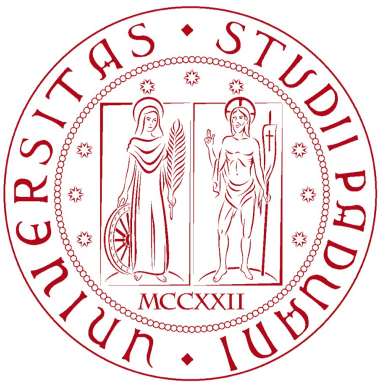

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Corso di Laurea Triennale in Matematica

Tesi di Laurea

Formule di quadratura algebriche per funzioni razionali

Relatore:
Prof. Alvise Sommariva

Laureando: Nicoló Zorzi
Matricola: 1146725

Anno Accademico 2023/2024
19/07/2024

Introduzione

Il proposito di questo lavoro è di studiare la performance di alcuni algoritmi per il calcolo di integrali di funzioni razionali prive di poli nell'intervallo di integrazione.

Da alcuni articoli é ben noto che la presenza di poli di funzioni razionali vicini agli estremi di integrazione puó richiedere l'uso di precisione estesa per ottenere i risultati desiderati ed intendiamo vedere sperimentalmente quanto questo asserto sia vero.

Sul tema di integrazione numerica di funzioni razionali sono state individuate varie opzioni. W. Gautschi ha proposto delle formule di tipo Gaussiano per una certa classe di funzioni razionali, fornendo open-source i codici Matlab che ne permettono il calcolo. La loro implementazione é particolarmente complessa, specie nel calcolo della matrice di Jacobi richiesta dall'algoritmo di Golub-Welsch per poter ottenere i nodi e pesi di quadratura.

Una alternativa che tratta una famiglia piú vasta di problemi é stata proposta da K. Deckers, A. Mougaida, H. Belhadjsalah in *Algorithm 973: Extended Rational Fejér Quadrature Rules Based on Chebyshev Orthogonal Rational Functions*. Gli autori forniscono codici Matlab di natura open-source, di utilizzo non immediato, vista la difficoltá dell'utente nel caratterizzare il problema.

Esistono pure routines alternative. In Matlab l'utente puó considerare la routine built-in `Integral`, tratta da una versione di L. Shampine, che calcola adattivamente l'integrale definito richiesto. Differentemente nel modulo Chebfun, in cui si approssima l'integranda mediante una funzione di natura polinomiale a tratti, anche individuando le sue singolaritá, si puó calcolare l'integrale definito richiesto mediante la funzione `sum`.

In questa tesi, dopo alcune premesse e una descrizione dei singoli algoritmi, abbiamo effettuato dei test numerici per vedere le performances dei singoli algoritmi, con integrande aventi poli esterni ma prossimi agli estremi di integrazione dell'intervallo di definizione $(-1, 1)$. In particolare mostreremo le tolleranze raggiunte e i tempi di calcolo, evidenziando quando il singolo algoritmo non abbia raggiunto lo scopo richiesto.

Il codice Matlab utilizzato é disponibile open-source presso GitHub [16].

Capitolo 1

Premesse

Il proposito di questa sezione è di introdurre le nozioni di base utili a comprendere come approssimare integrali definiti di una vasta famiglia di funzioni razionali.

In particolare mostreremo come determinare numericamente le formule gaussiane, procedendo con lo studio e analisi del condizionamento, e infine introdurremo l'algoritmo di Chebyshev.

Il contenuto di questa sezione prende spunto essenzialmente dalle monografie di Walter Gautschi [6], [4].

1.1 Nozioni preliminari

Sia $d\lambda = w(t)dt$ una misura definita su un intervallo $I \subset \mathbb{R}$, con $w(t)$ funzione peso. Ricordiamo che:

Definizione 1 *La funzione $w : I \mapsto \mathbb{R}$ è una funzione peso se:*

- *w non negativa in $\mathbb{R}$,*
- *per ogni $k \in \mathbb{N}$ esiste ed è finito $\int_{\mathbb{R}} |t|^k w(t)dt$,*
- *se $\int_{\mathbb{R}} g(t)w(t)dt = 0$ per una qualche funzione continua e non negativa g, allora $g = 0$ sul supporto di $d\lambda$.*

Definiamo per u, $v \in L_{2,w}$ il prodotto interno rispetto alla misura $d\lambda$

$$(u, v) = \int_{\mathbb{R}} u(t)v(t)d\lambda(t),$$

e la norma di $u \in L_{2,w}$

$$\|u\| = \sqrt{(u, u)}.$$

Definizione 2 *Il prodotto interno $(\cdot,\cdot)$ è definito positivo su $\mathbb{P}$, lo spazio dei polinomi reali, se $\|u\| > 0$ per ogni $u \in \mathbb{P}$ con $u \neq 0$. $(\cdot,\cdot)$ è definito positivo su $\mathbb{P}_m$ se $\|u\| > 0$ per ogni $u \in \mathbb{P}_m$ con $u \neq 0$.*

Definizione 3 *I polinomi monici reali $\pi_k(t) = t^k + \ldots$, $k = 0, 1, \ldots$, sono i polinomi monici ortogonali rispetto alla misura $d\lambda$ se soddisfano*

$$(\pi_k, \pi_s) = 0 \text{ per } k \neq s \quad k, s = 0, 1, \ldots,$$
$$\|\pi_k\| > 0 \text{ per } k = 0, 1, \ldots.$$

Tali polinomi verranno indicati con $\pi_k(\cdot)$ o $\pi_k(\cdot, d\lambda)$. Elenchiamo alcune proprietà:

Teorema 1 *Se il prodotto interno $(\cdot,\cdot)$ è definito positivo su $\mathbb{P}$ allora esiste un'unica sequenza infinita $\{\pi_k\}_{k\in\mathbb{N}}$ di polinomi monici ortogonali rispetto a $d\lambda$.*

Il prodotto interno associato alla misura $d\lambda = w(t)dt$, con $w(t)$ funzione peso, è definito positivo in $\mathbb{P}$ pertanto esiste ed è unica la sequenza di polinomi monici ortogonali $\{\pi_k(\cdot, d\lambda)\}_{k\in\mathbb{N}}$.

Teorema 2 *Se il prodotto interno $(\cdot,\cdot)$ è definito positivo su $\mathbb{P}_m$ ma non in $\mathbb{P}_n$ per $n > m$ allora esiste un numero finito $m+1$ di polinomi monici ortogonali $\pi_0, \pi_1, \ldots, \pi_m$.*

Teorema 3 *Tutti gli zeri di π_k, $k \geq 1$, sono reali, semplici e si trovano all'interno di supp($d\lambda$).*

Teorema 4 *Sia $d\lambda(t) = w(t)dt$ una misura simmetrica, ovvero $w(t) = w(-t)$. Allora*

$$\pi_k(-t, d\lambda) = (-1)^k \pi(t, d\lambda), \quad k = 0, 1, \ldots.$$

Da questo teorema si evince che che se $d\lambda$ è una misura simmetrica π_k è pari se k è pari, mentre π_k è dispari se k è dispari. In ogni caso gli zeri di π_k sono simmetrici rispetto all'origine.

Si consideri ora una misura discreta

$$d\lambda_N = \sum_{v=1}^{N} w_{v,N} \delta_{t_{v,N}}(t)$$

il cui supporto consiste in N punti distinti $t_{1,N}, \ldots, t_{N,N}$ e i cui pesi $w_{1,N}, \ldots, w_{1,N}$ sono strettamente positivi.

Il prodotto interno associato a tale misura è

$$(u, v)_N = \int_{\mathbb{R}} u(t)v(t)d\lambda_N(t) = \sum_{v=1}^{N} w_{v,N} u(t_{v,N})v(t_{v,N}).$$

Tale prodotto è definito positivo in $\mathbb{P}_{N-1}$, ma non in $\mathbb{P}_n$, $n \geq N$. In tal caso esistono solo i polinomi ortogonali monici $\pi_0, \pi_1, \ldots, \pi_{N-1}$ rispetto alla misura discreta $d\lambda_N$.

1.2 Formule di quadratura gaussiane

Sia la formula di quadratura

$$I^n_{d\lambda}(f) := \sum_{v=1}^{n} w_v f(t_v) \approx \int_{\mathbb{R}} f(t) d\lambda(t) \tag{1.1}$$

con pesi $w_v \in \mathbb{R}$, e nodi $t_v \in \mathbb{R}$ a due a due distinti.

Definizione 4 *La formula di quadratura* (1.1) *ha grado di precisione m se*

$$I^n_{d\lambda}(p) := \sum_{v=1}^{n} w_v p(t_v) = \int_{\mathbb{R}} p(t) d\lambda(t) \quad \forall p \in \mathbb{P}_m.$$

Definizione 5 *Una formula di quadratura $I^n_{d\lambda}$ è detta interpolatoria se i pesi sono*

$$w_k = \int_{\mathbb{R}} L_k(t) w(t) dt, \quad k = 1, \ldots, n,$$

dove $L_k(t)$ è il k-esimo polinomio di Lagrange rispetto ai punti $t_1, \ldots, t_n$

$$L_k(t) = \prod_{i=1, i\neq k}^{n} \frac{(t - t_i)}{(t_k - t_i)}.$$

Si può dimostrare che una formula di quadratura è interpolatoria se e solo se ha grado di precisione $n - 1$.

Per ogni misura $d\lambda = w(t)dt$ ed n, esiste la formula di quadratura

$$\sum_{v=1}^{n} w^G_v p(t^G_v) = \int_{\mathbb{R}} p(t) d\lambda(t)$$

con grado di precisione $2n - 1$, detta *formula gaussiana*.

Teorema 5 *Per ogni $n \geq 1$ esistono, e sono unici, i nodi $t^G_1, \ldots, t^G_n$ e pesi $w^G_1, \ldots, w^G_n$ per cui la relativa formula di quadratura abbia grado di precisione $2n - 1$.*

I nodi sono gli zeri del polinomio ortogonale di grado n $\pi_n(\cdot, d\lambda)$, e i pesi sono

$$w^G_k = \int_{\mathbb{R}} L_k(t) w(t) dt = \int_{\mathbb{R}} L^2_k(t) w(t) dt, \quad k = 1, \ldots, n,$$

dove $L_k(t)$ è il k-esimo polinomio di Lagrange rispetto ai nodi $t^G_1, \ldots, t^G_n$

Si può osservare che i pesi sono strettamente positivi. Per quanto affermato dai teoremi (3) e (4), i nodi di una formula Gaussiana, sono distinti, reali e appartengono all'interno di supp($d\lambda$), inoltre se la misura $d\lambda$ è simmetrica allora i nodi sono simmetrici rispetto all'origine. Si può dimostrare che, a parità del numero di nodi, non è possibile ottenere una formula di quadratura con grado di precisione maggiore di $2n - 1$.

Formuliamo il seguente teorema di convergenza relativo alle formule Gaussiane, derivabile dal teorema di Polya-Steklov, provato da Stieltjes.

Teorema 6 *Sia $\{I^n_{d\lambda}\}_n$ una sequenza di formule Gaussiane a n punti, relative alla misura $d\lambda = w(t)dt$, con $w : I \mapsto \mathbb{R}$ funzione peso definita su un intervallo I limitato. Allora si ha*

$$E_n = \int_{\mathbb{R}} f(t)d\lambda(t) - I^n_{d\lambda}(f) \to 0, \quad \forall f \in C(I).$$

1.3 Algoritmo di Golub-Welsch

Si considerino i polinomi ortogonali monici di grado k per $k = 0, \ldots, n-1$. Essendo una base di polinomi monici di $\mathbb{P}_{n-1}$, si ha che

$$\pi_{k+1}(t) - t\pi_k(t) = -\alpha_k\pi_k(t) - \beta_k\pi_{k-1}(t) + \sum_{v=0}^{k-2} \gamma_{kv}\pi_v(t), \qquad k = 0, \ldots, n-1,$$

assumendo che $\pi_{-1}(t) = 0$. Applicando il prodotto interno per $\pi_k(t)$ a entrambi i membri dell'equazione, si ottiene $-(t\pi_k, \pi_k) = -\alpha_k(\pi_k, \pi_k)$, da cui

$$\alpha_k = \frac{(t\pi_k, \pi_k)}{(\pi_k, \pi_k)}, \quad k = 0, 1, \ldots, n-1.$$

Allo stesso modo, applicando il prodotto interno per π_{k-1}, con $k \geq 1$, si ottiene $-(t\pi_k, \pi_{k-1}) = -\beta_k(\pi_{k-1}, \pi_{k-1})$, e poiché $(t\pi_k, \pi_{k-1}) = (\pi_k, t\pi_{k-1}) = (\pi_k, \pi_k + q_{k-1}) = (\pi_k, \pi_k)$ (q_{k-1} è un polinomio con grado $< k$, perciò ortogonale a π_k), risulta

$$\beta_k = \frac{(\pi_k, \pi_k)}{(\pi_{k-1}, \pi_{k-1})}, \quad k = 1, 2, \ldots, n-1.$$

Infine, considerando di nuovo il prodotto interno per π_v, con $v \leq k-2$, sui membri dell'equazione iniziale, si ottiene

$$-(t\pi_k, \pi_v) = \gamma_{kv}(\pi_v, \pi_v),$$

ma $(t\pi_k, \pi_v) = (\pi_k, t\pi_v) = 0$, perché $t\pi_v$ ha grado minore di k, quindi $\gamma_{kv} = 0$. Si ottiene così la *formula di ricorsione a tre termini*

$$\pi_{k+1}(t) = (t - \alpha_k)\pi_k(t) - \beta_k\pi_{k-1}(t), \qquad k = 0, \ldots, n-1 \tag{1.2}$$
$$\pi_0(t) = 1, \qquad \pi_{-1}(t) = 0$$

con

$$\begin{aligned} \alpha_k &= \frac{(t\pi_k, \pi_k)}{(\pi_k, \pi_k)} \qquad k = 0, \ldots, n-1 \\ \beta_k &= \frac{(\pi_k, \pi_k)}{(\pi_{k-1}, \pi_{k-1})} \qquad k = 1, \ldots, n-1 \end{aligned} \tag{1.3}$$

soddisfatta dai polinomi ortogonali. Visto che β_0 non è definito dalla formula precedente, si assume usualmente

$$\beta_0 = \int_{\mathbb{R}} d\lambda(t).$$

Definita la *matrice di Jacobi*

$$J = J(d\lambda) = \begin{pmatrix} \alpha_0 & \sqrt{\beta_1} & & & \\ \sqrt{\beta_1} & \alpha_1 & \sqrt{\beta_2} & & \\ & \sqrt{\beta_2} & \alpha_2 & \sqrt{\beta_3} & \\ & & \ddots & \ddots & \ddots \end{pmatrix},$$

matrice simmetrica tridiagonale infinita, si indica con $J_n = J_n(d\lambda) = [J]_{n\times n}$ la sottomatrice contenente le prime n righe e n colonne di J. Vale il seguente teorema:

Teorema 7 *Siano $t_1^G, \ldots, t_n^G$ e $w_1^G, \ldots, w_n^G$ rispettivamente i nodi e pesi della formula gaussiana a n punti.*

Allora i nodi sono gli autovalori di J_n, invece per ogni w_k^G vale

$$w_k^G = \beta_0 u_{k,1}^2, \tag{1.4}$$

dove $\beta_0 = \int_{\mathbb{R}} d\lambda(t)$ e $u_{k,1}$ è la prima componente dell'autovettore normalizzato u_k relativo all'autovalore t_k^G

Basandosi sul teorema (7) l'algoritmo di Golub-Welsch determina le formule gaussiane ad n punti risolvendo un problema di calcolo degli autovalori e autovettori, rispetto a una matrice reale simmetrica tridiagonale. Esistono diversi metodi accurati e veloci per risolvere tale problema, ad esempio la routine Matlab `gauss.m`.

