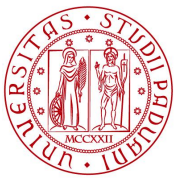


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Corso di Laurea Triennale in Matematica

Cubatura adattiva su domini poligonali

Relatore:
Prof. Alvise Sommariva

Laureando: Sara Alessi
Matricola: 1164363

Anno Accademico 2022/2023

15/12/2023

Indice

Introduzione

Il proposito di questa tesi é di calcolare numericamente l'integrale di una funzione continua su una regione bivariata di tipo poligonale $\mathcal{P}$, con un errore stimato sotto una soglia fissata dall'utente.

Procedure di questo tipo sono state sviluppate su geometrie piú semplici o particolari, come nel rettangolo o nel disco.

In Chebfun é ad esempio possibile calcolare tale integrale su domini come il disco e il quadrato. Nella versione ufficiale di Matlab é presente la routine `integral2`, basata su `TwoD` implementata da Lawrence F. Shampine e presentata nel lavoro *Matlab Program for Quadrature in 2D*. Tale procedura, permette nuovamente il calcolo numerico di integrali su rettangoli e dischi, con una precisione fissata.

D'altro canto, sono ben note formule di cubatura algebriche, con nodi interni e pesi positivi su $\mathcal{P}$, anche a bassa cardinalitá (formule *compresse*). Nel caso del triangolo, in particolare, sono state inoltre determinate formule quasi minimali, per gradi di precisione minori di 20. Mediante queste formule é possibile fornire un'approssimazione dell'integrale richiesto, senza peró adattarsi alla specificitá dell'integranda.

Il proposito di questa tesi, come anticipato, è di presentare un algoritmo adattivo su regioni poligonali generiche, e di verificarne la bontá confrontando i risultati con formule di cubatura classiche e compresse.

Nel primo capitolo, dopo una definizione sulle formule di cubatura, vengono presentate le regole di cubatura compresse e di seguito la possibilità che si ha, grazie a queste, di trattare domini poligonali $\mathcal{P}$ di tipo generale. In prima istanza, illustriamo la tecnica basata sulla triangolazione di $\mathcal{P}$. A tal proposito, utilizzeremo una triangolazione di tipo minimale di $\mathcal{P}$ (ovvero che richiede un numero minimo di triangoli). Di seguito su ogni triangolo utilizzeremo una regola quasi-minimale, ottenendo facilmente una formula composta su $\mathcal{P}$.

Nel secondo capitolo viene trattata la teoria sulla cubatura adattiva e viene descritto l'algoritmo per il calcolo approssimato di integrali con formule adattive su domini poligonali. Saranno qui introdotte le principali funzioni Matlab che verranno poi utilizzate nei test numerici.

Il terzo capitolo mostra attraverso figure e tabelle, i risultati numerici ottenuti applicando l'algoritmo adattivo descritto nel capitolo precedente a domini poligonali convessi, concavi o non semplicemente connessi.

Infine faremo uno studio numerico su queste formule, nell'ambito di domini poligonali non semplicemente connessi, di interesse in ottica.

Capitolo 1

Formule di cubatura algebriche su poligoni

Il proposito di una formula di cubatura consiste nell'approssimare un integrale definito mediante la somma pesata di valutazioni di una funzione calcolati in un numero finito di punti.

A tal proposito, sia $\Omega \subset \mathbb{R}^d$ un dominio di integrazione, che supporremo compatto, $f \in C(\Omega)$ una funzione continua. Indicheremo di seguito con $\mathbb{P}_n$ l'insieme dei polinomi bivariati aventi grado totale al piú n, ovvero del tipo

$$p(x,y) = \sum_{i=0}^{n} \sum_{j=0}^{n-i} a_{i,j} x^i y^j,$$

per opportune scelte di $a_{i,j} \in \mathbb{R}$.

Definita la formula di cubatura

$$Q(f) := \sum_{i=1}^{n} w_i f(x_i)$$

avente *pesi* $w_i \in \mathbb{R}$ e *nodi* $x_i \in \mathbb{R}^d$, si desidera che questa approssimi l'integrale definito

$$I(f) = \int_{\Omega} f(x) dx.$$

In questa direzione, risulta fondamentale definire il *grado di precisione* di Q_n.

Definizione 1.0.1 (Grado di precisione)**.** *Una formula Q ha grado algebrico di precisione almeno δ se se e solo se $Q(f) = I(f)$ per ogni $f \in \mathbb{P}_\delta$. Inoltre ha grado di precisione esattamente δ se e solo se ha grado algebrico di precisione almeno δ ed esiste un polinomio di grado totale $\delta + 1$ per cui non lo sia.*

Presentiamo in questo capitolo un algoritmo che calcoli le formule di cubatura con grado algebrico di precisione $ADE = \delta$ su un poligono $\mathcal{P}$ definito per unione, intersezione o differenza di altri poligoni. A tal proposito bisogna tenere in considerazione che tali poligoni possono avere geometrie complicate potendo essere non semplici, molteplicemente connessi o non connessi.

1.0.1 Determinazione di formule di cubatura

Su classici domini bivariati quale ad esempio il disco, il quadrato, il triangolo, sono presenti varie formule con pesi positivi e nodi interni, in vari casi anche di cardinalitá molto bassa.

Indirizziamo la nostra attenzione a determinare numericamente formule su domini poligonali.

A tal proposito, una strategia suggerita in letteratura consiste essenzialmente nel:

- suddividere opportunamente il poligono $\mathcal{P}$ in sottodomini $\Omega_1, \ldots, \Omega_\mu$ piú semplici da trattare;
- considerare su ogni Ω_k, $k = 1, \ldots, \mu$ una formula di cubatura di tipo PI (ovvero con pesi positivi e nodi interni alla regione $\mathcal{P}$) con $ADE = \delta$, per ottenere una formula composta di cubatura PI con grado di precisione δ sul poligono $\mathcal{P}$;
- se la formula così ottenuta risulta avere un alto numero di nodi, talvolta é utile comprimerla come suggerito teoricamente dal teorema di Caratheodory-Tchakaloff e ottenere una formula PI con al massimo $N_\delta = \frac{(\delta+1)(\delta+2)}{2}$ nodi e pesi positivi.

Seppure questo approccio sembri semplice, ci sono vari dettagli da discutere.

Un classico approccio per determinare una buona regola di cubatura su un poligono $\mathcal{P}$ consiste nel suddividere $\mathcal{P}$ in regioni poligonali più facili da trattare $\Omega_1, \ldots, \Omega_\nu$, ad esempio attraverso triangolazione o quadrangolazione, e di seguito applicare una nota formula a bassa cardinalità su ciascuna di esse.

Nel caso della triangolazione, $\mathcal{P}$ viene descritto come unione di triangoli $\Omega_1, \ldots, \Omega_\nu$, tali che $\mathcal{P} = \cup_{i=1}^{\nu} \Omega_i$, e inoltre se $i \neq j$ allora $\Omega_i \cap \Omega_j$ é nulla o al piú un vertice o un lato di Ω_i.

Similmente, $\Omega_1, \ldots, \Omega_\nu$ é una quadrangolazione di $\mathcal{P}$, se Ω_k, $k = 1, \ldots, \mu$ é un quadrangolo, $\mathcal{P} = \cup_{j=1}^{\mu} \Omega_j$ e se $i \neq j$ allora $\Omega_i \cap \Omega_j$ é nulla o al piú unione di vertici o lati di Ω_i.

Si vede facilmente che ogni poligono convesso è *triangolabile* mediante $\nu = (n-2)$ triangoli fissando un vertice e collegandolo ai suoi vertici non adiacenti.

Dal punto di vista della cubatura numerica, per ragioni di complessitá computazionale, si può ricorrere ad algoritmi di triangolazione o quadrangolazione minimali, ovvero richiedenti un numero minimo di triangoli o quadrilateri.

E' noto infatti che qualsiasi poligono semplice o semplicemente connesso con n vertici può essere triangolato in $\nu = n - 2$ triangoli o $\frac{n-2}{2} \leq \nu \leq n - 2$ quadrilateri, con ν spesso vicino al limite inferiore.

In Matlab viene fornito l'ambiente `polyshape` che permette tra le sue varie routines, pure di triangolare poligoni molto generali. Ammette anche l'utilizzo di operazioni booleane come l'intersezione, la differenza, l'unione e la differenza simmetrica tra poligoni. Per esempio in [1] è stata testata la qualità della triangolazione Ω_i su domini poligonali

molto differenti con n lati, ottenendo $\mathcal{P} = \bigcup_{i=1}^{\nu} \Omega_i$ con $\nu \approx n$ oppure $\nu = n - 2$ per triangolazioni minimali.
Come regioni, dati n angoli equispaziati in $[0, 2\pi]$ detti $t_k = \frac{2\pi k}{n}$ con $k = 1, ..., n$, sono state prese:

- un poligono regolare $\mathcal{P}^{(1)}$ con vertici

  $$v_k = (cos(t_k), sin(t_k))$$

  sulla circonferenza unitaria di centro l'origine;

- una cardioide poligonale $\mathcal{P}^{(2)}$ con vertici

  $$v_k = (cos(t_k)(1 - cos(t_k)), sin(t_k)(1 - cos(t_k))$$

  ottenuti dall'equazione parametrica generale della cardioide, definita come

  $$\begin{cases} x = acos(1 + cos(\theta)) \\ y = asin(1 + cos(\theta)) \end{cases}$$

  con $\theta \in [0, 2\pi]$ e $a > 0$;

- la Lemniscata di Bernoulli poligonale $\mathcal{P}^{(3)}$ con vertici

  $$v_k = (\frac{\sqrt{2}cos(t_k)}{1 + sin^2(t_k)}, \frac{\sqrt{2}cos(t_k)sin(t_k)}{1 + sin^2(t_k)})$$

  ottenuti dall'equazione cartesiana: $(x^2 + y^2)^2 - 2a^2(x^2 + y^2) = 0$ con $a \in \mathbb{R}$. La Lemniscata è definita come il luogo dei punti del piano per i quali il prodotto delle distanze da due punti fissi $(a, 0)$ e $(-a, 0)$ è uguale ad a^2.

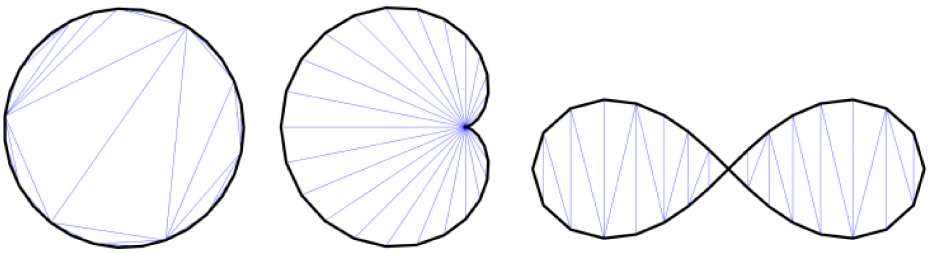


Figura 1.1: Triangolazione dei poligoni $\mathcal{P}^{(1)}, \mathcal{P}^{(2)}, \mathcal{P}^{(3)}$ per n=32

La Lemniscata di Bernoulli possiede un punto doppio (nodo) nell'origine, quindi non è un dominio semplice, ma può essere ugualmente trattata correttamente per $n = 1000$. A partire da $n = 2000$ il numero di triangoli della Lemniscata di Bernoulli diventa inferiore a $n - 4$ che corrisponde alla cardinalità della triangolazione minimale.

1.1 Determinazione di regole quasi minime

Data una suddivisione $\mathcal{P} = \bigcup_{i=1}^{n} \Omega_i$, per avere una regola di cubatura PI su $\mathcal{P}$, per l'additività dell'integrale, è sufficiente definire una regola PI su ogni Ω_i.

Ogni triangolo è biunivocamente mappato in qualsiasi altro triangolo tramite trasformazione affine. Conoscendo quindi una regola di cubatura su un triangolo di riferimento $\mathcal{T}^*$, dopo la conversione dei nodi in coordinate baricentriche, è semplice ottenere un altro triangolo $\mathcal{T}$ variando i pesi proporzionalmente alla loro area.
Esempio 1. Sia $\mathcal{T}^*$ il triangolo di vertici (0,0), (0,1), (1,0) e $\phi_{k\,k=1,...,N_k}$ una base dello spazio dei polinomi bivariati $\mathbb{P}_\delta$ con grado totale non superiore a δ, cardinalità $N_\delta = \frac{(\delta+1)(\delta+2)}{2}$ e tale che

$$\int_{\mathcal{T}^*} \phi_k(x)dx = \sum_{i=1}^{m} w_i^* \phi_k(\xi_i^*), \xi_i^* \in \mathcal{T}^*, \quad w_i^* > 0, k = 1, ..., N_\delta \tag{1.1}$$

cioè tale che la regola di cubatura PI (ξ_i^*, w_i^*) abbia $ADE = \delta$.
Sia ora $\xi_i^* = (x_i^*, y_i^*)$ e si denotino con $(x_i^*, y_i^*, 1 - x_i^* - y_i^*)$ le coordinate baricentriche di ξ_i^*, $\mu(\mathcal{T})$ l'area di $\mathcal{T}$ e V_1, V_2, V_3 i vertici di $\mathcal{T}$.
Essendo $\mu(\mathcal{T}^*) = 1/2$, la regola di cubatura su $\mathcal{T}$ con nodi:

$$\xi_i = x_i^* V_1 + y_i^* V_2 + (1 - x_i^* - y_i^*)V_3, \quad i = 1, ..., m \tag{1.2}$$

e pesi:

$$w_i = \frac{\mu(\mathcal{T})}{\mu(\mathcal{T}^*)} w_i^*, \quad i = 1, ..., m \tag{1.3}$$

ha $ADE = \delta$ su $\mathcal{T}$, cioè integra esattamente ogni polinomio in $\mathbb{P}_\delta$ su $\mathcal{T}$.

Alla luce di questo risultato è necessario calcolare regole di cubatura con un certo ADE solo sul triangolo di riferimento $\mathcal{T}^*$. A tal proposito utilizziamo regole minimali che, tra le regole PI con $ADE = \delta$, sono quelle con il minor numero di nodi. Nonostante esistano poche regole minimali multivariate, la bibliografia sulle regole PI a bassa cardinalità per un ADE fissato, è più ampia. Queste sono chiamate *regole quasi minime*.
Una tecnica per determinarle è la seguente.

Se le basi dei momenti sono noti: $\gamma_k = \int_{\mathcal{T}^*} \phi_k(x)dx, k = 1, ..., N_\delta$ si calcolano le soluzioni (ξ_i, w_i) con $i = 1, ..., m$ del problema non lineare 1.1 e $m \leq N_\delta$ più piccolo possibile, attraverso ottimizzazione numerica.
Questo non è semplice, soprattutto quando il numero di equazioni N_δ è grande.
Al fine di ridurre il numero di equazioni, le simmetrie dei nodi permettono di cercare l'insieme ottimale in una famiglia meno generale, nonchè di risolvere sistemi lineari più piccoli.

Ricordiamo che è possibile utilizzare qualsiasi triangolazione per determinare una regola PI sul poligono $\mathcal{P}$, ma una triangolazione minimale ha il vantaggio di mantenere la cardinalità più bassa possibile, dato che in ogni triangolo viene utilizzato un numero fisso di punti.
In tabella sono elencate le migliori regole quasi minime utilizzate per implementare una formula PI con $ADE = \delta$ su qualsiasi triangolo o su un poligono triangolarizzato $\mathcal{P}$.

