UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Elaborato d'Esame

CALCOLO NUMERICO

MASSIMO NOCENTINI

Professore: *Luigi Brugnano*

Assistente: Alessandra Sestini

Anno Accademico 2010-2011

INDICE

INTRODUZIONE

Queste note contengono tutto il mio materiale di studio per l'esame di Calcolo Numerico.

Sintassi esercizi

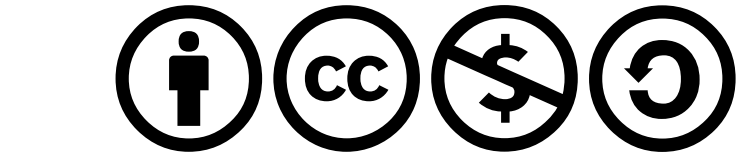
Per gli esercizi utilizzo questa sintassi **Exercise n[(m)]**, dove n rappresenta la numerazione locale all'interno delle sezioni di questo documento, m rappresenta l'identificativo dell'esercizio fissato nel libro di testo **Calcolo Numerico** di *Brugnano, Magherini, Sestini* pubblicato da *Master*, prima edizione. Il reference (m) non è sempre presente, come evidenziato dall'uso delle parentesi quadre.

LICENZE

Solo questa sezione dedicata alle licenze verrà scritta completamente in inglese.

Text contents

All the text content is distributed under:
This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.or nc-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.



Octave sources

All the Octave sources are distributed under, where the word "Software" is referred to all of the sources that are present in this work:
Copyright (c) 2011 Massimo Nocentini
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the Software), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED AS IS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1

ERRORI ED ARITMETICA FINITA

1.1 DISCRETIZZAZIONE

Questi errori sorgono tutte le volte che si vuole modellare un problema matematico (formulato nel continuo) con un sistema discreto. Per ottenere questo, uso dei processi di discretizzazione, dei quali mi interessa controllare quanto "bene" approssimano il problema che voglio modellare.

Più specificamente, l'errore di troncamento *locale* corrisponde all'errore introdotto nel passo di integrazione corrente assumendo esatto il valore di partenza, mentre l'errore di troncamento *globale* rappresenta l'effetto di tutti gli errori precedenti.

Per i seguenti due esercizi si utilizza la formula di Taylor-Peano, con il resto "infinitesimo di ordine superiore" rispetto al massimo grado n a cui si arresta lo sviluppo e quindi consente di ottenere una approssimazione locale, cioè in un intorno di x_0.

Exercise 1.1.1 (1.1). *Sia* $x = \pi \approx 3.14159265$. *Considero come valore approssimato* $\tilde{x} = 3.1415$.

Calcolare il corrispondente errore relativo[1] ε_x.

Verificare che il numero di cifre decimali corrette nella rappresentazione approssimata di x *mediante* $\tilde{x}$ *all'incirca è dato da* $-\log_{10}|\varepsilon_x|$.

Ottengo:

$$\varepsilon_x = \frac{\tilde{x} - x}{x} = \frac{3.1415 - 3.14159265358979}{3.14159265358979}$$

```
octave:19> (3.1415-pi)/pi
ans = -2.94925536215087e-05
```

Adesso considero:

$$-\log_{10}|-2.9491 \times 10^{-5}| = -\left(\log_{10} 2.9491 + \log_{10} 10^{-5}\right) = -\log_{10} 2.9491 + 5\log_{10} 10$$

```
octave:18> -log10(2.9491) + 5*log10(10)
ans =  4.53031050085890e+00
```

L'approssimazione $\tilde{x}$ ha 4 cifre decimali corrette.

1 Dá l'ordine di grandezza rispetto alla base 10

Exercise 1.1.2. *Dimostrare che* $-\log_{10}|\varepsilon_x|$ *dà all'incirca il numero di cifre decimali corrette di* $\tilde{x}$*, con cui approssimo* x*.*

Dimostrazione. Sia r il numero di cifre decimali *esatte*, tale che $r = -\log_{10}|\varepsilon_x|$. Passando agli esponenti ottengo $|\varepsilon_x| = 10^{-r}$.

Dato che cerco un'approssimazione del numero di cifre decimali a partire dall' errore relativo, imposto due disequazioni per trovare un intervallo in cui r può variare.

Scrivo in forma normalizzata il valore esatto x e la sua approssimazione $\tilde{x}$, fissando $\beta = 10$ e $M \neq m$ perchè sto approssimando:

$$x \in \mathbb{R}, x \geqslant 0, x = m \times 10^e$$
$$\tilde{x} \in \mathbb{R}, \tilde{x} \geqslant 0, \tilde{x} = M \times 10^e$$

Per il *teorema 1.2* vale $1 \leqslant m < 10 \wedge 1 \leqslant M < 10$.

Adesso voglio trovare sia una maggiorazione che una minorazione di ε_x:

$$|\varepsilon_x| = \left|\frac{(M-m)\times 10^e}{m \times 10^e}\right| = \left|\frac{M-m}{m}\right|$$

Maggioro con:

$$\left|\frac{M-m}{m}\right| < 10 = max$$

In quanto il massimo lo ottengo quando $M = 9, m = 1$.

Minoro con:

$$min = \frac{1}{10} \leqslant \left|\frac{M-m}{m}\right|$$

In quanto il minimo lo ottengo quando $M = 8, m = 9$.

Adesso posso impostare le due disequazioni:

$$10^{-r-1} = \frac{|\varepsilon_x|}{10} \leqslant |\varepsilon_x| < 10|\varepsilon_x| = 10^{-r+1}$$

Le uguaglianze esterne valgono per quanto detto ad inizio prova. Passo ai logaritmi:

$$-r-1 \leqslant \log_{10}|\varepsilon_x| < -r+1$$
$$r+1 \geqslant -\log_{10}|\varepsilon_x| > r-1$$

Considero i rami $-r-1 \leqslant \log_{10}|\varepsilon_x|$ dalla prima e $-\log_{10}|\varepsilon_x| > r-1$ dalla seconda per avere l'intervallo di variazione di r:

$$-1-\log_{10}|\varepsilon_x| \leqslant r < 1-\log_{10}|\varepsilon_x|$$

□

Exercise 1.1.3 (1.2). *Dimostrare che, se $f(x)$ è sufficientemente regolare e $h > 0$ è una quantità piccola, allora:*

$$\frac{f(x_0+h)-f(x_0-h)}{2h} = f'(x_0) + O(h^2),$$
$$\frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} = f''(x_0) + O(h^2)$$

Per entrambe considero gli sviluppi di Taylor di $f(x_0+h)$ e $f(x_0-h)$ in x_0:

$$T(x_0+h) = f(x_0) + f'(x_0)h + \frac{f''(x_0)h^2}{2} + \frac{f'''(x_0)h^3}{6} + O(h^4)$$
$$T(x_0-h) = f(x_0) - f'(x_0)h + \frac{f''(x_0)h^2}{2} - \frac{f'''(x_0)h^3}{6} + O(h^4)$$

Devo prestare attenzione al segno di alcuni termini dello sviluppo [2]:

- nel primo sviluppo, la combinazione lineare ammette tutti segni positivi in quanto il passo di discretizzazione è positivo, ovvero si cerca di approssimare con valori $> x_0$.
- nel secondo invece, il fattore $((x_0 - h - x_0 = h))^k < 0, \forall k = 2n+1, n \in \mathbb{N}$ fa si che i termini dello sviluppo di grado dispari siano negativi, in quanto si sta discretizzando con un valore minore a x_0, di conseguenza $x - x_0 < 0$.

Sottraendo termine a termine e semplificando dove possibile ottengo la prima equazione:

$$T(x_0+h) - T(x_0-h) = 2f'(x_0)h + \frac{f'''(x_0)h^3}{3} + O(h^4)$$

dividendo per $2h$:

$$\frac{T(x_0+h)-T(x_0-h)}{2h} = f'(x_0) + \frac{f'''(x_0)h^2}{3} + O(h^3)$$

Osserviamo che, per $h \to 0$, la quantità $O(h^3)$ diminuisce più velocemente del termine

$$\frac{f'''(x_0)h^2}{3}$$

per cui possiamo dedurre che si approssima la derivata prima con una quantità $O(h^2)$.

Per la seconda equazione invece che sottrarre termine a termine, sommiamo, ottenendo:

$$T(x_0+h) + T(x_0-h) = 2f(x_0) + f''(x_0)h^2 + O(h^4)$$
$$\frac{T(x_0+h)-2f(x_0)+T(x_0-h)}{h^2} = f''(x_0) + O(h^2)$$

2 Attenzione: nello sviluppo di Taylor di una funzione $f(x)$, le derivate n-esime vengono calcolate nel punto x_0 in cui si vuole centrare lo sviluppo, e non nell'argomento della funzione

ovvero la quantità al primo membro approssima la $f''(x_0)$ con un errore dell'ordine $O(h^2)$.

1.2 CONVERGENZA

Exercise 1.2.1 (1.3)**.** *Dimostrare che il metodo iterativo*

$$x_{n+1} = \phi(x_n)$$

convergente a x, deve verificare la condizione di* ***consistenza***

$$x^* = \phi(x^*)$$

ovvero la soluzione cercata deve essere un punto fisso per la funzione di iterazione che definisce il metodo.

Dimostrazione. Suppongo che il metodo ϕ sia monotono e convergente a x^*. Definisco il preordine $\rightarrow$ per catturare la convergenza:

$$\rightarrow = \{(x_n, x_{n+1}) : x_{n+1} = \phi(x_n) \wedge \lim_{n\to\infty} x_n = x^*\}$$

Suppongo *per assurdo* che

$$\phi(x^*) = x^{\triangle} \neq x^*$$

Per l' ipotesi $x_n \rightarrow x^*$ (per la transitività di $\rightarrow$), posso applicare la monotonia di ϕ rispetto a $\rightarrow$:

$$\phi(x_n) \rightarrow \phi(x^*) = x^{\triangle}$$

ma questo contraddice l'ipotesi che ϕ converge a x^*. □

Observation 1.2.1. *La prova precedente è stata scritta da me però rivedendola ad un ricevimento con la professoressa Sestini, abbiamo visto che richiede che* ϕ *sia monotona. Questa richiesta è più restrittiva rispetto agli argomenti trattati nel testo. Una prova più semplice è la seguente:*

Dimostrazione. Sia ϕ continua, per l'equazione (1.2) del testo, vale $x_n \rightarrow x^*$ e $x_{n+1} \rightarrow x^*$. Per la definizione del metodo iterativo si ha:

$$x_{n+1} = \phi(x_n)$$

Dato che ϕ si assume continua e per l'equazione (1.2) allora vale

$$\begin{array}{c} x_{n+1} = \phi(x_n) \\ \downarrow \quad \downarrow \\ x^* = \phi(x^*) \end{array}$$

E questo dimostra che se il metodo è convergente allora soddisfa la condizione necessaria di consistenza. □

Observation 1.2.2. *L'ultima frase della precedente prova è importante, ovvero se un metodo converge alla soluzione esatta x^* allora soddisfa la condizione necessaria, non è vero il contrario. Infatti un metodo potrebbe soddisfare la condizione necessaria alla convergenza ma non convergere alla soluzione esatta.*

Exercise 1.2.2. *Considerare il metodo iterativo:*

$$x_{n+1} = \phi(x_n) = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right), \quad x_0 = 2$$

dimostrare che il metodo genera una successione di approssimazioni tale che $x_n \to \sqrt{2}$.

Dimostrazione. Questo metodo è a passo *singolo* in quanto usa un solo innesto per calcolare x_{n+1}. Affinchè il metodo iterativo sia convergente, per l'esercizio 1.2.1 posso costruire un'equazione per trovare il punto fisso di ϕ:

$$\phi(x) = \frac{1}{2}\left(x + \frac{2}{x}\right) = x$$

Manipolando: $x + \frac{2}{x} = 2x \Rightarrow x^2 + 2 = 2x^2 \Rightarrow 2 = x^2$. □

Exercise 1.2.3 (1.4)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Eseguendo il codice riportato nella sezione Metodo iterativo, questo è l'output di Octave sulla mia macchina:

```
octave:8> iterative(2, 1.5)
next =  1.42857142857143e+00
Difference with last step:0.0714285714285714
next =  1.41463414634146e+00
Difference with last step:0.0139372822299655
next =  1.41421568627451e+00
Difference with last step:0.000418460066953008
next =  1.41421356268887e+00
Difference with last step:2.12358564088966e-06
next =  1.41421356237310e+00
Difference with last step:3.15774073555986e-10
Difference with sqrt(2):0
```

In 5 passi si raggiunge la precisione richiesta, uno in più rispetto ai risultati mostrati in *Tabella 1.1* del libro.

1.3 ROUND-OFF

Exercise 1.3.1 (1.5)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Scrivo le rappresentazioni in formato stringa:

$$\alpha_0\alpha_1\ldots\alpha_{15} = 0\underbrace{1\ldots1}_{13}01 \to 32765$$
$$\alpha_0\alpha_1\ldots\alpha_{15} = 0\underbrace{1\ldots1}_{13}10 \to 32766$$
$$\alpha_0\alpha_1\ldots\alpha_{15} = 0\underbrace{1\ldots1}_{13}11 \to 32767$$
$$\alpha_0\alpha_1\ldots\alpha_{15} = 1\underbrace{0\ldots0}_{15} \to -32768$$

Ad ogni passo incremento di uno, questo comporta che incrementando la rappresentazione di 32767 si riporta sul bit del segno α_0. Per la definizione della funzione di valutazione ottengo $val(1\underbrace{0\ldots0}_{15}) = -2^{15} = -32768$.

Exercise 1.3.2 (1.6). *$\mathcal{M}$ ha un numero finito di elementi.*

Dimostrazione. Per far vedere che $\mathcal{M}$ ha un numero finito di elementi posso far vedere che vale $equinumerous(\mathcal{M},\{1,\ldots,n\})$, con $n \in \mathbb{N}$. Per la definizione della relazione $equinumerous$ posso costruire una funzione f *one-to-one, onto* tale che $f : \mathcal{M} \to \mathbb{N}$.

Considerare la rappresentazione in formato stringa $x = \alpha_0\alpha_1\ldots\alpha_m\beta_1\ldots\beta_s$, $\forall x \in \mathcal{M}$.

Per ogni rappresentazione costruisco una nuova rappresentazione che astrae da α e β, ovvero
$x = \alpha_0\alpha_1\ldots\alpha_m\beta_1\ldots\beta_s = \delta_0\ldots\delta_{m+s} = x'$ con $\alpha_0 = \delta_0,\ldots,\alpha_m = \delta_m,\beta_1 = \delta_{m+1},\ldots,\beta_s = \delta_{m+s}$.

Questa rappresentazione usa $m+s+1$ simboli, quindi scelta una base b si possono avere al massimo b^{m+s+1} configurazioni. Per la definizione di $\mathcal{M}$ in cui si aggiunge anche lo zero si nota che con la costruzione data in questa prova lo zero è gia considerato (considerando la rappresentazione con tutti i simboli $\delta_i = 0$).

□

Exercise 1.3.3 (1.6). $r_1 = b^{-\nu} \wedge r_2 = (1-b^{-m})b^{\varphi}$, *con* $\varphi = b^s - \nu$.

Suppongo che x sia normalizzato.