Osservazione 1 Sia $U = [u_1, \ldots, u_n]$ la matrice ortogonale che ha come colonne gli autovettori normalizzati di J_n. Dal momento che $t_1^G, \ldots, t_n^G$ sono gli autovalori di J_n, si ha che

$$J_n U = U D_t, \quad \text{con} \quad D_t = \operatorname{diag}(t_1^G, \ldots, t_n^G), \quad U^T U = I.$$

La formula (1.4) può essere scritta in forma vettoriale come

$$\sqrt{w^T} = \sqrt{\beta_0} e_1^T U \quad \text{con } \sqrt{w} = \left[\sqrt{w_1^G}, \ldots, \sqrt{w_n^G}\right]^T, \quad e_1 = [1, 0, \ldots, 0]^T.$$

Se $Q = U^T$ le due equazioni precedenti diventano

$$Q^T D_t Q = J_n, \quad Q^T \sqrt{w} = \sqrt{\beta_0} e_1.$$

Allora otteniamo

$$\begin{pmatrix} 1 & 0^T \\ 0 & Q^T \end{pmatrix} \begin{pmatrix} 1 & \sqrt{w^T} \\ \sqrt{w} & D_t \end{pmatrix} \begin{pmatrix} 1 & 0^T \\ 0 & Q \end{pmatrix} = \begin{pmatrix} 1 & \sqrt{w^T} Q \\ Q^T \sqrt{w} & Q^T D_t Q \end{pmatrix}$$
$$= \begin{pmatrix} 1 & \sqrt{\beta_0} e_1^T \\ \sqrt{\beta_0} e_1 & J_n \end{pmatrix}. \tag{1.5}$$

Questa similitudine tra matrici tramite una trasformazione ortogonale suggerisce, sebbene non dimostri, che il passaggio dai nodi e pesi di Gauss ai coefficienti di ricorrenza è un processo ben condizionato. Inoltre la stessa formula permette di derivare l'algoritmo di Lanczos di cui vedremo un'applicazione nel prossimo capitolo.

1.4 Procedura di Stieltjes e algoritmo di Lanczos

Al fine di calcolare i coefficienti di ricorrenza presentiamo due diversi procedimenti.

Il primo di questi è la *procedura di Stieltjes*. Dal momento che $\pi_0 = 1$ è noto, si ricava α_0 da (1.3).

Conoscendo α_0 e β_0, applicando la relazione (1.2) per $k = 1$, si ricava π_1. Applicando nuovamente (1.3), per $k = 1$, si ricava α_1 e β_1, con i quali si calcola π_2 dalla formula (1.2) per $k = 1$. Continuando il procedimento fino a $k = n - 1$ si ricavano i coefficienti della matrice J_n. La maggiore difficoltà di questo metodo è il calcolo dei prodotti interni nelle formule (1.3). Questo richiede l'integrazione rispetto alla misura $d\lambda$, cosa non banale specialmente per misure "non standard".

Consideriamo il caso particolare in cui la misura $d\lambda_N$ sia discreta con nodi distinti $t_{1,N}, \ldots, t_{N,N}$, e pesi $w_{v,N} \geq 0$ per $v = 1, \ldots, N$. Siano $\sqrt{w} = [\sqrt{w_{1,N}}, \ldots, \sqrt{w_{N,N}}]^T$, $e_1 = [1, 0, \ldots, 0]^T \in \mathbb{R}^N$, la matrice diagonale $D_t = diag(t_{1,N}, \ldots, t_{N,N})$, e la matrice di Jacobi $J_N(d\lambda_N) \in \mathbb{R}^{N \times N}$ relativa alla misura $d\lambda_N$. Dal momento che si definisce

$$\int_{\mathbb{R}} p(t) d\lambda_N = \sum_{v=1}^{N} w_{v,N} p(t_{v,N})$$

per ogni polinomio p (quindi anche per ogni polinomio di grado $\leq 2N - 1$), questa stessa formula può essere interpretata come formula Gaussiana rispetto alla misura $d\lambda_N$. Per quanto discusso nell'osservazione (1) esiste una matrice ortogonale $Q_N \in \mathbb{R}^{N \times N}$ tale per cui

$$\begin{pmatrix} 1 & 0^T \\ 0 & Q_N^T \end{pmatrix} \begin{pmatrix} 1 & \sqrt{w^T} \\ \sqrt{w} & D_t \end{pmatrix} \begin{pmatrix} 1 & 0^T \\ 0 & Q_N \end{pmatrix} = \begin{pmatrix} 1 & \sqrt{\beta_0(d\lambda_N)} e_1^T \\ \sqrt{\beta_0(d\lambda_N)} e_1 & J_N(d\lambda_N) \end{pmatrix}. \tag{1.6}$$

In generale, data una matrice reale simmetrica A e una matrice tridiagonale T per cui gli elementi che non stanno sulla diagonale sono non negativi, esiste sempre una matrice ortogonale Q per cui $Q^T A Q = T$, inoltre Q e T sono univocamente determinate da A e la prima colonna di Q. L'*algoritmo di Lanczos* è in grado di produrre le matrici Q e T a partire da A. La struttura e dimostrazione dell'algoritmo sono esposte dettagliatamente in [6] (pg. 212-213). Osservando che la formula (1.6) è scritta esattamente nella forma $Q^T A Q = T$, e che conosciamo la prima colonna di Q ($e_1 \in \mathbb{R}^{N+1}$) e la matrice tridiagonale A (data $d\lambda_N$, conosciamo automaticamente D_t e $\sqrt{w}$), la matrice T, e in particolare $J_N(d\lambda_N)$, può essere ricavata utilizzando l'algoritmo di Lanczos.

1.5 Metodo dei momenti

Esiste un metodo, basato sui momenti

$$\mu_k = \int_{\mathbb{R}} t^k d\lambda(t) \qquad k = 0, 1, \dots$$

che permette di esprimere i coefficienti di ricorrenza in termini dei determinanti di Hankel:

$$\begin{array}{c} \alpha_k(d\lambda) = \frac{D'_{k+1}}{D_{k+1}} - \frac{D'_k}{D_k} \\ \beta_k(d\lambda) = \frac{D_{k+1} Dk-1}{D_k^2} \end{array} \qquad k = 0, 1, 2, \dots$$

dove

$$D_{-1} = D_0 = 1, \quad D_1 = \mu_0, \quad D_m = \begin{vmatrix} \mu_0 & \mu_1 & \dots & \mu_{m-1} \\ \mu_1 & \mu_2 & \dots & \mu_m \\ \dots & \dots & \dots & \dots \\ \mu_{m-1} & \mu_m & \dots & \mu_{2m-2} \end{vmatrix} \quad m \geq 2$$

$$D'_0 = 0, \quad D'_1 = \mu_1, \quad D'_m = \begin{vmatrix} \mu_0 & \mu_1 & \dots & \mu_{m-2} & \mu_m \\ \mu_1 & \mu_2 & \dots & \mu_{m-1} & \mu_{m+1} \\ \dots & \dots & \dots & \dots & \dots \\ \mu_{m-1} & \mu_m & \dots & \mu_{2m-3} & \mu_{2m-1} \end{vmatrix} \quad m \geq 2$$

Non è conveniente usare queste formule, in primo luogo perché richiedono il calcolo di un determinante o la fattorizzazione triangolare della rispettiva matrice, per cui il costo computazionale può diventare molto alto. Inoltre, e spesso si tratta del problema più rilevante, il calcolo dei coefficienti a partire dai momenti della misura $d\lambda$ può essere fortemente mal condizionato.

In altre parole non è conveniente usare la mappa

$$K_n : \mathbb{R}^{2n} \to \mathbb{R}^{2n} \qquad \mu \mapsto \rho$$

con

$$\mu = \mu(d\lambda) = [\mu_0, \mu_1, \dots, \mu_{2n-1}]$$

il vettore dei primi $2n$ momenti, e

$$\rho = \rho(d\lambda) = [\alpha_0, \dots, \alpha_{n-1}, \beta_0, \dots, \beta_{n-1}]$$

vettore dei primi $2n$ coefficienti di ricorrenza, a causa del suo alto numero di condizionamento.

Un modo per superare questi problemi è di implementare la mappa K_n a partire dai momenti modificati m_k rispetto a un sistema di polinomi $\{p_k\}_k$, così definiti:

$$m_k = \int_{\mathbb{R}} p_k(t) d\lambda(t) \qquad k = 0, 1, 2, \dots.$$

A seconda della scelta del sistema di polinomi e quindi dei momenti m_k, la nuova mappa

$$K_n : \mathbb{R}^{2n} \to \mathbb{R}^{2n} \qquad m \mapsto \rho,$$

con $m = [m_0, m_1, \dots, m_{2n-1}]$, si spera sia meglio condizionata della mappa con i momenti classici.

1.6 Condizionamento di K_n

Per analizzare il condizionamento di K_n conviene introdurre le applicazioni

$$\begin{aligned} &G_n : \mathbb{R}^{2n} \to \mathbb{R}^{2n} \qquad \mu \text{ (oppure } m) \mapsto \gamma \\ &H_n : \mathbb{R}^{2n} \to \mathbb{R}^{2n} \qquad \gamma \mapsto \rho, \\ &\text{per cui valga } K_n = H_n \circ G_n \end{aligned}$$

con $\gamma = \gamma(d\lambda) = [t_1^G, \ldots, t_n^G, w_1^G, \ldots, w_n^G]$ il vettore dei nodi e pesi della formula gaussiana a n punti. Essendo H_n un'applicazione ben condizionata, il condizionamento di K_n sarà simile a quello di G_n.

Cominciamo con il definire il numero di condizionamento per funzioni in più variabili:

Definizione 6 *Se $f : \mathbb{R}^m \to \mathbb{R}^n$ è una funzione Fréchet-differenziabile in $x \in \mathbb{R}^m$, si definisce*

$$(\text{cond}_{abs} f)(x) = \left\| \frac{\partial f(x)}{\partial x} \right\|$$

il numero di condizionamento assoluto *di f in x rispetto a un'appropriata norma matriciale.*

Partiamo dallo studio del condizionamento di $G_n : \mu \mapsto \gamma$ e calcoliamo il numero di condizionamento con norma infinito. Dal momento che le formule gaussiane sono esatte in $\mathbb{P}_{2n-1}$ allora

$$\sum_{v=1}^{n} w_v^G (t_v^G)^r = \int_{\mathbb{R}} t^r d\lambda(t) = \mu_r, \qquad r = 0, 1, \ldots, 2n-1.$$

Ciò significa che la mappa G_n equivale a risolvere il sistema non lineare di incognite i vettori $w^G = [w_1^G, \ldots, w_n^G]$, e $t^G = [t_1^G, \ldots, t_n^G]$. Se $F_n : \gamma \mapsto \mu$ è la mappa definita da tale sistema

$$F(\gamma) = \mu, \qquad F_k(\gamma) = \sum_{v=1}^{n} w_v^G (t_v^G)^k, \quad k = 0, 1, \ldots, 2n-1$$

allora risulta

$$\frac{\partial F}{\partial \gamma} = T\Lambda,$$

dove Λ è la matrice diagonale di ordine $2n$

$$\Lambda = \text{diag}(1, \ldots, 1, w_1^G, \ldots, w_n^G),$$

e T è la matrice di confluenza di Vandermonde rispetto alla famiglia di polinomi $\{t^k\}_k$

$$T = \begin{pmatrix} 1 & \ldots & 1 & 0 & \ldots & 0 \\ t_1^G & \ldots & t_n^G & 1 & \ldots & 1 \\ (t_1^G)^2 & \ldots & (t_n^G)^2 & 2t_1^G & \ldots & 2t_n^G \\ \vdots & & \vdots & \vdots & & \vdots \\ (t_1^G)^{2n-1} & \ldots & (t_n^G)^{2n-1} & (2n-1)(t_1^G)^{2n-2} & \ldots & (2n-1)(t_n^G)^{2n-2} \end{pmatrix}$$

Essendo G_n la mappa inversa di F_n, si ha

$$\frac{\partial G_n}{\partial \mu} = \left(\frac{\partial F}{\partial \gamma}\right)^{-1} = \Lambda^{-1}T^{-1},$$

pertanto

$$(\mathrm{cond}G_n)(\mu) = \|\Lambda^{-1}T^{-1}\|.$$

Ora, scegliendo come norma matriciale $\|\cdot\| = \|\cdot\|_\infty$, e poiché $\sum_{v=1}^n w_v^G = \mu_0$ implica che $(w_v^G)^{-1} > \mu_0^{-1}$, segue che

$$(\mathrm{cond}G_n)(\mu) > \min(1, \mu_0^{-1})\|T^{-1}\|.$$

Se $\mathrm{supp}(d\lambda) \subset \mathbb{R}_+$, tramite stime sulla norma di T^{-1}, si ottiene

$$(\mathrm{cond}G_n)(\mu) > \frac{\prod_{v=1}^n (1+t_v^G)^2}{\min_{1\le v\le n}\left\{(1+t_v^G)\prod_{s=1,s\neq v}^n (t_v^G - t_s^G)^2\right\}}$$

e dal fatto che i punti t_v^G sono gli zeri dei polinomi ortogonali $\pi_n(\cdot, d\lambda)$,

$$(\mathrm{cond}G_n)(\mu) > \frac{\pi_n^2(-1)}{\min_{1\le v\le n}\{(1+t_v^G)[\pi_n'(t_v^G)]^2\}}.$$

Dal momento che i polinomi ortogonali crescono rapidamente, rispetto al grado, quando l'argomento è fuori dal supporto della misura, il numeratore, che inoltre è elevato al quadrato, cresce molto rapidamente, invece il denominatore cresce moderatamente rispetto a n. Possiamo concludere che G_n diventa rapidamente mal condizionata al crescere di n.

Consideriamo la mappa $G_n : m \mapsto \gamma$, dove $m \in \mathbb{R}^{2n}$ è il vettore dei momenti modificati. Supponiamo che il sistema di polinomi $\{p_k\}_k$, che definisce i momenti, sia un sistema di polinomi ortogonali rispetto a una misura ds.

Per comodità, analizzeremo il condizionamento di $\tilde{G}_n : \tilde{m} \mapsto \gamma$, con

$$\tilde{m} = [\tilde{m}_0, \tilde{m}_1, \ldots, \tilde{m}_{2n-1}], \quad \tilde{m}_k = \frac{m_k}{\|p_k\|_{ds}}, \quad \|p_k\|_{ds} = \sqrt{(p_k, p_k)_{ds}}.$$

Si precisa che la trasformazione diagonale $m \mapsto \tilde{m}$ è ben condizionata, pertanto non modifica il condizionamento di G_n. Questa volta si studierà il numero di condizionamento di $\tilde{G}_n$ rispetto alla norma di Frobenius $\|\cdot\| = \|\cdot\|_F$. Inoltre, per ogni nodo t_v^G della formula gaussiana a n punti, indichiamo con h_v e k_v i polinomi elementari interpolatori di Hermite associati ai nodi $t_1^G, \ldots, t_n^G$, che soddisfano

$$\begin{array}{ll} h_v(t_u^G) = \delta_{vu}, & h_v'(t_u^G) = 0, \\ k_v(t_u^G) = 0, & k_v'(t_v^G) = \delta_{vu}, \end{array} \qquad v, u = 1, 2, \ldots, n.$$

Seguendo un procedimento simile a quello utilizzato per calcolare il condizionamento di $\tilde{G}_n$ segue il teorema:

Teorema 8 *Il numero di condizionamento assoluto di $\tilde{G}_n$ rispetto alla norma di Frobenius è dato da*

$$(\text{cond}\tilde{G}_n)(\tilde{m}) = \left\{ \int_{\mathbb{R}} g_n(t; d\lambda) ds(t) \right\}^{\frac{1}{2}},$$

dove

$$g_n(t; d\lambda) = \sum_{v=1}^{n} \left(h_v^2(t) + \frac{1}{\lambda_v^2} k_v^2(t) \right)$$

e h_v e k_v sono i polinomi elementari di hermite interpolatori rispetto ai nodi di Gauss $t_1^G, t_2^G, \ldots, t_n^G$.

Da questo teorema si evince che cond$\tilde{G}_n$ è determinato dalla misura $d\lambda$, dalla quale dipende il polinomio $g_n(\cdot; d\lambda)$, e dalla misura ds, rispetto alla quale viene integrato il polinomio stesso. Inoltre, dalle proprietà

$$\begin{aligned} &g_n(t) > 0 \quad \text{in } \mathbb{R} \\ &g_n(t_v^G) = 1, \quad g_n'(t_v^G) = 0, \quad v = 0, 1, \ldots, n, \end{aligned}$$

viene suggerito che g_n non si discosta di molto da 1 sul supporto di $d\lambda$, specialmente se il supporto di quast'ultima è un intervallo finito, e i nodi t_v^G hanno una distribuzione ad arccos nell'intervallo. g_n cresce molto rapidamente fuori da supp$(d\lambda)$, per cui è ragionevole scegliere ds che abbia supporto contenuto in supp$(d\lambda)$

1.7 Algoritmo di Chebyshev modificato

Sviluppiamo ora l'algoritmo di Chebyshev, che a partire dal vettore m dei primi $2n$ momenti modificati restituisce il vettore ρ dei primi $2n$ coefficienti di ricorrenza, cioè implementa la mappa $K_n : m \to \rho$.

