^x - \frac{\mathbf{Z}_n^y - \mathbf{R}_{i,j}^y}{\mathbf{V}_i^y - \mathbf{Z}_n^y} \alpha_{i,r} - \frac{\mathbf{R}_{i,j}^y - \mathbf{V}_i^y}{\mathbf{V}_i^y - \mathbf{Z}_n^y} b_{3,r}^{\nu}, \quad (6)$$

sostituisco quanto ottenuto nella seconda equazione ricavando in questo modo  $b_{3,r}^{\nu}$

$$b_{3,r}^{\nu} \underbrace{\left( \frac{\mathbf{Z}_n^x - \mathbf{V}_i^x}{\gamma} \frac{\mathbf{R}_{i,j}^y - \mathbf{V}_i^y}{\mathbf{V}_i^y - \mathbf{Z}_n^y} - \frac{\mathbf{V}_i^x - \mathbf{R}_{i,j}^x}{\gamma} \right)}_{(\mathbf{V}_i^y - \mathbf{Z}_n^y)^{-1}} = \frac{\mathbf{R}_{i,j}^x - \mathbf{Z}_n^x}{\gamma} \alpha_{i,r} + \frac{1}{3} \frac{\mathbf{Z}_n^x - \mathbf{V}_i^x}{\mathbf{V}_i^y - \mathbf{Z}_n^y} \alpha_{i,r}^x - \frac{\mathbf{Z}_n^x - \mathbf{V}_i^x}{\gamma} \frac{\mathbf{Z}_n^y - \mathbf{R}_{i,j}^y}{\mathbf{V}_i^y - \mathbf{Z}_n^y} \alpha_{i,r} - \frac{1}{3} \alpha_{i,r}^y$$

$$b_{3,r}^{\nu} = \alpha_{i,r} \underbrace{\left( (\mathbf{V}_i^y - \mathbf{Z}_n^y) \frac{\mathbf{R}_{i,j}^x - \mathbf{Z}_n^x}{\gamma} - \frac{\mathbf{Z}_n^x - \mathbf{V}_i^x}{\gamma} (\mathbf{Z}_n^y - \mathbf{R}_{i,j}^y) \right)}_{=1} \alpha_{i,r} + \frac{1}{3} (\mathbf{Z}_n^x - \mathbf{V}_i^x) \alpha_{i,r}^x - \frac{1}{3} (\mathbf{V}_i^y - \mathbf{Z}_n^y) \alpha_{i,r}^y$$

cioè abbiamo finalmente ottenuto:

$$b_{3,r}^{\nu} = \alpha_{i,r} + \frac{1}{3} \langle (\alpha_{i,r}^x, \alpha_{i,r}^y), \mathbf{Z}_n - \mathbf{V}_i \rangle. \quad (7)$$

Per trovare definitivamente  $b_{2,r}^{\nu}$  dobbiamo sostituire (7) dentro (6). Dopo aver fatto ciò e separato i termini  $\alpha_{i,r}$ ,  $\alpha_{i,r}^x$ ,  $\alpha_{i,r}^y$ .

$$b_{2,r}^{\nu} = \alpha_{i,r} \underbrace{\left( -\frac{\mathbf{Z}_n^y - \mathbf{R}_{i,j}^y}{\mathbf{V}_i^y - \mathbf{Z}_n^y} - \frac{\mathbf{R}_{i,j}^y - \mathbf{V}_i^y}{\mathbf{V}_i^y - \mathbf{Z}_n^y} \right)}_{=1} + \alpha_{i,r}^x \underbrace{\left( \frac{1}{3} \frac{\gamma}{\mathbf{V}_i^y - \mathbf{Z}_n^y} - \frac{1}{3} (\mathbf{Z}_n^x - \mathbf{V}_i^x) \frac{\mathbf{R}_{i,j}^y - \mathbf{V}_i^y}{\mathbf{V}_i^y - \mathbf{Z}_n^y} \right)}_{=\frac{1}{3}(\mathbf{R}_{i,j}^x - \mathbf{V}_i^x)} + \frac{1}{3} (\mathbf{R}_{i,j}^y - \mathbf{V}_i^y) \alpha_{i,r}^y$$

cioè proprio

$$b_{2,r}^{\nu} = \alpha_{i,r} + \frac{1}{3} \langle (\alpha_{i,r}^x, \alpha_{i,r}^y), \mathbf{R}_{i,j} - \mathbf{V}_i \rangle. \quad (8)$$

- Per individuare  $b_{4,r}^v$  è sufficiente ripetere la stessa procedura utilizzata per  $b_{2,r}^v$  ma scegliendo il triangolo  $\mathcal{T}'_{m'} = \langle \mathbf{V}_i, \mathbf{Z}_n, \mathbf{R}_{i,k} \rangle \in \Delta_{PS}$ .

**Nota 1.4.** Mostriamo  $\frac{1}{3}\mathbf{V}_i + \frac{1}{3}\mathbf{R}_{i,j} + \frac{1}{3}\mathbf{Z}_n = \left(1 - \frac{\lambda_{ji}}{2\sigma_m^t}\right) \mathbf{S}_{i,n}^t + \frac{\lambda_{ji}}{2\sigma_m^t} \mathbf{Q}_m^t$ .

$$\begin{aligned} & \left(1 - \frac{\lambda_{ji}}{2\sigma_m^t}\right) \mathbf{S}_{i,n}^t + \frac{\lambda_{ji}}{2\sigma_m^t} \underbrace{\mathbf{Q}_m^t}_{\mathbf{S}_{i,n}^t + \frac{2}{3}\sigma_m^t(\mathbf{V}_j - \mathbf{V}_i)} = \\ &= \mathbf{S}_{i,n}^t + \frac{\lambda_{ji}}{2\sigma_m^t} \frac{2}{3}\sigma_m^t(\mathbf{V}_j - \mathbf{V}_i) = \\ &= \mathbf{S}_{i,n}^t + \frac{1}{3} \underbrace{\lambda_{ji}\mathbf{V}_j}_{\mathbf{R}_{i,j} - \lambda_{ij}\mathbf{V}_i} - \frac{1}{3}\lambda_{ji}\mathbf{V}_i = \\ &= \mathbf{S}_{i,n}^t + \frac{1}{3}\mathbf{R}_{i,j} - \frac{1}{3}\lambda_{ij}\mathbf{V}_i - \frac{1}{3}\lambda_{ji}\mathbf{V}_i \\ &\stackrel{\lambda_{ij} + \lambda_{ji} = 1}{=} \underbrace{\mathbf{S}_{i,n}^t}_{\frac{2}{3}\mathbf{V}_i + \frac{1}{3}\mathbf{Z}_n} + \frac{1}{3}\mathbf{R}_{i,j} - \frac{1}{3}\mathbf{V}_i = \\ &= \frac{1}{3}\mathbf{V}_i + \frac{1}{3}\mathbf{R}_{i,j} + \frac{1}{3}\mathbf{Z}_n. \end{aligned}$$

La verifica di  $\frac{1}{3}\mathbf{V}_i + \frac{1}{3}\mathbf{R}_{i,k} + \frac{1}{3}\mathbf{Z}_n = \left(1 - \frac{\lambda_{ki}}{2\sigma_{m'}^t}\right) \mathbf{S}_{i,n}^t + \frac{\lambda_{ki}}{2\sigma_{m'}^t} \mathbf{Q}_{m'}^t$  risulta perfettamente analoga, mentre due parole su  $\frac{1}{3}\mathbf{V}_i + \frac{2}{3}\mathbf{Z}_n = \left(1 - \frac{\xi_{n,j}}{2\sigma_m^t} - \frac{\xi_{n,k}}{2\sigma_{m'}^t}\right) \mathbf{S}_{i,n}^t + \frac{\xi_{n,j}}{2\sigma_m^t} \mathbf{Q}_m^t + \frac{\xi_{n,k}}{2\sigma_{m'}^t} \mathbf{Q}_{m'}^t$ , sono necessarie.

$$\begin{aligned} & \left(1 - \frac{\xi_{n,j}}{2\sigma_m^t} - \frac{\xi_{n,k}}{2\sigma_{m'}^t}\right) \mathbf{S}_{i,n}^t + \frac{\xi_{n,j}}{2\sigma_m^t} \underbrace{\mathbf{Q}_m^t}_{\mathbf{S}_{i,n}^t + \frac{2}{3}\sigma_m^t(\mathbf{V}_j - \mathbf{V}_i)} + \frac{\xi_{n,k}}{2\sigma_{m'}^t} \underbrace{\mathbf{Q}_{m'}^t}_{\mathbf{S}_{i,n}^t + \frac{2}{3}\sigma_{m'}^t(\mathbf{V}_j - \mathbf{V}_i)} = \\ &= \mathbf{S}_{i,n}^t + \frac{1}{3}\xi_{n,j}(\mathbf{V}_j - \mathbf{V}_i) + \frac{1}{3}\xi_{n,k}(\mathbf{V}_k - \mathbf{V}_i) \stackrel{\xi_{n,j}\mathbf{V}_j + \xi_{n,k}\mathbf{V}_k = \mathbf{Z}_n - (1 - \xi_{n,j} - \xi_{n,k})\mathbf{V}_i}{=} \\ &= \underbrace{\mathbf{S}_{i,n}^t}_{\frac{2}{3}\mathbf{V}_i + \frac{1}{3}\mathbf{Z}_n} + \frac{1}{3}\mathbf{Z}_n - \underbrace{\frac{1}{3}(1 - \xi_{n,j} - \xi_{n,k})\mathbf{V}_i}_{-\frac{1}{3}\mathbf{V}_i} - \frac{1}{3}(\xi_{n,j} + \xi_{n,k})\mathbf{V}_i = \\ &= \frac{1}{3}\mathbf{V}_i + \frac{2}{3}\mathbf{Z}_n. \end{aligned}$$

**Nota 1.5.** In maniera non troppo differente dalla nota precedente, dobbiamo far vedere che  $\frac{1}{3}\mathbf{V}_i + \frac{2}{3}\mathbf{R}_{i,j} = \left(1 - \frac{\lambda_{ji}}{2\sigma_{i,j}^e}\right) \mathbf{S}_{i,j}^e + \frac{\lambda_{ji}}{2\sigma_{i,j}^e} \mathbf{Q}_{i,j}^e$ . Partendo ancora una volta dal termine a destra:

$$\begin{aligned} & \left(1 - \frac{\lambda_{ji}}{2\sigma_{i,j}^e}\right) \mathbf{S}_{i,j}^e + \frac{\lambda_{ji}}{2\sigma_{i,j}^e} \underbrace{\mathbf{Q}_{i,j}^e}_{\mathbf{S}_{i,j}^e + \frac{2}{3}\sigma_{i,j}^e(\mathbf{V}_j - \mathbf{V}_i)} = \\ &= \mathbf{S}_{i,j}^e + \frac{\lambda_{ji}}{2\sigma_{i,j}^e} \frac{2}{3}\sigma_{i,j}^e(\mathbf{V}_j - \mathbf{V}_i) = \\ &= \mathbf{S}_{i,j}^e + \frac{1}{3}\lambda_{ji}\mathbf{V}_j - \frac{1}{3}\lambda_{ji}\mathbf{V}_i = \end{aligned}$$

$$\begin{aligned}
& \stackrel{\lambda_{ji}V_j = R_{i,j} - \lambda_{ij}V_i}{=} S_{i,j}^e + \frac{1}{3}R_{i,j} - \frac{1}{3}\lambda_{ij}V_i - \frac{1}{3}\lambda_{ji}V_i = \\
& \stackrel{\lambda_{ij} + \lambda_{ji} = 1}{=} \underbrace{S_{i,j}^e}_{\frac{2}{3}V_i + \frac{1}{3}R_{i,j}} + \frac{1}{3}R_{i,j} - \frac{1}{3}V_i = \\
& = \frac{1}{3}V_i + \frac{2}{3}R_{i,j}.
\end{aligned}$$

**Nota 1.6.** Ricaviamo  $b_{10,r}^v$ ,  $b_{11,r}^v$ ,  $b_{12,r}^v$  sfruttando le condizioni di raccordo  $C^1$  tra superfici di Bézier in  $\langle V_i, Z_n, R_{i,j} \rangle$  e  $\langle Z_n, R_{i,j}, V_j \rangle$  sul lato comune  $Z_n R_{i,j}$ .

Anzitutto ci occorrono le coordinate baricentriche di  $V_j$  rispetto a  $\langle V_i, Z_n, R_{i,j} \rangle$ . Invertendo  $R_{i,j} = \lambda_{ij}V_i + \lambda_{ji}V_j$  otteniamo  $V_j = -\lambda_{ji}^{-1}\lambda_{ij}V_i + \lambda_{ji}^{-1}R_{i,j}$ , perciò le coordinate cercate sono  $(-\lambda_{ji}^{-1}\lambda_{ij}, 0, \lambda_{ji}^{-1})$ .

