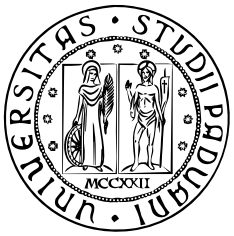
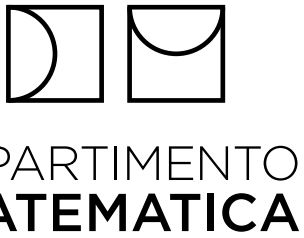


Università degli Studi di Padova



Università degli Studi di Padova
Dipartimento di Matematica "Tullio Levi-Civita"
Corso di laurea in Matematica

Quadratura subperiodica in Python

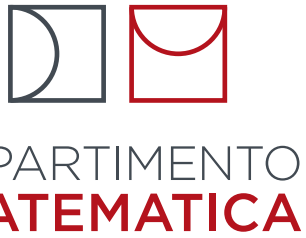
Relatore:
Prof. Alvise Sommariva

Correlatore:
Dott.ssa Laura Rinaldi

Candidato:
Mattia Storgato
Matricola 2073661

Anno Accademico 2024/2025 - 19.09.2025

Università degli Studi di Padova



Università degli Studi di Padova
Dipartimento di Matematica "Tullio Levi-Civita"
Corso di Laurea in Matematica

Quadratura subperiodica in Python

Relatore:
Prof. Alvise Sommariva

Correlatore:
Dott.ssa Laura Rinaldi

Candidato:
Mattia Storgato
Matricola 2073661

Anno Accademico 2024/2025 - 19.09.2025

Alla mia famiglia,
sostegno ed esempio di una vita.

Ai miei amici,
la miglior distrazione dalle serietà.

A me,
ai miei difetti
e alla mia voglia di migliorarmi.

Indice

Introduzione

L'obiettivo di questa tesi è calcolare numericamente l'integrale di un polinomio trigonometrico in un subintervallo di $[0, 2\pi]$ attraverso le formule di quadratura Gaussiane subperiodiche proposte da G. Da Fies e M. Vianello in *Trigonometric Gaussian quadrature on subintervals of the period*, implementando le routines in Python invece che in Matlab.

Di conseguenza, fissato il grado n, si defineranno delle routines Python tali da calcolare nodi $\{x_k\}_{k=1,\dots,n} \subset [\alpha, \beta]$ e rispettivi pesi $\{w_k\}_{k=1,\dots,n} \subset \mathbb{R}^+$ che definiscono formule di quadratura per cui

$$\int_\alpha^\beta f(x)dx = \sum_{k=1}^{n} w_k f(x_k)$$

qualora f appartenga allo spazio $\mathbb{T}_n$ dei polinomi trigonometrici di grado n

$$\mathbb{T}_n = \text{span}\{1, \cos(k\theta), \sin(k\theta) | 1 \leq k \leq n\}.$$

Nel primo capitolo definiremo la quadratura subperiodica, mostrando come utilizzare una formula di quadratura algebrica per calcolare nodi di integrazione di una funzione trigonometrica. Di seguito introdurremo le regioni bivariate generate da *linear blending* di archi facendone vari esempi.

Nel secondo capitolo vedremo in dettaglio l'algoritmo alla base di `trigauss` spiegandone i vari passaggi e allegando i programmi utilizzati.

Nel terzo capitolo confronteremo i risultati nei due linguaggi tanto dal punto di vista numerico che in termini di velocitá di calcolo, sia nel caso di integrazione su intervalli che su domini bivariati di tipo *linear blending*. In particolare mostreremo che i nodi e i pesi sono numericamente uguali, faremo dei tests sul calcolo di una particolare integranda, mostrando infine che i tempi richiesti in Matlab e Python sono paragonabili.

Capitolo 1

Quadratura subperiodica

L'idea alla base delle formule di quadratura univariata consiste nell'approssimare un integrale, definito su un intervallo (α, β), attraverso la somma pesata di valutazioni di una funzione in un numero finito di punti $\{x_k\}$ detti *nodi*, ovvero

$$\int_\alpha^\beta f(x)dx \approx \sum_{k=1}^{N_n} w_k f(x_k).$$

In questo lavoro considereremo il caso di formule esatte per i polinomi trigonometrici $\mathbb{T}_n([-\omega, \omega])$ di grado n, con $\omega = \frac{\beta-\alpha}{2}$, ovvero del tipo

$$p_n(x) = \sum_{k=0}^{n} a_k \cos(kx) + \sum_{k=1}^{n} b_k \sin(kx).$$

concentrandoci sul caso in cui $0 < \frac{\beta-\alpha}{2} \leq \pi$, noto in letteratura come quadratura trigonometrica *subperiodica* e quindi in generale valida anche in sottointervalli di $[0, 2\pi)$.

1.1 Risultati preliminari

Per prima cosa introduciamo dei risultati di base necessari per enunciare la formula di quadratura trigonometrica subperiodica.

Il primo passo sarà quello di ricordare la formula gaussiana algebrica per la quadratura, concentrandoci nel determinare i nodi e pesi della formula che saranno indispensabili per la Proposizione 1.1.3 che legheranno la quadratura gaussiana subperiodica a quella algebrica.

Teorema 1.1.1 (Esistenza e unicità delle formule Gaussiane)**.** *Per ogni $n \geq 1$ esistono e sono unici nodi $x_1, ..., x_n$ e pesi $w_1, ..., w_n$ per cui il grado di precisione sia almeno $2n-1$ ovvero*

$$I_w(p_{2n-1}) = \int_a^b p_{2n-1}(x)w(x)dx = \sum_{k=1}^{n} w_k p_{2k-1}(x_k) \tag{1.1}$$

per ogni $p_{2n-1} \in \mathbb{P}_{2n-1}$.

Questo tipo di quadratura, detta *gaussiana* è molto utilizzata per il calcolo di integrali definiti esattamente per via del suo alto grado di precisione $2n-1$ in relazione al numero di nodi n.

Questa formula Gaussiana adopera come nodi $\{\xi_i\}_{i\geq 0}$ gli zeri del polinomio ortogonale $p_n(x)$, $n \in \mathbb{N}$, della famiglia $\{p_k(x)\}_{k\geq 0}$ derivanti dalla funzione peso 1.6 , ovvero che rispettano il vincolo

$$\int_{-1}^{1} p_j(x)p_i(x)w(x)dx = 0 \quad \text{per } j \neq i. \tag{1.2}$$

Il generico polinomio ortogonale p_n é valutabile in un punto arbitrario x, qualora lo siano $p_0, p_1, \ldots, p_{n-1}$, come esposto dal seguente teorema.

Teorema 1.1.2 (Ricorsione a 3 termini di Christoffel)**.** *Sia* $\{\phi_k\}_{k=0,1,\ldots,n}$ *una famiglia triangolare di polinomi monici in* (a, b) *e ortogonali rispetto alla funzione peso* w.
Si supponga $\phi_{-1}(x) = 0$, $\phi_0(x) = 1$, *allora per* $n \geq 0$

$$\phi_{n+1}(x) = (x - \alpha_n)\phi_n - \beta_n\phi_{n-1} \tag{1.3}$$

dove si ha

$$\alpha_n = \frac{(x\phi_n, \phi_n)_{2,w}}{(\phi_n, \phi_n)_{2,w}}, \quad \beta_n = \frac{(\phi_n, \phi_n)_{2,w}}{(\phi_{n-1}, \phi_{n-1})_{2,w}} \tag{1.4}$$

con

$$(\phi_i, \phi_j)_{2,w} = \int_a^b \phi_i(x)\phi_j(x)w(x)dx, \tag{1.5}$$

$\alpha_n \in \mathbb{R}$ *e* $\beta_n \in \mathbb{R}^+$.

Il Teorema è valido per funzioni peso classiche, quali quelle di tipo *Jacobi*, *Laguerre*, *Hermite* e funzioni peso atipiche come quella che vedremo nella prossima Proposizione 1.1.3.

Nel caso ció non sia possibile, i coefficienti verranno calcolati numericamente attraverso l'algoritmo di Chebyshev modificato che vedremo nel dettaglio in 2.0.1. L'obiettivo è quello di calcolare i coefficienti in modo ricorsivo, partendo quindi da α_0 e β_0.

Per quanto riguarda il calcolo dei nodi e dei pesi di quadratura, per una funzione peso generica w, un classico approccio é quello dell'algoritmo di Golub-Welsch [**GW**], [**G1**], [**G2**].

In questo, si definisce la matrice di Jacobi (relativamente a w)

$$J = \begin{pmatrix} -\alpha_0 & \sqrt{\beta_1} & & \\ \sqrt{\beta_1} & -\alpha_1 & \ddots & \\ & \ddots & \ddots & \sqrt{\beta_n} \\ & & \sqrt{\beta_n} & -\alpha_n \end{pmatrix}$$

dove $\alpha_k \in \mathbb{R}$ e $\beta_k \in \mathbb{R}^+$ definiti come nel precedente Teorema 1.1.2.

Di seguito i nodi e pesi sono ricavabili tramite l'opportuno calcolo di autovalori e autovettori di J.

Per il calcolo di formule gaussiane per la quadratura trigonometrica subperiodica vale il seguente risultato.