Partiamo considerando la famiglia di polinomi monici che soddisfa la relazione di ricorrenza a tre termini

$$\begin{aligned} p_{k+1}(t) &= (t - a_k) p_k(t) - b_k p_{k-1}(t), \quad k = 0, 1, \ldots, \\ p_0(t) &= 1, \quad p_{-1}(t) = 0, \end{aligned} \tag{1.7}$$

con $a_k \in \mathbb{R}$ e $b_k \geq 0$ (i momenti m_k saranno calcolati rispetto ai polinomi p_k). Se $a_k = b_k = 0$ si otterrebbe l'algoritmo di Chebyshev originale che implementa la mappa $K_n : \mu \to \rho$, e che, come già mostrato, è mal condizionata.
Introduciamo il *'momento misto'*

$$\sigma_{k,l} = \int_{\mathbb{R}} \pi_k(t) p_l(t) d\lambda(t).$$

Si può notare che $\sigma_{k,l} = 0$ se $k > l$, e

$$\int_{\mathbb{R}} \pi_k(t)^2 d\lambda(t) = \int_{\mathbb{R}} \pi_k(t) t p_{k-1}(t) d\lambda(t) = \sigma_{k,k} \qquad k \geq 1.$$

Dalla relazione $\sigma_{k+1,k-1} = 0$ si ottiene

$$0 = \int_{\mathbb{R}} ((t - \alpha_k)\pi_k(t) - \beta_k \pi_{k-1}(t)) p_{k-1}(t) d\lambda(t),$$

quindi

$$\begin{aligned} \beta_k &= \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}} \qquad k = 1, 2, \ldots, \\ (\beta_0 &= m_0). \end{aligned}$$

Allo stesso modo,

$$\begin{aligned} 0 = \sigma_{k+1,k} &= \int_{\mathbb{R}} ((t - \alpha_k)\pi_k(t) - \beta_k \pi_{k-1}(t))\, p_k(t) d\lambda(t) \\ &= \int_{\mathbb{R}} \pi_k(t) t p_k(t) d\lambda(t) - \alpha_k \sigma_{k,k} - \beta_k \sigma_{k-1,k}. \end{aligned}$$

Usando la relazione $tp_k(t) = p_{k+1}(t) + a_k p_k(t) + b_k p_{k-1}(t)$ si può scrivere

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

da cui

$$\begin{aligned} \alpha_0 &= a_0 + \frac{\sigma_{0,1}}{\sigma_{0,0}} \\ \alpha_k &= a_k - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}} + \frac{\sigma_{k,k+1}}{\sigma_{k,k}} \qquad k = 1, 2, \ldots \end{aligned}$$

Calcoliamo $\sigma_{k,l}$:

$$\begin{aligned} \sigma_{k,l} &= \int_{\mathbb{R}} ((t - \alpha_{k-1})\pi_{k-1}(t) - \beta_{k-1}\pi_{k-2}(t)) p_l(t) d\lambda(t) \\ &= \int_{\mathbb{R}} \pi k - 1(t)(p_{l+1}(t) + a_l p_l(t) + b_l p_{l-1}(t)) d\lambda(t) - \alpha_{k-1}\sigma_{k-1,l} - \beta_{k-1}\sigma_{k-2,l} \\ &= \sigma_{k-1,l+1} - (\alpha_{k-1} - a_l)\sigma_{k-1,l} - \beta_{k-1}\sigma_{k-2,l} + b_l \sigma_{k-1,l-1}. \end{aligned}$$

E' possibile ora scrivere l'algoritmo di Chebyshev modificato per calcolare α_k, β_k con $k = 0, 1, \ldots, n-1$:

Inizializzazione:

$$\begin{aligned} \alpha_0 &= a_0 + \frac{m_1}{m_0}, \\ \beta_0 &= m_0, \\ \sigma_{-1,l} &= 0 \qquad l = 1, 2, \ldots, 2n-2, \\ \sigma_{0,l} &= m_l \qquad l = 0, 1, \ldots, 2n-1. \end{aligned}$$

Continuazione (se $n > 1$) per $k = 1, 2, \ldots, n-1$:

$$\begin{aligned}
\sigma_{k,l} =& \sigma_{k-1,l+1} - (\alpha_{k-1} - a_l)\sigma_{k-1,l} - \beta_{k-1}\sigma_{k-2,l} + b_l\sigma_{k-1,l-1}, \\
& l = k, k+1, \ldots, 2n-k-1, \\
\alpha_k =& a_k - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}} + \frac{\sigma_{k,k+1}}{\sigma_{k,k}}, \\
\beta_k =& \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}}.
\end{aligned}$$

L'algoritmo richiede come input $\{m_l\}_{l=0}^{2n-1}$ e $\{a_k, b_k\}_{k=0}^{2n-2}$, e calcola $\{\alpha_k, \beta_k\}_{k=0}^{n-1}$, con una complessità in termini di operazioni aritmetiche di $\mathscr{O}(n^2)$.

La bontà dell'algoritmo dipende anche dall'efficienza del calcolo dei momenti modificati, per il quale è possibile applicare metodi diversi a seconda della misura e della famiglia di polinomi scelti, come vedremo nell'ultimo capitolo.

Capitolo 2

Metodi di integrazione

In questo capitolo verranno esposti diversi metodi di integrazione che verranno applicati al calcolo di integrali di funzioni razionali nel prossimo capitolo. Tra questi troviamo le formule di quadratura di Gauss razionale, le routine `integral` di Matlab e `rfejer`, la formula di Gauss-Legendre e alcuni metodi di integrazione di Chebfun e Mathematica.

2.1 Formule di quadratura razionali

Si supponga di dover calcolare l'integrale di una funzione che sia regolare all'interno dell'intervallo di integrazione, mentre al di fuori presenti un certo numero di poli. È ragionevole usare per un problema di questo tipo delle formule di quadratura che siano esatte, non solo per polinomi, ma anche per funzioni razionali che abbiano gli stessi poli della funzione da integrare, o per lo meno quelli più vicini all'intervallo di integrazione. Quanto introdotto in questa sezione é tratto da [5], e [3].

Sia $d\lambda = w(t)dt$ una misura sulla retta reale, con $w(t)$ funzione peso. Siano inoltre $\zeta_\mu \in \mathbb{C}$, $\mu = 1, \ldots, M$ distinti e $s_\mu \geq 1$ per cui

$$\zeta_\mu \neq 0,\ 1 + \zeta_\mu t \neq 0 \text{ per } t \in \overline{\text{supp}(d\lambda)},\ \text{e}\ \sum_{\mu=1}^{M} s_\mu = m.$$

Con queste ipotesi, formuliamo il seguente teorema, che ha lo scopo di definire delle formule di quadratura esatte per funzioni razionali:

Teorema 9 *Siano m, $d \in \mathbb{N}$ per cui valga $0 \leq m \leq d$ e il polinomio di grado m*

$$\omega_m(t) = \prod_{\mu=1}^{M} (1 + \zeta_\mu t)^{s_\mu}.$$

Sia $\mathbb{Q}_{\omega_m} = \text{span}\{g : g(t) = (1+\zeta_\mu t)^{-s},\ \mu = 1,\dots,M,\ s = 1,\dots,s_\mu\}$. *e assumiamo che esista la formula di quadratura*

$$\sum_{v=1}^{n} w_v^* f(t_v^*) \approx \int_{\mathbb{R}} f(t)\frac{d\lambda(t)}{\omega_m(t)}, \tag{2.1}$$

rispetto alla misura modificata $d\lambda/\omega_m$, *con grado di precisione* $d-1$, *e che i nodi* t_v^* *siano contenuti nel supporto di* $d\lambda$.

Definiti

$$t_v = t_v^*, \quad w_v = w_v^*\omega_m(t_v^*), \quad v = 1,\dots,n \tag{2.2}$$

allora

$$\sum_{v=1}^{n} w_v f(t_v) \approx \int_{\mathbb{R}} f(t)d\lambda(t), \tag{2.3}$$

è una formula di quadratura esatta per le funzioni

$$f \in \mathbb{S}_d = \mathbb{Q}_{\omega_m} \oplus \mathbb{P}_{d-m-1}. \tag{2.4}$$

Viceversa, se una formula di quadratura del tipo (2.3), che soddisfi $t_v \in supp(d\lambda)$, *è esatta per (2.4), allora esiste la formula (2.1) con* t_v^*, w_v^* *definiti da (2.2).*

Si noti che i poli delle funzioni razionali in $\mathbb{S}_d$ sono $-\frac{1}{\zeta_\mu}$, hanno molteplicità s_μ, e si trovano al di fuori della chiusura del supporto di $d\lambda$.

Per integrare una funzione f continua in $\text{supp}(d\lambda)$, scegliamo i valori ζ_μ e s_μ in modo che i poli e le rispettive molteplicità delle funzioni in $\mathbb{S}_d$ coincidano con i poli più vicini a $\text{supp}(d\lambda)$ e le rispettive molteplicità di f. Motiviamo questa affermazione stimando l'errore commesso dalla formula (2.3).

Dalle formule (2.1) e (2.2) segue

$$\int \frac{p(t)}{\omega_m(t)}d\lambda(t) = \sum_{v=1}^{n} w_v \frac{p(t_v)}{\omega_m(t_v)}, \quad p \in \mathbb{P}_{d-1}.$$

Sia

$$\mathscr{E}_{d,m}(f) = \inf_{p\in\mathbb{P}_{d-1}} \left\| \frac{p}{\omega_m} - f \right\|_\infty = \left\| \frac{p^*}{\omega_m} - f \right\|_\infty$$

per qualche p^* appartenente a $\mathbb{P}_{d-1}$. Allora l'errore commesso dalla formula (2.3) è

$$\begin{aligned} |E_n(f)| &= \left| \int_{\mathbb{R}} f(t)d\lambda(t) - \sum_{v=1}^{n} w_v f(t_v) \right| \\ &= \left| \int_{\mathbb{R}} \left(f(t) - \frac{p^*(t)}{\omega_m(t)} \right) d\lambda(t) - \sum_{v=1}^{n} w_v \left(f(t_v) - \frac{p^*(t_v)}{\omega_m(t_v)} \right) \right| \\ &\le \mathscr{E}_{d,m}(f) \left(\int_{\mathbb{R}} d\lambda(t) + \sum_{v=1}^{n} |w_v| \right). \end{aligned}$$

Dalla stima di $E_n(f)$ si può dedurre che se f è una funzione che può essere ben approssimata su $\text{supp}(d\lambda)$ da una funzione p/ω_m, con p polinomio, o equivalentemente $f\omega_m$ può essere ben approssimata da un polinomio p, allora l'errore commesso dalla formula risulterà piccolo. Moltiplicando f per ω_m con un'opportuna scelta degli zeri di ω_m, è possibile "rimuovere" da f i poli più vicini a $\text{supp}(d\lambda)$, pertanto la risultante funzione $f\omega_m$, che avrà i poli più distanti dal dominio di integrazione, sarà più facilmente approssimabile da un polinomio.

2.2 Formule di quadratura di Gauss razionale

Costruiamo una formula di quadratura del tipo (2.3), scegliendo $d = 2n$ e $0 \leq m \leq 2n$. Chiameremo una formula di quadratura di questo tipo formula di *Gauss razionale*.

Per il teorema precedente questa dipende dall'esistenza della formula Gaussiana rispetto alla misura $d\hat{\lambda} = d\lambda/\omega_m$

$$\sum_{v=1}^{n} w_v^G p(t_v^G) = \int_{\mathbb{R}} p(t) d\hat{\lambda}(t), \quad p \in \mathbb{P}_{2n-1}, \tag{2.5}$$

Supponiamo che ζ_μ con $\mu = 1, \ldots, M$ siano valori reali, e se per qualche k ζ_k è un numero complesso con molteplicità s_k, allora esiste s per cui ζ_s è il complesso coniugato di ζ_k con molteplicità $s_s = s_k$. In tal caso i poli includeranno, oltre ai poli reali, il rispettivo complesso coniugato di ogni polo complesso con la stessa molteplicità. Dunque ω_m sarà un polinomio reale, e se $\text{supp}(d\lambda)$ è connesso, allora $1/\omega_m$ è reale, limitato e a segno costante sul supporto di $d\lambda$, pertanto la formula (2.5) esiste ed è unica.

Possiamo determinare i nodi e pesi della formula (2.5) con l'algoritmo di Golub-Welsch, infatti t_v^G con $v = 1, \ldots, n$ sono gli autovalori della matrice di Jacobi

$$\hat{J}_n = J_n(d\hat{\lambda}) = \begin{pmatrix} \hat{\alpha}_0 & \sqrt{\hat{\beta}_1} & & \\ \sqrt{\hat{\beta}_1} & \hat{\alpha}_1 & \ddots & \\ & \ddots & \ddots & \sqrt{\hat{\beta}_{n-1}} \\ & & \sqrt{\hat{\beta}_{n-1}} & \hat{\alpha}_{n-1} \end{pmatrix},$$

e $w_v^G = \hat{\beta}_0 u_{v,1}^2$, dove $u_{v,1}$ è la prima componente dell'autovettore normalizzato u_v relativo all'autovalore t_v^G.

Il calcolo della matrice $\hat{J}_n$, e quindi dei coefficienti di ricorrenza $\hat{\alpha}_v$, $\hat{\beta}_v$, è un problema che però può essere mal condizionato e costoso dal punto di vista computazionale. Questo accadrebbe impiegando i momenti classici o i determinanti di Hankel, come mostrato nel capitolo precedente. Inoltre, usando la procedura di Stiltjes, ricavare i coefficienti di ricorrenza richiederebbe il calcolo di prodotti interni rispetto alla misura modificata $d\hat{\lambda} = d\lambda/\omega_m$, che difficilmente sarà una misura *standard* per cui siano disponibili formule ottimali per il calcolo

dei coefficienti, come invece esistono per misure più comuni (la routine Matlab `r_jacobi.m` ad esempio calcola i coefficienti di ricorrenza rispetto alla misura di Jacobi).

Per evitare di impiegare direttamente la misura $d\hat{\lambda}$ si può approssimare la stessa con una misura discreta che abbia come supporto un numero finito di punti. Sia una famiglia di misure discrete

$$d\lambda_N = \sum_{v=1}^{N} w_{v,N}\delta_{t_{v,N}}(t) \quad \text{tale per cui} \quad \lim_{N\to\infty}\int_{\mathbb{R}} p(t)d\lambda_N(t) = \int_{\mathbb{R}} p(t)d\hat{\lambda}(t) \quad \forall p \in \mathbb{P} \tag{2.6}$$

con

$$\delta_{t_{v,N}}(t) = \begin{cases} 1 & t = t_{v,N} \\ 0 & \text{altrimenti} \end{cases}, \qquad \int_{\mathbb{R}} p(t)d\lambda_N(t) = \sum_{v=1}^{N} w_{v,N}p(t_{v,N}),$$

e nodi $t_{v,N} \in \text{supp}(d\lambda)$. La proprietà (2.6) vale se, per esempio, $t_{v,N}$ e $w_{v,N}$ sono i nodi e pesi di una formula di quadratura interpolatoria rispetto alla misura $d\hat{\lambda}$. Il prodotto interno discreto associato alla misura $d\lambda_N$ gode della proprietà

$$\lim_{N\to\infty}(u;v)_N = (u;v) \quad \forall u, v \in \mathbb{P}. \tag{2.7}$$

Indichiamo con $\hat{\alpha}_{k,N}$ e $\hat{\beta}_{k,N}$ rispettivamente $\alpha_k(d\lambda_N)$ e $\beta_k(d\lambda_N)$. Se vale la proprietà di convergenza (2.7), allora si dimostra ([6], teorema 2.32) che

$$\lim_{N\to\infty}\hat{\alpha}_{k,N} = \hat{\alpha}_k, \quad \lim_{N\to\infty}\hat{\beta}_{k,N} = \hat{\beta}_k, \quad \forall k.$$

In virtù di questa proprietà $\hat{\alpha}_k$ e $\hat{\beta}_k$ possono essere approssimati da $\hat{\alpha}_{k,N}$ e $\hat{\beta}_{k,N}$ a qualsiasi precisione, per N sufficientemente elevato.

Individuata una misura $d\lambda_N$ con cui si intende approssimare $d\hat{\lambda}$, si procede con il calcolo dei coefficienti $\hat{\alpha}_{k,N}$ e $\hat{\beta}_{k,N}$ applicando uno dei seguenti metodi.

Il primo metodo è la procedura di *Stieltjes discretizzata*: questa consiste nell'applicare la procedura di Stieltjes utilizzando il prodotto interno discreto $(\cdot;\cdot)_N$ per calcolare $\hat{\alpha}_{k,N}$ e $\hat{\beta}_{k,N}$.

Se $t^G_{v,N}$, $w^G_{v,N}$ per $v = 1, \ldots, N$ sono i nodi e pesi della formula Gaussiana ad N punti rispetto alla misura $d\lambda$, scegliamo come prodotto interno discreto

$$(u;v)_N = \sum_{v=1}^{N} u(t^G_{v,N})v(t^G_{v,N})\frac{w^G_{v,N}}{\omega_m(t^G_{v,N})},$$

con $t_{v,N} = t^G_{v,N}$ e $w_{v,N} = \frac{w^G_{v,N}}{\omega_m(t^G_{v,N})}$.

