UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Corso di Laurea Triennale in Matematica

Tesi di Laurea

Implementazione in Python di un algoritmo per la cubatura numerica su elementi poligonali con un lato curvo

Relatore:
Prof. Alvise Sommariva

Laureando: Giovanni Traversin
Matricola: 2002232

Correlatrice:
Dott.ssa Laura Rinaldi

Anno Accademico 2024/2025
19/09/2025

Indice

Introduzione

In questa tesi consideriamo il problema di implementare in PYTHON alcune formule di quadratura algebriche su elementi poligonali con un lato curvo.

Questo problema è stato discusso da E. Artioli, A. Sommariva, M. Vianello in *Algebraic cubature on polygonal elements with a circular edge*, dove gli autori hanno introdotto un algoritmo che determina una formula algebrica con grado di esattezza prefissato, nodi nel dominio e pesi positivi. Gli ingredienti utilizzati sono formule, determinate via *linear blending*, applicate su domini poligonali e di tipo circolare. Le formule di quadratura sono state poi utilizzate all'interno di un codice per *virtual elements* al fine di risolvere certi problemi alle derivate parziali.

I codici presenti nella sopracitata pubblicazione sono stati scritti in MATLAB. Lo sforzo di questa tesi è di tradurli in PYTHON, facendo attenzione che i risultati ottenuti nei due linguaggi siano numericamente equivalenti, nonostante in alcune parti si utilizzino routines diverse.

Nel primo capitolo, di natura introduttiva, vengono specificati i dettagli principali sviluppati in [1], il *linear blending* e la tecnica di *compressione* di formule di quadratura.

Nel secondo capitolo descriviamo i codici utilizzati, come pure le difficoltà incontrate nella traduzione.

Nel capitolo finale paragoniamo i codici open-source MATLAB con quelli PYTHON, al fine di mostrare che sono numericamente equivalen-

ti, mediante tests sia su nodi e pesi ottenuti, sia sulla qualità di queste formule in ambienti diversi.

La conclusione è che, numericamente, i codici sono effettivamente equivalenti e i tempi di calcolo sono in genere paragonabili. Nel caso compresso, le formule possono essere diverse, ma sempre con il grado di esattezza prefissato.

Capitolo 1

Costruzione della formula di cubatura

In questo capitolo verranno descritti i processi e i teoremi mediante i quali è possibile sviluppare una formula di cubatura algebrica per una funzione definita su un elemento poligonale circolare. Questa parte si basa sul lavoro [1] e ne descriviamo per completezza i dettagli.

1.1 Definizione e costruzione di un elemento poligonale circolare

Poiché il dominio di integrazione è un elemento poligonale circolare, altresì detto *"poligono circolare"*, si procede innanzitutto col descrivere la sua costruzione.

Una tale figura piana si ottiene partendo da un poligono convesso $\mathcal{P}$ qualsiasi avente $P_1, \ldots, P_\ell$ come vertici ordinati in senso antiorario. Il poligono circolare viene costruito sostituendo il segmento $\overline{P_\ell P_1}$ di $\mathcal{P}$ con l'arco di circonferenza $\widehat{P_\ell P_1}$ che interseca il poligono unicamente in P_1 e P_ℓ. Denominiamo in seguito con $\mathcal{C}$ il centro della circonferenza di raggio r di cui il segmento $\widehat{P_\ell P_1}$ è arco.

Un dominio Ω con queste caratteristiche viene solitamente ottenuto in seguito all'unione o alla differenza tra un poligono convesso e un disco ad esso sovrapposto. Nel primo caso, in cui l'arco è esterno al poligono, il poligono circolare risulta convesso, mentre nel secondo caso, in cui l'arco è interno al poligono, concavo.

Siccome per $\ell \in \{2, 3\}$ il dominio di integrazione coincide rispettivamente con un generico segmento o settore circolare (cf. [5]), si considererà senza perdita di generalità $\ell \geq 4$.

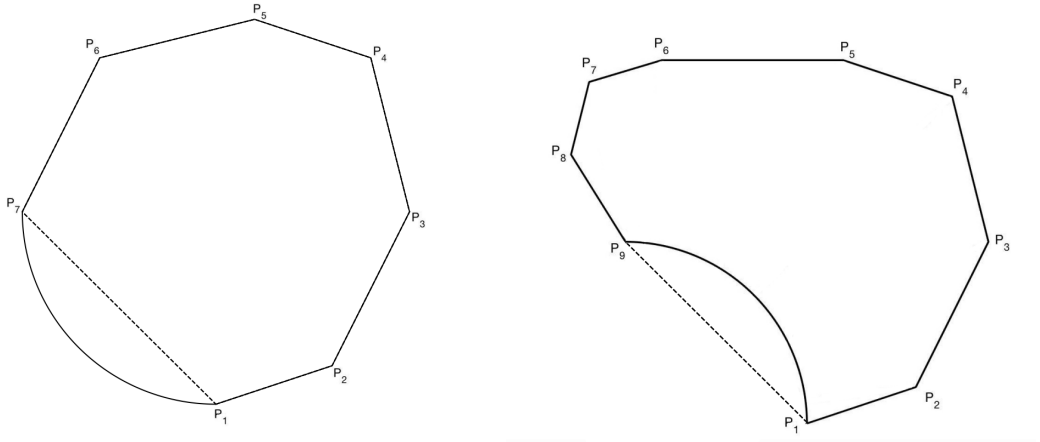

Figura 1.1: Esempi di poligono circolare rispettivamente convesso e concavo

1.2 Suddivisione del dominio

Per semplificare la costruzione della formula di cubatura, vedremo in seguito che è conveniente suddividere il dominio Ω in più sottodomini disgiunti. In questa sezione si descriverà la suddivisione adottata nei programmi in calce, la quale ripercorre a sua volta i codici implementati in [11].

Innanzitutto si nota che, qualora il poligono circolare sia convesso, è immediata la suddivisione di Ω nel poligono $\mathcal{P}$ di ℓ lati e nel relativo segmento circolare associato.

Differentemente, nei casi in cui la parte circolare $\widehat{P_\ell P_1}$ risulta concava rispetto al poligono convesso $\mathcal{P}$, la suddivisione non è altrettanto banale.

Siano (r_k, θ_k) le coordinate polari dei punti P_k per $k \in \{1, \dots, \ell\}$, dove r_k è la distanza tra P_k e $\mathcal{C}$ e $[\theta_1, \theta_\ell] \subset [0, 2\pi)$. Da tale costruzione risulta necessariamente che $r_1 = r_\ell = r$. Senza perdere di generalità, si considererà $\theta_1 \leq \theta_\ell$.

Si formino innanzitutto due insiemi in cui raggruppare tutti i vertici consecutivi di $\mathcal{P}$ che non ricadono all'interno di (θ_1, θ_ℓ). I vertici raccolti a partire da P_1 andranno a formare l'insieme $\{P_1, ..., P_{\ell_1}\}$, mentre quelli consecutivi rispetto a P_ℓ daranno origine all'insieme $\{P_{\ell_2}, ..., P_\ell\}$ con $\ell_1 < \ell_2$. Poiché $\mathcal{P}$ è convesso, questi due insiemi raggruppano tutti i vertici esterni all'angolo (θ_1, θ_ℓ).

Si raccolgano poi all'interno dell'insieme $\{P_{\ell_1+1}, ..., P_{\ell_2-1}\}$ tutti i vertici rimanenti, ovvero quelli che si trovano all'interno dell'area tra θ_1 e θ_ℓ.

A partire dai vertici in $\{P_{\ell_1+1}, ..., P_{\ell_2-1}\}$ è possibile definire una successione di quadrilateri circolari Q_i con $i \in \{\ell_1 + 1, ..., \ell_2 - 2\}$, la cui frontiera è determinata dai segmenti $\overline{P_k^* P_k}$, $\overline{P_k P_{k+1}}$, $\overline{P_{k+1} P_{k+1}^*}$ e dall'arco $\widehat{P_k^* P_k}$. Di questi, va riservata particolare attenzione agli elementi Q_{ℓ_1} e Q_{ℓ_2} (delimitati rispettivamente il primo da $\overline{P_1 P_{\ell_1}}$, $\overline{P_{\ell_1} P_{\ell_1+1}}$, $\overline{P_{\ell_1+1} P_{\ell_1+1}^*}$ e $\widehat{P_{\ell_1+1}^* P_1}$ e il secondo da $\overline{P_{\ell_2-1}^* P_{\ell_2-1}}$, $\overline{P_{\ell_2-1} P_{\ell_2}}$, $\overline{P_{\ell_2} P_\ell}$ e $\widehat{P_\ell P_{\ell_2-1}^*}$) i quali è preferibile siano ridotti a dei settori circolari o ancora meglio, quando possibile, a dei segmenti.

Dalla suddivisione compiuta è immediato che la chiusura dell'insieme $\Omega \setminus \bigcup_{k=\ell_1}^{\ell_2} \mathcal{Q}_k$ coincida al più con due poligoni convessi.

1.2.1 Casi convessi particolari

È interessante notare come la suddivisione di Ω appena descritta possa portare a diverse forme di casi peculiari a seconda del numero di

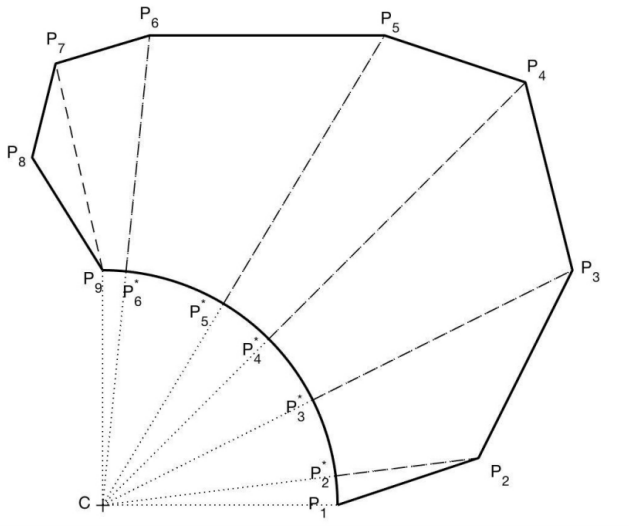

Figura 1.2: Un poligono curvilineo e la suddivisione in sottodomini semplici da trattare per quanto riguarda la cubatura numerica.

elementi all'interno di ciascun insieme di vertici.

Si supponga ad esempio che uno o entrambi tra $\{P_1, \ldots, P_{\ell_1}\}$ e $\{P_{\ell_2}, \ldots, P_\ell\}$ sia formato da un singolo punto o al più da una coppia di punti. In tal caso la chiusura di $\Omega \setminus \bigcup_{k=\ell_1}^{\ell_2} \mathcal{Q}_k$, a seconda che l'insieme con al più due punti sia uno solo o entrambi, risulterebbe vuota o coinciderebbe con un solo poligono convesso. Oppure si supponga che $\{P_{\ell_1+1}, \ldots, P_{\ell_2-1}\}$ risulti vuoto. O ancora che il poligono circolare in questione non sia altro che un segmento circolare, ovvero un poligono circolare in cui i vertici $P_2, \ldots, P_{\ell-1}$ sono collassati sul segmento congiungente P_1 e P_ℓ. Nonostante ciò, comunque la suddivisione operata è valida ed è sempre in grado di restituire al più due poligoni convessi.

È inoltre interessante osservare come l'intera struttura della suddivisione si basi sulla convessità di $\mathcal{P}$. Questo perché, qualora all'interno di un poligono circolare $\mathcal{P}$ fosse non convesso, basterebbe suddividere preliminarmente questo in due poligoni convessi disgiunti per essere poi in grado di applicare ugualmente la formula di divisione del dominio

scelta.

1.3 Cubatura algebrica tramite unione ad arco

Dopo aver definito la suddivisione da applicare al dominio Ω, proseguiamo allo sviluppo di una formula di cubatura sullo stesso. In particolare si richiede che tale formula abbia grado di esattezza algebrica (ADE) pari ad n e sia nella forma

$$\int\int_{\Omega} p(x,y)\ dxdy = \sum_{i=1}^{M_n} \omega_i\ p(x_i,y_i),\ \forall\ p \in \mathbb{P}_n^2\ , \tag{1}$$

dove $\mathbb{P}_n^2$ si riferisce allo spazio dei polinomi algebrici in due variabili di grado al più n.

Grazie alla proprietà additiva degli integrali, per definire una formula con queste proprietà è sufficiente fornirne una equivalente ristretta alle componenti distinte in cui si è diviso Ω.

1.3.1 Costruzione della formula di cubatura sulle singole componenti

Per quanto riguarda la chiusura di $\Omega\ \setminus\ \bigcup_{k=\ell_1}^{\ell_2} \mathcal{Q}_k$, nei casi in cui $\widehat{P_\ell P_1}$ è convesso, la formula di quadratura è immediata. Nei poligoni convessi infatti, dopo aver suddiviso ciascun poligono in $m-2$ triangoli, dove m indica il numero di lati del poligono, una qualsiasi formula di cubatura del simplesso con le coordinate baricentriche ha le caratteristiche richieste. Per citarne alcuni esempi, formule di questo tipo sono le formule di Stroud, la cui cardinalità è $(n+1)^2/4$ e in generale, a pari ADE, non è minima. Per formule di questa natura, si ricorda che un limite dal basso è stato fornito anche da Möller per ADE non troppo elevati. In particolare, tramite le sue formule, è possibile sapere a

priori il numero di nodi richiesti da una formula, come evidenziato in [3] e [4].

Nello specifico, in questa tesi, si è scelto di seguire i codici illustrati ed implementati in [11] anche per scegliere quali formule di cubatura usare in questo caso.

Invece, per quanto riguarda i quadrilateri circolari Q_k, per ottenere una formula di cubatura con ADE $= n$, si adatterà ai poligoni circolari quanto trattato in [5] a proposito della *quadratura subperiodica trigonometrica gaussiana* e della tecnica detta *arc blending*.

Sia Q un quadrilatero circolare qualsiasi. Esso può essere riscritto come la trasformazione bilineare algebrico-trigonometrica di un rettangolo nella forma

$$\mathcal{Q} = \{(x, y) = \sigma(t, \theta) = tP(\theta) + (1 - t)Q(\theta) \qquad (t, \theta) \in [0, 1] \times [\alpha, \beta]\} \tag{2}$$

dove $P(\theta)$ e $Q(\theta)$ sono curve parametriche con parametro di grado 1.

In particolare esse hanno formula

$$P(\theta) = A_1 \cos(\theta) + B_1 \sin(\theta) + C_1, \quad Q(\theta) = A_2 \cos(\theta) + B_2 \sin(\theta) + C_2, \tag{3}$$

con $A = (A_1, A_2)$, $B = (B_1, B_2)$, $\theta \in [\alpha, \beta]$ sotto le condizioni: $\alpha - \beta \leq \pi$ e $A_i, B_i, C_i \in \mathbb{R}^2$ per $i \in \{1, 2\}$.

Questo tipo di scrittura viene chiamata *unione lineare di archi subperiodici* (o *arc blending*), dal momento che gli angoli in essa considerati appartengono sempre a un sottointervallo del periodo. Tramite questo tipo di trasformazioni, è possibile ottenere svariate sezioni di dischi o di domini analoghi legati agli archi di circonferenze o ellissi, come trattato in [5] e [6]

Si reidentifichino dunque i vertici di Q, ordinati consecutivamente tramite coordinate cartesiane, ponendo $V_i = (\xi_i, \eta_i)$ per $i \in \{1, 2, 3, 4\}$. Risulta che $\widehat{V_1V_4}$ coincide con l'arco di circonferenza di centro $C = (x_0, y_0)$ e raggio r di cui α e β sono le coordinate angolari rispettiva-

mente di V_1 e V_4 rispetto a C. Poiché non vi è perdita di generalità, si assumerà $\beta - \alpha < \pi$.

Definiamo inoltre le variabili s e ϕ nel seguente modo:

$$s = \sin\left(\frac{\beta \ - \ \alpha}{2}\right) \qquad\qquad \phi = \frac{\beta \ + \ \alpha}{2}.$$

È immediato osservare dalle definizioni date che per definire un quadrilatero circolare come trasformazione bilineare è sufficiente porre per (2) e (3)

$$A_1 = (r, 0), \qquad B_1 = (0, r), \qquad C_1 = C = (x_0, y_0)$$

$$A_2 = \frac{\sin(\phi)}{2s}(V_3 - V_2), \quad B_2 = \frac{\cos(\phi)}{2s}(V_3 - V_2), \quad C_2 = \frac{1}{2}(V_3 + V_2). \quad (4)$$

In questa notazione, A_1, B_1 e C_1 corrispondono alla rappresentazione polare dell'arco $\widehat{V_1V_4}$, mentre A_2, B_2 e C_2 sono una rappresentazione trigonometrica biettiva del segmento $\overline{V_2V_3}$ lungo l'intervallo $[\alpha, \beta]$ parametrizzata da

$$\frac{V_3 + V_2}{2} + \tau\frac{V_3 + V_2}{2} \ \textit{con} \quad \tau \in [-1, 1],$$

$$\tau = \sin\left(\frac{\theta - \phi}{s}\right) = \frac{\sin(\theta)\cos(\phi) - \cos(\theta)\sin(\phi)}{s} \quad e \quad \theta \in [\alpha, \beta].$$

Poiché all'interno di $\mathcal{Q}$ lo jacobiano $J\sigma$ non varia mai di segno e la stessa funzione σ è iniettiva se e solo se lo sono $P(\theta)$ e $Q(\theta)$ (o equivalentemente se qualsiasi coppia di segmenti $tP(\theta_1) + (1-t)Q(\theta_1)$ e $tP(\theta_1) + (1-t)Q(\theta_2)$ con $\theta_1, \theta_2 \in [\alpha, \beta]$ si interseca solo ad una delle estremità), come osservato in [5], possiamo scrivere $\forall\ p \in \mathbb{P}_n^2$

$$\int\int_{\Omega} p(x, y)\ dxdy = \int\int_{[0,1]\times[\alpha,\beta]} p(\sigma(t, \theta))[\pm \det(J\sigma(t, \theta))]\ dtd\theta \quad (5)$$

a seconda del segno del Jacobiano.