- La configurazione che rappresenta il minimo numero in valore assoluto (non considero il simbolo α_0 del segno) è:

$$r_1 = \alpha_1.\alpha_2\ldots\alpha_m\beta_1\ldots\beta_s = 1.\underbrace{0\ldots0}_{m-1}\underbrace{0\ldots0}_{s}$$

  A cui corrisponde la valutazione $val(r_1) = b^{-\nu}$.

- La configurazione che rappresenta il massimo numero in valore assoluto (non considero il simbolo α_0 del segno) è:

$$r_2 = \alpha_1.\alpha_2 \dots \alpha_m \beta_1 \dots \beta_s = (b-1).\underbrace{(b-1)}_{m-1}\underbrace{(b-1)}_{s}$$

Calcolo adesso la valutazione di $r_2 = \rho b^{e-\nu}$. Ricavo e:

$$e = (b-1)\sum_{i=1}^{s} b^{s-i} = (b-1)\sum_{i=0}^{s-1} b^i = (b-1)\frac{b^s-1}{b-1} = b^s - 1$$

Ricavo r_2:

$$\begin{aligned}
r_2 &= \left((b-1)\sum_{i=1}^{m} b^{1-i}\right) b^{b^s-1-\nu} = \left((b-1)\sum_{i=1}^{m} b^{-(i-1)}\right) b^{b^s-1-\nu} = \\
&= \left((b-1)\sum_{i=1}^{m} \left(\frac{1}{b}\right)^{i-1}\right) b^{b^s-1-\nu} = \left((b-1)\frac{\left(\frac{1}{b}\right)^m - 1}{\left(\frac{1}{b}\right)-1}\right) b^{b^s-1-\nu} = \\
&= \left((b-1)\frac{b\left(b^{-m}-1\right)}{b\left(\left(\frac{1}{b}\right)-1\right)}\right) b^{b^s-1-\nu} = \left((b-1)\frac{b\left(b^{-m}-1\right)}{1-b}\right) b^{b^s-1-\nu} = \\
&= \left((b-1)\frac{b\left(1-b^{-m}\right)}{b-1}\right) b^{b^s-1-\nu} = \left(b\left(1-b^{-m}\right)\right) b^{b^s-1-\nu} = \left(1-b^{-m}\right) b^{b^s-\nu}
\end{aligned}$$

Exercise 1.3.4. *Dimostrare che vale* $|\mathcal{M}| = b^m(exp_{max} - exp_{min} + 1) + 1$.

Dimostrazione. Usando i risultati dell'esercizio precedente posso riscrivere:

$$\begin{aligned}
exp_{max} &= b^s - 1 - \nu, \quad exp_{min} = -\nu \\
|\mathcal{M}| &= b^m(b^s - 1 - \nu + \nu + 1) + 1 = b^{m+s} + 1
\end{aligned}$$

Cerco adesso di dare una spiegazione meno formale, fissando come parametri per la rappresentazione che uso in questo piccolo esempio $b = 2, m = 3, s = 2, \nu = 2$, con mantisse normalizzate (implica $\alpha_1 = 1$). Segue quindi $exp_{max} = 1, exp_{min} = -2$.

Per quanto riguarda le mantisse che posso rappresentare sono

$$mantisse = \{1.00, 1.01, 1.10, 1.11\}$$

Per quanto riguarda i possibili esponenti, ottenuti partendo da exp_{min} sommando uno fino a exp_{max} (sommo uno per la definizione dell'esponente per la rappresentazione di numeri reali $\eta = \sum_{i=1}^{s} \beta_i b^{s-i} - \nu$):

$$esponenti = \left\{2^{-2} = \frac{1}{4}, 2^{-1} = \frac{1}{2}, 2^0 = 1, 2^1 = 2\right\}$$

Per cui posso rappresentare $|mantisse \times esponenti| = 16$ numeri positivi, a cui vanno aggiunti i 16 opposti e lo zero, per un totale di $33 = 2^5 + 1 = 2^3(1 - (-2) + 1) + 1 = 2^{3+2} + 1 = |\mathcal{M}|$. □

Exercise 1.3.5 (1.8). *Per il testo dell'esercizio consultare il libro di testo.*

Dato che rappresento mediante arrotondamento:

$$u = \frac{1}{2}b^{1-m} \quad , b = 10$$
$$\log_{10} u = \log_{10}\left(\frac{1}{2}10^{1-m}\right) = -\log_{10} 2 + (1-m)$$
$$m = 1 - \log_{10} 2 - \log_{10} u \quad , u = 4.66 \times 10^{-10}$$

```
octave:10> 1 - log10(2) - log10(4.66e-10)
ans =  1.00305840876460e+01
```

Observation 1.3.1. *Moltiplicare e dividere per la base* b *di lavoro è ad errore zero se c'è posto sufficiente nell'esponente.*

Cerco di spiegare il significato della precedente osservazione con un esempio. Sia $n = 1.1111 \times b^{-v}$, il quale ha il minimo esponente rappresentabile. Se moltiplico e successivamente divido per 10^{-4} devo dapprima denormalizzare, in quanto non posso rappresentare $b^{-(v+4)}$, ottengo quindi:

$$\frac{10^{-4} \times n}{10^{-4}} = \frac{0.0001 \times b^{-v}}{10^{-4}} = 10^4 \times 0.0001 \times b^{-v} = 1.0000 \times b^{-v}$$

Perdendo informazione sulle 4 cifre decimali di n.

Exercise 1.3.6 (1.9). *Per il testo dell'esercizio consultare il libro di testo.*

Formalizzo la richiesta: $-\log_{10} u = r$, con r uguale al numero di cifre esattamente rappresentate nella mantissa. Per definizione di u segue che $u > 0$, questo mi permette di non considerare i valori assoluti.

Distinguo i due casi, relativamente al metodo di rappresentazione usato:

PER TRONCAMENTO

$$u = b^{1-m}$$
$$-\log_{10} b^{1-m} = r$$
$$(m-1)\log_{10} b = r \Rightarrow \{\text{fix } b = 10\}$$
$$\Rightarrow m = 1 - \log_{10} u$$

PER ARROTONDAMENTO

$$u = \frac{b^{1-m}}{2}$$
$$-\log_{10}\frac{b^{1-m}}{2} = r$$
$$-\left(-\log_{10} 2 + (1-m)\log_{10} b\right) = r$$
$$\log_{10} 2 + (m-1)\log_{10} b = r \Rightarrow \{\text{fix } b = 10\}$$
$$\Rightarrow m = 1 - (\log_{10} u + \log_{10} 2) = 1 - (\log_{10} 2u)$$

Exercise 1.3.7 (1.10). *Per il testo dell'esercizio consultare il libro di testo.*

1. per il *più grande numero di macchina* x considero la mantissa normalizzata, quindi per la definizione dello standard IEEE754 valgono: il massimo esponente che posso usare $e = 2046$, $\nu = 1023$. La mantissa massima che posso fissare è $\alpha_1.\alpha_2 \dots \alpha_m$: sempre per la definizione dello standard segue che $m = 53$ e preso x normalizzato, segue $\alpha_1 = 1$. Ottengo la rappresentazione $mantissa(x) = 1.\underbrace{1 \dots 1}_{52}$, con:

$$\rho = \sum_{i=1}^{m} b^{1-i} = \sum_{i=0}^{m-1} b^{-i} = \frac{b\,(b^{-m}-1)}{1-b} = \frac{b^{1-m}-b}{1-b} =$$
$$= \{b = 2 \text{ by IEEE754 definition}\} = -2(2^{-m}-1) = 2(1-2^{-m})$$

$$val(x) = \rho \times b^{2046-1023} = 2(1-2^{-53})2^{1023}$$

```
octave:18> 2*(1-2^-53)*2^1023
ans =  1.79769313486232e+308
octave:19> realmax
ans =  1.79769313486232e+308
```

2. per il *più piccolo numero di macchina* x considero la mantissa normalizzata, quindi per la definizione dello standard IEEE754 valgono: il minimo esponente che posso usare $e = 1$, $\nu = 1023$. La mantissa minima che posso fissare è $\alpha_1.\alpha_2 \dots \alpha_m$: sempre per la definizione dello standard segue che $m = 53$ e preso x normalizzato, segue $\alpha_1 = 1$. Ottengo la rappresentazione $mantissa(x) = 1.\underbrace{0 \dots 0}_{52}$, con:

$$val(x) = b^0 \times b^{1-1023} = 2^{-1022}$$

```
octave:4> 2^-1022
ans =  2.22507385850720e-308
octave:5> realmin
ans =  2.22507385850720e-308
```

3. per il *più piccolo numero di macchina denormalizzato* x considero la mantissa denormalizzata, quindi per la definizione dello standard IEEE754 valgono: il minimo esponente che posso usare $e = 0$, $\nu = 1022$. La mantissa minima che posso fissare è $\alpha_1.\alpha_2 \dots \alpha_m$: sempre per la definizione dello standard segue che $m = 53$ e preso x denormalizzato, segue $\alpha_1 = 0$. Ottengo la rappresentazione $mantissa(x) = 0.\underbrace{0 \dots 0}_{51}1$, con:

$$val(x) = b^{-52} \times b^{-1022} = 2^{-1074}$$

```
octave:10> 2^-1074
ans =  4.94065645841247e-324
```

4. per la *precisione di macchina* considero la definizione dello standard IEEE754; segue quindi $m = 53$ e $b = 2$.

$$u = \frac{b^{1-m}}{2} = \frac{2^{1-53}}{2} = 2^{-53}$$

```
octave:27> 2^-53
ans =  1.11022302462516e-16
octave:28> eps
ans =  2.22044604925031e-16
```

Exercise 1.3.8 (1.11). *Spiegare il non funzionamento delle seguenti istruzioni:*

```
x = 0;
delta = 0.1;
while x ~= 1
  x = x + delta
end
```

Questo programma non termina, produce un ciclo infinito. Si ha questo comportamento perchè non è possibile rappresentare correttamente il numero 0.1 in macchina. Scrivo la sua rappresentazione in base 2:

$$0.1 \times 2 = 0.2$$
$$0.2 \times 2 = 0.4$$
$$0.4 \times 2 = 0.8$$
$$0.8 \times 2 = 1.6$$
$$0.6 \times 2 = 1.2$$
$$0.2 \times 2 = 0.4$$

Quindi $\texttt{delta} = (0.1)_{10} = (0.\overline{00011})_2$, posso normalizzare e ottengo $\texttt{delta} = 1.1\underbrace{00110011\ldots}_{51} \times 2^{-4}$.

Sommando `delta` ripetutamente ad `x` non sarà possibile raggiungere l'uguaglianza $x = 1$, quindi la guardia del while sarà sempre vera.

Una possibile soluzione è di irrobustire la guardia con $x \leqslant 1$, produce una computazione finita, ma non esegue lo stesso numero di passi della versione originale del problema.

Exercise 1.3.9 (1.12). *Per il testo dell'esercizio consultare il libro di testo.*

Calcolare $\sqrt{x^2+y^2}$ usando l'uguaglianza dell'aritmetica classica $x^2 = x \times x$ non è una buona strategia in quanto potrebbe non produrre un output valido ($\forall x \geqslant \frac{realmax}{.9\times10^{155}} = \frac{1.8\times10^{308}}{.9\times10^{155}}, x \times x = \infty$) in aritmetica finita. Questo lo dimostra il seguente codice:

```
octave:23> x = .9e155
x =  9.00000000000000e+154
octave:24> x*x
ans = Inf
```

Posso quindi riformulare il problema introducendo una variabile m tale che:

$$m = max\{|x|, |y|\}$$
$$\sqrt{x^2+y^2} = m\sqrt{\left|\frac{x}{m}\right|^2 + \left|\frac{y}{m}\right|^2}$$

Posso osservare che:

$$\sqrt{x^2+y^2} = \begin{cases} |x|\sqrt{1+\left|\frac{y}{m}\right|^2} & \text{se } m = |x| \\ |y|\sqrt{1+\left|\frac{x}{m}\right|^2} & \text{se } m = |y| \end{cases}$$

Lo schema che posso costruire da questo esempio è di utilizzare una buona approssimazione ($\sqrt{1+\ldots}$) sommando poi un errore piccolo (il termine quadratico). Questo aspetto verrà approfondito nelle osservazioni successive.

Exercise 1.3.10 (1.13). *Implementare i seguenti programmi:*

$$\left(\left(\frac{eps}{2}+1\right)-1\right)\left(\frac{2}{eps}\right)$$
$$\left(\frac{eps}{2}+(1-1)\right)\left(\frac{2}{eps}\right)$$

Questo l'help della funzione *eps*:

```
1 octave:31> help eps
  'eps' is a built-in function

     Return a scalar, matrix or N-dimensional array whose
        elements are
     all eps, the machine precision.  More precisely, 'eps
        ' is the
6    relative spacing between any two adjacent numbers in
        the machine's
     floating point system.  This number is obviously
        system dependent.
```

```
    On machines that support IEEE floating point
       arithmetic, 'eps' is
    approximately 2.2204e-16 for double precision and
       1.1921e-07 for
    single precision.
```

Questo il codice che implementa:

```
octave:30> ((eps/2 + 1) - 1) * (2/eps)
ans =  0.00000000000000e+00
octave:31> (eps/2 + (1 - 1)) * (2/eps)
ans =  1.00000000000000e+00
```

Perchè:

```
octave:46> 2/eps
ans =  9.00719925474099e+15
octave:47> eps/2
ans =  1.11022302462516e-16
octave:48> eps/2 + 1
ans =  1.00000000000000e+00
```

Exercise 1.3.11 (1.14). *Per il testo dell'esercizio consultare il libro di testo.*

Questo il codice che implementa:

```
octave:50> (1e300-1e300)*1e300
ans =  0.00000000000000e+00
octave:51> (1e300*1e300)-(1e300*1e300)
ans = NaN
```

Perchè:

```
octave:52> (1e300*1e300)
ans = Inf
```

Osservo che *octave:50* riesce ad eseguire la computazione in quanto $1e300 \in \mathcal{M}$, mentre *octave:51* restituisce *NaN* perchè $1e300 \times 1e300 \notin \mathcal{M}$ (*overflow*) e operazioni che coinvolgono *Inf* restituiscono *NaN* per l'implementazione di Octave. Non vale la proprietà distributiva, riporto l'help di Octave:

```
octave:53> help NaN
'NaN' is a built-in function

    Return a scalar, matrix, or N-dimensional array whose
       elements are
```

```
    all equal to the IEEE symbol NaN (Not a Number). NaN is the
    result of operations which do not produce a well defined numerical
    result. Common operations which produce a NaN are arithmetic with
    infinity (Inf - Inf), zero divided by zero (0/0), and any
    operation involving another NaN value (5 + NaN).

    Note that NaN always compares not equal to NaN (NaN != NaN). This
    behavior is specified by the IEEE standard for floating point
    arithmetic. To find NaN values, use the 'isnan' function.
```

Exercise 1.3.12 (1.15). *Per il testo dell'esercizio consultare il libro di testo.*

In aritmetica finita quello che si chiede di calcolare è, usando l'equazione 1.18:

$$\begin{aligned} \mathrm{fl}(d) &= \mathrm{fl}(\mathrm{fl}(\mathrm{fl}(x) + \mathrm{fl}(y)) + \mathrm{fl}(z)) \\ \mathrm{fl}(e) &= \mathrm{fl}(\mathrm{fl}(x) + \mathrm{fl}(\mathrm{fl}(y) + \mathrm{fl}(z))) \end{aligned}$$