Se u e v sono polinomi, tale prodotto equivale ad applicare la formula Gaussiana di misura $d\lambda$ alla funzione continua su $\text{supp}(d\lambda)$ $\frac{u(t)v(t)}{\omega_m(t)}$, pertanto, per il teorema (6), il prodotto interno discreto scelto rispetta la proprietà di convergenza (2.7). Stieltjes discretizzato è implementato nella routine Matlab `stieltjes.m`.

Essendo $d\lambda_N$ una misura discreta, in alternativa a Stieltjes discretizzato è possibile calcolare i coefficienti $\hat{\alpha}_{k,N}$ e $\hat{\beta}_{k,N}$ tramite l'*algoritmo di Lanczos*. La routine `lanczos.m` fornisce un'implementazione in Matlab dell'algoritmo.

Al fine di approssimare i coefficienti di ricorrenza con con una tolleranza ϵ si può utilizzare il seguente criterio: scelta una sequenza di valori interi $N_1 < N_2 < N_3 < \ldots$, si itera Stieltjes discretizzato o Lanczos su $N = N_1, N_2, N_3, \ldots$, finché

$$\max_{0\leq k\leq n-1} \left| \frac{\hat{\beta}_{k,N_{i+1}} - \hat{\beta}_{k,N_i}}{\hat{\beta}_{k,N_{i+1}}} \right| \leq \epsilon. \tag{2.8}$$

Accenniamo brevemente a una serie di metodi che ha come scopo il calcolo dei coefficienti di ricorrenza $\alpha_k(d\hat{\lambda})$ e $\beta_k(d\hat{\lambda})$ a partire dai coefficienti $\alpha_k(d\lambda)$ e $\beta_k(d\lambda)$, che si suppone siano noti.

Si premette che la misura modificata può essere $d\hat{\lambda} = (u/v)d\lambda$ con u e v polinomi reali non nulli in $\mathrm{supp}(d\lambda)$. Tali metodi possono essere scomposti in passaggi elementari: dal momento che u e v possono essere fattorizzati in termini del tipo $t - x$, oppure $(t - x)^2 + y^2$ con $x, y \in \mathbb{R}$, è possibile a ogni passaggio moltiplicare o dividere progressivamente $d\lambda$ per un fattore di u o rispettivamente di v alla volta. Dunque è sufficiente saper applicare tali metodi nei casi in cui $d\lambda$ sia moltiplicata o divisa per una funzione lineare o quadratica. Nel caso in cui $d\hat{\lambda} = ud\lambda$, dove $u = t-x$ oppure $(t-x)^2+y^2$ si possono ricavare i coefficienti $\hat{\alpha}_k$ e $\hat{\beta}_k$ tramite l'algoritmo 2.5 o rispettivamente 2.6 in [6]. Se invece $d\hat{\lambda} = d\lambda/v$, occorre usare l'algoritmo 2.8 nel caso v sia una funzione lineare, oppure, se è quadratica, l'algoritmo 2.9 in [6].

Vediamo alcune applicazioni di questi algoritmi al caso di nostro interesse, ovvero al calcolo dei coefficienti di ricorrenza per la misura modificata $d\hat{\lambda} = \omega_m d\lambda$. Supponiamo che esista la misura discreta

$$d\lambda_N = \sum_{v=1}^{N} w_{v,N}\delta_{t_{v,N}}(t)$$

con $w_{v,N}$ non necessariamente positivi e $N > n$, tale per cui la formula di quadratura rispetto alla misura $d\hat{\lambda}$

$$\sum_{v=1}^{N} f(t_{v,N})w_{v,N} \approx \int_{\mathbb{R}} f(t)d\hat{\lambda}(t) \tag{2.9}$$

abbia grado di precisione $2n - 1$. Allora rappresentando i coefficienti $\hat{\alpha_k}$ e $\hat{\beta_k}$ tramite la formula (1.3), si dimostra per induzione che $\hat{\alpha_k} = \alpha_k(d\lambda_N)$ e $\hat{\beta_k} = \beta_k(d\lambda_N)$ per $k = 0, 1, \ldots, n - 1$. Applicando Stieltjes discretizzato o l'algoritmo di Lanczos rispetto a $d\lambda_N$ si ricavano tali coefficienti. Rimane da ricavare la formula di quadratura (2.9).

Mostriamo come ricavare la formula (2.9) nel caso ω_m abbia radici reali semplici fuori da $\mathrm{supp}(d\lambda)$: sia $s_\mu = 1$ e $\zeta_\mu \in \mathbb{R}$ per $\mu = 1, \ldots, M$ (quindi

$M = m$). Applicando la scomposizione in frazioni parziali a ω_m si ottiene

$$\frac{1}{\omega_m(t)} = \frac{1}{\prod_{v=1}^{m}(1+\zeta_v t)} = \sum_{v=1}^{m} \frac{c_v}{t+1/\zeta_v}, \quad \text{dove} \quad c_v = \frac{\zeta_v^{m-2}}{\prod_{u=1,u\neq v}^{m}(\zeta_v - \zeta_u)}.$$

Allora

$$\int_{\mathbb{R}} f(t)\hat{\lambda}(t) = \sum_{v=1}^{m} \int_{\mathbb{R}} f(t) \frac{c_v d\lambda(t)}{t+1/\zeta_v}.$$

Il membro a destra dell'uguaglianza coinvolge integrali rispetto alla misura $d\lambda$ diviso una funzione lineare. Per ognuna di tali misure, si possono ricavare i primi n coefficienti di ricorrenza tramite l'opportuno algoritmo sopra citato. Ottenuti i coefficienti rispetto alla misura $d\lambda(t)/(t+1/\zeta_v)$, si ottiene la formula Gaussiana

$$\sum_{r=1}^{n} w_r^{(v)} f(t_r^{(v)}) \approx \int_{\mathbb{R}} f(t) \frac{c_v d\lambda(t)}{t+1/\zeta_v}.$$

Allora la formula di quadratura

$$\sum_{v=1}^{m}\sum_{r=1}^{n} w_r^{(v)} f(t_r^{(v)}) \approx \sum_{v=1}^{m} \int_{\mathbb{R}} f(t) \frac{c_v d\lambda(t)}{t+1/\zeta_v} = \int_{\mathbb{R}} f(t) \frac{d\lambda(t)}{\omega_m(t)}$$

è esatta per $f \in \mathbb{P}_{2n-1}$, dunque per

$$\begin{aligned} t_{(v-1)n+r} &= t_r^{(v)}, \\ w_{(v-1)n+r} &= w_r^{(v)}, \end{aligned} \qquad r = 1, 2, \ldots, n, \quad v = 1, 2, \ldots, m$$

e $N = nm$ si ottiene la formula (2.9).

Supponiamo che ω_m abbia radici complesse coniugate semplici: sia $\zeta_\mu = \xi_\mu + i\eta_\mu$, $\zeta_{\mu+m/2} = \bar{\zeta}_\mu$, per $\mu = 1, \ldots, m/2$, dove $\xi_\mu \in \mathbb{R}$, $\eta_\mu > 0$ e m è pari. È possibile scomporre nuovamente $1/\omega_m$ in

$$\frac{1}{\omega_m(t)} = \sum_{v=1}^{m/2} \frac{c_v + d_v t}{\left(t + \frac{\xi_v}{\xi_v^2+\eta_v^2}\right)^2 + \left(\frac{\eta_v}{\xi_v^2+\eta_v^2}\right)^2},$$

con

$$\begin{aligned} c_v &= \frac{1}{\eta_v}\left(\frac{\xi_v}{\xi_v^2+\eta_v^2} Im(p_v) + \frac{\eta_v}{\xi_v^2+\eta_v^2} Re(p_v)\right), \\ d_v &= \frac{1}{\eta_v} Im(p_v), \\ p_v &= \prod_{u=1,u\neq v}^{m/2} \frac{(\xi_v + i\eta_v)^2}{(\xi_v-\xi_u)^2 - (\eta_v^2 - \eta_u^2) + 2i\eta_v(\xi_v - \xi_u)}. \end{aligned}$$

Da notare che i termini in cui è stato scomposto $1/\omega_m$ presentano un termine quadratico al denominatore e un termine lineare al numeratore, se $d_v \neq 0$. Come

nel caso precedente, è possibile costruire la formula (2.9) impiegati però questa volta gli algoritmi 2.5 e 2.9 in [6].

In [3] vengono riportate le scomposizioni in frazioni parziali per ω_m polinomio con radici di molteplicità 2, sia reali, che complesse coniugate, con l'eventuale aggiunta di un polo semplice reale, alle quali seguono l'applicazione degli algoritmi citati sopra.

2.3 La routine `integral` di Matlab

La routine `integral` di Matlab è una funzione adattiva globale per calcolare numericamente integrali. I metodi di integrazione adattivi non lavorano uniformemente sul dominio di integrazione, ma riconoscono i punti in cui l'integranda presenta delle *particolarità*, come singolarità, punti di discontinuità o variazioni molto forti, e concentrano su tali zone il proprio lavoro al fine di migliorare l'approssimazione dell'integrale. Vediamo nel dettaglio come avviene la chiamata di questa funzione in Matlab:

```
q = integral(f,tmin,tmax)
```

riceve in input l'integranda `f` di tipo function handle, e i valori scalari `tmin` e `tmax`, che possono essere sia reali (eventualmente `Inf` o `-Inf`) sia complessi (se sono entrambi finiti). La funzione approssima l'integrale di `f` sul cammino rettilineo nel piano complesso che va da `tmin` a `tmax`. Se `f` rappresenta una funzione $f(t)$ che assume valori scalari, con t variabile scalare, allora `y=f(t)` deve poter accettare come argomento un vettore `t`, e ritornare il vettore `y` della stessa dimensione contenente le valutazioni di f sugli elementi di `t`; dunque se ad esempio serve integrare $f(t) = \frac{1}{t}$ sull'intervallo $[1, 2]$ la funzione in input sarà `f=@(t)1./t` che può valutare un vettore `t` e tornare il vettore `y=f(t)` della stessa dimensione.

```
q = integral(f,tmin,tmax,Nom1,Val1,Nom2,Val2,...)
```

permette di specificare una o più coppie di parametri opzionali `Nomi,Vali`, dove `Nomi` indica il nome del parametro aggiuntivo i-esimo e `Vali` il rispettivo valore.

Ad esempio, se indichiamo come parametri opzionali inseriamo `Nom1=AbsTol`, e `Val1=1e-11`, allora `integral` approssimerà l'integrale con un errore assoluto (stimato) di 10^{-11}.

Con `Nom1=RelTol` è possibile specificare l'errore relativo `Val1`. I valori di default per `AbsTol` e `RelTol` sono rispettivamente `1e-10` e `1e-6`. Qualunque siano questi due parametri, `integral` cercherà di soddisfare

```
abs(q - Q) <= max(AbsTol,RelTol*abs(q)),
```

dove `Q` è il valore esatto dell'integrale da calcolare.

Un parametro opzionale molto importante è `Waypoints` con valore un vettore `v` di punti reali o complessi. Se `tmax` e `tmin` appartengono alla retta reale in tale vettore si possono specificare le discontinuità dell'integranda, oppure i punti

nell'intervallo di integrazione in prossimità dei quali l'integranda varia molto velocemente, in modo da aumentare il numero di valutazioni di `f` in tali zone, al fine di incrementare la precisione con cui si calcola l'integrale. Se invece `tmax`, `tmin` o qualsiasi punto del vettore `v` è complesso, e gli estremi di integrazione sono finiti, verrà calcolato l'integrale lungo il cammino rettilineo a tratti che va da `tmin` a `v(1)`, da `v(1)` a `v(2)` e così via, fino al tratto che va da `v(end)` a `tmax`. Non conviene indicare con `Waypoints` eventuali singolarità dell'integranda. È invece preferibile dividere l'intervallo di integrazione in modo che queste siano agli estremi, e sommare gli integrali risultanti.

La routine `integral` è una versione più recente e facile da usare di un'altra funzione per il calcolo di integrali, `quadgk`. Quest'ultima a sua volta è un'implementazione per Matlab dell'algoritmo `quadva` [12]. Altre funzioni impiegate nell'integrazione numerica, ma più datate, sono `quad` e `quadl`.

Analizziamo più nel dettaglio `quadva` enfatizzando in particolare due caratteristiche che possiede in comune con `integral`: il fatto che sia una funzione di integrazione adattiva, e che in essa venga implementata la vettorizzazione, aspetto che aumenta notevolmente l'efficienza della routine rispetto alle versioni precedenti `quadl` e `quad`.

La chiamata di `quadva` avviene nella seguente maniera:

```
[q,err]=quadva(f,interval,reltol,abstol)
```

La chiamata restituisce un'approssimazione `q` dell'integrale della funzione `f` nell'intervallo $[a, b]$, con a=`interval(1)` e b=`interval(end)`, e una stima dell'errore commesso `err`.

Come avviene per `integral`, la funzione `f`, sebbene rappresenti una funzione con argomento scalare t a valori scalari $f(t)$, può ricevere per argomento un vettore `t` e restituire il vettore `y=f(t)` delle valutazioni di f sulle componenti di `t`.

Le tolleranze per l'errore relativo e assoluto con cui approssimare l'integrale sono indicate nel codice da `rtol` e `atol`, e possono essere modificate dall'utente specificando i parametri `reltol` e `abstol`.

Mostriamo il funzionamento dell'algoritmo. Quando `quadva` viene chiamata, ad un dato momento della sua esecuzione produce un'approssimazione `qOK` dell'integrale di f su una porzione dell'intervallo $[a, b]$. Questa approssimazione viene considerata sufficientemente accurata dalla routine rispetto alle tolleranze per gli errori `reltol` e `abstol`. La restante porzione di $[a, b]$ è immagazzinata nella matrice `subs` come un insieme di sottointervalli disgiunti: ogni colonna di `subs` ha due entrate, una contenente l'estremo sinistro, l'altra l'estremo destro di ogni sottointervallo. Successivamente ad ogni sottointervallo viene applicata la formula di quadratura di Gauss-Kronrod. Il numero di punti impiegato da tale formula di quadratura in `quadva` è indicato da `samples` e vale 15. Per ogni sottointervallo sono calcolati i nodi della formula di Gauss-Kronrod, che vengono poi inseriti in un unico vettore riga valutato dall'integranda (ricordiamo che l'integranda deve accettare vettori come argomento) in un'unica chiamata, che restituisce il vettore riga `ft`. Le coordinate di `ft` vengono riposizionate in una

matrice mediante la built-in function `ft=reshape(ft,samples,[])`: ogni colonna di tale matrice contiene tutte le valutazioni dell'integranda di un determinato sottointervallo. I pesi della formula di quadratura sono immagazzinati nella matrice diagonale `w` mentre il vettore riga `halfh` contiene la metà dell'ampiezza di ogni sottointervallo. La riga di codice

```
qsubs= sum(w*ft).*halfh
```

applica la formula di Gauss-Kronrod contemporaneamente in ogni sottointervallo, utilizzando la moltiplicazione per vettori e la built-in function `sum`. Allo stesso tempo viene costruito il vettore `errsubs`, il cui elemento `errsubs(i)` è la stima dell'errore commesso da `qsubs(i)` nell'approssimare l'integrale nel sottointervallo i-esimo. A questo punto otteniamo l'approssimazione dell'integrale su tutto l'intervallo $[a, b]$ con

```
q=qOK+sum(qsubs),
```

sommando a `qOK` (approssimazione accurata) le approssimazioni appena calcolate `qsubs` sui sottointervalli di `subs`, e viene prodotta la stima dell'errore `err` commesso da `q` sull'intero intervallo. Se la stima dell'errore è abbastanza piccola, la funzione termina la sua esecuzione, altrimenti cerca i sottointervalli in cui l'approssimazione non è stata abbastanza accurata. Se si desidera calcolare l'integrale su $[a, b]$ con un errore massimo pari a τ, è sufficiente approssimare l'integrale su ogni sottointervallo $[\alpha, \beta]$ con un errore minore di $\tau\frac{\beta-\alpha}{b-a}$. Definita la tolleranza

```
tol=max(atol,rtol*abs(q))
```

si cercano i sottointervalli in cui l'approssimazione non è stata adeguata. Utilizzando la build-in function `find` nella riga di codice

```
ndx=find(abs(errsubs)<=(2/(b-a))*halfh*tol
```

si trovano gli indici di `qsubs` per cui l'approssimazione è accettabile, ossia l'errore è minore della tolleranza. I sottointervalli corrispondenti a tali indici saranno aggiunti alla porzione di $[a, b]$ per cui l'integrale è considerato sufficientemente accurato e saranno rimossi dal vettore degli intervalli `subs` su cui la routine è attiva, infine viene aggiornato il valore `qOK`:

```
qOK = qOK + sum(qsubs(ndx)), subs(:,ndx) = [].
```

La funzione procede al passo successivo, ripetendo la procedura sui sottointervalli dimezzati rimasti in `subs`.