δ	N^*_δ	δ	N^*_δ	δ	N^*_δ	δ	N^*_δ	δ	N^*_δ
1	1	11	27	21	85	31	181	41	309
2	3	12	32	22	93	32	193	42	324
3	4	13	36	23	100	33	204	43	339
4	6	14	42	24	109	34	214	44	354
5	7	15	46	25	117	35	228	45	370
6	11	16	52	26	130	36	243	46	385
7	12	17	57	27	141	37	252	47	399
8	16	18	66	28	150	38	267	48	423
9	19	19	70	29	159	39	282	49	435
10	24	20	78	30	171	40	295	50	453

Figura 1.2: Cardinalità N^*_δ delle regole quasi minime su triangoli con $ADE = \delta$

1.2 Sottocampionamento di Caratheodory-Tchakaloff

Mostriamo ora come da una regola con alta cardinalità, pesi positivi e $ADE = \delta$, possiamo estrarre una regola con lo stesso grado di esattezza, pesi positivi, ma con cardinalità al più uguale alla dimensione N_δ dello spazio polinomiale $\mathbb{P}_\delta$.

Per la compressione di misure discrete viene applicato un algoritmo denominato CATCH (acronimo di Caratheodory-Tchakaloff). Questo permette di determinare regole di cubatura su poligoni non semplici, come la Lemniscata di Bernoulli.

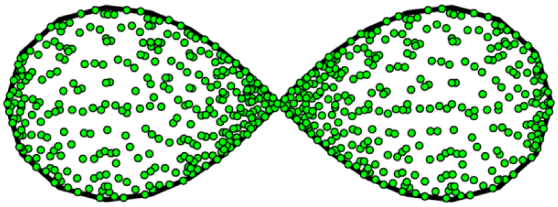


Figura 1.3: Regola di cubatura con grado di precisione $\delta = 10$. Il numero di punti è 672, molto più grande di 66, cioè della dimensione dello spazio polinomiale $\mathbb{P}_{10}$.

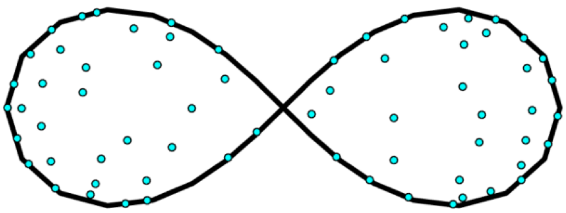


Figura 1.4: Regola di cubatura con grado di precisione $\delta = 10$. Dopo la compressione, il numero di punti è 66, molto più piccolo di 672.

L'algoritmo CATCH seleziona da un'ampia discretizzazione di una data regione, un numero molto piccolo di punti di campionamento (pesati) su forme complesse come i

poligoni, mantenendo numericamente invariate le stime di approssimazione dei minimi quadrati.

Lo strumento teorico chiave che utilizziamo è la versione discreta del teorema di Tchakaloff. Essa afferma che per ogni misura a supporto compatto, esiste una formula di cubatura algebrica positiva con cardinalità non superiore alla dimensione del grado di esattezza dello spazio polinomiale.

Teorema 1.2.1. *Sia μ una misura discreta multivariata il cui supporto è un insieme $X = P_i \subset \mathbb{R}, i = 1, ..., M$ con corrispondenti pesi positivi $\lambda = \lambda_i, i = 1, ..., M, M = card(X) > N_\delta$ e $N_\delta = dim({\mathbb{P}_\delta}^d) = \binom{\delta+d}{d}$. Allora, esiste una formula di cubatura per la misura discreta μ, con nodi $Q_j \subset X$ e pesi positivi $w = w_j, j = 1, ..., m$ con $m \leq N_\delta$ tale che*

$$\int_X p(x)d\mu = \sum_{i=1}^{M} \lambda_i p(P_i) = \sum_{j=1}^{m} w_j p(Q_j) \quad \forall p \in {\mathbb{P}_\delta}^d \tag{1.4}$$

Nella cubatura questo metodo può essere così interpretato:
data una regola PI, che chiamiamo (X, λ), con $ADE = \delta$ e cardinalità $M > n_\delta$, possiamo comprimerla in una regola PI con cardinalità non superiore a N_λ e che chiamiamo (T_λ, w).

Per implementare il risultato, data una qualsiasi base polinomiale ϕ_k di $\mathbb{P}_\delta$, definiamo la matrice di Vandermonde

$$V = V_n(X) = \{\phi_j(\xi_i)\}, \quad 1 \leq i \leq M, 1 \leq j \leq N_\delta,$$

sia $\gamma = V^T\lambda$ il vettore dei momenti delle basi polinomiali $\{\phi_j\}$ rispetto alla misura discreta originale e consideriamo il sistema del momento sottodeterminato $V^T u = \gamma$. Il Teorema 1.2.1 afferma che esiste una soluzione sparsa non negativa u^* di questo sistema, i cui pesi sono al massimo N_δ, e determina i corrispondenti punti di campionamento ridotti $T_\delta = \{t_j\} \subseteq X$ che definiamo punti di Caratheodory-Tchakaloff (CATCH) di X.

Per ottenere queste regole compresse esistono 2 diversi approcci:

- tramite Programmazione Lineare (LP);
- tramite Programmazione Quadratica (QP).

Programmazione Lineare:
Consiste nel risolvere il sistema

$$\text{LP} : \begin{cases} \min c^T u \\ V^T u = \gamma, \quad u \geq 0 \end{cases}$$

in cui i vincoli identificano un politopo in $\mathbb{R}^M$ e il vettore c è scelto essere linearmente indipendente dalle righe di V^T, così la funzione da minimizzare non è costante nel politopo. La soluzione del problema di Programmazione Lineare è un vertice del politopo che può

essere calcolato con il metodo del simplesso. Le componenti non nulle di tale vertice restituiscono i pesi e gli indici dei nodi interni a X.

Programmazione Quadratica:
Richiede la soluzione del problema dei Minimi Quadrati Non Negativi (NNLS)

$$\text{NNLS} : \begin{cases} \min \|V^T u - \gamma\|_2 \\ u \geq 0 \end{cases}$$

La soluzione può essere ottenuta attraverso l'algoritmo di Lawson-Hanson che cerca una soluzione sparsa. Ancora, le componenti non nulle di tale soluzione restituiscono i pesi e gli indici dei nodi interni a X.
L'applicazione di questo algoritmo porta ad un residuo $\epsilon = \|V^T u^* - \gamma\|_2 \leq 10^{-14}$ per $\delta \leq 30$, cioè estremamente piccolo.

Capitolo 2

Cubatura adattiva

Il proposito di questa sezione é di analizzare il codice per la cubatura adattiva di un dominio poligonale, argomento di questo lavoro.

A tale scopo, data una regione poligonale mediante la successione dei suoi vertici, triangoliamo il poligono fornendo i vertici della triangolazione e la lista delle connessioni di ogni vertice. In altre parole descriviamo ogni triangolo attraverso il legame che esiste tra i suoi lati e i suoi vertici.

Per ogni triangolo possiamo calcolare gli integrali approssimati con alta e bassa precisione, ottenendo quindi due diverse approssimazioni dell'integrale e una stima dell'errore.

L'integrale approssimato della regione poligonale si ottiene, per additività dell'integrale, come somma degli integrali approssimati calcolati su ogni triangolo. L'errore totale è la differenza in valore assoluto tra la somma degli integrali approssimati a bassa precisione e la somma degli integrali approssimati ad alta precisione dei triangoli. Se questo errore non è inferiore alla soglia stabilita cerchiamo quale triangolo della triangolazione ha l'errore massimo e procediamo con il raffinamento. Cioè suddividiamo il triangolo con una nuova triangolazione e per ogni nuovo triangolo ottenuto calcoliamo gli integrali ad alta e bassa precisione e gli errori.
Ripetiamo questa tecnica fino al raggiungimento della precisione stabilita.

Per fare questo definiamo una funzione con i seguenti input:

- i vertici `variabile del vertice` del dominio poligonale che scriviamo sotto forma di una matrice $M \times 3$, in cui le righe k-esime rappresentano le coordinate cartesiane k-esime dei vertici della regione poligonale; l'ordine di scrittura dei vertici del poligono è antiorario e il primo e l'ultimo vertice devono essere diversi;
- una integranda `f`;
- una tolleranza assoluta `tol`;
- un numero massimo di raffinamenti pari a 5000;

ottenendo al termine della procedura come output:

- l'approssimazione dell'integrale della funzione sui triangoli;
- l'integrale con precisione alta che chiamiamo `IH`;
- l'integrale con precisione più bassa che chiamiamo `IL`.

Tra gli output opzionali:

- un `flag`, che indica se l'esecuzione è andata a buon fine e il codice ha fornito un risultato;
- il numero di iterazioni totali `iter`, cioè quanti raffinamenti successivi ha dovuto compiere la funzione.

Il codice, qualora non stabilito, fissa di default una tolleranza `tol` pari a 10^{-6}.

La funzione polyshape

Il codice utilizza l'ambiente Matlab `polyshape` per determinare la triangolazione minimale della regione poligonale, che risulta essere dominio di integrazione e di conseguenza reputiamo importante descrivere le proprietá di tale toolbox.

La funzione `polyshape` é stata introdotta nella versione Matlab R2017b, e definisce domini poligonali a partire da vertici dati. A differenza della definizione classica di poligono, `polyshape` può creare anche regioni discontinue e buchi dato che le proprietà di un oggetto `polyshape` descrivono i suoi vertici, le regioni solide e i possibili buchi. Vedremo nell'ultimo capitolo un'applicazione di questo tipo al design ottico.

Come primo semplice esempio illustrativo consideriamo il quadrato unitario avente vertici (0,0), (1,0), (1,1), (0,1). Applicando la funzione `polyshape` si crea il seguente oggetto:

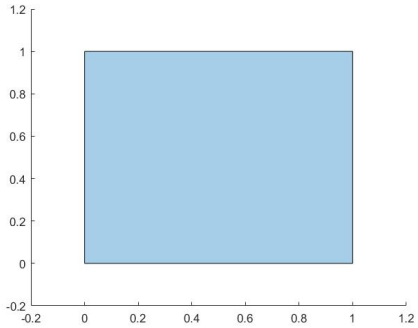


Figura 2.1: Quadrato unitario ottenuto applicando la funzione `polyshape`.

con le seguenti proprietà:

- i vertici sono disposti in una matrice di dimensione 4×2;
- il numero di regioni totali è 1 e il numero di buchi è 0.

Consideriamo il caso piú complesso di un poligono non semplicemente connesso. In questo caso `polyshape` definisce correttamente tanto la frontiera esterna quanto quella interna:

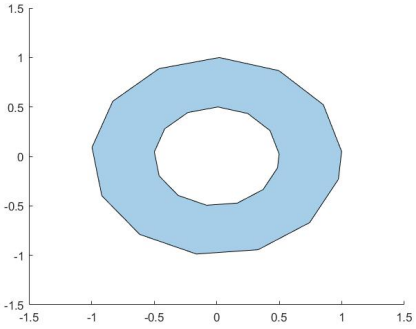


Figura 2.2: Poligono non semplicemente connesso ottenuto applicando la funzione `polyshape`.

Triangolando un poligono attraverso `polyshape` si ottengono proprietà che descrivono i vertici e la connessione dei triangoli. La matrice che rappresenta la lista di connessioni ha le seguenti caratteristiche:

- ogni riga rappresenta un triangolo;
- ogni valore della matrice corrisponde ad un vertice del poligono, quindi se i valori della matrice vanno da 1 a n, il poligono ha n vertici.

A questo punto abbiamo l'elenco dei vertici dei triangoli con cui è stato triangolato il poligono e la lista di connessioni.

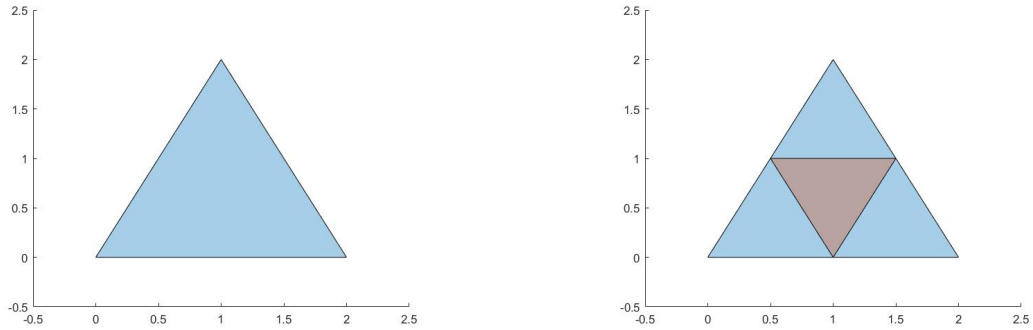
Raffinamento

Illustriamo il processo di raffinamento con cui andiamo ad aggiornare la triangolazione, con l'intento di ottenere un valore dell'integrale richiesto al di sotto della tolleranza `tol` richiesta dall'utente.

Ad ogni iterazione, controlliamo dapprima di non aver usato troppi triangoli (nel nostro esempio é sembrato opportuno settare tale parametro a 5000), cioè di non aver raffinato troppo.

Se cosí non é e l'errore compiuto é superiore alla tolleranza richiesta:

1. si individua il triangolo con l'errore massimo;
2. lo si cancella dalla lista dei triangoli della triangolazione iniziale;
3. si procede a triangolare questo triangolo; a tal proposito utilizziamo come vertici, oltre ai precedenti, i punti medi di ogni lato del triangolo, ottenendo cosí in generale 4 nuovi triangoli;



(a) Triangolo da raffinare

(b) Nuova triangolazione

Figura 2.3: Esempio di triangolazione che utilizza i punti medi di ogni lato del triangolo.

4. per ogni nuovo triangolo così ottenuto calcoliamo come prima i vertici, gli integrali di ordine massimo e minimo e gli errori;

5. aggiorniamo le liste con i nuovi triangoli, i valori degli integrali e la stima dell'errore;

6. aggiorniamo, in virtú di questi contributi, il valore dell'integrale sul dominio poligonale e le stime dell'errore, reiterando se necessario tali operazioni.

Nota 1. Alcuni codici adattivi scritti in Matlab, scelgono di raffinare ad ogni iterazione un certo numero di sottodomini (e non, come nel nostro esempio, uno solo). Questo in generale riduce il numero di iterazioni, ma chiede pure alle routines di agire vettorialmente, per migliorare i tempi di calcolo.

Tale vantaggio ha come problematica la complicazione delle strutture usate e del codice Matlab. In una trattazione futura, si cercherá di seguire tale strada, cercando di minimizzare questi aspetti.

Calcolo delle regole di integrazione

Come anticipato, per calcolare formule composte su un poligono, intendiamo usufruire di formule a bassa cardinalitá in un certo numero di triangoli. A tal proposito nel nostro codice utilizziamo le formule introdotte da Lyness e Jespersen in [3], in forma baricentrica. Queste hanno nodi interni al dominio, pesi positivi, sono simmetriche e hanno bassa cardinalitá, qualitá molto utile in un codice adattativo perché permettono di fare un basso numero di valutazioni della integranda.