Proposizione 1.1.3. *Siano $\{\xi_j, \lambda_j\}_{1\leq j\leq n+1}$ i nodi e i pesi positivi della formula di quadratura Gaussiana algerica per la funzione peso:*

$$w(x) = \frac{2\sin(\omega/2)}{\sqrt{1 - x^2\sin^2(\omega/2)}}, \quad x \in (-1, 1),\ \omega \in (0, \pi]. \tag{1.6}$$

Allora per $0 < \beta - \alpha \leq 2\pi$ vale la seguente formula di quadratura Gaussiana trigonometrica:

$$\int_\alpha^\beta f(\theta)d\theta = \sum_{j=1}^{n+1} \lambda_j f(\theta_j + \mu), \quad \forall f \in \mathbb{T}_n([\alpha, \beta]), \quad \mu = \frac{\alpha + \beta}{2}, \tag{1.7}$$

dove

$$\theta_j = 2\arcsin(\xi_j \sin(\omega/2)) \in (-\omega, \omega), \quad j = 1, 2, ..., n + 1, \quad \omega = \frac{\beta - \alpha}{2}. \tag{1.8}$$

Per quanto precedentemente detto, questa formula è esatta nello spazio $(2n+1)$-dimensionale dei polinomi trigonometrici $\mathbb{T}_n([-\omega, \omega])$ di grado al più n. Questo dà una validità aggiunta a questo lavoro che fornisce un'alternativa valida e più precisa alle classiche formule di quadratura algebriche.

La funzione peso in questione deriva naturalmente dalla trasformazione di variabile (1.8). Si vede infatti che derivando entrambi i lati dell'equazione ricaviamo

$$d\theta = \frac{2\sin(\omega/2)}{\sqrt{1 - x^2\sin^2(\omega/2)}}dx$$

e sostituendo nell'integrale da calcolare troviamo

$$\int_\alpha^\beta f(\theta)d\theta = \int_{-1}^{1} f(2\arcsin(x\sin(\omega/2)) + \mu) \cdot \frac{2\sin(\omega/2)}{\sqrt{1 - x^2\sin^2(\omega/2)}}dx.$$

Altro aspetto fondamentale è la positività dei pesi che contribuisce a dare stabilità all'algoritmo e quindi permette una maggiore precisione del calcolo numerico.

Notiamo il caso speciale in cui $\omega = \pi$; sostituendo ω nella formula del peso (1.6) si nota che quest'ultimo è quello di Chebyshev, ovvero

$$w_C(x) = \frac{1}{\sqrt{1 - x^2}}. \tag{1.9}$$

La principale difficoltá risulta quindi la valutazione dei nodi e dei pesi rispetto alla funzione peso (1.6). Se intendiamo applicare l'algoritmo di Golub-Welsch dobbiamo

disporre dei corrispondenti coefficienti di ricorrenza $\{\alpha_k\}$ e $\{\beta_k\}$, in questo caso non noti analiticamente.

A tal proposito, come viene studiato in **[FV]**, definiti i polinomi di Chebyshev $T_n(x) = \cos(n \arccos(x))$,

$$w_j = \frac{1}{2n+1}\left(m_0 + 2\sum_{k=1}^{n} m_{2k} T_{2k}(\tau_j)\right), \quad j = 1, 2, ...2n+1$$

si calcolano i momenti modificati di Chebyshev rispetto alla funzione peso (1.6).

$$m_s = \int_{-1}^{1} \frac{T_s(x) 2\sin(\omega/2)}{\sqrt{1 - x^2 \sin^2(\omega/2)}} dx, \quad s = 0, 1, ... \tag{1.10}$$

A partire da questi, come descritto in **[G1]** e **[G2]** si possono calcolare convenientemente i coefficienti di ricorrenza $\{\alpha_k\}$ e $\{\beta_k\}$, mediante il cosiddetto *algoritmo di Chebyshev* e di seguito i nodi e i pesi di quadratura di una formula gaussiana rispetto alla funzione peso in questione.

Un aspetto che sarà fondamentale nell'implementazione sarà osservare che i momenti m_s con s dispari si annullano; sono infatti l'integrale di una funzione dispari calcolata in un intervallo simmetrico rispetto all'origine.

1.2 Linear Blending e relative formule di quadratura

Ora vediamo il passo successivo, che studieremo in questa sezione: troviamo una formula di quadratura Gaussiana prodotto, andremo infatti ad introdurre Ω, regione del piano generata tramite linear blending di archi, ed enunceremo una formula che ci permette di calcolare l'integrale di una funzione $f(x, y)$ su Ω.

Iniziamo definendo le curve trigonometriche piane P, Q che parametrizzate assumono la forma:

$$\begin{aligned} P(\theta) &= A_1 \cos(\theta) + B_1 \sin(\theta) + C_1, \\ Q(\theta) &= A_2 \cos(\theta) + B_2 \sin(\theta) + C_2, \end{aligned} \tag{1.11}$$

con $\theta \in (\alpha, \beta)$ e $0 \leq \alpha, \beta \leq 2\pi$. A_i, B_i, C_i sono dei vettori bidimensionali dai valori

$$A_i = (a_{i1}, a_{i2}), \quad B_i = (b_{i1}, b_{i2}) \quad , C_i = (c_{i1}, c_{i2}), \quad i = 1, 2.$$

Da qui definiamo Ω, la combinazione convessa dei due archi:

$$\Omega = \{(x, y) = U(t, \theta) = tP(\theta) + (1-t)Q(\theta),\ (t, \theta) \in [0, 1] \times [\alpha, \beta]\} \tag{1.12}$$

con la mappa U iniettiva in $(0, 1) \times (\alpha, \beta)$. Fatto questo, per enunciare la formula di quadratura dobbiamo prima osservare $JU(t, \theta)$, matrice Jacobiana di $U(t, \theta)$, e il suo comportamento. La prima caratteristica importante è che non cambia segno nel prodotto di intervalli, successivamente vediamo il risultato del calcolo del determinante:

$$|detJU(t, \theta)| = \pm(tu(\theta) + v(\theta)) \tag{1.13}$$

dove

$$u(\theta) = u_0 + u_1\cos(\theta) + u_2\sin(\theta),$$
$$v(\theta) = v_0 + v_1\cos(\theta) + v_2\sin(\theta) + v_3\cos(\theta)\sin(\theta) + v_4\sin^2(\theta)$$

con

$$u_0 = (a_{11} - a_{21})(b_{12} - b_{22}) + (a_{12} - a_{22})(b_{21} - b_{11}),$$
$$u_1 = (b_{12} - b_{22})(c_{11} - c_{21}) + (b_{21} - b_{11})(c_{12} - c_{22}),$$
$$u_2 = (a_{11} - a_{21})(c_{12} - c_{22}) + (a_{12} - a_{22})(c_{21} - c_{11}),$$

e

$$v_0 = b_{21}(a_{22} - a_{12}) + b_{22}(a_{11} - a_{21}),$$
$$v_1 = b_{21}(c_{22} - c_{12}) + b_{22}(c_{11} - c_{21}),$$
$$v_2 = a_{21}(c_{12} - c_{22}) + a_{22}(c_{21} - c_{11}),$$
$$v_3 = a_{12}a_{21} - a_{11}a_{22} + b_{11}b_{22} - b_{12}b_{21},$$
$$v_4 = a_{12}b_{21} - a_{11}b_{22} + a_{21}b_{12} - a_{22}b_{11}.$$

Una formula di quadratura come la abbiamo anticipata su una regione Ω combina la formula trigonometrica già vista nel Teorema 1.1.3 e la formula di Gauss-Legendre nella seguente proposizione:

Proposizione 1.2.1. *Si consideri il dominio*

$$\Omega = \{(x,y) = U(t,\theta),\ (t,\theta) \in [0,1]\times[\alpha,\beta],\ 0 < \beta - \alpha \le 2\pi\}, \tag{1.14}$$

dove la trasformazione $U(t,\theta)$ è iniettiva in $(t,\theta) \in (0,1)\times(\alpha,\beta)$. Allora vale la seguente formula gaussiana prodotto con $\frac{n^2}{2} + O(n)$ nodi:

$$\iint_\Omega f(x,y)\,dxdy = I_n(f) = \sum_{j=1}^{n+k+1}\sum_{i=1}^{\lceil\frac{n+h+1}{2}\rceil} W_{ij}\, f(x_{ij}, y_{ij}), \quad \forall f \in \mathbb{P}_n^2 \tag{1.15}$$

dove con $\mathbb{P}_n^2$ si denota lo spazio dei polinomi bivariati di grado totale n, con

$$h = \begin{cases} 0, & \text{se } u_i = 0, \\ 1, & \text{altrimenti,} \end{cases} \qquad k = \begin{cases} 0, & \text{se } u_1 = u_2 = 0 \text{ e } v_j = 0, \\ 1, & \text{se } v_3 = v_4 = 0 \text{ e almeno uno tra } u_1, u_2, v_1, v_2 \neq 0, \\ 2, & \text{se } v_3 \neq 0 \text{ o } v_4 \neq 0, \end{cases}$$

con $i = 0,1,2$ e $j = 1,...,4$, in (1.15) abbiamo

$$(x_{ij}, y_{ij}) = U(t_i^{GL}, \theta_j + \mu), \qquad W_{ij} = \left|\det JU\big(t_i^{GL}, \theta_j + \mu\big)\right| \omega_i^{GL}\,\lambda_j, \tag{1.16}$$

dove $\{(\theta_j+\mu, \lambda_j)\}$ sono i nodi e i pesi della formula Gaussiana trigonometrica (1.1.1) con grado di precisione $n+k$ su $[\alpha,\beta]$, e $\{(t_i^{GL}, \omega_i^{GL})\}$ i nodi e i pesi della formula di Gauss-Legendre con grado di precisione $n+h$ su $[0,1]$.

1.2.1 Settori ellittici ed esempi di blending

Vediamo ora in dettaglio il caso dei settori ellittici, ottenuti tramite blending lineare di archi di ellisse, che saranno oggetto di test nei capitoli successivi. Dalla Proposizione appena vista 1.2.1 si ricavano i nodi e i pesi per le formule di quadratura prodotto su regioni del tipo settore ellittico.