È evidente che con questo procedimento la routine non lavora uniformemente su tutto l'intervello di integrazione, ma si concentra nei sottointervalli in cui l'approssimazione dell'integrale non risulta accurata rispetto al criterio scelto. In questo senso la routine è un esempio di funzione adattiva.

Invece la vettorizzazione del codice in `quadva` si può osservare dal fatto che nel procedimento esposto i punti in cui viene valutata l'integranda sono inseriti nello stesso vettore, in modo da poter valutare l'integranda con un'unica

chiamata di `f`, ottenendo `ft`. In seguito vengono calcolate le approssimazioni utilizzando built-in function e operazioni su vettori, riducendo al minimo le chiamate dell'integranda e di altre funzioni. Questo accorgimento viene usato perché il calcolo vettoriale risulta molto efficiente in Matlab, è infatti indicativo del tempo di esecuzione della routine, il numero di valutazioni vettoriali dell'integranda e non il numero di valutazioni punto per punto.

La routine `quadva` permette di risolvere problemi di integrazione per funzioni con singolarità isolate. Un modo generale di procedere è dividere l'intervallo $[a, b]$ in sottointervalli in modo che le singolarità siano presenti agli estremi. Successivamente vengono applicati particolari cambi di variabili all'integrale di un sottointervallo, (vedi [12], sezione 4.2) in modo da indebolire o eliminare le singolarità, o ricondurre un integrale su un intervallo infinito a uno equivalente su un intervallo finito. In tali casi la formula di quadratura di Gauss-Kronrod, utilizzata da `quadva` in ogni sottointervallo, ha il vantaggio di essere una formula aperta, ossia non valuta l'integrada agli estremi dell'intervallo, dove sono presenti le singolarità. Se l'integranda presenta dei punti di discontinuità, dei picchi molto acuti, o delle zone in cui varia molto rapidamente, conviene dividere $[a, b]$ in sottointervalli in cui l'integranda sia regolare; `interval` consente all'utente di specificare i punti in cui sono presenti tali anomalie, punti che verranno presi come estremi dei sottointervalli di `subs`.

2.4 La routine `rfejer` in Matlab

La routine `rfejer` implementa una particolare tipologia della formula di quadratura di Fejér razionale basata sulle funzioni ortogonali razionali (ORF) di Chebyshev per approssimare integrali del tipo

$$\int_{-1}^{1} f(t)dt.$$

Mentre le formule di Fejér I e II sono performanti per integrande che hanno andamenti polinomiali, questa si adatta meglio per funzioni che presentano un certo insieme di singolarità al di fuori dell'intervallo di integrazione $[-1, 1]$.

Tale formula di Fejér razionale è descritta dettagliatamente nell'articolo [1], mentre in questa sezione verranno esposte le caratteristiche fondamentali della formula e la struttura della routine.

Consideriamo l'intervallo $I = [a, b]$ e l'insieme di poli $A = \{\alpha_1, \alpha_2, \dots\} \subset \overline{\mathbb{C}} \backslash I$. Definiamo i fattori

$$Z_k(t) := Z_{\alpha_k}(t) = \frac{t}{1 - t/\alpha_k} \quad k = 1, 2, \dots$$

e le funzioni

$$b_0(t) = 1, \quad b_k(t) = b_{k-1}(t) Z_k(t), \quad k = 1, 2, \dots.$$

Consideriamo gli spazi di funzioni razionali con poli in A generati da queste funzioni:

$$\mathscr{L}_{-1} = 0, \quad \mathscr{L}_k = \text{span}\{b_0, b_1, \ldots, b_k\}.$$

Utilizzando la funzione ortogonale razionale (ORF) φ_n è possibile costruire ([1], cap. 2, 3) la formula di quadratura

$$\sum_{v=1}^{n} f(t_v) w_v \approx \int_{\mathbb{R}} f(t) d\lambda(t), \tag{2.10}$$

con $d\lambda$ misura non necessariamente positiva e $\text{supp}(d\lambda) \subset I$, esatta per funzioni appartenenti allo spazio $\mathscr{L}_{n-1}$.

φ_n è definita nel modo seguente. Si consideri una misura $d\mu$ positiva e limitata sui boreliani dell'intervallo I. Definiamo il prodotto interno

$$(f, g)_{d\mu} = \int_{\mathbb{R}} f(t)\overline{g(t)} d\mu(t) \qquad f, g \in L^2(d\mu).$$

Si ricava dalla base $\{b_0, b_1, \ldots, b_n\}$ la base ortonormale $\{\varphi_0, \varphi_1, \ldots, \varphi_n\}$, per cui $\varphi_k \in \mathscr{L}_k \backslash \mathscr{L}_{k-1}$, $\varphi_k \perp_\mu \mathscr{L}_{k-1}$, e $(\varphi_k, \varphi_j)_\mu = \delta_{k,j}$, per $0 \le k, j \le n$.

Se $I = [-1, 1]$ gli zeri di φ_n sono distinti e appartenenti all'intervallo $(-1, 1)$ se e solo se $\alpha_n \in \overline{\mathbb{R}} \backslash I$. I nodi t_v della formula (2.10) sono gli n zeri di φ_n, mentre i pesi w_v si ricavano da questi stessi punti tramite la procedura presente in [1], cap. 2.

Da notare che se $\alpha_k = \infty$ per $k \ge 0$, allora $Z_k(t) = t$, e $b_k(t) = t^k$, pertanto $\mathscr{L}_{n-1} = \mathbb{P}_{n-1}$ e la formula di quadratura risulta interpolatoria (polinomiale).

La formula di Fejèr razionale implementata dalla routine `rfejer` si ricava utilizzando $d\lambda(t) = dt$ la misura di Legendre con supporto l'intervallo $I = [-1, 1]$ e $d\mu(t) = (1 - t^2)^{-1/2}$ o $d\mu(t) = (1 - t^2)^{1/2}$, rispettivamente la misura di Chebyshev del tipo I e del tipo II.

L'insieme dei poli delle funzioni razionali in $\mathscr{L}_{n-1}$ è $A_n = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. I poli sono ordinati in modo tale che se $\alpha_k \in \mathbb{C} \backslash \mathbb{R}$, allora α_{k-1} o α_{k+1} è uguale a $\overline{\alpha}_k$, inoltre α_k e $\overline{\alpha}_k$ devono avere la stessa molteplicità. Si suppone inoltre che $\alpha_n = \infty$ in modo che φ_n abbia n zeri distinti in I.

Vediamo nel dettaglio il funzionamento della routine.

Dato il vettore dei poli `sgl` di lunghezza N, la chiamata

```
x=rfejer(sgl)
```

ritorna i nodi `x(1,:)` e i pesi `x(2,:)` della formula di Fejér razionale a $N+M+1$ punti. M è il numero di poli per i quali non è presente il proprio complesso coniugato in `sgl`. Tra gli output, il vettore `x(3,1:end-1)` contiene la sequenza dei poli al completo rispetto alla quale è calcolata la formula, ossia con l'aggiunta di eventuali complessi coniugati mancanti. La subfunction `pcheck` aggiunge i poli mancanti e li ordina in modo che ogni polo sia seguito o preceduto dal proprio complesso coniugato. La funzione `rcheb` calcola i nodi della formula rispetto a tali poli (gli zeri di ORF rispetto alla misura di Chebyshev).

La chiamata

```
x=rfejer(sgl,f)
```

calcola iterativamente su `sgl` l'integrale sull'intervallo $[-1, 1]$ della funzione `f`.

A ogni iterazione viene approssimato l'integrale richiesto con la formula di Fejèr razionale rispetto solamente ad alcuni dei poli presenti in `sgl`, mentre all'iterazione successiva viene approssimato l'integrale, incrementando il numero dei poli rispetto ai quali viene calcolata la formula. Le iterazioni continuano finché l'approssimazione dell'integrale non converge all'integrale richiesto o la formula di quadratura comincia a deteriorarsi a causa del condizionamento dell'algoritmo, altrimenti le iterazioni terminano quando sono stati impiegati tutti i poli di `sgl`.

La chiamata della routine `rfejer`

```
[x,err,rel,y]=rfejer(sgl)
```

restituisce inoltre per ogni elemento in `x` l'accuratezza stimata `err` e il motivo `rel` per il quale si sono fermate le iterazioni. In particolare

- `rel=1` indica che le iterazioni si sono fermate per convergenza numerica,
- `rel=2` indica che è stato raggiunto il massimo numero di iterazioni,
- `rel=3` indica che le iterazioni si sono fermate a causa del deterioramento nel calcolo della formula di Fejér.

Inoltre `y(1,:)` sono i nodi, `y(2,:)` i pesi e `y(3,1:end-1)` la sequenza dei poli impiegati nell'ultima iterazione.

La convergenza delle approssimazioni tra iterazioni successive è verificata dalla subfunction `cvg` e da un parametro p_k che assume valore iniziale $p_1 = 0$. Alla k-esima iterazione il parametro assume valore 1 se la differenza relativa dell'approssimazione corrente con quella dell'iterazione precedente è minore della quantità $\sqrt{\texttt{itol}}$, altrimenti vale 0.

La coppia (p_k, p_{k-1}) é indicativa del fatto che la formula sta convergendo o divergendo all'integrale richiesto, per cui a seconda del valore assunto da p_k e p_{k-1} (vedi [1] pg 17) il numero massimo di iterazioni verrà modificato in modo da migliorare l'approssimazione o evitare che quest'ultima peggiori.

A ogni iterazione viene verificato che la somma dei pesi calcolati sia sempre uguale a 2. Inoltre la funzione `errW` controlla l'accuratezza di tutti i pesi calcolando

$$\int_{-1}^{1} f_k(t)dt \qquad f_k(t) = \begin{cases} (t-\alpha_k)^{-m_k} & k > 0 \text{ e } \alpha_k \neq \infty \\ t^{m_k} & k = 0 \text{ o } \alpha_k = \infty \end{cases},$$

con m_k la molteplicità di α_k; per tali integrali infatti la formula di Fejér deve essere esatta.

Aggiungiamo alcune considerazioni riguardo i poli impiegati dalla routine. Data una funzione f con singolarità $\{\gamma_j\}_{j>0}$, dal punto di vista teorico ([1] cap. 6), per ottenere la miglior approssimazione di $\int_{-1}^{1} f(t)dt$ conviene scegliere

come poli le singolarità di f. Tuttavia dal punto di vista numerico, questa non si rivela essere sempre la scelta ottimale.

A seconda di dove si trovano le singolarità di f, scegliere tali punti come poli può comportare un moderato miglioramento della convergenza all'integrale, a discapito di un costo computazionale molto più alto; questo caso si presenterebbe se le singolarità si trovassero molto distanti dall'intervallo di integrazione.

Inoltre le singolarità potrebbero non essere note e troppo costose computazionalmente da determinare.

Infine l'accuratezza dei nodi e pesi calcolati può deteriorarsi velocemente al crescere del numero di iterazioni per particolari scelte dei poli: questo succederebbe se alcune singolarità fossero molto vicine tra loro o troppo vicine a I.

Nel caso in cui i poli abbiano un valore assoluto maggiore di `pinf=5/eps`, ovvero sono molto distanti dall'intervallo di integrazione, sono sostituiti da poli infiniti.

Nel caso di singolarità molto vicine tra loro è conveniente sostituirle con un unico polo che abbia come molteplicità il numero di singolarità vicine. Nella routine i poli che presentano una distanza relativa l'uno dall'altro minore di `ptol=5*eps` sono rimpiazzati da un polo multiplo.

Se le singolarità sono molto vicine a I la formula di quadratura dovrebbe essere combinata con metodi adattivi, che tramite dei cambi di variabile allontanano le singolarità dall'intervallo di integrazione (vedi Remark 6.2, [1]).

Per calcolare $\int_a^b f(t)dt$ con $[a,b]$ intervallo generico (non necessariamente limitato), è possibile applicare le trasformazioni τ presenti in Table 1 in [1] e ottenere l'integrale equivalente $\int_{-1}^1 h(t)dt = \int_a^b f(t)dt$ con $h = f \circ \tau \cdot \tau'$. La funzione Matlab `transf` implementa queste trasformazioni.

Riguardo alla complessità dell'algoritmo forniamo il seguente teorema:

Teorema 10 *Supponiamo che l'integranda possa essere valutata a un costo computazionale di o(n), con n il numero di nodi e pesi della formula di Fejér razionale, e che k sia il numero di iterazioni dopo le quali la routine converge numericamente o comincia a deteriorarsi. Allora la routine ha un costo computazionale dell'ordine* $\mathcal{O}(k^2 n) + \mathcal{O}(n^2)$.

2.5 Formula di Gauss-Legendre

In questa sezione parleremo della formula di Gauss-Legendre, delle sue caratteristiche e di un particolare metodo per calcolarla. La formula di Gauss-Legendre è la regola di quadratura gaussiana

$$\sum_{v=1}^{n} f(t_v) w_v \approx \int_{-1}^{1} f(t)dt$$

associata alla funzione peso $w(t) = 1$ con supporto l'intervallo $[-1, 1]$. Dal teorema (5) si evince che i nodi t_v sono gli zeri del polinomio ortogonale $\pi_n(\cdot, dt)$, mentre i pesi sono $w_v = \int_{-1}^1 L_v(t)dt = \int_{-1}^1 L_v(t)^2 dt$.

Nella maggior parte delle applicazioni, ad esempio nell'approssimare integrali di funzioni analitiche o C^∞, le formule gaussiane richiedono un numero esiguo di nodi. Allo stesso modo metodi adattivi per l'integrazione che suddividono ricorsivamente il dominio di integrazione, come la routine `quadva`, utilizzano spesso formule gaussiane di basso grado. Tuttavia per alcune applicazioni, come l'integrazione di funzioni razionali, o funzioni con poli molto vicini al dominio di integrazione, sono necessarie formule gaussiane di grado alto.

L'algoritmo di cui parleremo permette di ottenere la formula di Gauss-Legendre a un costo computazionale relativamente basso di $\mathcal{O}(n)$ operazioni, calcolando i nodi e i pesi alla precisione di macchina per qualsiasi valore di $n \geq 100$.

Riportiamo la tabella (3.16) che mostra gli errori e il tempo di esecuzione con cui è stata calcolata la formula di Gauss-Legendre al variare del numero di nodi. Altri metodi per calcolare la formula di Gauss-Legendre soffrono di un

n	$\epsilon_{\text{abs}}\{t_k\}$	$\epsilon_{\text{rm}}\{w_k\}$	$\epsilon_{\text{mr}}\{w_k\}$	$\epsilon_{\text{quad}}\{t_k, w_k\}$	cputime (sec)
100	$1.18e-16$	$1.15e-16$	$1.25e-15$	$1.71e-16$	0.0085
1000	$1.63e-16$	$8.27e-16$	$1.92e-15$	$1.11e-16$	0.0105
10000	$1.78e-16$	$1.14e-15$	$1.69e-15$	$1.11e-16$	0.0261
100000	$2.22e-16$	$1.09e-15$	$1.48e-15$	$4.44e-16$	0.2600
1000000	$3.33e-16$	$2.70e-15$	$3.02e-15$	$6.66e-16$	2.3198

Tabella 2.1: Questa tabella riporta gli errori (assoluto, relativo massimo, massimo relativo e di quadratura) e la cputime di esecuzione, relativamente alla formula di Gauss-Legendre utilizzando l'algoritmo descritto in questa sezione con $n = 10^2, \ldots, 10^6$ punti.

costo computazionale molto alto, oppure di un errore che cresce rapidamente in funzione di n, al punto da rendere impossibile l'impiego della formula quando n è molto alto. La tabella (2.2) mette a confronto gli andamenti degli errori e i costi computazionali di alcuni metodi comunemente usati per il calcolo della formula di Gauss-Legendre.

Il seguente algoritmo si basa sulla ricerca degli zeri dei polinomi di Legendre tramite il metodo di Newton, e utilizza particolari espansioni asintotiche per valutare i polinomi di Legendre e le sue derivate.

Indichiamo con P_n il polinomio di Legendre di grado n. $P_n(\cdot)$ e $\pi_n(\cdot, dt)$ sono uguali a meno di una costante moltiplicativa, pertanto hanno gli stessi zeri, ovvero i nodi t_k della formula di Gauss-Legendre.