- Calcoliamo  $b_{10,r}^v$ . Usando le regole di raccordo:

$$\underbrace{b_{021}}_0 = -\lambda_{ji}^{-1}\lambda_{ij} \underbrace{b_{102}}_{b_{5,r}^v} + \lambda_{ji}^{-1} \underbrace{b_{003}}_{b_{10,r}^v} \Rightarrow b_{10,r}^v = \lambda_{ji} b_{5,r}^v.$$

- Per quanto riguarda  $b_{11,r}^v$ :

$$\underbrace{b_{111}}_0 = -\lambda_{ji}^{-1}\lambda_{ij} \underbrace{b_{111}}_{b_{6,r}^v} + \lambda_{ji}^{-1} \underbrace{b_{012}}_{b_{11,r}^v} \Rightarrow b_{11,r}^v = \lambda_{ji} b_{6,r}^v.$$

- Ed infine  $b_{12,r}^v$ :

$$\underbrace{b_{201}}_0 = -\lambda_{ji}^{-1}\lambda_{ij} \underbrace{b_{120}}_{b_{7,r}^v} + \lambda_{ji}^{-1} \underbrace{b_{021}}_{b_{12,r}^v} \Rightarrow b_{12,r}^v = \lambda_{ji} b_{7,r}^v.$$

**Nota 1.7.** Veniamo adesso all'individuazione dell'altezza sull'incentro  $b_{13,r}^v$ : condizioni di raccordo tra  $\langle R_{i,j}, V_i, Z_n \rangle$  e  $\langle V_i, Z_n, R_{i,k} \rangle$  lungo  $\overline{V_i Z_n}$ . Componente non banale del ragionamento è la ricerca delle coordinate baricentriche di  $R_{i,k}$  rispetto a  $\langle R_{i,j}, V_i, Z_n \rangle$ .

$$\begin{aligned}
R_{i,k} &= \lambda_{ik}V_i + \lambda_{ki}V_k \stackrel{V_k = (Z_n - \xi_{n,i}V_i - \xi_{n,j}V_j)\xi_{n,k}^{-1}}{=} \lambda_{ik}V_i + \lambda_{ki}\xi_{n,k}^{-1}Z_n - \lambda_{k,i}\xi_{n,k}^{-1}\xi_{n,i}V_i - \lambda_{ki}\xi_{n,k}^{-1}\xi_{n,j}V_j = \\
&= (\lambda_{ik} - \lambda_{ki}\xi_{n,k}^{-1}\xi_{n,i})V_i + \lambda_{ki}\xi_{n,k}^{-1}Z_n - \lambda_{ki}\xi_{n,k}^{-1}\xi_{n,j} \underbrace{V_j}_{} = \\
&= \lambda_{ji}^{-1}R_{i,j} - \lambda_{ji}^{-1}\lambda_{ij}V_i \\
&= (\lambda_{ik} - \lambda_{ki}\xi_{n,k}^{-1}\xi_{n,i} + \lambda_{ki}\lambda_{ji}^{-1}\lambda_{ij}\xi_{n,k}^{-1}\xi_{n,j})V_i + \lambda_{ki}\xi_{n,k}^{-1}Z_n - \lambda_{ki}\lambda_{ji}^{-1}\xi_{n,k}^{-1}\xi_{n,j}R_{i,j}.
\end{aligned}$$

Le coordinate cercate sono:  $(\tau_1, \tau_2, \tau_3) := (-\lambda_{ki}\lambda_{ji}^{-1}\xi_{n,k}^{-1}\xi_{n,j}, \lambda_{ik} - \lambda_{ki}\xi_{n,k}^{-1}\xi_{n,i} + \lambda_{ki}\lambda_{ji}^{-1}\lambda_{ij}\xi_{n,k}^{-1}\xi_{n,j}, \lambda_{ki}\xi_{n,k}^{-1})$ , utilizziamo ora le condizioni di raccordo.

$$\underbrace{b_{021}}_{b_{14,r} = \lambda_{ik}b_{7,r}^v} = \tau_1 \underbrace{b_{102}}_{b_{12,r}^v = \lambda_{ij}b_{7,r}^v} + \tau_2 \underbrace{b_{012}}_{b_{7,r}^v} + \tau_3 \underbrace{b_{003}}_{b_{13,r}^v}$$

Invertendo, lascio al lettore i calcoli indicati:

$$\begin{aligned}
b_{13,r}^v &= \tau_3^{-1}(\lambda_{ik}b_{7,r}^v - \tau_1\lambda_{ij}b_{7,r}^v - \tau_2b_{7,r}^v) = \\
&= \underbrace{(\lambda_{ik}\tau_3^{-1} - \lambda_{ij}\tau_3^{-1}\tau_1 - \tau_3^{-1}\tau_2)}_{\xi_{n,i}} b_{7,r}^v = \xi_{n,i} b_{7,r}^v.
\end{aligned}$$

### 1.3 Una nota sulla non-negatività degli elementi di base

**Nota 1.8.** Facciamo vedere, a titolo di esempio, che  $b_{2,1}^v$  è la prima coordinata baricentrica del punto dominio  $\frac{2}{3}\mathbf{V}_i + \frac{1}{3}\mathbf{R}_{i,j}$  rispetto al triangolo  $\mathbf{Q}_i^v = \langle \mathbf{Q}_{i,1}, \mathbf{Q}_{i,2}, \mathbf{Q}_{i,3} \rangle$ .

Il passaggio alle coordinate baricentriche è analogo a quanto descritto in (2), stavolta la matrice da invertire è:

$$\underbrace{\begin{bmatrix} \mathbf{Q}_{i,1}^x & \mathbf{Q}_{i,2}^x & \mathbf{Q}_{i,3}^x \\ \mathbf{Q}_{i,1}^y & \mathbf{Q}_{i,2}^y & \mathbf{Q}_{i,3}^y \\ 1 & 1 & 1 \end{bmatrix}}_M$$

ed otteniamo, sempre mediante Laplace,

$$M^{-1} = \frac{1}{\gamma} \begin{bmatrix} \mathbf{Q}_{i,2}^y - \mathbf{Q}_{i,3}^y & \mathbf{Q}_{i,3}^x - \mathbf{Q}_{i,2}^x & \mathbf{Q}_{i,2}^x \mathbf{Q}_{i,3}^y - \mathbf{Q}_{i,2}^y \mathbf{Q}_{i,3}^x \\ \mathbf{Q}_{i,3}^y - \mathbf{Q}_{i,1}^y & \mathbf{Q}_{i,1}^x - \mathbf{Q}_{i,3}^x & \mathbf{Q}_{i,1}^y \mathbf{Q}_{i,3}^x - \mathbf{Q}_{i,1}^x \mathbf{Q}_{i,3}^y \\ \mathbf{Q}_{i,1}^y - \mathbf{Q}_{i,2}^y & \mathbf{Q}_{i,2}^x - \mathbf{Q}_{i,1}^x & \mathbf{Q}_{i,1}^y \mathbf{Q}_{i,2}^x - \mathbf{Q}_{i,1}^x \mathbf{Q}_{i,2}^y \end{bmatrix},$$

con

$$\gamma := \det M = \mathbf{Q}_{i,1}^x \mathbf{Q}_{i,2}^y + \mathbf{Q}_{i,2}^x \mathbf{Q}_{i,3}^y + \mathbf{Q}_{i,3}^x \mathbf{Q}_{i,1}^y - \mathbf{Q}_{i,1}^x \mathbf{Q}_{i,3}^y - \mathbf{Q}_{i,2}^x \mathbf{Q}_{i,1}^y - \mathbf{Q}_{i,3}^x \mathbf{Q}_{i,2}^y.$$

Dobbiamo quindi mostrare che  $b_{2,1}^v = [M^{-1}(\frac{2}{3}\mathbf{V}_i + \frac{1}{3}\mathbf{R}_{i,j}, 1)]_1$ . Sviluppando questo ultimo prodotto otteniamo:

$$\left[ M^{-1} \left( \frac{2}{3}\mathbf{V}_i + \frac{1}{3}\mathbf{R}_{i,j} \right) \right]_1 = \frac{1}{\gamma} \left( \frac{2}{3}\mathbf{V}_i^x \mathbf{Q}_{i,2}^y - \frac{2}{3}\mathbf{V}_i^y \mathbf{Q}_{i,3}^x + \frac{1}{3}\mathbf{R}_{i,j}^x \mathbf{Q}_{i,2}^y - \frac{1}{3}\mathbf{R}_{i,j}^y \mathbf{Q}_{i,3}^x + \frac{2}{3}\mathbf{V}_i^y \mathbf{Q}_{i,3}^x - \frac{2}{3}\mathbf{V}_i^x \mathbf{Q}_{i,2}^y + \right. \\ \left. + \frac{1}{3}\mathbf{R}_{i,j}^x \mathbf{Q}_{i,3}^x - \frac{1}{3}\mathbf{R}_{i,j}^y \mathbf{Q}_{i,2}^x + \mathbf{Q}_{i,2}^x \mathbf{Q}_{i,3}^y - \mathbf{Q}_{i,2}^y \mathbf{Q}_{i,3}^x \right) = \dots$$

effettuiamo ora le sostituzioni  $\mathbf{V}_i^x = \alpha_{i,1} \mathbf{Q}_{i,1}^x + \alpha_{i,2} \mathbf{Q}_{i,2}^x + \alpha_{i,3} \mathbf{Q}_{i,3}^x$  e  $\mathbf{V}_i^y = \alpha_{i,1} \mathbf{Q}_{i,1}^y + \alpha_{i,2} \mathbf{Q}_{i,2}^y + \alpha_{i,3} \mathbf{Q}_{i,3}^y$ ,

$$\dots = \frac{1}{\gamma} \left( \frac{2}{3}\alpha_{i,1} \mathbf{Q}_{i,1}^x \mathbf{Q}_{i,2}^y + \underbrace{\left( \frac{2}{3}\alpha_{i,2} + \frac{2}{3}\alpha_{i,3} - 1 \right)}_{\left( \frac{2}{3} - \frac{2}{3}\alpha_{i,1} \right) - 1 = -\frac{2}{3}\alpha_{i,1} - \frac{1}{3}} \mathbf{Q}_{i,2}^y \mathbf{Q}_{i,3}^x - \frac{2}{3}\alpha_{i,1} \mathbf{Q}_{i,1}^x \mathbf{Q}_{i,3}^y + \underbrace{\left( 1 - \frac{2}{3}\alpha_{i,2} - \frac{2}{3}\alpha_{i,3} \right)}_{1 - \left( \frac{2}{3} - \frac{2}{3}\alpha_{i,1} \right) = \frac{1}{3} + \frac{2}{3}\alpha_{i,1}} \cdot \right. \\ \left. \cdot \mathbf{Q}_{i,2}^x \mathbf{Q}_{i,3}^y + \frac{2}{3}\alpha_{i,1} \mathbf{Q}_{i,1}^y \mathbf{Q}_{i,3}^x - \frac{2}{3}\alpha_{i,1} \mathbf{Q}_{i,1}^y \mathbf{Q}_{i,2}^x + \frac{1}{3}\mathbf{R}_{i,j}^x (\mathbf{Q}_{i,2}^y - \mathbf{Q}_{i,3}^y) + \frac{1}{3}\mathbf{R}_{i,j}^y (\mathbf{Q}_{i,3}^x - \mathbf{Q}_{i,2}^x) \right) = \dots$$

A questo punto completo a  $\gamma$  e separo le componenti residue dei singoli termini:

$$\dots = \underbrace{\frac{\alpha_{i,1}\gamma}{\gamma}}_{\alpha_{i,1}} + \frac{1}{3\gamma} \left( -\alpha_{i,1} \mathbf{Q}_{i,1}^x \mathbf{Q}_{i,2}^y + \underbrace{(\alpha_{i,1} - 1)}_{-\alpha_{i,2} - \alpha_{i,3}} \mathbf{Q}_{i,2}^y \mathbf{Q}_{i,3}^x + \alpha_{i,1} \mathbf{Q}_{i,1}^x \mathbf{Q}_{i,3}^y + \underbrace{(1 - \alpha_{i,1})}_{\alpha_{i,2} + \alpha_{i,3}} \mathbf{Q}_{i,2}^x \mathbf{Q}_{i,3}^y \right. \\ \left. - \alpha_{i,1} \mathbf{Q}_{i,1}^y \mathbf{Q}_{i,3}^x + \alpha_{i,1} \mathbf{Q}_{i,1}^y \mathbf{Q}_{i,2}^x + \mathbf{R}_{i,j}^x (\mathbf{Q}_{i,2}^y - \mathbf{Q}_{i,3}^y) + \mathbf{R}_{i,j}^y (\mathbf{Q}_{i,3}^x - \mathbf{Q}_{i,2}^x) \right) = \dots$$

Ora, sfruttando l'uguaglianza  $\mathbf{V}_i^x = \alpha_{i,1} \mathbf{Q}_{i,1}^x + \alpha_{i,2} \mathbf{Q}_{i,2}^x + \alpha_{i,3} \mathbf{Q}_{i,3}^x$  operiamo le sostituzioni

$-\alpha_{i,1}\mathbf{Q}_{i,1}^x\mathbf{Q}_{i,2}^y - \alpha_{i,3}\mathbf{Q}_{i,3}^x\mathbf{Q}_{i,2}^y = \mathbf{Q}_{i,2}^y(-\mathbf{V}_i^x + \alpha_{i,2}\mathbf{Q}_{i,2}^x)$  e  $\alpha_{i,1}\mathbf{Q}_{i,1}^x\mathbf{Q}_{i,3}^y + \alpha_{i,2}\mathbf{Q}_{i,2}^x\mathbf{Q}_{i,3}^y = \mathbf{Q}_{i,3}^y(\mathbf{V}_i^x - \alpha_{i,3}\mathbf{Q}_{i,3}^x)$ :

$$\dots = \alpha_{i,1} + \frac{1}{3\gamma} (\mathbf{Q}_{i,2}^y(-\mathbf{V}_i^x + \alpha_{i,2}\mathbf{Q}_{i,2}^x) - \alpha_{i,2}\mathbf{Q}_{i,2}^y\mathbf{Q}_{i,3}^x + \mathbf{Q}_{i,3}^y(\mathbf{V}_i^x - \alpha_{i,3}\mathbf{Q}_{i,3}^x) + \alpha_{i,3}\mathbf{Q}_{i,2}^x\mathbf{Q}_{i,3}^y + \dots \\ - \alpha_{i,1}\mathbf{Q}_{i,1}^y\mathbf{Q}_{i,3}^x + \alpha_{i,1}\mathbf{Q}_{i,1}^y\mathbf{Q}_{i,2}^x + \mathbf{R}_{i,j}^x(\mathbf{Q}_{i,2}^y - \mathbf{Q}_{i,3}^y) + \mathbf{R}_{i,j}^y(\mathbf{Q}_{i,3}^x - \mathbf{Q}_{i,2}^x)) = \dots$$

Allo stesso modo sfruttiamo  $\mathbf{V}_i^y = \alpha_{i,1}\mathbf{Q}_{i,1}^y + \alpha_{i,2}\mathbf{Q}_{i,2}^y + \alpha_{i,3}\mathbf{Q}_{i,3}^y$  per ricavare le sostituzioni  $\alpha_{i,1}\mathbf{Q}_{i,1}^y\mathbf{Q}_{i,2}^x + \alpha_{i,2}\mathbf{Q}_{i,2}^y\mathbf{Q}_{i,3}^x = \mathbf{Q}_{i,2}^x(\mathbf{V}_i^y - \alpha_{i,3}\mathbf{Q}_{i,3}^y)$  e  $-\alpha_{i,2}\mathbf{Q}_{i,2}^y\mathbf{Q}_{i,3}^x - \alpha_{i,3}\mathbf{Q}_{i,3}^y\mathbf{Q}_{i,2}^x = \mathbf{Q}_{i,3}^x(-\mathbf{V}_i^y + \alpha_{i,1}\mathbf{Q}_{i,1}^y)$ , riordinando inoltre i termini:

$$\dots = \alpha_{i,r} + \frac{1}{3} \left( (\mathbf{R}_{i,j}^x - \mathbf{V}_i^x) \frac{\mathbf{Q}_{i,2}^y - \mathbf{Q}_{i,3}^y}{\gamma} + (\mathbf{R}_{i,j}^y - \mathbf{V}_i^y) \frac{\mathbf{Q}_{i,3}^x - \mathbf{Q}_{i,2}^x}{\gamma} \right). \quad (9)$$

A patto di provare che

$$\begin{cases} \alpha_{i,1}^x = \frac{\mathbf{Q}_{i,2}^y - \mathbf{Q}_{i,3}^y}{\gamma} \\ \alpha_{i,1}^y = \frac{\mathbf{Q}_{i,3}^x - \mathbf{Q}_{i,2}^x}{\gamma} \end{cases} \quad (10)$$

riconosco in (9) proprio il valore  $b_{2,1}^v$  così come lo abbiamo ricavato in (8).