Ci sarà però una differenza tra pesi e nodi in base alla scelta di $P(\theta)$ e $Q(\theta)$, di seguito vediamo come il campionamento dei nodi, derivante dagli archi, vada ad influenzare i risultati finali. Vediamo un esempio pratico di un arco a segmento circolare corrispondente all'intervallo $[-\pi/3, \pi/3]$, con raggio 0.5. Elenchiamo ora le diverse parametrizzazioni di quest'ultimo:

- $P(\theta) = \big(R\cos(\theta),\, R\sin(\theta)\big)$ e $Q(\theta) = \big(R\cos(\theta),\, -R\sin(\theta)\big), \quad \theta \in [0, \frac{\pi}{3}]$;
- $P(\theta) = \big(R\cos(\beta),\, R\sin(\theta)\big)$ e $Q(\theta) = \big(R\cos(\theta),\, R\sin(\theta)\big), \quad \theta \in \big[-\frac{\pi}{3},\, \frac{\pi}{3}\big]$;
- $P(\theta) = (0.25,\, 0)$ e $Q(\theta) = \big(R\cos(\theta),\, R\sin(\theta)\big), \quad \theta \in \big[-\frac{\pi}{3},\, \frac{\pi}{3}\big]$.

La seguente figura 1.1 rappresenta come ognuna delle parametrizzazioni si traduce in termini di nodi di una formula di quadratura con grado di esattezza 8.

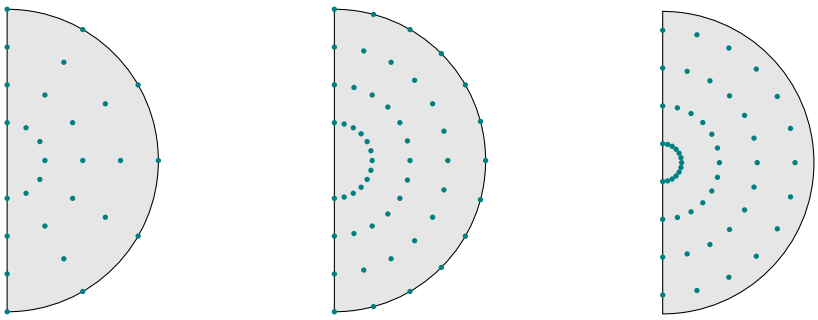


Figura 1.1: Le figure illustrano i rispettivi nodi delle formule di quadratura con grado di precisione $\delta = 8$ relativi alle tre parametrizzazioni.

Un altro esempio interessante é definito dal blending lineare tra l'origine e un arco di ellisse centrato in $(0, 0)$ in 2 intervalli limitati, il primo $[-\frac{\pi}{6}, \frac{\pi}{6}]$ e il secondo $[\frac{2}{9}\pi, \frac{7}{18}\pi]$

$$P(\theta) = (0, 0), \quad Q(\theta) = A_2\cos(\theta) + B_2\sin(\theta), \tag{1.17}$$

al variare di A_2 e di B_2.

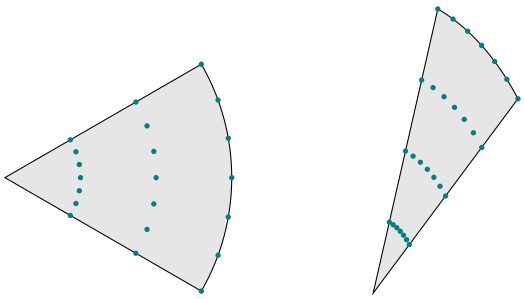


Figura 1.2: Le figure illustrano i rispettivi nodi delle formule di quadratura con grado di precisione $\delta = 8$ relativi a 2 esempi di settori sferici ed ellittici.

Con la stessa notazione della Proposizione 1.2.1, per un settore sferico, si ha $u_1 = u_2 = 0$ e $v_1 = v_2 = v_3 = v_4 = 0$ e quindi, $h = 1$ e $k = 0$. Da

$$\left|\det JU(t_i^{GL}, \theta_j + \mu)\right| = \left|t_i^{GL} u_0 + v_0\right|,$$

si ricava

$$\iint_S f(x,y)\,dx\,dy = I_n(f) = \sum_{j=1}^{n+1} \sum_{i=1}^{\lceil \frac{n+2}{2} \rceil} \left|t_i^{GL} u_0 + v_0\right| \, w_i^{GL} \, \lambda_j \, f\big(U(t_i^{GL}, \theta_j + \mu)\big), \tag{1.18}$$

per ogni $f \in \mathbb{P}_n^2$.

Capitolo 2

Implementazione

In questo capitolo esamineremo in dettaglio la routine `trigauss`, sia dal punto di vista teorico che in quello implementativo in Python.

In particolare ci focalizzeremo

- sull'algoritmo di Chebyshev modificato, implementato nella funzione `chebyshev`, che permette il calcolo dei coefficienti di ricorrenza,
- sull'algoritmo di Golub-Welsch `gauss`, che determina i nodi e i pesi della formula gaussiana a partire dai coefficienti di ricorrenza.

Gli algoritmi `chebyshev` e `gauss` permetteranno la definizione della routine `trigauss`, che prende di input:

- n che fissa il grado di precisione trigonometrico richiesto dall'utente;
- le variabili α, β che definiscono l'intervallo di definizione (α, β).

`trigauss` fornisce quale output un vettore di dimensione $(n+1) \times 2$ avente quale

- prima colonna i nodi angolari, contenuti in $(-\omega + \mu, \omega + \mu)$;
- seconda colonna i pesi corrispondenti.

2.0.1 Algoritmo di Chebyshev modificato

L'obiettivo dell'algoritmo di Chebyshev modificato è di trovare i coefficienti di ricorsione $\{\alpha_k, \beta_k\}_{k\geq 0}$ dei polinomi ortogonali $\{\pi_k\}_{k\geq 0}$ relativi alla funzione peso w, che nel nostro caso particolare é (1.6).

Supponiamo sia nota, o facile da determinare, una certa famiglia di polinomi ortogonali $\{p_k\}_{k\geq 0}$ relativamente a una funzione peso. In particolare, supponiamo di avere a disposizione i rispettivi coefficienti di ricorsione $(a_k, b_k)_{k\geq 0}$, scelti semplici da calcolare, definiti entrambi nello stesso intervallo (α, β)

Definiamo i momenti misti $\sigma_{k,l}$ come

$$\sigma_{k,l} = \int_{\alpha}^{\beta} \pi_k(x) p_l(x) w(x) dx \tag{2.1}$$

dove π_k é il polinomio ortogonale di grado k rispetto alla funzione peso w di nostro interesse.

Il proposito é che, partendo dai momenti misti $\{\sigma_{k,l}\}$, si ottengano ricorsivamente i coefficienti di ricorsione desiderati $\{\alpha_k, \beta_k\}$ relativamente alla funzione peso w.

Osserviamo che dalle proprietá dei polinomi ortogonali

- si ha $\sigma_{k,l} = 0$ se $k > l$;
- dalla formula di ricorrenza per i polinomi ortogonali p_k e dalle loro proprietá di ortogonalitá

$$\sigma_{k,k} = \int_\alpha^\beta \pi_k^2(x)w(x)dx = \int_\alpha^\beta \pi_k(x)xp_{k-1}(x)w(x)dx \quad k \geq 1. \tag{2.2}$$

La prima uguaglianza di (2.2) deriva dal fatto che $\{\pi_k\}_{k\geq 0}$ è una famiglia di polinomi ortogonali e quindi sostituendo a $p_l(x)$ una combinazione lineare dei primi $l+1$ polinomi ortogonali $\pi_0(x), ..., \pi_l(x)$ l'integrale si annulla per la proprietà (1.2).

La seconda discende dalla prima e dalla formula di Christoffel (1.1.2).

Sapendo che $\sigma_{k+1,k-1} = 0$ ricaviamo

$$0 = \int_\alpha^\beta ((x - \alpha_k)\pi_k(x) - \beta_k\pi_{k-1}(x))p_{k-1}(x)w(x)dx, \tag{2.3}$$

e con facili conti

$$\beta_k = \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}} \quad k = 1, 2, ... \tag{2.4}$$

con $\beta_0 = m_0$. Inoltre da $\sigma_{k+1,k} = 0$

$$0 = \int_\alpha^\beta ((x - \alpha_k)\pi_k(x) - \beta_k\pi_{k-1}(x))p_k(x)w(x)dx, \tag{2.5}$$

si ricava

$$\begin{aligned} \alpha_0 &= a_0 + \frac{m_1}{m_0}, \\ \alpha_k &= a_k - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}} + \frac{\sigma_{k,k+1}}{\sigma_{k,k}}, \end{aligned} \tag{2.6}$$

per $k = 1, 2, ..., 2n-1$. Infatti usando l'uguglianza $xp_k(x) = p_{k+1}(x) + a_kp_k(x) + b_kp_{k-1}(x)$ troviamo

$$0 = \sigma_{k,k+1} + (a_k - \alpha_k)\sigma_{k,k} - \beta_k\sigma_{k-1,k}. \tag{2.7}$$

Concludiamo osservando che

$$0 = \int_\alpha^\beta ((x - \alpha_{k-1})\pi_{k-1}(x) - \beta_{k-1}\pi_{k-2}(x))p_l(x)w(x)dx, \tag{2.8}$$

e sostituendo $xp_l(x)$ come sopra ricaviamo

$$\sigma_{k,l} = \sigma_{k-1,l+1} - (\alpha_{k-1} - a_l)\sigma_{k-1,l} - \beta_{k-1}\sigma_{k-2,l} + b_l\sigma_{k-1,l-1}. \tag{2.9}$$

L'algoritmo di Chebyshev modificato calcola $\sigma_{k,l}$ in modo ricorsivo

$$\begin{aligned} \sigma_{-1,l} &= 0, \quad l = 1, 2, ..., 2n-2, \\ \sigma_{0,l} &= m_l, \quad l = 0, 1, ..., 2n-1, \end{aligned} \tag{2.10}$$

e per $k = 1, 2, ..., 2n-1$:

$$\sigma_{k,l} = \sigma_{k-1,l+1} - (\alpha_{k-1} - a_l)\sigma_{k-1,l} - \beta_{k-1}\sigma_{k-2,l} + \beta_l\sigma_{k-1,l-1}, \quad l = k, \ldots, 2n-k-1. \tag{2.11}$$

Si conclude ricavando i coefficienti $\{\alpha_k, \beta_k\}$ dalle formule (2.4) e (2.6).