I pesi vengono calcolati usando la formula

$$w_k = \frac{2}{(1-t_k^2)[P_n'(t_k)]^2}. \tag{2.11}$$

Dal momento che gli zeri dei polinomi di Legendre si concentrano prevalentemente in prossimità degli estremi dell'intervallo ± 1, in tali zone, per n particolarmente elevato, i punti t_k possono diventare molto densi e difficilmente

distinguibili gli uni dagli altri. Al fine di evitare questo inconveniente i nodi verranno espressi con $t_k = \cos\theta_k$, con $\theta_k \in [0, \pi]$. Inoltre, sostituendo $\cos\theta_k$ al posto di t_k si può riscrivere la formula dei pesi nel seguente modo:

$$w_k = \frac{2}{[\frac{d}{d\theta}P_n(\cos\theta_k)]^2}. \tag{2.12}$$

Questo accorgimento permette di prevenire la cancellazione numerica dovuta al termine $(1 - t_k^2)$ quando $t_k \approx \pm 1$.

Dal momento che gli zeri dei polinomi di Legendre sono semplici, il metodo di Newton ha convergenza quadratica se i punti iniziali sono sufficientemente vicini agli zeri. I punti di Chebyshev garantirebbero la convergenza del metodo di Newton, tuttavia è possibile scegliere delle approssimazioni iniziali migliori in modo da diminuire le iterazioni.

Poiché i polinomi di Legendre soddisfano $P_n(-t) = (-1)^n P_n(t)$, gli zeri t_k, e di conseguenza anche i pesi w_k, sono simmetrici rispetto a $t = 0$. Allora è sufficiente calcolare i nodi sull'intervallo $t \in [0, 1)$, o equivalentemente su $\theta \in (0, \pi/2]$. Adottiamo la notazione $\bar{k} = n - k + 1$ in modo che $t_{\bar{k}}$ sia il k-esimo nodo più vicino a 1.

Utilizziamo la formula

$$t_{\bar{k}} = \left\{1 - \frac{(n-1)}{8n^3} - \frac{1}{384n^4}\left(39 - \frac{28}{\sin^2\phi_k}\right)\right\}\cos\phi_k + \mathscr{O}(n^{-5}) \tag{2.13}$$

per approssimare i nodi distanti da 1, con $\phi_k = (k - 1/4)\pi/(n + 1/2)$. Per nodi vicini a 1 utilizziamo invece la formula

$$t_{\bar{k}} = \cos\left(\psi_k + \frac{\psi_k \cot\psi_k - 1}{8\psi_k(n + 1/2)^2}\right) + j_k^2\mathscr{O}(n^{-5}) \tag{2.14}$$

con $\psi_k = j_k/(n+1/2)$, dove j_k è la k-esima radice di $J_0(t)$, la funzione di Bessel del primo tipo.

Dunque a partire dal punto iniziale $\theta_k^{[0]} = \arccos(t_k^{[0]})$, dove

$$t_k^{[0]} = \{\ \text{(2.13) per } t \le 1/2 \text{ (i.e. } \theta \ge \pi/3)\text{(2.14) per } t > 1/2 \text{ (i.e. } \theta < \pi/3),$$

si applica il metodo di Newton

$$\theta_k^{[j+1]} = \theta_k^{[j]} - P_n(\cos\theta_k^{[j]})\left(-\sin\theta_k^{[j]}P_n'(\cos\theta_k^{[j]})\right)^{-1} \qquad j \ge 0.$$

P_n' soddisfa l'equazione $(1-t^2)P_n'(t) = -ntP_n(t) + nP_{n-1}(t)$ o equivalentemente rispetto alla variabile θ

$$-\sin\theta\frac{d}{d\theta}P_n(\cos\theta) = -n\cos\theta P_n(\cos\theta) + nP_{n-1}(\cos\theta).$$

Allora per eseguire ogni iterazione del metodo di Newton è sufficiente calcolare le valutazioni di P_n e P_{n-1}. Per conoscere tali valutazioni non è necessario

ricavare i polinomi di Legendre, ma è possibile impiegare delle loro approssimazioni mediante le formule asintotiche. Viene utilizzata la formula asintotica di frontiera (formula (3.12), [9]) per valutare P_n nei punti in prossimità di ± 1, invece per valutare P_n nei punti interni all'intervallo viene utilizzata la formula asintotica interna (formula (3.9), [9]).

È necessario definire con precisione le regioni interna ed esterna sulle quali verranno impiegate le due formule. La formula asintotica di frontiera richiede uno sforzo computazionale molto maggiore rispetto a quella interna per essere calcolata, essendo costituita da funzioni non elementari come la funzione di Bessel del primo tipo. Allora si cercherà di usare la formula asintotica interna per ricavare quanti più nodi possibili.

Sperimentalmente, impiegando la formula di espansione interna nel metodo di Newton per $n = 100, 500, 1000, 5000, 10000$, si riesce a ricavare tutti i nodi con un'accuratezza dell'ordine della precisione di macchina a eccezione dei primi 6 più vicini a 1 (figura 3.5 [9]): in base a tali risultati viene definita regione di frontiera, quella costituita dai primi e ultimi 10 nodi.

Una volta calcolati i nodi si possono calcolare i rispettivi pesi w_k dalle formule (2.11) o (2.12).

Citiamo di seguito altri metodi usati per il calcolo della formula di Gauss-Legendre riportando i rispettivi andamenti asintotici degli errori e dei tempi di esecuzione:

- L'algoritmo di Golub-Welsch (GW), di cui abbiamo già parlato nel precedente capitolo, ricava i nodi dagli autovalori della matrice di Jacobi, mentre i pesi sono ottenuti dai rispettivi autovettori. Tale algoritmo compie un numero di operazioni pari a $\mathscr{O}(n^2)$ grazie alla struttura vantaggiosa della matrice, e dal fatto che è necessario calcolare solo la prima componente degli autovettori. L'errore relativo cresce come $\mathscr{O}(n)$ nei nodi e $\mathscr{O}(n^2)$ nei pesi, tuttavia sperimentalmente (figura 2.1 [9]) gli errori sembrano avere un andamento più simile a $\mathscr{O}(\sqrt{n})$ e $\mathscr{O}(n^{3/2})$.

- L'algoritmo REC utilizza la relazione di ricorrenza a tre termini e le iterazioni del metodo di Newton per convergere agli zeri dei polinomi ortogonali. Tale algoritmo ha una complessità di $\mathscr{O}(n^2)$, e l'errore relativo sui pesi cresce come $\mathscr{O}(n)$, pertanto viene adoperato per valori bassi di n. Osserviamo invece che i nodi possono essere calcolati alla precisione di macchina quasi indipendentemente da n.

- L'algoritmo di Glasier-Liu-Rohklin (GLR) adopera il metodo di Newton e l'espansione di Taylor dei polinomi di Legendre e delle sue derivate per calcolare la formula di Legendre. Questo algoritmo è veloce ma il massimo errore relativo nei nodi cresce come $\mathscr{O}(n)$.

- Il metodo descritto in questa sezione (HT) si rivela essere molto preciso come si può notare dalla tabella (3.16), infatti gli errori sia sui nodi che sui pesi sono nell'ordine della precisione di macchina e crescono molto lentamente rispetto al numero di nodi n, con un costo computazionale contenuto di $\mathscr{O}(n)$.

Una routine con cui è possibile calcolare la formula di Gauss-Legendre è la funzione Chebfun `legpts`. Il comando

```
[t,w]=legpts(n,METODO)
```

calcola i nodi `t` e pesi `w` della formula di Gauss-Legendre a n punti, permettendo all'utente di scegliere quale metodo utilizzare.

La routine utilizzava di default l'algoritmo REC per $n < 129$, mentre per $n \geq 129$ l'algoritmo GLR. Successivamente GLR è stato sostituito dall'algoritmo HT. Recentemente l'algoritmo HT è stato sostituito dall'algoritmo di Bogaert per $n \geq 100$, mentre per $n < 100$ la routine impiega l'algoritmo REC.

Il parametro opzionale `METODO` specifica il metodo:

- se `METODO='REC'` il calcolo della formula avviene tramite l'algoritmo REC,
- se `METODO='ASY'` l'algoritmo di Bogaert,
- se `METODO='GW'` viene impiegato l'algoritmo di Golub-Welsch.

Algoritmo	$\epsilon_{\text{abs}}\{t_k\}$	$\epsilon_{\text{rm}}\{w_k\}$	$\epsilon_{\text{mr}}\{w_k\}$	$\epsilon_{\text{quad}}\{t_k, w_k\}$	cputime
GW	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(n^{3/2})$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
REC	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(n^{1.7})$
GLR	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{0.66})$	$\mathcal{O}(n)$
HT	$\mathcal{O}(\log(\log(n)))$	$\mathcal{O}(\log(\log(n)))$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Tabella 2.2: Questa tabella mostra l'andamento degli errori e della cputime di esecuzione delle varie formule impiegate per calcolare la formula di Gauss-Legendre in funzione del numero di nodi n.

2.6 Integrazione tramite `Chebfun`

In questa sezione verranno esposte alcune routine implementate in ambiente Chebfun adibite all'integrazione numerica.

Chebfun è stato costruito sull'idea che è possibile approssimare una funzione regolare alla precisione di macchina con il polinomio interpolante (anche a tratti) su un certo numero ottimale di nodi di Chebyshev (in ogni tratto).

Alcune formule di quadratura per integrare tali funzioni sono quella di Fejér di tipo I e II e la formula di Clenshaw-Curtis.

A tal proposito sia $d\lambda(t) = w(t)dt$ una misura definita nell'intervallo $[-1, 1]$, con $w : [-1, 1] \mapsto \mathbb{R}$ funzione peso. La formula di quadratura di *Fejér di tipo I* rispetto alla misura $d\lambda(t) = w(t)dt$ ha come nodi i punti di Chebyshev

$$t_k^{F1} = \cos\left(\frac{(2k-1)\pi}{2n}\right), \quad k = 1, \ldots, n, \tag{2.15}$$

ovvero gli zeri del polinomio di Chebyshev $T_n(t) = \cos(n \arccos(t))$. I pesi sono $w_k^{F1} = \int_{-1}^{1} L_k(t) d\lambda(t)$, tuttavia si possono esprimere con la formula

$$w_k^{F1} = \frac{1}{n} \left(\gamma_0 + 2 \sum_{v=1}^{n-1} \gamma_v T_v(t_k^{F1}) \right), \quad \gamma_v = \int_{-1}^{1} T_v(t) d\lambda(t).$$

La formula di quadratura di *Fejér di tipo II* rispetto alla misura $d\lambda$ ha come nodi i punti

$$t_k^{F2} = \cos(\theta_k), \quad \theta_k = \frac{k\pi}{n+1}, \quad k = 1, \ldots, n, \tag{2.16}$$

ovvero gli zeri del polinomio di Chebyshev del tipo II

$$U_n(t) = \frac{\sin((n+1)\arccos(t))}{\sin(\arccos(t))}.$$

I pesi si ricavano dalla formula

$$w_k^{F2} = \frac{2\sin(\theta_k)}{n+1} \sum_{v=0}^{n-1} \lambda_v \sin((v+1)\theta_k), \quad \lambda_v = \int_{-1}^{1} U_v(t) d\lambda(t).$$

La formula di *Clenshaw-Curtis* è la formula interpolatoria con nodi

$$t_k^{CC} = \cos\left(\frac{(k-1)\pi}{n-1} \right), \quad k = 1, \ldots, n. \tag{2.17}$$

Da notare che questi sono esattamente i nodi della formula di Fejér II a $n-2$ punti insieme agli estremi dell'intervallo -1, 1. I pesi sono

$$\begin{aligned}
w_1^{CC} &= \frac{1}{2(n-1)} \left(\gamma_0 + \sum_{v=1}^{n-2} \gamma_v + \gamma_n \right), \\
w_k^{CC} &= \frac{1}{1-t_k^2} w_{k-1}^{F2}, \quad k = 2, \ldots, n-1, \\
w_n^{CC} &= \frac{1}{2(n-1)} \left(2 \sum_{v=0}^{n-1} (-1)^v \gamma_v - \gamma_0 + (-1)^{n-2} \gamma_{n-1} \right),
\end{aligned}$$

dove w_k^{F2} sono i pesi della formula di Fejér II ad $n-2$ punti. Presentiamo alcuni metodi di integrazione implementati in Chebfun.

Il comando `sum(F)` utilizza la formula di Clenshaw-Curtis rispetto alla misura di Legendre per calcolare l'integrale definito della funzione chebfun `F` sull'intervallo $[-1, 1]$, convertendo la funzione in una serie di Chebyshev e applicando la celebrata FTT (Fast Fourier Transform) [8]. Il costo computazionale del metodo è di $\mathcal{O}(n \log n)$ operazioni dove n è il numero di nodi utilizzati dalla formula di quadratura. Tale metodo viene applicato a ogni tratto regolare di `F` (ossia a ogni polinomio a tratti di `F`).

Il comando `[t,w]=chebpts(n)` calcola esplicitamente i nodi e pesi della formula di Clenshaw-Curtis a n punti rispetto alla misura di Legendre. Questa routine è un'implementazione dell'algoritmo pubblicato da Waldvogel [14], che permette di calcolare i pesi di Clenshaw-Curtis rispetto alla misura di Legendre tramite trasformazioni discrete di Furier (DFT) di vettori di numeri complessi. Il costo computazionale dell'algoritmo è di $\mathcal{O}(n \log n)$ operazioni. In alternativa il comando `[t,w]=chebpts(n,d)` calcola i nodi e pesi della formula di Fejér I per `d=1` e Fejér II per `d=2`.

2.7 Integrazione simbolica con `Mathematica`

Data una funzione $f(t)$, risolvere il problema $\int f(t)dt$ equivale a trovare una funzione $F(t)$ per cui $F'(t) = f(t)$, ovvero una primitiva di $f(t)$. A differenza dei metodi di integrazione numerici, i metodi di integrazione simbolici al fine di risolvere integrali definiti o indefiniti cercano solitamente di trovare una primitiva dell'integranda. Alcuni algoritmi e routine adibiti all'integrazione simbolica si basano su metodi come la fattorizzazione in frazioni parziali ([7] capitolo 11) impiegata nell'integrazione di funzioni razionali, e l'algoritmo di Reisch ([7] capitolo 12).

Un esempio di tali routine è `Integrate` del software Mathematica. La chiamata `Integrate[f(t),t]` è in grado calcolare l'integrale indefinito rispetto alla variabile `t` (aggiungendo gli estremi di integrazione `tmin` e `tmin` alla chiamata viene calcolato l'integrale definito) di funzioni `f(t)` esprimibili come prodotto, somma o composizione di funzioni esponenziali, trigonometriche, iperboliche, razionali e le rispettive inverse, e quando è possibile il risultato è espresso in termini di queste stesse funzioni. Tuttavia per molte di queste funzioni Mathematica non è in grado di calcolare l'integrale (ad esempio per la funzione $e^t/\log(t)$) oppure le primitive sono espresse in termini di funzioni speciali, come ad esempio `Erf`$(x) = \int_0^x 2e^{-t^2}/\sqrt{\pi}dt$, `Li`$(x) = \int_0^x t/\log(t)dt$ o `FresnelS`$(x) = \int_0^x \sin(\pi t^2/2)dt$.

É comunque possibile valutare numericamente un integrale definito tramite la funzione `N` o `NIntegrate`. Ad esempio `N`$[\int_2^3 e^t/\log(t)dt]$ fornisce quale risultato 13.6817.

Anche in Matlab sono implementati strumenti e funzioni per il calcolo simbolico, in particolare per l'interazione è fornita la funzione `int`. Similmente a `Integrate`, la chiamata `int(f,t,tmin,tmax)` calcola l'integrale definito della funzione `f` da `tmin` a `tmax` rispetto alla variabile simbolica `t`, invece omettendo gli estremi di integrazione viene calcolata la primitiva.

Capitolo 3

Esperimenti numerici

In questa sezione svolgeremo alcuni esperimenti numerici per testare la validità delle formule di integrazione introdotte nel capitolo precedente. In particolare calcoleremo gli integrali rispetto alla misura di Legendre nell'intervallo $[-1, 1]$ di diverse tipologie di funzioni che presentano dei poli al di fuori del dominio di integrazione $[-1, 1]$.

Quali integrande considereremo

$$f_1(t) = \frac{1}{\sin(t - t_0)},$$
$$f_2(t) = \frac{1}{\cos(t - t_0) - 1},$$
$$f_3(t) = \frac{\cos t}{t^2 + \delta^2},$$
$$f_4(t) = \frac{\cos t}{(t^2 + \delta^2)(t - t_0)}, \text{ con } t_0 = 1 + \delta, \text{ e } \delta > 0.$$

Per ogni test varieremo il valore di δ in modo da modificare la distanza dei poli dall'intervallo di integrazione, pur mantenendoli al di fuori dello stesso. Inoltre calcoleremo l'integrale di ogni funzione con precisione crescente, in modo da osservare quale precisione può raggiungere ogni formula di quadratura impiegata. Il valore esatto di ogni integrale verrà calcolato con la funzione `Integrate` di Mathematica.