Consideriamo come triangolo di riferimento il triangolo unitario di vertici $(0,0)$, $(0,1)$, $(1,0)$ e utilizziamo le coordinate baricentriche poichè possono essere ben adattate ad ogni tipo di triangolo $\mathcal{T}$ con una semplice trasformazione a seconda dei suoi vertici.
Le regole di integrazione possono essere ad alta o bassa precisione; per determinarle operiamo su ogni triangolo $\mathcal{T}$ come segue:

1. calcoliamo l'area di $\mathcal{T}$ attraverso la funzione Matlab `polyarea`, che in generale calcola l'area di un poligono definito tramite i suoi vertici;

2. approssimiamo il valore dell'integrale definito su $\mathcal{T}$ mediante le regole ad alta e bassa precisione; utilizziamo formule nel triangolo di riferimento in cui la somma dei pesi vale 1 per poi essere adattata al triangolo richiesto, moltiplicando i pesi per l'area;

3. per l'integrale a bassa cardinalità utilizziamo la matrice contenente i nodi e i pesi di una formula di Lyness e Jespersen di grado 9; si tratta di una matrice $N \times 4$ in cui le prime tre colonne rappresentano le coordinate cartesiane dei nodi, mentre la quarta colonna rappresenta i pesi;

4. per l'integrale a bassa cardinalità utilizziamo la matrice contenente i nodi e i pesi di una formula di Lyness e Jespersen di grado 11.

Nota 2. Per ottenere meno valutazioni possibili si può adottare la seguente strategia: si fa in modo che la formula di grado più basso abbia dei nodi contenuti nella formula di grado più alto e si procede con tutte le valutazioni per il grado più alto che verranno poi ereditate dalle formule con il grado più basso.

2.1 Descrizione dell'algoritmo

In questa sezione forniamo una traccia dell'implementazione del metodo in forma di codice Matlab.
Il codice ha quali input:

- la matrice dei vertici del poligono: `vertices`;
- la funzione da integrare `f`;
- la tolleranza assoluta `tol`.

E produce come output:

- l'approssimazione di `f` sul triangolo;
- l'approssimazione dell'integrale con alta precisione `IH`;
- l'approssimazione dell'integrale con bassa precisione `IL`.

La procedura effettua le seguente operazioni:

- Fissa il massimo numero di triangoli `max_triangles` = `5000` e `flag` = `0`;
- pone come liste vuote le liste `L1` degli integrali con precisione alta, bassa e la lista degli errori;
- controlla a che classe appartengono i vertici dati in input attraverso la funzione `class(vertices)`:

  - se la classe è di tipo `polyshape` procede con il punto successivo;
  - se la classe non è di tipo `polyshape`, pone `XV` la prima colonna della matrice dei vertici, `YV` la seconda colonna e attraverso la funzione Matlab `polyshape` calcola `PG` = `polyshape`(`XV`, `YV`). A questo punto `PG` = `vertices` sono i nuovi vertici che sono utilizzati come input.

- si triangola `PG`, ponendo `tri` la nuova triangolazione: `tri` = `triangulation`(`PG`) che possiede dei vertici: `tri_vertices=tri.Points` e una lista di connessioni: `tri_conn_list=tri.ConnectivityList`;

- per ogni triangolo, vengono definiti i vertici, un integrale a bassa precisione `L1_integrals_L`, un integrale ad alta precisione `L1_integrals_H` e gli errori commessi `L1_errors`;

- sommando gli integrali a bassa e ad alta precisione dei triangoli si ottengono le approssimazioni dell'integrale totale a bassa e alta precisione: `IL=sum(L1_integrals_L)`; `IH=sum(L1_integrals_H)`; l'errore è `AE=abs(IL-IH)`.

Raffinamento:

- attraverso un ciclo `while`, finchè gli errori relativo e assoluto sono grandi, maggiori della tolleranza `tol`, verifichiamo che il numero dei triangoli usati nella triangolazione sia minore del numero massimo di triangoli ammissibili. Altrimenti il ciclo termina e `flag=1`;

- determiniamo quale triangolo della triangolazione `L1` ha l'errore massimo: `max`(`L1_errors`) e lo chiamiamo `kmax`;

- attraverso la funzione Matlab `setdiff` che calcola la differenza tra insiemi, cancelliamo dalla lista `L1` il triangolo `kmax`;

- per il triangolo `kmax` calcoliamo una nuova triangolazione ottenendo, attraverso la funzione `generate_triangle_sons`, i 4 triangoli figli. Per ogni triangolo figlio calcoliamo i vertici, gli integrali ad alta e bassa precisione e gli errori.

- Aggiorniamo la lista `L1` sostituendo il triangolo con l'errore massimo con i dati della nuova triangolazione.

Capitolo 3

Test Numerici

Il proposito di questa sezione è quello di verificare numericamente la qualitá della procedura di cubatura adattiva.

Faremo vari esempi con diversi domini poligonali e diverse funzioni integrande e controlleremo che i valori degli integrali ottenuti sono gli stessi applicando una formula di cubatura classica e una formula di cubatura compressa.

Utilizziamo in particolare quali regioni:

1. un poligono convesso,
2. un poligono concavo;
3. una regione poligonale non semplicemente connessa (forata).

Per ogni dominio testeremo tre diverse funzioni integrande:

1. la funzione di *Franke* che corrisponde alla somma di quattro esponenziali:

$$f_1(x,y) = \frac{3}{4}e^{-\frac{(9x-2)^2+(9y-2)^2}{4}} + \frac{3}{4}e^{-(\frac{(9x+1)^2}{49}+\frac{(9y+1)}{10})} + \frac{1}{2}e^{-(\frac{(9x-7)^2+(9y-3)^2}{4})} - \frac{1}{5}e^{-((9x-4)^2+(9y-7)^2)} \tag{3.1}$$

2. una funzione *oscillante*:

$$f_2(x,y) = 2\cos(10x)\sin(10y) + \sin(10xy) \tag{3.2}$$

3. una funzione con *bassa regolaritá* (singolaritá di tipo radice):

$$f_3(x,y) = \sqrt{x^2+y^2} \tag{3.3}$$

Osserviamo che mentre le prime due funzioni sono intere, la terza presenta una singolarità di tipo *radice* nell'origine $(0,0)$.

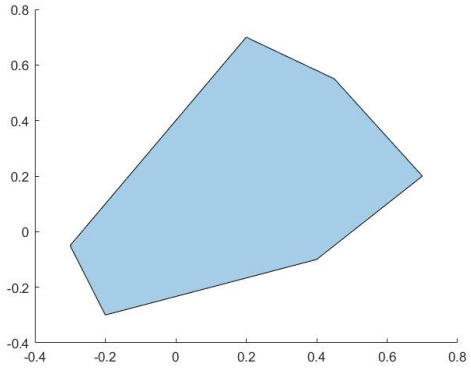
I codici Matlab per i test numerici sono stati testati in un computer con processore *Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz* e RAM da 16,0 GB.

3.1 Dominio convesso

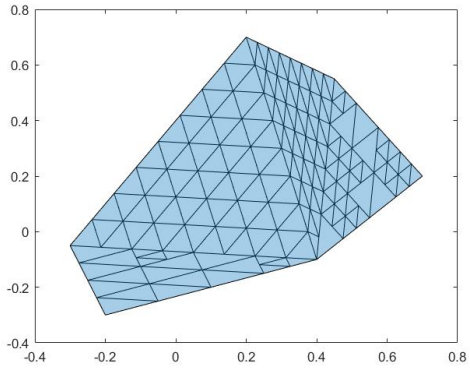
In questo primo dominio $\mathcal{P}_1$ consideriamo un poligono convesso con i seguenti vertici:

1. $P_1 = (-0.2, -0.3)$;
2. $P_2 = (0.4, -0.1)$;
3. $P_3 = (0.7, 0.2)$;
4. $P_4 = (0.45, 0.55)$;
5. $P_5 = (0.2, 0.7)$;
6. $P_6 = (-0.3, -0.05)$.

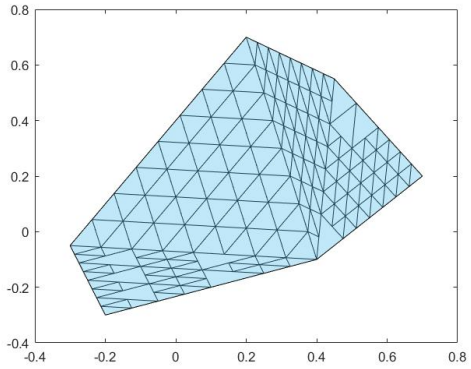
In base alla funzione integranda scelta, l'algoritmo adattivo triangola iterativamente il dominio $\mathcal{P}_1$, come si puó vedere nella Figura 3.1. Notiamo inoltre che il punto di singolarità è interno al dominio e i triangoli sono più densi in quel punto. Questo è ben visibile con la funzione radice $\sqrt{x^2 + y^2}$.



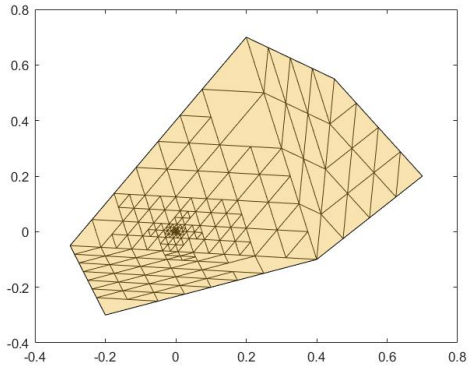
(a) Dominio poligonale convesso.



(b) Triangolazione relativa al calcolo di $I(f_1)$.



(c) Triangolazione relativa al calcolo di $I(f_2)$.



(d) Triangolazione relativa al calcolo di $I(f_3)$.

Figura 3.1

Approssimiamo numericamente l'integrale $I(f_k)$ con la formula di cubatura adattiva, come pure con una formula di cubatura composta e una di tipo compresso, al variare del grado di precisione ADE. Nei test considereremo in particolare i gradi $ADE = 10$, 20, 30, 40 e 50. Listiamo quindi in tabelle i valori ottenuti dai test numerici ed evidenziamo le cifre corrette, relativamente a quelle calcolate dalla procedura adattiva. Osserviamo che quest'ultima determina in realtá due valori dell'integrale. Il primo, `IH`, é una migliore approssimazione di $I(f_k)$ rispetto al secondo `IL`. Al termine della procedura, l'errore assoluto tra `IH` e `IL` é minore della tolleranza richiesta, che nei nostri esempi é posta pari a 10^{-14}. Ci si aspetta quindi che l'errore assoluto tra `IH`, risultato utilizzato nelle tabelle, e $I(f_k)$ sia minore o uguale a 10^{-14}.

Si può osservare che maggiore è il grado di precisione della formula composta, migliore risulta l'approssimazione dell'integrale, qualora l'errore non sia prossimo alla precisione di macchina. E' possibile però perdere della precisione per gradi troppo elevati, soprattutto con formule compresse, per cui non si suggerisce di andare oltre grado 40.

Nelle tabelle indichiamo con `Err.1` l'errore relativo della formula di cubatura classica, mentre con `Err.2` l'errore relativo della formula di cubatura compressa. Otteniamo gli stessi valori dell'integrale nei tre diversi casi e una migliore approssimazione dell'integrale a grado 30.

Anche il numero di iterazioni cambia, come ci si aspetta dalle caratteristiche delle funzioni analizzate: si ottengono 65 iterazioni totali applicando la funzione di Franke, 76 iterazioni applicando la funzione oscillante e 103 con la funzione radice. Dai grafici delle triangolazioni si può notare, infatti, che ci sono delle regioni con un maggior numero di triangoli. Si tratta di regioni che necessitano di un migliore raffinamento dal momento che presentano degli errori elevati, come abbiamo visto nel capitolo precedente. Si rileva che il tempo di esecuzione è dell'ordine di centesimi di secondo e l'errore assoluto è inferiore alla tolleranza imposta.

	Integrale di riferimento:**3.819001153074244e-01**		Errore: 9.16e-15	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	**3.81**8948992600042e-01	**3.81**7692737690221e-01	1.4e-05	3.4e-04
20	**3.8190011**52639768e-01	**3.8190011**69188163e-01	1.1e-10	4.2e-09
30	**3.819001153074**161e-01	**3.81900115306**6505e-01	2.2e-14	2.0e-12
40	**3.81900115307423**e-01	**3.81900115307423**e-01	5.4e-15	5.4e-15
50	**3.81900115307422**e-01	**3.8190011**70509785e-01	5.7e-15	4.6e-09

Tabella 3.1: Dominio convesso, funzione di Franke.

	Integrale di riferimento:**2.649031211251618e-01**		Errore: 9.99e-16	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	**2.64**7772918461133e-01	**2.6**64180664307604e-01	4.8e-04	5.7e-03
20	**2.6490312112**44600e-01	**2.649031210**614460e-01	2.6e-12	2.4e-10
30	**2.649031211251618e-01**	**2.64903121125516**46e-01	2.1e-16	1.0e-14
40	**2.64903121125516**20e-01	**2.64903121125516**43e-01	4.2e-16	9.4e-15
50	**2.64903121125161**9e-01	**2.649031**231923774e-01	2.1e-16	7.8e-09

Tabella 3.2: Dominio convesso, funzione integranda oscillante.

	Integrale di riferimento:**1.925059338438616e-01**		Errore: 7.77e-16	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	**1.92**1295495303242e-01	**1.92**2391575749928e-01	2.0e-03	1.4e-03
20	**1.92**4409264334022e-01	**1.92**4405691464555e-01	3.4e-04	3.4e-04
30	**1.9250**37694022266e-01	**1.9250**36347090470e-01	1.1e-05	1.2e-05
40	**1.9250**95025774439e-01	**1.925**102931223703e-01	1.9e-05	2.3e-05
50	**1.9250**78457494181e-01	**1.9250**78739045346e-01	9.9e-06	1.0e-05

Tabella 3.3: Dominio convesso, funzione con singolaritá di tipo *radice* f_3.