La routine `chebyshev` consiste nell'implementazione di questo algoritmo. Nel dettaglio, dati in input

- i vettori `mom` dei momenti modificati,
- la matrice `abm` le cui colonne determinano i coefficienti $(a_k, b_k)_{k\geq 0}$,

calcola i coefficienti di ricorsione dei polinomi ortogonali $\{\pi_k\}_{k\geq 0}$.

La traduzione da Matlab a Python della routine `chebyshev` descritta da W. Gautschi non presenta particolari difficoltà.

```
function [ab,normsq]=chebyshev(N,mom,abm)

ab(1,1)=abm(1,1)+mom(2)/mom(1); ab(1,2)=mom(1);

if N==1, normsq(1)=mom(1); return, end

sig(1,1:2*N)=0; sig(2,:)=mom(1:2*N);
for n=3:N+1
    for m=n-1:2*N-n+2
        sig(n,m)=sig(n-1,m+1)-(ab(n-2,1)-abm(m,1))*sig(n-1,m)...
            ...-ab(n-2,2)*sig(n-2,m)+abm(m,2)*sig(n-1,m-1);
    end
    ab(n-1,1)=abm(n-1,1)+sig(n,n)/sig(n,n-1)-sig(n-1,n-1)...
        .../sig(n-1,n-2);
    ab(n-1,2)=sig(n,n-1)/sig(n-1,n-2);
end
for n=1:N, normsq(n)=sig(n+1,n); end; normsq=normsq';
```

Codice `chebyshev` in Matlab

```
def chebyshev(N, mom, abm):

    # inizializza ab
    ab = np.zeros((N, 2))
    ab[0, 0] = abm[0, 0] + mom[1] / mom[0]
    ab[0, 1] = mom[0]
```

```
    if N == 1:
        normsq = np.array([mom[0]])
        return ab, normsq.reshape(-1, 1)

    # Inizializziamo matrice sig
    sig = np.zeros((N + 1, 2 * N)) #righe da 0 a N, colonne da 0 a 2N-1
    sig[1, :] = mom[0:2*N]

    # ciclo per il calcolo di ab
    for n in range(2, N + 1):
        for m in range(n - 3, 2 * N - n + 1):
            sig[n,m]=sig[n-1,m+1]+(ab[n-2,0]-abm[m,0])*sig[n-1,m]...
                ...-ab[n-2,1]*sig[n-2,m]+abm[m,1]*sig[n-1,m-1]
        ab[n-1,0]=abm[n-1,0]+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]

    normsq = np.zeros(N)
    for n in range(N):
        normsq[n] = sig[n + 1, n]

    return ab, normsq.reshape(-1, 1)
```

Codice `chebyshev` in Python

Il vettore `normsq` contiene le norme al quadrato dei polinomi ortogonali di cui abbiamo calcolato i coefficienti.

É importante osservare che non è restrittiva la scelta dei polinomi $\{p_k\}_{k\geq 0}$, calcolando i momenti di conseguenza si troveranno dei coefficienti differenti. Come vedremo in seguito, l'unico aspetto che è influenzato da questo fattore è la complessità computazionale.

2.0.2 Algoritmo di Golub-Welsch

Questo algoritmo ha lo scopo di ricavare nodi e pesi di integrazione della formula gaussiana dati i coefficienti di ricorsione (1.1.2) dei polinomi ortogonali relativi a una specifica funzione peso, nel nostro caso (1.6).

La formula di Christoffel (1.1.2) e vediamo la sua equazione matriciale:

$$x \cdot \begin{pmatrix} \pi_0(x) \\ \pi_1(x) \\ \vdots \\ \pi_{n-1}(x) \end{pmatrix} = \begin{pmatrix} -\alpha_0 & 1 & & \\ \beta_1 & -\alpha_1 & \ddots & \\ & \ddots & \ddots & 1 \\ & & \beta_n & -\alpha_n \end{pmatrix} \cdot \begin{pmatrix} \pi_0(x) \\ \pi_1(x) \\ \vdots \\ \pi_{n-1}(x) \end{pmatrix} + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \pi_n(x) \end{pmatrix} \tag{2.12}$$

o in modo equivalente in notazione matriciale:

$$x\tilde{\pi}(x) = T\tilde{\pi}(x) + \pi_n(x)e_n \tag{2.13}$$

dove $\tilde{\pi}(x)$ è il vettore contenente tutti i polinomi ortogonali, T è la matrice tridiagonale e e_n il vettore canonico n-esimo. Pertanto $\pi_n(t_j) = 0$ se e solo se:

$$t_j\tilde{\pi}(t_j) = T\tilde{\pi}(t_j), \tag{2.14}$$

dove $\{t_j\}_{j\geq 0}$ sono gli autovalori della matrice tridiagonale T. Per trovare quindi gli zeri del polinomio ortogonale $\pi_n(x)$, con n scelto, che è a noi sconosciuto, utilizzeremo l'algoritmo di Golub-Welsch, quest'ultimo va a calcolare gli autovalori della matrice T attraverso la decomposizione spettrale. Per semplificare i calcoli però, definiamo la matrice J_n di Jacobi, simile a T, di questa forma:

$$DTD^{-1} = J_n = \begin{pmatrix} \alpha_0 & \sqrt{\beta_1} & & \\ \sqrt{\beta_1} & \alpha_1 & \ddots & \\ & \ddots & \ddots & \sqrt{\beta_n} \\ & & \sqrt{\beta_n} & \alpha_n \end{pmatrix}, \quad \exists D. \tag{2.15}$$

Ricordiamo la proprietà fondamentale della similitudine tra matrici: gli autovalori sono invarianti per essa, quindi applicheremo l'algoritmo alla matrice tridiagonale simmetrica J_n.

Come approfondito in **[GW]** dall'identità di Christoffel-Darboux, che associa pesi w_j a polinomi, sappiamo che

$$w_j(\tilde{\pi}(t_j))^T(\tilde{\pi}(t_j)) = 1, \quad j = 1, 2, ..., n, \tag{2.16}$$

dove $\tilde{\pi}(t_j)$ corrisponde all'autovettore normalizzato associato all'autovalore t_j. Considerati $\{q_j\}_{j\geq 0}$ gli autovettori di J_n avremmo

$$Jq_j = t_j q_j, \quad j = 1, 2 \dots n, \tag{2.17}$$

con $q_j^T q_j = 1$.

Da questo, con la notazione $q_j^T = (q_{0,j}, ..., q_{n-1,j})$, si ricava

$$q_{0,j}^2 = w_j(\pi_0(t_j))^2, \tag{2.18}$$

e da qui possiamo enunciare il seguente teorema:

Teorema 2.0.1. *Siano rispettivamente $t_1 \dots t_n$ e $w_1 \dots w_n$ i nodi e i pesi della formula gaussiana a n punti. Allora i nodi sono gli autovettori di J_n, e i pesi:*

$$w_j = \beta_0 q_{k,1}^2, \tag{2.19}$$

dove $\beta_0 = \int_{\mathbb{R}} w(x)dx$ e $q_{k,1}$ è la prima componente dell'autovettore q_j relativo all'autovalore t_j.

Di conseguenza, se si riescono a calcolare gli autovalori della matrice T e la prima componente degli autovettori, si trovano i risultati sperati: nodi e pesi della formula di quadratura di Gauss.

Questo si converte in Python nel programma `gauss` che applica i risultati del Teorema 2.0.1 nei seguenti codici.

```
function xw=gauss(N,ab)
J=zeros(N);
for n=1:N
    J(n,n)=ab(n,1);
end
for n=2:N
  J(n,n-1)=sqrt(ab(n,2));
  J(n-1,n)=J(n,n-1);
end
[V,D]=eig(J);
[D,I]=sort(diag(D));
V=V(:,I);
xw=[D ab(1,2)*V(1,:)'.^2];
```

Codice `gauss` in Matlab

```
def gauss(N, ab):

    # Costruiamo la matrice di Jacobi J
    J = np.zeros((N, N))
    for n in range(N):
        J[n, n] = ab[n, 0]
    for n in range(1, N):
        J[n, n - 1] = np.sqrt(ab[n, 1])
        J[n - 1, n] = J[n, n - 1]

    # Calcola autovalori (nodi) e autovettori
    autoval, autovet = np.linalg.eigh(J)

    # Ordina gli autovalori e riordina gli autovettori di conseguenza
    sorted_indices = np.argsort(autoval)
    nodes = autoval[sorted_indices]      # Ordina i nodi
    autovet = autovet[:, sorted_indices]

    # Calcola i pesi
    weights = ab[0, 1] * (autovet[0, :]**2)

    xw = np.column_stack((nodes, weights))

    return xw
```

Codice `gauss` in Python

Così dati in input, nel codice `gauss`, il vettore $(n+1)\times 2$ `ab`, ricavato dall'esecuzione di `chebyshev`, contenente i coefficienti di ricorsione di $\{\pi_j\}_{j\geq 0}$, ci vengono restituiti i nodi e i pesi della formula algebrica di Gauss.

2.0.3 La routine `trigauss`

La routine `trigauss`, precedentemente implementata in Matlab, dato un intervallo angolare $[\alpha,\beta]$ restituisce pesi e nodi per la formula Gaussiana trigonometrica. Il

proposito di questa sottosezione é di mostrare le considerazioni che abbiamo fatto per la sua traduzione in Python.