Infatti, dal momento che $(p \circ \sigma) \in \mathbb{P}_n \otimes \mathbb{T}_n$ e $\det(J\sigma) \in \mathbb{P}_1 \otimes \mathbb{T}_2$ si ha che $(p \circ \sigma)[\pm \det(J\sigma)] \in \mathbb{P}_{n+1} \otimes \mathbb{T}_{n+2}$, in cui $\mathbb{P}_n$ e $\mathbb{T}_n$ sono gli spazi dei polinomi algebrici e trigonometrici di grado al più n.

Questa struttura tensoriale è ciò che porta a propendere per l'utilizzo di una formula di quadratura prodotto via quadratura Gaussiana algebrica, purché nei sottointervalli del periodo sia possibile sviluppare una buona formula di quadratura trigonometrica. Tale proprietà viene formalizzata nella seguente proposizione sui *sottoperiodi trigonometrici della formula di quadratura Gaussiana*:

Proposizione 1. *Siano (ξ_j, λ_j), $1 \leq j \leq k+1$, rispettivamente i nodi e i pesi positivi della formula di quadratura algebrica Gaussiana per la funzione peso*

$$w(x) = \frac{2\sin(\omega/2)}{\sqrt{1 - x^2 \sin^2(\omega/2)}} \quad x \in (-1, 1),\ \omega \in (0, \pi]$$

.

Allora dati $0 < \beta - \alpha \leq 2\pi$, per la seguente formula trigonometrica di quadratura Gaussiana vale l'equivalenza

$$\int_\alpha^\beta f(\theta)\ d\theta = \sum_{j=1}^{k+1} \lambda_j f(\theta_j)\ ,\ \forall\ f \in \mathbb{T}_k\ , \tag{6}$$

all'interno della quale impostiamo le costanti ω, ϕ, θ_j ai seguenti valori per $1 \leq j \leq k+1$:

$$\omega = (\beta - \alpha)/2, \quad \phi = (\beta + \alpha)/2, \quad \theta_j = \phi + 2\arcsin(\xi_j \sin(\omega)).$$

La validità della proposizione è dimostrata in [6], mentre alcune delle sue applicazioni sono illustrate in [5]. A partire da questa proposizione è poi possibile svilupparne una seconda, ristretta alle formule prodotto di tipo gaussiano sui quadrilateri circolari.

Proposizione 2. *Sia $\mathcal{Q}$ un quadrilatero circolare nella forma* (2)-(4). *Allora, definiti $\{\theta_j, \lambda_j\}$ rispettivamente come i nodi e i pesi della formula trigonometrica gaussiana con grado di precisione n+2 su $[\alpha, \beta]$ e $\{t^G L_i, w^G L_i\}$ come i nodi e i pesi di Gauss-Legendre della formula con grado di precisione n+1 su [0,1],*

$$\int\int_{\mathcal{Q}} p(x,y)\ dxdy = \sum_{j=1}^{n+2} \sum_{i=1}^{\lceil \frac{n+1}{2} \rceil} W_{ij}\ p(x_{ij}, y_{ij}),\ \forall p \in \mathbb{P}_n^2\ , \tag{7}$$

in cui

$$(x_{ij}, y_{ij}) = \sigma(t_i^{GL}, \theta_j),\ 0 < W_{ij} = |\det(J\sigma(t_i^{GL}, \theta_j)| w_i^{GL} \lambda_j\ . \tag{8}$$

La dimostrazione segue direttamente dalla prima proposizione e da (5), mentre la sua formulazione nel caso generale viene esaminata più approfonditamente in [5]. In questo modo, su ciascuna delle componenti in cui è stato suddiviso il dominio Ω, convesso o meno, si è definita una formula di cubatura del tipo richiesto.

1.3.2 Compressione della formula di cubatura

La formula di quadratura definita nella sezione precedente ha cardinalità $(n+2)\lceil \frac{n+1}{2} \rceil = \frac{n^2}{2} + \mathcal{O}(n)$ e sarà il punto di partenza per definire una seconda formula ad essa equivalente di cardinalità inferiore. Dalle osservazioni sul teorema di Tchakaloff in [10] è noto che una formula di cubatura con ADE $= n$ e cardinalità non inferiore a

$$N = dim(\mathbb{P}_n^2) = \binom{n+2}{2} = \frac{(n+1)(n+2)}{2} \tag{9}$$

può essere sempre compressa in una formula con le stesse proprietà i cui nodi sono un sottoinsieme ricalibrato degli originali e la cui cardinalità non è mai superiore ad N.

Questo studio si concentrerà principalmente sul caso discreto bivariato, ma è importante specificare che questi risultati hanno una validità molto più generale per ogni misura discreta o continua e in qualsiasi dimensione, come mostrato nella trattazione del Teorema di Tchakaloff in [10]. Quanto segue descrive la compressione della formula definita nella sezione precedente, ripercorrendo il processo di costruzione in [8]. Tale compressione è valida per ogni misura discreta e in qualsiasi dimensione.

Si consideri allora una formula di cubatura definita come in (1) su un dominio $\Omega \subset \mathbb{R}$ di cardinalità $M = M_n > N$, in cui n è il grado di precisione. Si definiscano $X = \{(x_i, y_i)\}$ e $\mathrm{W} = \{(w_i)\}$ gli insiemi contenenti rispettivamente gli M_n nodi e pesi, sia $p_1, \ldots, p_n$ una base per $\mathbb{P}_n^2$ e si calcoli a partire da tali elementi la corrispondente matrice di Vandermonde

$$V = V_n(x) = [v_{ij}] = [p_j(x_i, y_i)] \in \mathbb{R}^{M \times N}. \tag{10}$$

A partire dalla trasposta di questa matrice possiamo impostare il seguente sistema di uguaglianze dei momenti sotto vincoli di non negatività

$$V^t u = b = V^t w \ , \quad u \geq 0. \tag{11}$$

Per questo sistema di uguaglianze, la ricerca di una formula di cubatura polinomialmente esatta con X come supporto è equivalente alla ricerca di una soluzione sparsa con al più M valori non negativi.

Come si può vedere in [2], grazie al teorema di Caratheodory per le combinazioni coniche in dimensioni finite applicato alle colonne di V^t, una tale soluzione sparsa non solo è garantita, ma in generale non è neppure unica e il numero di valori non negativi non è mai superiore ad N.

Per un problema come (11), la letteratura matematica ha a sua disposizione svariati studi, quali [8], [10] e [12], che descrivono più di un metodo attendibile per la sua risoluzione.

Una possibile strada consiste nel considerare il sistema di equazioni come un programma lineare

$$\begin{cases} \min c^t u \\ V^t u = b \ , \quad u \geq 0 \end{cases}$$

e risolverlo tramite algoritmi di ottimizzazione quali il metodo del simplesso e delle due fasi.

Nel costruire questo programma, u rappresenta il vettore incognito dei pesi, $V^t u = b$ sono i vincoli di esattezza forniti dalla matrice di Vandermonde (10) e c^t è il vettore dei costi della funzione obiettivo generato artificialmente in modo da essere linearmente indipendente dai vincoli.

Sotto questa luce, la validità delle osservazioni fatte precedentemente risulta immediata. I vincoli identificano come regione delle soluzioni ammissibili un politopo di $\mathbb{R}^M$ sempre non vuoto, dal momento che $b = V^t w \geq 0$, e dunque il sistema ha sempre almeno una soluzione ammissibile. Tuttavia, dal momento che il politopo è limitato, segue anche che esiste sempre almeno una soluzione ottima con almeno $M-N$ componenti uguali a 0 che l'algoritmo del simplesso è sicuramente in grado di trovare. Infine, il vettore c dei coefficienti della funzione obiettivo è costruito in modo da essere linearmente indipendente dalle righe di V^t, dunque è possibile sapere a priori che tale soluzione sarà non costante e non banale.

Eventualmente, qualora non ci si volesse discostare troppo dal teorema di Tchakaloff, è anche possibile considerare il sistema come un problema di ottimizzazione quadratica. In questo caso, esso coincide con un programma lineare di ottimizzazione convessa per i minimi quadrati non negativi (NNLS - NonNegative Least Square) nella forma

$$u^* : ||V^t u^* - b||_2 = \min_{u \geq 0} ||V^t u - b||_2 \tag{12}$$

in cui u^* può essere calcolato tramite il *Metodo di ottimizzazione di Lawson-Hanson*, trattato in [7], il quale è particolarmente adatto per la ricerca di soluzioni sparse.

Si consideri adesso una soluzione sparsa u per (11) e la successione di indici i_k, con $k \in \{1, \dots, m \leq N < M\}$, tali che $u_{i_k} \neq 0$. È possibile riscrivere la formula (1) nella forma compressa ricercata

$$\int\int_{\Omega} p(x,y)\ dxdy = \sum_{i=1}^{M}\ \omega_i\ p(x_{i_k}, y_{i_k}) = \sum_{k=1}^{m} u_{i_k} p(x_{i_k}, y_{i_k}),\ \forall\ p \in \mathbb{P}_n^2. \tag{13}$$

In virtù dei risultati in [8], possiamo dire di aver operato una valida compressione di Caratheodory-Tchakaloff alla formula di cubatura per poligoni circolari iniziale definita in (7) e (8).

Nel prossimo capitolo verranno mostrate tramite tabelle numeriche le differenze nelle tempistiche e nei risultati degli integrali a seconda che si usi la formula compressa o meno e a seconda che si usi il programma MATLAB o PYTHON.

Capitolo 2

Traduzione da MATLAB a PYTHON

In questo capitolo si descriveranno la traduzione e il funzionamento degli algoritmi già implementati in MATLAB in [11] sulla base dei teoremi visti nel Capitolo 1. A ciò seguiranno una serie di esempi e risultati numerici per mettere a confronto l'implementazione in PYTHON con quella in MATLAB.

Tutti i codici PYTHON essenziali a questa tesi sono stati inseriti in calce.

2.1 Gli algoritmi principali

Durante la traduzione, si è cercato di sviluppare gli algoritmi in modo da poterli utilizzare anche singolarmente in ambienti esterni alla funzione finale `polygcirc`. Per tale ragione, e poiché spesso i programmi implementati si appoggiano alla libreria `numpy`, gli elementi che è necessario definire come vettori vengono resi tali ogni volta all'interno del programma. In questo modo i valori possano di volta in volta venire dati in input alle varie funzioni sia in forma di liste che in forma di vettori.

2.1.1 Polygcirc

Trattasi del programma principale, al quale sono finalizzate tutte le altre funzioni implementate in questa tesi.

Al suo interno si ripercorrono fedelmente tutti i vari passaggi descritti nel Capitolo 1 per arrivare ad una formula di cubatura compressa per i poligoni circolari con ADE $= n$.

Dati in input la lista dei vertici della figura ordinati in senso antiorario, gli estremi, il centro ed il raggio del segmento circolare e la convessità o concavità dell'arco, la funzione produce i nodi e i pesi della formula di cubatura (1) tramite le funzioni `circtrap` e `polygauss`. A tal proposito, come visto nella sezione 1.3.1, il programma opera in maniere differenti a seconda che il poligono circolare sia concavo o convesso.

Infine, come nella sezione 1.3.2, i nodi e i pesi trovati vengono compressi tramite la funzione `comprexcub`.

Il programma restituisce poi in output i nodi e i pesi sia originali che compressi, sotto forma di matrici di tre colonne in cui la prima e la seconda colonna contengono ascisse e ordinate dei nodi, mentre la terza i pesi ad essi associati

Poiché però in MATLAB, a seconda delle variabili richieste in output, i programmi possono procedere o meno al calcolo effettivo dei parametri definiti all'interno dei codici, mentre invece in PYTHON vengono sempre calcolati tutti, al programma in PYTHON sono stati aggiunti una condizione `if` finale ed una `flag` opzionale in input, per consentire una miglior valutazione dei tempi di calcolo tra i due linguaggi. In particolare, la `flag` è collegata direttamente ed esclusivamente all'ultima condizione `if`, e, qualora non venga specificata, è preimpostata per assumere il valore `True`. Tali aggiunte consentono di non calcolare necessariamente i nodi e i pesi compressi per la formula di cubatura e di distinguere tra i tempi necessari al calcolo dei nodi e dei pesi

compressi e i tempi necessari per calcolare solo quelli gaussiani. Tali modifiche sono state fondamentali per poter raccogliere i dati in 3.6, 3.2.1 e 3.2.2.

2.1.2 Polygauss

Questa funzione permette il calcolo di nodi e pesi di tipo gaussiano su un poligono qualsiasi. All'interno di `polygcirc`, essa produce nodi e pesi per tutte le componenti poligonali prive di lati curvi.

Come per la sua controparte MATLAB, questa funzione richiede che i lati del poligono forniti in input siano dati come lista di vertici consecutivi ordinati in senso antiorario, di cui il primo coincide con l'ultimo.

Inoltre, se in input non viene dato anche un rettangolo contenente il poligono circolare, è la funzione stessa a produrre il minimo rettangolo contenente l'elemento, per poi procedere al calcolo di nodi e pesi bidimensionali sul dominio tramite operazioni sui nodi monodimensionali ottenuti dalla funzione `cubature_rules_1D`. A seconda dell'input dato a tale funzione, questi valori possono venire calcolati tramite le due regole di Fejer, la regola di Clenshaw-Curtis o l'algoritmo di Gauss-Legendre.

2.1.3 Circtrap e gqellblend

La formula `circtrap` serve a calcolare nodi e pesi per l'integrazione numerica su tutte quelle componenti del dominio che presentano lati curvi. Queste tendenzialmente sono quadrilateri circolari, anche detti trapezi circolari, da cui il nome `circtrap`. Essa tuttavia non calcola direttamente i pesi, ma piuttosto prende in input i vertici della figura e, in base al centro ed al raggio che generano il lato curvo, trasforma i vertici del quadrilatero circolare in coordinate polari per poi riordinarli in modo da trasformare il dominio di integrazione in un

dominio standard, per poi ricavarne effettivamente nodi e pesi tramite `gqellblend`.

Questa seconda funzione calcola i nodi e i pesi per una formula di cubatura gaussiana di tipo prodotto per polinomi bivariati di grado totale $\leq n$ definita su una regione piana ottenuta tramite combinazione convessa di due archi trigonometrici con equazioni parametriche, concordemente a quanto detto nel Capitolo 1 riguardo a (5). Essa sfrutta i codici `gauss` e `r_jacobi` per ricavare la formula gaussiana sull'intervallo $[0, 1]$ e `trigauss` per calcolare la formula trigonometrica gaussiana sull'arco. In particolare è possibile identificare `trigauss` con la formula trigonometrica di quadratura gaussiana e `gqellblend` con la formula di cubatura tramite unioni ad arco, entrambe discusse nella sezione 1.3.1 attenendosi all'implementazione dei codici in [14]. Dalla combinazione dei risultati ottenuti, `gqellblend` restituisce la variabile `xyw`, una matrice di tre colonne ciascuna rispettivamente per ascisse, ordiate e pesi dei nodi per la formula di cubatura.

2.1.4 Comprexcub

In questa funzione, i programmi per la costruzione della matrice di Chebyshev-Vandermonde `chebvand` e `chebpolys` fanno sì che una formula di cubatura, data in input sotto forma di nodi e relativi pesi, venga compressa come indicato da Tchakaloff sulla base dei risultati ottenuti in (13).

In particolare questa funzione ha come obiettivo la risoluzione del problema (11) discusso nel Capitolo 1, il quale ammetteva più approcci differenti per essere risolto. Per rispettare tale libertà di approccio, al programma può essere dato come input il parametro `pos` con cui selezionare l'algoritmo risolutivo.

In particolare, se

- non viene dato nessun valore a `pos`, il programma risolve automaticamente (11) come un programma lineare di ottimizzazione convessa per i minimi quadrati non negativi tramite la funzione `optimize.nnls` della libreria `Scipy`,

- *pos* = 1 il programma risolve il sistema tramite l'algoritmo di Lawson-Hanson accelerato dalla strategia di selezione a blocco (Deviation Maximization). In particolare l'implementazione dell'algoritmo di Lawson-Hanson utilizzato in questa traduzione, sfrutta il codice implementato in: `https://github.com/laura-rinaldi/Cheap_Splinegauss`.

In MATLAB era anche contemplata una terza via risolutiva, tramite risoluzione sparsa di un sistema lineare standard con matrice non quadrata. Purtroppo in PYTHON, a parte `optimize.nnls`, i codici per la risoluzione dei sistemi lineari tendono ad evitare approcci che restituiscano soluzioni sparse. Per tale ragione, la libertà di scelta del codice sorgente risulta non completamente replicabile in PYTHON.

Coerentemente con quanto detto nel primo capitolo, in base a [8], [10] e [12], questi algoritmi portano tutti a risultati equivalenti.

2.1.5 Demo_polygcirc

Si tratta di una funzione grafica e demo che, se da un lato permette di visualizzare ciò che fa la formula di cubatura, ovvero il dominio e i nodi di cubatura, dall'altro permette anche di visualizzare i risultati ottenuti tramite `polygcirc`.