Dimostriamo infine che le uguaglianze in (10) sono verificate: visto che  $\alpha_{i,1}^x$  è la prima coordinata baricentrica di  $\mathbf{e}_1$  rispetto al triangolo  $\mathcal{Q}_i^y$  otteniamo

$$\alpha_{i,1}^x = [\mathcal{M}^{-1}\mathbf{e}_1]_1 = [\mathcal{M}^{-1}]_{11} = \frac{\mathbf{Q}_{i,2}^y - \mathbf{Q}_{i,3}^y}{\gamma}.$$

Analogamente  $\alpha_{i,1}^y$  è la prima coordinata baricentrica di  $\mathbf{e}_2$  sempre rispetto a  $\mathcal{Q}_i^y$  e dunque:

$$\alpha_{i,1}^y = [\mathcal{M}^{-1}\mathbf{e}_2]_1 = [\mathcal{M}^{-1}]_{12} = \frac{\mathbf{Q}_{i,3}^x - \mathbf{Q}_{i,2}^x}{\gamma}.$$

## 2 Implementazione con qualche commento

Riportiamo adesso il codice scritto per produrre le figure utilizzate nella trattazione "Superfici B-spline cubiche di Powell-Sabin": sono state realizzate tre classi Python volte a gestire dominio, raffinamento di Powell-Sabin, insiemi di determinazione minimi (classe `DomainTriangulation`); triangoli di sollevamento  $\mathcal{Q}_i^y$ ,  $\mathcal{Q}_{i,n}^t$  e segmenti  $\mathcal{Q}_{i,j}^e$  (classe `LiftTriangles`); costruzione e visualizzazione delle corrispettive superfici B-spline (classe `PS_Spline`).

Prima di presentare e descrivere ciascuna classe, nell'ordine specifico di dipendenza, vediamo il funzionamento di alcune librerie impiegate nello sviluppo.

### 2.1 Alcuni prerequisiti

#### 2.1.1 Gestire le triangolazioni con `matplotlib.tri` e `matplotlib.pyplot`

Una classe utile a memorizzare triangolazioni e ad offrire elementi per elaborarle è `matplotlib.tri.Triangulation`: fornendo un array `numpy` di ascisse di punti `tr_x`, un

array di ordinate `tr_y` e una matrice `triangles` tale che ogni riga sia della forma  $[i \ j \ k]$ , individuando un triangolo tra l' $i$ -esimo, il  $j$ -esimo e il  $k$ -esimo punto, il comando `Triangulation(tr_x, tr_y, triangles)` genera l'oggetto triangolazione.

Se il parametro `triangles` non viene fornito una triangolazione a partire dai punti indicati è costruita mediante l'algoritmo di Delaunay.

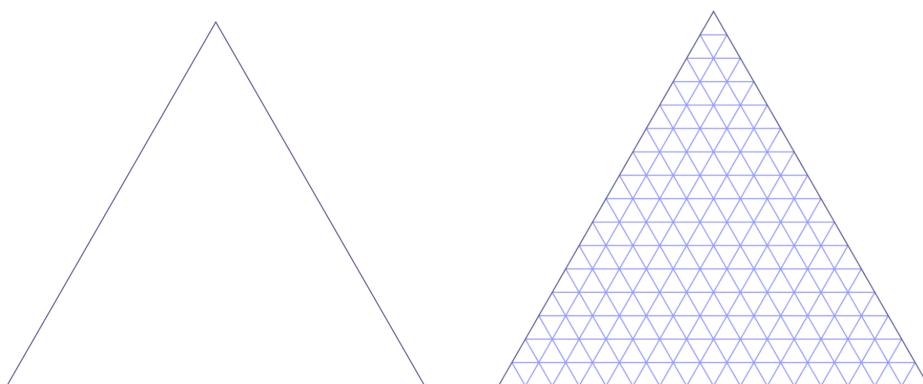
Particolarmente utile risulta l'attributo `neighbors` dell'oggetto `Triangulation`: si tratta di una matrice tale che alla riga  $i$ -esima memorizza le posizioni dei triangoli vicini al triangolo in posizione  $i$ , il valore  $-1$  è associato a nessun triangolo adiacente lungo uno specifico lato.

Una triangolazione `tr` viene visualizzata grazie al comando `tripplot(tr)`, sfruttando il metodo `tripplot` fornito da `matplotlib.pyplot`; superfici su domini triangolari sono visualizzate da `plot_trisurf` in maniera tridimensionale oppure mediante curve di livello da `tricontour`.

In particolare, una superficie è definita indicando una triangolazione `plotTr` di tipo `Triangulation` e un array `triHeights` che alla posizione di ciascun punto associa l'altezza corrispettiva nella superficie: il comando `plot_trisurf(plotTr, triHeights)` visualizza la superficie.

Nel nostro contesto `plotTr` sarà un raffinamento ulteriore di  $\Delta_{PS}$ , costruito prendendo triangolini i cui vertici sono ottenuti da combinazioni baricentriche equispaziate dei vertici del triangolo di partenza: un tale raffinamento è detto uniforme. Si realizzano raffinamenti uniformi mediante l'oggetto `UniformTriRefiner`: anzitutto un raffinatore `ref` è creato dal comando `ref = UniformTriRefiner(tr)` e successivamente la triangolazione raffinata, in modo che contenga  $4^s$  sottotriangoli equispaziati, è prodotta dal comando `tr1 = ref.refine_triangulation(subdiv=s)`.

Mostriamo un esempio, ponendo  $s = 3$  il triangolo di partenza viene equamente raffinato in  $4^3 = 64$  triangolini:



## 2.1.2 Risolvere problemi di minimo con `scipy.optimize`

Per la costruzione dei triangoli  $Q_i^v$ , nella classe `LiftTriangles`, risulta necessario, come vedremo, risolvere uno specifico problema di massimo: `scipy.optimize` mette a disposizione soltanto il metodo `minimize`, vista tuttavia la relazione  $\max_D f = -\min_D(-f)$  ogni

problema di massimo può essere riscritto come uno di minimo. Diamo due dettagli sul funzionamento di `minimize` con utilizzo di vincoli lineari, acquisiti mediante oggetti di tipo `LinearConstraint`.

- Un vincolo lineare assume la forma, detta  $\mathbf{x} \in \mathbb{R}^d$  candidata soluzione ottima:

$$\mathbf{v}^{(1)} \leq A\mathbf{x} \leq \mathbf{v}^{(2)}, \quad A \in \mathbb{R}^{d \times d}, \quad \mathbf{v}^{(i)} \in \mathbb{R}^d. \quad (11)$$

Con le convenzioni  $\mathbf{v}_j^{(i)} = \infty$  nel caso di eventuale non limitatezza e  $\mathbf{v}_j^{(1)} = \mathbf{v}_j^{(2)}$  nel caso di vincoli di uguaglianza, il vincolo (11) è codificato dal comando `LinearConstraint(A, v1, v2)`.

- Il minimo di  $f$  ricavato mediante un metodo iterativo a partire dal valore iniziale  $\mathbf{x}_0$  si ricava da `minimize(f, x0, constraints=ConstrList).x`, dove `ConstrList` è una lista di vincoli lineari nella forma descritta sopra.

## 2.2 La classe DomainTriangulation

Come anticipato sopra, tale classe ha lo scopo di inizializzare il dominio creando, a partire dalla triangolazione  $\Delta$  data in fase di costruzione, il raffinamento  $\Delta_{PS}$ . Sono inoltre offerti strumenti di visualizzazione:  $\Delta$  con o senza  $\Delta_{PS}$  ed eventuale insieme di determinazione minimo.

Unico punto sul quale sono necessarie delucidazioni, visto che il codice è già di per sé ampiamente commentato, sono i dettagli relativi all'esecuzione del raffinamento di Powell-Sabin.

- Per ogni triangolo  $\mathcal{T}_n = \langle \mathbf{V}_i, \mathbf{V}_j, \mathbf{V}_k \rangle \in \Delta$ , l'incentro è individuato dalle coordinate baricentriche  $(\xi_{n,i}, \xi_{n,j}, \xi_{n,k}) := \left( \frac{\overline{\mathbf{V}_j \mathbf{V}_k}}{\ell}, \frac{\overline{\mathbf{V}_k \mathbf{V}_i}}{\ell}, \frac{\overline{\mathbf{V}_i \mathbf{V}_j}}{\ell} \right)$  rispetto a  $\mathcal{T}_n$  stesso, detto  $\ell := \overline{\mathbf{V}_i \mathbf{V}_j} + \overline{\mathbf{V}_j \mathbf{V}_k} + \overline{\mathbf{V}_k \mathbf{V}_i}$  il relativo perimetro;
- le coordinate cartesiane dei punti di suddivisione  $\mathbf{R}_{t,j}$  e relative coordinate baricentriche  $(\lambda_{ij}, \lambda_{ji})$  rispetto a  $\langle \mathbf{V}_i, \mathbf{V}_j \rangle$ ,  $(\mu_{nn'}, \mu_{n'n})$  rispetto a  $\langle \mathbf{Z}_n, \mathbf{Z}_{n'} \rangle$  si ricavano risolvendo il sistema lineare:

$$\underbrace{(1-t)\mathbf{Z}_n + t\mathbf{Z}_{n'}}_{\mu_{nn'}} = \underbrace{(1-s)\mathbf{V}_i + s\mathbf{V}_j}_{\lambda_{ij}},$$

alla fine (risparmiamo i calcoli) si ottiene, se  $\mathbf{Z}_n^x \neq \mathbf{Z}_{n'}^x$

$$\begin{cases} s = \lambda_{ji} = \frac{\mathbf{V}_i^y(\mathbf{Z}_{n'}^x - \mathbf{Z}_n^x) - \mathbf{V}_i^x(\mathbf{Z}_{n'}^y - \mathbf{Z}_n^y) + \mathbf{Z}_n^x \mathbf{Z}_{n'}^y - \mathbf{Z}_{n'}^x \mathbf{Z}_n^y}{(\mathbf{Z}_{n'}^x - \mathbf{Z}_n^x)(\mathbf{V}_i^y - \mathbf{V}_j^y) - (\mathbf{Z}_{n'}^y - \mathbf{Z}_n^y)(\mathbf{V}_i^x - \mathbf{V}_j^x)} \\ t = \mu_{n'n} = \frac{\mathbf{V}_i^y - \mathbf{Z}_n^x + s(\mathbf{V}_j^x - \mathbf{V}_i^x)}{\mathbf{Z}_{n'}^x - \mathbf{Z}_n^x} \end{cases}$$

se invece  $\mathbf{Z}_n^x = \mathbf{Z}_{n'}^x$ , sicuramente avremo  $\mathbf{Z}_n^y \neq \mathbf{Z}_{n'}^y$  e di conseguenza i valori

$$\begin{cases} s = \lambda_{ji} = \frac{\mathbf{V}_i^x(\mathbf{Z}_{n'}^y - \mathbf{Z}_n^y) - \mathbf{V}_i^y(\mathbf{Z}_{n'}^x - \mathbf{Z}_n^x) + \mathbf{Z}_n^y \mathbf{Z}_{n'}^x - \mathbf{Z}_{n'}^y \mathbf{Z}_n^x}{(\mathbf{Z}_{n'}^y - \mathbf{Z}_n^y)(\mathbf{V}_i^x - \mathbf{V}_j^x) - (\mathbf{Z}_{n'}^x - \mathbf{Z}_n^x)(\mathbf{V}_i^y - \mathbf{V}_j^y)} \\ t = \mu_{n'n} = \frac{\mathbf{V}_i^x - \mathbf{Z}_n^y + s(\mathbf{V}_j^y - \mathbf{V}_i^y)}{\mathbf{Z}_{n'}^y - \mathbf{Z}_n^y}. \end{cases}$$

Questo è il codice relativo alla classe:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import random as rnd
4 from matplotlib.tri import Triangulation
5 from numpy.linalg import norm
6
7 class DomainTriangulation:
8     def __init__(self, startTr: Triangulation):
9         self.startTr = startTr
10
11     # Creazione della triangolazione raffinata.
12     self.PowellSabin()
13
14     def PowellSabin(self):
15         # Esegue raffinamento di Powell-Sabin.
16         # lamb, mu, xi sono dizionari che memorizzano le coordinate
17         # baricentriche necessarie alla costruzione della Powell-Sabin B-spline
18
19         # R_ij = lamb[i,j] * V_i + lamb[j,i] * V_j;
20         # R_ij = mu[n,n'] * Z_n + mu[n',n] * Z_n';
21         # Z_n = xi[n,i] * V_i + xi[n,j] * V_j + xi[n,k] * V_k.
22         self.lamb = {}
23         self.mu = {}
24         self.xi = {}
25
26         # incenters e splitPoints memorizzano rispettivamente gli
27         # incentri e punti di suddivisione sui lati.
28         # incenters[n] = Z_n;
29         # splitPoints[i,j] = (R_ij, b) con i < j e b==True <=> V_iV_j e'
30         # un lato di bordo;
31         self.incenters = {}
32         self.splitPoints = {}
33
34         # TList[i] e' una lista di elementi della forma (tpos, j, in):
35         # tpos e' la posizione in PS_triangles di un triangolo di Powell-Sabin
36         # avente V_i come vertice, j e' l'indice del corrispettivo R_ij e in l'
37         # incentro Z_n.
38         self.TList = {}
39
40         # Salvo i punti della triangolazione iniziale in un solo array.
41         self.startPoints = np.array(list(zip(self.startTr.x, self.startTr.y)))
42
43         # startEges e' un dizionario che memorizza per ogni triangolo la
44         # lista di lati (i,j) con i<j tali che il lato V_iV_j stia nel
45         # triangolo n.
46         self.startEdges = {}
47
48         for n in range(len(self.startTr.triangles)):
49             # Salvo le coordinate baricentriche dell'incentro qualora
50             # non abbia ancora trattato il presente triangolo.
51             self.saveIncenter(n)
52
53             # Salvo le coordinate baricentriche dei punti di
54             # suddivisione.
55             self.saveSplitPoints(n)
```