Usando il *Teorema 1.4* posso riscrivere la prima equazione:

$$\begin{aligned} \mathrm{fl}(d) &= \mathrm{fl}(\mathrm{fl}(x + x\varepsilon_x + y + y\varepsilon_y) + \mathrm{fl}(z)) = \\ &= \mathrm{fl}(x + x\varepsilon_x + y + y\varepsilon_y + x\varepsilon_w + x\varepsilon_x\varepsilon_w + y\varepsilon_w + y\varepsilon_y\varepsilon_w + \mathrm{fl}(z)) \end{aligned}$$

Posso non considerare i termini in cui compare un prodotto di due errori relativi $\varepsilon_1\varepsilon_2$, ottenendo:

$$\mathrm{fl}(d) = \mathrm{fl}(x + x\varepsilon_x + y + y\varepsilon_y + x\varepsilon_w + y\varepsilon_w + \mathrm{fl}(z))$$

Posso sviluppare il termine $\mathrm{fl}(z)$ e successivamente la funzione fl più esterna:

$$\begin{aligned} \mathrm{fl}(d) &= \mathrm{fl}(x + x\varepsilon_x + y + y\varepsilon_y + x\varepsilon_w + y\varepsilon_w + z + z\varepsilon_z) = \\ &= x + x\varepsilon_x + y + y\varepsilon_y + x\varepsilon_w + y\varepsilon_w + z + z\varepsilon_z + x\varepsilon_q + x\varepsilon_x\varepsilon_q + y\varepsilon_q + y\varepsilon_y\varepsilon_q + x\varepsilon_w\varepsilon_q + \\ &+ y\varepsilon_w\varepsilon_q + z\varepsilon_q + z\varepsilon_z\varepsilon_q \end{aligned}$$

Non considero i termini con $\varepsilon_1\varepsilon_2$ e sviluppo $\mathrm{fl}(d)$:

$$d + d\varepsilon_d = x + x\varepsilon_x + y + y\varepsilon_y + x\varepsilon_w + y\varepsilon_w + z + z\varepsilon_z + x\varepsilon_q + y\varepsilon_q + z\varepsilon_q$$

Per la definizione del problema $d = x + y + z$ (in aritmetica esatta questa uguaglianza vale, senza dover specificare le parentesi, perchè vale la proprietà associativa):

$$\varepsilon_d = \frac{x(\varepsilon_x + \varepsilon_w + \varepsilon_q) + y(\varepsilon_y + \varepsilon_w + \varepsilon_q) + z(\varepsilon_z + \varepsilon_q)}{x + y + z}$$

Con argomento simmetrico si dimostra la seconda equazione $fl(e) = fl(fl(x) + fl(fl(y) + fl(z)))$, ottenendo:

$$\varepsilon_d = \frac{x(\varepsilon_x + \varepsilon_q) + y(\varepsilon_y + \varepsilon_w + \varepsilon_q) + z(\varepsilon_z + \varepsilon_w + \varepsilon_q)}{x + y + z}$$

Fisso $\varepsilon_{max} = max\{|\varepsilon_x|, |\varepsilon_y|, |\varepsilon_w|, |\varepsilon_z|, |\varepsilon_q|\}$ posso quindi maggiorare ε_d:

$$|\varepsilon_d| \leqslant \frac{|x| + |y| + |z|}{|x + y + z|}\varepsilon_{max}$$

Osservo che se $xy > 0 \wedge yz > 0$ allora $\varepsilon_d \leqslant \varepsilon_{max}$ ed il problema è ben condizionato, altrimenti se $xy < 0 \vee yz < 0 \vee xz < 0 \vee (x + y + z) \to 0$ allora il problema è mal condizionato.

Exercise 1.3.13. *Riprendere l'esercizio precedente, siano* $a, b, c \in \mathcal{M}, b = 10, m = 4$, *con rappresentazione della mantissa mediante arrotondamento. Verificare le disuguaglianza:*

$$(a + b) + c \neq a + (b + c)$$

con $a = 2.000 \times 10^{-4}, b = 4.000 \times 10^{-4}, c = 7.000 \times 10^{0}$.

$$\begin{aligned} fl(fl(a + b) + c) &= fl(6.000 \times 10^{-4} + 7.000 \times 10^{0}) = \\ &= fl(0.0006 \times 10^{0} + 7.000 \times 10^{0}) = \\ &= fl(7.0006 \times 10^{0}) = 7.001 \times 10^{0} \end{aligned}$$

$$\begin{aligned} fl(a + fl(b + c)) &= fl(2.000 \times 10^{-4} + 7.000 \times 10^{0}) = \\ &= fl(0.0002 \times 10^{0} + 7.000 \times 10^{0}) = \\ &= fl(7.0002 \times 10^{0}) = 7.000 \times 10^{0} \end{aligned}$$

Nei precedenti passaggi è sbagliato calcolare $fl(a), fl(b), fl(c)$ in quanto, per ipotesi, a, b, c sono già un numeri di macchina (e quindi già affetti da errore).

Exercise 1.3.14 (1.16)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Formalmente devo studiare:

$$\tilde{y} = f(\tilde{x}), \qquad y = f(x) = \sqrt{x}$$

Posso riscrivere $\tilde{y}, \tilde{x}$ introducendo gli errori relativi e, per la definizione di errore relativo, vale:

$$y(1+\varepsilon_y) = f(x(1+\varepsilon_x))$$

La precedente relazione utilizza l'implementazione esatta di f, quindi considero lo sviluppo di f al primo ordine centrato in x:

$$\begin{aligned} y(1+\varepsilon_y) &= f(x) + f'(x)(x + x\varepsilon_x - x) + O(\varepsilon_x^2) \\ y\varepsilon_y &= f'(x)x\varepsilon_x + O(\varepsilon_x^2) \end{aligned}$$

Per le ipotesi del problema, usando quanto detto ad inizio prova, vale:

$$f(x) = x^{\frac{1}{2}} \qquad f'(x) = \frac{1}{2}x^{-\frac{1}{2}} = \frac{1}{2\sqrt{x}}$$

Posso approssimare:

$$|\varepsilon_y| \approx \left|\frac{1}{2\sqrt{x}}\frac{x}{y}\right||\varepsilon_x| = \left|\frac{1}{2\sqrt{x}}\frac{x}{\sqrt{x}}\right||\varepsilon_x| = \frac{|\varepsilon_x|}{2}$$

Ottengo $k = \frac{1}{2}$ come richiesto, la funzione $f(x) = \sqrt{x}$ è sempre ben condizionata, la prova è conclusa. □

Exercise 1.3.15 (1.17). *Per il testo dell'esercizio consultare il libro di testo.*

Per la definizione del problema vale:

$$\begin{aligned} x_1 = 0.12345678 \quad x_2 &= 0.12341234 \\ fl(x_1) = 1.235 \times 10^{-1} \quad fl(x_2) &= 1.234 \times 10^{-1} \\ y = x_1 - x_2 &= 0.00004444 = 4.444 \times 10^{-5} \\ fl(x_1) - fl(x_2) &= 0.001 \times 10^{-1} = 1.000 \times 10^{-4} \end{aligned}$$

Osservo che le due differenze, quella esatta con quella in aritmetica finita, non hanno nessuna cifra in comune, vengono perse tutte le cifre significative.

L'errore relativo che si commette sull'intera operazione è:

$$\varepsilon_y = \frac{4.444 \times 10^{-5} - 1.000 \times 10^{-4}}{4.444 \times 10^{-5}} = \frac{4.444 \times 10^{-5} - 10.000 \times 10^{-5}}{4.444 \times 10^{-5}}$$

```
1 octave:5> (4.444e-5 - 10e-5) / 4.444e-5
  ans = -1.2502
```

Se considero l'analisi del condizionamento ottengo:

$$\varepsilon_{x_1} = \frac{1.2345678 \times 10^{-1} - 1.235 \times 10^{-1}}{1.2345678 \times 10^{-1}}$$

$$\varepsilon_{x_2} = \frac{1.2341234 \times 10^{-1} - 1.234 \times 10^{-1}}{1.2341234 \times 10^{-1}}$$

```
octave:10> x1 = (1.2345678e-1 - 1.235e-1) / 1.2345678e-1
x1 = -3.50082028706699e-04
octave:11> x2 = (1.2341234e-1 - 1.234e-1) / 1.2341234e-1
x2 =  9.99900009998966e-05
octave:17> epsilon_Max = max(abs(x1),abs(x2))
epsilon_Max =  3.50082028706699e-04
```

$$\varepsilon_{max} = \max\{|\varepsilon_{x_1}|, |\varepsilon_{x_2}|\} = |\varepsilon_{x_1}| = 3.50082028706699 \times 10^{-4}$$

$$|\varepsilon_y| \leqslant \frac{|1.2345678 \times 10^{-1}| + |1.2341234 \times 10^{-1}|}{|1.2345678 \times 10^{-1} - 1.2341234 \times 10^{-1}|}\varepsilon_{max}$$

```
octave:15> ((abs(1.2345678e-1)+abs(1.2341234e-1))/abs
   (1.2345678e-1 - 1.2341234e-1))*max(abs(x1),abs(x2))
ans =  1.94474442742179e+00
```

Torna infatti la stima dell'errore fatta considerando l'errore sull'intera operazione $|\varepsilon_y| = 1.2502$ con quella fatta considerando il condizionamento sui dati in ingresso$|\varepsilon_y| \leqslant 1.94474442742179$.

Exercise 1.3.16 (1.18)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Implemento:

```
octave:1> format long e
octave:2> a = 0.1
a =  1.000000000000000e-01
octave:3> b = 0.099999999999
b =  9.999999999900000e-02
octave:4> a-b
ans =  1.000000563385549e-12
```

Eseguo lo studio sul condizionamento. Per il *Teorema 1.4* il massimo errore che posso commettere usando il metodo di arrotondamento è $u = \frac{1}{2}b^{1-m}$. Lo standard IEEE754 fissa $b = 2, m = 53$, quindi $u = 2^{-53}$. Questo errore è comune per entrambi in quanto non devo applicare la funzione fl ai due numeri a, b (in quanto sono già rappresentati in macchina commento il massimo errore u), quindi $\varepsilon_{max} = \varepsilon_a = \varepsilon_b = 2^{-53} = 1.11022302462516 \times 10^{-16}$.

Per l'equazione 1.23 vale:

$$\varepsilon_y \leqslant \frac{|1.000000000000000 \times 10^{-1}| + |9.999999999900000 \times 10^{-2}|}{|1.000000000000000 \times 10^{-1} - 9.999999999900000 \times 10^{-2}|}\varepsilon_{max}$$

$$\varepsilon_y \leqslant \frac{1.999999999999000 \times 10^{-1}}{1.000000563385549 \times 10^{-12}}\varepsilon_{max} = (1.999988732342 49 \times 10^{11})\varepsilon_{max} =$$

$$= 2.22043353963751 \times 10^{-5}$$

Si osserva che $k \in O(10^{11})$, il problema è mal condizionato.

Queste le giustificazioni dei valori precedenti:

```
octave:15> epsilon_Max = 2^-53
epsilon_Max =  1.11022302462516e-16
octave:16> a+b
ans =  1.99999999999000e-01
octave:17> abs(a-b)
ans =  1.00000563385549e-12
octave:18> (a+b)/(abs(a-b))
ans =  1.99998873234249e+11
octave:19> (a+b)/(abs(a-b))*epsilon_Max
ans =  2.22043353963751e-05
```

Calcolo adesso l'errore relativo, considerando come valore esatto la differenza $a - b = 1 \times 10^{-12}$:

$$\varepsilon_x = \frac{(1.00000563385549 \times 10^{-12} - 1 \times 10^{-12})}{1 \times 10^{-12}}$$

Ottengo:

```
octave:4> (1.00000563385549e-12 - 1e-12)/1e-12
ans =  5.63385549010602e-06
```

1.4 CONSIGLI PER OPERAZIONI DI MACCHINA

La seguente lista porta dei cosigli su come eseguire le operazioni di macchina a quando si deve implementare un problema formulato in aritmetica esatta.

1. *good approximation plus a small correction term* ecco alcuni esempi

$$a + \frac{b-a}{2} \quad \text{è migliore di} \quad \frac{a+b}{2} \quad \text{hint: } a - \frac{a}{2} = \frac{a}{2}$$

$$x - \frac{x^2 - a}{2x} \quad \text{è migliore di} \quad \frac{1}{2}\left(x + \frac{a}{x}\right) \quad \text{hint: } x - \frac{x}{2} = \frac{x}{2}$$

$$x_2 - y_2\frac{x_2 - x_1}{y_2 - y_1} \quad \text{è migliore di} \quad \frac{y_2x_1 - x_2y_1}{y_2 - y_1} \quad \text{hint: add } x_2 - x_2 \text{ and factor } -x_2$$

2. *add the small term first* quando devo sommare una collezione di valori alla quale appartengono valori relativamente piccoli è una buona idea ordinare i valori in modo decrescente ed iniziare a sommare dai valori più piccoli ai valori più grandi.

3. *be careful when substracting almost equals numbers* la differenza è una operazione di macchina mal condizionata, quindi deve essere utilizzata con

cautela. È utile nei termini di correzione (vedi due punti sopra) nella riformulazione di un problema, mentre è causa di cancellazione numerica se utilizzata nel calcolare informazioni essenziali come una prima approssimazione (vedi esercizio 1.3.16).

4. *avoid large partial result on the road to a small final answer* prendiamo come esempio la funzione $f(x) = e^x$ ed il suo sviluppo di MacLaurin $Ml(x)$. Se applico ad $x = -10$, ottengo $f(-10) = 4.54e-5, Ml(-10) \approx 2700$. Una soluzione potrebbe essere riformulare il problema, calcolando $Ml(10)$ al posto di $Ml(-10)$ e prendendo il reciproco.

5. *use mathematical reformulation to avoid 3. and 4.*

6. *series expansion can supplement 5.*

7. *use integer calculation when possible* come fatto nel metodo Metodo di bisezione, è migliore impostare un ciclo di dimensione finita e conosciuta al posto di utilizzare una guardia che deve essere ogni volta valutata.

1.4.1 *Spacing between machine numbers'*

La spaziatura fra due coppie di numeri di macchina non è sempre la stessa, ma varia per ogni $b^{e_{min}+i}$ con $i = 0, \ldots, e_{max} - e_{min}$. La spaziatura fra numeri piccoli in valore assoluto (con $i \to 0$) è più raffinata rispetto alla spaziatura fra numeri grandi in valore assoluto (con $i \to e_{max} - e_{min}$).

2

RADICI DI UNA EQUAZIONE

2.1 METODI ITERATIVI

Exercise 2.1.1 (2.2). *Per il testo dell'esercizio consultare il libro di testo.*

Il metodo da trovare deve convergere a $x^* = \sqrt[m]{a}$ che, per l'esercizio 1.2.1, deve verificare la condizione di consistenza:

$$x^* = \phi(x^*), \qquad \phi(x^*) = x^* - \frac{f(x^*)}{f'(x^*)}$$

Implementando con ϕ il metodo di Newton che si chiede di usare. Unendo le due uguaglianze sopra si ottiene:

$$\begin{aligned} x^* &= x^* - \frac{f(x^*)}{f'(x^*)} \\ 0 &= f(x^*) \end{aligned} \tag{2.1}$$

La (2.1) mi da un vincolo per la ricerca della f (questo vale per qualsiasi f!). Posso adesso sfruttare la richiesta di convergenza:

$$\begin{aligned} x &= \sqrt[m]{a} \\ f(x) &= x^m - a = 0 \end{aligned} \tag{2.2}$$

Ho quindi costruito una funzione che rispetta il vincolo espresso nella (2.1). Scrivo il metodo iterativo:

$$\begin{aligned} x_{i+1} &= \phi(x_i) \\ \phi(x_i) &= x_i - \frac{x_i^m - a}{m x_i^{m-1}} = \frac{m x_i^m - x_i^m + a}{m x_i^{m-1}} = \\ &= \frac{(m-1)x_i^m + a}{m x_i^{m-1}} = \frac{m-1}{m} x_i + \frac{a}{m x_i^{m-1}} \end{aligned}$$

Verifico adesso che il metodo rispetti la condizione necessaria per la convergenza.