Tale funzione ha alcuni poligoni circolari definiti come casi al suo interno e sfrutta la libreria `matplotlib` per restituire una figura rappresentante il poligono circolare relativo al caso scelto, suddiviso come nella sezione 1.2, con in evidenza i nodi compressi di grado n (impostati con n=4). Eventualmente, qualora ve ne sia necessità, è possi-

bile aggiungere ulteriori poligoni circolari all'elenco dei casi forniti dal programma nella sezione `match`.

Oltre alle figure, il programma ritorna anche una tabella di valori che saranno utilizzati ampiamente nel Capitolo 3.
Questi valori comprendono:

- il numero dell'esperimento;
- il grado di precisione algebrica ADE;
- il tipo di compressione utilizzata in `comprexcub`;
- in numero di nodi e pesi trovati dalla formula di cubatura non compressa;
- il numero di nodi e pesi a seguito della compressione;
- il numero di pesi negativi, che dovrebbe essere sempre 0;
- il tempo di calcolo dei nodi non compressi;
- il tempo di calcolo dei nodi compressi;
- l'errore di ricostruzione dei momenti.

Per concludere, all'interno del programma sono state apportate alcune modifiche rispetto a MATLAB, al fine di soddisfare alcune necessità di questa tesi, ovvero

- la funzione adesso ha dei valori facoltativi in input che permettono di selezionare a piacimento anche il grado di precisione della formula e il tipo di compressione;
- la tabella dei valori ora presenta anche l'errore di ricostruzione dei momenti calcolato da `comprexcub`;

- alcune righe di codice non presenti nel codice sorgente ma utili per la determinazione di alcuni risultati in questa tesi sono state lasciate all'interno del programma sotto forme di commenti. In particolare, quelle sotto la sezione *Nodes to copy in Matlab* permettono di stampare nodi e pesi compressi e non in un formato che MATLAB è in grado di leggere come input, mentre quelle contrassegnate come *Plots with sorted weights* permettono di riprodurre dei grafici dei pesi ordinati come quelli che saranno mostrati in seguito in (3.3).

2.2 Le differenze tra i linguaggi

Le difficoltà incontrate durante la traduzione in PYTHON degli algoritmi implementati in questa tesi sono state principalmente di tipo tecnico e notazionale e non legate alla struttura degli algoritmi.

Per un miglior parallelismo tra i linguaggi, si è deciso di mantenere il più possibile la sequenzialità dei codici MATLAB, modificando la struttura del programma solo nei punti dove era strettamente necessario a causa delle differenze di impostazione dei linguaggi. I casi più evidenti in cui si possono notare queste modifiche sono le funzioni

- `chebvand`,
- `r_jacobi`,
- `polygauss`,

nelle quali non vi sono condizioni interne al programma per la verifica degli input, ma sono gli stessi input che vengono impostati a priori, concordemente alle condizioni in MATLAB, qualora non venissero definiti.

Un ulteriore ostacolo intrinseco nella traduzione tra questi due linguaggi è la diversa indicizzazione delle collezioni di elementi. Infatti,

in Python, liste, tuple, dizionari ed elementi di questo genere identificano il proprio primo elemento con l'indice 0, mentre in Matlab partono dall'indice 1. Riguardo a ciò, per la maggior parte dei cicli `for` incontrati, si è cercato di fare in modo che il ciclo vagliasse fin da subito valori diminuiti di 1, al fine di mantenere il codice interno al ciclo in Python il più simile possibile a quello Matlab. Si rinuncia a tale approccio solo in alcuni casi specifici, come ad esempio in `chebvand`, in cui i valori vagliati dal ciclo `for` hanno un'effettiva validità numerica all'interno dell'algoritmo.

2.3 Limiti per una IA

Dopo una prima traduzione manuale dei codici, sono state fatte anche delle prove con alcune IA e programmi di traduzione. Le IA utilizzate principalmente sono state ChatGPT e Gemini.

Quando il codice Matlab veniva copiato su Google Colab e cominciava ad essere tradotto manualmente, la funzione forniva suggerimenti talvolta affidabili quando si traducevano linee di codice che presentavano una qualche affinità alle precedenti. Tuttavia spesso non apportava le modifiche corrette agli indicatori di posizione per un elemento di una lista, scrivendo `lista(posizione+1)` anziché `lista[posizione]`.

Per quanto riguarda ChatGPT invece, sono stati rilevati alcuni risultati interessanti. Per alcune funzioni più semplici, quali ad esempio

- `chebpolys`,
- `points2distances`,
- `gqellblend`,

le funzioni fornite dall'IA riportavano alcune alterazioni strutturali rispetto al codice originario, ma risultavano nel complesso funzionanti e in alcune occasioni più rapide delle funzioni tradotte.

Tuttavia, quando l'IA ha dovuto tradurre funzioni più intricate, quali `chebvand` o `polygauss`, vi erano delle imperfezioni che rendevano inutilizzabile il codice. Innanzitutto, le funzioni definite precedentemente, quali

- `gauss`,
- `r_jacobi`,
- `cubature_rules_1D`,

venivano scambiate per dei parametri e aggiunte alle variabili richieste in input, nonostante poi nel codice venissero utilizzate come funzioni.

Dove possibile, l'IA ha sempre tentato di unire i cicli `for` per alleggerire il peso computazionale della funzione. Questo in funzioni come `chebvand` ha compromesso le dimensioni della matrice di Chebyshev-Vandermonde costruita, risultando in un output scorretto.

Infine, ChatGPT ha avuto difficoltà ad approcciarsi a quei vettori e a quelle matrici che in Matlab venivano generate progressivamente. Questo avviene perché in Matlab viè la possibilità di generare progressivamente una variabile senza averla mai definita prima, mentre in Python va sempre definita una variabile a priori per poi modificarla progressivamente. In questi casi, l'IA traduceva il programma senza adattarsi alle esigenze del nuovo linguaggio, andando a modificare una variabile mai definita prima.

Nonostante apparentemente questi siano errori banali e di semplice risoluzione, qualora si facesse tradurre all'IA un codice senza avere idea della nuova forma che dovrebbe assumere, si avrebbero non poche difficoltà nella ricerca ed identificazione dell'errore nel programma.

Il programma ideale per la traduzione di un programma da Matlab a Python molto probabilmente si avvicinerebbe, almeno concettualmente, al traduttore online Mat2py, il quale ha implementati al

suo interno algoritmi che traducono in maniera pedante ma precisa i codici da un linguaggio all'altro.

Anche questo programma, tuttavia, non è esente da difetti estremamente invalidanti, dal momento che, per quanto riguarda le librerie, Mat2py è assolutamente autoreferenziale, ma i codici a sua disposizione non sono aggiornati in maniera attiva ed hanno forti lacune. Per tale ragione i codici da lui tradotti sono spesso privi delle funzioni numeriche adeguate ai calcoli che gli è richiesto compiere, funzioni numeriche che sono tuttavia ampiamente fornite da altre librerie quali `numpy` o `scipy`.

Capitolo 3

Esperimenti numerici

In questo capitolo si applicheranno le formule MATLAB e PYTHON su elementi poligonali con un lato curvo.

Per evidenziare l'equivalenza della traduzione, per ogni dominio di integrazione si effettuerà preliminarmente un'analisi delle cardinalità delle formule e degli errori su nodi e pesi.

Nelle relative tabelle saranno visualizzati, in norma infinito, i massimi errori per nodi e pesi, compressi e non, per ciascun grado di precisione. In altri termini, se

- $x^{(m)}$ e $w^{(m)}$ sono i nodi della formula originale in MATLAB,
- $x^{(p)}$ e $w^{(p)}$ sono i nodi della formula originale in PYTHON,
- $x_c^{(m)}$ e $w_c^{(m)}$ sono i nodi della formula compressa in MATLAB,
- $x_c^{(p)}$ e $w_c^{(p)}$ sono i nodi della formula compressa in PYTHON,

verranno valutati

$$\begin{aligned} E_{nodes} &= \|x^{(m)} - x^{(p)}\|_\infty \\ E_{weights} &= \|w^{(m)} - w^{(p)}\|_\infty \\ E_{nodes}^{(c)} &= \|x_c^{(m)} - x_c^{(p)}\|_\infty \\ E_{weights}^{(c)} &= \|w_c^{(m)} - w_c^{(p)}\|_\infty \end{aligned}$$

In particolare, sia all'interno del programma MATLAB che all'interno del programma PYTHON, la funzione `comprexcub` risolve il programma lineare (11) con la funzione `scipy.optimize.nnls`.

In ognuno degli esempi che seguiranno saranno considerate le approssimazioni di alcuni integrali tramite formule aventi gradi di precisione ADE pari a $2, 4, 6, 8$ e 10. Questi valori sono stati scelti avendo in mente le applicazioni ai *virtual elements*, in cui generalmente i gradi di precisione non sono troppo alti.

Per ognuno dei gradi di esattezza ADE si raccoglieranno i seguenti dati:

- il valore corretto *INT* dell'integrale, definito applicando la formula di cubatura (7) con grado di precisione 60;
- i valori numerici *Int_M*, *Int_P* dell'integrale, ottenuti tramite la formula di cubatura (7), rispettivamente in MATLAB e in PYTHON;
- i valori numerici *Compr_Int_M*, *Compr_Int_P* dell'integrale, ottenuti tramite la formula di cubatura compressa (13), rispettivamente in MATLAB e in PYTHON;
- le cardinalità delle formule di cubatura *Card_M*, *Card_P*, *Compr_Card_M*, *Compr_Card_P*, ottenute rispettivamente in MATLAB e in PYTHON, relativamente alla formula originale e a quella compressa;
- gli errori relativi *ER_M*, *ERC_M*, *ER_P*, *ERC_P*, compiuti dai codici in MATLAB e in PYTHON, relativamente dalla formula originale e da quella compressa;
- i tempi di calcolo *Time_M*, *Time_P* necessari per l'esecuzione di `polygcirc`, rispettivamente in MATLAB e in PYTHON.

3.1 Esperimenti su un elemento poligonale con un lato curvo e convesso

Quale primo dominio, si consideri il poligono circolare convesso avente come vertici i punti (0.25, 0), (0.4, 0.05), (0.5, 0.25), (0.45, 0.45), (0.3, 0.5), (0.1, 0.45), (0, 0.25) e come lato curvo l'arco di circonferenza di centro (0.25, 0.25) e raggio 0.25 avente come estremi i punti (0.25, 0) e (0, 0.25).

Il dominio è rappresentato dalle figure 3.1, assieme rispettivamente all'insieme di nodi delle formule originali e compresse per $ADE = 10$. In tali figure si evidenzia, come noto, che i nodi sono interni al dominio.

Di seguito, nella figura 3.2, sono riportati i valori dei pesi ordinati in senso crescente, tanto per le formule originali quanto per quelle compresse. Si vede immediatamente la differeza di cardinalità e che i pesi, per entrambe le formule, sono positivi.

Nella tabella 3.1 sono stati indicati, per ciascuno dei gradi stabiliti, le cardinalità di tali formule.

ADE	Card_M	Card_P	Compr_Card_M	Compr_Card_P
2	52	52	6	6
4	161	161	15	15
6	330	330	28	28
8	559	559	45	45
10	848	848	66	66

Tabella 3.1: Cardinalità delle formule di cubatura originali e compresse tanto in MATLAB quanto in PYTHON.

Dalla tabella 3.2, invece, si vede che le formule originali sono numericamente uguali in termini di nodi e pesi. Per quanto riguarda le loro versioni compresse, nonostante vengano chiamate routines diverse di NNLS, anche queste risultano numericamente equivalenti.

ADE	E_{nodes}	$E_{weights}$	$E^{(c)}_{nodes}$	$E^{(c)}_{weights}$
2	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$4.9 \cdot 10^{-16}$	$4.2 \cdot 10^{-16}$
4	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$5.3 \cdot 10^{-16}$	$4.6 \cdot 10^{-16}$
6	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$5.6 \cdot 10^{-16}$	$5.2 \cdot 10^{-16}$
8	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$5.6 \cdot 10^{-16}$	$5.2 \cdot 10^{-16}$
10	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$5.3 \cdot 10^{-16}$	$7.6 \cdot 10^{-16}$

Tabella 3.2: Scarti per nodi e pesi, sia compressi che non, tra MATLAB e PYTHON sotto forma di norma infinito per il dominio di integrazione convesso.

Di seguito abbiamo considerato l'approssimazione dell'integrale di $f_1(x, y) = e^{(x-y)}$ sul'elemento in analisi, il cui risultato di riferimento

$$INT = 0.18924464927762087$$

è stato ottenuto mediante una formula non compressa con grado di precisione $ADE = 60$.

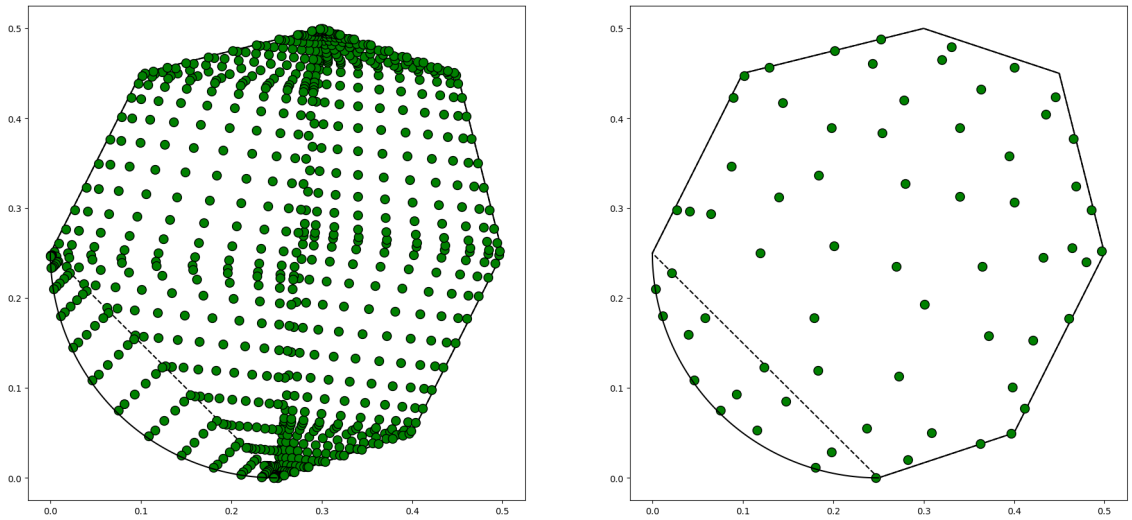

Figura 3.1: Nodi per le formule di cubatura con $ADE = 10$, relativamente alla formula originale (a sinistra) e alla sua versione compressa (a destra).

Nelle tabelle 3.3 e 3.4 vengono riportati i valori ottenuti utilizzando formule di quadratura originali e compresse, tanto in MATLAB quanto

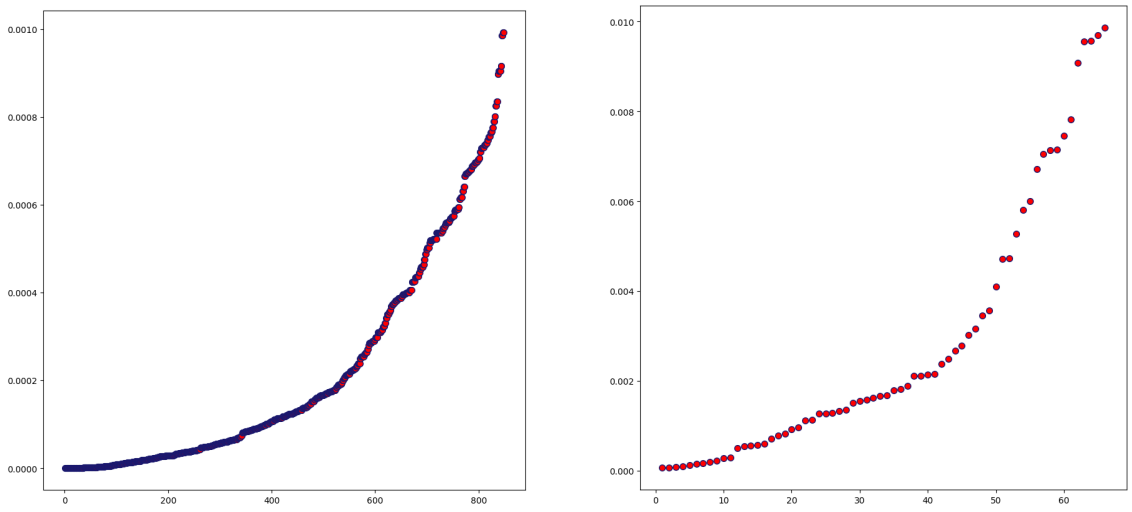

Figura 3.2: Pesi ordinati per valore delle formule di cubatura con $ADE = 10$, relativamente alla formula originale (a sinistra) e alla sua versione compressa (a destra).

in PYTHON, mentre in 3.5 si trovano gli errori relativi compiuti. Come atteso, i risultati raggiunti sono estremamente simili.

ADE	Int_M	Int_P
2	0.1892445479685858	0.18924454796858572
4	0.1892446492775920	0.18924464927759205
6	0.1892446492776211	0.18924464927762100
8	0.1892446492776211	0.18924464927762114
10	0.1892446492776216	0.18924464927762155

Tabella 3.3: Calcolo dell'integrale proposto con le formula di cubatura originali, tanto in MATLAB quanto in PYTHON.

Si raccolgono infine nelle ultime due tabelle in 3.6 i tempi di calcolo medi di tutte le formule utilizzate nei due linguaggi di programmazione. Dal momento che i tempi di valutazione per l'integrale sono trascurabili, le tempistiche dei programmi sono essenzialmente quelle del calcolo per la formula di quadratura. In particolare, ognuno di que-

ADE	Compr_Int_M	Compr_Int_P
2	0.1892014422189217	0.18920144221892168
4	0.1892447779632670	0.18924477796326708
6	0.1892446492824988	0.18924464928249871
8	0.1892446492776253	0.18924464927762522
10	0.1892446492776212	0.18924464927762111

Tabella 3.4: Calcolo dell'integrale proposto con le formula di cubatura compressa, tanto in MATLAB quanto in PYTHON.