Per prima cosa dobbiamo ricavare i momenti modificati di Chebyshev (1.10) rispetto alla funzione peso (1.6). A tal proposito, senza calcolare ogni integrale numericamente, aumentando così la complessità computazionale, possiamo utilizzare una formula di ricorsione dei momenti pari. La formula è del tipo

$$a_n m_{2n+2} + b_n m_{2n} + c_n m_{2n-2} = d_n, \quad n = 1, 2, \dots \tag{2.20}$$

come mostrato in **[P]** in cui vengono forniti i coefficienti. In questo modo è richiesto solamente conoscere il primo e l'ultimo momento modificato (ovvero m_0 e m_{2n}) e risolvere il rispettivo sistema lineare, con la seguente matrice tridiagonale dei coefficienti ricorsivi

$$\begin{pmatrix} b_1 & a_1 & & \\ c_2 & b_2 & \ddots & \\ & \ddots & \ddots & a_{n-1} \\ & & c_n & b_n \end{pmatrix} \tag{2.21}$$

Concentriamo la nostra attenzione al calcolo di m_0 e m_{2n}.
Per quanto concerne

- il momento m_0, questo equivale a 2ω e si ricava risolvendo l'integrale (1.10) in pochi passaggi;
- il momento m_{2n}, utilizziamo la funzione `quad` di Python importata dal pacchetto di integrazione `scipy.integrate`. Questa ricorsione non è infatti stabile se svolta *in avanti*, partendo da m_0 e m_2, ma lo è se si risolve il sistema attraverso l'algoritmo `tridisolve`, che presi in input i vettori contenenti i coefficienti restituisce il vettore dei momenti. Questo programma si basa sul più noto algoritmo di Thomas, un caso particolare di quello di eliminazione di Gauss, con un costo computazionale più basso, di $O(N)$.

La funzione `tridisolve` si implementa come di seguito.

```
function x = tridisolve(a,b,c,d)
x = d;
n = length(x);
for j = 1:n-1
   mu = a(j)/b(j);
   b(j+1) = b(j+1) - mu*c(j);
   x(j+1) = x(j+1) - mu*x(j);
end
x(n) = x(n)/b(n);
for j = n-1:-1:1
   x(j) = (x(j)-c(j)*x(j+1))/b(j);
end
end
```

Codice `tridisolve` in Matlab

```
def tridisolve(a, b, c, d):
    x = np.copy(d)
    n = len(x)

    for j in range(n - 1):
        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 - 1] = x[n - 1] / b[n - 1]
    for j in range(n - 2, -1, -1):
        x[j] = (x[j] - c[j] * x[j + 1]) / b[j]
    return x
```

Codice `tridisolve` in Python

Una domanda che sorge spontanea é: perché utilizzare i momenti composti e non quelli classici, che sarebbero anche più semplici da calcolare? I momenti classici risultano più instabili e per questo motivo propagano meglio gli errori; infatti, utilizzando i momenti composti l'approssimazione risulta più precisa. Abbiamo già osservato che i momenti dispari si annullano e per questo troveremo un vettore con gli indici dispari (in Python) uguali a 0. Questo vettore va però normalizzato, come anche i polinomi di Chebyshev.

Approfondiamo meglio quest'ultimo aspetto. I polinomi di Chebyshev vanno normalizzati a polinomi monici nel seguente modo:

$$P_k(x) = \frac{T_k(x)}{2^{k-1}}, \quad P_0 = T_0, \quad k \geq 1, \tag{2.22}$$

così da ricondurci alle ipotesi del teorema 1.1.2 e trovare i coefficienti di ricorsione che servono, come già visto, per richiamare la funzione `chebyshev`. I coefficienti, di questi polinomi, sono inoltre già noti e non è quindi necessario aumentare la complessità computazionale: proprio per questo la scelta dei polinomi noti su cui usare l'algoritmo di Chebyshev modificato è ricaduta su $\{T_k(x)\}_{k\geq 0}$. Nodi e pesi della formula di Gauss in realtà possono infatti essere ricavati utilizzando altre famiglie di polinomi ortogonali, perdendo però l'efficienza del programma. Allo stesso modo normalizziamo i momenti

$$m_s = \int_{-1}^{1} T_s(x)w(x)dx = \frac{1}{2^{s-1}} \int_{-1}^{1} P_s(x)w(x)dx. \tag{2.23}$$

Il vettore contenente i primi $2n$ coefficienti di ricorsione dei polinomi monici di Chebyshev è `abm`, come sopra indicato, sono già noti e sono (usando la notazione di (1.1.2)):

$$\begin{aligned} &\alpha_i = 0, \quad \forall i, \\ &\beta_0 = \pi, \quad \beta_1 = \frac{1}{2}, \quad \beta_i = \frac{1}{4}, \quad i = 2, ..., 2n+1. \end{aligned} \tag{2.24}$$

A questo punto richiamiamo, prima, la funzione `chebyshev` e, poi, con i coefficienti dei polinomi ortogonali trovati, tali che valga (1.2) per la funzione peso (1.6), la

funzione `gauss`, che restituisce nodi e pesi della formula algebrica gaussiana (1.1.1); tutti i nodi trovati andranno poi trasformati in nodi angolari tramite la formula (1.8).

Il programma termina con la restituzione del vettore $(n+1) \times 2$ `tw` inclusivo di nodi angolari e pesi.

Di seguito concludiamo il capitolo con i codici di `trigauss`.

```
    function tw=trigauss(n,alpha,beta)
    omega=(beta-alpha)/2;

    z(1)=2*omega;
    z(n+1)=quadgk(@(t)cos(2*n*acos(sin(t/2)/sin(omega/2))),...
        ...-omega,omega,'MaxIntervalCount',5000);
    temp=(2:2:2*n-1);
    dl=1/4-1./(4*(temp-1));
    dc=1/2-1/sin(omega/2)^2-1./(2*(temp.^2-1));
    du=1/4+1./(4*(temp+1));
    d=4*cos(omega/2)/sin(omega/2)./(temp.^2-1)';
    d(n-1)=d(n-1)-du(n-1)*z(n+1);
    z(2:n)=tridisolve(dl(2:n-1),dc(1:n-1),du(1:n-2),d(1:n-1));
    mom=zeros(1,2*n+2);
    mom(1:2:2*n+1)=z(1:n+1);

    k=(3:length(mom));
    mom(3:end)=exp((2-k)*log(2)).*mom(3:end);

    abm(:,1)=zeros(2*n+1,1);
    abm(:,2)=0.25*ones(2*n+1,1); abm(1,2)=pi; abm(2,2)=0.5;

    [ab,normsq]=chebyshev(n+1,mom,abm);
    xw=gauss(n+1,ab);

    tw(:,1)=2*asin(sin(omega/2)*xw(:,1))+(beta+alpha)/2;
    tw(:,2)=xw(:,2);
end
```

Codice `trigauss` in Matlab

```
def trigauss(n, alpha, beta):
    omega = (beta - alpha) / 2

    # ricorsione dei momenti di Chebyshev modificati
    z = np.zeros(n + 2)
    z[0] = 2 * omega
    integral_result, _ = quad(lambda t: np.cos(2 * n *np.arccos(...
        ...np.sin(t/2)/np.sin(omega/2))),-omega, omega, limit=5000)
    z[n] = integral_result

    #vettore ausiliare
    temp = np.arange(2, 2 * n, 2)
    #vettori di ricorsione dei momenti
    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)

    #vettore dei momenti
    mom = np.zeros(2 * n + 2)
    mom[0:2*n+2:2] = z[0:n+1]

    # normalizzazione dei momenti
    k_vals = np.arange(2, len(mom))+1;
    mom[2:] = np.exp((2 - k_vals) * np.log(2)) * mom[2:]

    # coefficienti di ricorsione dei polinomi ortogonali
    abm = np.zeros((2 * n + 1, 2))
    abm[:, 1] = 0.25
    abm[0, 1] = np.pi
    abm[1, 1] = 0.5

    #algoritmo di chebyshev modificato
    ab, normsq = chebyshev(n + 1, mom, abm)

    # formula gaussiana per i pesi e nodi
    xw = gauss(n + 1, ab)

    # angoli e pesi per la formula trigonometrica
    tw = np.zeros((n + 1, 2))
    tw[:, 0] = 2 * np.arcsin(np.sin(omega / 2) * xw[:, 0])...
        ...+ (beta + alpha) / 2
    tw[:, 1] = xw[:, 1]

    return tw
```

Codice `trigauss` in Python

Capitolo 3

Test numerici in Python e Matlab

In questo capitolo mostriamo attraverso dei test numerici che il programma `trigauss` convertito in Python si comporta numericamente come l'originale in Matlab.

Per limitare le differenze tra i risultati nei 2 linguaggi, utilizziamo funzioni il più possibile equivalenti. Un punto di rilievo, per spiegare tale sforzo, sta nel calcolo numerico del momento m_0. Il programma Matlab utilizza `quadgk`, funzione che per calcolare l'integrale definito applica il metodo di cubatura adattiva di Gauss-Kronrod, così per il codice Python abbiamo scelto la corrispondente funzione `quad` importata da `scipy.integrate`, con settaggi che permettano quantitá uguali (a meno di una tolleranza dell'ordine della precisione di macchina).

Nei test, oltre al paragone numerico, ci interessiamo a considerare i tempi di calcolo delle procedure Matlab e Python.

Nella nostra analisi consideriamo 2 subintervalli con ampiezze di ordine diverso, ovvero

- $\Omega_1 = (\pi/6, \pi/4)$, di semiampiezza $\omega \approx 0.2618$;
- $\Omega_2 = (\pi/32, \pi/31)$, di semiampiezza $\omega \approx 0.0016$.