```

46     # Inizializzo lista di triangoli e punti del raffinamento PS.
47     PS_triangles = []
48     trRow = 0
49     PS_points = []
50     ptRow = 0
51
52     # Colloco i nuovi triangoli: il dizionario pointsPos memorizza
53     # le posizioni-riga (ptRow) in PS_points dei punti già collocati.
54     pointsPos = {}
55     for n in range(len(self.startTr.triangles)):
56         # collocamento dell'incentro
57         PS_points.append(self.incenters[n])
58         pointsPos['in' + str(n)] = ptRow
59         ptRow += 1
60
61         for edge in self.startEdges[n]:
62             # collocamento del punto di suddivisione
63             if not edge in pointsPos:
64                 PS_points.append(self.splitPoints[edge][0])
65                 pointsPos['ed' + str(edge)] = ptRow
66                 ptRow += 1
67
68             for v in edge:
69                 # collocamento dei vecchi vertici
70                 if not v in pointsPos:
71                     PS_points.append(self.startPoints[v])
72                     pointsPos['pt' + str(v)] = ptRow
73                     ptRow += 1
74
75             # collocamento dei nuovi triangoli e aggiornamento
76             TList.
77                 PS_triangles.append(np.array([pointsPos['pt'+str(v)]
78 ,pointsPos['in'+str(n)],pointsPos['ed'+str(edge)]]))
79                 if not v in self.TList:
80                     self.TList[v] = []
81                     self.TList[v].append((trRow, edge[(edge.index(v)+1)
82 ], self.incenters[n]))
82                 trRow += 1
83
84             # Creo l'oggetto triangolazione PS-raffinata.
85             self.PS_points = np.array(PS_points)
86             self.PS_triangles = np.array(PS_triangles)
87             self.PowellSabinTr = Triangulation(np.array(PS_points)[:,0],np.
88 array(PS_points)[:,1],np.array(PS_triangles))
89
90             def saveIncenter(self, n: int):
91                 # Esegue l'inserimento dell'incentro per il triangolo n-esimo
92                 solo
93                 # se non è già stato effettuato.
94                 if not (n,self.startTr.triangles[n,0]) in self.xi:
95                     # Vertici del triangolo corrente.
96                     v0 = self.startPoints[self.startTr.triangles[n,0]]
97                     v1 = self.startPoints[self.startTr.triangles[n,1]]
98                     v2 = self.startPoints[self.startTr.triangles[n,2]]
99
100                     # Coordinate baricentriche dell'incentro.
101                     perim = norm(v1-v0) + norm(v2-v1) + norm(v0-v2)
102                     self.xi[n,self.startTr.triangles[n,0]] = norm(v2-v1)/perim

```

```

98         self.xi[n,self.startTr.triangles[n,1]] = norm(v0-v2)/perim
99         self.xi[n,self.startTr.triangles[n,2]] = norm(v1-v0)/perim
100
101     # Coordinate cartesiane dell'incentro.
102     self.incenters[n] = self.xi[n,self.startTr.triangles[n,0]]*
v0 + self.xi[n,self.startTr.triangles[n,1]]*v1 + self.xi[n,self.
startTr.triangles[n,2]]*v2
103
104     def saveSplitPoints(self, n: int):
105         # Esegue l'inserimento dei punti di suddivisione per il
106         # triangolo n-esimo.
107         for tr_index in range(3):
108             # Vertici del lato corrente.
109             i = min(self.startTr.triangles[n,tr_index],self.startTr.
triangles[n,(tr_index+1)%3])
110             j = max(self.startTr.triangles[n,tr_index],self.startTr.
triangles[n,(tr_index+1)%3])
111             vi = self.startPoints[i]
112             vj = self.startPoints[j]
113
114             # Aggiorno startEdges.
115             if not n in self.startEdges:
116                 self.startEdges[n] = []
117                 self.startEdges[n].append((i,j))
118
119             # Lato al bordo.
120             if self.startTr.neighbors[n,tr_index] == -1:
121                 self.lamb[i,j] = 1/2
122                 self.lamb[j,i] = 1/2
123                 self.splitPoints[i,j] = (1/2 * vi + 1/2 * vj, True)
124
125             # Lato interno.
126             else:
127                 # Ricerca del triangolo adiacente ed inserimento del
128                 # relativo incentro, se necessario.
129                 n_prime = self.startTr.neighbors[n,tr_index]
130                 self.saveIncenter(n_prime)
131
132                 # Selezione degli incentri coinvolti nella suddivisione.
133                 zn = self.incenters[n]
134                 zn_prime = self.incenters[n_prime]
135
136                 # Calcolo delle coordinate baricentriche del punto di
137                 # suddivisione.
138                 diffz = zn_prime - zn
139                 diffv = vi - vj
140
141                 if diffz[0] != 0:
142                     self.lamb[j,i] = (vi[1]*diffz[0]-vi[0]*diffz[1]+zn
143 [0]*zn_prime[1]-zn_prime[0]*zn[1])/(diffz[0]*diffv[1]-diffz[1]*diffv
144 [0])
145                     self.lamb[i,j] = 1 - self.lamb[j,i]
146                     self.mu[n_prime,n] = (vi[0]-zn[0]-self.lamb[j,i]*
diffv[0])/diffz[0]
147                     self.mu[n,n_prime] = 1 - self.mu[n_prime,n]
148                 else:
149                     self.lamb[j,i] = (vi[0]*diffz[1]-vi[1]*diffz[0]+zn
150 [1]*zn_prime[0]-zn_prime[1]*zn[0])/(diffz[1]*diffv[0]-diffz[0]*diffv

```

```

[1])
145             self.lamb[i,j] = 1 - self.lamb[j,i]
146             self.mu[n_prime,n] = (vi[1]-zn[1]-self.lamb[j,i]*diffv[1])/diffz[1]
147             self.mu[n,n_prime] = 1 - self.mu[n_prime,n]
148
149         # Calcolo del punto di suddivisione.
150         self.splitPoints[i,j] = (self.lamb[i,j] * vi + self.lamb
151 [j,i] * vj, False)
152
153     def getMultiIndices(self):
154         # Metodo che genera tutti i multi-indici per spline cubiche.
155         return [(i,j,k) for i in range(4) for j in range(4) for k in
156         range(4) if i+j+k==3]
157
158     def getZnOpposites(self, n:int):
159         # Metodo che restituisce una lista di tuple (m,m'), dove m e m'
160         sono posizioni in PS_triangles di triangoli che condividono un lato
161         ZnRij, e una lista di tuple (i,j), ciascuna corrispondente all'Rij
162         condiviso.
163         res1 = []
164         res2 = []
165         for i in self.startTr.triangles[n]:
166             for el in self.TList[i]:
167                 if norm(el[2]-self.incenters[n]) == 0:
168                     res1.append((el[0],self.PowellSabinTr.neighbors[el
169 [0],1]))
170                     res2.append((i,el[1]))
171         return (res1, res2)
172
173     def getRijOpposite(self, n: int, edge: tuple):
174         # Metodo che restituisce la posizione m' del triangolo PS
175         giacente sul lato edge=(i,j), avente in comune il vertice Vi con il
176         triangolo PS di partenza.
177         n_prime = self.getCommonTriangle(n, edge)
178         (i,j) = edge
179         for el in self.TList[i]:
180             if el[1] == j and norm(el[2]-self.incenters[n_prime]) == 0:
181                 return el[0]
182         raise ValueError('Edge lato di bordo.')
183
184     def getCommonTriangle(self, n: int, edge: tuple):
185         # Metodo che restituisce la posizione in startTr.triangles del
186         triangolo avente in comune il lato edge=(i,j) con il triangolo in
187         posizione n dato.
188         (i,j) = edge
189         if not i in self.startTr.triangles[n] or not j in self.startTr.
190         triangles[n]:
191             raise ValueError('Inserted edge doesn not belong to triangle
192 .')
193
194         tri = self.startTr.triangles[n]
195         (ind1,ind2) = (min(list(tri).index(i),list(tri).index(j)), max(
196         list(tri).index(i),list(tri).index(j)))
197         if (ind1,ind2) == (0,2):
198             return self.startTr.neighbors[n,2]
199         else:
200             return self.startTr.neighbors[n,ind1]

```

```

188
189     def domainPlot(self, showTitle=False, showPS=True, showDP=False,
190     labels=True, legend=True, showExecute=True, linewidths=0.6, fontsize
191     =8, color='#5a5e99', colorPS='#969dff', save=False):
192         # Visualizza il dominio della spline con relative triangolazioni
193         .
194         # showTitle: se True mostra il titolo;
195         # showPS: se True mostra raffinamento di Powell-Sabin;
196         # showDP: se True mostra proiezioni sul dominio dei punti di
197         # controllo;
198         # labels: se True mostra nomi dei punti;
199         # legend: se True mostra la legenda;
200         # showExecute: se True esegue plt.show() dentro il metodo;
201         # linewidths: spessore delle linee;
202         # fontsize: la dimensione del font per i punti;
203         # color: il colore per la triangolazione originale;
204         # colorPS: il colore per la triangolazione Powell-Sabin;
205         # save: se True, salva l'immagine.
206         if showTitle:
207             plt.title('Triangolazione iniziale e suddivisione Powell-
208 Sabin.')
209
210         if showExecute:
211             plt.axis('equal')
212             plt.axis('off')
213
214         # Visualizzazione della triangolazione originale e, se richiesto
215         ,
216         # del relativo raffinamento.
217         plt.triplot(self.startTr, color=color, linewidth=linewidths,
218         label='triangolazione $\Delta$')
219         if showPS:
220             plt.triplot(self.PowellSabinTr, linestyle='dotted',
221             linewidth=linewidths, color=colorPS, label='triangolazione $\Delta_{PS}$')
222
223         # Etichettatura dei punti, se richiesta.
224         if labels:
225             incr= 0.01
226             # incentri
227             for n in self.incenters:
228                 x = self.incenters[n][0]
229                 y = self.incenters[n][1]
230                 plt.text(x+incr,y+incr,'$Z_{'+str(n+1)}$',fontsize=
231                 fontsize)
232
233             # vertici
234             for v in range(len(self.startPoints)):
235                 x = self.startPoints[v,0]
236                 y = self.startPoints[v,1]
237                 plt.text(x+incr,y+incr,'$V_{'+str(v+1)}$',fontsize=
238                 fontsize)
239
240             # punti suddivisione
241             for r in self.splitPoints:
242                 x = self.splitPoints[r][0][0]
243                 y = self.splitPoints[r][0][1]
244                 plt.text(x+incr,y+incr,'$R_{'+str(r[0]+1)}+', '+str(r

```

```

[1]+1)+'}$', fontsize=fontsize)

235
236     # Visualizzazione di punti di controllo (con IDM=insieme di
determinazione minimo),
237     # se richiesta.
238     if showDP:
239         multiInd = self.getMultiIndices()

240
241     # Inizializzazione liste di punti di controllo e insieme di
determinazione minimo.
242     DP = []
243     MDS = []

244
245     for v in range(len(self.startPoints)):
246         # Per ogni vertice un triangolo adiacente e' scelto a
caso.
247         trChosen = self.TList[v][rnd.randint(0, len(self.TList[v])
])-1)][0]

248
249         for tr_index, w, _ in self.TList[v]:
250             triangle = self.PS_triangles[tr_index]
# Veritci del triangolo corrente.
251             p1 = self.PS_points[triangle[0]]
252             p2 = self.PS_points[triangle[1]]
253             p3 = self.PS_points[triangle[2]]

254
255             for (i,j,k) in multiInd:
# i vertici della triangolazione originale sono
nell'IDM
256                 if (i,j,k) == (3,0,0):
257                     MDS.append(p1)

258
259                 # i baricentri dei triangoli PS sono nell'IDM
260                 elif (i,j,k) == (1,1,1):
261                     MDS.append(1/3*p1 + 1/3*p2 + 1/3*p3)

262
263                 # i punti a indici (2,1,0) e (2,0,1) per un
triangolo scelto a caso sono nell'IDM
264                 elif tr_index == trChosen and ((i,j,k) ==
(2,1,0) or (i,j,k) == (2,0,1)):
265                     MDS.append(i/3*p1 + j/3*p2 + k/3*p3)

266
267                 # i punti a indici (1,0,2) se appartenenti ad un
lato di bordo sono nell'IDM
268                 elif self.splitPoints[min(v,w),max(v,w)][1] and
(i,j,k) == (1,0,2):
269                     MDS.append(i/3*p1 + j/3*p2 + k/3*p3)

270
271                 # tutti gli altri sono normali punti di
controllo
272                 else:
273                     DP.append(i/3*p1 + j/3*p2 + k/3*p3)

274
275                 # Salvataggio dei punti di controllo e dell'IDM.
276                 self.DP = np.array(DP)
277                 self.MDS = np.array(MDS)

278
279                 # Visualizzazione dei punti di controllo e IDM.

```

```

282         plt.plot(self.DP[:,0],self.DP[:,1], color=colorPS, linestyle
283 = ' ', marker='o', markersize=2)
284         plt.plot(self.MDS[:,0],self.MDS[:,1], color=color, linestyle
285 = ' ', marker='o', markersize=2, label='ins. determ. minimo')
286
287     # Esecuzioni di legenda, plt.show() e salvataggio, se richieste.
288     if showExecute:
289         if legend:
290             (h,l) = plt.gca().get_legend_handles_labels()
291             if showDP:
292                 plt.legend(handles=[h[0],h[2],h[4]],labels=[l[0],l
293 [2],l[4]], fontsize='x-small')
294             else:
295                 plt.legend(handles=[h[0],h[2]],labels=[l[0],l[2]],
296 fontsize='x-small')
297             if save:
298                 name = input('Nome del file?\n')
299                 plt.savefig(name, dpi=600)
300             plt.show()

```

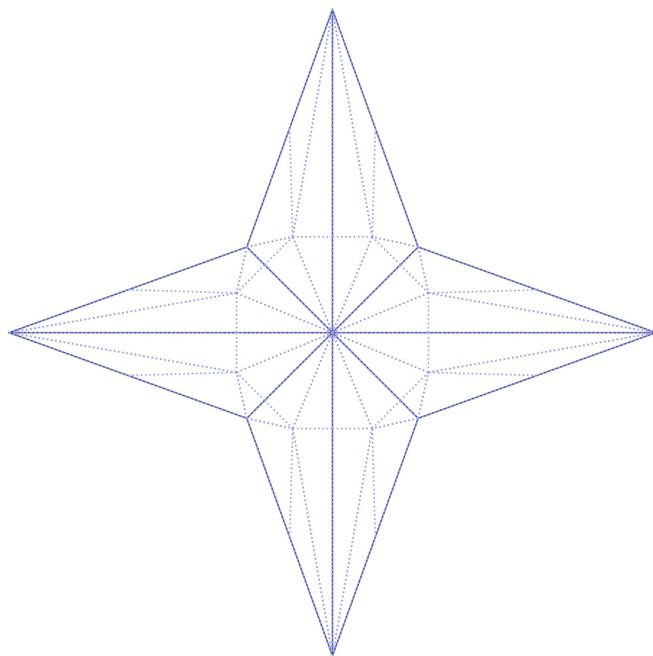
### 2.2.1 Alcuni esempi di utilizzo

Carichiamo la triangolazione relativa al dominio della stella a 4 punte e visualizziamo le triangolazioni  $\Delta$ ,  $\Delta_{PS}$ .