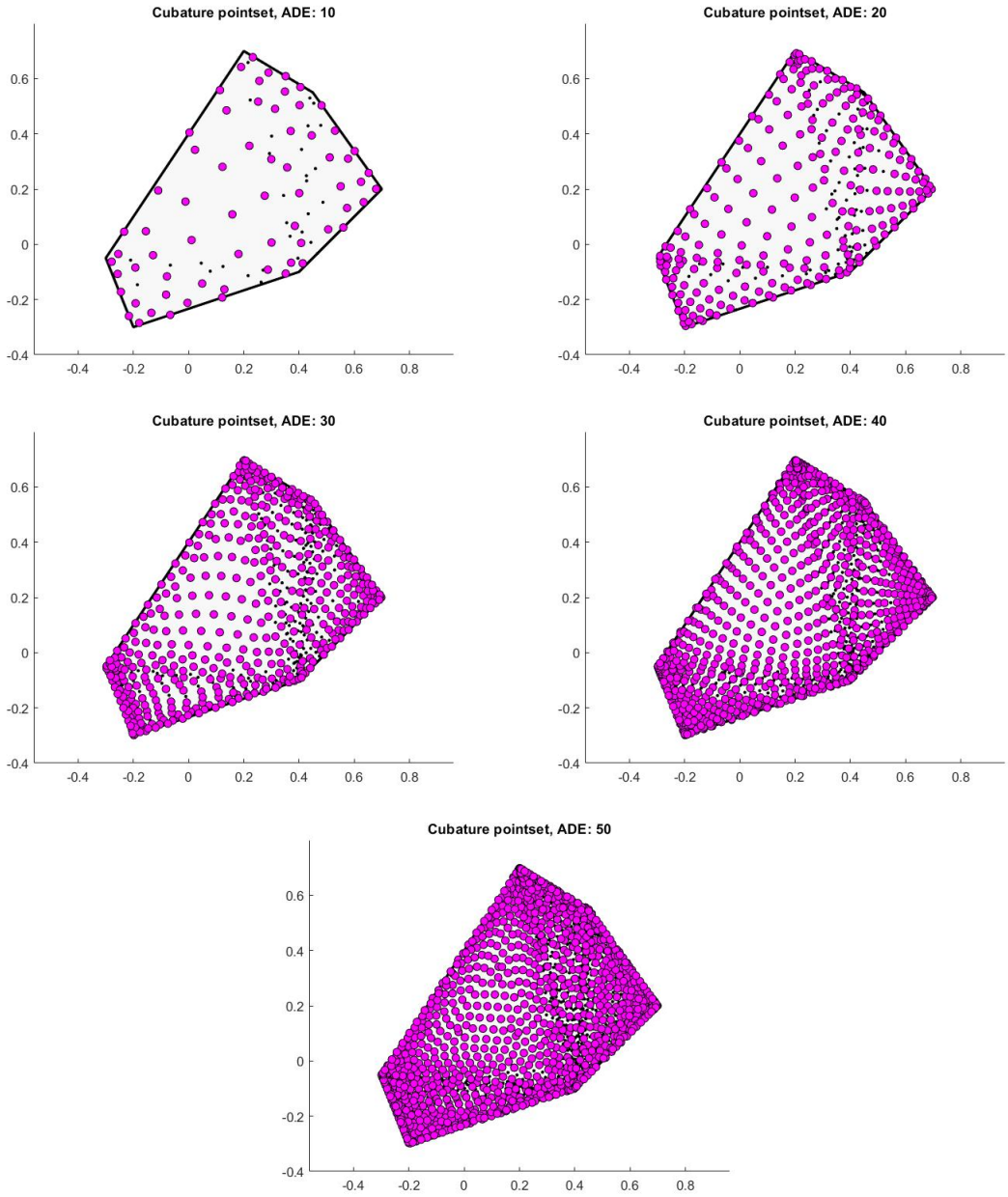


Figura 3.2: Set di nodi di cubatura da grado 10 a grado 50, relativi al poligono convesso $\mathcal{P}_1$. I puntini in nero sono i nodi della formula composta, in magenta quelli della formula compressa.

Nella figura 3.2 abbiamo illustrato come cambia il set di punti di cubatura al variare del grado di precisione ADE della formula.

I puntini neri rappresentano i punti di cubatura della formula composta, mentre i cerchietti magenta rappresentano i nodi della formula di cubatura compressa.

3.2 Dominio concavo

Consideriamo ora una regione concava $\mathcal{P}_2$ con vertici:

1. $P_1 = (-0.5, -0.3)$;
2. $P_2 = (0.45, 0.2)$;
3. $P_3 = (0.45, -0.3)$;
4. $P_4 = (0.45, 0.45)$;
5. $P_5 = (0.45, 0.55)$;
6. $P_6 = (0.2, 0.7)$;
7. $P_7 = (-0.3, 0.45)$;
8. $P_8 = (-0.05, 0.2)$.

Come in precedenza consideriamo come integrande:

1. la funzione di *Franke* che corrisponde alla somma di quattro esponenziali:

$$f_1(x,y) = \frac{3}{4}e^{-\frac{(9x-2)^2+(9y-2)^2}{4}} + \frac{3}{4}e^{-(\frac{(9x+1)^2}{49}+\frac{(9y+1)}{10})} + \frac{1}{2}e^{-(\frac{(9x-7)^2+(9y-3)^2}{4})} - \frac{1}{5}e^{-((9x-4)^2+(9y-7)^2)} \tag{3.4}$$

2. una funzione *oscillante*:

$$f_2(x,y) = 2\cos(10x)\sin(10y) + \sin(10xy) \tag{3.5}$$

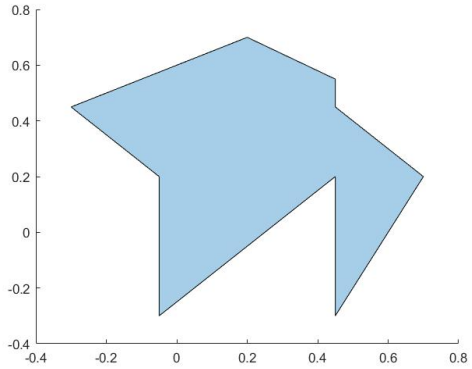
3. una funzione con *bassa regolaritá* (singolaritá di tipo radice):

$$f_3(x,y) = \sqrt{x^2+y^2}. \tag{3.6}$$

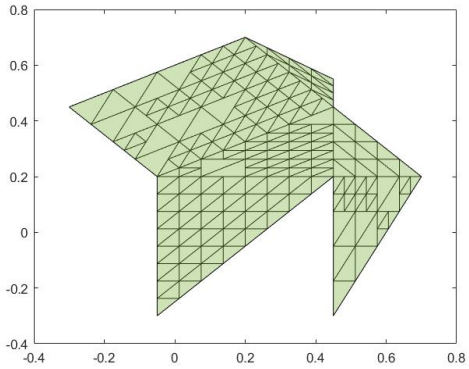
Anche in questo caso il punto di singolarità della terza funzione, ovvero $(0,0)$ è interno al dominio $\mathcal{P}_2$.

Si ha una buona approssimazione dell'integrale nei casi della funzione di Franke f_1 e della funzione oscillante f_2 mentre, come ci si aspetta, la precisione è minore per la funzione con singolaritá di tipo *radice* f_3. Per la funzione f_1 abbiamo ottenuto il risultato dopo 91 iterazioni, per la funzione f_2 dopo 114 iterazioni e per la funzione f_3 dopo 95 iterazioni. Quindi anche per il dominio concavo il codice termina correttamente, senza superare il massimo numero di triangoli ammessi.

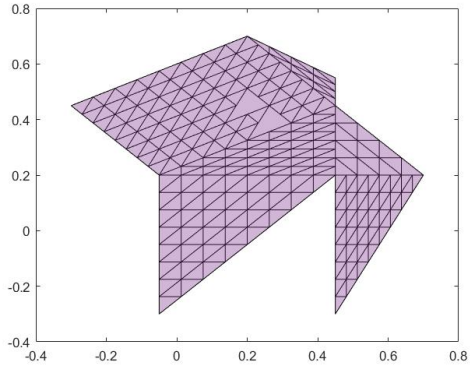
Il tempo di esecuzione rimane dell'ordine di centesimi di secondo e l'errore assoluto è ancora inferiore alla tolleranza.



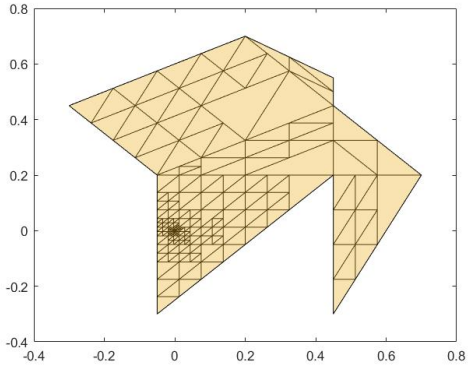
(a) Dominio poligonale convesso.



(b) Triangolazione relativa al calcolo di $I(f_1)$.



(c) Triangolazione relativa al calcolo di $I(f_2)$.



(d) Triangolazione relativa al calcolo di $I(f_3)$.

Figura 3.3

	Integrale di riferimento:**3.206839363924226e-01**		Errore: 5.72e-15	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	**3.2068**44940717878e-01	**3.20**7557194572831e-01	1.7e-06	2.2e-04
20	**3.206839363924**965e-01	**3.206839**478772964e-01	2.3e-13	3.6e-08
30	**3.206839363924225**e-01	**3.20683936**4134780e-01	3.5e-16	6.6e-11
40	**3.206839363924225**e-01	**3.206839363924200**e-01	1.7e-16	8.1e-15
50	**3.206839363924225**e-01	**3.206839363924265**e-01	3.5e-16	1.2e-14

Tabella 3.4: Dominio concavo, funzione di Franke f_1.

	Integrale di riferimento:**1.671899128627978e-01**		Errore: 9.38e-15	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	**1.6718**72867652980e-01	**1.6**64905018643162e-01	1.6e-05	4.2e-03
20	**1.671899128627**969e-01	**1.67189912**8897964e-01	5.0e-15	1.6e-10
30	**1.671899128627977**e-01	**1.671899128627**991e-01	3.3e-16	7.6e-15
40	**1.671899128627977**e-01	**1.671899128627**836e-01	1.7e-16	8.5e-14
50	**1.671899128627977**e-01	**1.671899128627**858e-01	3.3e-16	7.2e-14

Tabella 3.5: Dominio concavo, funzione oscillante f_2.

	Integrale di riferimento:**2.003977155678807e-01**		Errore: 8.13e-15	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	**2.00**2839202901461e-01	**2.00**2316879215180e-01	5.7e-04	8.3e-04
20	**2.0039**33337776805e-01	**2.0039**37555785518e-01	2.2e-05	2.0e-05
30	**2.0039**88397739829e-01	**2.00397**1331017665e-01	5.6e-06	2.9e-06
40	**2.0039**81788348605e-01	**2.0039**84945031053e-01	2.3e-06	3.9e-06
50	**2.00397**4079043856e-01	**2.00397**2849236441e-01	1.5e-06	2.1e-06

Tabella 3.6: Dominio concavo, funzione con singolaritá di tipo *radice* f_3.

Il metodo compresso applicato alla funzione oscillante fornisce una buona approssimazione dell'integrale fino a grado di precisione 30.

In termini di tempo, la compressione non è troppo costosa per gradi bassi, ma diventa rilevante (dell'ordine di 10 secondi) per gradi maggiori di 30.

Abbiamo visto quindi che sia per il dominio convesso che per il dominio concavo, le formule con grado δ di precisione prefissato forniscono al crescere di δ un risultato prossimo a quelle delle formule adattive. Questo fatto é rilevante perché é un indice della *soliditá* della formula adattiva introdotta in questo lavoro.

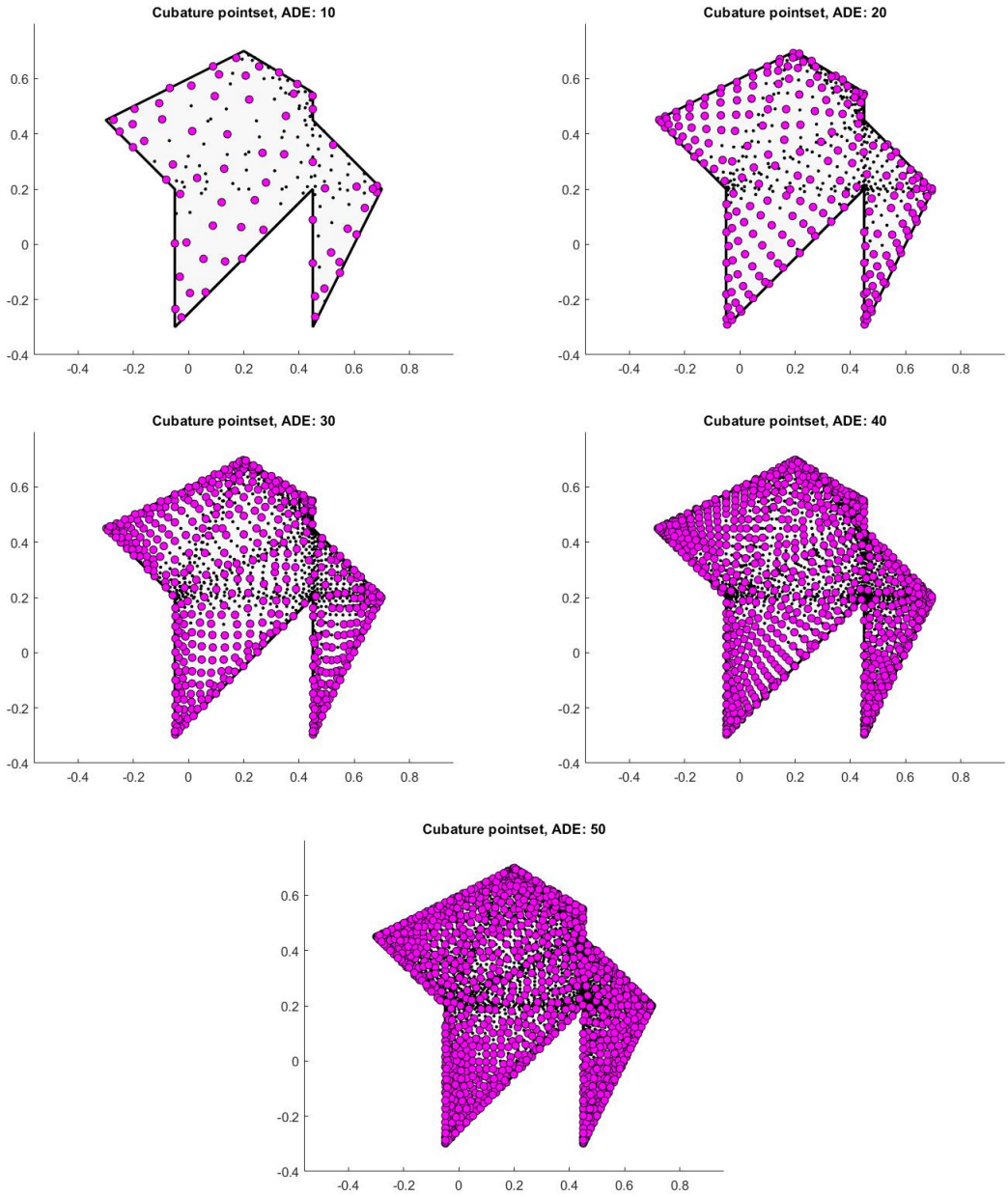


Figura 3.4: Set di nodi di cubatura da grado 10 a grado 50, relativi al poligono concavo $\mathcal{P}_2$. I puntini in nero sono i nodi della formula composta, in magenta quelli della formula compressa.

3.3 Dominio non semplicemente connesso

In questo ultimo test consideriamo un dominio poligonale (non semplicemente connesso) $\mathcal{P}_3$ definito come la differenza tra una regione poligonale esterna con vertici:

1. $P_1 = (1, 0)$;
2. $P_2 = (0.766, -0.6428)$;
3. $P_3 = (0.1736, -0.9848)$;
4. $P_4 = (-0.5, -0.866)$;
5. $P_5 = (-0.9397, -0.342)$;
6. $P_6 = (-0.9397, 0.342)$;
7. $P_7 = (-0.5, 0.866)$;
8. $P_8 = (0.1736, 0.9848)$;
9. $P_9 = (0.766, 0.6428)$

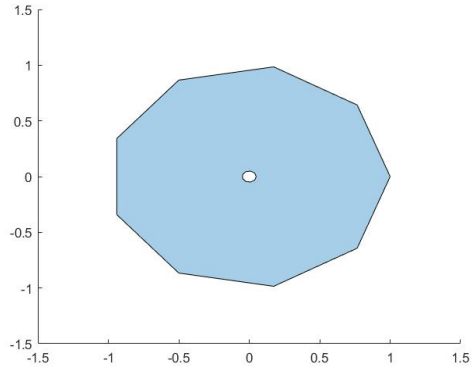
e una regione poligonale interna con vertici:

1. $Q_1 = (0.05, 0)$;
2. $Q_2 = (0.0383, 0.0321)$;
3. $Q_3 = (0.0087, 0.0492)$;
4. $Q_4 = (-0.025, 0.0433)$;
5. $Q_5 = (-0.047, 0.0171)$;
6. $Q_6 = (-0.047, -0.0433)$;
7. $Q_7 = (-0.025, -0.0433)$;
8. $Q_8 = (0.0087, -0.0492)$;
9. $Q_9 = (0.0383, -0.0321)$.