ADE	ER_M	ER_P	ERC_M	ERC_P
2	$5.4 \cdot 10^{-7}$	$5.4 \cdot 10^{-7}$	$2.3 \cdot 10^{-4}$	$2.3 \cdot 10^{-5}$
4	$1.5 \cdot 10^{-13}$	$1.5 \cdot 10^{-13}$	$6.8 \cdot 10^{-7}$	$6.8 \cdot 10^{-7}$
6	$1.3 \cdot 10^{-15}$	$7.3 \cdot 10^{-16}$	$2.6 \cdot 10^{-11}$	$2.6 \cdot 10^{-11}$
8	$1.2 \cdot 10^{-15}$	$1.5 \cdot 10^{-15}$	$2.3 \cdot 10^{-14}$	$2.3 \cdot 10^{-14}$
10	$4.0 \cdot 10^{-15}$	$3.7 \cdot 10^{-15}$	$1.6 \cdot 10^{-15}$	$1.3 \cdot 10^{-15}$

Tabella 3.5: Errori relativi nel calcolo dell'integrale con le formula di cubatura compressa, tanto in MATLAB quanto in PYTHON.

sti risultati è ottenuto dalla media dei tempi di 20 lanci del programma `polygcirc` per ciascun grado di precisione.

ADE	Rule_Time_M	Rule_Time_P
2	$7.458 \cdot 10^{-4}$	$3.220 \cdot 10^{-3}$
4	$8.884 \cdot 10^{-4}$	$2.142 \cdot 10^{-3}$
6	$5.278 \cdot 10^{-4}$	$4.142 \cdot 10^{-3}$
8	$5.586 \cdot 10^{-4}$	$1.210 \cdot 10^{-2}$
10	$6.396 \cdot 10^{-4}$	$6.255 \cdot 10^{-3}$

ADE	C_Rule_Int_M	C_Rule_Int_P
2	$4.026 \cdot 10^{-3}$	$5.388 \cdot 10^{-3}$
4	$3.035 \cdot 10^{-3}$	$9.168 \cdot 10^{-3}$
6	$2.934 \cdot 10^{-3}$	$1.400 \cdot 10^{-2}$
8	$4.258 \cdot 10^{-3}$	$4.886 \cdot 10^{-2}$
10	$1.406 \cdot 10^{-2}$	$2.776 \cdot 10^{-2}$

Tabella 3.6: Tempi di calcolo con le formule di cubatura originali e compresse, tanto in MATLAB quanto in PYTHON.

Si nota immediatamente che il programma nel suo linguaggio d'origine risulta più veloce rispetto al codice PYTHON.

Come secondo esperimento, calcoliamo numericamente l'integrale di una funzione meno regolare, come

$$f_2(x, y) = ((x - 2/5)^2 + (y - 1/3)^2)^{\frac{5}{2}}$$

sullo stesso dominio definito per il primo esempio. Nel caso della funzione in esame, l'irregolarità è data dal punto singolare $(x_0, y_0) \equiv (2/5, 1/3)$ che è interno al poligono curvilineo convesso Ω. Poiché il numero di nodi e pesi ottenuti dipende esclusivamente dalla figura in esame, si avrà di conseguenza che le cardinalità degli insiemi contenenti i nodi risulteranno invariate.

Utilizzando il codice MATLAB con ADE=60 otteniamo il valore di riferimento

$$INT = 3.4852942237109879 \cdot 10^{-4}.$$

Poiché, come detto in precedenza, i tempi di calcolo degli integrali sono trascurabili e le tempistiche complessive sono approssimate a quelle del calcolo per la formula di quadratura, per i tempi di questo integrale si farà riferimento alla tabella 3.6.

ADE	Int_M	Int_P
2	$3.4693882669925368 \cdot 10^{-4}$	$3.4693882669925357 \cdot 10^{-4}$
4	$3.4852956661042306 \cdot 10^{-4}$	$3.4852956661042311 \cdot 10^{-4}$
6	$3.4852942373675845 \cdot 10^{-4}$	$3.4852942373675834 \cdot 10^{-4}$
8	$3.4852942190582232 \cdot 10^{-4}$	$3.4852942190582233 \cdot 10^{-4}$
10	$3.4852942217568768 \cdot 10^{-4}$	$3.4852942217568735 \cdot 10^{-4}$

Tabella 3.7: Calcolo dell'integrale con le formula di cubatura originali, tanto in MATLAB quanto in PYTHON.

Nuovamente, gli integrali risultano estremamente simili, a riprova dell'equivalenza dei codici nonostante la diversità dei linguaggi.

ADE	Compr_Int_M	Compr_Int_P
2	$5.2625000443101136 \cdot 10^{-4}$	$5.2625000443101136 \cdot 10^{-4}$
4	$3.4669365134521070 \cdot 10^{-4}$	$3.4669365134521037 \cdot 10^{-4}$
6	$3.4847637578881458 \cdot 10^{-4}$	$3.4847637578881398 \cdot 10^{-4}$
8	$3.4853234110670140 \cdot 10^{-4}$	$3.4853234110670145 \cdot 10^{-4}$
10	$3.4852899344312332 \cdot 10^{-4}$	$3.4852899344312325 \cdot 10^{-4}$

Tabella 3.8: Calcolo dell'integrale con le formula di cubatura compressa, tanto in MATLAB quanto in PYTHON.

ADE	ER_M	ER_P	ERC_M	ERC_P
2	$4.6 \cdot 10^{-3}$	$4.6 \cdot 10^{-3}$	$5.1 \cdot 10^{-1}$	$5.1 \cdot 10^{-1}$
4	$4.1 \cdot 10^{-7}$	$4.1 \cdot 10^{-7}$	$5.3 \cdot 10^{-3}$	$5.3 \cdot 10^{-3}$
6	$3.9 \cdot 10^{-9}$	$3.9 \cdot 10^{-9}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-4}$
8	$1.3 \cdot 10^{-9}$	$1.3 \cdot 10^{-9}$	$8.4 \cdot 10^{-6}$	$8.4 \cdot 10^{-6}$
10	$5.6 \cdot 10^{-10}$	$5.6 \cdot 10^{-10}$	$1.2 \cdot 10^{-6}$	$1.2 \cdot 10^{-6}$

Tabella 3.9: Errori relativi nel calcolo dell'integrale con le formula di cubatura compressa, tanto in MATLAB quanto in PYTHON.

L'equivalenza di tali risultati è ulteriormente evidenziata dagli errori relativi, che diminuiscono rapidamente verso la precisione di macchina, com'è possibile constatare dalla tabella 3.9.

3.2 Esperimenti su un elemento poligonale con un lato curvo e concavo

In questa sezione saranno proposti degli esperimenti di quadratura numerica su due elementi poligonali con un lato curvo di tipo concavo.

Analogamente ai casi con lato convesso, si analizzerà innanzitutto la cardinalità delle formule proposte. Nel caso delle formule compresse si constaterà che, pur risultando diverse, entrambe possiedono le

proprietà richieste, ovvero un adeguato grado di precisione e dei pesi positivi.

Come integranda consideriamo per entrambi i domini la funzione analitica

$$f_3(x, y) = e^{-(x-0.2)^2-(y-0.2)^2}. \tag{1}$$

Si tenga a mente che il proposito di questa trattazione non si limita all'osservazione dell'errore al variare del grado di precisione, ma richiede anche di verificare che il codice PYTHON esegua operazioni numericamente equivalenti a quelle proposte in MATLAB.

Nuovamente, per ottenere le formule compresse in PYTHON, è stata utilizzata la routine `comprexcub` in cui l'algoritmo di Lawson-Hanson è implementato dalla funzione `scipy.optimize.nnls`.

3.2.1 Un primo elemento poligonale con lato curvo e concavo

In questa sezione consideriamo l'elemento composto dalla spezzata che unisce i vertici (0.25,0.2), (0.2,0.25), (0, 0.25), (0.25,0) e dall'arco della circonferenza di centro (0,0) e raggio 0.25 unente i punti (0,0.25), (0.25,0).

Il dominio è descritto nella figure 3.3, assieme al corrispettivo insieme di nodi delle formule originali e compresse per $ADE = 10$. Come per l'esempio convesso, si propone anche qui, nella figura 3.4, il grafico dei nodi ordinati rispettivamente originali e compressi, nuovamente con $ADE = 10$.

Si riporta poi nella tabella 3.10, come per tutti gli altri domini trattati, il numero di nodi utilizzati nelle forme di integrazione.

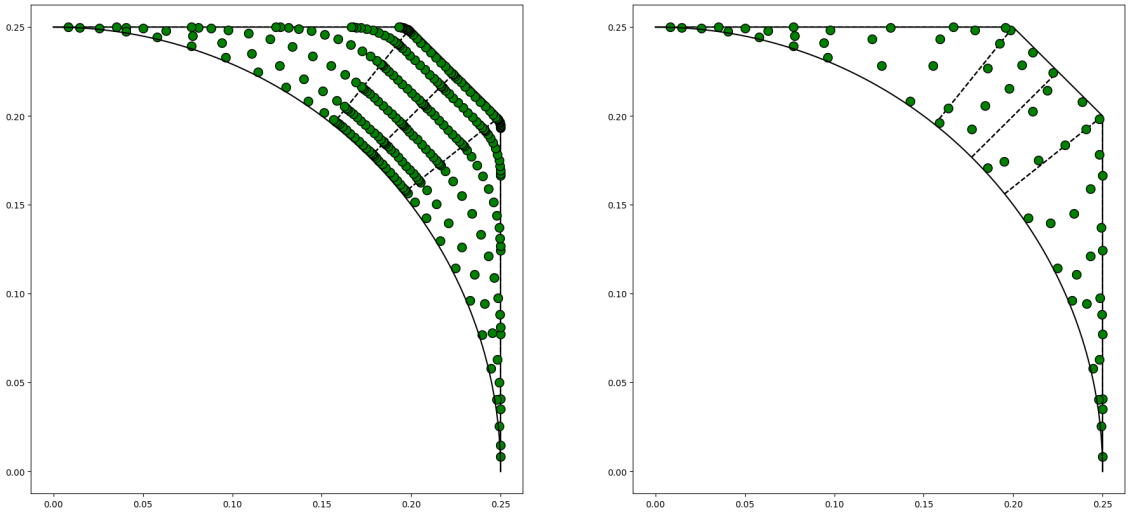

Figura 3.3: Nodi per le formule di cubatura con $ADE = 10$, relativamente alla formula originale (a sinistra) e alla sua versione compressa (a destra).

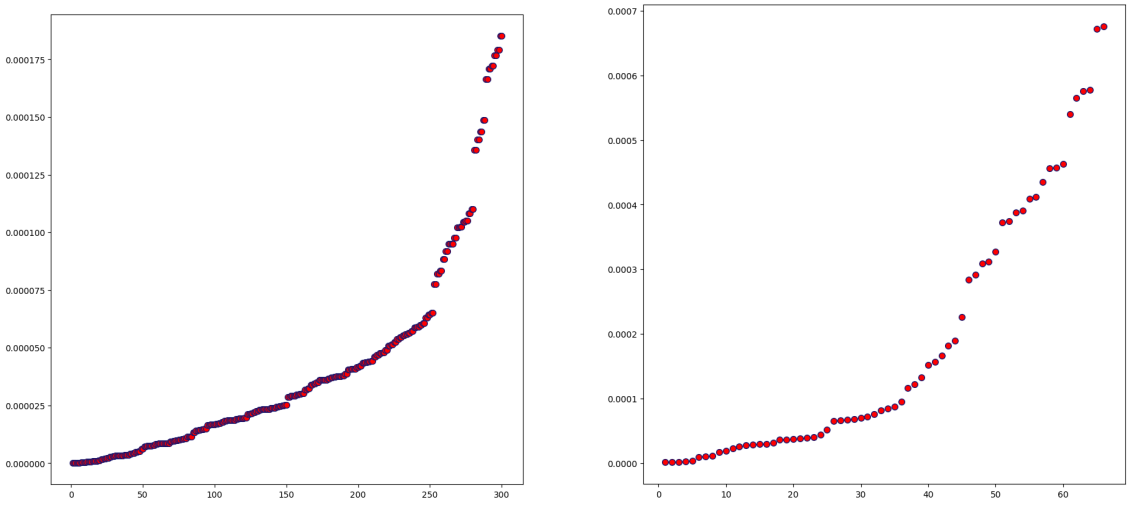

Figura 3.4: Pesi ordinati per valore delle formule di cubatura con $ADE = 10$, relativamente alla formula originale (a sinistra) e alla sua versione compressa (a destra).

ADE	Card_M	Card_P	Compr_Card_M	Compr_Card_P
2	36	36	6	6
4	78	78	15	15
6	136	136	28	28
8	210	210	45	45
10	300	300	66	66

Tabella 3.10: Cardinalità delle formule di cubatura originali e compresse tanto in MATLAB quanto in PYTHON.

Come atteso, anche in questo caso (tabella 3.10) e nel seguente, come sarà possibile constatare dalla tabella 3.17, indipendentemente dal linguaggio di programmazione utilizzato, il programma restituisce sempre lo stesso numero di nodi, sia applicando la formula originale sia calcolandone la controparte compressa.

Si confrontano quindi le formule proposte dagli algoritmi implementati nei due linguaggi: dalla tabella 3.11, si verifica nuovamente che le formule originali sono numericamente uguali in termini di nodi e pesi; al contrario, per quanto riguarda le loro versioni compresse, queste non sono equivalenti.

ADE	E_{nodes}	$E_{weights}$	$E^{(c)}_{nodes}$	$E^{(c)}_{weights}$
2	$5.3 \cdot 10^{-16}$	$3.9 \cdot 10^{-16}$	$1.0 \cdot 10^{-1}$	$5.3 \cdot 10^{-3}$
4	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$8.0 \cdot 10^{-2}$	$3.9 \cdot 10^{-3}$
6	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$1.1 \cdot 10^{-14}$
8	$5.3 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$1.5 \cdot 10^{-1}$	$8.2 \cdot 10^{-4}$
10	$5.9 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$5.9 \cdot 10^{-16}$	$7.9 \cdot 10^{-10}$

Tabella 3.11: Scarti per nodi e pesi, sia compressi che non, tra MATLAB e PYTHON sotto forma di norma infinito per il primo dominio di integrazione concavo.

È interessante notare come, in tutte e tre le tabelle 3.2, 3.11 e 3.18, nodi e pesi non compressi presentino un errore estremamente piccolo riconducibile alla precisione di macchina, mentre, per quanto riguarda nodi e pesi compressi, vi sono più risultati in cui la differenza dei valori assume dimensioni tutt'altro che indifferenti. Tuttavia, alla luce di quanto visto nella sezione 1.3.2 riguardo ai sistemi lineari (10) e (11), sappiamo che la formula di compressione non ha necessariamente soluzione unica. Tali differenze nei risultati, dunque, sono causate dalle differenze intrinseche dei due linguaggi e dei programmi che risolvono il sistema all'interno della funzione `comprexcub`, le quali portano a trovare in PYTHON un insieme di nodi e pesi differente, ma non per questo meno attendibile rispetto a quello trovato in MATLAB.

Per mostrare meglio questo dettaglio, confrontiamo nella tabella 3.12 gli errori di ricostruzione dei momenti calcolati da `comprexcub` rispettivamente in MATLAB e in PYTHON.

In dettaglio, il parametro `momerr` valuta l'errore sui momenti in norma 2 di una certa base polinomiale a grado n (cf. [10]) e, qualora sia vicino alla precisione di macchina, esso indica che la formula è numericamente esatta.

Quindi, indicati con

- $\texttt{momerr}_M$ il valore del parametro `momerr` ottenuto dal codice MATLAB,
- $\texttt{momerr}_P$ il valore del parametro `momerr` ottenuto dal codice PYTHON,

dalla tabella 3.12 deduciamo che le formule compresse, per quanto diverse, sono entrambe numericamente esatte.

ADE	$\texttt{momerr}_M$	$\texttt{momerr}_P$
2	$5.1 \cdot 10^{-19}$	$2.0 \cdot 10^{-18}$
4	$6.0 \cdot 10^{-19}$	$1.3 \cdot 10^{-18}$
6	$8.3 \cdot 10^{-19}$	$5.3 \cdot 10^{-19}$
8	$9.2 \cdot 10^{-19}$	$6.1 \cdot 10^{-19}$
10	$1.1 \cdot 10^{-18}$	$5.6 \cdot 10^{-19}$

Tabella 3.12: Errori dei momenti calcolati da `comprexcub` tanto in MATLAB quanto in PYTHON.

Per quanto riguarda l'integrale della funzione (1), utilizzando il codice MATLAB con ADE=60 otteniamo il valore di riferimento

$$INT = 1.2106206423126596 \cdot 10^{-2}.$$

Come per gli esperimenti precedenti, anche qui i tempi di calcolo del codice riportati sono stati ottenuti dalla media dei tempi di 20 lanci del programma per ciascun grado di precisione, trascurando i tempi per la valutazione degli integrali. In questo test, i sorgenti in Matlab risultano leggermente più rapidi dei corrispondenti in Python, i quali riescono comunque a rimanere nell'ordine dei centesimi/millesimi di secondo.