```

1 points_x = np.array([8, 9.06, 12, 9.06, 8, 6.94, 4, 6.94, 8])
2 points_y = np.array([2, 4.94, 6, 7.06, 10, 7.06, 6, 4.94, 6])
3 triangles = np.array([[0, 1, 8], [0, 7, 8], [1, 2, 8], [2, 3, 8],
4 [3, 4, 8], [4, 5, 8], [5, 6, 8], [6, 7, 8]])
5 tr1 = Triangulation(points_x, points_y, triangles)
6 dm = DomainTriangulation(tr1)
7 dm.domainPlot(labels=False, legend=False)

```

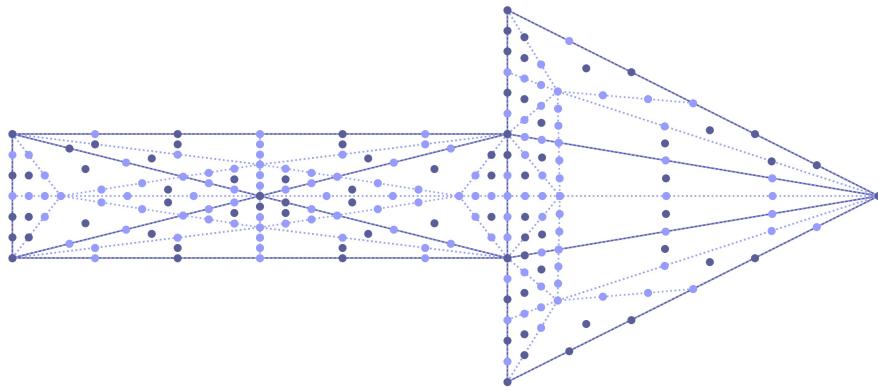


Vediamo ora come si visualizzano i punti dominio, evidenziando con colore più scuro quelli appartenenti all'insieme di determinazione minimo: esaminiamo nel dettaglio il caso del dominio a forma di freccia.

```

8  points_x = np.array([1, 3, 5, 5, 8, 5, 5, 1])
9  points_y = np.array([0, 0.5, 0, -1, 0.5, 2, 1, 1])
10 tr = np.array([[0, 2, 1], [1, 2, 6], [1, 6, 7], [0, 1, 7],
11           [2, 3, 4], [2, 4, 6], [4, 5, 6]])
12 tr2 = Triangulation(points_x, points_y, tr)
13 dm = DomainTriangulation(tr2)
14 dm.domainPlot(labels=False, legend=False, showDP=True)

```



## 2.3 La classe LiftTriangles

Introduciamo gli strumenti necessari alla costruzione e visualizzazione delle strutture di sollevamento  $\mathcal{Q}_i^v$ ,  $\mathcal{Q}_{i,n}^t$ ,  $\mathcal{Q}_{i,j}^e$ . Sebbene non sia necessario mostrare tali strutture esplicitamente per la realizzazione delle superficie spline, tale classe, oltre a soddisfare le esigenze di illustrazione in fase di esposizione, ricava anche i fondamentali  $\alpha_{i,r}$ ,  $\alpha_{i,r}^x$ ,  $\alpha_{i,r}^y$ ,  $\sigma_m^t$ ,  $\sigma_{i,j}^e$ .

Senza approfondire troppo tale questione, un'esigenza di ricerca ottimale di tali strutture deriva dal tentativo di dotare gli elementi in  $\mathbb{S}_{3,1}(\Delta_{PS})$  di una griglia di controllo alla quale appartengono opportuni sollevati di  $\mathcal{Q}_i^v$ ,  $\mathcal{Q}_{i,n}^t$ ,  $\mathcal{Q}_{i,j}^e$ : si veda [2] per maggiori informazioni. Sempre in [2] sono forniti valori ottimali per i coefficienti  $\sigma$ :

$$\sigma_m^t = \frac{\lambda_{ji}}{2} \max \left\{ \frac{\xi_{n,j}}{\lambda_{ji}} + \frac{\xi_{n,k}}{\lambda_{ki}}, 1 \right\}; \quad \sigma_{ij}^e = \frac{\lambda_{ji}}{2}.$$

Per quanto riguarda invece i triangoli  $\mathcal{Q}_i^v$  andremo a prendere, tra quelli che soddisfano i requisiti di contenimento del vertice  $V_i$  e relativi punti dominio a distanza 1, quello di area minima. Tale esigenza di minimalità è giustificata da una maggiore flessibilità della superficie al movimento di punti sulla griglia di controllo: ricordiamo che un sollevato di  $\mathcal{Q}_i^v$  giace su di essa.

Vediamo come si impone il problema di ottimizzazione relativo alla ricerca di  $\mathcal{Q}_i^v$  di area minima. Supponiamo che le nostre incognite siano i coefficienti  $\alpha_{i,r}$ ,  $\alpha_{i,r}^x$ ,  $\alpha_{i,r}^y$ , i punti  $\mathbf{Q}_{i,r}^v$  verranno adattati di conseguenza. Dalla relazione matriciale che definisce i coefficienti

$\alpha_{i,r}, \alpha_{i,r}^x, \alpha_{i,r}^y$  come coordinate baricentriche di  $\mathbf{V}_i, \mathbf{e}_1, \mathbf{e}_2$  rispettivamente:

$$\underbrace{\begin{bmatrix} \alpha_{i,1} & \alpha_{i,2} & \alpha_{i,3} \\ \alpha_{i,1}^x & \alpha_{i,2}^x & \alpha_{i,3}^x \\ \alpha_{i,1}^y & \alpha_{i,2}^y & \alpha_{i,3}^y \end{bmatrix}}_M \underbrace{\begin{bmatrix} \mathbf{Q}_{i,1}^x & \mathbf{Q}_{i,1}^y & 1 \\ \mathbf{Q}_{i,2}^x & \mathbf{Q}_{i,2}^y & 1 \\ \mathbf{Q}_{i,3}^x & \mathbf{Q}_{i,3}^y & 1 \end{bmatrix}}_N = \begin{bmatrix} \mathbf{V}_i^x & \mathbf{V}_i^y & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (12)$$

passando ai determinanti mediante la formula di Binet, sfruttando il fatto che i  $\mathbf{Q}_{i,r}$  sono fissati una volta conosciuti gli  $\alpha_{i,r}$ , che  $\det N$  è l'area del triangolo  $\mathcal{Q}_i^V$  otteniamo che tale misura è minima se  $\det M$  è massimo.

Visto che  $\alpha_{i,r}, \alpha_{i,r}^x, \alpha_{i,r}^y$  sono coordinate baricentriche possiamo considerare  $\alpha_{i,3}^x, \alpha_{i,3}^y$  fissate dalle prime due componenti e scegliere  $\alpha_{i,1}^x \alpha_{i,2}^y - \alpha_{i,1}^y \alpha_{i,2}^x$  come funzione obiettivo per la massimizzazione di tale determinante.

Veniamo ora ai vincoli: essendo  $\alpha_{i,r}, \alpha_{i,r}^x, \alpha_{i,r}^y$  coordinate baricentriche di punti e vettori imporremo

$$\begin{aligned} \alpha_{i,1} + \alpha_{i,2} + \alpha_{i,3} &= 1 \\ \alpha_{i,1}^x + \alpha_{i,2}^x + \alpha_{i,3}^x &= 0; \\ \alpha_{i,1}^y + \alpha_{i,2}^y + \alpha_{i,3}^y &= 0 \end{aligned} \quad (13)$$

visto inoltre che, ne abbiamo parlato nella nota 1.8,  $(b_{s,1}^v, b_{s,2}^v, b_{s,3}^v)$  per  $s = 1, 2, 3, 4$  sono coordinate baricentriche di un punto dominio a distanza al più 1 dal vertice  $\mathbf{V}_i$  imporre la loro positività assicura che tutti i tali punti dominio siano interni a  $\mathcal{Q}_i^V$ . Ci stiamo riferendo ai vincoli:

$$b_{1,r}^v, b_{2,r}^v, b_{3,r}^v, b_{4,r}^v \geq 0. \quad (14)$$

Ricapitolando, per ogni vertice  $\mathbf{V}_i$  andiamo a risolvere il problema:

$$\begin{cases} \max (\alpha_{i,1}^x \alpha_{i,2}^y - \alpha_{i,1}^y \alpha_{i,2}^x), \\ \alpha_{i,1} + \alpha_{i,2} + \alpha_{i,3} = 1 \\ \alpha_{i,1}^x + \alpha_{i,2}^x + \alpha_{i,3}^x = 0 \\ \alpha_{i,1}^y + \alpha_{i,2}^y + \alpha_{i,3}^y = 0 \\ \alpha_{i,r} + \frac{1}{3} \langle (\alpha_{i,r}^x, \alpha_{i,r}^y), \mathbf{R}_{i,j} - \mathbf{V}_i \rangle \geq 0 \quad \forall \langle \mathbf{V}_i, \mathbf{Z}_n, \mathbf{R}_{i,j} \rangle \in \Delta_{PS} \\ \alpha_{i,r} + \frac{1}{3} \langle (\alpha_{i,r}^x, \alpha_{i,r}^y), \mathbf{Z}_n - \mathbf{V}_i \rangle \geq 0 \quad \forall \langle \mathbf{V}_i, \mathbf{Z}_n, \mathbf{R}_{i,j} \rangle \in \Delta_{PS} \end{cases} \quad (15)$$

e a ricavare le coordinate dei vertici  $\mathbf{Q}_{i,1}^V, \mathbf{Q}_{i,2}^V, \mathbf{Q}_{i,3}^V$  risolvendo il sistema lineare (12).

Vediamo come il problema (15) è adattato nella sintassi `scipy.optimize`: le variabili vengono assemblate nel vettore  $\boldsymbol{\alpha} := (\alpha_{i,1}, \alpha_{i,2}, \alpha_{i,3}, \alpha_{i,1}^x, \alpha_{i,2}^x, \alpha_{i,3}^x, \alpha_{i,1}^y, \alpha_{i,2}^y, \alpha_{i,3}^y)$ , vincoli lineari vengono inseriti nella forma `LinearConstraint` come illustrato di seguito:

- i vincoli (13) relativi alle coordinate baricentriche vengono invocati mediante il comando `LinearConstraint(A, v1, v2)` con

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix},$$

$$\mathbf{v}^{(1)} = \mathbf{v}^{(2)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},$$

in modo da rappresentare un vincolo lineare della forma  $A\boldsymbol{\alpha} = \mathbf{v}^{(1)}$ .

- veniamo invece ai (14):  $b_{1,r}^v = \alpha_{i,r} \geq 0$  per  $r = 1, 2, 3$  vengono modellizzati con

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\mathbf{v}^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{v}^{(2)} = \begin{bmatrix} +\infty \\ +\infty \\ +\infty \end{bmatrix}$$

in modo da avere  $\mathbf{v}^{(1)} \leq A\boldsymbol{\alpha} \leq \mathbf{v}^{(2)}$ .

I  $b_{2,r}^v, b_{3,r}^v, b_{4,r}^v \geq 0$  vengono gestiti triangolo per triangolo: supponiamo di lavorare su  $\langle \mathbf{V}_i, \mathbf{Z}_n, \mathbf{R}_{i,j} \rangle \in \Delta_{PS}$ .

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \frac{1}{3}(\mathbf{R}_{i,j}^x - \mathbf{V}_i^x) & 0 & 0 & \frac{1}{3}(\mathbf{R}_{i,j}^y - \mathbf{V}_i^y) & 0 & 0 \\ 0 & 1 & 0 & 0 & \frac{1}{3}(\mathbf{R}_{i,j}^x - \mathbf{V}_i^x) & 0 & 0 & \frac{1}{3}(\mathbf{R}_{i,j}^y - \mathbf{V}_i^y) & 0 \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{3}(\mathbf{R}_{i,j}^x - \mathbf{V}_i^x) & 0 & 0 & \frac{1}{3}(\mathbf{R}_{i,j}^y - \mathbf{V}_i^y) \\ 1 & 0 & 0 & \frac{1}{3}(\mathbf{Z}_n^x - \mathbf{V}_i^x) & 0 & 0 & \frac{1}{3}(\mathbf{Z}_n^y - \mathbf{V}_i^y) & 0 & 0 \\ 0 & 1 & 0 & 0 & \frac{1}{3}(\mathbf{Z}_n^x - \mathbf{V}_i^x) & 0 & 0 & \frac{1}{3}(\mathbf{Z}_n^y - \mathbf{V}_i^y) & 0 \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{3}(\mathbf{Z}_n^x - \mathbf{V}_i^x) & 0 & 0 & \frac{1}{3}(\mathbf{Z}_n^y - \mathbf{V}_i^y) \end{bmatrix}}_A,$$

$$\mathbf{v}^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{v}^{(2)} = \begin{bmatrix} +\infty \\ +\infty \\ +\infty \\ +\infty \\ +\infty \\ +\infty \end{bmatrix}$$

Veniamo ora all'implementazione vera e propria.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib.tri import Triangulation
4 from scipy.optimize import LinearConstraint, minimize
5 from numpy.linalg import norm, solve
6 from DomainTriangulation import DomainTriangulation
7
8 class LiftTriangles:
9     def __init__(self, startTr: Triangulation):
10         # Inizializzazione delle triangolazioni sul dominio.
11         self.domain = DomainTriangulation(startTr)
12
13         # Creazione dei triangoli di sollevamento.
14         self.makeLiftTriangles()
15
16     def makeLiftTriangles(self):
17         # Inizializzazione liste dei triangoli di sollevamento e
18         # relativi vertici.
19         VSollTrList = []
20         VSollVxList = []
21         TSollTrList = []
22         TSollVxList = []
23         TSollLabels = []           # lista di pedici dei vertici per i
24         triangoli TSoll.