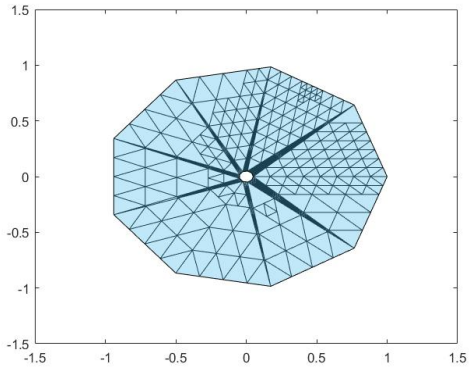
Le due regioni sono state definite dalla funzione `Polyshape` e il punto di singolarità $(0, 0)$ di f_3, seppure non interno, è molto vicino al dominio $\mathcal{P}_3$.

Le iterazioni necessarie per il calcolo di integrali nel caso della funzione f_1 sono 181, 574 per la funzione f_2 e 715 per la funzione f_3 e la procedura adattiva termina correttamente.

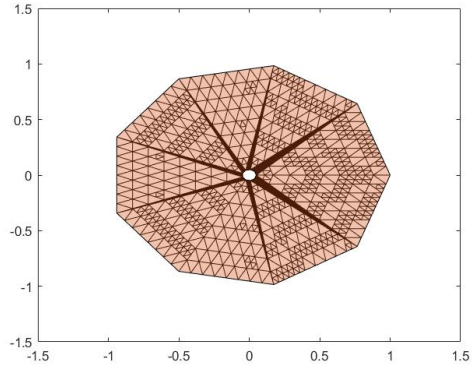
Osserviamo però che porre la singolarità nell'origine, cioè fuori dal dominio poligonale, rende comunque il problema difficile numericamente. Illustriamo i risultati di quest'ultimo esperimento nella Tabella 3.9.



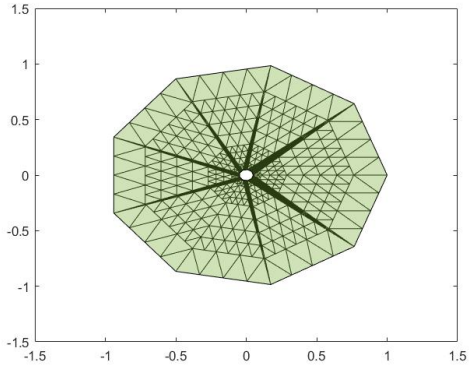
(a) Dominio poligonale convesso



(b) Triangolazione relativa al calcolo di $I(f_1)$.



(c) Triangolazione relativa al calcolo di $I(f_2)$.



(d) Triangolazione relativa al calcolo di $I(f_3)$.

Figura 3.5

	Integrale di riferimento:**1.726905103143922e+00**		Errore: 1.38e-14	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	**1.726**875458525239e+00	**1.7**31428976403848e+00	1.7e-05	2.6e-03
20	**1.72690**5043144981e+00	**1.72**7608428780175e+00	3.5e-08	4.1e-04
30	**1.726905103**097045e+00	**1.726**826009009623e+00	2.7e-11	4.6e-05
40	**1.726905103144**235e+00	**1.72690**6520382193e+00	1.8e-13	8.2e-07
50	**1.726905103144**239e+00	**1.726905**112438873e+00	1.8e-13	5.4e-09

Tabella 3.7: Dominio non semplicemente connesso, funzione di Franke

	Integrale di riferimento:**-1.945058697438995e-16**		Errore: 9.82e-15	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	-1.542209382003612e-05	-1.139098559872917e-01	1.54e-05	1.14e-01
20	-1.193364851381773e-12	-1.946908226503341e-04	1.19e-12	1.95e-04
30	-1.945058697438995e-16	-1.001022181812417e-14	1.94e-16	1.00e-14
40	-1.387778780781446e-16	1.231931223699689e-13	1.39e-14	1.23e-13

Tabella 3.8: Dominio non semplicemente connesso, funzione oscillante f_2. Gli errori `Err.1`, `Err.2` in tabella sono di tipo assoluto, visto che l'integrale richiesto é molto prossimo a 0.

	Integrale di riferimento:**1.851086004090770e+00**		Errore: 1.84e-14	
Grado	Formula composta	Formula compressa	Err.1	Err.2
10	**1.85108**3367994063e+00	**1.8510**35129846242e+00	1.4e-06	2.7e-05
20	**1.8510860**03307192e+00	**1.85060**4299571634e+00	4.2e-10	2.6e-04
30	**1.851086004**120307e+00	**1.85100**3916463066e+00	1.6e-11	4.4e-05
40	**1.85108600409**0822e+00	**1.8510**18830637660e+00	2.8e-14	3.6e-05
50	**1.8510860040907**65e+00	**1.8510**66691120349e+00	2.8e-15	1.0e-05

Tabella 3.9: Dominio non semplicemente connesso, funzione f_3 con singolaritá di tipo radice.

Anche per il dominio non semplicemente connesso le formule composte e compresse forniscono un risultato prossimo a quello delle formule adattive al crescere del grado di precisione. Si ottengono maggiori errori utilizzando la funzione oscillante e per questo nei test numerici ci siamo fermati al grado di precisione 40.

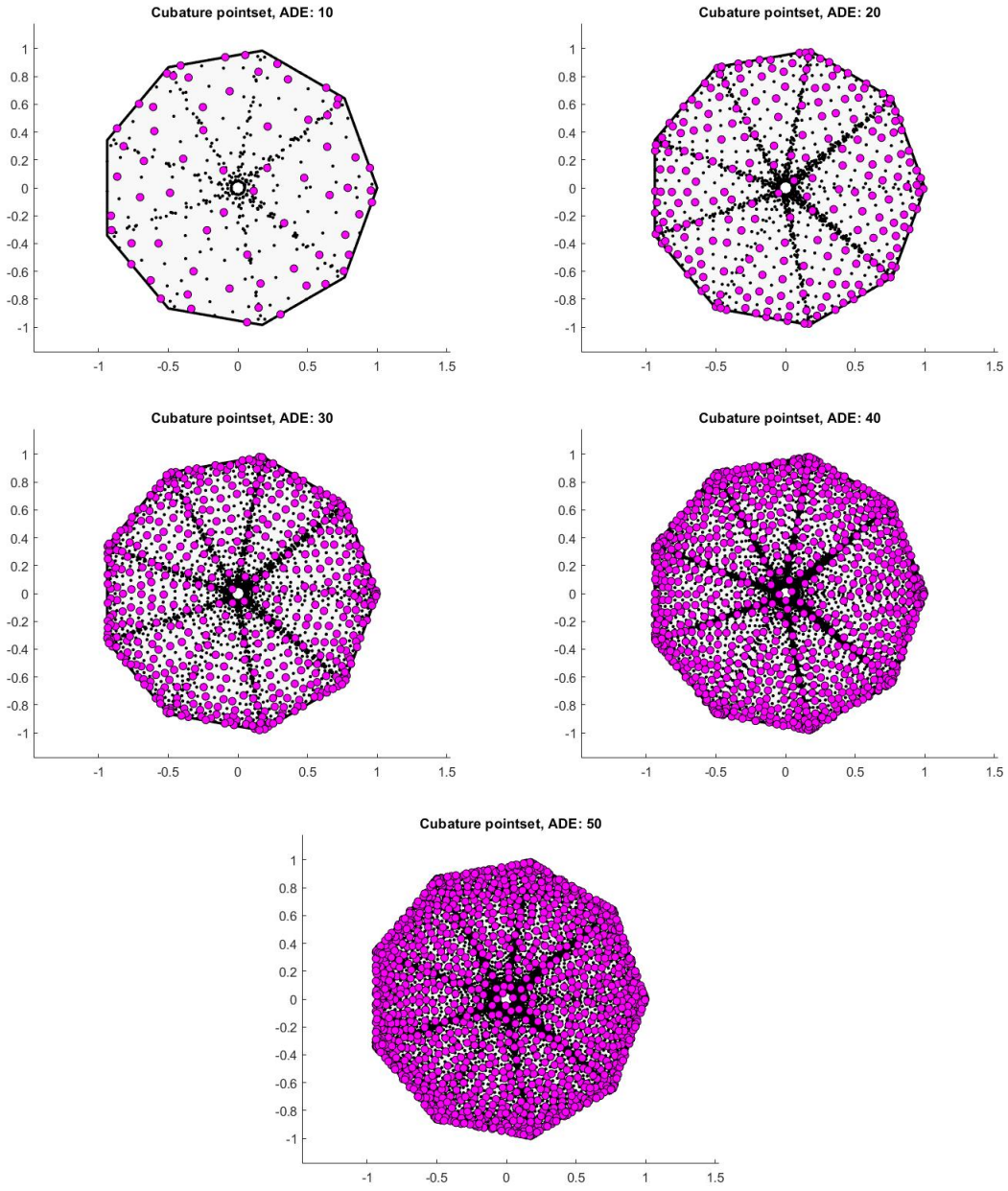


Figura 3.6: Set di nodi di cubatura da grado 10 a grado 50, relativi alla regione poligonale non semplicemente connessa $\mathcal{P}_3$. I puntini in nero sono i nodi della formula composta, in magenta quelli della formula compressa.

Capitolo 4

Una applicazione al design ottico

Consideriamo, in questo capitolo, un problema riguardante il design ottico e la progettazione di lenti per telescopi.

L'uso della cubatura a bassa cardinalità nel design ottico iniziò con un articolo di Forbes [6] del 1988 che suggeriva di adottare regole di quadratura gaussiana per un efficiente ray tracing numerico su aperture circolari o ellittiche. Il ray tracing è una tecnica per rappresentare la propagazione di fronti d'onda attraverso vari mezzi; combinato con l'ottimizzazione dei parametri del sistema ottico, può migliorare le prestazioni di imaging o di illuminazione.

Uno dei parametri ottici rilevanti è il Root Mean Square Wavefront Error (RMSWE), cioè l'errore quadratico medio del fronte d'onda:

$$\mathrm{RMSWE}_\Omega = \sqrt{\int_\Omega W^2(x)\frac{dx}{A} - \left(\int_\Omega W(x)\frac{dx}{A}\right)^2}, \quad A = area(\Omega) \tag{4.1}$$

dove Ω è la regione di integrazione della pupilla e W è il fronte d'onda ottico, di solito approssimato da un'espansione troncata di Zernike di basso grado.

Consideriamo il caso di un modello a tre dischi per una lente con pupilla circolare principale oscurata al centro e con vignettatura (cioè con luminosità ridotta alla periferia rispetto al centro). Questo è un modello per il Large Synoptic Survey Telescope (LSST), oggi chiamato Osservatorio Vera Rubin. Si tratta di un progetto di telescopio riflettore in costruzione in Cile e che, una volta attivo a partire dal 2025, collezionerà più di 20 terabyte di dati ogni notte per 10 anni per ottenere immagini dettagliate del cielo australe. Rispetto ad altri telescopi della stessa grandezza, questo telescopio ha un centro di gravità molto basso che gli consente di spostarsi velocemente e con precisione da un'area di cielo ad un'altra.

E' il primo osservatorio progettato per studiare l'Universo dinamico e in evoluzione.

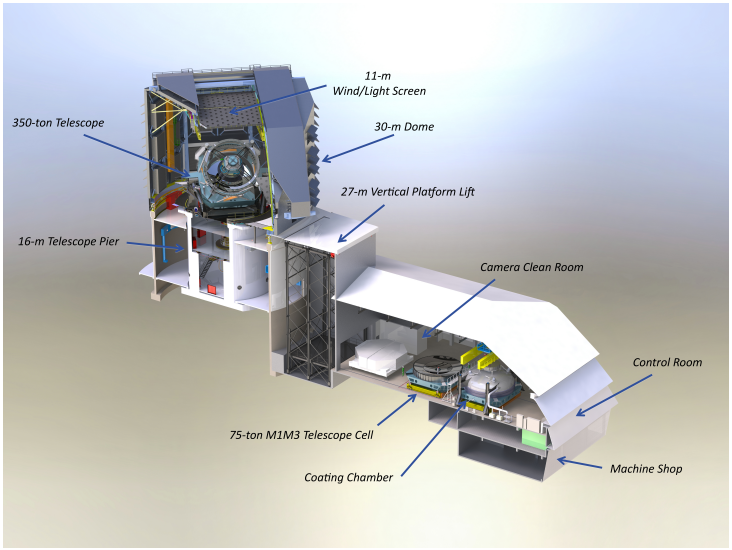


Figura 4.1: Osservatorio Vera Rubin. Credits: LSST/NOIRLab/NSF.

La configurazione diventa più complicata con un modello a 5 dischi. In questo modello l'oscuramento e la vignettatura vengono creati da 4 dischi coassiali.

Esistono diverse configurazioni in base alla posizione dei centri e dei raggi dei dischi coassiali. Tutti i dischi sono approssimati da poligoni regolari con un numero molto alto di lati, in questo modo la regione non oscurata è poligonale e ottenibile dalla funzione Matlab `polyshape` e dalle routines `intersect`, `union`, `subtract`.

I 5 cerchi sono così definiti:

- un disco unitario che rappresenta la "pupilla" di centro $C_1 = (0, 0)$;
- 2 dischi interni con centri: $C_2 = (0, 0), C_3 = (0, -0.1184)$ e raggi rispettivamente: $r_2 = 0.6210$ e $r_3 = 0.5663$;
- 2 dischi esterni con centri: $C_4 = (0, -0.84), C_5 = (0, -0.3761)$ e raggi: $r_4 = 1.0761$ e $r_5 = 1.2810$.

Scegliamo di approssimare questi cerchi mediante poligoni regolari con 100 lati. La regione di integrazione è ottenuta sottraendo all'intersezione dei due dischi esterni, l'unione dei due dischi interni, così da ottenere la regione poligonale in Figura 4.2.

Valutiamo quindi gli integrali:

1. con le formule adattive;
2. con le formule composte;
3. con le formule compresse;

e calcoliamo gli errori assoluti e relativi.

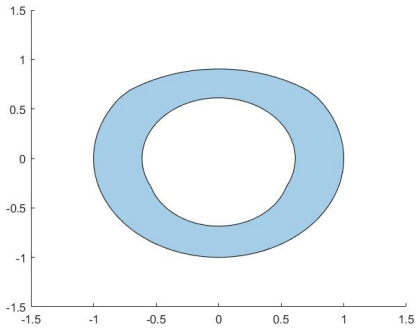


Figura 4.2: Il dominio poligonale $\mathcal{P}$ preso in considerazione negli esperimenti numerici.