ADE	Int_M	Int_P
2	$1.2106186263541621 \cdot 10^{-2}$	$1.2106186263541615 \cdot 10^{-2}$
4	$1.2106206436421048 \cdot 10^{-2}$	$1.2106206436421046 \cdot 10^{-2}$
6	$1.2106206423119728 \cdot 10^{-2}$	$1.2106206423119719 \cdot 10^{-2}$
8	$1.2106206423126557 \cdot 10^{-2}$	$1.2106206423126570 \cdot 10^{-2}$
10	$1.2106206423126566 \cdot 10^{-2}$	$1.2106206423126559 \cdot 10^{-2}$

Tabella 3.13: Calcolo dell'integrale con le formula di cubatura originali, tanto in MATLAB quanto in PYTHON.

ADE	Compr_Int_M	Compr_Int_P
2	$1.2106300117877378 \cdot 10^{-2}$	$1.2106300117877363 \cdot 10^{-2}$
4	$1.2106206381439225 \cdot 10^{-2}$	$1.2106206381439236 \cdot 10^{-2}$
6	$1.2106206423156058 \cdot 10^{-2}$	$1.2106206423156051 \cdot 10^{-2}$
8	$1.2106206423126503 \cdot 10^{-2}$	$1.2106206423126500 \cdot 10^{-2}$
10	$1.2106206423126570 \cdot 10^{-2}$	$1.2106206423126572 \cdot 10^{-2}$

Tabella 3.14: Calcolo dell'integrale con le formula di cubatura compressa, tanto in MATLAB quanto in PYTHON.

ADE	ER_M	ER_P	ERC_M	ERC_P
2	$1.7 \cdot 10^{-6}$	$1.7 \cdot 10^{-6}$	$7.7 \cdot 10^{-6}$	$7.7 \cdot 10^{-6}$
4	$1.1 \cdot 10^{-9}$	$1.1 \cdot 10^{-9}$	$3.4 \cdot 10^{-9}$	$3.4 \cdot 10^{-9}$
6	$5.7 \cdot 10^{-13}$	$5.7 \cdot 10^{-13}$	$2.4 \cdot 10^{-12}$	$2.4 \cdot 10^{-12}$
8	$3.2 \cdot 10^{-15}$	$2.1 \cdot 10^{-15}$	$7.6 \cdot 10^{-15}$	$7.9 \cdot 10^{-15}$
10	$2.4 \cdot 10^{-15}$	$3.0 \cdot 10^{-15}$	$2.5 \cdot 10^{-15}$	$2.0 \cdot 10^{-15}$

Tabella 3.15: Errori relativi nel calcolo dell'integrale con le formula di cubatura compressa, tanto in MATLAB quanto in PYTHON.

ADE	Rule_Time_M	Rule_Time_P
2	$1.658 \cdot 10^{-3}$	$3.190 \cdot 10^{-3}$
4	$1.805 \cdot 10^{-3}$	$3.639 \cdot 10^{-3}$
6	$1.700 \cdot 10^{-3}$	$4.594 \cdot 10^{-3}$
8	$2.189 \cdot 10^{-3}$	$1.175 \cdot 10^{-2}$
10	$2.398 \cdot 10^{-3}$	$1.580 \cdot 10^{-2}$

ADE	C_Rule_Int_M	C_Rule_Int_P
2	$2.210 \cdot 10^{-3}$	$3.755 \cdot 10^{-3}$
4	$2.405 \cdot 10^{-3}$	$4.284 \cdot 10^{-3}$
6	$3.047 \cdot 10^{-3}$	$5.749 \cdot 10^{-3}$
8	$4.785 \cdot 10^{-3}$	$1.557 \cdot 10^{-2}$
10	$7.418 \cdot 10^{-3}$	$3.477 \cdot 10^{-2}$

Tabella 3.16: Tempi di calcolo con le formule di cubatura originali e compresse, tanto in MATLAB quanto in PYTHON.

3.2.2 Secondo esperimento su un elemento poligonale con lato curvo e concavo

Si consideri quale secondo elemento poligonale con lato curvo e concavo, quello composto dalla spezzata che unisce i vertici (0.4,0.05), (0.5,0.25), (0.45,0.45), (0.3, 0.5), (0.1, 0.45), (0, 0.25), (0.25,0) e dall'arco della circonferenza di centro (0,0) e raggio 0.25 unente i punti (0,0.25), (0.25,0).

Il dominio è descritto nelle figure 3.5, assieme al corrispettivo insieme di nodi delle formule originali e compresse per $ADE = 10$. Come per gli altri domini trattati, si mostrano anche i grafici dei pesi ordinati, originali e compressi, nelle figure 3.6.

Come in precedenza, la cardinalità delle formule risulta uguale, tanto nel caso originale quanto in quello compresso.

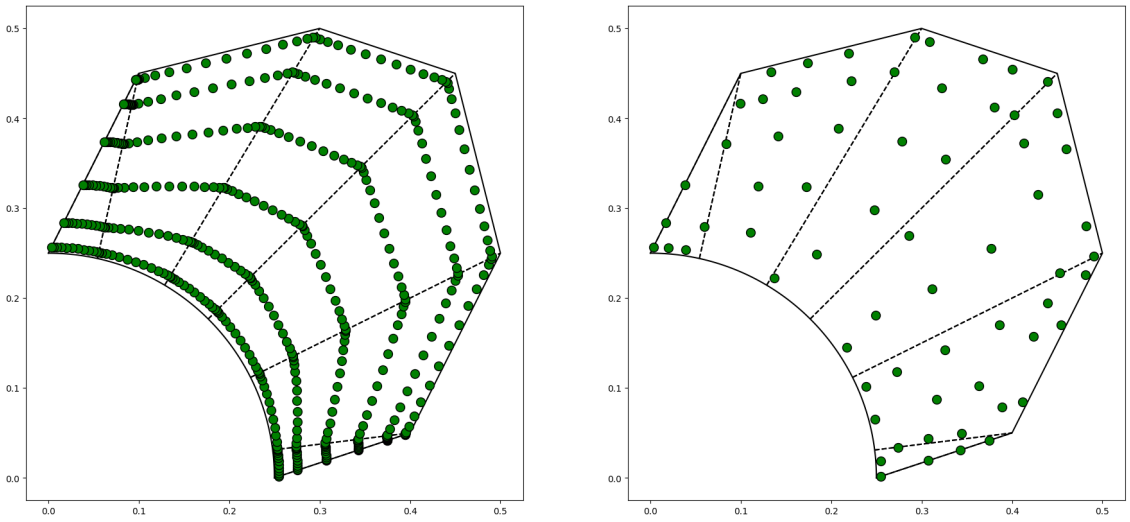

Figura 3.5: Nodi per le formule di cubatura con $ADE = 10$, relativamente alla formula originale (a sinistra) e alla sua versione compressa (a destra).

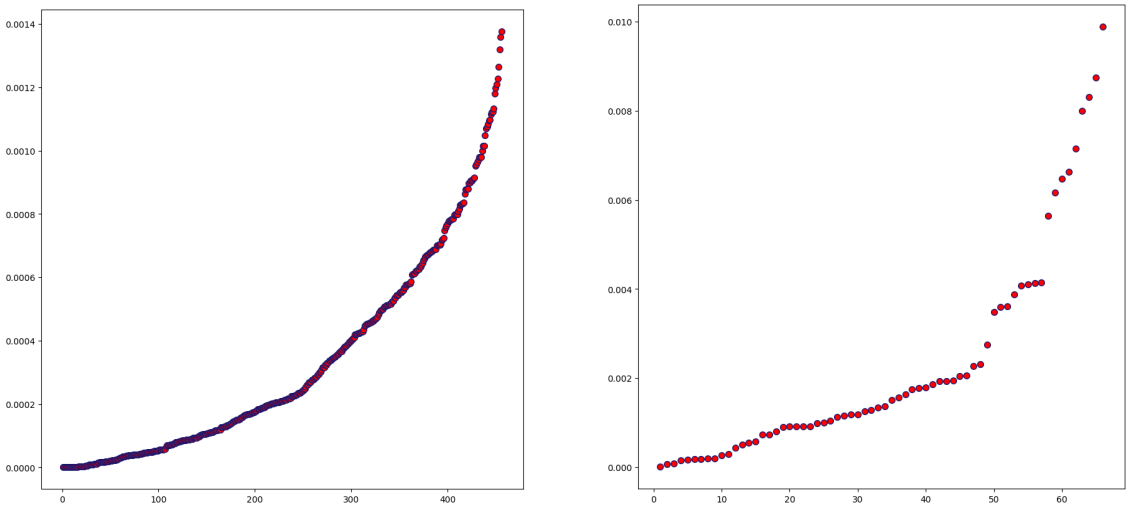

Figura 3.6: Pesi ordinati per valore delle formule di cubatura con $ADE = 10$, relativamente alla formula originale (a sinistra) e alla sua versione compressa (a destra).

Nel paragone tra nodi e pesi, nelle formule prodotte dai due linguaggi di programmazione, come per il dominio concavo precedente, le formule originali sono numericamente uguali, mentre non lo sono

ADE	Card_M	Card_P	Compr_Card_M	Compr_Card_P
2	56	56	6	6
4	120	120	15	15
6	208	208	28	28
8	320	320	45	45
10	456	456	66	66

Tabella 3.17: Cardinalità delle formule di cubatura originali e compresse tanto in MATLAB quanto in PYTHON.

quelle compresse. Nonostante questo, con ragionamenti analoghi, risultano nuovamente entrambe numericamente esatte, come descritto dalle tabelle 3.18 e 3.19.

ADE	E_{nodes}	$E_{weights}$	$E^{(c)}_{nodes}$	$E^{(c)}_{weights}$
2	$5.0 \cdot 10^{-16}$	$4.7 \cdot 10^{-16}$	$4.4 \cdot 10^{-16}$	$4.9 \cdot 10^{-16}$
4	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$4.4 \cdot 10^{-16}$	$4.8 \cdot 10^{-16}$
6	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$3.3 \cdot 10^{-1}$	$1.7 \cdot 10^{-2}$
8	$5.3 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$1.9 \cdot 10^{-1}$	$5.0 \cdot 10^{-3}$
10	$5.6 \cdot 10^{-16}$	$5.0 \cdot 10^{-16}$	$3.0 \cdot 10^{-1}$	$6.7 \cdot 10^{-3}$

Tabella 3.18: Scarti per nodi e pesi, sia compressi che non, tra MATLAB e PYTHON sotto forma di norma infinito per il secondo dominio di integrazione concavo.

ADE	$\texttt{momerr}_M$	$\texttt{momerr}_P$
2	$7.4 \cdot 10^{-18}$	$5.9 \cdot 10^{-18}$
4	$9.0 \cdot 10^{-18}$	$7.1 \cdot 10^{-18}$
6	$9.9 \cdot 10^{-18}$	$6.8 \cdot 10^{-18}$
8	$9.3 \cdot 10^{-18}$	$6.3 \cdot 10^{-18}$
10	$1.4 \cdot 10^{-17}$	$5.5 \cdot 10^{-18}$

Tabella 3.19: Errori dei momenti calcolati da `comprexcub` per il secondo elemento concavo tanto in MATLAB quanto in PYTHON.

Per quanto riguarda l'integrale della funzione (1), utilizzando il codice MATLAB con ADE=60, si ottiene il valore di riferimento

$$INT = 0.145174464220773263845.$$

ADE	Int_M	Int_P
2	$1.4516536090623977 \cdot 10^{-1}$	$1.4516536090623974 \cdot 10^{-1}$
4	$1.4517449328603046 \cdot 10^{-1}$	$1.4517449328603046 \cdot 10^{-1}$
6	$1.4517446415284663 \cdot 10^{-1}$	$1.4517446415284657 \cdot 10^{-1}$
8	$1.4517446422089852 \cdot 10^{-1}$	$1.4517446422089839 \cdot 10^{-1}$
10	$1.4517446422077290 \cdot 10^{-1}$	$1.4517446422077293 \cdot 10^{-1}$

Tabella 3.20: Calcolo dell'integrale con le formula di cubatura originali, tanto in MATLAB quanto in PYTHON.

ADE	Compr_Int_M	Compr_Int_P
2	$1.4519145766083637 \cdot 10^{-1}$	$1.4519145766083622 \cdot 10^{-1}$
4	$1.4517441656674904 \cdot 10^{-1}$	$1.4517441656674912 \cdot 10^{-1}$
6	$1.4517446407001958 \cdot 10^{-1}$	$1.4517446407001944 \cdot 10^{-1}$
8	$1.4517446422105307 \cdot 10^{-1}$	$1.4517446422105293 \cdot 10^{-1}$
10	$1.4517446422077313 \cdot 10^{-1}$	$1.4517446422077310 \cdot 10^{-1}$

Tabella 3.21: Calcolo dell'integrale con le formula di cubatura compressa, tanto in MATLAB quanto in PYTHON.

I valori degli integrali e degli errori relativi sono praticamente identici, a riprova dell'equivalenza numerica delle formule.

Come in precedenza valutiamo i tempi di calcolo del codice ottenuti dalla media dei tempi di 20 lanci del programma per ciascun grado di precisione, trascurando i tempi per la valutazione degli integrali. Anche in questo test, i sorgenti in Matlab risultano leggermente più rapidi dei corrispondenti in Python.

ADE	ER_M	ER_P	ERC_M	ERC_P
2	$6.3 \cdot 10^{-5}$	$6.3 \cdot 10^{-5}$	$1.2 \cdot 10^{-4}$	$1.2 \cdot 10^{-4}$
4	$2.0 \cdot 10^{-7}$	$2.0 \cdot 10^{-7}$	$3.3 \cdot 10^{-7}$	$3.3 \cdot 10^{-7}$
6	$4.7 \cdot 10^{-10}$	$4.7 \cdot 10^{-10}$	$1.0 \cdot 10^{-9}$	$1.0 \cdot 10^{-9}$
8	$8.6 \cdot 10^{-13}$	$8.6 \cdot 10^{-13}$	$1.9 \cdot 10^{-12}$	$1.9 \cdot 10^{-12}$
10	$2.5 \cdot 10^{-15}$	$2.3 \cdot 10^{-15}$	$9.6 \cdot 10^{-16}$	$1.1 \cdot 10^{-15}$

Tabella 3.22: Errori relativi nel calcolo dell'integrale con le formula di cubatura compressa, tanto in MATLAB quanto in PYTHON.

ADE	Rule_Time_M	Rule_Time_P
2	$2.545 \cdot 10^{-3}$	$5.410 \cdot 10^{-3}$
4	$1.546 \cdot 10^{-3}$	$5.432 \cdot 10^{-3}$
6	$1.845 \cdot 10^{-3}$	$6.905 \cdot 10^{-3}$
8	$1.991 \cdot 10^{-3}$	$1.698 \cdot 10^{-2}$
10	$2.786 \cdot 10^{-3}$	$2.169 \cdot 10^{-2}$

ADE	C_Rule_Int_M	C_Rule_Int_P
2	$2.406 \cdot 10^{-3}$	$5.375 \cdot 10^{-3}$
4	$2.337 \cdot 10^{-3}$	$6.218 \cdot 10^{-3}$
6	$3.376 \cdot 10^{-3}$	$8.459 \cdot 10^{-3}$
8	$4.514 \cdot 10^{-3}$	$2.845 \cdot 10^{-2}$
10	$9.514 \cdot 10^{-3}$	$5.807 \cdot 10^{-2}$

Tabella 3.23: Tempi di calcolo con le formule di cubatura originali e compresse, tanto in MATLAB quanto in PYTHON.

A seguito degli esperimenti trattati, è possibile concludere che, seppur con occasionali differenze per quanto riguarda gli algoritmi risolutivi, i codici possono definirsi in tutto e per tutto equivalenti, dal momento che anche gli scarti a livello di tempi non presentano differenze significative.

Bibliografia

[1] E. Artioli, A. Sommariva, M. Vianello, *Algebraic cubature on polygonal elements with a circular edge*, Computers Mathematics with Applications, Volume 79, Volume 7, 1 April 2020, pp.2057-2066.

[2] C. Caratheodory, *Über den Variabilittsbereich der Koeffizienten von Potenzreihen, die gegebene Werte nicht annehmen*, Math. Ann. 64 (1907), pp.95–115.

[3] R. Cools, *Constructing cubature formulae: the science behind the art*, Acta Numer. 6 (1997), pp.1–54.

[4] R. Cools, *An encyclopaedia of cubature formulas*, J. Complexity 19 (2003), pp.445–453.

[5] G. Da Fies, A. Sommariva, M. Vianello, *Algebraic cubature by linear blending of elliptical arcs*, Appl. Numer. Math. 74 (2013), pp.49–61.

[6] G. Da Fies, M. Vianello, *Trigonometric Gaussian quadrature on subintervals of the period*, Electron. Trans. Numer. Anal. 39 (2012), pp.102–112.

[7] C.L. Lawson, R.J. Hanson, *Solving least squares problems, Classics in Applied Mathematics*, 15, SIAM, Philadelphia, 1995.

[8] F. Piazzon, A. Sommariva, M. Vianello, *Caratheodory-Tchakaloff Subsampling*, Dolomites Res. Notes Approx. DRNA 10 (2017), pp.5–14.

[9] L. Rinaldi, *GitHub codes*, `https://github.com/laura-rinaldi/Cheap_Splinegauss`

[10] A. Sommariva, M. Vianello, *Compression of multivariate discrete measures and applications*, Numer. Funct. Anal. Optim. 36 (2015), pp.1198–1223.

[11] A. Sommariva, *Cubature codes and pointsets*, `http://www.math.unipd.it/~alvise/software.html`

[12] M. Tchernychova, *Caratheodory cubature measures*, Tesi di Dottorato in Matematica (supervisore T.Lyons), Università di Oxford, 2015.