```

```

23     VSollVxList = []
24     VSollLabels = []
25
26     # TSollFinder e' un dizionario tale da associare ad ogni coppia
27     # (n,i), dove n e' un triangolo della triangolazione di partenza e V_i
28     # un suo vertice, la posizione in TSollTr del triangolo di sollevamento
29     # corrispondente.
30     self.TSollFinder = {}
31
32     # Inizializzazione liste dei coefficienti sigma.
33     # TSollSigma[m] = sigma_m^t;
34     # BESolSigma[i,j] = sigma_(i,j)^e
35     self.TSollSigma = {}
36     self.ESolSigma = {}
37
38     # Inizializzazione del dizionario dei coefficienti alpha.
39     # alpha[i][j] = alpha_i,(j+1) j=0,1,2;
40     # alpha[i][j] = alpha^x_i,(j-2) j=3,4,5;
41     # alpha[i][j] = alpha^y_i,(j-5) j=6,7,8.
42     self.alpha = {}
43
44     # Triangoli di sollevamento associati ai vertici.
45     for i in range(len(self.domain.startPoints)):
46         # Ricavo il triangolo per il vertice corrente.
47         liftTr = self.findVertexLiftTriangle(i)
48
49         # Aggiungo punti e triangolo in lista.
50         codTr = []
51         for pt in liftTr:
52             VSollVxList.append(pt)
53             codTr.append(len(VSollVxList)-1)
54             VSollTrList.append(np.array(codTr))
55
56         # Creo la triangolazione contenente i triangoli di sollevamento
57         # associati ai vertici.
58         self.VSollVx = np.array(VSollVxList)
59         self.VSollTr = np.array(VSollTrList)
60         self.VSoll = Triangulation(self.VSollVx[:,0], self.VSollVx[:,1],
61         self.VSollTr)
62
63         # Triangoli di sollevamento associati alla triangolazione.
64         for n in range(len(self.domain.startTr.triangles)):
65             for i in self.domain.startTr.triangles[n]:
66                 codTr = []
67
68                 # Aggiungo il punto S_{i,n}^t.
69                 TSollVxList.append(2/3*self.domain.startPoints[i]+1/3*
70 self.domain.incenters[n])
71                 TSollLabels.append('$S^t_{'+str(i+1)+','+str(n+1)+'}$')
72                 codTr.append(len(TSollVxList)-1)
73
74                 # Ricerca dei triangoli PS m e m' a vertice V_i.
75                 mList, jList = self.getPSTriangles(n,i)
76
77                 # Calcolo i coefficienti sigma^t.
78                 mx = max(self.domain.xi[n,jList[0]]/self.domain.lamb[
79                 jList[0],i] + self.domain.xi[n,jList[1]]/self.domain.lamb[jList[1],i
80                 ], 1)

```

```

73         for l in range(2):
74             self.TSollSigma[mList[l]] = self.domain.lamb[jList[l]
75             ],i]/2 * mx
76
77             # Aggiunta dei punti Q_m^t e Q_m'^t.
78             for l in range(2):
79                 TSollVxList.append(TSollVxList[len(TSollVxList)-1-1]
80                 + 2/3*self.TSollSigma[mList[l]]*(self.domain.startPoints[jList[l]]-
81                 self.domain.startPoints[i]))
82                 TSollLabels.append('$Q^t_{'+str(mList[l]+1)}')
83                 codTr.append(len(TSollVxList)-1)
84
85             # Aggiunta del triangolo di sollevamento e aggiornamento
86             # di TSollFinder.
87             TSollTrList.append(np.array(codTr))
88             self.TSollFinder[n,i] = len(TSollTrList)-1
89
90             # Creo la triangolazione contenente i triangoli di sollevamento
91             # associati ai vertici.
92             self.TSollVx = np.array(TSollVxList)
93             self.TSollTr = np.array(TSollTrList)
94             self.TSollVxLabels = TSollLabels
95             self.TSoll = Triangulation(self.TSollVx[:,0], self.TSollVx[:,1],
96             self.TSollTr)
97
98             # Segmenti di sollevamento associati ai lati di bordo.
99             # ESollPos[i,j] memorizza la posizione di S^e_{ij} in ESollVx.
100            self.ESollPos = {}
101            for edge in self.domain.splitPoints:
102                # se il lato edge sta nel bordo si ricavano i punti S^e_ij e
103                Q_{ij}^e
104                if self.domain.splitPoints[edge][1]:
105                    for i in range(2):
106                        # punti S^e_{ij}
107                        ESollVxList.append(2/3*self.domain.startPoints[edge[1]] +
108                        1/3*self.domain.startPoints[edge[0]])
109                        ESollLabels.append('$S^e_{'+str(edge[i]+1)}, '+str(
110                        edge[(i+1)%2]+1)'}
111
112                        # aggiorno ESollPos
113                        self.ESollPos[edge[i],edge[(i+1)%2]] = len(
114                        ESollVxList)-1
115
116                        # coefficienti sigma^e_{ij}
117                        self.ESollSigma[edge[i],edge[(i+1)%2]] = self.domain
118                        .lamb[edge[(i+1)%2],edge[i]]/2
119
120                        # punti Q^e_{ij}
121                        ESollVxList.append(ESollVxList[-1] + 2/3 * self.
122                        ESollSigma[edge[i],edge[(i+1)%2]] * (self.domain.startPoints[edge[(i+1)%2]] -
123                        self.domain.startPoints[edge[i]]))
124                        ESollLabels.append('$Q^e_{'+str(edge[i]+1)}, '+str(
125                        edge[(i+1)%2]+1)'}
126
127            # Salvo vertici ed etichette dei segmenti di sollevamento al
128            # bordo.
129            self.ESollVxLabels = ESollLabels
130            self.ESollVx = np.array(ESollVxList)

```

```

116
117     def findVertexLiftTriangle(self, i: int):
118         # Inizializzazione della lista di vincoli LinearConstraint.
119         ConstrList = []
120
121         # Vincoli delle coordinate baricentriche.
122         A = np.array([[1,1,1,0,0,0,0,0,0],
123                      [0,0,0,1,1,1,0,0,0],
124                      [0,0,0,0,0,0,1,1,1]])
125         lb = ub = np.array([1,0,0])
126         ConstrList.append(LinearConstraint(A,lb,ub))
127
128         # Vincolo di non-negativita' degli alfa_i,r.
129         A = np.eye(3,9)
130         lb = np.zeros(3)
131         ub = np.infty * np.ones(3)
132         ConstrList.append(LinearConstraint(A,lb,ub))
133
134         # Vincoli associati ai triangoli.
135         for tr_index,_,_ in self.domain.TList[i]:
136             tr = self.domain.PS_triangles[tr_index]
137
138             # Costruzione della matrice di vincolo LinearConstraint.
139             def matr(diffx, diffz):
140                 return np.block([[np.eye(3), diffx[0]/3 * np.eye(3),
141                                 diffx[1]/3 * np.eye(3)],
142                                 [np.eye(3), diffz[0]/3 * np.eye(3),
143                                 diffz[1]/3 * np.eye(3)]])
144
145             # Creazione del vincolo associato a tr e inserimento in
146             # lista.
147             lb = np.zeros(6)
148             ub = np.infty * np.ones(6)
149             Vi = self.domain.PS_points[tr[0]]
150             Zn = self.domain.PS_points[tr[1]]
151             Rij = self.domain.PS_points[tr[2]]
152             ConstrList.append(LinearConstraint(matr(Rij-Vi,Zn-Vi),lb,ub))
153
154         # vettore iniziale
155         a0 = np.array([1/3,1/3,1/3,1,-1,0,2,-1,1])
156
157         # funzione obiettivo
158         def f(a):
159             return -a[3]*a[7]+a[6]*a[4]
160
161         # Esecuzione del problema di ottimizzazione per la ricerca del
162         # triangolo di area minima.
163         alpha = minimize(f,a0,constraints=ConstrList).x
164
165         # Memorizzazione dei coefficienti alpha.
166         self.alpha[i] = alpha
167
168         # Ricavo le coordinate del triangolo di sollevamento risolvendo
169         # opportuno sistema lineare.
170         M = np.block([[alpha[0:3]],[alpha[3:6]],[alpha[6:9]]])
171         B = np.block([[self.domain.startPoints[i], 1],[1,0,0],[0,1,0]])
172         Q = solve(M,B)

```

```

168
169     # Restituisco le coordinate Qi,1 Qi,2 Qi,3 ottenute.
170     return (Q[0,0:2], Q[1,0:2], Q[2,0:2])
171
172 def getPSTriangles(self, n: int, i: int):
173     # Fornisce (mList, jList): mList e' la lista delle posizioni
174     # in PS_Triangles dei triangoli di PS aventi Vi come vertice,
175     # jList e' la lista
176     # dei Vj adiacenti.
177     mList = []
178     jList = []
179     for el in self.domain.TList[i]:
180         if norm(el[2]-self.domain.incenters[n]) == 0:
181             mList.append(el[0])
182             jList.append(el[1])
183     return (mList, jList)
184
185 def liftPlot(self, showTitle=False, labels=True, legend=True,
186 showExecute=True, fontsize=7, \
187     save=False, onlyLifts=False):
188     # Visualizza i triangoli di sollevamento sul dominio.
189     # showTitle: se True mostra il titolo;
190     # labels: se True mostra nomi dei punti;
191     # legend: se True mostra la legenda;
192     # fontsize: la dimensione del font per i punti;
193     # showExecute: se True esegue plt.show() dentro il metodo;
194     # fontsize: la dimensione del font per i punti;
195     # save: se True salva l'immagine;
196     #onlyLifts: se True mostra soltanto le strutture di sollevamento
197
198     if showTitle:
199         plt.title('Triangoli di sollevamento per la costruzione di
una base $\\mathbb{S}_3^1(\\Delta_{PS})$.')
200
201     # Se richiesta, mostro triangolazione sottostante.
202     if not onlyLifts:
203         self.domain.domainPlot(legend=False, showExecute=False,
204 labels=False, color='#c6c5c6', \
205             colorPS='#c6c5c6', showDP=True)
206
207         # aggiustamento degli assi
208         plt.axis('equal')
209
210         # triangoli di sollevamento relativi ai vertici
211         plt.triplot(self.VSoll, color='#fd3612', linewidth=0.9, label='
triangoli $\\mathcal{Q}_i^v$')
212
213         # triangoli di sollevamento relativi ai triangoli PS
214         plt.triplot(self.TSoll, color='#0e03fb', linewidth=0.9, label='
triangoli $\\mathcal{Q}_{i,n}^t$')
215
216         # segmenti di sollevamento ai lati di bordo
217         for i in range(int(len(self.ESollVx)/2)):
218             plt.plot(self.ESollVx[2*i:2*(i+1),0], self.ESollVx[2*i:2*(i
+1),1], \
219                 label='segmenti $\\mathcal{Q}_{i,j}^e$', color='#fac900',
220                 linewidth=0.9)

```

```

217     # Etichettatura dei punti, se richiesta.
218     if labels:
219         incr= 0.01
220         # vertici
221         for v in range(len(self.domain.startPoints)):
222             x = self.domain.startPoints[v,0]
223             y = self.domain.startPoints[v,1]
224             plt.text(x+incr,y+incr,'$V_{'+str(v+1)}$',fontsize=
225                     fontsize)
226
227         # punti associati ai vertici
228         for i in range(len(self.VSollTr)):
229             for j in range(len(self.VSollTr[i])):
230                 x = self.VSollVx[self.VSollTr[i,j]][0]
231                 y = self.VSollVx[self.VSollTr[i,j]][1]
232
233                 # non visualizzo etichetta quando ho coincidenza con
234                 # un vertice
235                 if norm(self.VSollVx[self.VSollTr[i,j]]-self.domain.
236 startPoints[i]) > 1e-2:
237                     plt.text(x+incr,y+incr,'$Q^v_{'+str(i+1)}{'+str
238 (j+1)}$',fontsize=fontsize)
239
240         # punti associati ai triangoli PS
241         for i in range(len(self.TSollVxLabels)):
242             x = self.TSollVx[i][0]
243             y = self.TSollVx[i][1]
244             plt.text(x+incr,y+incr,self.TSollVxLabels[i],fontsize=
245                     fontsize)
246
247         # segmenti al bordo
248         for i in range(len(self.ESollVxLabels)):
249             x = self.ESollVx[i][0]
250             y = self.ESollVx[i][1]
251             plt.text(x-4*incr,y-4*incr,self.ESollVxLabels[i],
252                     fontsize=fontsize)
253
254     # Esecuzioni di legenda, plt.show() e salvataggio, se richieste.
255     if showExecute:
256         if legend:
257             (h,l) = plt.gca().get_legend_handles_labels()
258             plt.legend(handles=[h[5],h[7],h[9]],labels=[l[5],l[7],l
259 [9]], fontsize='x-small')
260         if save:
261             name = input('Nome del file?\n')
262             plt.savefig(name, dpi=600)
263             plt.show()

```

### 2.3.1 Alcuni esempi di utilizzo

Mostriamo i  $\mathcal{Q}_i^v$ ,  $\mathcal{Q}_{i,n}^t$ ,  $\mathcal{Q}_{i,j}^e$  nel caso del dominio della stella a 5 punte: i triangoli  $\mathcal{Q}_i^v$  sono evidenziati in rosso, i  $\mathcal{Q}_{i,n}^t$  in blu e i segmenti  $\mathcal{Q}_{i,j}^e$  in giallo.

```

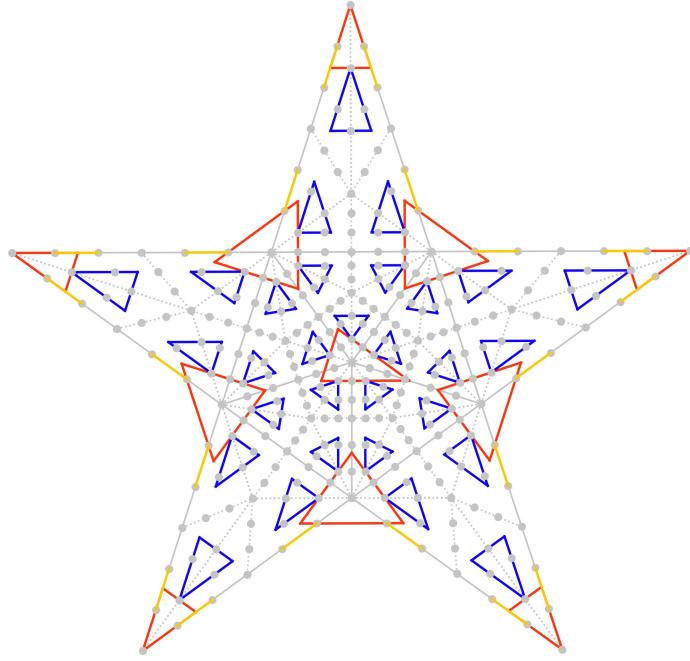
1 points_x = np.array([4.95, 7.86, 10.79, 9.66, 12.57, 8.97, 7.84, 6.74,
2           3.13, 6.05, 7.86])
2 points_y = np.array([3.09, 5.22, 3.11, 6.53, 8.66, 8.65, 12.08, 8.64,
           8.63, 6.52, 7.11])

```

```

3 tr = np.array([[0, 1, 9],[1, 2, 3],[3, 4, 5],[5, 6, 7],[7, 8, 9],
4 [1, 9, 10],[1, 3, 10],[3, 5, 10],[5, 7, 10],[7, 9, 10]])
5 tr3 = Triangulation(points_x, points_y,tr)
6 lt = LiftTriangles(tr3)
7 lt.liftPlot(save=True,labels=False, legend=False)

```



## 2.4 La classe PS\_Spline

Veniamo ora a quello che è il cuore dell'implementazione: introduciamo le tecniche di costruzione delle superfici  $\mathbb{S}_{3,1}(\Delta_{PS})$  alla luce di quanto introdotto sinora.

Sappiamo che  $s \in \mathbb{S}_{3,1}(\Delta_{PS})$  è una superficie di Bézier triangolare lungo ciascun elemento di  $\Delta_{PS}$ , quello che andiamo a fare è il plotting di ciascuna di queste superfici il cui risultato complessivo risulterà  $C^1$ , a patto di aver svolto tutto correttamente.