Dimostrazione. Per l'esercizio 1.2.1 vale:

$$\begin{aligned} x &= \frac{m-1}{m} x + \frac{a}{m x^{m-1}} \\ m x^m &= (m-1)x^m + a \\ x^m(m+1-m) &= a \\ x^m &= a \end{aligned} \tag{2.3}$$

Il metodo ϕ soddisfa la condizione necessaria, converge al valore richiesto e questo termina la prova. □

Exercise 2.1.2 (2.3)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Per la definizione del metodo delle secanti vale:

$$x_{i+1} = \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})}$$

Come per l'esercizio precedente, devo verificare che il metodo soddisfi la condizione necessaria di convergenza [1]. La verifica fornisce un vincolo per la funzione f da ricercare: $f(x) = 0$, quindi agisco come nell'esercizio precedente, considerando la richiesta di convergenza:

$$\begin{aligned} x &= \sqrt{a} \\ f(x) = x^2 - a &= 0 \end{aligned} \tag{2.4}$$

Posso adesso sostituire nella definizione del metodo:

$$x_{i+1} = \frac{(x_i^2 - a)x_{i-1} - (x_{i-1}^2 - a)x_i}{x_i^2 - a - (x_{i-1}^2 - a)} = \frac{x_i^2 x_{i-1} - a x_{i-1} - x_{i-1}^2 x_i + a x_i}{x_i^2 - x_{i-1}^2} =$$
$$= \frac{a(x_i - x_{i-1}) + x_i x_{i-1}(x_i - x_{i-1})}{(x_i - x_{i-1})(x_i + x_{i-1})} = \frac{(x_i - x_{i-1})(a + x_i x_{i-1})}{(x_i - x_{i-1})(x_i + x_{i-1})} = \frac{a + x_i x_{i-1}}{x_i + x_{i-1}}$$

Verifico che il metodo caratterizzato soddisfi la condizione necessaria di convergenza.

Dimostrazione. Per l'esercizio 1.2.1 vale:

$$\lim_{i\to\infty} x_i = x^* = \sqrt{a} \quad \Rightarrow \quad x_{i+1} = x_i = x_{i-1} = x, \quad \text{segue che}$$
$$x = \frac{a + x^2}{2x}$$
$$2x^2 = a + x^2$$
$$x^2 = a$$

E questo termina la prova. □

Comparando con l'*Esercizio 1.4* del libro di testo si vede che il metodo proposto in tale esercizio è la caratterizzazione del metodo delle secanti qui discusso con la caratterizzazione di $a = 2$:

$$x_{i+1} = \frac{2 + x_i x_{i-1}}{x_i + x_{i-1}}$$

1 qui mi è difficile far vedere che vale la condizione necessaria, in quanto si annulla il denominatore

Observation 2.1.1 (Sul teorema di punto fisso)**.** *Se parto da due punti nell'intervallo* $x, y \in I = (x^* - \delta, x^* + \delta)$*, allora applicando il metodo iterativo* ϕ *ai due punti ottengo una nuova coppia* $\phi(x) = x_1, \phi(y) = y_2$*, la cui distanza* $|x_1 - y_1|$ *è strettamente minore della distanza dei punti di partenza in quanto* $L < 1$ *per ipotesi del teorema: ottengo quindi* $|x_1 - y_1| < |x - y|$*. In questo modo sono riuscito ad ottenere una nuova coppia di punti più vicina.*

Questo è quello che avrei voluto in quanto una volta applicato il metodo ad una coppia, ottengo una coppia di punti più vicina: se riesco ad applicare nuovamente il metodo alla coppia appena costruita riuscirei ad ottenere una nuova coppia di punti più vicini rispetto alla ultima coppia generata. Ripetendo un numero sufficiente di volte il metodo ϕ *riesco a diminuire la distanza tra i punti delle coppie, fino a far degenerare una coppia di punti distinti allo stesso punto* (x_k, x_k)*, ovvero al punto fisso del metodo, e quindi convergere alla soluzione.*

Importante quindi è partire da una coppia di punti che appartengono all'intervallo I *per poter applicare almeno una volta il metodo.*

Posso commentare le implicazioni del teorema:

1. x^* *è l'unico punto fisso di* ϕ *in* I*: questo permette al metodo di non oscillare tra due punti fissi (soluzioni) e inoltre di non creare non determinismo nella scelta della soluzione a cui converge il metodo*

2. *se* $x_0 \in I \Rightarrow \phi(x_{i-1}) = x_i \in I, \forall i = \{1, 2, 3, \ldots\}$*: questo è molto importante perchè assicura che ogni punto generato dal metodo* ϕ *appartiene all'intervallo se il punto di innesco* $x_0 \in I$*. In questo modo, tornando a ragionare per coppie come fatto nei paragrafi precedenti di questa osservazione, se il punto di innesco appartiene all'intervallo* I *allora tutti i nuovi punti che posso costruire con il metodo* ϕ *apparterranno anch'essi all'intervallo e quindi posso ragionare a coppie* (x_{i-1}, x_i) *e applicare il teorema ad ogni coppia per* $i \in \{1, 2, \ldots\}$*, ottenendo quindi che i punti della coppia* (x_i, x_{i+1}) *saranno più vicini dei punti della coppia* (x_{i-1}, x_i)*.*

   Dimostrazione. Per induzione su i:

   BASE se $x_0 \notin I$ allora la tesi è vera (vacously true)

   HP INDUTTIVA suppungo vero che $x_k = \phi(x_{k-1}) \in I, \forall k \in \{1, \ldots i\}$

   PASSO INDUTTIVO dimostro per $k = i + 1$:

   $$|x_{k+1} - x^*| = |\phi(x_k) - \phi(x^*)| \leqslant l|x_k - x^*|$$

   Per ipotesi induttiva $x_k, x^* \in I$ questo implica che $l|x_k - x^*| < \delta$. Per la proprietà transitiva della relazione $\leqslant$ vale $|x_{k+1} - x^*| < \delta$, ovvero $x_{k+1} \in I$ e questo termina il passo induttivo.

   □

3. $\lim_{i\to\infty} x_i = x^*$, *ovvero il metodo converge alla soluzione* x^* *punto fisso di* ϕ *per la condizione necessaria.*

   Nella prova riportata nel libro la regoletta per impostare l'indice del termine l *è questa:*

$$l^k|x_j - x^*|, j = i - a \quad \Rightarrow \quad k = i - j$$

Exercise 2.1.3. *Dire se il seguente metodo iterativo è convergente e se si a quale punto converge.*

$$x_{k+1} = \sqrt{x_k + 1}$$

Formalizzo il metodo con la funzione ϕ:

$$\phi(x) = \sqrt{x+1}$$

Se il metodo converge, per la condizione necessaria, converge al suo punto fisso, quindi lo determino:

$$x = \phi(x) = \sqrt{x+1}$$
$$x^2 = x + 1$$

Implica che le due soluzioni sono $x_{1,2} = \frac{1\pm\sqrt{5}}{2}$.

Non ho dimostrato che il metodo converge, ho solo trovato il punto di convergenza se il metodo converge. Dimostro adesso la convergenza controllando che siano vere le ipotesi del teorema di punto fisso:

$$\exists\delta > 0 \exists l, 0 \leqslant l < 1 : \forall x, y \in I = (x^* - \delta, x^* + \delta) :$$
$$|\phi(x) - \phi(y)| < l|x - y|$$

Dato che ϕ è derivabile allora sviluppo $\phi(y)$ in x con resto al primo ordine, per applicare il metodo due volte nello stesso punto:

$$T_{\phi(y)}(x) = \phi(x) + \phi'(\xi)(y - x), \quad \xi \in [x, y] (\Rightarrow \xi \in I)$$

Posso sostituire nella precedente disuguaglianza:

$$\begin{aligned} |\phi(x) - \phi(x) - \phi'(\xi)(y - x)| &< l|x - y| \\ |\phi'(\xi)(x - y)| &< l|x - y| \\ |\phi'(\xi)||x - y| &< l|x - y| \\ |\phi'(\xi)| &< l < 1 \end{aligned}$$

Devo quindi verificare:

$$|\phi'(\xi)| < 1$$
$$\phi'(x) = \frac{1}{2\sqrt{x+1}}$$
$$\frac{1}{2\sqrt{x+1}} < 1$$
$$1 < 2\sqrt{x+1}, \forall x \geqslant 0$$

Sono nelle ipotesi del teorema di punto fisso [2] quindi per l'implicazione del teorema valgono

- $x = \frac{1\pm\sqrt{5}}{2}$ è l'unico punto fisso di ϕ in I
- tutti i punti x_i generati dal metodo appartengono all'intervallo I
- il metodo converge a $x = \frac{1\pm\sqrt{5}}{2}$.

METODI CON RAPPRESENTAZIONE GRAFICA

I prossimi metodi che verranno descritti utilizzeranno lo strumento Octave per la rappresentazione grafica dei metodi per illustrare in modo più chiaro il comportamento del metodo iterativo in questione.

Tutti i grafici hanno uno schema comune, ovvero sono composti da tre curve:

- la curva cyan rappresenta la funzione $f(x)$ che si sta studiando
- la curva blu, composta da pochi simboli '+', rappresenta punti calcolati dal metodo per convergere allo zero x^*
- la curva rossa rappresenta il comportamento del metodo, visualizzando la sequenza con cui si procede per convergere alla soluzione

2.2 METODO DI BISEZIONE

Riporto il codice di pagina 23:

```
octave:12> p = poly([1.1*ones(1,20) pi])
p =
 Columns 1 through 6:
   1.0000e+00  -2.5142e+01   2.9902e+02  -2.2396e+03
      1.1860e+04  -4.7254e+04
 Columns 7 through 12:
   1.4711e+05  -3.6678e+05   7.4461e+05  -1.2444e+06
      1.7234e+06  -1.9847e+06
 Columns 13 through 18:
   1.9008e+06  -1.5096e+06   9.8794e+05  -5.2718e+05
      2.2572e+05  -7.5702e+04
 Columns 19 through 22:
   1.9159e+04  -3.4410e+03   3.9100e+02  -2.1135e+01
octave:13> polyval(p,pi)
ans =  2.0207e-04
```

[2] l'unica cosa che rimane da trovare è l

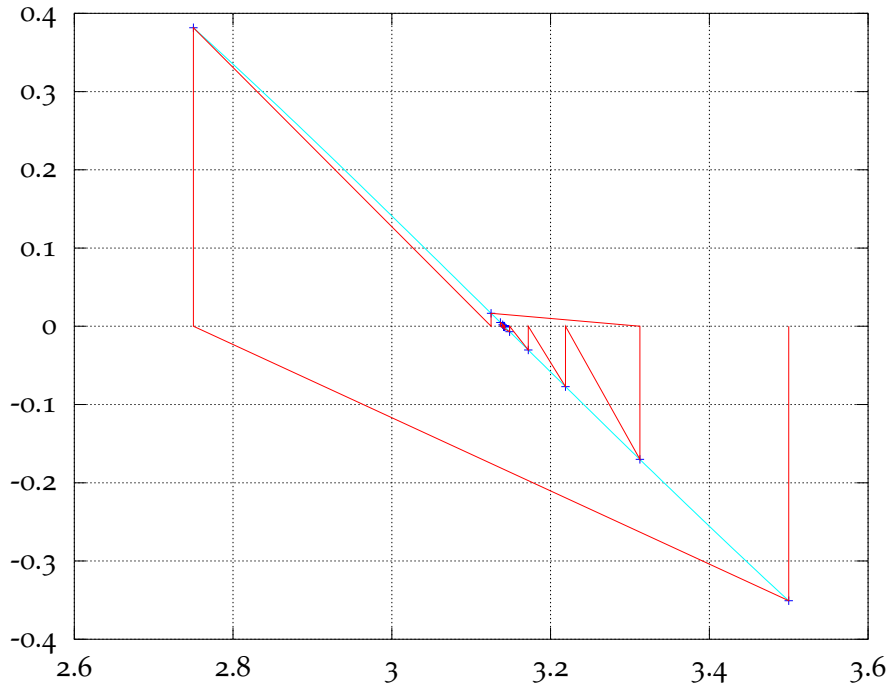
Lo scopo della funzione poly(r) è quello di creare un vettore di coefficienti di un polinomio p, tale che le radici di p appartengono al vettore r (poly : Root[] → PolynomialCoefficient[]). Valutando quindi il polinomio in una sua radice (*octave:13*) in aritmetica esatta dovrei ottenere 0, mentre in aritmetica finita non è vero ($p(\pi) = ans = 2.0207e-04 \neq 0$).

Exercise 2.2.1. *Implementare il metodo di bisezione ed applicarlo alla funzione* sin(x) *con intervallo iniziale* $[2,5]$ *ed una tollerenza* $tol_X = 10^{-14}$.

Per l'implementazione del codice vedere Metodo di bisezione.

```
octave:45> [x, i, imax, ascisse] = bisectionMethod('sin',
   2, 5, 1e-14)
x =  3.14159265358980e+00
i =  4.60000000000000e+01
imax =  4.90000000000000e+01
ascisse = [too long to report here]
octave:46> length(ascisse)
ans =  4.60000000000000e+01
octave:47> xsin = min(ascisse):0.01:max(ascisse)
octave:48> ysin = feval('sin', xsin)
octave:49> [prepX, prepY] =
   prepareForPlottingMethodSegments(ascisse, "sin", "")
octave:50> plot(xsin, ysin, "c", ascisse, feval('sin',
   ascisse), "b+", prepX,prepY, "r")
octave:51> grid
octave:52> print 'bisectionPlotOutput.tex' '-dTex' '-S800,
    600'
```

Si raggiunge la tolleranza richiesta in 46 passi, tre in meno delle iterazioni massime possibili. Questo l'output del comando *octave:52*:



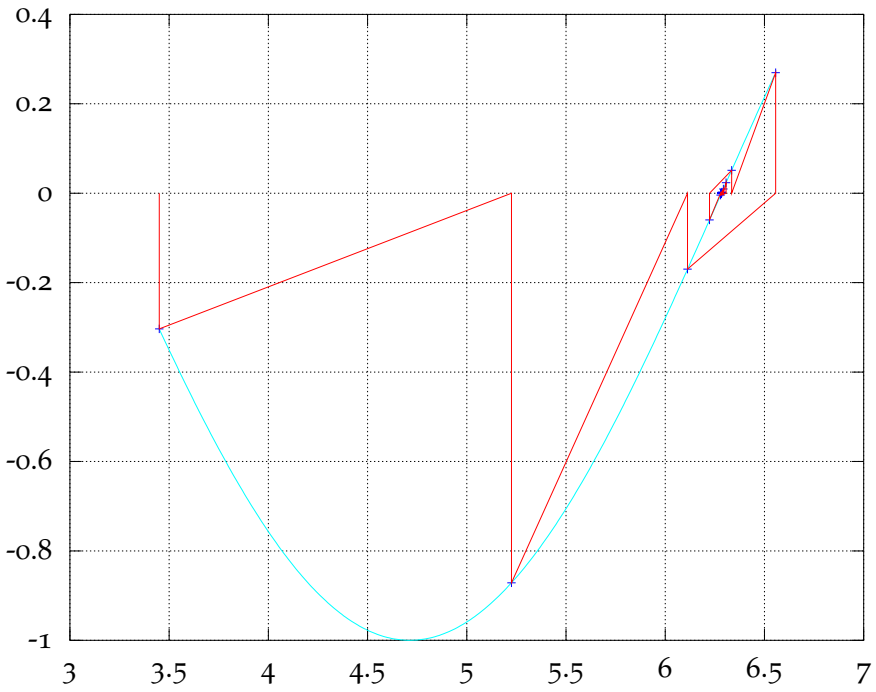
Exercise 2.2.2. *Implementare il metodo di bisezione ed applicarlo alla funzione* $\sin(x)$ *con intervallo iniziale* $[-0.1, 7]$*, in modo da avere due zeri nell' intervallo di confidenza, ed una tollerenza* $\mathrm{tol}_X = 10^{-14}$.