Consideriamo come integrande dei polinomi di grado 15 calcolati in modo random: $g(x, y) = (c_1 + c_2 x + c_3 y)^{15}$ dove c_1, c_2, c_3 sono 3 numeri random.

A seguito di 10 test numerici effettuati, otteniamo i seguenti risultati:

	Formula composta	Regola compressa	Formula adattativa
N. di nodi	9292	136	-
cputime medio	3.12e-01	5.96e-01	6.46e-03

Notiamo che il tempo di esecuzione medio è dell'ordine dei decimi di secondo per le regole algebriche e dell'ordine dei centesimi di secondo per la formula adattativa.

Abbiamo inoltre calcolato gli errori logaritmici, cercando l'esponente medio tra i valori degli errori ottenuti dai test random. Gli errori assoluti e relativi ottenuti dalla regola compressa sono dell'ordine della precisione di macchina, quindi minori della tolleranza $tol = 10^{-14}$ fissata.

Nell'immagine seguente sono rappresentati in nero i nodi di cubatura della formula composta, in magenta i nodi della regola di cubatura compressa. E' inoltre rappresentata la triangolazione ottenuta per il dominio non semplicemente connesso considerato.

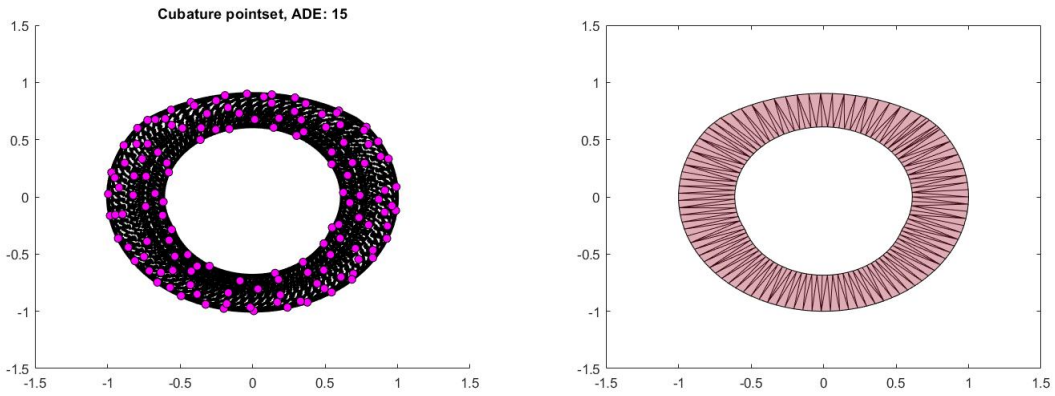


Figura 4.3: Set di nodi di cubatura e triangolazione del dominio poligonale $\mathcal{P}$.

Quale ulteriore esempio consideriamo il calcolo di $\mathrm{RMSWE}_{\mathcal{P}}$ nel caso in cui il fronte d'onda ottico é un polinomio random di grado 7. La tolleranza usata per l'algoritmo adattativo é di 10^{-14}.

Abbiamo utilizzato formule di quadratura aventi grado 14, visto che dobbiamo integrare numericamente $\int_{\mathcal{P}} W^2(x)/A\,dx$ con $W \in \mathbb{P}_7$.

A tal proposito, listiamo nella Tabella 4.1 i risultati ottenuti da 20 tests, su polinomi random di grado 7. Come in precedeza, si verifica la soliditá della procedura adattativa.

Formula composta	Regola compressa	Formula adattiva
$2.227393432645477e+00$	$2.227393432645474e+00$	$2.227393432645478e+00$
$2.141817334017690e+00$	$2.141817334017687e+00$	$2.141817334017690e+00$
$2.228885426779486e+00$	$2.228885426779482e+00$	$2.228885426779487e+00$
$2.499840438881276e+00$	$2.499840438881272e+00$	$2.499840438881278e+00$
$2.328093302775953e+00$	$2.328093302775948e+00$	$2.328093302775953e+00$
$2.320917246267085e+00$	$2.320917246267080e+00$	$2.320917246267086e+00$
$2.091656287784537e+00$	$2.091656287784533e+00$	$2.091656287784539e+00$
$2.324733053899267e+00$	$2.324733053899263e+00$	$2.324733053899268e+00$
$2.013632760187703e+00$	$2.013632760187699e+00$	$2.013632760187704e+00$
$2.176484285833863e+00$	$2.176484285833859e+00$	$2.176484285833864e+00$
$2.180058808949723e+00$	$2.180058808949719e+00$	$2.180058808949725e+00$
$1.905576525479443e+00$	$1.905576525479439e+00$	$1.905576525479444e+00$
$2.367235542372974e+00$	$2.367235542372970e+00$	$2.367235542372974e+00$
$2.450643333503705e+00$	$2.450643333503700e+00$	$2.450643333503705e+00$
$2.063895070207879e+00$	$2.063895070207875e+00$	$2.063895070207881e+00$
$2.293201578620802e+00$	$2.293201578620799e+00$	$2.293201578620803e+00$
$1.638780982770047e+00$	$1.638780982770044e+00$	$1.638780982770049e+00$
$2.294005397099513e+00$	$2.294005397099510e+00$	$2.294005397099514e+00$
$2.170197467408528e+00$	$2.170197467408525e+00$	$2.170197467408528e+00$
$2.527901425706289e+00$	$2.527901425706284e+00$	$2.527901425706289e+00$

Tabella 4.1: $\mathrm{RMSWE}_{\mathcal{P}}$ nel caso in cui il fronte d'onda ottico é un polinomio random di grado 7, sulla regione poligonale $\mathcal{P}$ in oggetto.

Bibliografia

[1] B. Bauman, A. Sommariva e M. Vianello, *Compressed cubature over polygons with applications to optical design*, Journal of Computational and Applied Mathematics, Volume 370 (2020), 112658.

[2] F. Piazzon, A. Sommariva e M. Vianello, *Caratheodory-Tchakaloff Subsampling*, Dolomites Res. Notes Approx. DRNA 10 (2017).

[3] J.N. Lyness, D. Jespersen, *Moderate Degree Symmetric Quadrature Rules for the Triangle*, IMA Journal of Applied Mathematics, Volume 15 (1975), pp.19-32.

[4] J.N. Lyness, R. Cools, *A Survey of Numerical Cubature over Triangles*, Proc. Sympos. Appl. Math., 48, AMS, Providence, 1994.

[5] R.J. Renka, R. Brown, *Algorithm 792: Accuracy Tests of ACM Algorithms for Interpolation of Scattered Data in the Plane*, ACM Transactions on Mathematical Software, Volume 25, Issue 1 (1999), pp.78-94.

[6] G.W. Forbes, *Optical system assessment for design: numerical ray tracing in the Gaussian pupil*, J. Opt. Soc. Am. A 5, Vol. 5, Issue 11 (1988), pp. 1943-1956.

[7] L.F. Shampine, *Matlab program for quadrature in 2D*, Applied Mathematics and Computation Volume 202 (2008), pp.266-274.

Codici Matlab

Funzione Matlab `demo_2D_cub_adaptive`:

```
function demo2D_cub_adaptive(domain_example,function_example,nV)

% Object:
% 1. Demo on algebraic polynomial cubature on polygonal domains.
% 2. It compares the results with some algebraic rules.
% Dates:
% Written on 29/10/2020: M. Vianello;
%
% Modified on:
% 08/11/2023: A. Sommariva.

% domain_example  : domain to be tested (in a list of examples)
% function_example: string that chooses the integrand;
% nV              : degrees of exactness of the rules.

if nargin < 1, domain_example=1; end
if nargin < 2, function_example=1; end
if nargin < 3, nV=1:10; end

[vertices,P,domain_str]=gallery_polygons(domain_example);
[g,gstr]=gallery_functions_2D(function_example);

% ........ reference value ........
degR=NaN; tol=10^(-14);
tic;
[IH,IL,flag,iters,L1_vertices]=cub_polygon_adaptive(vertices,g,tol);
IR=IH; % integral reference value
cpuR=toc;

% ......... comparisons ...........
for k=1:length(nV)

    n=nV(k);

    fprintf('\n \t \t -> ADE: %3.0f',n);

    % ........ full rule ........
    tic;
    xv=vertices(:,1); yv=vertices(:,2); iv=[];
    [xyw,xvc,yvc,pgon,tri,xywc]=cub_polygon(n,xv,yv,iv);
```

```
    cpu(k,1)=toc;
    gX=feval(g,xyw(:,1),xyw(:,2)); w=xyw(:,3);
    I(k)=w'*gX;
    AE(k)=abs(I(k)-IR); RE(k)=AE(k)./abs(IR);
    cardX(k)=size(xyw,1);

    % ........ compressed rule ........
    X=xyw(:,1:2); u=w;
    tic; [XC,wc,momerrL,dbox]=dCATCH(n,X,w); cpu(k,2)=toc;
    gXC=feval(g,XC(:,1),XC(:,2));
    IC(k)=wc'*gXC;
    AEC(k)=abs(IC(k)-IR); REC(k)=AEC(k)./abs(IR);
    cardXC(k)=size(XC,1);

end

% ..... statistics ....
fprintf('\n \t ........................................');
fprintf('\n \t Domain: polygon');
fprintf('\n \t Function: '); disp(gstr);
fprintf('\n \t ........................................');
fprintf('\n \t |  n  | card X | cardXC |   cpuf   |   cpuc   |  ');
fprintf('\n \t ........................................');
for k=1:length(nV)
    fprintf('\n \t | %3.0f | %6.0f | %6.0f | %1.2e | %1.2e |',...
        nV(k),cardX(k),cardXC(k),cpu(k,1),cpu(k,2));
end
fprintf('\n \t ........................................');
fprintf('\n \t | %3.0f | %6.0f | %6.0f | %1.2e | %1.2e |',...
    NaN,NaN,0,cpuR,0);
fprintf('\n \t ........................................');
fprintf('\n \n');


fprintf('\n \t ........................................');
fprintf('\n \t                RESULTS BY ALGEBRAIC RULES ');
fprintf('\n \t ........................................');
fprintf('\n \t |  n  |          If          |          Ic          |
    REf   |   REc   |  ');
fprintf('\n \t ........................................');
for k=1:length(nV)
    fprintf('\n \t | %3.0f | %1.15e | %1.15e | %1.1e | %1.1e |',...
        nV(k),I(k),IC(k),AEC(k),REC(k));
end
fprintf('\n \t ........................................');
fprintf('\n \n')

fprintf('\n \t ........................................');
fprintf('\n \t               RESULTS BY ADAPTIVE ROUTINE');
fprintf('\n \t ........................................');
fprintf('\n \t | its |          IH          |          IL          |
    AE    |    RE   | flag | ');
fprintf('\n \t ........................................');
```

```
fprintf('\n \t | %3.0f | %1.15e | %1.15e | %1.1e | %1.1e |   %1.0g  |'
    ,...
     iters,IR,IL,abs(IR-IL),abs((IR-IL)/(IR+(IR == 0))),flag);
fprintf('\n \t .....................................');
fprintf('\n \n')
fprintf('\n \t .....................................');
fprintf('\n \t                          ADDITIONAL NOTES');
fprintf('\n \t .....................................');
if flag == 0
    fprintf('\n \t * the adaptive routine terminated successfully');
else
    fprintf('\n \t * the adaptive routine did not terminated
   successfully');
end
fprintf('\n \t * the cubature result provided by adaptive rule is: %1.15
   e',IR);
fprintf('\n \t * the absolute error estimate is                  : %1.2e
   ',abs((IR-IL)));
fprintf('\n \t * the relative error estimate is                  : %1.2e
   ',abs((IR-IL)/IR));
fprintf('\n \t * the cputime required by the adaptive rule is    : %1.2e
   ',cpuR)
fprintf('\n \t .....................................');
fprintf('\n \n');



% % ..... plots .....
clear_figure(1)
figure(1);
plot_polygon(vertices,xyw,XC);
hold on;
title_str=['Cubature pointset, ADE: ',num2str(n)];
title(title_str);
axis equal;
hold off;


clear_figure(2)
figure(2)
cartTRI=length(L1_vertices);
for k=1:cartTRI
    triL=L1_vertices{k}; triL(end+1,:)=triL(1,:);
    plot(triL(:,1),triL(:,2),'k-');
    hold on;
end
plot(pgon);
hold off
```

Funzione Matlab `demo_2D_cub_adaptive_optics`:

```
function demo2D_cub_adaptive_optics(nV)

% Object:
% 1. Demo on algebraic polynomial cubature on polygonal domains.
% 2. It compares the results with some algebraic rules.

% Dates:
% Written on 29/10/2020: M. Vianello;
%
% Modified on:
% 08/11/2023: A. Sommariva.

% domain_example  : domain to be tested (in a list of examples)
% function_example: string that chooses the integrand;
% nV              : degrees of exactness of the rules.

if nargin < 1, nV=1:15; end

[vertices,P,domain_str]=gallery_polygons(4);
% [g,gstr]=gallery_functions_2D(function_example);

% ........ reference value ........
degR=NaN; tol=10^(-14);


% ......... comparisons ...........
for k=1:length(nV)

    n=nV(k);

    fprintf('\n \t \t -> ADE: %3.0f',n);

    % ........ full rule ........
    tic;
    xv=vertices(:,1); yv=vertices(:,2); iv=[];
    [xyw,xvc,yvc,pgon,tri,xywc]=cub_polygon(n,xv,yv,iv);
    cpu(k,1)=toc;
    cardX(k)=size(xyw,1);

    % ........ compressed rule ........
    X=xyw(:,1:2); w=xyw(:,3);
    tic; [XC,wc,momerrL,dbox]=dCATCH(n,X,w); cpu(k,2)=toc;
    cardXC(k)=size(XC,1);

    for kk=1:10

        c1=rand(1); c2=rand(1); c3=rand(1);
        g=@(x,y) (c1+c2*x+c3*y).^k;
        gstr='Polynomial';

        tic;
```

```
        [IH,IL,flag,iters,L1_vertices]=cub_polygon_adaptive(vertices,g,
   tol);
        IR=IH; % integral reference value
        cpuR=toc;

        gX=feval(g,xyw(:,1),xyw(:,2)); w=xyw(:,3);
        I(kk)=w'*gX;
        AE(kk)=abs(I(kk)-IR); RE(kk)=AE(kk)./abs(IR);

        gXC=feval(g,XC(:,1),XC(:,2));
        IC(kk)=wc'*gXC;
        AEC(kk)=abs(IC(kk)-IR); REC(kk)=AEC(kk)./abs(IR);

    end

    AE(k)=10^(mean(log10(AE)));       RE(k)=10^(mean(log10(RE)));
    AEC(k)=10^(mean(log10(AEC)));     REC(k)=10^(mean(log10(REC)));


end

% ..... statistics ....
fprintf('\n \t ..................................................');
fprintf('\n \t Domain: polygon');
fprintf('\n \t Function: '); disp(gstr);
fprintf('\n \t ..................................................');
fprintf('\n \t |  n  | card X | cardXC |   cpuf   |   cpuc   |  ');
fprintf('\n \t ..................................................');
for k=1:length(nV)
    fprintf('\n \t | %3.0f | %6.0f | %6.0f | %1.2e | %1.2e |',...
        nV(k),cardX(k),cardXC(k),cpu(k,1),cpu(k,2));
end
fprintf('\n \t ..................................................');
fprintf('\n \t | %3.0f | %6.0f | %6.0f | %1.2e | %1.2e |',...
    NaN,NaN,0,cpuR,0);
fprintf('\n \t ..................................................');
fprintf('\n \n');


fprintf('\n \t ..................................................');
fprintf('\n \t                          RESULTS BY ALGEBRAIC RULES ');
fprintf('\n \t ..................................................');
fprintf('\n \t |  n  |   REf   |   REc   |  ');
fprintf('\n \t ..................................................');
for k=1:length(nV)
    fprintf('\n \t | %3.0f | %1.1e | %1.1e |',...
        nV(k),AEC(k),REC(k));
end
fprintf('\n \t ..................................................');
fprintf('\n \n')
```

```
% % ..... plots .....
clear_figure(1)
figure(1);
plot_polygon(vertices,xyw,XC);
hold on;
title_str=['Cubature pointset, ADE: ',num2str(n)];
title(title_str);
hold off;


clear_figure(2)
figure(2)
cartTRI=length(L1_vertices);
for k=1:cartTRI
    triL=L1_vertices{k}; triL(end+1,:)=triL(1,:);
    plot(triL(:,1),triL(:,2),'k-');
    hold on;
end
plot(pgon);
hold off
```