[13] G. Traversin, PYTHON *package for numerical cubature on polygonal elements with one curved edge* `https://github.com/2002232/Squaring-polygonal-elements-with-one-curved-side-in-Python.git`

[14] M. Vianello, *UBP:* MATLAB *package for subperiodic trigonometric quadrature and multivariate applications*, `http://www.math.unipd.it/~marcov/subp.html`

Appendice

Vengono di seguito riportate le principali routines tradotte in PYTHON in questa tesi, con i relativi commenti ed eventuale documentazione.

Gli stessi codici sono disponibili, anche come programmi singoli, in forma open-source al link

`https://github.com/2002232/Squaring-polygonal-elements-with-one-curved-side-in-Python.git`

Librerie utilizzate

```
# Codes transaltion
import numpy as np
from scipy import integrate
from math import gamma
from scipy.optimize import nnls

# Numerical results
import time
import statistics
import matplotlib.pyplot as plt
```

LHDM e DM

```
def LHDM(C, d, options=None, verbose=0):

# Code author: Laura Rinaldi
# https://github.com/laura-rinaldi/Cheap_Splinegauss

# Lawson-Hanson algorithm accelerated by Deviation Maximization (DM).

#     Parameters:
#         C (numpy.ndarray): Least squares matrix.
#         d (numpy.ndarray): Right-hand side vector.
#         options (dict, optional): Optimization parameters:
#             - init (bool): Use ULS initialization if True.
#             - tol (float): Tolerance for projected residual.
#             - k (int): Max number of indices added to Passive set
#               each iteration.
#             - thres (float): Threshold for angle between columns (0 to 1).
#         verbose (int, optional): Verbosity level.

#     Returns:
#         x (numpy.ndarray): Nonnegative solution minimizing ||C*x - d||.
#         resnorm (float): Squared residual norm ||C*x - d||^2.
#         exitflag (int): Exit condition (1: success, 0: exceeded iteration).
#         outeriter (int): Number of outer iterations.
#
    if options is None:
        options = {}
```

```
m, n = C.shape
nZeros = np.zeros(n)
wz = np.zeros(n)
itmax = 2 * m

# Initialize sets
P = np.zeros(n, dtype=bool)
Z = np.ones(n, dtype=bool)
x = np.zeros(n)

thres = options.get('thres', 0.2222)
thres_w = options.get('thres_w', 0.8)
k = options.get('k', max(1, m // 20))
tol = options.get('tol',10*np.finfo(float).eps*np.linalg.norm(C, 1)*len(C))
LHDMflag = k > 1

if verbose:
    print(f"LHDM({k}){'with ULS initialization'if\
    options.get('init',False)else ''}")

if LHDMflag:
    Cnorm = C / np.linalg.norm(C, axis=0)

# Initialize residual and dual variables
resid = d - C @ x
w = C.T @ resid
outeriter = 0
totiter = 0

while np.any(Z) and (np.any(w[Z] > tol) or np.any(x[P] < 0)) and totiter < itmax:
    outeriter += 1
    totiter += 1

    wz[P] = -np.inf
    wz[Z] = w[Z]

    if outeriter == 1 or not LHDMflag:
        t = np.argmax(wz)
    else:
        t = DM(Cnorm, wz, k, thres, thres_w)
        t = t[:min(len(t), m - np.sum(P))]

    addedP = np.shape(t)
    z = np.zeros_like(x)
    P[t] = True
    Z[t] = False
```

```
        z[P] = np.linalg.lstsq(C[:, P], d, rcond=None)[0]

        iter = 0
        removedP = 0

        while np.any(z[P] <= 0) and totiter < itmax:
            totiter += 1
            iter += 1
            Q = (z <= 0) & P
            alpha = np.min(x[Q] / (x[Q] - z[Q]))
            x = x + alpha * (z - x)
            t = np.where((np.abs(x) < tol) & P)[0]
            removedP += len(t)
            Z[(np.abs(x) < tol) & P] = True
            P = ~Z
            z[P] = np.linalg.lstsq(C[:, P], d, rcond=None)[0]

        x = z
        resid = d - C @ x
        w = C.T @ resid

    exitflag = 1 if outeriter < itmax else 0
    resnorm = np.dot(resid, resid)

    return x #, resnorm, exitflag, outeriter



def DM(Cnorm, wz, k, thres, thres_w):

# Code author: Laura Rinaldi
# https://github.com/laura-rinaldi/Cheap_Splinegauss

#     Deviation Maximization

#     :param Cnorm: 2D NumPy array
#     :param wz: 1D NumPy array
#     :param k: integer
#     :param thres: threshold value
#     :param thres_w: weight threshold multiplier
#     :return: list of indices

    wzI = np.sort(wz)[::-1]  # Sort in descending order
    I = np.argsort(wz)[::-1]  # Get sorted indices
    t = I[0]
```

```
    p = [t]

    thres_wloc = thres_w * wzI[0]
    C = np.where(wzI > thres_wloc)[0]  # Get indices of wzI above threshold

    n = C.shape[0]
    add = 1

    for i in range(1, n):
        c = C[i]
        max_dev = np.max(np.abs(Cnorm[:, I[c]].T @ Cnorm[:, p]))

        if max_dev < thres:
            p.insert(0, I[c])
            add += 1

        if add >= k:
            break

    return p
```

Chebpolys

```
def chebpolys(deg,x1):

# Code author: Giovanni Traversin
# Release date: 05 Sept 2025

# computes the Chebyshev-Vandermonde matrix on the real line by recurrence

# INPUT:
# deg = maximum polynomial degree
# x = list or array of abscissas

# OUTPUT:
# T = Chebyshev-Vandermonde matrix at x,
# T(i,j+1)=T_j(x_i), j=0,...,deg

    x = np.array(x1)
    n = len(x)
    T= np.zeros((n,deg+1))
    t0 = np.ones(n)
    t1 = x[:]
```

```
    T[:,0] = t0
    T[:,1] = t1

    for i in range(2,deg+1):
      t2 = 2*x*t1 - t0
      T[:,i] = t2
      t0 = t1[:]
      t1 = t2[:]

    return T
```

Chebvand

```
def chebvand(deg, x, rect = None):

# Code author: Giovanni Traversin
# Release date: 05 Sept 2025

# INPUT:
# deg = polynomial degree
# x = 2-column array or 2 arrays of the same length of point coordinates
# rect = 4-component vector such that the rectangle
# [rect(1),rect(2)] x [rect(3),rect(4)] contains X
# If you do not compile the input for rect, the program will use as rect the
# smaller rectangle containing all the points in X.

# OUTPUT:
# V = Chebyshev-Vandermonde matrix at x, graded lexic. order

  X = np.array(x)
  if rect is None:
    rect = [min(X[:,0]), max(X[:,0]), min(X[:,1]), max(X[:,1])]

  # couples with length less or equal to deg
  # graded lexicographical order
  j = np.arange(deg + 1)
  j1, j2 = np.meshgrid(j,j)
  jj = j1 + j2
  dim = (deg+1)*(deg+2)//2
  couples = np.zeros((dim, 2), dtype=int)

  for s in range(deg+1):
    good = np.argwhere(jj == s)
```

```
    a = s*(s+1)//2
    for i  in range(len(good)):
      couples[a+i] = [j1[good[i][1],good[i][0]], j2[good[i][1],good[i][0]]]

  # mapping the mesh in the square [-1,1]^2
  a = rect[0]; b = rect[1]; c = rect[2]; d = rect[3]
  # map = [(2*X[:,0]-b-a)/(b-a) , (2*X[:,1]-d-c)/(d-c)]

  # Chebyshev-Vandermonde matrix on the mesh
  T1 = chebpolys(deg, (2*X[:,0]-b-a)/(b-a))
  T2 = chebpolys(deg, (2*X[:,1]-d-c)/(d-c))

  return  T1[:, couples[:, 0]] * T2[:, couples[:, 1]] # = V
```

Comprexcub

```
def comprexcub(deg, x, omega, pos):

 # Code author: Giovanni Traversin
 # Release date: 05 Sept 2025

  # compression of bivariate cubature formulas by Tchakaloff points
  # or approximate Fekete points
  # useful, for example, in node reduction of algebraic cubature formulas
  # see the web page: http://www.math.unipd.it/~marcov/Tchakaloff.html

  # by Federico Piazzon, Alvise Sommariva and Marco Vianello
  # , May 2016


  # INPUT:
  # deg: polynomial exactness degree
  # X: 2-column array of point coordinates
  # omega: 1-column array of weights
  # pos: NNLS for pos=1, QR with column pivoting for pos=0

  # OUTPUT:
  # pts: 2-column array of extracted points
  # w: 1-column array of corresponding weights (positive for pos=1)
  # momerr: moment reconstruction error

  X = np.array(x)
  if len(X[0])>2:
```

```
    X = X.T
  rect = [min(X[:,0]), max(X[:,0]), min(X[:,1]), max(X[:,1])]
  V=chebvand(deg,X,rect)
  Q, R = np.linalg.qr(V)
  Q = np.real(Q)
  orthmom = Q.T @ omega
  match pos:
    case 1:
      weights = LHDM(Q.T,orthmom)
    case _:
      weights = nnls(Q.T,orthmom)[0]
  ind = np.where(abs(weights)>0)
  w = weights[ind]
  return X[ind, :][0], w, np.linalg.norm( Q[ind,:][0].T @ w - orthmom)
```

Gauss

```
def gauss(N, albet):

# Code author: Giovanni Traversin
# Release date: 05 Sept 2025

#    Gauss quadrature rule.
#
#    Given a weight function w encoded by the nx2 array albet of the
#    first n recurrence coefficients for the associated orthogonal
#    polynomials, the first column of ab containing the n alpha-
#    coefficients and the second column the n beta-coefficients,
#    the call xw=GAUSS(n,ab) generates the nodes and weights xw of
#    the n-point Gauss quadrature rule for the weight function w.
#    The nodes, in increasing order, are stored in the first
#    column, the n corresponding weights in the second column, of
#    the nx2 array xw.

  ab = np.array(albet)
  if len(ab[:,0]) < N:
      raise ValueError('Input array ab is too short')
  J = np.zeros((N, N))

  for i in range(N):
      J[i][i] = ab[i][0]
  for j in range(N-1):
      J[j+1][j] = np.sqrt(ab[j+1][1])
      J[j][j+1] = J[j+1][j]
```

```
  D, V = np.linalg.eigh(J)
  I = np.argsort(D)
  D = np.sort(D)
  V = V[:,I]

  return np.transpose((D , ab[0][1]*V[0,:]**2))
```

R_Jacobi

```
def r_jacobi(N,a=0,b=[]):

# Code author: Giovanni Traversin
# Release date: 05 Sept 2025

#  Recurrence coefficients for monic Jacobi polynomials.
#
#    ab=R_JACOBI(n,a,b) generates the first n recurrence
#    coefficients for monic Jacobi polynomials with parameters
#    a and b. These are orthogonal on [-1,1] relative to the
#    weight function w(t)=(1-t)^a(1+t)^b. The n alpha-coefficients
#    are stored in the first column, the n beta-coefficients in
#    the second column, of the nx2 array ab. The call ab=
#    R_JACOBI(n,a) is the same as ab=R_JACOBI(n,a,a) and
#    ab=R_JACOBI(n) the same as ab=R_JACOBI(n,0,0).
#
#    Supplied by Dirk Laurie, 6-22-1998; edited by Walter
#    Gautschi, 4-4-2002.

  if b == []:
    b = a
  if N<=0 or a<=-1 or b<=-1:
    raise ValueError('parameter(s) out of range')
  nu = (b-a)/(a+b+2)
  mu = (2**(a+b+1))*gamma(a+1)*gamma(b+1)/gamma(a+b+2)

  # Important note: scipy.special.gamma() takes arbitrary np.arrays as input.
  # math.gamma() requires float or single-element numpy arrays, which a
  # crippling limitation in many use cases. {
  # The function scipy.special.gamma allows complex numbers, even though
  # math.gamma doesn't.

  if N == 1:
```

```
    return np.array([nu,mu])
  NO = int(N)
  n = np.array(range(1,NO))
  nab = 2*n+a+b
  A = np.append((b**2-a**2)*np.ones(NO-1)/(nab*(nab+2)),nu)
  n = n[1:]
  nab = nab[1:]
  B1 = 4*(a+1)*(b+1)/((a+b+2)**2*(a+b+3))
  B = 4*(n+a)*(n+b)*n*(n+a+b)/((nab**2)*(nab+1)*(nab-1))
  ab = [mu] + [B1]
  for i in B:
    ab = ab + [i]
  ab = [A] + [ab]


  return   np.transpose(ab)
```

Chebyshev

```
def chebyshev(N, Mom, Abm=False):

# Code author: Giovanni Traversin
# Release date: 05 Sept 2025

#  Modified Chebyshev algorithm.

#  Given a weight function w encoded by its first 2n modified
#  moments, stored in the (row) vector mom, relative to monic
#  polynomials defined by the (2n-1)x2 array abm of their
#  recurrence coefficients, [ab,normsq]=CHEBYSHEV(n,mom,abm)
#  generates the array ab of the first n recurrence coefficients
#  of the orthogonal polynomials for the weight function w, and
#  the vector normsq of their squared norms. The n alpha-
#  coefficients are stored in the first column, the n beta-
#  coefficients in the second column, of the nx2 array ab. The
#  call [ab,normsq]=CHEBYSHEV(n,mom) does the same, but using the
#  classical Chebyshev algorithm. If n is larger than the sizes
#  of mom and abm warrant, then n is reduced accordingly.

  mom = np.array(Mom)
  normsq = []
  if N <= 0:
     raise ('N out of range')
```

```
  if N > len(mom)/2:
    N = len(mom)/2

  if type(Abm) == bool:
    abm = np.zeros((2,int(2*N-1)))
  else:
    abm = np.array(Abm)
  if len(abm[0]) == 2 and len(abm)!=2:
      abm = abm.T

  if N > (len(abm[0])+1)/2:
    N = (len(abm[0])+1)/2

  if N == 1:
    normsq = [mom[0]]

  N = int(N)
  ab =  np.array([[abm[0][0]+mom[1]/mom[0], mom[0]]]+[[0,0]]*(N-1))
  sig = np.zeros((N+1,2*N))
  sig[1] = mom[0:int(2*N)]

  #NOTE: abm is a matrix with two raws instead of a matrix with two
  # columns, therefore the indexes are reversed.

  for n in range(2,N+1):
    for m in range(n-1,(2*N-n+1)):
      sig[n][m] = sig[n-1][m+1]-(ab[n-2][0]-abm[0][m])*sig[n-1][m]-\
      ab[n-2][1]*sig[n-2][m]+abm[1][m]*sig[n-1][m-1]
    ab[n-1,0] = abm[0][n-1] + sig[n][n]/sig[n][n-1] - \
    sig[n-1][n-1]/sig[n-1][n-2]
    ab[n-1,1] = sig[n][n-1]/sig[n-1][n-2]

  for i in range(N):
    normsq.append(sig[i+1][i])


  return ab, np.array(normsq)
```

Tridisolve

```
def tridisolve(a,b,c,d):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025
```

```
  #   TRIDISOLVE  Solve tridiagonal system of equations.
  # From Cleve Moler's Matlab suite
  # http://www.mathworks.it/moler/ncmfilelist.html
  #     x = TRIDISOLVE(a,b,c,d) solves the system of linear equations
  #     b(1)*x(1) + c(1)*x(2) = d(1),
  #     a(j-1)*x(j-1) + b(j)*x(j) + c(j)*x(j+1) = d(j), j = 2:n-1,
  #     a(n-1)*x(n-1) + b(n)*x(n) = d(n).
  #
  #   The algorithm does not use pivoting, so the results might
  #   be inaccurate if abs(b) is much smaller than abs(a)+abs(c).
  #   More robust, but slower, alternatives with pivoting are:
  #     x = T\d where T = diag(a,-1) + diag(b,0) + diag(c,1)
  #     x = S\d where S = spdiags([[a; 0] b [0; c]],[-1 0 1],n,n)

  x = d
  n = len(x)-1
  for j in range(0,n):
    mu = a[j]/b[j]
    b[j+1] = b[j+1] - mu*c[j]
    x[j+1] = x[j+1] - mu*x[j]

  x[n] = x[n]/b[n]
  for i in range(n)[::-1]:
    x[i] = (x[i]-c[i]*x[i+1])/b[i]

  return x
```

Trigauss

```
def trigauss(n, alpha, beta):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  # Original code by Gaspare Da Fies and Marco Vianello,
  # 8 Nov 2011

  # computes the n+1 angles and weights of a trigonometric gaussian
  # quadrature formula on [alpha,beta], 0<beta-alpha<=pi

  # uses the routines chebyshev.m, gauss.m from
  # W. Gautschi repository (not available online)
  # we suggest to put the following statements
```

```
# ab = zeros(N,2); sig = zeros(N+1,2*N);
# at the beginning of the body of chebyshev.m to speed-up execution

# input:
# n: trigonometric degree of exactness
# [alpha,beta]: angular interval, 0<beta-alpha<=pi

# output:
# tw: (n+1) x 2 array of (angles,weights)

# the formula integrates the canonical trigonometric basis with accuracy
# from about 10^(-15) (small omega) to about 10^(-13) (omega-->pi)
# up to n=300

def integrand(t, n, omega):
  return np.cos(2*n*np.arccos(np.sin(t/2)/np.sin(omega/2)))

omega=(beta-alpha)/2       # half-length of the angular interval

#modified Chebyshev moments by recurrence
z = np.zeros(n+1)
z[0] = 2 * omega
z[n] = integrate.quad(integrand,-omega, \
                      omega, args=(n, omega), limit=5000)[0]

temp = np.arange(2, 2*n, 2)  # 2,4,...,2n-2
dl = 1/4 - 1/(4*(temp-1))
dc = 1/2 - 1/(np.sin(omega/2)**2) - 1/(2*(temp**2-1))
du = 1/4 + 1/(4*(temp+1))
d = 4 * np.cos(omega/2)/np.sin(omega/2)/(temp**2 - 1)
d[n-2] = d[n-2] - du[n-2] * z[n]
z[1:n] = tridisolve(dl[1:n-1], dc[0:n-1], du[0:n-2], d[0:n-1])
mom = np.zeros(2*n+2)
mom[0:2*n+1:2] = z  # fill even indices (0,2,4,...) with z

# normalization of the moments (monic polynomials)
k = np.arange(3, len(mom)+1)
mom[2:] *= np.exp((2-k)*np.log(2))

# recurrence coeffs of the monic Chebyshev polynomials
abm = np.zeros((2*n+1, 2))
abm[:,1] = 0.25
abm[0,1] = np.pi
abm[1,1] = 0.5

# recurrence coeffs for the monic OPS w.r.t. the weight function
```

```
  # w(x)=2*sin(omega/2)/sqrt(1-sin^2(omega/2)*x^2) by the
  # modified Chebyshev algorithm
  ab, normsq = chebyshev(n+1,mom,abm)

  # Gaussian formula for the weight function above
  xw = gauss(n+1,ab)

  # angles and weights for the trigonometric gaussian formula
  return  np.column_stack((2*np.arcsin(np.sin(omega/2)*xw[:,0]) + \
   (beta+alpha)/2, xw[:,1]))
```