Per l'implementazione del codice vedere Metodo di bisezione.

```
octave:51> [x, i, imax, ascisse] = bisectionMethod('sin',
    -0.1, 7, 1e-14)
x =  6.28318530717958e+00
i =  4.70000000000000e+01
imax =  5.00000000000000e+01
ascisse = [too long to report here]
octave:52> length(ascisse)
ans =  4.70000000000000e+01
octave:53> xsin = min(ascisse):0.01:max(ascisse)
octave:54> ysin = feval('sin', xsin)
octave:55> [prepX, prepY] =
   prepareForPlottingMethodSegments(ascisse, "sin", "")
octave:56> plot(xsin, ysin, "c", ascisse, feval('sin',
   ascisse), "b+", prepX,prepY, "r")
octave:57> grid
```

```
octave:58> print 'bisectionWithTwoRootsPlotOutput.tex' '-
    dTex' '-S800, 600'
```

Si raggiunge la tolleranza richiesta in 47 passi, tre in meno delle iterazioni massime possibili. Questo l'output del comando *octave:58*:



Da questo esercizio si vede che il metodo di bisezione converge comunque ad una radice (in questa applicazione a 2π), anche nel caso in cui nell'intervallo di confidenza ci sono più zeri della funzione.

2.3 METODO DI NEWTON

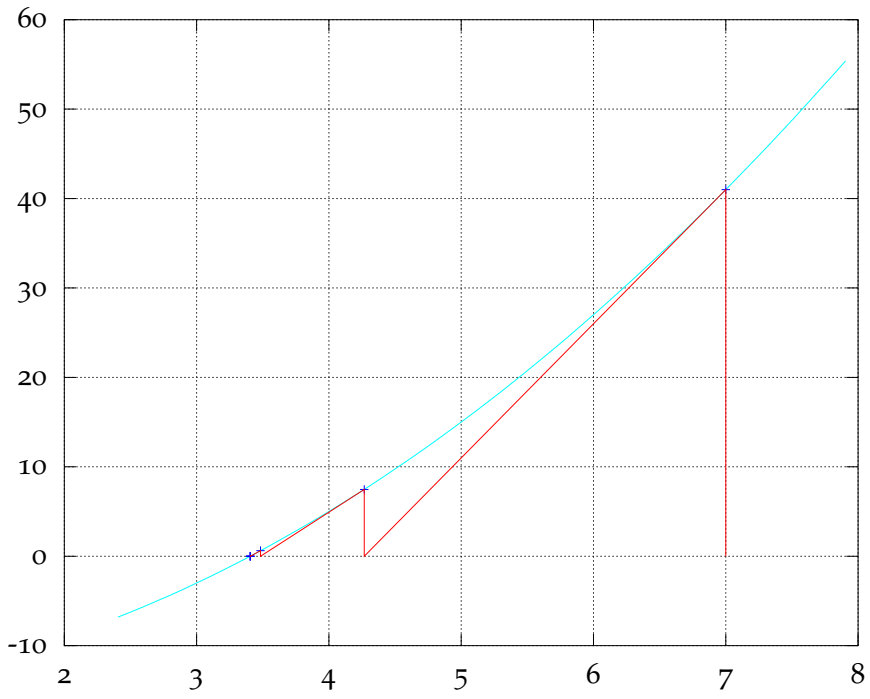
Exercise 2.3.1. *Implementare il metodo di newton ed applicarlo alla funzione* singleZero, *con innesco iniziale* $x_0 = 7$, *una tollerenza assoluta e relativa* $\mathrm{tol}_X = \mathrm{rTol}_X = 10^{-14}$ *ed un numero massimo di iterazioni* $i_{max} = 10^2$.

Per l'implementazione del codice vedere Metodo di Newton.

```
octave:112> [x, i, ascisse] = newtonMethod('singleZero','
    singleZeroDerivative',7, 1e2, 1e-14, 1e-14, '
    incrementCriterion')
x =  3.40512483795333e+00
```

```
i =  5.00000000000000e+00
ascisse = [too long to report here]
octave:113> xSingleZero = min(ascisse)-1:0.1:max(ascisse)
   +1
octave:114> ySingleZero = invokeDelegate('singleZero',
   xSingleZero)
octave:115> [prepX, prepY] =
   prepareForPlottingMethodSegments(ascisse, '
   invokeDelegate', 'singleZero')
octave:116> plot(xSingleZero, ySingleZero, "c", ascisse,
   invokeDelegate('singleZero', ascisse), "b+", prepX,
   prepY, "r")
octave:117> grid
octave:118> print 'newtonPlotOutput.tex' '-dTex' '-S800,
   600'
```

Si raggiunge la tolleranza richiesta in 5 passi. Questo l'output del comando *octave:118*:



Observation 2.3.1. *Per l'applicazione eseguita sopra ho utilizzato il criterio di arresto* incremento, *il metodo converge, ma posso studiare il condizionamento che affetta*

ogni valutazione della precisione richiesta. Questo il codice che dimostra questo condizionamento, comparando l'applicazione del metodo usando il criterio di arresto per residuo*:*

```
[x, i, incAscisse, incChecked] = newtonMethod('singleZero'
    ,'singleZeroDerivative', 7, 1e2, 10^(-14), 10^(-14), '
    incrementCriterion');
octave:138> i
i =  5.00000000000000e+00
octave:139> errors = errorMonitor(incAscisse)
errors =
    4.12195121951219e+00   9.88875669244497e+00
       8.93522817393261e+01   8.95110152113953e+03
       9.18666331414061e+07   7.66765947567831e+15
octave:140> incChecked
incChecked =
    2.73333333333333e+00   7.83682983682984e-01
       7.70978577615202e-02   7.60913136916397e-04
       7.41319183816813e-08   8.88178419700125e-16
octave:141> [x, i, resAscisse, resChecked] = newtonMethod(
    'singleZero','singleZeroDerivative', 7, 1e2, 10^(-14),
    10^(-14), 'residueCriterion');
octave:142> resChecked
resChecked =
    2.73333333333333e+00   7.83682983682984e-01
       7.70978577615201e-02   7.60913136916414e-04
       7.41319183470686e-08   6.82317562092805e-16
octave:143> i
i =  5.00000000000000e+00
```

Si osserva che i valori utilizzati nel controllo per il raggiungimento della precisione richiesta sono uguali tranne l'ultimo, il metodo che usa il criterio residueCriterion *è più accurato, in quanto non è affetto dalla cancellazione numerica (per l'ultima coppia di ascisse trovata il fattore di amplificazione è* $\mathrm{k} = 7.66765947567831\mathrm{e}+15$*).*

I due metodi utilizzano comunque lo stesso numero di passi per convergere alla soluzione.

Observation 2.3.2. *Posso effettuare una nuova comparazione: se chiedo una precisione di* $\mathrm{tol}_X = 1\mathrm{e}-16$*, si osserva che il metodo non converge se si utilizza il criterio di arresto per* incremento*, mentre converge usando il criterio per* residuo*. Questo il codice che dimostra quanto detto:*

```
octave:5> [x, i]=newtonMethod('singleZero','
    singleZeroDerivative', 7, 1e2, 10^(-16), 10^(-16),'
    incrementCriterion');
```

```
Il metodo non converge.
octave:6> [x, i]=newtonMethod('singleZero','
   singleZeroDerivative', 7, 1e2, 10^(-16), 10^(-16),'
   residueCriterion');
octave:7> i
i =  6
octave:8>
```

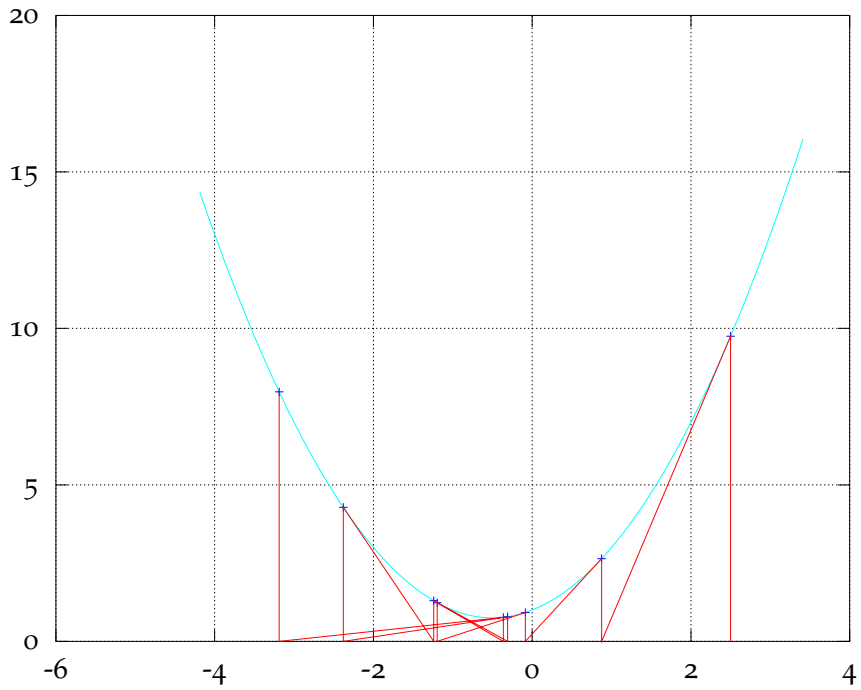
A parità di numero massimo di passi, il secondo criterio permette al metodo di convergere.

Exercise 2.3.2. *Implementare il metodo di newton ed applicarlo alla funzione* functionWithNoRealZero*, con innesco iniziale* $x_0 = 2.5$*, una tollerenza assoluta e relativa* $tol_X = rTol_X = 10^{-14}$ *ed un numero massimo di iterazioni* $i_{max} = 7$*.*

Per l'implementazione del codice vedere Metodo di Newton.

```
octave:112> [x, i, ascisse] =
newtonMethod('functionWithNoRealZero','
   functionWithNoRealZeroDerivative', 2.5, 7, 1e-14, 1e
   -14, 'incrementCriterion')
Il metodo non converge.
x = -3.18786023393994e+00
i =  7.00000000000000e+00
ascisse = [too long to report here]
octave:113> xNoZero = min(ascisse)-1:0.1:max(ascisse)+1
octave:114> yNoZero = invokeDelegate('
   functionWithNoRealZero', xNoZero)
octave:115> [prepX, prepY] =
   prepareForPlottingMethodSegments(ascisse, '
   invokeDelegate', 'functionWithNoRealZero')
octave:116> plot(xNoZero, yNoZero, "c", ascisse,
   invokeDelegate('functionWithNoRealZero', ascisse), "b+"
   , prepX, prepY, "r")
octave:117> grid
octave:118> print 'newtonNoZeroPlotOutput.tex' '-dTex' '-
   S800, 600'
```

Non si raggiunge la convergenza, infatti vengono fatti il massimo possibile dei passi fissati dal parametro i_{max}. Questo l'output del comando *octave:118*:



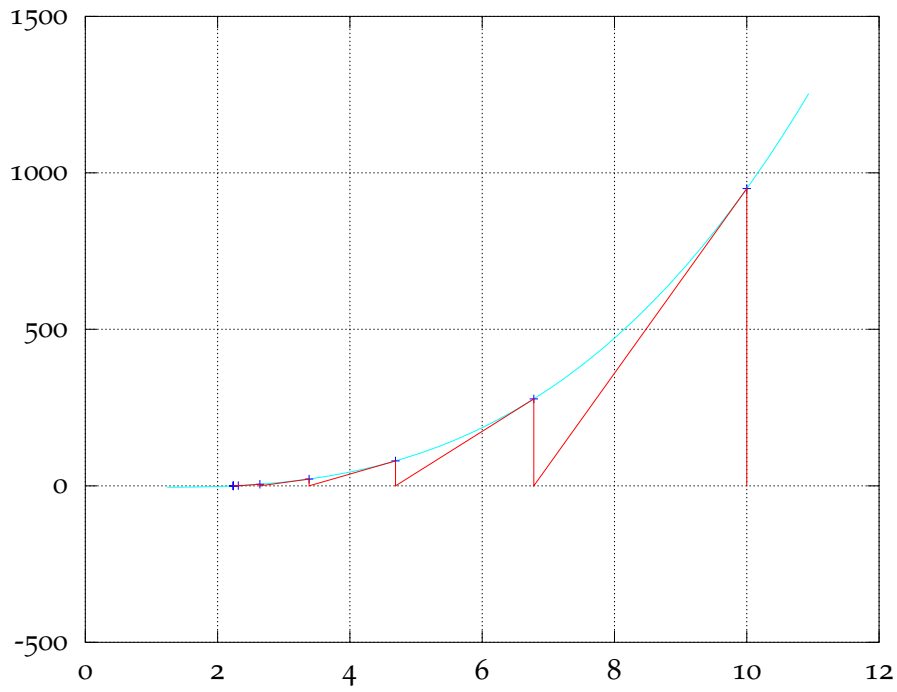
Exercise 2.3.3 (2.4). *Per il testo dell'esercizio consultare il libro di testo.*

Per l'implementazione del codice vedere Metodo di Newton.
Nel primo caso studio il punto di innesco $x_0 = 10$:

```
octave:112> [x, i, ascisse] =
newtonMethod('functionNewtonRecursion','
    functionNewtonRecursionDerivative', 10, 5e1, 1e-14, 1e
    -14, 'incrementCriterion')
x =  2.23606797749979e+00
i =  9.00000000000000e+00
ascisse = [too long to report here]
octave:113> xNoZero = min(ascisse)-1:0.1:max(ascisse)+1
octave:114> yNoZero = invokeDelegate('
    functionNewtonRecursion', xNoZero)
octave:115> [prepX, prepY] =
    prepareForPlottingMethodSegments(ascisse, '
    invokeDelegate', 'functionNewtonRecursion')
octave:116> plot(xNoZero, yNoZero, "c", ascisse,
    invokeDelegate('functionNewtonRecursion', ascisse), "b+
    ", prepX, prepY, "r")
octave:117> grid
```

```
octave:118> print 'newtonRecursivePlotOutput.tex' '-dTex'
    '-S800, 600'
```

Si raggiunge la convergenza con 9 passi. Questo l'output del comando *octave:118*:

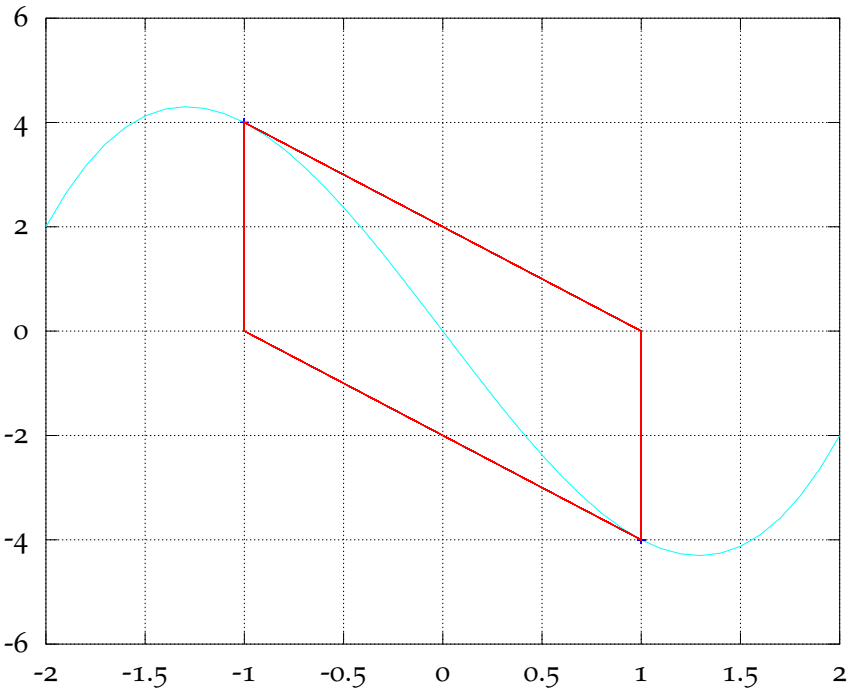


Nel secondo caso studio il punto di innesco $x_0 = 1$:

```
octave:112> [x, i, ascisse] =
newtonMethod('functionNewtonRecursion','
    functionNewtonRecursionDerivative', 1, 5e1, 1e-14, 1e
    -14, 'incrementCriterion')
Il metodo non converge.
x = -1.00000000000000e+00
i =  5.00000000000000e+01
ascisse = [too long to report here]
octave:113> xNoZero = min(ascisse)-1:0.1:max(ascisse)+1
octave:114> yNoZero = invokeDelegate('
    functionNewtonRecursion', xNoZero)
octave:115> [prepX, prepY] =
    prepareForPlottingMethodSegments(ascisse, '
    invokeDelegate', 'functionNewtonRecursion')
```

```
octave:116> plot(xNoZero, yNoZero, "c", ascisse,
    invokeDelegate('functionNewtonRecursion', ascisse), "b+
    ", prepX, prepY, "r")
octave:117> grid
octave:118> print 'newtonRecursiveNotConvergencePlotOutput
    .tex' '-dTex' '-S800,600'
```

Il metodo non converge creando una finestra come si vede nel grafico. Questo l'output del comando *octave:118*:



2.4 VARIANTI DEL METODO DI NEWTON

Exercise 2.4.1 (2.5). *Per il testo dell'esercizio consultare il libro di testo.*

Per l'implementazione del codice vedere Metodo di Newton, Molteplicità dello zero nota, Molteplicità dello zero non nota - variante Aitken.

I seguenti risultati sono stati generati invocando lo script Script eser 2.5:

```
octave:221> scriptExercise25
```

Nella seguente tabella riporto l'applicazione dei metodi richiesti:

	function25first		function25second	
Newton standard				
tol_X	steps	x	steps	
0.01	$3.60000000000000e+01$	$1.18248003631401e+00$	$4.50000000000000e+01$	1.18529
0.0001	$8.00000000000000e+01$	$1.00176964345428e+00$	$9.00000000000000e+01$	1.00165
$1e-06$	$1.24000000000000e+02$	$1.00001716153733e+00$	$1.33000000000000e+02$	1.00001
$1e-08$	$1.68000000000000e+02$	$1.00000016642808e+00$	$1.77000000000000e+02$	1.00000
$1e-10$	$2.11000000000000e+02$	$1.00000000179331e+00$	$2.21000000000000e+02$	1.00000
$1e-12$	$2.55000000000000e+02$	$1.00000000001739e+00$	$2.65000000000000e+02$	1.00000
$1e-14$	$2.99000000000000e+02$	$1.00000000000017e+00$	$3.08000000000000e+02$	1.00000
$1e-16$	$3.48000000000000e+02$	$1.00000000000000e+00$	$3.56000000000000e+02$	1.00000
Newton modificato				
tol_X	steps	x	steps	
0.01	$1.00000000000000e+00$	$1.00000000000000e+00$	$4.00000000000000e+00$	1.00000
0.0001	$1.00000000000000e+00$	$1.00000000000000e+00$	$5.00000000000000e+00$	1.00000
$1e-06$	$1.00000000000000e+00$	$1.00000000000000e+00$	$5.00000000000000e+00$	1.00000
$1e-08$	$1.00000000000000e+00$	$1.00000000000000e+00$	$6.00000000000000e+00$	1.00000
$1e-10$	$1.00000000000000e+00$	$1.00000000000000e+00$	$6.00000000000000e+00$	1.00000
$1e-12$	$1.00000000000000e+00$	$1.00000000000000e+00$	$6.00000000000000e+00$	1.00000
$1e-14$	$1.00000000000000e+00$	$1.00000000000000e+00$	$6.00000000000000e+00$	1.00000
$1e-16$	$1.00000000000000e+00$	$1.00000000000000e+00$	$7.00000000000000e+00$	1.00000
Newton Aitken				
tol_X	steps	x	steps	
0.01	$2.00000000000000e+00$	$1.00000000000000e+00$	$6.00000000000000e+00$	9.99999
0.0001	$2.00000000000000e+00$	$1.00000000000000e+00$	$7.00000000000000e+00$	1.00000
$1e-06$	$2.00000000000000e+00$	$1.00000000000000e+00$	$7.00000000000000e+00$	1.00000
$1e-08$	$2.00000000000000e+00$	$1.00000000000000e+00$	$7.00000000000000e+00$	1.00000
$1e-10$	$2.00000000000000e+00$	$1.00000000000000e+00$	$8.00000000000000e+00$	1.00000
$1e-12$	$2.00000000000000e+00$	$1.00000000000000e+00$	$8.00000000000000e+00$	1.00000
$1e-14$	$3.00000000000000e+00$	$1.00000000000000e+00$	$8.00000000000000e+00$	1.00000
$1e-16$	$3.00000000000000e+00$	$1.00000000000000e+00$	$8.00000000000000e+00$	1.00000

Observation 2.4.1. *Ho costruito lo script Script eser 2.5 - Newton Stop Criteria Comparison per effettuare un confronto tra i due metodi, nel momento in cui ho compilato la precedente tabella non avevo ancora implementato il metodo di Newton con la possibilità di specificare il criterio di arresto, pertanto questa osservazione è da ritenersi una aggiunta alla tabella precedente.*

```
octave:13> scriptExercise25NewtonComparison
```

Ottengo:

```
application of the Newton
   method with increment
   stop criteria
application with tolx =0.01
x =  1.18248003631401e+00
i =  3.60000000000000e+01
application with tolx
   =0.0001
x =  1.00176964345428e+00
i =  8.00000000000000e+01
application with tolx =1e-06
x =  1.00001716153733e+00
i =  1.24000000000000e+02
application with tolx =1e-08
x =  1.00000016642808e+00
i =  1.68000000000000e+02
application with tolx =1e-10
x =  1.00000000179331e+00
i =  2.11000000000000e+02
application with tolx =1e-12
x =  1.00000000001739e+00
i =  2.55000000000000e+02
application with tolx =1e-14
x =  1.00000000000017e+00
i =  2.99000000000000e+02
application with tolx =1e-16
x =  1.00000000000000e+00
i =  3.48000000000000e+02

application of the second
   function
application with tolx =0.01
x =  1.18529447118482e+00
i =  4.50000000000000e+01
application with tolx
   =0.0001
x =  1.00165056388247e+00
i =  9.00000000000000e+01
application with tolx =1e-06
x =  1.00001778848790e+00
i =  1.33000000000000e+02
application with tolx =1e-08
x =  1.00000017250842e+00
i =  1.77000000000000e+02
application with tolx =1e-10
x =  1.00000000167294e+00
i =  2.21000000000000e+02
application with tolx =1e-12
x =  1.00000000001622e+00
i =  2.65000000000000e+02
application with tolx =1e-14
x =  1.00000000000017e+00
i =  3.08000000000000e+02
application with tolx =1e-16
x =  1.00000000000000e+00
i =  3.56000000000000e+02
```

```
application of the Newton
    method with residue stop
    criteria
application with tolx =0.01
x =  1.18248003631401e+00
i =  3.60000000000000e+01
application with tolx
    =0.0001
x =  1.00176964345428e+00
i =  8.00000000000000e+01
application with tolx =1e-06
x =  1.00001716153733e+00
i =  1.24000000000000e+02
application with tolx =1e-08
x =  1.00000016642808e+00
i =  1.68000000000000e+02
application with tolx =1e-10
x =  1.00000000179331e+00
i =  2.11000000000000e+02
application with tolx =1e-12
x =  1.00000000001739e+00
i =  2.55000000000000e+02
application with tolx =1e-14
x =  1.00000000000017e+00
i =  2.99000000000000e+02
application with tolx =1e-16
x =  1.00000000000000e+00
i =  3.43000000000000e+02
```

```
application of the second
    function
application with tolx =0.01
x =  1.18529447118482e+00
i =  4.50000000000000e+01
application with tolx
    =0.0001
x =  1.00165056388247e+00
i =  9.00000000000000e+01
application with tolx =1e-06
x =  1.00001778848790e+00
i =  1.33000000000000e+02
application with tolx =1e-08
x =  1.00000017250842e+00
i =  1.77000000000000e+02
application with tolx =1e-10
x =  1.00000000167294e+00
i =  2.21000000000000e+02
application with tolx =1e-12
x =  1.00000000001622e+00
i =  2.65000000000000e+02
application with tolx =1e-14
x =  1.00000000000017e+00
i =  3.08000000000000e+02
application with tolx =1e-16
x =  1.00000000000000e+00
i =  3.52000000000000e+02
```

L'esecuzione dello script permette di evidenziare che il metodo con criterio di arresto per residuo *permette di convergere più velocemente alla soluzione, infatti si guadagnano 5 passi per prima funzione e 4 per la seconda.*

Exercise 2.4.2 (2.9)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Tutti i codici riguardanti i metodi richiesti sono implementati per aumentare la "robustezza" delle iterazioni (controllando che eventualmente i denominatori delle varie funzioni siano non nulli, . . .).

2.5 METODO QUASI NEWTON

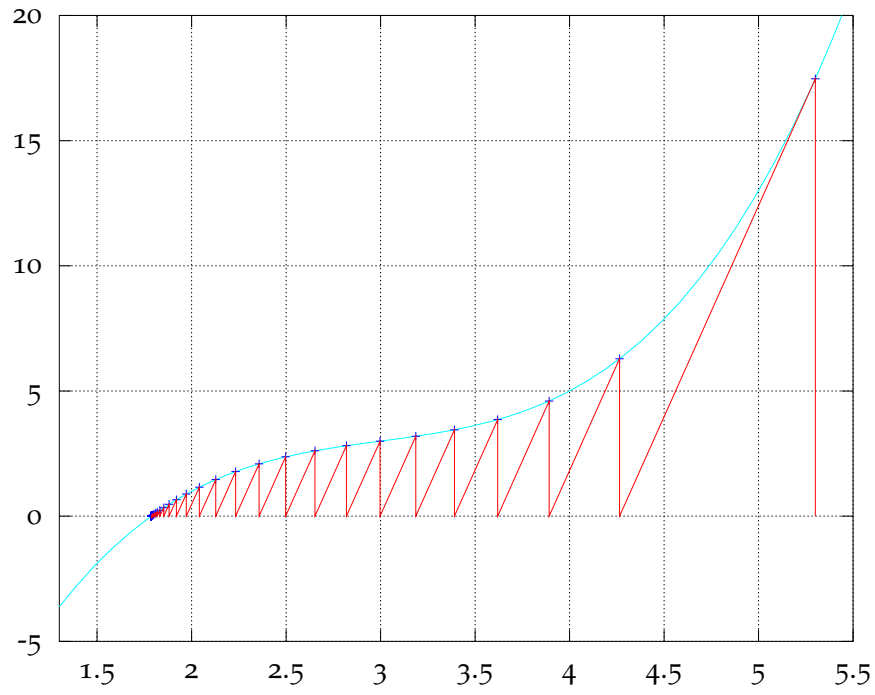
Exercise 2.5.1. *Implementare il metodo delle corde ed applicarlo alla funzione* chordConvergenceFunction*, con innesco iniziale* $x_0 = 5.3$*, una tollerenza assoluta e*

relativa $\mathrm{tol}_X = \mathrm{rTol}_X = 10^{-14}$ *ed un numero massimo di iterazioni* $\mathrm{i}_{max} = 10^5$.

Per l'implementazione del codice vedere Metodo delle corde.

```
octave:112> [x, i, ascisse] =
chordMethodLinearCriteria('chordConvergenceFunction','
    chordConvergenceFunctionDerivative',5.3, 1e5, 1e-14, 1e
    -14)
x =  1.78658833723778e+00
i =  9.40000000000000e+01
ascisse = [too long to report here]
octave:113> xSingleZero = min(ascisse)-1:0.1:max(ascisse)
    +1
octave:114> ySingleZero = invokeDelegate('
    chordConvergenceFunction', xSingleZero)
octave:115> [prepX, prepY] =
    prepareForPlottingMethodSegments(ascisse, '
    invokeDelegate', 'chordConvergenceFunction')
octave:116> plot(xSingleZero, ySingleZero, "c", ascisse,
    invokeDelegate('chordConvergenceFunction', ascisse), "b
    +", prepX, prepY, "r")
octave:117> axis([1.3, 5.5, -5, 20])
octave:118> grid
octave:119> print 'chordPlotOutput.tex' '-dTex' '-S800,
    600'
```

Si raggiunge la tolleranza richiesta in 94 passi. Questo l'output del comando *octave:119*:



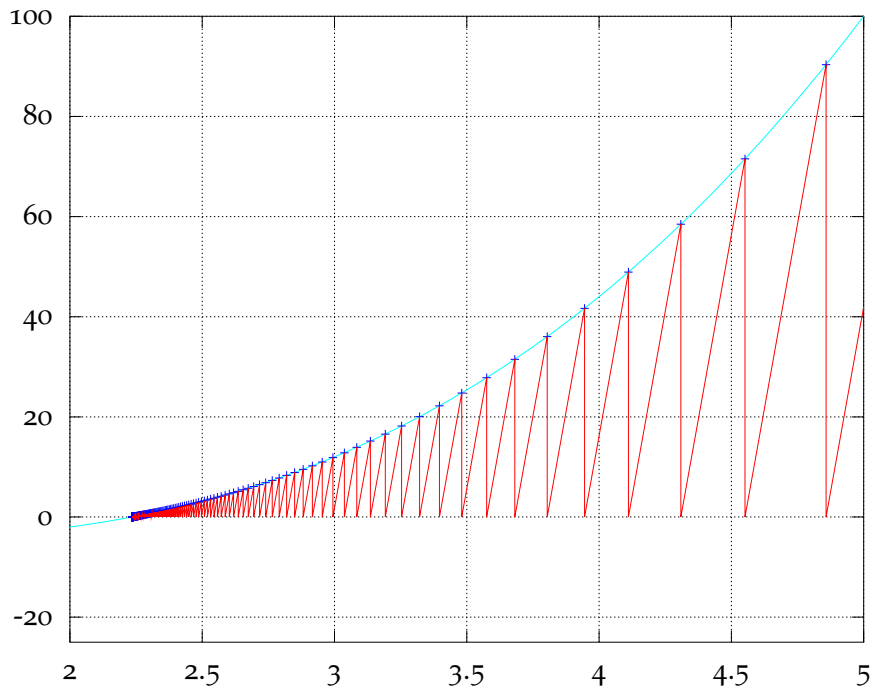
Exercise 2.5.2. *Applicare il metodo delle corde in riferimentoa all'esercizio 2.3.3 Usare una tollerenza assoluta e relativa* $\mathrm{tol}_X = \mathrm{rTol}_X = 10^{-14}$ *ed un numero massimo di iterazioni* $i_{max} = 10^5$ *e punto di innesco* $x_0 = 10$.