Funzione Matlab `gallery_functions_2D`:

```
function [f,fs]=gallery_functions_2D(example)

% OBJECT
% Battery of bivariate test functions.

% INPUT
% example: parameter that chooses the function;
% OUTPUT
% f : function to be tested
% PAPERS
% 1. Algorithm 792: Accuracy Tests of ACM Algorithms for Interpolation
   of
%    Scattered Data in the Plane
%    ROBERT J. RENKA and RON BROWN

if nargin < 1, example=1; end
if isempty(example), example=1; end


switch example
    case 1
        %f=@(x,y) franke(x,y);

        f=@(x,y).75*exp(-((9*x-2).^2 + (9*y-2).^2)/4) + ...
         .75*exp(-((9*x+1).^2)/49 - (9*y+1)/10) + ...
         .5*exp(-((9*x-7).^2 + (9*y-3).^2)/4) - ...
         .2*exp(-(9*x-4).^2 - (9*y-7).^2);
        fs='Renka, Brown, F1: franke(x,y)';
    case 2
        % Renka, Brown, F7
        f=@(x,y) 2*cos(10*x).*sin(10*y)+sin(10*x.*y);
```

```
        fs='Renka, Brown, F7: 2*cos(10*x).*sin(10*y)+sin(10*x.*y);';
    case 3
        alpha=1/2;
        f=@(x,y) (x.^2+y.^2).^alpha;
        fs=['(x.^2+y.^2).^alpha,    ','alpha=',num2str(alpha,'%.5g')];
end
```

Funzione Matlab gallery_polygons:

```
function [vertices,P,domain_str]=gallery_polygons(example)

P=[];

switch example


    case 1
        domain_str='CONVEX POLYGON';
        shift=[0.3 0.3];
        vertices=[0.1 0; 0.7 0.2; 1 0.5; 0.75 0.85; 0.5 1; 0 0.25];
        vertices=bsxfun(@minus,vertices,shift);

    case 2
        domain_str='NON CONVEX POLYGON';
        shift=[0.3 0.3];
        vertices=(1/4)*[1 0; 3 2; 3 0; 4 2; 3 3; 3 0.85*4; 2 4; ...
            0 3; 1 2];
        vertices=bsxfun(@minus,vertices,shift);

    case 3 % domain with holes (using polyshape)
        domain_str='DOMAIN WITH HOLES';
        Nsides=10;
        th=linspace(0,2*pi,Nsides); th=(th(1:end-1))';
        % first polygon.
        polygon1=[cos(th) sin(th)]; P1=polyshape(polygon1);
        % second polygon.
        polygon2=0.05*[cos(th) sin(th)]; P2=polyshape(polygon2);

        P=subtract(P1,P2);
        vertices=P.Vertices;

    case 4
        domain_str='DOMAIN WITH HOLES: OPTICS';
        Nsides=100;
        y=[0          0    -0.1184    -0.1184    -0.3761];
        r=[1.0000     0.6120     0.5663     1.0761     1.2810];
        th=linspace(0,2*pi,Nsides); th=(th(1:end-1))';
        C1=[0 y(1)]; P1v=C1+r(1)*[cos(th) sin(th)]; P1=polyshape(P1v);
        C2=[0 y(2)]; P2v=C2+r(2)*[cos(th) sin(th)]; P2=polyshape(P2v);
        C3=[0 y(3)]; P3v=C3+r(3)*[cos(th) sin(th)]; P3=polyshape(P3v);
        C4=[0 y(4)]; P4v=C4+r(4)*[cos(th) sin(th)]; P4=polyshape(P4v);
        C5=[0 y(5)]; P5v=C5+r(5)*[cos(th) sin(th)]; P5=polyshape(P5v);
        Pout=intersect(P1,P4);
```

```
        Pout=intersect(Pout,P5);
        Pin=union(P2,P3);
        P=subtract(Pout,Pin);
        vertices=P.Vertices;

    otherwise
        domain_str='SQUARE';
        vertices=[-1 -1; 1 -1; 1 1; -1 1];
end

if isempty(P) & nargout >= 2
    P=polyshape(vertices);
end
```

Funzione Matlab cub_polygon_adaptive:

```
function [IH,IL,flag,iters,L1_vertices]=cub_polygon_adaptive(vertices,f,
   tol)

% Input:
% vertices : it is a M x 3 matrix, where the k-th row represents the
%            cartesian coordinates of the k-th vertex of the polygonal
%            region (counterclockwise order);
% f        : function to integrate over the  triangle defined by
%            vertices;
% tol      : absolute cubature error tolerance.

% Output:
% I      : approximation of integral of "f" over the  triangle
%          defined by vertices;
% IH     : high order approximation of the integral;
% IL     : low order approximation of the integral;
% L1_vertices: vertices of the triangles considered by the algorithm.

% Subroutines:
% 1. add_triangle_data
% 2. generate_triangle_sons
% 3. cub_tri
% 4. lyness_jespersen

% Dates:
% Initial version: January 05, 2021, by A. Sommariva


% ................. Troubleshooting and settings
   ........................

if nargin < 3, tol=10^(-6); end

% max number of triangles in which the domain is partitioned
max_triangles=5000;

% ........ Procedure start up, data stored in structures ............
```

```
% Main strategy:
% First we triangulate the polygon, providing the vertices of the
% triangulation and its connectivity list (description of each
% triangle via its vertices).
% We compute low and high order integrals on each triangle and then keep
% erasing that one with worst error from the list, subdivide it in
% simplices, compute integrals in each of them and updating the list.

flag=0;

L1_vertices={}; L1_integrals_L=[]; L1_integrals_H=[]; L1_errors=[];

% If input is not a polyshape determines its polyshape "PG".
S=class(vertices);
if strcmp(S,'polyshape') == 0
    XV=vertices(:,1); YV=vertices(:,2); PG=polyshape(XV,YV);
else
    PG=vertices;
end

tri=triangulation(PG);
tri_vertices=tri.Points; tri_conn_list=tri.ConnectivityList;

for k=1:size(tri_conn_list,1)
    verticesL=tri_vertices((tri_conn_list(k,:))',:);
    [L1_vertices,L1_integrals_L,L1_integrals_H,L1_errors]=...
        add_triangle_data(verticesL,f,L1_vertices,L1_integrals_L,...
        L1_integrals_H,L1_errors);
end

IL=sum(L1_integrals_L); IH=sum(L1_integrals_H); AE=abs(IL-IH);

% ..................... successive refinement
   ........................

iters=1;

while (AE >= tol) & (AE >= tol*IH)

    N=length(L1_integrals_L);

    % Too many triangles: exit with errors
    if N > max_triangles, flag=1; return; end

    % Determine triangle with worst error.
    [Ierr_max,kmax]=max(L1_errors);
    vertices_kmax=L1_vertices{kmax};

    % Erase "kmax" triangle from LIST 1
    k_ok=setdiff(1:N,kmax);

    if length(k_ok) > 0
        L1_vertices={L1_vertices{k_ok}};
```

```
        L1_integrals_L=L1_integrals_L(k_ok);
        L1_integrals_H=L1_integrals_H(k_ok);
        L1_errors=L1_errors(k_ok);
    else
        L1_vertices={}; L1_integrals_L=[]; L1_integrals_H=[]; L1_errors
   =[];
    end

    new_triangles_cell=generate_triangle_sons(vertices_kmax);

    % Replace worst triangle "data", with "those" of a triangulation.
    for j=1:4
        verticesL=new_triangles_cell{j};
        [L1_vertices,L1_integrals_L,L1_integrals_H,L1_errors]=...
            add_triangle_data(verticesL,f,L1_vertices,L1_integrals_L,...
            L1_integrals_H,L1_errors);
    end

    IL=sum(L1_integrals_L); IH=sum(L1_integrals_H); AE=abs(IL-IH);

    iters=iters+1;

end



function [L1_vertices,L1_integrals_L,L1_integrals_H,L1_errors]=...
    add_triangle_data(vertices,f,L1_vertices,L1_integrals_L,...
    L1_integrals_H,L1_errors)


L1_vertices{end+1}=vertices;
[IL,IH]=cub_tri(f,vertices);
L1_integrals_L(end+1)=IL;
L1_integrals_H(end+1)=IH;
L1_errors(end+1)=abs(IL-IH);



function triangles_cell=generate_triangle_sons(vertices)

% Input:
% vertices : it is a 3 x 3 matrix, where the k-th row represents the
%            cartesian coordinates of the k-th vertex (counterclockwise)
   ;
% f        : function to integrate over the triangle defined by
%            vertices;

% Output:
% L2_data :

OA=(vertices(1,:)); OB=(vertices(2,:)); OC=(vertices(3,:));
```

```
% ..... compute midpoints
    ..........................

OAB_mid=(OA+OB)/2; OAC_mid=(OA+OC)/2;  OBC_mid=(OB+OC)/2;

% ..... triangle data
    ..............................


triangles_cell{1}=[OA;OAB_mid;OAC_mid];
triangles_cell{2}=[OAB_mid; OB; OBC_mid];
triangles_cell{3}=[OBC_mid; OC; OAC_mid];
triangles_cell{4}=[OAB_mid; OBC_mid; OAC_mid];



function [IL,IH]=cub_tri(f,vertices)

% Object:
% Computation of low order and higher order rules of integral of "f" on
    the
% triangle T determined by vertices.

% Input:
% f: function to integrate;
% vertices: 3 x 3 matrix whose k-th row represents the cartesian
% coordinates of the k-th vertex.

% Output:
% IL: low order approximation of the integral of "f" on the triangle T;
% IH: high order approximation of the integral of "f" on the triangle T.

tri_area=polyarea(vertices(:,1),vertices(:,2));

% ..... low order rule
    ..........................

xyw_bar=lyness_jespersen(9);

% nodes and weights
lambda=xyw_bar(:,1:3);
nodes=lambda*vertices;

% Low integral
w=tri_area*xyw_bar(:,4);

fXY=feval(f,nodes(:,1),nodes(:,2));
IL=w'*fXY;

% ..... high order rule
    ..........................

xyw_bar=lyness_jespersen(11);
```

```
% nodes and weights
lambda=xyw_bar(:,1:3);
nodes=lambda*vertices;

% Low integral
w=tri_area*xyw_bar(:,4); fXY=feval(f,nodes(:,1),nodes(:,2)); IH=w'*fXY;



function xyw_bar=lyness_jespersen(deg)

switch deg
    case   9

        % ALG. DEG.:    9
        % PTS CARD.:   19
        % NEG. W.  :    0
        % OUT PTS. :    0
        % M.E.INF. : 3.1e-16

        xyw_bar=[
            3.33333333333335257719909350271337e-01
   3.33333333333314829616256247390 99e-01
   3.33333333333332593184650249895 64e-01
   9.71357962827988224985276133338630e-02
            4.89682519198736176946340492577292e-01
   4.89682519198739063526204517984297e-01
   2.06349616025247595274549894384108e-02
   3.13347002271390229211078803928103e-02
            4.89682519198737009613608961444697e-01
   2.06349616025245374828500644071028e-02
   4.89682519198738397392389742890373e-01
   3.13347002271387731209273397325887e-02
            2.06349616025248878969922117221358e-02
   4.89682519198738230858936049116892e-01
   4.89682519198736898591306498929043e-01
   3.13347002271393143546518444964022e-02
            4.37089591492937745709213004374760e-01
   4.37089591492935525263163754061679e-01
   1.25820817014126673516472010305733e-01
   7.78275410047742782770896496913338e-02
            4.37089591492937745709213004374760e-01
   1.25820817014125674315749847664847e-01
   4.37089591492936524463885916702566e-01
   7.78275410047739035768188387010014e-02
            1.25820817014127700472769788575533e-01
   4.37089591492935580774314985319506e-01
   4.37089591492936690997339610476047e-01
   7.78275410047746946107238841250364e-02
            1.88203535619033524017851277676527e-01
   1.88203535619031969705616802457371e-01
   6.23592928761934617298834382381756e-01
```

```
  7.96477389272102626049942841746088e-02
           1.88203535619033940351485512110230e-01
  6.23592928761932951964297444646945e-01
  1.88203535619033135439792658871738e-01
  7.96477389272104707718114013914601e-02
           6.23592928761936171611068857600912e-01
  1.88203535619031553371982568023668e-01
  1.88203535619032275016948574375419e-01
  7.96477389272100266826015513288439e-02
           4.47295133944526565605848134055123e-02
  4.47295133944527745217811798283947e-02
  9.10540973211094617489891334116692e-01
  2.55776756586980312524470804191878e-02
           4.47295133944526010494335821476852e-02
  9.10540973211094728512193796632346e-01
  4.47295133944526357439031016838271e-02
  2.55776756586979271690385218107622e-02
           9.10540973211094395445286409085384e-01
  4.47295133944528369718263149934501e-02
  4.47295133944527675828872759211663e-02
  2.55776756586981388053025909812277e-02
           3.68384120547366744613526634566369e-02
  7.41198598784498008384957756788936e-01
  2.21962989160765289398113964125514e-01
  4.32835393772897300546098620088742e-02
           3.68384120547364316000660267036437e-02
  2.21962989160765927776353123590525e-01
  7.41198598784497564295747906726319e-01
  4.32835393772896051545195916787634e-02
           7.41198598784497897362655294273281e-01
  3.68384120547358973052354258470587e-02
  2.21962989160766205332109279879660e-01
  4.32835393772890292263255673788080e-02
           7.41198598784497564295747906726319e-01
  2.21962989160766371865562973653141e-01
  3.68384120547360638386891196205397e-02
  4.32835393772892443320365885028878e-02
           2.21962989160765400420416426641168e-01
  3.68384120547361401665220626000519e-02
  7.41198598784498452474167606851552e-01
  4.32835393772891610653097416161472e-02
           2.21962989160765011842357807836379e-01
  7.41198598784498452474167606851552e-01
  3.68384120547364801723233540542424e-02
  4.32835393772895010711110330703377e-02
           ];

   case  11

       % ALG. DEG.:   11
       % PTS CARD.:   28
       % NEG. W.  :    0
       % OUT PTS. :    0
```

```
      % M.E.INF. : 4.2e-16

      xyw_bar=[
          3.33333333333333425851918718763045e-01
  3.33333333333333370340767487505218e-01
  3.33333333333331482961625624739l0e-01
  8.79773011622321937652557721776247e-02
          2.59891409282874893960091355893383e-02
  2.59891409282874824571152316821099e-02
  9.48021718143424951819042689749040e-01
  8.74431155373607202352381762011646e-03
          2.59891409282874408237518082387396e-02
  9.48021718143425062841345152264694e-01
  2.59891409282875240904786551254801e-02
  8.74431155373604947211862992162423e-03
          9.48021718143425173863647614780348e-01
  2.59891409282874408237518082387396e-02
  2.59891409282873853126005769809126e-02
  8.74431155373604773739515394481714e-03
          9.42875026479226829856372660287889e-02
  9.42875026479227384967884972866159e-02
  8.11424994704154634028725467942422e-01
  3.80815719939349636713465940829337e-02
          9.42875026479225580855469956986781e-02
  8.11424994704154745051027930458076e-01
  9.42875026479227384967884972866159e-02
  3.80815719939349636713465940829337e-02
          8.11424994704154634028725467942422e-01
  9.42875026479225997189104191420483e-02
  9.42875026479227662523641129155294e-02
  3.80815719939349428546648823612486e-02
          4.94636775017213814464867027709261e-01
  4.94636775017213869976018258967088e-01
  1.07264499655723155591147133236518e-02
  1.88554480561312423625430767515354e-02
          4.94636775017213758953715796451434e-01
  1.07264499655723415799668529757582e-02
  4.94636775017213869976018258967088e-01
  1.88554480561312527708839326123780e-02
          1.07264499655723502535842328597937e-02
  4.94636775017213814464867027709261e-01
  4.94636775017213814464867027709261e-01
  1.88554480561312597097778365196064e-02
          2.07343382614511434480775164956867e-01
  2.07343382614511434480775164956867e-01
  5.85313234770977075527298438828439e-01
  7.21596975447395400093952844144951e-02
          2.07343382614511378969623933699040e-01
  5.85313234770977297571903363859747e-01
  2.07343382614511351214048318070127e-01
  7.21596975447395122538196687855816e-02
          5.85313234770977297571903363859747e-01
  2.07343382614511351214048318070127e-01
```

```
2.07343382614511351214048318070127e-01
7.21596975447395122538196687855816e-02
         4.389078057004920929706770493794447e-01
4.38907805700492148481828280637274e-01
1.22184388599015814058645901241107e-01
6.93291387055358782065539458017156e-02
         4.3890780570049220399297951189510 1e-01
1.22184388599015814058645901241107e-01
4.38907805700491926437223355605965e-01
6.93291387055358782065539458017156e-02
         1.22184388599015911203160555942304e-01
4.38907805700492148481828280637274e-01
4.38907805700491926437223355605965e-01
6.93291387055359059621295614306291e-02
         7.85647334377654633569776739674958e-17
8.58870281282636427455656757956604e-01
1.41129718717363461522040779527742e-01
7.36238378330058337167818294233257e-03
         1.57669518850046390999681416911743e-16
1.41129718717363572544343242043396e-01
8.58870281282636316433354295440949e-01
7.36238378330061546406248851326382e-03
         8.588702812826364274556567579566 4e-01
1.42062476402819843287610091505200e-16
1.41129718717363433766465163898829e-01
7.36238378330060505572163265242125e-03
         8.588702812826364274556567579566 4e-01
1.41129718717363350499738317012088e-01
2.22044604925031308084726333618164e-16
7.36238378330062066823291644368510e-03
         1.41129718717363572544343242043396e-01
2.02916231519083255217512748366332e-16
8.58870281282636205411051832925295e-01
7.36238378330063020921203431612412e-03
         1.41129718717363378255313932641002e-01
8.58870281282636427455656757956604e-01
2.22044604925031308084726333618164e-16
7.36238378330061199461553655964963e-03
         4.48416775891304561496575331602799e-02
6.77937654882590168270439789921511e-01
2.77220667528279340885433157382067e-01
4.10563154292885590379569293872919e-02
         4.48416775891304908441270526964217e-02
2.77220667528279451907735619897721e-01
6.77937654882590057248137327405857e-01
4.10563154292885729157447372017486e-02
         6.77937654882590168270439789921511e-01
4.48416775891304561496575331602799e-02
2.77220667528279396396584388639894e-01
4.10563154292885590379569293872919e-02
         6.77937654882590168270439789921511e-01
2.77220667528279340885433157382067e-01
4.48416775891304908441270526964217e-02
```

```
   4.10563154292885451601691215728351e-02
            2.77220667528279340885433157382067e-01
   4.48416775891305255385965722325636e-02
   6.77937654882590168270439789921511e-01
   4.10563154292885798546386411089770e-02
            2.77220667528279285374281926124240e-01
   6.77937654882590279292742252437165e-01
   4.48416775891303798218245901807677e-02
   4.10563154292885590379569293872919e-02
            ];

end
```