Per verificare che i codici forniscano formule numericamente equivalenti, calcoleremo attraverso le formule prodotte in Matlab e Python, a paritá di grado di precisione:

- il massimo errore tra i nodi e i pesi nei due sistemi di calcolo, ovvero

$$\begin{aligned} E_{nodi} &= ||x_m - x_p||_\infty \\ E_{pesi} &= ||w_m - w_p||_\infty \end{aligned} \tag{3.1}$$

  con $\{x_m, w_m\}$ e $\{x_p, w_p\}$ vettori contenenti i nodi e i pesi determinati con `trigauss` rispettivamente in Matlab e in Python;

- l'errore relativo compiuto nell'approssimare

$$I = \int_{\Omega_i} 5 + \frac{1}{2}\sin(17x) - 6\cos(14x)\, dx, \quad i = 1, 2$$

tra gli integrali calcolati numericamente dai 2 linguaggi rispettivamente I_m , I_p, ovvero

$$\begin{aligned}\tilde{E} &= \frac{|I - I_p|}{I} \\ &= \frac{\left|I - \sum_{j=1}^{n} w_{p,j} f(x_{p,j})\right|}{I},\end{aligned} \tag{3.2}$$

con $x_p = (x_{p,j})$, $w_p = (w_{p,j})$.

Quale valore di riferimento I prendiamo quello fornito dalla formula definita su Matlab da `trigauss` per $ADE = 17$, essendo l'integranda un polinomio trigonometrico di grado 17.

Per effettuare tale confronto

1. applichiamo la versione Python di `trigauss` nell'intervallo richiesto e salviamo il vettore risultante, xwp, in un file leggibile in Matlab con le funzioni `savemat` e `files.download`;
2. carichiamo il file `.mat` in Matlab e applichiamo la funzione `confronto` che, applica trigauss e genera il vettore risultante xwm, e calcola la norma infinito a $|xwm - xwp|$;
3. eseguiamo la routine Matlab, per poi calcolare le quantitá richieste.

La sopracitata funzione `confronto` ha quale codice

```
function [errxw]=confronto(n,alpha,beta,xwp)
xwm=trigauss(n,alpha,beta);

errxw(1) = norm(xwm(:,1) - xwp(:,1),inf);
errxw(2) = norm(xwm(:,2) - xwp(:,2),inf);
```

Codice `confronto` in Matlab

Allo stesso modo per il secondo test: il procedimento è molto simile, l'unica differenza consiste nel salvare direttamente i risultati del calcolo dell'integrale in Python e poi confrontarli con quelli in Matlab.

```
function [errA,errR]=integ(n,alpha,beta,intP)
xwm=trigauss(n,alpha,beta);
xwm17=trigauss(17,alpha,beta);
f=@(x) 5+(1/2)*sin(17*x)-6*cos(14*x);

intM=sum(xwm(:,2).*f(xwm(:,1)));
int17=sum(xwm17(:,2).*f(xwm17(:,1)));

errA=abs(intM- intP);
```

```
errR=abs(int17- intP)/int17;
```

Codice `integ` in Matlab

Per determinare i tempi di calcolo, utilizziamo le funzioni `tic`, `toc` di Matlab e la funzione `time.time` importata dal pacchetto `time` di Python. Poiché il tempo di esecuzione però potrebbe variare da un'esecuzione all'altra, svolgiamo 100 test e di seguito riportiamo un risultato medio.

Ogni codice Python che è stato sviluppato in questo lavoro sarà anche reperibile su Github **[S]** e liberamente fruibile.

3.0.1 Intervallo $(\pi/6, \pi/4)$

Come anticipato, cominciamo con il paragone dei codici nell'intervallo $(\pi/6, \pi/4)$, verificando se i programmi Python hanno un livello di accuratezza simile ai corrispettivi in Matlab, aspettandoci un errore prossimo alla precisione macchina.

Nella tabella 3.1, considerando i gradi n pari a 5, 10, 15, 20, eseguiamo i confronti dei nodi e dei pesi determinati dai due sistemi di calcolo. Da questi risultati vediamo che la differenza tra i due linguaggi risulta dell'ordine della precisione di macchina e quindi ci aspettiamo che i risultati della loro applicazione al calcolo di integrali di polinomi trigonometrici di grado n forniscano essenzialmente lo stesso risultato.

n	E_{nodi}	E_{pesi}
5	0	9.02×10^{-17}
10	1.11×10^{-16}	1.35×10^{-16}
15	1.11×10^{-16}	1.70×10^{-16}
20	0	4.51×10^{-17}

Tabella 3.1: Massimi errori nei nodi e nei pesi, tra le formule ottenute in Matlab e Python, relativamente al primo intervallo $(\pi/6, \pi/4)$.

Nella tabella 3.2, analizziamo i risultati del test nell'approssimare

$$I = \int_{\pi/6}^{\pi/4} 5 + \frac{1}{2}\sin(17x) - 6\cos(14x)dx \approx 2.062453518370601$$

per i gradi n pari a 5, 10, 15, 20, con a sinistra l'errore assoluto tra i 2 linguaggi, ovvero $E_{assoluto} = |I_m - I_p|$, mentre a destra l'errore relativo del risultato in Python rispetto al risultato esatto I calcolato in Matlab con $n = 17$, grado del polinomio, $E_{relativo} = \frac{|I - I_p|}{I}$. Visto il valore dell'integrale, gli errori relativi e quelli assoluti sono dello stesso ordine, tolto quello di ordine 5, in cui probabilmente il grado di precisione della formula è troppo basso (si osservi che l'integranda è un polinomio trigonometrico di grado 17).

n	$E_{assoluto}$	$E_{relativo}$
5	1.78×10^{-15}	3.31×10^{-11}
10	4.44×10^{-16}	6.46×10^{-16}
15	8.88×10^{-16}	1.51×10^{-15}
20	4.44×10^{-16}	1.29×10^{-15}

Tabella 3.2: Errori relativi agli integrali calcolati sul primo intervallo $(\pi/6, \pi/4)$.

Ci concentriamo infine sui tempi di calcolo, mostrando tali risultati nella tabella 3.3, in cui vengono esposti i tempi t_p, t_m rispettivamente in Python e in Matlab. Questi sono tempi medi rispetto a 100 iterazioni e vengono calcolati attraverso $\sum_{k=1}^{100} t_k/100$, con t_k il tempo necessario per svolgere il k-esimo esperimento.

n	$t_p[s]$	$t_m[s]$
5	4.53×10^{-4}	2.91×10^{-4}
10	1.25×10^{-3}	2.69×10^{-4}
15	1.71×10^{-3}	2.50×10^{-4}
20	2.79×10^{-3}	3.38×10^{-3}

Tabella 3.3: Tempi medi in secondi t_p, t_m dei programmi nei 2 linguaggi (rispettivamente Python e Matlab), per il calcolo di formule con grado di precisione trigonometrico n.

Da questa analisi, si vede che Matlab offre dei risultati in un tempo minore rispetto a Python, anche se le differenze risultano minime.

3.0.2 Intervallo $(\pi/32, \pi/31)$

Ora esaminiamo un intervallo di ampiezza minore, per vedere se ció influenza il paragone tra i codici nei 2 linguaggi.

Per prima cosa valutiamo se i nodi e i pesi proposti dai codici Matlab e Python coincidono. Come per il precedente intervallo, dalla tabella 3.4 si ricava che i risultati sono numericamente equivalenti.

n	E_{nodi}	E_{pesi}
5	0	5.01×10^{-19}
10	0	2.30×10^{-18}
15	0	1.32×10^{-18}
20	0	1.39×10^{-18}

Tabella 3.4: Errori dei linguaggi nei nodi e nei pesi calcolati con il metodo della norma per il secondo intervallo $(\pi/32, \pi/31)$.

Come conseguenza ci aspettiamo che qualora i codici dovessero essere applicati per approssimare integrali di funzioni continue, i risultati ottenuti sarebbero paragonabili.

Nella tabella successiva sono elencati i risultati del secondo test, in cui valutiamo numericamente

$$I = \int_{\pi/32}^{\pi/31} 5 + \frac{1}{2}\sin(17x) - 6\cos(14x)dx \approx 0.014112808373797$$

per i gradi n pari a 5, 10, 15, 20, con a sinistra l'errore assoluto tra i 2 linguaggi $E_{assoluto} = |I_m - I_p|$ mentre a destra l'errore relativo $E_{relativo} = \frac{|I-I_p|}{I}$ del codice Python. L'integrale esatto I, come nel caso precedente, sarà calcolato in Matlab col grado 17 di `trigauss`, grado del polinomio trigonometrico su cui stiamo facendo le nostre valutazioni.

n	$E_{assoluto}$	$E_{relativo}$
5	3.47×10^{-18}	6.15×10^{-16}
10	3.47×10^{-18}	3.69×10^{-16}
15	3.47×10^{-18}	2.46×10^{-16}
20	1.91×10^{-17}	9.83×10^{-15}

Tabella 3.5: Errori relativi agli integrali calcolati sul secondo intervallo $(\pi/32, \pi/31)$.

Come nell'intervallo precedente, con la stessa notazione, analizziamo la differenza dei tempi di calcolo. La tabella sottostante evidenzia che le differenze sono trascurabili, anche se il codice Matlab in genere risulta piú veloce.

n	$t_p[s]$	$t_m[s]$
5	5.06×10^{-4}	2.93×10^{-4}
10	7.28×10^{-4}	1.64×10^{-4}
15	1.80×10^{-3}	2.82×10^{-4}
20	2.95×10^{-3}	2.58×10^{-4}

Tabella 3.6: Tempi medi di applicazione t_p, t_m dei programmi nei 2 linguaggi (rispettivamente Python e Matlab), per il calcolo di formule con grado di precisione trigonometrico n.