Per l'implementazione del codice vedere Metodo delle corde.

```
octave:112> [x, i, ascisse] = chordMethodLinearCriteria('
    functionNewtonRecursion','
    functionNewtonRecursionDerivative',10,1e5, 1e-10, 1e
    -10)
x =  2.23606797759911e+00
i =  6.67000000000000e+02
ascisse = [too long to report here]
octave:113> xSingleZero = min(ascisse)-1:0.1:max(ascisse)
    +1
octave:114> ySingleZero = invokeDelegate('
    functionNewtonRecursion', xSingleZero)
octave:115> [prepX, prepY] =
    prepareForPlottingMethodSegments(ascisse, '
    invokeDelegate', 'functionNewtonRecursion')
octave:116> plot(xSingleZero, ySingleZero, "c", ascisse,
    invokeDelegate('functionNewtonRecursion', ascisse), "b+
    ", prepX, prepY, "r")
```

```
octave:117> axis([2, 5, -25, 100])
octave:118> grid
octave:119> print 'chordNewtonRecursionPlotOutput.tex' '-
    dTex' '-S800, 600'
```

Si raggiunge la tolleranza richiesta in 667 passi. Questo l'output del comando *octave:119*:

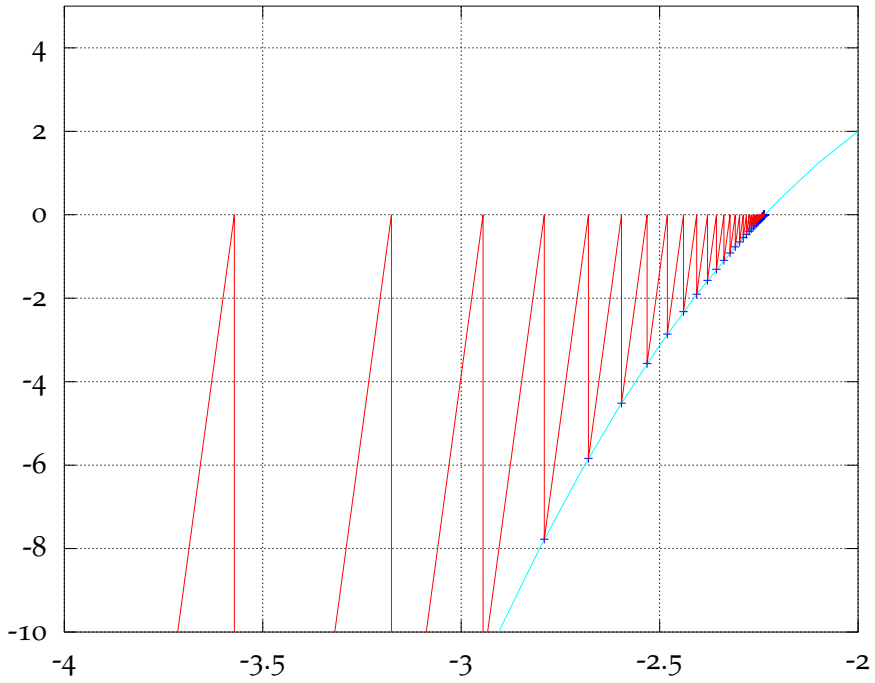


Se invece voglio trovare la soluzione negativa:

```
octave:112> [x, i, ascisse] = chordMethodLinearCriteria('
    functionNewtonRecursion','
    functionNewtonRecursionDerivative',-5,1e5, 1e-5, 1e-5)
i =  7.30000000000000e+01
ascisse = [too long to report here]
octave:113> xSingleZero = min(ascisse)-1:0.1:max(ascisse)
    +1
octave:114> ySingleZero = invokeDelegate('
    functionNewtonRecursion', xSingleZero)
octave:115> [prepX, prepY] =
    prepareForPlottingMethodSegments(ascisse, '
    invokeDelegate', 'functionNewtonRecursion')
```

```
octave:116> plot(xSingleZero, ySingleZero, "c", ascisse,
   invokeDelegate('functionNewtonRecursion', ascisse), "b+
   ", prepX, prepY, "r")
octave:117> axis([-4, -2, -10, 5])
octave:118> grid
octave:119> print 'chordNewtonRecursionNegativePlotOutput.
   tex' '-dTex' '-S800,600'
```

Si raggiunge la tolleranza richiesta in 73 passi. Questo l'output del comando *octave:119*:



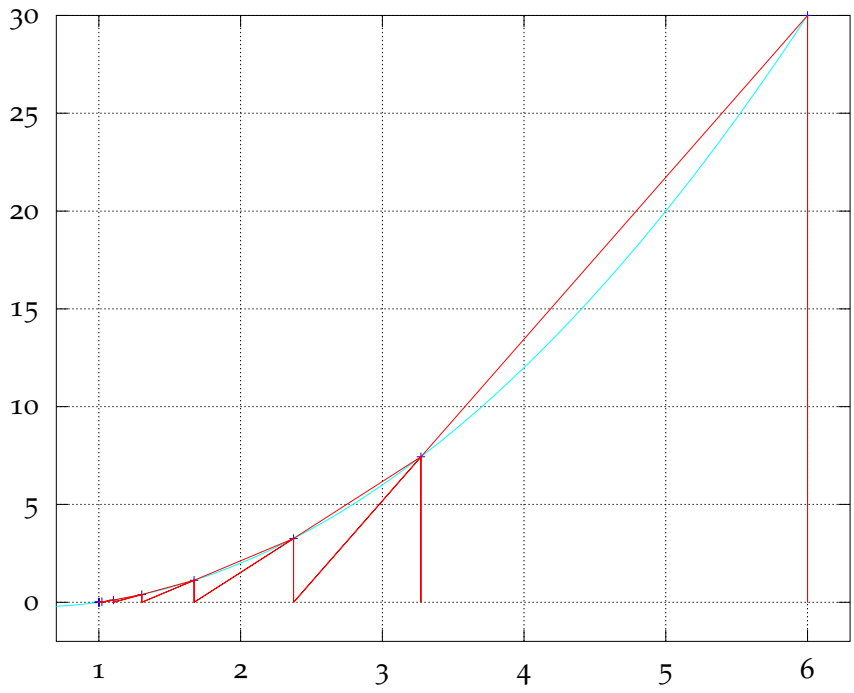
Observation 2.5.1. *Il metodo delle secanti converge più rapidamente quando la derivata non è molto ripida: in questo esempio accade il contrario, ovvero la derivata è molto ripida, il metodo converge lentamente ma si guadagna in precisione per l'approssimazione fornita.*

Exercise 2.5.3. *Implementare il metodo delle corde ed applicarlo alla funzione* secantConvergenceFunction*, con innesco iniziale* $x_0 = 6$*, una tollerenza assoluta e relativa* $tol_X = rTol_X = 10^{-14}$ *ed un numero massimo di iterazioni* $i_{max} = 10^5$.

Per l'implementazione del codice vedere Metodo delle secanti.

```
octave:45> [x, i, ascisse] = secantMethod('
   secantConvergenceFunction','
   secantConvergenceFunctionDerivative',6,
1e5, 1e-14, 1e-14)
x =  1.00000000000000e+00
i =  1.10000000000000e+01
ascisse = [too long to report here]
octave:113> xSingleZero = min(ascisse)-1:0.1:max(ascisse)
   +1
octave:114> ySingleZero = invokeDelegate('
   secantConvergenceFunction', xSingleZero)
octave:115> [prepX, prepY] =
   prepareForPlottingSecantMethodSegments(ascisse, '
   invokeDelegate', 'secantConvergenceFunction')
octave:116> plot(xSingleZero, ySingleZero, "c", ascisse,
   invokeDelegate('secantConvergenceFunction', ascisse), "
   b+", prepX, prepY, "r")
octave:118> grid
octave:119> axis([0.7, 6.3, -2, 30])
octave:120> print 'secantPlotOutput.tex' '-dTex' '-S800,
   600'
```

Si raggiunge la tolleranza richiesta in 11 passi. Questo l'output del comando *octave:120*:



Observation 2.5.2. *Se provo ad eseguire il metodo di Newton sulla stessa funzione ottengo:*

```
octave:22> [x, i] =
newtonMethod('secantConvergenceFunction','
    secantConvergenceFunctionDerivative',6, 1e5, 1e-14, 1e
    -14, 'residueCriterion')
x =  5.00000000000000e+00
i =  8.00000000000000e+00
```

Più veloce di tre passi.

Exercise 2.5.4 (2.7 e 2.8)**.** *Per i testi degli esercizi consultare il libro di testo.*

Ho costruito lo script Script eser 2.7, lanciandolo

```
octave:13> scriptExercise27
```

ottengo:

```
bisection method
tol_x =0.01
x =  7.34375000000000e-01
i =  6.00000000000000e+00
tol_x =0.0001
x =  7.39013671875000e-01
i =  1.20000000000000e+01
tol_x =1e-06
x =  7.39084243774414e-01
i =  1.90000000000000e+01
tol_x =1e-08
x =  7.39085137844086e-01
i =  2.40000000000000e+01
tol_x =1e-10
x =  7.39085133187473e-01
i =  3.00000000000000e+01
tol_x =1e-12
x =  7.39085133214758e-01
i =  3.90000000000000e+01
tol_x =1e-14
x =  7.39085133215156e-01
i =  4.40000000000000e+01
tol_x =1e-16
x =  7.39085133215161e-01
i =  5.20000000000000e+01
i =  4.00000000000000e+00
tol_x =1e-08
x =  7.39085133215161e-01
i =  4.00000000000000e+00
tol_x =1e-10
x =  7.39085133215161e-01
i =  4.00000000000000e+00
tol_x =1e-12
x =  7.39085133215161e-01
i =  5.00000000000000e+00
tol_x =1e-14
x =  7.39085133215161e-01
i =  5.00000000000000e+00
tol_x =1e-16
x =  7.39085133215161e-01
i =  5.00000000000000e+00
```

```
Newton method
tol_x =0.01
x =  7.39112890911362e-01
i =  2.00000000000000e+00
tol_x =0.0001
x =  7.39085133385284e-01
i =  3.00000000000000e+00
tol_x =1e-06
x =  7.39085133215161e-01
```

```
chord method
tol_x =0.01
x =  7.37506890513243e-01
i =  1.40000000000000e+01
tol_x =0.0001
x =  7.39071365298945e-01
i =  2.60000000000000e+01
tol_x =1e-06
x =  7.39085311606762e-01
i =  3.70000000000000e+01
tol_x =1e-08
x =  7.39085134772174e-01
i =  4.90000000000000e+01
tol_x =1e-10
x =  7.39085133228750e-01
i =  6.10000000000000e+01
tol_x =1e-12
x =  7.39085133214985e-01
i =  7.20000000000000e+01
tol_x =1e-14
x =  7.39085133215159e-01
i =  8.40000000000000e+01
tol_x =1e-16
x =  7.39085133215161e-01
i =  9.40000000000000e+01
```

```
secant method
tol_x =0.01
x =  7.39119361911629e-01
i =  3.00000000000000e+00
tol_x =0.0001
x =  7.39085112127464e-01
i =  4.00000000000000e+00
tol_x =1e-06
x =  7.39085133215001e-01
i =  5.00000000000000e+00
tol_x =1e-08
x =  7.39085133215161e-01
i =  6.00000000000000e+00
tol_x =1e-10
x =  7.39085133215161e-01
i =  6.00000000000000e+00
tol_x =1e-12
x =  7.39085133215161e-01
i =  6.00000000000000e+00
tol_x =1e-14
x =  7.39085133215161e-01
i =  7.00000000000000e+00
tol_x =1e-16
x =  7.39085133215161e-01
i =  7.00000000000000e+00
```

Per l'ultima iterazione del metodo delle corde ho ottenuto uno warning di divionse per zero.

3

SISTEMI LINEARI

3.1 ESERCIZI PRELIMINARI

Exercise 3.1.1 (3.2). *Se* $A, B \in \mathbb{R}^{n\times n}$ *triangolari inferiori valgono rispettivamente:*

$$A + B = C \quad \wedge \quad AB = D$$

con $C, D \in \mathbb{R}^{n\times n}$ *triangolari inferiori.*

Proof of A+B=C. Per definizione di triangolare inferiore devo dimostrare che $c_{ij} = 0$ se $i < j$.

Per la definizione dell'operatore $+(,)$ tra matrici ottengo

$$c_{ij} = a_{ij} + b_{ij}$$

Per ipotesi A, B sono triangolari inferiori, quindi $a_{ij} = b_{ij} = 0$ se $i < j$, quindi $c_{ij} = 0$ se $i < j$ e questo termina la prova. □

Proof of AB=D. Per definizione di triangolare inferiore devo dimostrare che $c_{ij} = 0$ se $i < j$.

Per definizione dell'operatore $\cdot(,)$ tra matrici, d_{ij} è dato dal prodotto dell'i-esima riga con la j-esima colonna, formalmente se $i < j$

$$d_{ij} = \left(\boldsymbol{e}_i^T A\right)\left(B\boldsymbol{e}_j\right) = \begin{bmatrix} a_{i1} \cdots a_{ii} \underbrace{0 \cdots 0}_{n-i} \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ b_{i+1,j} \\ \vdots \\ b_{nj} \end{bmatrix} = 0$$

Svolgendo il prodotto riga-colonna si osserva che le prime i componenti della colonna annullano le prime i componenti della riga e le ultime $n-i$ componenti della riga annullano le ultime $n-i$ componenti della colonna, quindi si sommano tutti elementi nulli, per cui

$$d_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = 0 \quad \text{se } i < j$$

e questo termina la prova. □

Con argomento simmetrico si dimostra che vale lo stesso asserto se $A, B, C, D \in \mathbb{R}^{n\times n}$ triangolari superiori.

Exercise 3.1.2 (3.3). *Se $A, B \in \mathbb{R}^{n\times n}$ triangolari inferiori a diagonale unitaria allora $AB = C$, con $C \in \mathbb{R}^{n\times n}$ triangolare inferiore a diagonale unitaria.*

Dimostrazione. A, B sono triangolari inferiori, quindi per l'esercizio 3.1.1, anche C è triangolare inferiore. Rimane da dimostrare che $c_{ii} = 1$ per $i \in \{1, \dots, n\}$.