Funzione Matlab cub_polygon:

```
function [xyw,xvc,yvc,pgon,tri,xywc]=cub_polygon(ade,xv,yv,iv)

% Important: this "cub_polygon" version requires at least Matlab 9.3.0.

% Input:
% ade: algebraic degree of precision of the rule
% xvc: abscissae of the vertices.
% yvc: ordinates of the vertices.
% iv: index of the polygon components. Example: if part of the boundary
   is
%     made by the first 5 coordinates and two other non connected
%     components by other 10 and 12 coordinates, then iv=[5,10,12].
%
% Important:
% 1. differently from classical codes on polygons, the first and
% last vertex must NOT be equal.
% 2. if the polygon is of polyshape class, set "xv" such polygon.

% Output:

% xyw: N x 3 matrix, where (x,y) with x=xyw(:,1), y=xyw(:,2) are the
   nodes
%      and w=xyw(:,3) are the weights.
% xvc, yvc: if "xv" is a polyshape object then "xvc","yvc" are column
   vectors,
%    otherwise they are cell arrays.
% pgon: polygon coded in polyshape form.
% tri: polygon triangulation in polyshape form.

% Routines required.

% 1. cub_triangle (external)
% 2. rule_conversion (attached here)

% Copyrights.

%% Copyright (C) 2007-2018 Alvise Sommariva, Marco Vianello.
%%
%% 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
%%
%% Author:  Alvise Sommariva <alvise@euler.math.unipd.it>
%%          Marco Vianello   <marcov@euler.math.unipd.it>
%%
%% Date: December 07, 2018.


S=class(xv);

S_double = strcmp(S,'double');

% TROUBLESHOOTING.
if S_double % distinguishing polyshape class from double.
    if nargin < 3
        pts=xv;
        if size(pts,1) <= size(pts,2) % working with column vectors.
            pts=xv';
        end
        if pts(1,:) == pts(end,:)
            pts=pts(1:end-1,:);
        end
        xv=pts(:,1); yv=pts(:,2); iv=length(xv);
    else
        if size(xv,1) <= size(xv,2) % working with column vectors.
            xv=xv';
        end
        if size(yv,1) <= size(yv,2) % working with column vectors.
            yv=yv';
        end
        %     if [xv(1) yv(1)] == [xv(end) yv(end)]
        %         xv=xv(1:end-1); yv=yv(1:end-1);
        %     end
        if nargin < 4
            iv=length(xv);
        else
            if isempty(iv)
                iv=length(xv);
            end
        end
    end
```

```
    % Here we convert the given information into cells of coordinates,
   where
    % each component represent a geometrical component of the polygon,
   not
    % connected or at most tangent to other geometrical components.

    xvc={xv(1:iv(1))}; yvc={yv(1:iv(1))};
    ii1=1; ii2=iv(1);
    for ii=2:length(iv)
        ii1=ii2+1;
        ii2=ii2+iv(ii);
        xvl=xv(ii1:ii2);
        yvl=yv(ii1:ii2);

        xvc{end+1}=xvl; yvc{end+1}=yvl;
    end

    pgon = polyshape(xvc,yvc);
else
    pgon=xv;
    [xvc,yvc]=boundary(pgon);
end

tic;
tri = triangulation(pgon);
t1=toc;

TCL=tri.ConnectivityList;
X = tri.Points(:,1);
Y = tri.Points(:,2);

N=size(TCL,1);

% fprintf('\n \t SIDES: %5.0f TRIANGLES: %5.0f CPU: %2.2e',length(xv),N,
   t1);

% clf;
% hold on;
% triplot(tri);
% for ii=1:N
%     TTloc=TCL(ii,:);
%     Xloc=X(TTloc'); Xloc=[Xloc;Xloc(1)];
%     Yloc=Y(TTloc'); Yloc=[Yloc;Yloc(1)];
%
% end

% rule in barycentric coordinates
[xyw,xyw_bar]=cub_triangle(ade);


xyw=[];
for ii=1:N
```

```
    TTloc=TCL(ii,:);
    Xloc=X(TTloc'); Yloc=Y(TTloc');
    xywloc=rule_conversion(xyw_bar,[Xloc Yloc]);
    xyw=[xyw; xywloc];
end


if nargout >=6
    [nodes,w]=dCATCH(ade,xyw(:,1:2),xyw(:,3));
    xywc=[nodes w];
else
    xywc=[];
end


function xyw=rule_conversion(xyw_bar,vertices)

% INPUT:
% ade: ALGEBRAIC DEGREE OF EXACTNESS.
% vertices: 3 x 2 MATRIX OF VERTICES OF THE SIMPLEX.

% OUTPUT:
% xw: NODES AND WEIGHTS OF STROUD CONICAL RULE TYPE OF ADE ade ON THE
   SIMPLEX
%     WITH VERTICES vertices.

bar_coord=xyw_bar(:,1:3);
xx=bar_coord*vertices;

A=polyarea(vertices(:,1),vertices(:,2));
ww=A*xyw_bar(:,4);

xyw=[xx ww];
```

Funzione Matlab dCATCH:

```
function [nodes,w,momerr,dbox] = dCATCH(deg,X,u,LHDM_options,verbose,...
    dim_poly,method)

% Object:
% This routine implements the Caratheodory-Tchakaloff d-variate discrete
% measure compression.
% Examples of its application are probability measures (designs) or
% quadrature formulas.
% Moments are invariant (close to machine precision) up to degree "deg"
% and adapt to the (numerical) dimension of the polynomial space on the
% point set X.
% The routine works satisfactorily for low/moderate degrees, depending
   on
% the dimension.

% Input:
% deg: polynomial exactness degree;
% X: d-column array of point coordinates;
```

```
% * u: 1-column array of nonnegative weights, or nonnegative scalar in
%    case of equal weights;
% * LHDM_options: structure containing the values of optimization
%       parameters (see "LHDM.m" for details);
% * verbose: 0: no information from routine;
%            1: relevant information from routine;
% * dim_poly: dimension of polynomial space (if known in advance, useful
%   for instance on the sphere or its portions).
% Note: the variables with an asterisk "*" are not mandatory and can be
% also set as empty matrix.

% Output:
% nodes: d-column array of extracted mass points coordinates; the
   variable
%    "nodes" has rows that are also rows of "X", i.e. nodes represent a
%    subset of "X";
% w: 1-column array of corresponding new positive weights;
% momerr: moment reconstruction error;
% * dbox: variable that defines a hyperrectangle with
%    sides parallel to the axis, containing the domain (or pointset X in
%    the discrete case).
%    It is a matrix with dimension "2 x d", where "d" is the dimension
   of
%    the space in which it is embedded the domain.
%    For instance, for a 2-sphere, it is "d=3", for a 2 dimensional
%    polygon it is "d=2".
%    As example, the set "[-1,1] x [0,1]" is described as
%                          "dbox=[-1 0; 1 1]".

% Data:
% Written on 26/07/2020 by M. Dessole, F. Marcuzzi, M. Vianello.
% Last update by:
% 04/01/2020: A. Sommariva.

% ..........................  Function Body
   ..............................

% ..... troubleshooting .....
if nargin < 7, method=1; end
if nargin < 6, dim_poly=[]; end
if nargin < 5, verbose=[]; end
if nargin < 4, LHDM_options=[]; end
if nargin < 3, u=[]; end

if isempty(verbose), verbose=0; end
if isempty(LHDM_options)
    dim = size(X,2);
    LHDM_options = struct( 'k', ceil(nchoosek(2*deg+dim,dim)/(deg*dim))
   ,...
        'init', false, 'thres', 0.2222, 'thres_w', 0.8);

end
if isempty(u), u=1; end
```

```
if not(isempty(dim_poly))
    if length(u) <= dim_poly
        nodes=X; w=u; momerr=0; dbox=[];
        return
    end
end



% ..... Main code below .....

% Vandermonde-like matrix of a u-orthogonal polynomial basis on X
[U,~,~,~,dbox]=dORTHVAND(deg,X,u,[],[],[],dim_poly);

if verbose
    fprintf('Vandermonde matrix size = %d x %d \n', size(U,1), size(U,2)
   );
end

if size(U,1)<=size(U,2)
    if verbose
        fprintf('Vandermonde matrix not underdetermined: no compression'
   );
        fprintf('\n');
    end
    % no compression expected
    nodes=X;

    if isscalar(u), u=u*ones(size(Q,1),1); end % weights saved as scalar
    w=u;
    momerr = 0;

else

    % further orthogonalization to reduce the conditioning
    [Q,~]=qr(U,0);

    % new moments
    if isscalar(u), u=u*ones(size(Q,1),1); end % weights saved as scalar
    orthmom=Q'*u;
    [nodes, w, momerr]= NNLS(X, u, U, Q, orthmom,method);

end



function [nodes, w, momerr]= NNLS(X, u, U, Q, orthmom,method)

% Object:
% Caratheodory-Tchakaloff points and weights via accelerated NNLS

if nargin < 6, method=2; end
```

```
switch method
    case 1
        weights = lsqnonneg(Q',orthmom);
    case 2
        weights=LHDM(Q',orthmom);
end

% indexes of nonvanishing weights and compression
ind=find(abs(weights)>0);
nodes=X(ind,:);
w=weights(ind);

momerr=norm(U(ind,:)'*w-U'*u);

% tic;
% if  isfield(options,'lsqnonneg')
%     if options.lsqnonneg
%         if verbose
%             fprintf('Matlab lsqnonneg \n');
%         end
%         [weights,~,~,output] = lsqnonneg(Q',orthmom);
%         iter = output.iterations;
%         cardP = [];
%     else
%         [weights,~,~,iter]=LHDM(Q',orthmom,options,verbose);
%     end
% else
%     [weights,~,~,iter]=LHDM(Q',orthmom,options,verbose);
% end
% e = toc;
%
% % indexes of nonvanishing weights and compression
% ind=find(abs(weights)>0);
% nodes=X(ind,:);
% w=weights(ind);
%
% % moment reconstruction error
% momerr=norm(U(ind,:)'*w-U'*u);
%
% % displaying results
% if verbose
%     fprintf('NNLS number of outer iterations = %d \n', iter);
%     fprintf('NNLS elapsed time = %.6f s  \n', e);
%     fprintf('initial design cardinality = %4.0f \n',size(X,1));
%     fprintf('concentrated support cardinality = %4.0f \n',length(w));
%     fprintf('compression ratio = %4.0f \n',size(X,1)/length(w));
%     fprintf('moment reconstruction error = %4.2e \n \n',momerr);
% end
```