Gqellblend

```
def gqellblend(n, A, B, C, alpha, beta):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  # Original code by Gaspare Da Fies, Alvise Sommariva and Marco Vianello

  # 2 June 2011

  # computes the nodes and weights of a product gaussian formula
  # exact on total-degree bivariate polynomials of degree <=n
  # on the planar region R obtained by linear blending (convex combination)
  # of two trigonometric arcs with parametric equations
  # P(theta)=A1*cos(theta)+B1*sin(theta)+C1
  # Q(theta)=A2*cos(theta)+B2*sin(theta)+C2
  # namely
  # R = {(x,y)=t*P(theta)+(1-t)*Q(theta), t in [0,1], theta in [alpha,beta],
  # 0<beta-alpha<=2*pi}

  # uses the routines:
  #
  # r_jacobi.m, gauss.m from
  # www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html
  #
  # trigauss.m
  # http://www.math.unipd.it/~marcov/mysoft/trigauss.m
  # this will be soon substituted by an optimized version input

  # input:
  # n: algebraic degree of exactness
  # A,B,C: 2x2 matrices of the parametric arc coefficients:
```

```
# [alpha,beta]: angular interval, 0<beta-alpha<=2*pi

# output:
# xyw: 3 columns array of (xnodes,ynodes,weights)

S1 = abs((A[0][0]-A[1][0])*(B[0][1]-B[1][1])+(A[0][1]-A[1][1])*\
 (B[1][0]-B[0][0]))>10*np.finfo(float).eps
S2 = abs((C[0][0]-C[1][0])*(B[0][1]-B[1][1])+(C[0][1]-C[1][1])*\
 (B[1][0]-B[0][0]))>10*np.finfo(float).eps
S3 = abs((A[0][0]-A[1][0])*(C[0][1]-C[1][1])+(A[0][1]-A[1][1])*\
 (C[1][0]-C[0][0]))>10*np.finfo(float).eps

if S1 or S2 or S3:
  h=1
else:
  h=0

S4 = abs(A[0][1]*A[1][0]-A[0][0]*A[1][1]-B[0][1]*B[1][0]+\
         B[0][0]*B[1][1])>10*np.finfo(float).eps
S5 = abs(A[0][1]*B[1][0]-A[0][0]*B[1][1]+B[0][1]*A[1][0]-\
         B[0][0]*A[1][1])>10*np.finfo(float).eps
S6 = abs(B[1][0]*(C[0][1]-C[1][1])-B[1][1]*(C[0][0]-C[1][0]))\
>10*np.finfo(float).eps
S7 = abs(A[1][0]*(C[0][1]-C[1][1])-A[1][1]*(C[0][0]-C[1][0]))\
>10*np.finfo(float).eps
S8 = abs((C[0][0]-C[1][0])*(B[0][1]-B[1][1])+(C[0][1]-C[1][1])*\
 (B[1][0]-B[0][0]))>10*np.finfo(float).eps
S9 = abs((A[0][0]-A[1][0])*(C[0][1]-C[1][1])+(A[0][1]-A[1][1])*\
 (C[1][0]-C[0][0]))>10*np.finfo(float).eps

if S4 or S5:
  k = 2
elif S6 or S7 or S8 or S9:
  k = 1
else:
  k = 0

# trigonometric gaussian formula on the arc
tw=trigauss(n+k,alpha,beta)

# algebraic gaussian formula on [0,1]
ab=r_jacobi(int(np.ceil((n+h+1)/2)),0,0)

xw=gauss(int(np.ceil((n+h+1)/2)),ab)
xw[:,0]=xw[:,0]/2+1/2
xw[:,1]=xw[:,1]/2
```

```
  #creating the grid
  t, theta = np.meshgrid(xw[:,0], tw[:,0])
  w1, w2 = np.meshgrid(xw[:,1], tw[:,1])

  # nodal cartesian coordinates and weights
  theta = theta.T.flatten()
  t = t.T.flatten()
  w1 = w1.T.flatten()
  w2 = w2.T.flatten()
  s = np.sin(theta)
  c = np.cos(theta)
  p1 = A[0][0]*c+B[0][0]*s+C[0][0]
  p2 = A[0][1]*c+B[0][1]*s+C[0][1]
  q1 = A[1][0]*c+B[1][0]*s+C[1][0]
  q2 = A[1][1]*c+B[1][1]*s+C[1][1]
  dp1 = -A[0][0]*s+B[0][0]*c
  dp2 = -A[0][1]*s+B[0][1]*c
  dq1 = -A[1][0]*s+B[1][0]*c
  dq2 = -A[1][1]*s+B[1][1]*c

  # xyw[:,0] = p1*t+q1*(1-t), p2*t+q2*(1-t)
  # xyw[:,1] = p2*t+q2*(1-t)
  # xyw[:,2] = np.abs((p1-q1)*(dp2*t+dq2*(1-t))-\
  # (p2-q2)*(dp1*t+dq1*(1-t)))*w1*w2)

  return  np.transpose((p1*t+q1*(1-t), p2*t+q2*(1-t), np.abs((p1-q1)*\
   (dp2*t+dq2*(1-t))-(p2-q2)*(dp1*t+dq1*(1-t)))*w1*w2))
```

Points2distances

```
def points2distances(p):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  # Create euclidean distance matrix from point matrix.

  points = np.array(p)

  # All inner products between points.
  distances = points @ np.transpose(points)
```

```
  # Vector of squares of norms of points.
  lsq = np.diag(distances)

  a = np.array([[i]*len(points) for i in lsq])

  # distances = np.sqrt(a+np.transpose(a)-2*distances)
  return np.sqrt(a+np.transpose(a)-2*distances)
```

Cubature_rules_1D

```
def cubature_rules_1D(n,cubature_type):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  # SEE WALDVOGEL PAPER. ADDED NODES

  # Weights of the Fejer2, Clenshaw-Curtis and Fejer1 quadrature by DFTs
  # n>1. Nodes: x_k = cos(k*pi/n)

  N = np.array(range(1,n,2))
  l = len(N)
  K = np.array(range(0,n-l))

  match cubature_type:
    case 1:       # FEJER 1
      v0 = np.append([2*np.exp(1j*np.pi*K/n)/(1-4*K**2)],[0]*(l+1))
      v1 = v0[:-1]+np.conjugate(v0[:0:-1])
      weights = np.real(np.fft.ifft(v1))  # Imaginary residual
      k = np.arange(.5 , n+.5)
      nodes = np.cos(k*np.pi/n)

    case 2:       # FEJER 2
      v0 = np.append(2/N/(N-2), [1/N[-1]]+ [0]*(n-l))
      v2 = -v0[:-1]-v0[:0:-1]
      weights = np.append(np.fft.ifft(v2),0)
      k = np.array(range(0,n+1))
      nodes = np.cos(k*np.pi/n)

    case 3:       # CLENSHAW CURTIS
      g0 = -np.ones(n)
      g0[l] = g0[l]+n
      g0[n-l] = g0[n-l]+n
```

```
      g=g0/(n**2-1+n%2)
      v0 = np.append(2/N/(N-2), [1/N[-1]]+ [0]*(n-1))
      v2 = -v0[:-1]-v0[:0:-1]
      wcc = np.real(np.fft.ifft(v2+g))  # Imaginary residual
      weights = np.append(wcc, wcc[0])
      k = np.array(range(0,n+1))
      nodes = np.cos(k*np.pi/n)

    case 4:       # GAUSS LEGENDRE
      beta = .5/np.sqrt(1-1/((2*np.array(range(1,n+1)))**2))
      T = np.diag(beta,1)+np.diag(beta,-1)
      nodes,V=np.linalg.eigh(T)
      index = np.argsort(nodes)
      nodes = np.sort(nodes)
      weights = 2*V[0]**2


  return nodes, weights
```

auto_rotation

```
def auto_rotation(p, V1, V2):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  # AUTOMATIC ROTATION OF A CONVEX POLYGON SO THAT "GAUSSIAN POINTS",
  # AS IN THE PAPER THEY ARE ALL CONTAINED IN THE CONVEX POLYGON.
  # SEE THE PAPER FOR DETAILS.

  polygon_bd = np.array(p)
  vertex_1 = np.array(V1)
  vertex_2 = np.array(V2)

  # FIND DIRECTION AND ROTATION ANGLE.
  if len(vertex_1) == 0:
    # COMPUTING ALL THE DISTANCES BETWEEN POINTS.
    # A LITTLE TIME CONSUMING AS PROCEDURE.
    distances = points2distances(polygon_bd)
    max_distances = distances.max(1)
    max_distance = distances.max()
    max_row_comp = np.argmax(max_distances)
    max_col_comp = []
```

```
    for i in distances:
      max_col_comp.append(np.argmax(i))
    vertex_1 = np.array(polygon_bd[max_col_comp[max_row_comp]])
    vertex_2 = np.array(polygon_bd[max_row_comp])
    direction_axis = (vertex_2-vertex_1)/max_distance
  else:
    direction_axis = (vertex_2-vertex_1)/np.linalg.norm(vertex_2-vertex_1)

  rot_angle_x = np.arccos(direction_axis[0])
  rot_angle_y = np.arccos(direction_axis[1])

  if rot_angle_y <= np.pi/2:
    if rot_angle_x <= np.pi/2:
      rot_angle = -rot_angle_y
    else:
      rot_angle = rot_angle_y
  else:
    if rot_angle_x <= np.pi/2:
      rot_angle = np.pi-rot_angle_y
    else:
      rot_angle = rot_angle_y

  # CLOCKWISE ROTATION.
  rot_matrix = np.array([[np.cos(rot_angle), np.sin(rot_angle)], \
   [-np.sin(rot_angle), np.cos(rot_angle)]])
  number_sides = len(polygon_bd[:,0])-1
  polygon_bd_rot = (rot_matrix @ polygon_bd.T).T
  axis_abscissa = rot_matrix @ vertex_1.T

  return polygon_bd_rot, rot_matrix, rot_angle, axis_abscissa, \
  vertex_1, vertex_2
```

Polygauss

```
def polygauss(N, p, P1=[], Q1=[], rotation = 1):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  #-----------------------------------------------------------------------
  # REFERENCE PAPER:
  # [1] A. SOMMARIVA and M. VIANELLO
```

```
# "Gauss-like and triangulation-free cubature over polygons".
#
# INPUT:
#
# N     : DEGREE OF THE 1 DIMENSIONAL GAUSS-LEGENDRE RULE.
#
# polygon_sides: IF THE POLYGON HAS "L" SIDES, "boundary.pts" IS A
#         VARIABLE CONTAINING ITS VERTICES, ORDERED COUNTERCLOCKWISE.
#         AS LAST ROW MUST HAVE THE COMPONENTS OF THE FIRST VERTEX.
#         IN OTHER WORDS, THE FIRST ROW AND LAST ROW ARE EQUAL.
#         "polygon_sides" IS A "L+1 x 2" MATRIX.
#
#            --------- NOT MANDATORY VARIABLES ---------
#
# rotation: 0: NO ROTATION.
#           1: AUTOMATIC.
#           2: PREFERRED DIRECTION ROTATION BY P, Q.
#
# P, Q: DIRECTION THAT FIXES THE ROTATION.
#
# OUTPUT:
#
# xyw     : THE GAUSS LIKE FORMULA PRODUCES THE NODES (xyw(:,1),xyw(:,2))
#           AND THE WEIGHTS xyw(:,3) OF A CUBATURE RULE ON THE POLYGON.
#
#-------------------------------------------------------------------------
# EXAMPLE 1 (NO ROTATION.)
#--------------------------
#
# >> xyw=polygauss_2013(2,[0 0; 1 0; 1 1; 0 1; 0 0],0)
#
# xyw =
#
#     0.2113    0.2113    0.2500
#     0.2113    0.7887    0.2500
#     0.7887    0.2113    0.2500
#     0.7887    0.7887    0.2500
#
# >>
#
#-------------------------------------------------------------------------
# EXAMPLE 2 (AUTO ROTATION.)
#----------------------------
#
# >> xyw=polygauss_2013(2,[0 0; 1 0; 1 1; 0 1; 0 0])
#
```

```
# xyw =
#
#      0.0683    0.0444    0.0078
#      0.3028    0.1972    0.0556
#      0.5374    0.3499    0.0616
#      0.6501    0.4626    0.0616
#      0.8028    0.6972    0.0556
#      0.9556    0.9317    0.0078
#      0.9317    0.9556    0.0078
#      0.6972    0.8028    0.0556
#      0.4626    0.6501    0.0616
#      0.3499    0.5374    0.0616
#      0.1972    0.3028    0.0556
#      0.0444    0.0683    0.0078
#      0.1008    0.0119    0.0078
#      0.4472    0.0528    0.0556
#      0.7935    0.0938    0.0616
#      0.9062    0.2065    0.0616
#      0.9472    0.5528    0.0556
#      0.9881    0.8992    0.0078
#      0.8992    0.9881    0.0078
#      0.5528    0.9472    0.0556
#      0.2065    0.9062    0.0616
#      0.0938    0.7935    0.0616
#      0.0528    0.4472    0.0556
#      0.0119    0.1008    0.0078
#
# >>

#--------------------------------------------------------------------------
# Copyright (C) 2007-2013 Marco Vianello and Alvise Sommariva
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
#
```

```
# Authors:
# Marco Vianello    <marcov@euler.math.unipd.it>
# Alvise Sommariva  <alvise@euler.math.unipd.it>
# Date: April 30, 2013.
#--------------------------------------------------------------------------

polygon_sides = np.array(p)
P = np.array(P1)
Q = np.array(Q1)
x_bd = polygon_sides[:,0]
y_bd = polygon_sides[:,1]

# "MINIMUM" RECTANGLE CONTAINING POLYGON.
x_min = min(x_bd)
x_max=max(x_bd)
y_min = min(y_bd)
y_max=max(y_bd);

cubature_type=4

# POLYGON ROTATION (IF NECESSARY).
match rotation:
  case 0:
    #print('\n \t [ROTATION]: NO.')
    rot_matrix = np.eye(2)
    axis_abscissa = np.array([x_min, y_max])-np.array([x_min, y_min])
  case 1:
    #print('\n \t [ROTATION]: AUTOMATIC')
    polygon_sides, rot_matrix, rot_angle, axis_abscissa, P, Q = \
    auto_rotation(polygon_sides,[],[])
  case 2:
    #print('\n \t [ROTATION]: PREFERRED DIRECTION');
    nrm_vect = np.linalg.norm(Q-P)
    if nrm_vect > 0:
      direction_axis = (Q-P)/nrm_vect
      polygon_sides,rot_matrix,rot_angle,axis_abscissa,P,Q = \
      auto_rotation(polygon_sides,P,Q)
    else:
      #print('\n \t [WARNING]: THE DIRECTION VECTOR IS NULL. ')
      #print('USING AUTOMATIC ROTATION.')
      polygon_sides,rot_matrix,rot_angle,axis_abscissa,P,Q = \
      auto_rotation(polygon_sides,P,Q)


# COMPUTE NODES AND WEIGHTS OF 1D GAUSS-LEGENDRE RULE.
# TAKEN FROM TREFETHEN PAPER "Is ... Clenshaw-Curtis?".
```

```
# DEGREE "N" (ORDER GAUSS PRIMITIVE)
s_N, w_N = cubature_rules_1D((N-1),cubature_type)
N_length = len(s_N)

# DEGREE "M" (ORDER GAUSS INTEGRATION)
M = N+1
s_M, w_M = cubature_rules_1D((M-1),cubature_type)

# L: NUMBER OF SIDES OF THE POLYGON.
L = len(polygon_sides[:,1])-1

#a=0.5;
a = axis_abscissa[0]

# COMPUTE 2D NODES (nodes_x,nodes_y) AND WEIGHTS "weights".
nodes_x=[]
nodes_y=[]
weights=[]

for i in range(L):
   x1=polygon_sides[i][0]
   x2=polygon_sides[i+1][0]
   y1=polygon_sides[i][1]
   y2=polygon_sides[i+1][1]
   if not (x1 == a and x2 == a):
    if y2-y1 != 0:
      if x2-x1 != 0:
        s_M_loc = s_M
        w_M_loc = w_M
      else:
        s_M_loc = s_N
        w_M_loc = w_N

      M_length = len(s_M_loc)
      half_pt_x = (x1+x2)/2
      half_pt_y = (y1+y2)/2
      half_length_x = (x2-x1)/2
      half_length_y = (y2-y1)/2

      #  GAUSSIAN POINTS ON THE SIDE.
      x_gauss_side = half_pt_x+half_length_x*s_M_loc; #SIZE: (M_loc,1)
      y_gauss_side = half_pt_y+half_length_y*s_M_loc; #SIZE: (M_loc,1)

      scaling_fact_plus = (x_gauss_side+a)/2  #SIZE: (M_loc,1)
      scaling_fact_minus = (x_gauss_side-a)/2  #SIZE: (M_loc,1)
```

```
        # SIZE:(M_loc,1)
        local_weights = (half_length_y*scaling_fact_minus)*w_M_loc
        # SIZE: (M_loc,N)
        term_1 = np.tile(np.transpose([scaling_fact_plus]),N_length)
        # SIZE: (M_loc,N)
        term_2 = np.tile(np.transpose([scaling_fact_minus]),N_length)
        rep_s_N = np.tile(s_N,(M_length,1))


        # x, y ARE STORED IN MATRICES. A COUPLE WITH THE SAME INDEX
        # IS A POINT, i.e. "P_i=(x(k),y(k))" FOR SOME "k".
        x= term_1+term_2*rep_s_N
        number_rows=len(x)
        number_cols=len(x[0])
        x = x.flatten('F')
        y = np.tile(y_gauss_side,N_length)
        # THE INVERSE OF A ROTATION MATRIX IS ITS TRANSPOSE.
        rot_gauss_pts = rot_matrix.T @ np.array([x,y])


        # GAUSS POINTS IN THE ORIGINAL SYSTEM.
        x_rot = rot_gauss_pts[0,:]
        y_rot = rot_gauss_pts[1,:]

        x_rot = np.reshape(x_rot,(number_cols,number_rows)).T
        y_rot = np.reshape(y_rot,(number_cols,number_rows)).T

        weights = np.append(weights, local_weights)
        for i in x_rot:
          nodes_x.append(i)
        for j in y_rot:
          nodes_y.append(j)

    #nodes_x = np.ravel(np.array(nodes_x),'F')
    #nodes_y = np.ravel(np.array(nodes_y),'F')
    #weights = np.ravel(np.outer(weights, w_N),'F')

    #return nodes_x, nodes_y, weights

    return np.transpose((np.ravel(np.array(nodes_x),'F'), \
                         np.ravel(np.array(nodes_y),'F'), \
                         np.ravel(np.outer(weights, w_N),'F')))
```