I test numerici sono stati eseguiti su un computer dotato di CPU AMD Ryzen 5 2600 Six-Core Processor 3,40 GHz, e 16 GB di memoria RAM.

Tests numerici sulla funzione f_1

La prima funzione che consideriamo è

$$f_1(t) = \frac{1}{\sin(t - t_0)},$$

che presenta un polo semplice in t_0.

Utilizziamo come primo metodo la formula di Gauss razionale

$$\sum_{v=1}^{n} w_v f(t_v) \approx \int_{-1}^{1} f(t)dt, \tag{3.1}$$

esatta per $f \in \mathbb{S}_d = \mathbb{Q}_{\omega_m} \oplus \mathbb{P}_{2n-2}$, dove $\omega_m(t) = t - t_0$. Per il teorema (9) ricordiamo che è necessario ricavare la formula Gaussiana

$$\sum_{v=1}^{n} w_v^G f(t_v^G) \approx \int_{-1}^{1} f(t)\frac{dt}{t-t_0}, \tag{3.2}$$

esatta per $f \in \mathbb{P}_{2n-1}$. I nodi e pesi di questa formula si ricavano tramite l'algoritmo di Golub-Welsch applicato a $\hat{J}_n = J_n(\frac{dt}{t-t_0})$, ovvero la matrice di Jacobi rispetto alla misura modificata $d\hat{\lambda}(t) = \frac{dt}{t-t_0}$.
I coefficienti di tale matrice sono $\hat{\alpha}_0, \hat{\alpha}_1, \ldots, \hat{\alpha}_{n-1}$ e $\hat{\beta}_1, \ldots, \hat{\beta}_{n-1}$.

Come abbiamo visto nel primo capitolo con l'algoritmo di Chebychev modificato possiamo calcolare questi coefficienti a partire dai momenti modificati

$$m_k = \int_{-1}^{1} p_k(t)\frac{dt}{t-t_0}, \quad k = 0, 1, \ldots, 2n-1.$$

Ricordiamo che $\{p_k\}_k$ deve essere una famiglia di polinomi monici che soddisfa una relazione di ricorrenza a tre termini

$$\begin{aligned} p_{k+1}(t) &= (t-a_k)p_k(t) - b_k p_{k-1}(t), \quad k = 0, 1, \ldots, \\ p_0(t) &= 1, \quad p_{-1}(t) = 0, \end{aligned}$$

con $a_k \in \mathbb{R}$ e $b_k \geq 0$. Dal momento che stiamo considerando la misura di Legendre a supporto compatto in $[-1, 1]$, conviene scegliere $\{p_k\}_k$ in modo che questi siano i polinomi monici ortogonali rispetto a una misura ds per cui $\text{supp}(ds) \subset \text{supp}(d\lambda)$. Infatti tale condizione può migliorare il condizionamento dell'algoritmo, come mostrato dall'analisi del condizionamento della mappa K_n che restituisce i coefficienti di ricorrenza a partire dai momenti modificati. In questo caso adopereremo i polinomi monici ortogonali rispetto alla misura di Chebyshev $ds = \frac{1}{\sqrt{1-t^2}}dt$ con supporto $[-1, 1]$. I polinomi richiesti e la relativa formula di ricorrenza sono:

$$\begin{aligned} p_0(x) \equiv 1, \quad p_k(x) &= \frac{T_k(x)}{2^{k-1}}, \quad k \geq 1 \\ a_k &= 0, \quad k \geq 0, \\ b_0 &= \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}}dt = \pi, \quad b_1 = \frac{1}{2}, \quad b_k = \frac{1}{4}, \quad k \geq 2, \end{aligned}$$

con $T_k(t) = \cos(k \arccos(t))$ il k-esimo polinomio di Chebyshev di tipo I.

A questo punto abbiamo tutto il necessario per calcolare la formula di Gauss razionale richiesta, pertanto procediamo alla descrizione della routine Matlab adibita all'implementazione della formula.

A tal proposito,

- i momenti modificati m_k, $k = 0, 1, \ldots, 2n-1$ sono stati calcolati utilizzando la funzione di integrazione `integral`;
- la routine `chebyshev`, a partire dai momenti modificati e dai coefficienti $\{a_k, b_k\}_{k=0}^{2n-2}$, calcola tramite l'algoritmo di Chebychev modificato i coefficienti di ricorrenza $\{\hat{\alpha}_k, \hat{\beta}_k\}_{k=0}^{n-1}$;
- la routine `xw=gauss(n,ab)`, dati in input un intero `n` e i coefficienti di ricorrenza `ab` (in questo caso $\{\hat{\alpha}_k, \hat{\beta}_k\}_{k=0}^{n-1}$), calcola tramite l'algoritmo di Golub-Welsch i nodi e pesi w_v^G, t_v^G, per $v = 1, \ldots, n$ della formula Gaussiana (3.2);
- infine ricaviamo la formula (3.1) in quanto $t_v = t_v^G$ e $w_v = w_v^G(t_v - t_0)$ per $v = 1, \ldots, n$.

Nella tabella sottostante riportiamo la *cputime* e il numero di nodi impiegati dalla formula di Gauss razionale per calcolare l'integrale della funzione f_1 con polo $t_0 = 1 + \delta$, con tolleranza per l'errore relativo indicata dalla prima riga della tabella. Osserviamo che riusciamo a raggiungere per ogni scelta di δ in 10^{-1}, 10^{-2}, 10^{-5}, 10^{-7}, un'errore inferiore a 10^{-10}. Tuttavia la formula non è in grado di calcolare l'integrale di f_1 con un polo in $t_0 = 1 + 10^{-5}$, e $1 + 10^{-7}$, con un'errore minore di 10^{-12}.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	6	0.13	7	0.13	9	0.15	11	0.19
10^{-2}	5	0.096	7	0.14	9	0.17	10	0.18
10^{-5}	5	0.10	7	0.13	8	0.14	–	–
10^{-7}	5	0.10	7	0.13	9	0.16	–	–

Tabella 3.1: Questa tabella riporta la cputime (in secondi) e il numero di nodi impiegati per calcolare l'integrale di f_1 con una tolleranza per l'errore relativo pari a $10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$. Il metodo di integrazione utilizzato è la formula di *Gauss razionale* calcolata con i momenti modificati. L'integrale di f_1 viene calcolato al variare di $t_0 = 1 + \delta$, con $\delta = 10^{-1}, 10^{-2}, 10^{-5}, 10^{-7}$.

Come indicato precedentemente, possiamo calcolare la formula di Gauss razionale (3.1) discretizzando la misura $d\hat{\lambda}(t)$, con cui calcolare i coefficienti $\{\hat{\alpha}_k, \hat{\beta}_k\}_{k=0}^{n-1}$. La misura discreta sarà

$$d\lambda_N = \sum_{v=1}^{N} w_{v,N}\delta_{t_{v,N}}(t), \quad \text{con} \quad t_{v,N} = t_{v,N}^G \text{ e } w_{v,N} = \frac{w_{v,N}^G}{t_{v,N}^G - t_0}, \tag{3.3}$$

dove $t_{v,N}^G$ e $w_{v,N}^G$, $v = 1, \ldots, N$ sono i nodi e pesi della formula di Gauss Legendre a N punti.

La funzione Matlab che implementa tale metodo è

```
[abmod,Ncap,kount]=r_mod(N,ab).
```

Questa

- riceve in input un intero N, e i coefficienti α_k, β_k per $k = 0, 1, \ldots, Nmax.$,
- fornisce quale output i coefficienti di ricorrenza $\hat{\alpha}_k$, $\hat{\beta}_k$ con $k = 0, 1, \ldots, N$, oltre ai parametri `Ncap` e `kount` che si riferiscono rispettivamente alla raffinatezza della discretizzazione della misura, e al numero di iterazioni eseguite per raggiungere una tale raffinatezza. Questi parametri sono restituiti dal metodo `mcdis`, nel caso in cui ω_m non sia una funzione costante.

Il metodo esegue la discretizzazione della misura modificata, applicando all'intervallo $[-1, 1]$ la formula di quadratura ad N punti con nodi e pesi definiti in (3.3). Tale formula è calcolata richiamando la routine `quadrat`. Dai nodi e pesi definiti in (3.3) si calcolano i coefficienti di ricorrenza modificati `abmod` applicando la procedura di Stieltjes o Lanczos. Questi ultimi sono implementati in `stieltjes` o `lanczos`. Inoltre il metodo può raffinare la discretizzazione della misura incrementando il numero di nodi della formula di quadratura. I nodi vengono incrementati fino a $Nmax$ oppure finchè il controllo (2.8) non risulta verificato.

Come riportato nella tabella sottostante, il metodo può calcolare l'integrale di f_1 con $t_0 = 1 + \delta$, con δ in 10^{-1}, 10^{-2}, 10^{-5}, 10^{-7}, con un errore relativo almeno di 10^{-12}, tuttavia se f_1 presenta poli più vicini all'intervallo di integrazione, per esempio $t_0 = 1 + 10^{-7}$ il metodo non riesce a fornire il risultato con errori inferiori o uguali a 10^{-6}.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	6	0.0071	7	0.0070	9	0.0082	11	0.0064
10^{-2}	5	0.010	7	0.011	9	0.011	10	0.011
10^{-5}	17	0.084	22	0.30	28	0.29	35	0.53

Tabella 3.2: In questa tabella riportiamo la cputime e il numero di nodi impiegati per calcolare l'integrale di f_1 con una tolleranza per l'errore relativo pari a $10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$. Il metodo di integrazione utilizzato è la formula di *Gauss razionale* discretizzato. L'integrale di f_1 viene calcolato al variare di $t_0 = 1 + \delta$, con $\delta = 10^{-1}, 10^{-2}, 10^{-5}$.

Ora utilizziamo, per i propositi indicati in precedenza, la routine di `rfejer`. Come spiegato nel capitolo precedente è necessario fornire in input il vettore `sgl` contenente i poli dell'integranda. Aiutandoci con la routine `encodeRFejer`, costruiremo `sgl`, in modo che contenga il polo t_0 oltre a un certo numero di poli infiniti.
Se `sgl=[t_0,inf,...,inf]` ha lunghezza $n - 1$ allora la chiamata

```
xw=rfejer(sgl)
```

calcola i nodi e pesi della formula di Fejer razionale a n punti.

Per quanto concerne i test numerici, il numero di nodi impiegato dalla formula della routine viene incrementato aggiungendo a ogni iterazione della formula poli infiniti nel vettore `sgl`, e lasciando invariati i poli finiti, che devono corrispondere ai poli dell'integranda (in questo caso t_0). Osserviamo che incrementando i nodi in questa maniera, aumenta la precisione della formula, sebbene per $\delta = 10^{-7}$ non riesca a calcolare l'integrale con un errore relativo minore di 10^{-12}.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	8	0.0020	11	0.0022	14	0.010	17	0.010
10^{-2}	8	0.0021	11	0.0021	14	0.010	17	0.011
10^{-5}	8	0.0022	10	0.0021	13	0.011	14	0.010
10^{-7}	7	0.080	10	0.0029	20	0.012	–	–

Tabella 3.3: Questa tabella riporta la cputime e il numero di nodi impiegati per calcolare l'integrale di f_1 con $t_0 = 1 + \delta$, con $\delta = 10^{-1}, 10^{-2}, 10^{-5}, 10^{-7}$ utilizzando la formula di `rfejer`.

Applichiamo quale confronto finale, i metodi `sum` di `Chebfun`

```
g=chebfun(f,'splitting','on'); I=sum(g);
```

e la funzione adattiva `integral`

```
I=integral(f,-1,1).
```

Dalla tabella seguente si nota come le routine sono in grado di calcolare gli integrali richiesti anche per valori di $\delta = 10^{-9}, 10^{-11}$, sebbene perdano un po' di precisione per i valori di δ più piccoli.

	`integral`		`sum`	
δ	Err.	cputime	Err.	cputime
10^{-1}	$1.0e-15$	0.0020	$8.8e-16$	0.0028
10^{-2}	$7.7e-16$	0.0010	$7.7e-16$	0.052
10^{-5}	$1.4e-12$	0.0011	$4.7e-13$	0.051
10^{-7}	$4.4e-11$	0.0011	$3.2e-11$	0.062
10^{-9}	$4.2e-09$	0.0015	$3.8e-09$	0.056
10^{-11}	$5.6e-10$	0.0016	$3.8e-08$	0.062

Tabella 3.4: Questa tabella riporta l'errore relativo e la cputime (in secondi) impiegati per calcolare l'integrale di f_1 con $t_0 = 1+\delta$, con le routine di `integral` e `sum` di Chebfun.

Quindi dopo questo test ricaviamo che per $t_0 = 1+\delta$, con $\delta = 10^{-1}, 10^{-2}, 10^{-5}, 10^{-7}$, l'integranda f_1 é tale che

- il metodo *Gauss razionale* fornisce risultati soddisfacenti, con tolleranze 10^{-10} in circa 1 decimo di secondo ma non é in grado per poli vicini a 1 di al piú 10^{-7} di andare a tolleranze inferiori;

- il metodo *Gauss razionale discretizzato* fornisce risultati soddisfacenti, con tolleranze di 10^{-12}, con tempi di calcolo da un 1 millesimo di secondo fino ad alcuni decimi di secondo, ma per poli vicini a 1 di al piú 10^{-7} non riesce a fornire i risultati richiesti anche a tolleranze dell'ordine di 10^{-6};

- la procedura *rfejer* fornisce risultati soddisfacenti, con tolleranze di 10^{-12}, con tempi di calcolo da un 1 millesimo di secondo fino ad alcuni centesimi di secondo ma non é in grado per poli vicini a 1 di al piú 10^{-7} di fornire i risultati richiesti a tolleranze inferiori a 10^{-12};

- le procedure *integral* e *sum* risultano in genere piú rapide dei metodi precedenti approssimando bene anche poli vicini a 1 di al piú 10^{-9}, in al piú centesimi di secondo, ma con errori non inferiori a 10^{-10}.

Tests numerici sulla funzione f_2

Passiamo alla seconda funzione:

$$f_2(t) = \frac{1}{\cos(t - t_0) - 1}.$$

Questa presenta un polo reale t_0 di molteplicità 2 vicino all'intervallo di integrazione.
Riportiamo nella tabella sottostante gli errori relativi e le cputime registrati nell'esecuzione della formula di Gauss razionale calcolata con i momenti modificati. Il metodo è in grado di integrare la funzione f_2 con il polo t_0 vicino al più 10^{-5} all'intervallo in tempi dell'ordine dei decimi di secondo, e utilizzando un numero esiguo di nodi (meno di 10). Tuttavia, come per la funzione f_1, la formula non è in grado di calcolare l'integrale con un errore minore di 10^{-10} quando t_0 ha una distanza minore di 10^{-5} dall'intervallo.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	4	0.101	5	0.107	6	0.122	7	0.139
10^{-2}	3	0.083	4	0.093	5	0.108	6	0.121
10^{-5}	2	0.066	3	0.079	–	–	–	–

Tabella 3.5: Questa tabella riporta la cputime (in secondi) e il numero di nodi impiegati per calcolare l'integrale di f_2 con una tolleranza per l'errore relativo pari a $10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$. Il metodo di integrazione utilizzato è la formula di *Gauss razionale* calcolata con i momenti modificati. L'integrale di f_2 viene calcolato al variare di $t_0 = 1 + \delta$, con $\delta = 10^{-1}, 10^{-2}, 10^{-5}$.

Il metodo di Gauss Razionale discretizzato ottiene i risultati riportati nella tabella sottostante: per $\delta = 10^{-1}$ il calcolo dell'integrale richiede pochi nodi e dei tempi di calcolo costanti, dell'ordine dei millesimi di secondo. Similmente avviene per $\delta = 10^{-2}$ con la differenza che le cputime richieste sono dell'ordine dei centesimi di secondo. Per $\delta = 10^{-5}$ i nodi richiesti sono più numerosi, con tempi di calcolo dell'ordine dei decimi di secondo, inoltre non è possibile raggiungere una precisione maggiore di 10^{-10}.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	4	0.0060	5	0.0057	6	0.0058	7	0.0054
10^{-2}	3	0.011	4	0.011	5	0.011	6	0.011
10^{-5}	23	0.31	32	0.62	–	–	–	–

Tabella 3.6: In questa tabella riportiamo la cputime e il numero di nodi impiegati per calcolare l'integrale di f_2 con una tolleranza per l'errore relativo pari a $10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$. Il metodo di integrazione utilizzato è la formula di *Gauss razionale* discretizzato. L'integrale di f_2 viene calcolato al variare di $t_0 = 1 + \delta$, con $\delta = 10^{-1}, 10^{-2}, 10^{-5}$.