L'ultima osservazione importante da fare riguarda la differenza nei tempi tra le diverse *run* dei programmi: anche se i tempi sono una media, quest'ultima varia anche di un $10\% - 20\%$, ma questo non è un problema visto l'ordine dei tempi, che è comunque molto basso.

3.1 Test su settori circolari

Una delle applicazioni di `trigauss`, consiste nell'utilizzo atto a calcolare numericamente l'integrale di funzioni bivariate su dei settori ellittici generati da *linear blending* (cf. Proposizione 1.2.1). Andremo quindi a utilizzare i linguaggi Matlab e Python per il calcolo numerico, mettendoli a confronto.

Supponiamo che il dominio Ω sia la combinazione convessa dei due archi:

$$\Omega = \{(x,y) = U(t,\theta) = tP(\theta) + (1-t)Q(\theta),\ (t,\theta) \in [0,1] \times [\alpha,\beta]\} \tag{3.3}$$

con la mappa U iniettiva in $(0,1) \times (\alpha,\beta)$, dove

$$\begin{aligned} P(\theta) &= A_1 \cos(\theta) + B_1 \sin(\theta) + C_1, \\ Q(\theta) &= A_2 \cos(\theta) + B_2 \sin(\theta) + C_2, \end{aligned} \tag{3.4}$$

con $\theta \in (\alpha,\beta)$ e $0 \le \alpha, \beta \le 2\pi$. $A_i,\ B_i,\ C_i$ sono le coppie

$$A_i = (a_{i1}, a_{i2}), \quad B_i = (b_{i1}, b_{i2}), \quad C_i = (c_{i1}, c_{i2}), \quad i = 1,2.$$

Per applicare la formula (1.18), ovvero

$$\iint_\Omega f(x,y)\,dx\,dy = I_n(f) = \sum_{j=1}^{n+1} \sum_{i=1}^{\lceil \frac{n+2}{2} \rceil} \left| t_i^{GL} u_0 + v_0 \right| \, w_i^{GL} \, \lambda_j \, f\big(U(t_i^{GL}, \theta_j + \mu)\big), \tag{3.5}$$

risultano necessarie

- la coppia $\{\theta_j, \lambda_j\}_j$ di nodi e pesi della formula Gaussiana trigonometrica con grado precisione $n+k$ sull'intervallo angolare $[\alpha,\beta]$.
- $\{t_j^{GL}, w_j^{GL}\}_j$ della formula di Gauss-Legendre con grado precisione $n+h$ sull'intervallo $[0,1]$.

A tal proposito, nella versione Python, adoperiamo

- `trigauss` per la formula gaussiana trigonometrica,
- per la formula di Gauss-Legendre la routine `r_jacobi` che calcola i coefficienti di ricorsione dei polinomi ortogonali (di tipo monico) per il peso di Jacobi che, usata insieme a `gauss` ricava, tramite l'algoritmo di Golub-Welsch, i pesi e i nodi relativi al peso di Jacobi

  $$w(x) = (1-x)^a (1+x)^b.$$

  Nel nostro caso con $a = b = 0$ ricaviamo i coefficienti relativi al peso di Legendre

  $$w(x) \equiv 1.$$

Con questo metodo troviamo $n+h$ nodi in $[-1,1]$ ma a noi serviranno in $[0,1]$, andranno quindi scalati per l'intervallo di riferimento.

3.1.1 Un primo dominio di tipo linear blending

Come primo dominio consideriamo il dominio di tipo *linear blending* in cui

$$\Omega = \{(x, y) = U(t, \theta)\}, \quad U(t, \theta) = tP(\theta) + (1 - t)Q(\theta) \tag{3.6}$$

con

$$P(\theta) = (0, 0), \quad Q(\theta) = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \cdot \cos(\theta) + \begin{pmatrix} 0 \\ 2 \end{pmatrix} \cdot \sin(\theta) \tag{3.7}$$

Se consideriamo l'intervallo angolare $[-\frac{\pi}{4}, \frac{\pi}{4}]$ troviamo un settore circolare centrato nell'origine, come nella figura, in cui evidenziamo anche i nodi di una formula con ADE pari a 3.

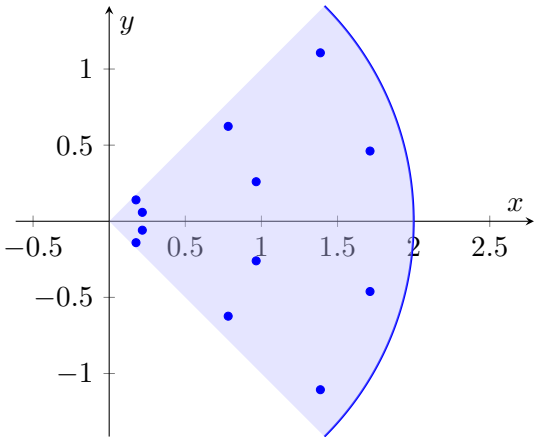


Figura 3.1: Settore circolare e nodi di una formula avente ADE pari a 3.

Per verificare l'esattezza dell'approssimazione numerica in Python, ovvero $I_p \approx I$, faremo come nella sezione precedente un confronto con i risultati in Matlab, $I_m \approx I$, e calcoleremo l'errore assoluto tra i due linguaggi:

$$E_{assoluto} = |I_p - I_m|$$

e l'errore relativo dei risultati in Python rispetto al risultato esatto I calcolato in Matlab con grado 8, lo stesso grado della funzione che vedremo di seguito nella formula (3.8):

$$E_{relativo} = \frac{|I - I_p|}{I}.$$

I risultati che cerchiamo saranno derivanti dal calcolo integrale attraverso (3.5) del polinomio algebrico di grado totale 8

$$f(x, y) = x - y^3 + x^7 y \ \in \mathbb{P}_8^2 \tag{3.8}$$

I programmi che utilizziamo per questo tipo di analisi numerica sono `gqcircsect` e `gqcircsegm` di G. Da Fies e M. Vianello, che abbiamo tradotto da Matlab a Python.

La routine `gqcircsect` calcola

$$\int_{-\omega}^{\omega}\int_{r_1}^{r_2} g(r,\theta) r\, dr\, d\theta, \tag{3.9}$$

con $g(r,\theta) = f(r\cos(\theta), r\sin(\theta))$.

Considereremo gli input $r1, r2$ rispettivamente raggio interno ed esterno di un settore circolare, nel nostro caso avremo $r1 = 0,\ r2 = 2$. Inoltre, considerando la notazione della proposizione 1.2.1, per i settori circolari ricaviamo con pochi passaggi che $h = 1, k = 0$ e quindi una formula con cardinalità $\lceil \frac{n+2}{2} \rceil \times (n+1)$.

```
    function xyw = gqcircsect(n,omega,r1,r2)
    tw=trigauss(n,-omega,omega);

    % algebraic gaussian formula on the radial segments
    ab=r_jacobi(ceil((n+2)/2),0,0);
    xw=gauss(ceil((n+2)/2),ab);
    xw(:,1)=xw(:,1)*(r2-r1)/2+(r2+r1)/2;
    xw(:,2)=xw(:,2)*(r2-r1)/2;

    % creating the polar grid
    [r,theta]=meshgrid(xw(:,1),tw(:,1));
    [w1,w2]=meshgrid(xw(:,2),tw(:,2));

    % nodal cartesian coordinates and weights
    xyw(:,1)=r(:).*cos(theta(:));
    xyw(:,2)=r(:).*sin(theta(:));
    xyw(:,3)=r(:).*w1(:).*w2(:);

end
```

Codice `gqcircsect` in Matlab

```
    def gqcircsect(n, omega, r1, r2):
    tw = trigauss(n, -omega, omega)

    m_radial = int(np.ceil((n + 2) / 2))
    ab = r_jacobi(m_radial, 0, 0)
    xw = gauss(m_radial, ab)
    xw[:, 0] = xw[:, 0] * (r2 - r1) / 2 + (r2 + r1) / 2
    xw[:, 1] = xw[:, 1] * (r2 - r1) / 2

    r, theta = np.meshgrid(xw[:, 0], tw[:, 0])
    w1, w2 = np.meshgrid(xw[:, 1], tw[:, 1])

    x = r * np.cos(theta)
    y = r * np.sin(theta)
    w = r * w1 * w2

    xyw = np.column_stack((x.flatten(), y.flatten(), w.flatten()))
```

```
    return xyw
```

Codice `gqcircsect` in Python

Per confrontare i dati utilizzeremo lo stesso procedimento già visto, il comando `savemat` di Python sarà ancora necessario per trasferire i dati su Matlab e calcolare gli errori. Ci aspettiamo un errore dell'ordine della precisione macchina. Nella tabella seguente si vedono gli errori, con la stessa notazione della sezione precedente, aggiungiamo inoltre una colonna che indica il numero dei nodi su cui viene integrata la funzione, equivalente alla cardinalità.

n	nodi	$E_{assoluto}$	$E_{relativo}$
5	24	2.67×10^{-15}	5.89×10^{-16}
10	66	2.67×10^{-15}	4.71×10^{-16}
15	144	2.40×10^{-15}	6.83×10^{-15}
20	231	2.67×10^{-15}	5.89×10^{-16}

Tabella 3.7: Errori relativi agli integrali di $g(x, y)$ calcolati sul settore circolare generato da linear blending, al variare di n.

Osserviamo ora una tabella che evidenzia i tempi di calcolo dei programmi appena utilizzati, con la stessa notazione della sezione precedente; ci aspettiamo risultati simili a quelli già visti.

n	$t_p[s]$	$t_m[s]$
5	1.27×10^{-3}	2.33×10^{-4}
10	3.02×10^{-3}	2.91×10^{-4}
15	5.56×10^{-3}	3.32×10^{-4}
20	4.23×10^{-3}	4.38×10^{-4}

Tabella 3.8: Tempi medi di applicazione t_p, t_m del programma `gqcircsect` nei 2 linguaggi (rispettivamente Python e Matlab).