Circtrap

```
def circtrap(n,a,b,c,d,cc,r):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  # by E. Artioli, A. Sommariva, M. Vianello
  # April 2018

  # n: polynomial exactness degree
  # a,b: circular arc extrema coords 1 x 2
  # c,d: base segment extrema coords 1 x 2, ac and bd are the sides
  # cc: circle center coords 1 x 2
  # r: circle radius

  # xyw: 3-column array xyw(:,1:2) nodes, xyw(:,3) weights

  A = np.array(a)
  B = np.array(b)
  C = np.array(c)
  D = np.array(d)
  CC = np.array(cc)

  Z = (A[0]-CC[0])+1j*(A[1]-CC[1])
  W = (B[0]-CC[0])+1j*(B[1]-CC[1])
  az = np.angle(Z)
  aw = np.angle(W)

  if az<=aw:
    if aw-az<=np.pi:
      alpha = az
      beta = aw
      U = C
      V = D
    else:
      alpha = aw
      beta = az+2*np.pi
      U = D
      V = C

  if az>aw:
    if az-aw<=np.pi:
      alpha = aw
      beta = az
      U = D
```

```
      V = C
    else:
      alpha = az
      beta = aw+2*np.pi
      U = C
      V = D

  om = (beta-alpha)/2
  g = (beta+alpha)/2
  s=2*np.sin(om)
  A1 = [[r*np.cos(g), r*np.sin(g)], [0, 0]]
  B1 = [[-r*np.sin(g), r*np.cos(g)], [(V[0]-U[0])/s , (V[1]-U[1])/s ]]
  C1 = [[CC[0], CC[1]], [(V[0]+U[0])/2, (V[1]+U[1])/2]]

  return gqellblend(n,A1,B1,C1,-om,om)
```

Polygcirc

```
def polygcirc(n,V,A1,B1,center,r,conv,pos=1):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  #---------------------------------------------------------------------------
  # OBJECT:

  # Computation of a basic and a compressed positive cubature formula
  # on a polygonal element with a circular side, namely the set
  # union (convex arc) or difference (concave arc) of a convex
  # polygonal element with a circular segment.
  #---------------------------------------------------------------------------
  # INPUT:

  # n: polynomial degree of exactness
  # A1,B1: extrema of the circular arc
  # center,r: circular arc center and radius
  # V: polygon vertices (2-column array of coords)
  #    note that to "v" are added by default the arc extrema, in
  #    couterclockwise order
  # conv: conv=1 for a convex arc, conv=0 for a concave arc
  #
  # WARNING: the figure vertices are a,v(1,:),...,v(end,:),b and MUST BE
  # in COUNTERCLOCKWISE order
  # (the arc ba is clockwise on the circle if concave and counterclockwise
```

```
# if convex)
#---------------------------------------------------------------------------
# OUTPUT:

# xyw: 3-column array of cubature nodes and positive weights
# xywc: 3-column array of compressed cubature nodes and positive weights
#---------------------------------------------------------------------------
# AUTHORS:

# Authors: E. Artioli, A. Sommariva and M. Vianello
# Written: April 24, 2018
# Revised: December 02, 2021
#---------------------------------------------------------------------------

# auxiliary function used to stack columns on xyw more easily

cc = np.array(center)
v = np.array(V)
a = np.array(A1)
b = np.array(B1)
L = []
P = []
subs = []

def stackxyw(xyw, nw):
  if len(xyw) == 0:
    return nw
  else:
    return np.vstack((xyw, nw))

# Convex scenario
if conv==1:
  P = np.vstack([a,v,b,a])
  xyw = np.vstack((polygauss(n,P), circtrap(n,b,a,b,a,cc,r)))
  L += [xyw[:,1].size]
  A = b[:]
  B = a[:]
  C = b[:]
  D = a[:]
  subs += [A,B,C,D]

#Concave scenario
else: #conv = 0
  k = 0; l = 0; t = 0; e=[]; z=[]; u=[]; eta=[]; xyw=[]
  aa=a-cc; bb=b-cc
  angleab = np.arccos((aa@bb)/(np.linalg.norm(aa)*np.linalg.norm(bb)))
```

```
anglea = np.angle(aa[0]+1j*aa[1])
angleb = np.angle(bb[0]+1j*bb[1])

if angleb<=np.pi:
  clockba = anglea>=angleb and anglea<angleb+np.pi
else:
  clockba = anglea>angleb-np.pi and anglea<=angleb
if len(v[:,0])==2:
  v = np.array([v[0], (v[0]+v[1])/2, v[1]])

vv = v-np.matlib.repmat(cc, len(v[:,0]), 1)

for i in range(len(v[:,1])):
  angle1 = np.arccos((vv[i]@aa)/(np.linalg.norm(aa)*np.linalg.norm(vv[i])))
  angle2 = np.arccos((vv[i]@bb)/(np.linalg.norm(bb)*np.linalg.norm(vv[i])))

  if angle1<angle2 and angle2>angleab and clockba==0:
    e += [v[i]]
    k+=1
  if clockba==1 or (angle1<=angleab and angle2<=angleab):
    z += [v[i]]
    zz = z[l]-cc
    theta = np.angle(zz[0]+1j*zz[1])
    u += [cc+r*np.array([np.cos(theta), np.sin(theta)])]
    l+=1
  if angle2<angle1 and angle1>angleab and clockba==0:
    eta += [v[i]]
    t+=1

e = np.array(e); z = np.array(z); u = np.array(u); eta  = np.array(eta)

if k>1:
  P1 = np.vstack((a, e, a))
  xyw = polygauss(n, P1)
  L += [xyw[:,0].size]

if k==0 and l>=1:
  if np.linalg.norm(u[0]-a)>10**(-14):
    nw = circtrap(n,a,u[0], z[0], z[0], cc, r)
    L += [nw[:,0].size]
    xyw = nw
    A = a; B = u[0,:]
    C = z[0]; D = z[0]
    subs += [A,B,C,D]

if k>=1 and l>=1:
```

```
  nw = circtrap(n,a,u[0],e[k-1],z[0],cc,r)
  L += [nw[:,0].size]
  xyw = stackxyw(xyw, nw)
  A = a; B = u[0,:]
  C = e[k-1,:]; D = z[0,:]
  subs += [A,B,C,D]

for j in range(1,l):
  nw=circtrap(n,u[j-1],u[j],z[j-1],z[j],cc,r)
  L += [nw[:,0].size]
  xyw = stackxyw(xyw, nw)
  A = u[j-1]; B = u[j]
  C = z[j-1]; D = z[j]
  subs += [A,B,C,D]

if t == 0 and l>=1:
  if np.linalg.norm(u[l-1]-b)>10**(-14):
    nw = circtrap(n,u[l-1],b,z[l-1,],z[l-1],cc,r)
    L += [nw[:,0].size]
    xyw = stackxyw(xyw, nw)
    A = u[l-1]; B = b
    C = z[l-1]; D = z[l-1]
    subs += [A,B,C,D]

if t>=1 and l>=1:
  nw = circtrap(n,u[l-1,:],b,z[l-1,:],eta[0,:],cc,r)
  L += [nw[:,0].size]
  xyw = stackxyw(xyw, nw)
  A = u[l-1]; B = b
  C = z[l-1]; D = eta[0]
  subs += [A,B,C,D]

if t>1:
  P2=np.vstack((b,eta,b))
  nw=polygauss(n,P2)
  L += [nw[:,1].size]
  xyw = stackxyw(xyw, nw)

if l==0:
  if k>=1 and t>=1:
    nw = circtrap(n,a,b,e[k-1],eta[0],cc,r)
    L += [nw[:,0].size]
    A = a; B = b
    C = e[k-1]; D = eta[0]
    subs += [A,B,C,D]
```

```
      if k==0 and t>=1:
        nw = circtrap(n,a,b,eta[0],eta[0],cc,r)
        L += [nw[:,0].size]
        A = a; B = b
        C = eta[0]; D = eta[0]
        subs += [A,B,C,D]

      if k>=1 and t==0:
        nw = circtrap(n,a,b,e[k-1],e[k-1],cc,r)
        L += [nw[:,0].size]
        A = a; B = b
        C = e[k-1,:]; D = e[k-1,:]
        subs += [A,B,C,D]
      xyw = stackxyw(xyw, nw)


  subs = np.array(subs)
  pts, w, momerr = comprexcub(n,[xyw[:,0], xyw[:,1]],xyw[:,2],pos)
  xywc = np.transpose([pts[:,0], pts[:,1], w])

  return  xyw, xywc, P, L, subs
```

demo_polygcirc

```
def demo_polygcirc(test_type=2,n=4,pos=2):

  # Code author: Giovanni Traversin
  # Release date: 05 Sept 2025

  # polynomial exactness degree
  # compression type: pos=0: lsqnonneg, pos=1: LHDM.
  # note: pos=0: may have negative weights
  # tests to determine median cputimes

  Ntests=10;

  match test_type:
    case 1:
      # TEST 1: concave arc
      cc=np.array([0, 0]); r=0.25   # circle defs; center "cc" and radius "r"
      a=np.array([0.25, 0])                       # first point of the circle
      b=np.array([0, 0.25])                       # last point of the circle
      v=np.array([[0.5, 0], [0.5, 0.5], [0, 0.5]]) # polygon vertices
      conv=0                                      # 0: concave arc
```

```
    case 2:
      # TEST 2: convex arc
      cc=np.array([.25, .25]); r=.25 # circle defs; center "cc" and radius "r"
      a=np.array([0.25, 0])                          # first point of the circle
      b=np.array([0, 0.25])                          # last point of the circle
      # polygon vertices
      v=np.array([[.4, .05], [.5, .25], [.45, .45],\
                  [.3, .5], [.1, .45]])
      conv=1                                         # 1: convex arc

    case 3:
      # TEST 3: concave arc
      cc=np.array([0, 0]); r=0.25   # circle defs; center "cc" and radius "r"
      a=np.array([0.25, 0])                          # first point of the circle
      b=np.array([0, 0.25])                          # last point of the circle
      v=np.array([[0.25, 0.2], [0.2, 0.25]])         # polygon vertices
      conv=0                                         # 0: concave arc

    case 4:
      # TEST 4 convex arc
      cc=np.array([0, 0]); r=0.25   # circle defs; center "cc" and radius "r"
      a=np.array([0.25, 0])                          # first point of the circle
      b=np.array([0, 0.25])                          # last point of the circle
      # polygon vertices
      v=np.array([[.4, .05], [.5, .25], [.45, .45], [.3, .5], [.1, .45]])
      conv=0                                         # 0: concave arc

# ..................... Compute cubature rules  .....................

  cpusC = []
  for k in range(Ntests):
    t = time.time()
    xyw = polygcirc(n,v,a,b,cc,r,conv,pos,False)
    cpusC += [time.time() - t]
  cpusM = [statistics.median(cpusC)]

  cpusC = []
  for k in range(Ntests):
    t = time.time()
    xyw,xywc,P,L,subs, momerr = polygcirc(n,v,a,b,cc,r,conv,pos)
    cpusC += [time.time() - t]
  cpusM += [statistics.median(cpusC)]

# ......................... Statistics  ............................
```

```
  print('\n \t EXAMPLE          : %d' %test_type)
  print('\t ALG. DEG. PREC. : %d' %n)

  match pos:
    case 1:
      print('\t COMPRESSION     : LHDM')
    case _:
      print('\t COMPRESSION     : lsqnonneg')

  print('\t NODES FULL RULE : %d' %xyw[:,0].size)
  print('\t NODES COMP RULE : %d' %xywc[:,0].size)
  w=xyw[:,2]; iwneg=np.where(w < 0)[0]; L=len(iwneg)
  print('\t NEG WEIGHTS FULL: %d' %L)
  wc=xywc[:,2]; iwnegc=np.where(wc < 0)[0]; Lc=len(iwnegc);
  print('\t FULL RULE CPU   : %1.3e' %cpusM[0])
  print('\t FULL + COMP CPU : %1.3e' %cpusM[1])
  print('\t MOMENTS\' ERROR  : %1.3e' %momerr)

#..................... Nodes to copy in Matlab ......................

  # Cubature Full Nodes
#  print('[')
#  for i in xyw:
#    print('%.15f  %.15f %.15f;'  %(i[0], i[1], i[2]))
#  print(']')

  # Cubature Comprex Nodes
#  print('[')
#  for i in xywc:
#    print('%.15f  %.15f %.15f;'  %(i[0], i[1], i[2]))
#  print(']')

#  ......................... Plots  ................................

  # ... plot parameters ...
  size_sides=2
  size_pt=4
  size_cmp=8
  size_in_sides=1

  # ... plot figure ...
  fig, axs = plt.subplots(figsize=(10, 10))
  vertices=np.vstack([a, v, b])
  tha = np.arctan2(a[1]-cc[1],a[0]-cc[0])
  thb = np.arctan2(b[1]-cc[1],b[0]-cc[0])
  LL=int(subs[:,0].size/4)
```

```
# Plot circular quadrangles (divisive dotted lines)
for kk in range(1,LL+1):
  CC = cc
  Li = 4*(kk-1)+1
  A=subs[Li-1]
  B=subs[Li]
  C=subs[Li+1]
  D=subs[Li+2]
  axs.plot([A[0], C[0]],[A[1], C[1]],'k--')
  axs.plot([B[0], D[0]],[B[1], D[1]],'k--')

axs.plot(vertices[:,0], vertices[:,1], 'k-')  # edges
Lin = 1
L = [L]
P = np.array(P)
for ii in range(len(L)):
  Lfin=L[ii]+Lin-1
  xywr = xyw[Lin:Lfin]
  Lin = Lfin+1
  iid = (ii+1)%7

  match iid:
    case 1:
      axs.plot(xywr[:,1],xywr[:,2],'b+')
    case 2:
      axs.plot(xywr[:,1],xywr[:,2],'r+')
    case 3:
      axs.plot(xywr[:,1],xywr[:,2],'c+',markeredgecolor='c',\
               markerfacecolor='c',markersize=size_pt)
    case 4:
      axs.plot(xywr[:,1],xywr[:,2],'m+')
    case 5:
      axs.plot(xywr[:,1],xywr[:,2],'g+')
    case 6:
      axs.plot(xywr[:,1],xywr[:,2],'y+')
    case _:
      axs.plot(xywr[:,1],xywr[:,2],'k+')

if np.size(P,0)>0:
  axs.plot(P[:,0],P[:,1],'k--')
axs.plot(xyw[:,0],xyw[:,1],'go',markeredgecolor='k', markersize=10)  #dots

if conv == 1:
  if tha<thb:
    tha = tha+2*np.pi
```

```
    th = np.linspace(thb,tha,100)
  else:
    th = np.linspace(tha,thb,100)

  xP=cc[0]+r*np.cos(th)
  yP=cc[1]+r*np.sin(th)
  axs.plot(xP,yP,'k-')

#.....................Plots with sorted weights .....................

#  fig, wpl = plt.subplots(figsize=(10, 10))
#  wpl.plot(list(range(1,len(xyw[:,2])+1)), np.sort(xyw[:,2]),'o',\
#           color='red', markersize=7, markeredgecolor='midnightblue')
#
#  fig, wplc = plt.subplots(figsize=(10, 10))
#  wplc.plot(list(range(1,len(xywc[:,2])+1)), np.sort(xywc[:,2]),'o',\
#            color='red', markersize=7, markeredgecolor='midnightblue')

  return
```