In particolare,

- in una prima fase vengono calcolate tutte le altezze relative ai punti dominio. Si tratta della parte più difficile da trasporre in codice, si è scelto di percorrere un approccio bruto in tre differenti cicli (round) di calcolo: va da sé che un approccio di programmazione parallela risulterebbe più efficiente, soprattutto nel caso di triangolazioni più cospicue. Siamo tuttavia contenti della velocità di esecuzione nel caso dei nostri (relativamente) semplici esempi. Maggiori dettagli sul calcolo vengono forniti nel prossimo paragrafo.
  - $\Delta_{PS}$  viene raffinata con `UniformTriRefiner` e per ciascun punto di suddetta triangolazione l'immagine sulla superficie è calcolata in maniera stabile con l'algoritmo di De Casteljau triangolare.
- Quest'ultimo, ad esempio descritto in [1], consiste nel porre presso il punto  $\mathbf{P} \in \mathcal{T}$  di coordinate baricentriche  $(\tau_1, \tau_2, \tau_3)$  come immagine di  $s$  il valore  $b_{000}^{(3)}$ . I valori  $b_{ijk}^{(d)}$  sono calcolati con la regola ricorsiva seguente, implementata nel metodo

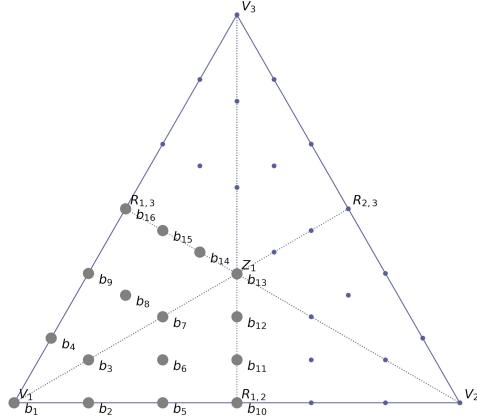
triDeCasteljau della classe PS\_Spline:

$$\begin{cases} b_{ijk}^{(0)} = b_{ijk} \\ b_{ijk}^{(d)} = \tau_1 b_{(i+1)jk}^{(d-1)} + \tau_2 b_{i(j+1)k}^{(d-1)} + \tau_3 b_{ij(k+1)}^{(d-1)} \end{cases} \quad (16)$$

#### 2.4.1 Dettagli sul calcolo delle altezze di Bézier

Le altezze relative ai punti dominio per  $s \in \mathbb{S}_{3,1}(\Delta_{PS})$  vengono calcolate cumulando i contributi dei vari elementi di base a supporto non nullo sullo specifico punto dominio, il tutto è poi completato raccordando in maniera  $C^1$ .

Sviluppiamo lo svolgimento su ogni triangolo  $\langle V_i, V_j, V_k \rangle$ , concentrandoci sul vertice  $V_i$ .



Si ricorda che, secondo la base di  $\mathbb{S}_{3,1}$  individuata, ogni  $s$  si scrive come:

$$s(\mathbf{P}) = \sum_{i=1}^{|\mathcal{V}|} \sum_{r=1}^3 c_{i,r}^v B_{i,r}^v(\mathbf{P}) + \sum_{m=1}^{|\Delta_{PS}|} c_m^t B_m^t(\mathbf{P}) + \sum_{l=1}^{|\partial\mathcal{E}_{PS}|} c_l^e B_l^e(\mathbf{P}).$$

Descriviamo anzitutto il primo "round" di calcolo.

- Lungo le altezze  $b_1, b_2, b_3, b_4$  si ha il contributo delle sole  $b_{s,r}^v$  per  $r = 1, 2, 3$ , e dunque

$$\begin{aligned} b_1 &= c_{i,1}^v \alpha_{i,1} + c_{i,2}^v \alpha_{i,2} + c_{i,3}^v \alpha_{i,3} \\ b_2 &= b_{2,1}^v c_{i,1}^v + b_{2,2}^v c_{i,2}^v + b_{2,3}^v c_{i,3}^v \\ b_3 &= b_{3,1}^v c_{i,1}^v + b_{3,2}^v c_{i,2}^v + b_{3,3}^v c_{i,3}^v \\ b_4 &= b_{4,1}^v c_{i,1}^v + b_{4,2}^v c_{i,2}^v + b_{4,3}^v c_{i,3}^v, \end{aligned}$$

ricordando che i  $b_{s,r}^v$  per  $r = 1, 2, 3$  e  $s = 1, 2, 3, 4$  si calcolano come illustrato alla nota 1.3.

- All'altezza  $b_6$  (ed equivalentemente  $b_8$  per il triangolo  $\mathcal{T}'_{m'} = \langle V_i, Z_n, R_{i,k} \rangle \in \Delta_{PS}$ ) contribuiscono i  $b_{6,r}^v$  relativi al vertice  $V_i$  e il  $b_3^t$  relativo al triangolo  $\mathcal{T}'_m = \langle V_i, Z_n, R_{i,j} \rangle \in \Delta_{PS}$ .

Visto che  $b_3^t = \frac{\lambda_{ji}}{2\sigma_m^t}$  e  $b_{6,r}^v = \left(1 - \frac{\lambda_{ji}}{2\sigma_m^t}\right) b_{3,r}^v$ , cumulando il tutto otteniamo:

$$\begin{aligned} b_6 &= \left(1 - \frac{\lambda_{ji}}{2\sigma_m^t}\right) b_{3,r}^v + \frac{\lambda_{ji}}{2\sigma_m^t} c_m^t \\ b_8 &= \left(1 - \frac{\lambda_{ki}}{2\sigma_{m'}^t}\right) b_{3,r}^v + \frac{\lambda_{ki}}{2\sigma_{m'}^t} c_{m'}^t \end{aligned}$$

- A  $b_7$  contribuiscono i  $b_{7,r}^v$  relativi al vertice  $\mathbf{V}_i$  e entrambi i  $b_5^t$  relativi ai triangoli  $\mathcal{T}'_m = \langle \mathbf{V}_i, \mathbf{Z}_n, \mathbf{R}_{i,j} \rangle, \mathcal{T}'_{m'} = \langle \mathbf{V}_i, \mathbf{Z}_n, \mathbf{R}_{i,k} \rangle$ . Visto che  $b_5^t = \frac{\xi_{n,j}}{2\sigma_m^t}, \frac{\xi_{n,k}}{2\sigma_{m'}^t}$  e che  $b_{7,r}^v = \left(1 - \frac{\xi_{n,j}}{2\sigma_m^t} - \frac{\xi_{n,k}}{2\sigma_{m'}^t}\right) b_{3,r}^v$ , otteniamo

$$b_7 = \left(1 - \frac{\xi_{n,j}}{2\sigma_m^t} - \frac{\xi_{n,k}}{2\sigma_{m'}^t}\right) b_3 + \frac{\xi_{n,j}}{2\sigma_m^t} c_m^t + \frac{\xi_{n,k}}{2\sigma_{m'}^t} c_{m'}^t.$$

- A  $b_{13}$  contribuiscono i  $b_{13,r}^v$  relativi ai vertici  $\mathbf{V}_i, \mathbf{V}_j, \mathbf{V}_k$  e i  $b_8^t$  relativi a tutti i triangoli  $\Delta_{PS}$  ottenuti suddividendo  $\langle \mathbf{V}_i, \mathbf{V}_j, \mathbf{V}_k \rangle$ . Visto che  $b_{13,r}^v = \xi_{n,i} b_{7,r}^v, b_8^t = \xi_{n,i} b_5^t$  e osservando che tutte e sole le altezze di base che contribuiscono a  $b_7$  sono le  $b_{7,r}^v$  e  $b_5^t$  otteniamo, cumulando ancora una volta il tutto:

$$b_{13} = \xi_{n,i} b_7^{(i)} + \xi_{n,j} b_7^{(j)} + \xi_{n,k} b_7^{(k)},$$

detti  $b_7^{(i)}, b_7^{(j)}, b_7^{(k)}$  i  $b_7$  relativi a triangoli contenenti vertici  $\mathbf{V}_i, \mathbf{V}_j, \mathbf{V}_k$  rispettivamente.

A questo punto per calcolare le restanti altezze risultano necessarie altezze associate a punti dominio interni ad altri triangoli in  $\Delta$ , è dunque necessario procedere con un secondo "round" di calcolo.

- Per quanto riguarda  $b_5$  (o analogamente  $b_9$ ) è necessario distinguere due casi: se siamo sul bordo basterà osservare che a  $b_5$  contribuiscono le altezze  $b_1^e$  e  $b_1^t$  ed avendo  $b_1^e = \frac{\lambda_{ji}}{2\sigma_{i,j}^e}$  otteniamo

$$b_5 = \left(1 - \frac{\lambda_{ji}}{2\sigma_{i,j}^e}\right) b_2 + \frac{\lambda_{ji}}{2\sigma_{i,j}^e} c_{i,j}^e.$$

Se invece non siamo sul bordo usiamo le condizioni di raccordo  $C^1$  e, detto  $b'_8$  l'altezza  $b_8$  del triangolo in  $\Delta_{PS}$  che condivide il lato  $\mathbf{V}_i \mathbf{R}_{ij}$ , otteniamo:

$$b_5 = \mu_{nn'} b_6 + \mu_{n'n} b'_8.$$

Terzo ed ultimo round di calcolo per i punti che rimangono: chiamiamo  $\tilde{b}_i$  l'altezza  $b_i$  del triangolo opposto rispetto al lato  $\overline{\mathbf{Z}_n \mathbf{R}_{i,j}}$ .

- Si utilizzano le condizioni di raccordo  $C^1$  ottenendo infine:

$$\begin{aligned} b_{10} &= \lambda_{ij} b_5 + \lambda_{ji} \tilde{b}_5 & b_{11} &= \lambda_{ij} b_6 + \lambda_{ji} \tilde{b}_6 & b_{12} &= \lambda_{ij} b_7 + \lambda_{ji} \tilde{b}_7 \\ b_{14} &= \lambda_{ik} b_7 + \lambda_{ki} \tilde{b}_7 & b_{15} &= \lambda_{ik} b_8 + \lambda_{ki} \tilde{b}_8 & b_{16} &= \lambda_{ik} b_9 + \lambda_{ki} \tilde{b}_9 \end{aligned}$$

Veniamo infine all'implementazione.

```

1 import math as mt
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib as mpl
5 from matplotlib.tri import Triangulation, UniformTriRefiner
6 from numpy.linalg import norm, solve
7 from mpl_toolkits.mplot3d import Axes3D

```

```

8  from DomainTriangulation import DomainTriangulation
9  from LiftTriangles import LiftTriangles
10
11 class PS_Spline:
12     def __init__(self, tr: Triangulation, Vcoeffs: np.array, Tcoeffs: np
.array, Ecoeffs: dict):
13         # Inizializzazione oggetti legati al dominio.
14         self.domain = DomainTriangulation(tr)
15         self.lift_triangles = LiftTriangles(tr)
16
17         # Controllo di conformita' di array e dizionario dei
18         # coefficienti.
19         if Vcoeffs.shape == (len(self.domain.startPoints),3):
20             self.Vcoeffs = Vcoeffs
21         else:
22             raise ValueError('Array dei coefficienti ai vertici non
conforme: richiesto (' + \
23                             str(len(self.domain.startPoints)) + ',3).')
24
25         if Tcoeffs.shape == (len(self.domain.PS_triangles),):
26             self.Tcoeffs = Tcoeffs
27         else:
28             raise ValueError('Array dei coefficienti ai triangoli PS non
conforme: richiesto (' + \
29                             str(len(self.domain.PS_triangles)) + ',).')
30
31         # Completamento del dizionario dei coefficienti ai lati di bordo
32
33         BoundEdgesList = []
34         self.Ecoeffs = Ecoeffs
35         for edge in self.domain.splitPoints:
36             if self.domain.splitPoints[edge][1]:
37                 BoundEdgesList.append((edge[0],edge[1]))
38                 BoundEdgesList.append((edge[1],edge[0]))
39
40             for el in BoundEdgesList:
41                 if not el in self.Ecoeffs:
42                     self.Ecoeffs[el] = 0
43
44         # Calcolo delle altezze dei punti di controllo.
45         self.setControlPoints()
46
47         # Calcolo dei valori immagine della superficie.
48         self.setSurface()
49
50     def setControlPoints(self):
51         # In controlPointsHeights[m,(i,j,k)] viene memorizzata l'altezza
52         # relativa al punto
53         # di controllo sul triangolo PS m individuato dalla tripla (i,j,
54         # k).
55         self.controlPointsHeights = {}
56
57         # Altezze sui vertici (punti (3,0,0))
58         for i in range(len(self.domain.startPoints)):
59             for tr,_,_ in self.domain.TList[i]:
60                 alpha = self.lift_triangles.alpha[i][0:3]
61                 self.controlPointsHeights[tr,(3,0,0)] = alpha[0]*self.
62                 Vcoeffs[i,0] + alpha[1]*self.Vcoeffs[i,1] + alpha[2]*self.Vcoeffs[i

```

```

,2]

58
59     # Primo round di calcolo delle altezze.
60     for n in range(len(self.domain.startTr.triangles)):
61         inHeight = []
62         for i in self.domain.startTr.triangles[n]:
63             # Ricavo la lista PSList dei triangoli PS nel triangolo
64             # afferenti al
65             # vertice i e dizionario jDict del corrispettivo j
66             # adiacente ad i.
67             PSLList, jList = self.lift_triangles.getPSTriangles(n,i)
68             inHeight.append((self.domain.xi[n,i],PSLList[0]))
69
70             # Recupero i coefficienti alpha.
71             alpha = self.lift_triangles.alpha[i]
72
73             for m_index in range(len(PSLList)):
74                 m = PSLList[m_index]
75                 m_prime = PSLList[(m_index+1)%2]
76                 j = jList[m_index]
77                 k = jList[(m_index+1)%2]
78
78                 # punti (2,0,1)
79                 (i_ord, j_ord) = (min(i,j),max(i,j))
80                 diffr = self.domain.splitPoints[i_ord,j_ord][0] -
81             self.domain.startPoints[i]
82                 b = np.zeros(3)
83                 b[0] = alpha[0] + 1/3*alpha[3]*diffr[0] + 1/3*alpha
84                 [6]*diffr[1]
85                 b[1] = alpha[1] + 1/3*alpha[4]*diffr[0] + 1/3*alpha
86                 [7]*diffr[1]
87                 b[2] = alpha[2] + 1/3*alpha[5]*diffr[0] + 1/3*alpha
88                 [8]*diffr[1]
89
90                 self.controlPointsHeights[m,(2,0,1)] = b.dot(self.
91             Vcoeffs[i])
92
93                 # punti (2,1,0)
94                 diffz = self.domain.incenters[n] - self.domain.
95             startPoints[i]
96                 b = np.zeros(3)
97                 b[0] = alpha[0] + 1/3*alpha[3]*diffz[0] + 1/3*alpha
98                 [6]*diffz[1]
99                 b[1] = alpha[1] + 1/3*alpha[4]*diffz[0] + 1/3*alpha
100                [7]*diffz[1]
101                b[2] = alpha[2] + 1/3*alpha[5]*diffz[0] + 1/3*alpha
102                [8]*diffz[1]
103
104                self.controlPointsHeights[m,(2,1,0)] = b.dot(self.
105             Vcoeffs[i])
106
107                # punti (1,1,1)
108                b2 = self.domain.lamb[j,i]/(2*self.lift_triangles.
109             TSollSigma[m])
110                b1 = 1-b2
111                self.controlPointsHeights[m,(1,1,1)] = b1*self.
112             controlPointsHeights[m,(2,1,0)] + b2*self.Tcoeffs[m]
113
114                # punti (1,2,0)
115                b2 = self.domain.xi[n,j]/(2*self.lift_triangles.