3.1.2 Un secondo dominio di tipo linear blending

In questo caso avremo un'area più complessa, in cui entrambe le curve sono diverse dalla funzione identicamente nulla, ovvero

$$P(\theta) = (\cos(x), -\sin(x)), \quad Q(\theta) = (\cos(x), \sin(x)). \tag{3.10}$$

Vediamo che anche in questo caso $U(t, \theta)$ è iniettiva nell'intervallo $[0, \pi/3]$ e delimita un'area non banale, come mostrato nella figura 3.2.

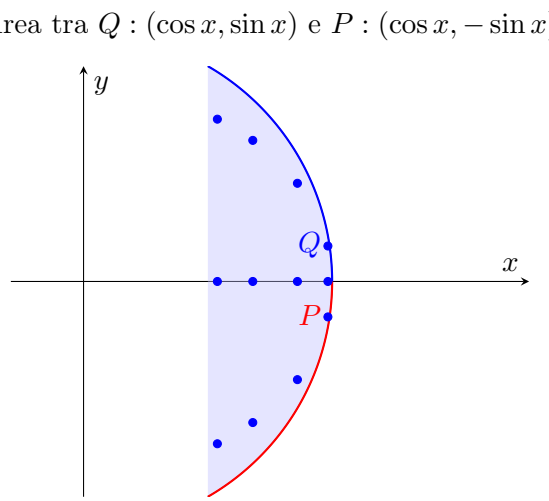


Figura 3.2: Segmento circolare e nodi di una formula avente ADE pari a 5.

Intendiamo calcolare l'integrale della medesima funzione f definita in (3.8), valutando l'errore compiuto. Anche in questo caso, poniamo $g(r, \theta) = f(r\cos(\theta), r\sin(\theta))$ e utilizziamo la funzione `gqcircsegm` che applica 1.15 a dei segmenti circolari. Con la notazione della preposizione 1.2.1, si calcola in pochi semplici passi che $h = 0, k = 2$. In questo caso particolare avremo che la cardinalità della formula sarà $\lceil \frac{n+1}{2} \rceil \times \lceil \frac{n+2}{2} \rceil$, andremo infatti a selezionare solamente i nodi interni alla figura. Vediamo di seguito i codici considerando l'input `r` come il raggio del segmento circolare.

```
function xyw = gqcircsegm(n,omega,r)
tw=trigauss(n+2,-omega,omega);

ab=r_jacobi(ceil((n+1)/2),0,0);
xw=gauss(ceil((n+1)/2),ab);

[t,theta]=meshgrid(xw(:,1),tw(1:ceil((n+2)/2),1));
[w1,w2]=meshgrid(xw(:,2),tw(1:ceil((n+2)/2),2));

s=sin(theta(:));
xyw(:,1)=r*cos(theta(:));
xyw(:,2)=r*t(:).*s;
xyw(:,3)=r^2*s.^2.*w1(:).*w2(:);
end
```

Codice `gqcircsegm` in Matlab

```
def gqcircsegm(n, omega, r):
tw = trigauss(n + 2, -omega, omega)

m_r = int(np.ceil((n + 1) / 2))
ab = r_jacobi(m_r, 0, 0)
```

```
    xw = gauss(m_r, ab)

    T, Theta = np.meshgrid(xw[:, 0], tw[1:m_r, 0], indexing='ij')
    W1, W2 = np.meshgrid(xw[:, 1], tw[1:m_r, 1], indexing='ij')

    s = np.sin(Theta).ravel()
    X = (r * np.cos(Theta)).ravel()
    Y = (r * T.ravel() * s).ravel()
    W = (r**2 * (s**2) * W1.ravel() * W2.ravel())

    xyw = np.column_stack([X, Y, W])
    return xyw
```

Codice `gqcircsegm` in Python

Studiamo ora gli errori su quest'ultimo dominio.

n	nodi	$E_{assoluto}$	$E_{relativo}$
5	12	1.13×10^{-16}	5.13×10^{-16}
10	36	2.56×10^{-16}	3.85×10^{-16}
15	72	3.89×10^{-16}	2.18×10^{-15}
20	121	3.41×10^{-16}	6.41×10^{-16}

Tabella 3.9: Errori relativi agli integrali di $g(x, y)$ calcolati sul segmento circolare generato da linear blending, al variare di n.

Gli errori sono nuovamente accettabili e vicini alla precisione macchina, abbiamo quindi trovato un'alternativa più che valida ai programmi Matlab per il calcolo di formule di quadratura Gaussiana. I tempi rimangono leggermente migliori in Matlab ma, analizzando la tabella, vediamo che la differenza è accettabile.

n	$t_p[s]$	$t_m[s]$
5	1.07×10^{-3}	3.31×10^{-4}
10	1.80×10^{-3}	3.06×10^{-4}
15	5.69×10^{-3}	3.41×10^{-4}
20	7.22×10^{-3}	4.55×10^{-4}

Tabella 3.10: Tempi medi di applicazione t_p, t_m del programma `gqcircsegm` nei 2 linguaggi (rispettivamente Python e Matlab).

3.1.3 Un terzo dominio di tipo linear blending

L'ultimo esempio consiste nello studio di una corona circolare definita nell'intervallo $[-\pi/6, \pi/6]$ con l'area delimitata dai seguenti archi:

$$P(\theta) = (3\cos(x), 3\sin(x)), \quad Q(\theta) = (\cos(x), \sin(x)), \tag{3.11}$$

come si può vedere dalla seguente immagine.

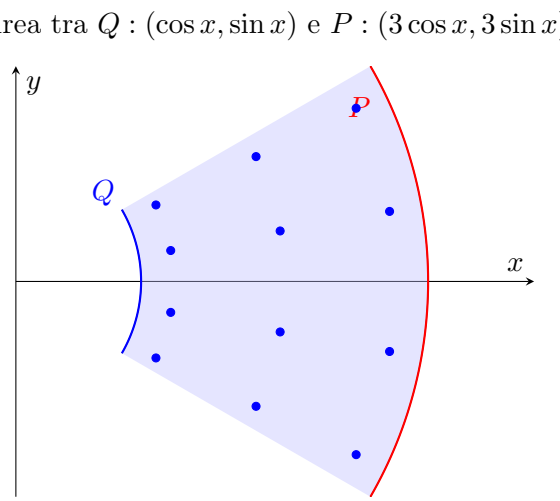


Figura 3.3: Sezione di una corona circolare e nodi di una formula avente ADE pari a 3.

In questo caso torniamo ad avere $h = 1$ e $k = 0$, con cardinalità della formula uguale a $\lceil \frac{n+2}{2} \rceil \times (n+1)$. Utilizzeremo nuovamente il programma `gqcircsect` con però $r_1 = 1,\ r_2 = 3$ e vediamo gli errori.

n	nodi	$E_{assoluto}$	$E_{relativo}$
5	24	2.56×10^{-13}	7.09×10^{-14}
10	66	1.55×10^{-13}	3.98×10^{-14}
15	144	1.88×10^{-13}	8.42×10^{-14}
20	231	1.20×10^{-13}	2.37×10^{-14}

Tabella 3.11: Errori relativi agli integrali di $g(x, y)$ calcolati sulla sezione di corona circolare generato da linear blending, al variare di n.

Quest'ultimo caso, come vediamo, ha errori leggermente maggiori rispetto a quelli precedenti, ma con un errore relativo dell'ordine di 10^{-14} possiamo ritenerci soddisfatti. Verifichiamo come ultima cosa se i tempi di calcolo sono dello stesso ordine del primo esempio, come ci aspettiamo, essendo la stessa routine.

n	$t_p[s]$	$t_m[s]$
5	1.88×10^{-3}	1.77×10^{-3}
10	2.02×10^{-3}	3.092×10^{-4}
15	2.84×10^{-3}	3.23×10^{-4}
20	3.81×10^{-3}	4.30×10^{-4}

Tabella 3.12: Tempi medi di applicazione t_p, t_m del programma `gqcircsect` nei 2 linguaggi (rispettivamente Python e Matlab).

Conclusioni

L'approfondimento che abbiamo fatto in questa tesi relativamente alla traduzione di alcune routines da Matlab in Python ha soddisfatto le aspettative infatti, gli esperimenti numerici hanno evidenziato che le formule prodotte sono equivalenti, con nodi e pesi uguali, a meno di errori dell'ordine della precisione macchina.

Un altro elemento che abbiamo analizzato è il tempo di calcolo. In particolare si é evidenziato che i programmi in Python per quanto rapidi, sono risultati in media leggermente più lenti paragonati agli equivalenti in Matlab.

In sintesi, questo lavoro ha permesso di validare l'approccio numerico e di evidenziare le differenze pratiche tra i vari linguaggi, offrendo così una visione completa sia dal punto di vista teorico sia da quello implementativo.

Bibliografia

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

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

[3] W. Gautschi, *Orthogonal polynomials: applications and computation*, Ada Numerica (1996), pp. 45–119.

[4] W. Gautschi, *Orthogonal polynomials (in Matlab)*, Journal of Computational and Applied Mathematics, Volume 178, Issues 1–2, 1 June 2005, pp.215–234.

[5] G.H. Golub, G.H. Welsch, *Calculation of Gauss Quadrature Rules*, Mathematics of Computation, Vol. 23, No. 106 (1969), pp. 221-230.

[6] R. Piessens, *Modified Clenshaw-Curtis integration and applications to numerical computation of integral transforms*, Numerical integration (Halifax, N.S., 1986), pp.35–51, NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci., 203, Reidel, Dordrecht, 1987.

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

[8] M. Storgato, *UBP:* Python *package for subperiodic trigonometric quadrature and multivariate applications*, `https://github.com/mattiastorgato/TRIGAUSS`.

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