La routine di `rfejer`, similmente alle due formule precedenti, richiede tempi di calcolo dell'ordine dei millesimi di secondo, e raggiunge la massima precisione richiesta per $\delta = 10^{-1}, 10^{-2}$. Per $\delta = 10^{-5}$ non è in grado di raggiungere una precisione maggiore di 10^{-10}.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	6	0.0033	7	0.0060	9	0.0077	11	0.0096
10^{-2}	5	0.0067	7	0.0062	9	0.0081	10	0.0093
10^{-5}	3	0.018	8	0.019	–	–	–	–

Tabella 3.7: Questa tabella riporta la cputime e il numero di nodi impiegati per calcolare l'integrale di f_2 con $t_0 = 1 + \delta$, con $\delta = 10^{-1}, 10^{-2}, 10^{-5}$ utilizzando la formula di `rfejer`.

I risultati ottenuti con `integral` e `sum` mostrano che queste due funzioni sono in grado di calcolare l'integrale senza problemi per $\delta = 10^{-1}, 10^{-2}$. Per $\delta = 10^{-4}$ c'è un calo della precisione che per $\delta = 10^{-5}$ diventa consistente.

	`integral`		`sum`	
δ	Err.	cputime	Err.	cputime
10^{-1}	$1.8e-15$	0.00044	$2.9e-15$	0.0027
10^{-2}	$1.6e-15$	0.00054	$4.3e-14$	0.053
10^{-4}	$2.9e-10$	0.00079	$3.2e-10$	0.061
10^{-5}	$1.6e-07$	0.00093	$2.3e-08$	0.084

Tabella 3.8: Questa tabella riporta l'errore relativo e la cputime (in secondi) impiegati per calcolare l'integrale di f_2 con $t_0 = 1+\delta$, con le routine di `integral` e `sum` di Chebfun.

Da questi test sulla funzione f_2 osserviamo:

- il metodo *Gauss razionale* calcolato con i momenti modificati fornisce risultati soddisfacenti, con tolleranze fino a 10^{-12} in circa 1 decimo di secondo, ma per poli distanti da 1 al piú 10^{-5} non è in grado di raggiungere tolleranze inferiori a 10^{-10};
- il metodo *Gauss razionale discretizzato* fornisce risultati soddisfacenti, con tolleranze 10^{-12}, con tempi di calcolo di circa 1 centesimo di secondo per poli distanti da 1 al più 10^{-2}, ma non è in grado di fornire i risultati richiesti a tolleranze dell'ordine di 10^{-10} per poli distanti da 1 al piú 10^{-5};
- la procedura *rfejer* fornisce risultati soddisfacenti, con tolleranze 10^{-10}, con tempi di calcolo nell'ordine dei millesimi di secondo ma non é in grado per poli vicini a 1 al piú 10^{-5} di fornire i risultati richiesti a tolleranze minori di 10^{-10};
- le procedure *integral* e *sum* sono in grado di raggiungere risultati soddisfacenti per poli distanti da 1 almeno 10^{-2}; per poli più vicini sono presenti delle perdite di precisione significative.

Tests numerici sulla funzione f_3

Eseguiamo ora gli stessi tests precedenti relativamente alla funzione

$$f_3(t) = \frac{\cos t}{t^2 + \delta^2}.$$

Questa presenta due poli complessi coniugati in $t_1 = i\delta$ e $t_2 = -i\delta$.

La formula di Gauss razionale calcolata con i momenti modificati è in grado di integrare f_3 con t_1 e t_2 vicini al più 10^{-5} all'intervallo di integrazione, in circa 1 decimo di secondo, e utilizzando meno di 6 nodi. Tuttavia per poli piú vicini non siamo in grado di calcolare l'integrale a una tolleranza accettabile.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	4	0.097	5	0.11	5	0.11	6	0.12
10^{-2}	3	0.081	4	0.094	5	0.11	6	0.12
10^{-5}	2	0.068	3	0.082	4	0.099	5	0.11

Tabella 3.9: Questa tabella riporta la cputime (in secondi) e il numero di nodi impiegati per calcolare l'integrale di f_3 con una tolleranza per l'errore relativo pari a $10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$. Il metodo di integrazione utilizzato è la formula di *Gauss razionale* calcolata con i momenti modificati. L'integrale di f_3 viene calcolato al variare della distanza $\delta = 10^{-1}, 10^{-2}, 10^{-5}$ dei poli dall'intervallo di integrazione.

La formula di Gauss razionale discretizzata è in grado di calcolare l'integrale in tempi brevi per $\delta = 10^{-1}, 10^{-2}$. Per $\delta = 10^{-3}$ invece, richiede fino a 158 nodi per calcolare l'integrale con una precisione superiore a 10^{-12}, allungando di conseguenza notevolmente i tempi di calcolo nell'ordine dei secondi.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	4	0.0060	5	0.0073	5	0.0075	6	0.0086
10^{-2}	9	0.052	11	0.069	14	0.10	16	0.125
10^{-3}	81	4.2	107	8.6	132	15	158	24

Tabella 3.10: In questa tabella riportiamo la cputime e il numero di nodi impiegati per calcolare l'integrale di f_3 con una tolleranza per l'errore relativo pari a $10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$. Il metodo di integrazione utilizzato è la formula di *Gauss razionale* discretizzato. L'integrale di f_3 viene calcolato al variare di $\delta = 10^{-1}, 10^{-2}, 10^{-3}$.

La routine `rfejer` richiede tempi di calcolo nell'ordine dei centesimi di secondo e in alcuni casi dei millesimi di secondo. Raggiunge una tolleranza di 10^{-10} per $\delta = 10^{-7}$.

Per quanto riguarda la formula adattiva `integral`, questa calcola l'integrale con un errore nell'ordine della precisione di macchina, in tempi inferiori al millesimo di secondo per valori di δ fino a 10^{-10}. La routine `sum` raggiunge un una precisione di 10^{-12} soltanto per valori di δ fino a 10^{-7}, con tempi di calcolo decisamente più lunghi della routine `integral`.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	6	0.0039	7	0.0060	9	0.0077	11	0.0096
10^{-2}	5	0.0035	7	0.0062	9	0.0085	9	0.0084
10^{-5}	3	0.0061	5	0.010	6	0.012	12	0.023
10^{-7}	3	0.0059	3	0.0058	14	0.029	–	–

Tabella 3.11: Questa tabella riporta la cputime e il numero di nodi impiegati per calcolare l'integrale di f_3 con $\delta = 10^{-1}, 10^{-2}, 10^{-5}, 10^{-7}$ utilizzando la formula di `rfejer`.

	`integral`		`sum`	
δ	Err.	cputime	Err.	cputime
10^{-1}	$1.5e-15$	0.00035	$2.9e-15$	0.0027
10^{-2}	$1.1e-13$	0.00042	$4.3e-14$	0.053
10^{-5}	$7.7e-14$	0.00068	$3.2e-10$	0.061
10^{-7}	$2.1e-15$	0.00087	$5.5e-12$	0.19
10^{-10}	$2.4e-16$	0.0012	$2.0e-03$	0.19

Tabella 3.12: Questa tabella riporta l'errore relativo e la cputime (in secondi) impiegati per calcolare l'integrale di f_3 al variare di δ, con le routine di `integral` e `sum` di Chebfun.

Da questi test sulla funzione f_3 facciamo le seguenti osservazioni:

- il metodo *Gauss razionale* calcolato con i momenti modificati fornisce risultati soddisfacenti, con tolleranza di 10^{-12} in circa 1 decimo di secondo, per poli vicini a 1 al piú 10^{-5};
- il metodo *Gauss razionale discretizzato* fornisce risultati soddisfacenti, con tolleranza di 10^{-12}. I tempi di calcolo sono dell'ordine dei millesimi di secondo per poli distanti 10^{-1} da 1 e dei centesimi di secondo se i poli distano 10^{-2} da 1. Per poli tali che $\delta = 10^{-3}$ i tempi di calcolo sono molto più lunghi, nell'ordine dei secondi;
- la procedura *rfejer* fornisce risultati soddisfacenti, con tolleranze di 10^{-12}, e con tempi di calcolo compresi tra il millesimi e centesimo di secondo, ma non é in grado per poli vicini a 1 al piú 10^{-7} di raggiungere una precisione di 10^{-12};
- la funzione *integral* raggiunge risultati soddisfacenti per poli vicini a 1 al piú 10^{-10}, con tempi di calcolo inferiori al millesimo di secondo. *sum* raggiunge ottimi risultati in termini di tempi, e in termini di errori, questi ultimi inferiori a 10^{-10} per poli vicini a 1 al piú 10^{-7}, mentre per poli più vicini

all'intervallo di integrazione non è in grado di fornire un'approssimazione accettabile.

Tests numerici sulla funzione f_4

Consideriamo quale ultimo esempio la funzione

$$f_4(t) = \frac{\cos t}{(t^2+\delta^2)(t-t_0)},$$

che presenta un polo reale semplice $t_0 = 1+\delta$ vicino all'estremo di integrazione 1, e due poli complessi coniugati $t_1 = i\delta$, e $t_2 = -i\delta$.

Riportiamo le tabelle dei test numerici effettuati con i diversi metodi di integrazione.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	4	0.10	5	0.11	6	0.13	6	0.12
10^{-2}	4	0.10	4	0.10	5	0.12	6	0.13
10^{-5}	2	0.076	3	0.090	–	–	–	–

Tabella 3.13: Questa tabella riporta la cputime (in secondi) e il numero di nodi impiegati per calcolare l'integrale di f_4 con una tolleranza per l'errore relativo pari a $10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$. Il metodo di integrazione utilizzato è la formula di Gauss razionale calcolata con i momenti modificati. L'integrale di f_4 viene calcolato al variare della distanza δ, tra l'intervallo di integrazione e poli $t_0 = 1+\delta$, $t_1 = i\delta$, $t_2 = -i\delta$, con $\delta = 10^{-1}, 10^{-2}, 10^{-5}$.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	4	0.015	5	0.015	6	0.014	5	0.016
10^{-2}	9	0.051	11	0.068	14	0.096	16	0.12
10^{-3}	81	4.1	107	8.3	132	15	158	25

Tabella 3.14: In questa tabella riportiamo la cputime e il numero di nodi impiegati per calcolare l'integrale di f_4, utilizzando la formula di *Gauss razionale* discretizzato. L'integrale di f_4 viene calcolato al variare di $\delta = 10^{-1}, 10^{-2}, 10^{-3}$.

	Err.=10^{-6}		Err.=10^{-8}		Err.10^{-10}		Err.=10^{-12}	
δ	n	cputime	n	cputime	n	cputime	n	cputime
10^{-1}	5	0.0026	7	0.0042	9	0.0062	11	0.0082
10^{-2}	5	0.0032	7	0.0053	9	0.0078	9	0.0073
10^{-5}	4	0.0054	4	0.0054	6	0.010	7	0.012

Tabella 3.15: Questa tabella riporta la cputime e il numero di nodi impiegati per calcolare l'integrale di f_4 utilizzando la formula di `rfejer`.

	`integral`		`sum`	
δ	Err.	cputime	Err.	cputime
10^{-1}	$1.7e-15$	0.00034	$9.0e-16$	0.0083
10^{-2}	$1.3e-13$	0.00041	$1.1e-15$	0.039
10^{-5}	$1.3e-13$	0.00068	$1.3e-15$	0.13
10^{-7}	$4.0e-15$	0.00089	$2.7e-13$	0.16
10^{-10}	$3.6e-16$	0.0012	$2.0e-03$	0.23

Tabella 3.16: La tabella riporta l'errore relativo e la cputime (in secondi) impiegati per calcolare l'integrale di f_4 al variare di δ, con le routine di `integral` e `sum` di Chebfun.

Da questi test sulla funzione f_4 risulta che:

- il metodo *Gauss razionale* calcolato con i momenti modificati fornisce risultati soddisfacenti soltanto per poli distanti 10^{-1} e 10^{-2} dall'intervallo di integrazione con tempi di calcolo che valgono circa 1 decimo di secondo. Se i poli distano meno di 10^{-5} i tempi di calcolo si allungano, e non vengono forniti risultati con tolleranze per l'errore inferiori a 10^{-10};

- il metodo *Gauss razionale discretizzato* fornisce risultati soddisfacenti, con una tolleranza di 10^{-12} solo per poli distanti al più 10^{-3} dall'intervallo di integrazione. Inoltre i tempi di calcolo sono contenuti tra il centesimo e il decimo di secondo quando $\delta = 10^{-1}, 10^{-2}$, mentre per $\delta = 10^{-3}$ i tempi di calcolo sono molto più lunghi, nell'ordine dei secondi;

- la procedura *rfejer* fornisce l'integrale richiesto per tutti i poli distanti al più 10^{-5} dall'intervallo di integrazione, con con una tolleranza di 10^{-12}, e con tempi di calcolo nell'ordine dei millesimi di secondo;

- la funzione *integral* raggiunge risultati soddisfacenti per poli vicini a 1 al piú 10^{-10}, con tempi di calcolo inferiori al millesimo in quasi tutti i casi considerati. *sum* raggiunge una tolleranza dell'errore di 10^{-13} per poli aventi una distanza dall'intervallo di integrazione di al più 10^{-7}. I tempi di calcolo sono compresi tra i millesimi e i decimi di secondo.

Commento finale

Come indicato, il calcolo di funzioni razionali con poli risulta un problema di non semplice soluzione.

Abbiamo confrontato delle routines che sfruttano la conoscenza dei poli, per integrare una famiglia di funzioni razionali con queste singolaritá. In questo caso si nota che la presenza di poli prossimi agli estremi di integrazione porta a non fornire il risultato richiesto con tolleranze attorno a 10^{-10}. Questo vale per le formule di tipo gaussiano come in quelle proposte da Deckers e collaboratori, che hanno comunque tempi di calcolo molto competitivi. Per quest'ultimo algoritmo risulta particolarmente ostica la descrizione dei parametri di input, per cui abbiamo creato una subroutine che li determini a partire dai poli noti.

Le alternative sono le routines adattative `integral` e `sum` di Chebfun, che hanno in genere fornito risultati accettabili anche per poli estremamente prossimi agli estremi. Dai confronti, `integral` risulta in genere piú rapida e robusta.

Il codice Matlab utilizzato é disponibile open-source nel cloud GitHub [16].

Bibliografia

[1] K. Deckers, A. Mougaida, H. Belhadjsalah, *Algorithm 973: Extended Rational Fejér Quadrature Rules Based on Chebyshev Orthogonal Rational Functions*, ACM Transactions on Mathematical Software, 43 (4) (2017), pp. 1–29.

[2] T.A. Driscoll, N. Hale, L.N. Trefethen, *Chebfun guide*, (2014).

[3] W. Gautschi, *Gauss-type Quadrature Rules for Rational Functions*, Numerical Integration IV, pp.111-130.

[4] W. Gautschi, *Orthogonal polynomials: applications and computation*, Acta Numerica 5 (1996), pp.45-119.

[5] W. Gautschi, *The use of rational functions in numerical quadrature*, Journal of Computational and Applied Mathematics, 133 (1–2) (2001), pp.111-126.

[6] W. Gautschi, *Orthogonal Polynomials Computation and Approximation*, Oxford University Press, 2004.

[7] K.O. Geddes, S.R. Czapor, G. Labahn. *Algorithms for computer algebra*. Springer Science & Business Media, 1992.

[8] W.M. Gentleman, *Implementing Clenshaw-Curtis quadrature I and II*, Journal of the ACM, 15 (1972), pp.337-346.

[9] N. Hale, A. Townsend, *Fast and accurate computation of Gauss-Legendre and Gauss-Jacobi quadrature nodes and weights*, SIAM Journal on Scientific Computing, 35, (2) (2013), pp.A652-A674.

[10] MathWorks, *Integral*, `https://it.mathworks.com/help/matlab/ref/integral.html`

[11] C. Moler, *Modernization of Numerical Integration, From Quad to Integral*, `https://blogs.mathworks.com/cleve/2016/05/23/modernization-of-numerical-integration-from-quad-to-integral/`

[12] L.F. Shampine, *Vectorized Adaptive Quadrature in MATLAB*, Journal of Computational and Applied Mathematics, 211 (2008), pp.131–140.

[13] A. Sommariva, *Fast Construction of Fejér and Clenshaw-Curtis rules for general weight functions*, Computers & Mathematics with Applications, 65 (4) (2013), pp. 682-693.

[14] J. Waldvogel. *Fast construction of the Fejér and Clenshaw–Curtis quadrature rules*, BIT Numerical Mathematics 46 (1) (2006), pp.195-202.

[15] Wolfram Research (1988, 2019), *Integrate, Wolfram Language function*, `https://reference.wolfram.com/language/ref/Integrate.html`

[16] N. Zorzi, Software per cubatura di funzioni razionali, `https://github.com/nicolozorzi/Codice_cubatura_funzioni-_razionali`