```

```

TSollSigma[m])
101         b3 = self.domain.xi[n,k]/(2*self.lift_triangles.
TSollSigma[m_prime])
102         b1 = 1 - b2 - b3
103         r = b1*self.controlPointsHeights[m,(2,1,0)] + b2*
self.Tcoeffs[m] + b3*self.Tcoeffs[m_prime]
104         self.controlPointsHeights[m,(1,2,0)] = r
105
106         # punti (0,3,0): altezze relative agli incentri.
107         for i in self.domain.startTr.triangles[n]:
108             PSLList,_ = self.lift_triangles.getPSTriangles(n,i)
109             for m in PSLList:
110                 b1 = self.controlPointsHeights[inHeight
[0][1],(1,2,0)]
111                 b2 = self.controlPointsHeights[inHeight
[1][1],(1,2,0)]
112                 b3 = self.controlPointsHeights[inHeight
[2][1],(1,2,0)]
113                 r = inHeight[0][0]*b1 + inHeight[1][0]*b2 + inHeight
[2][0]*b3
114                 self.controlPointsHeights[m,(0,3,0)] = r
115
116         # Secondo round di calcolo delle altezze.
117         for n in range(len(self.domain.startTr.triangles)):
118             for i in self.domain.startTr.triangles[n]:
119                 # Ricavo la lista PSLList dei triangoli PS nel triangolo
n afferenti al vertice i e dizionario jDict del corrispettivo j
adiacente ad i.
120             PSLList, jList = self.lift_triangles.getPSTriangles(n,i)
121
122             for m_index in range(len(PSLList)):
123                 m = PSLList[m_index]
124                 j = jList[m_index]
125
126                 # controllo se il lato ViRij sta sul bordo del
dominio
127                 (i_ord,j_ord) = (min(i,j),max(i,j))
128                 if self.domain.splitPoints[i_ord,j_ord][1]:
129                     # punti (1,0,2), caso al bordo
130                     b2 = self.domain.lamb[j,i]/(2*self.
lift_triangles.ESollSigma[i,j])
131                     b1 = 1-b2
132                     r = b1*self.controlPointsHeights[m,(2,0,1)] + b2
*self.Ecoeffs[i,j]
133                     self.controlPointsHeights[m,(1,0,2)] = r
134                 else:
135                     # punti (1,0,2), caso non al bordo
136                     n_prime = self.domain.getCommonTriangle(n,(i,j))
137                     m_prime = self.domain.getRijOpposite(n,(i,j))
138                     b1 = self.domain.mu[n,n_prime]
139                     b2 = self.domain.mu[n_prime,n]
140                     r = b1*self.controlPointsHeights[m,(1,1,1)] + b2
*self.controlPointsHeights[m_prime,(1,1,1)]
141                     self.controlPointsHeights[m,(1,0,2)] = r
142
143         # Terzo ed ultimo round di calcolo delle altezze.
144         for n in range(len(self.domain.startTr.triangles)):
145             (PSLList, ijList) = self.domain.getZnOpposites(n)

```

```

146
147     for m_index in range(len(PSList)):
148         (m,m_prime) = PSList[m_index]
149         (i,j) = ijList[m_index]
150         b1 = self.domain.lamb[i,j]
151         b2 = self.domain.lamb[j,i]
152
153         # punti (0,0,3)
154         r = b1*self.controlPointsHeights[m,(1,0,2)] + b2*self.
controlPointsHeights[m_prime,(1,0,2)]
155         self.controlPointsHeights[m,(0,0,3)] = r
156
157         # punti (0,1,2)
158         r = b1*self.controlPointsHeights[m,(1,1,1)] + b2*self.
controlPointsHeights[m_prime,(1,1,1)]
159         self.controlPointsHeights[m,(0,1,2)] = r
160
161         # punti (0,2,1)
162         r = b1*self.controlPointsHeights[m,(1,2,0)] + b2*self.
controlPointsHeights[m_prime,(1,2,0)]
163         self.controlPointsHeights[m,(0,2,1)] = r
164
165     def setSurface(self):
166         # Crea i valori immagine per la superficie, calcolando su ogni
punto della plotTriangulation il valore della superficie di Bezier
triangolare sul triangolo PS al quale il punto appartiene.
167
168         # Raffinamento uniforme della triangolazione Powell_Sabin ai
fini del plottaggio.
169         refiner = UniformTriRefiner(self.domain.PowellSabinTr)
170         self.plotTriangulation, self.TrIndex = refiner.
refine_triangulation(return_tri_index=True, subdiv=4)
171
172         # Salvo i punti della triangolazione iniziale in un solo array.
173         self.plotPoints = np.array(list(zip(self.plotTriangulation.x,
self.plotTriangulation.y)))
174
175         # Inizializzo l'array delle ordinate.
176         plotHeights = []
177
178         for point_index in range(len(self.plotPoints)):
179             point = self.plotPoints[point_index]
180
181             # Ricerco il triangolo PS al quale appartiene il punto.
182             m = self.TrIndex[point_index]
183             triangle = self.domain.PS_triangles[m]
184             tr_vertexes = np.array([self.domain.PS_points[i] for i in
triangle])
185
186             # Ricavo le coordinate baricentriche bc del punto rispetto
al triangolo PS.
187             bc = self.getBarycentricCoords(tr_vertexes,point)
188
189             # Calcolo l'ordinata associata al punto.
190             plotHeights.append(self.triDeCasteljau(m,(0,0,0),bc,3))
191
192             # Salvo array delle ordinate.
193             self.plotHeights = np.array(plotHeights)

```

```

194
195     def getBarycentricCoords(self, triangle: np.array, point: np.array):
196         # Ricava le coordinate baricentriche rispetto a triangle del
197         # punto point.
198         A = np.block([[triangle[:,0]],[triangle[:,1]],[[1,1,1]]])
199         b = np.array([point[0], point[1], 1])
200         return solve(A,b)
201
202     def triDeCasteljau(self, m:int, index:tuple, bc:np.array, deg:int):
203         # Esegue ricosivamente l'algoritmo di DeCasteljau triangolare
204         # per il calcolo dell'ordinata relativa al punto di coordinate
205         # baricentriche bc rispetto al triangolo m.
206         if deg == 0:
207             return self.controlPointsHeights[m,index]
208         else:
209             i0 = (index[0]+1, index[1], index[2])
210             i1 = (index[0], index[1]+1, index[2])
211             i2 = (index[0], index[1], index[2]+1)
212             return bc[0]*self.triDeCasteljau(m,i0,bc,deg-1) + bc[1]*self.
213             .triDeCasteljau(m,i1,bc,deg-1) + bc[2]*self.triDeCasteljau(m,i2,bc,
214             deg-1)
215
216
217     def spline3DPlot(self, cmap='twilight_shifted', curveLevels=True,
218     axis=False, save=False, lvs=60, axIncl=(40,270)):
219         # Visualizza la superficie B-spline.
220         # cmap: la mappa colori da utilizzare sulle linee di livello;
221         # cureLevels: se True visualizza la superficie sottoforma di
222         # curve-livello;
223         # axis: se True mostra gli assi cartesiani;
224         # save: se True salva l'immagine sottoforma di file;
225         # lvs: numero di livelli da mostrare;
226         # axIncl: posizione iniziale degli assi.
227         # Inizializzo i parametri della visualizzazione.
228         fig = plt.figure()
229         ax = fig.gca(projection='3d')
230         ax.view_init(axIncl[0], axIncl[1])           # rotazione assi
231         if not axis:
232             ax.axis('off')
233
234         # Se curveLevels e' True eseguo visualizzazione della superficie
235         # mediante curve di livello.
236         if curveLevels:
237             # Settaggio dei livelli.
238             tol = 6e-3
239             (mn,mx) = (min(self.plotHeights),max(self.plotHeights))
240             if mn < -tol:
241                 lvs = int(lvs/2)
242                 levels = np.block([np.linspace(mn,-tol,lvs),
243                                 np.linspace(tol,mx,lvs)])
244             else:
245                 levels = np.linspace(tol,mx,lvs)
246
247             # Preparazione della visualizzazione.
248             ax.tricontour(self.plotTriangulation, self.plotHeights,
249             levels=levels, cmap=cmap, linewidths=0.5)
250
251         # Caso di visualizzazione classica.

```

```

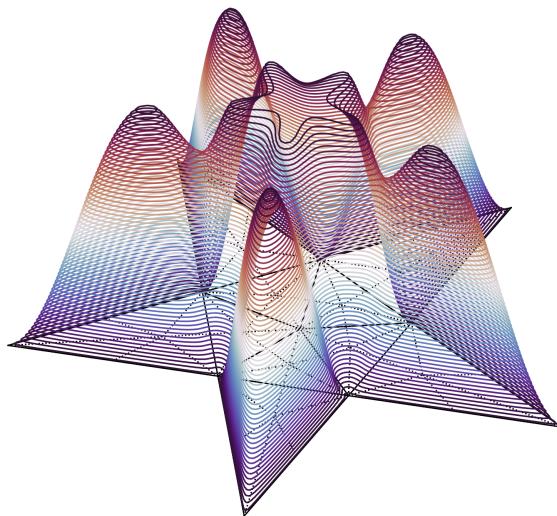
243     else:
244         ax.plot_trisurf(self.plotTriangulation, self.plotHeights)
245
246         # Costruzione della triangolazione sottostante.
247         self.domain.domainPlot(labels=False, legend=False, showExecute=
248 False, color='#000000', colorPS='#000000', linewidths=0.4)
249
250         # Salvo l'immagine, se richiesto.
251         if save:
252             name = input('Nome del file?\n')
253             plt.savefig(name, dpi=600)
254             plt.show()
255
256         # Mostro l'immagine a schermo.
257         plt.show()
258
259     def splineContourPlot(self, cmap='twilight_shifted', save=False,
260 axis=False, showLift=False, lvs=40):
261         # Visualizza curve di livello su grafico 2D della superficie B-
262         # spline.
263         # showLift: se true mostra i triangoli di sollevamento
264         # utilizzati nella costruzione della superficie;
265         # lvs: numero di livelli da mostrare.
266         if not axis:
267             plt.axis('off')
268         plt.axis('equal')
269
270         # Settaggio dei livelli.
271         (mn, mx) = (min(self.plotHeights), max(self.plotHeights))
272         tol = 6e-3
273         if mn < -tol:
274             lvs = int(lvs/2)
275             levels = np.block([np.linspace(mn, -tol, lvs), np.linspace(tol,
276 mx, lvs)])
277         else:
278             levels = np.linspace(tol, mx, lvs)
279
280         # Visualizzazione delle curve di livello.
281         plt.tricontour(self.plotTriangulation, self.plotHeights, levels=
282 levels, cmap=cmap, linewidths=0.4)
283
284         # Visualizzazione del dominio e dei triangoli di sollevamento,
285         # se richiesti.
286         self.domain.domainPlot(labels=False, legend=False, showExecute=
287 False, color='#000000', colorPS='#000000')
288         if showLift:
289             self.lift_triangles.liftPlot(onlyLifts=True, legend=False,
290 labels=False, showExecute=False)
291
292         # Salvo l'immagine, se richiesto.
293         if save:
294             name = input('Nome del file?\n')
295             plt.savefig(name, dpi=600)
296             plt.show()

```

## 2.4.2 Alcuni esempi di utilizzo

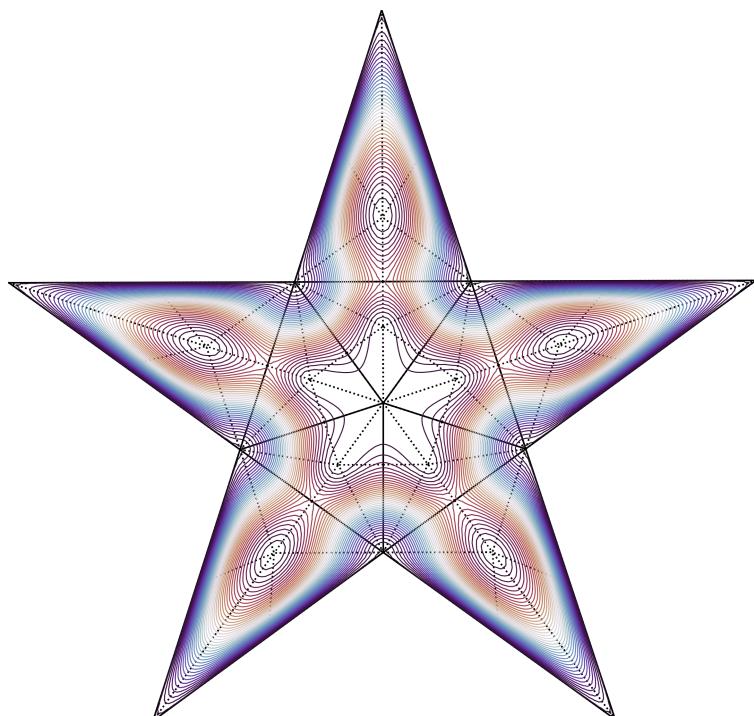
Visualizziamo la superficie B-Spline di Powell-Sabin sulla stella a 5 punte ottenuta settando  $c_{11,r}^v = 1$  (coefficienti di base relativi al vertice centrale) e tutti i coefficienti relativi ai triangoli  $c_m^t = 1$ .

```
1 varray = np.zeros((11,3))
2 earray = np.ones(60)
3 ps = PS_Spline(tr3, varray, earray, {})
4 ps.spline3DPlot()
5
```



Visualizzare le relative curve di livello risulta elementare.

```
6 ps.splineContourPlot()
7
```



## Riferimenti bibliografici

- [1] Lai M.J., Schumaker L.L., *Spline functions on triangulations*. Cambridge University Press (2007).
- [2] Groselj J., Speleers H., *Construction and analysis of cubic Powell-Sabin B-splines*. Computer Aided Geometric Design 57 (2017) 1-22.