Posso scrivere c_{ii} come prodotto dell'i-esima riga con la i-esima colonna, formalmente se $i \geqslant j$

$$c_{ii} = \left(\boldsymbol{e}_i^T A\right)(B\boldsymbol{e}_i) = \begin{bmatrix} a_{11} & & & & \\ \vdots & \ddots & & & \\ a_{i1} & \cdots & a_{ii} & & \\ \vdots & & & \ddots & \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} b_{11} & & & & \\ \vdots & \ddots & & & \\ b_{i1} & \cdots & b_{ii} & & \\ \vdots & & & \ddots & \\ b_{n1} & \cdots & \cdots & \cdots & b_{nn} \end{bmatrix} =$$

$$= \begin{bmatrix} a_{i1} & \cdots & a_{i,i-1} & a_{ii} & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ b_{i,i} \\ b_{i+1,i} \\ \vdots \\ b_{ni} \end{bmatrix} = \sum_{k=1}^{n} a_{in} b_{ni} = a_{ii} b_{ii} = 1$$

Svolgendo il prodotto riga-colonna si osserva che le prime $i-1$ componenti della colonna annullano le prime $i-1$ componenti della riga e le ultime $n-i$ componenti della riga annullano le ultime $n-i$ componenti della colonna, rimane solo il termine $a_{ii}b_{ii}$, ma per ipotesi $a_{ii} = b_{ii} = 1$ perchè A, B sono a diagonale unitaria, quindi $c_{ii} = 1$ e questo termina la prova. □

Exercise 3.1.3 (3.4). *Se A è triangolare inferiore allora A^{-1} è triangolare inferiore.*

Dimostrazione. Prova per assurdo.

Le ipotesi di assurdo sono: A triangolare inferiore ed A^{-1} non triangolare inferiore.

Per definizione di matrice inversa, chiamo $B = A^{-1}$, B non triangolare inferiore, vale:

$$\begin{bmatrix} a_{11} & & & & \\ \vdots & \ddots & & & \\ a_{i1} & \cdots & a_{ii} & & \\ \vdots & & & \ddots & \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & \cdots & \cdots & b_{1n} \\ \vdots & \ddots & & & \vdots \\ b_{i1} & \cdots & b_{ii} & & \vdots \\ \vdots & & & \ddots & \vdots \\ b_{n1} & \cdots & \cdots & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}$$

Studio come si ottiene la prima riga della matrice identità:

$$e_1^T I = \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix} = \begin{matrix} a_{11} \begin{bmatrix} b_{11} & \dots & b_{1n} \end{bmatrix} \\ +0 \begin{bmatrix} b_{21} & \dots & b_{2n} \end{bmatrix} \\ \vdots \\ +0 \begin{bmatrix} b_{n1} & \dots & b_{nn} \end{bmatrix} \end{matrix} = a_{11} \begin{bmatrix} b_{11} & \dots & b_{1n} \end{bmatrix}$$

Per ipotesi di assurdo B non è triangolare inferiore, quindi $\exists k \neq 1 : b_{1k} \neq 0$ e quindi anche $a_{11} b_{1k} \neq 0$. Ma questo è assurdo perchè deve valere

$$\begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix} e_k = \begin{bmatrix} a_{11} b_{11} & \dots & a_{11} b_{1n} \end{bmatrix} e_k$$
$$0 = a_{11} b_{1k}$$

Ma $a_{11} b_{1k} \neq 0$, cado in assurdo, quindi $B = A^{-1}$ deve essere triangolare inferiore. □

Exercise 3.1.4 (3.4)**.** *Se A è triangolare inferiore a diagonale unitaria allora A^{-1} è triangolare inferiore a diagonale unitaria.*

Dimostrazione. Per ipotesi A è triangolare inferiore, per l'esercizio 3.1.3, anche A^{-1} è triangolare inferiore. Rimane da dimostrare che $\forall i : b_{ii} = 1$ con $b_{kj} \in B = A^{-1}$.

Per definizione di matrice inversa, chiamo $B = A^{-1}$, vale:

$$\begin{bmatrix} 1 & & & & \\ \vdots & \ddots & & & \\ a_{i1} & \cdots & 1 & & \\ \vdots & & & \ddots & \\ a_{n1} & \cdots & \cdots & \cdots & 1 \end{bmatrix} \begin{bmatrix} b_{11} & & & & \\ \vdots & \ddots & & & \\ b_{i1} & \cdots & b_{ii} & & \\ \vdots & & & \ddots & \\ b_{n1} & \cdots & \cdots & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}$$

Prova per assurdo. Suppongo che B non sia a diagonale unitaria, supponiamo $b_{11} \neq 1$. Studio come si ottiene la prima riga della matrice identità:

$$e_i^T I = \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix} = \begin{matrix} 1 \begin{bmatrix} b_{11} & 0 & \dots & 0 \end{bmatrix} \\ +0 \begin{bmatrix} b_{21} & b_{22} & 0 & \dots & 0 \end{bmatrix} \\ \vdots \\ +0 \begin{bmatrix} b_{n1} & \dots & b_{nn} \end{bmatrix} \end{matrix} = \begin{bmatrix} b_{11} & 0 & \dots & 0 \end{bmatrix}$$

Affinchè sia vera l'uguaglianza deve valere $1 = b_{11}$. Ma questo è assurdo perchè per ipotesi di assurdo $b_{11} \neq 1$, quindi $B = A^{-1}$ è triangolare inferiore a diagonale unitaria. □

Exercise 3.1.5. *Se A è triangolare allora* $\det(A) = a_{11} \cdots a_{nn}$

Dimostrazione. Suppongo che A sia triangolare inferiore, lo stesso argomento si dimostra in modo simmetrico per le matrici triangolari superiori B: considerando B^T si torna ad una triangolare inferiore e quindi si può usare questa prova. Quindi A è della forma:

$$\begin{bmatrix} a_{11} & & & & \\ \vdots & \ddots & & & \\ a_{i1} & \cdots & a_{ii} & & \\ \vdots & & & \ddots & \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix}$$

Calcolo il determinante rispetto alla prima riga:

$$\det(A) = a_{11}A_{11} + a_{12}A_{12} \cdots + a_{1n}A_{1n}$$

Per ipotesi A è tringolare inferiore, quindi $a_{ij} = 0$ se $i < j$, quindi

$$\det(A) = a_{11}A_{11}$$

Calcolare A_{11} equivale a calcolare la funzione det della matrice B ottenuta da A cancellando la prima riga e la prima colonna:

$$B = \begin{bmatrix} a_{22} & & & & \\ \vdots & \ddots & & & \\ a_{i2} & \cdots & a_{ii} & & \\ \vdots & & & \ddots & \\ a_{n2} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix}$$

Ma anche questa matrice è una tringolare inferiore, posso riapplicare lo stesso ragionamento per il calcolo del determinante, ottenendo $\det(B) = a_{22}B_{11}$, considerare una nuova matrice C ottenuta da B cancellando la prima riga e la prima colonna. [1] Il procedimento ricorsivo si ferma quando si arriva alla sotto-matrice X con il solo elemento $x_{11} = a_{nn}$ e, per la definizione di det vale $\det(a_{nn}) = a_{nn}$. Ricomponenti i pezzi si ottiene $\det(A) = a_{11} \cdots a_{nn}$ e questo termina la prova. □

Exercise 3.1.6 (3.5, Lemma 3.2). $A \in \mathbb{R}^{n\times n}$ *triangolare.*
A è non singolare sse tutti i suoi minori principali sono non nulli.

Proof of ⇒. Per ipotesi A è tringolare e $\det(A) \neq 0$. Devo dimostrare che $\det(A_k) \neq 0, \forall k \in \{1, \ldots, n\}$.

Prova per induzione completa su k.

BASE per $k = 1$ ottengo $\det A_1 = a_{11}$. Per ipotesi A è non singolare quindi $\det A_1 = a_{11} \neq 0$, la base è dimostrata.

IPOTESI INDUTTIVA suppongo vero $\det(A_j) \neq 0, \forall j \in \{1, \ldots, k-1\}$.

PASSO INDUTTIVO devo dimostrare che $\det(A_k) \neq 0$. Per la definizione della funzione det:

$$\det(A) = a_{r1}A_{r1} + \ldots + a_{rn}A_{rn}, \quad \forall r \in 1, \ldots, n$$

con $A_{ij} = (-1)^{i+j}M_{ij}(A)$ e $M_{ij}(A)$ restituisce il determinante della matrice ottenuta non considerando l'i-esima riga e la j-esima colonna della matrice A passata come parametro.

Applico la definizione della funzione det alla sotto-matrice A_k: per ipotesi $\det(A) \neq 0$, quindi posso fissare l'indice $r = 1$ della riga rispetto a cui calcolo il determinante in quanto $\det(A)$ non cambia rispetto alla riga (e anche alle colonne) rispetto a cui viene calcolato. Ottengo:

$$\det(A_k) = a_{11}A_{k_{11}} + \ldots + a_{1n}A_{k_{1n}}$$

Per ipotesi A è triangolare (suppongo sia triangolare inferiore) quindi $a_{ij} = 0$ per $i < j$, quindi:

$$\det(A_k) = a_{11}A_{k_{11}}$$

Ancora usando l'ipotesi che A è triangolare, la struttura di triangolare viene preservata da ogni sua sotto-matrice principale, ovvero A_o è tringolare $\forall o \in$

1 si cancella sempre la prima riga e la prima colonna, in quanto calcolando il determinante di una tringolare inferiore rispetto alla prima riga, rimane solo il primo elemento $b_{11}, c_{11}, \ldots$ che coincidono rispettivamente con $a_{22}, a_{33}, \ldots$ perchè gli elementi della matrice non vengono modificati nel calcolo del determinante

$1, \dots, n$, quindi anche A_k è tringolare. Dato che devo calcolare il determinante della nuova matrice A' ottenuta da A_k non considerando la prima riga e la prima colonna ($A_{k_{11}} = (-1)^{1+1} M_{11}(A_k)$), devo calcolare il determinante di una nuova matrice triangolare. Calcolandolo sempre $\det(A')$ sulla prima riga ottengo $\det(A') = a_{22} A'_{22}$. Ripetendo il ragionamento fino ad arrivare alla sotto-matrice degenere ad un elemento $A^{k-1} = a_{kk}$, ottengo che:

$$\det(A_k) = a_{11} \cdots a_{kk}$$

Per ipotesi $\det(A) = a_{11} \cdots a_{nn} \neq 0$ [2] , allora anche $\det(A_k) \neq 0$ e questo completa il passo induttivo. □

Proof of $\Leftarrow$. Per ipotesi vale $\det(A_k) \neq 0, \forall k \in \{1, \dots, n\}$. Considero la sotto-matrice con $k = n$, quindi $A_n = A$. Applicando la funzione det ad entrambi i membri, per l'ipotesi si ottiene l'asserto. □

Exercise 3.1.7 (3.5, Lemma 3.3)**.** *Se* $A = LU$ *allora* $\det(A_k) = \det(U_k), \forall k \in \{1, \dots, n\}$

Dimostrazione. $\det(A_k) = \det(L_k U_k) = \det(L_k)\det(U_k)$ perchè L_k e U_k hanno la stessa dimensione. Ma L_k è tringolare inferiore a diagonale unitaria per costruzione della *fattorizzazione LU*, quindi $\det(L_k) = 1$, quindi si ottiene l'asserto. □

3.2 BEFORE PARTIAL PIVOTING

Exercise 3.2.1 (3.6)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Il numero di operazioni che compio è dato da:

$$\sum_{i=1}^{n} \left((n-i) + 2(n-i)^2\right) = \sum_{k=0}^{n-1} \left(k + 2k^2\right) = \sum_{k=0}^{n-1} k + 2\sum_{k=0}^{n-1} k^2$$

Con la prima somma si sommano i primi $n-1$ numeri naturali (escludendo lo zero), per la seconda somma posso applicare il suggerimento del testo $\sum_{i=0}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6}$, ottengo quindi:

$$\sum_{k=0}^{n-1} k + 2\sum_{k=0}^{n-1} k^2 = \frac{n(n-1)}{2} + \frac{2n(n-1)(2n-1)}{6} = n(n-1)\left(\frac{1}{2} + \frac{2n-1}{3}\right) =$$

$$= n(n-1)\frac{4n+1}{6} = (n^2-n)\frac{4n+1}{6} = \frac{4n^3-3n^2-n}{6} \approx \frac{2}{3}n^3$$

Considerando per l'ultima approssimazione solo il termine di grado massimo. □

2 È importante studiare l'*osservazione 3.1* del testo, in quanto fa notare che in genere non è vero che se $\det(A) = a_{11} \cdots a_{nn} \neq 0$ allora anche i minori principali sono non singolari. In questo caso posso utilizzare l'ipotesi in quanto sto supponendo che A sia triangolare per ipotesi.

Exercise 3.2.2 (3.7, 3.8)**.** *Per i testi degli esercizi consultare il libro di testo.*

Vedere il codice LUmethod.

Observation 3.2.1. *Posso vedere il* k*-esimo vettore di Gauss e la relativa matrice elementare come delle funzioni:*

$$g_k = g_k(\boldsymbol{v}) = \frac{1}{v_{kk}} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ v_{k+1} \\ \vdots \\ v_n \end{bmatrix} \quad , g_k : \mathbb{R}^n \to \mathbb{R}^n$$

$$L = L(\boldsymbol{v}) = I - g_k(\boldsymbol{v}) \boldsymbol{e}_k^T \quad , L : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n}$$

Inoltre al passo i*-esimo non si modifica la riga* i*: si modificano gli elementi sotto* a_{ii} *e tutta la sottomatrice* $A(i+1:n, i+1:n)$ *(scritta in notazione Octave).*

L'equazione del testo 3.22 scrive:

$$L = I + g_1 \boldsymbol{e}_1^T + \cdots + g_{n-1} \boldsymbol{e}_{n-1}^T = \begin{bmatrix} 1 & & & \\ g_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ g_{n1} & \cdots & g_{n,n-1} & 1 \end{bmatrix}$$

con

$$g_{ik} = \boldsymbol{e}_i^T g_k(A^{(k)} \boldsymbol{e}_k) = \boldsymbol{e}_i^T \left(\frac{1}{a_{kk}^{(k)}} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ a_{k+1,k}^{(k)} \\ \vdots \\ a_{nk}^{(k)} \end{bmatrix} \right)$$

Exercise 3.2.3 (3.9, Lemma 3.4)**.** *Se* A *è diagonale dominante per righe allora lo sono anche tutte le sue sotto-matrici principali.*

Dimostrazione. Per ipotesi A è diagonale dominante per righe, quindi posso costruire questa disuguaglianza:

$$|a_{ii}| > \sum_{j=1 j \neq i}^{n} |a_{ij}| \geqslant \sum_{j=1 j \neq i}^{k} |a_{ij}|, \quad k \leqslant n$$

con k indice della sotto-matrice di ordine k. La disuguaglianza a destra dimostra l'asserto e la prova è terminata. □

Exercise 3.2.4 (3.9, Lemma 3.5)**.** *A è diagonale dominante per righe sse A^T è diagonale dominante per colonne.*

Proof of $\Rightarrow$. Per ipotesi A è diagonale dominante per righe, ovvero:

$$|a_{ii}| > \sum_{j=1 j\neq i}^{n} |a_{ij}|$$

Costruisco la trasposta: se $a_{ij} \in A$ allora $a_{ji} \in A^T$. Riscrivo la definizione precedente:

$$|a_{ii}| > \sum_{j=1 j\neq i}^{n} |a_{ji}|$$

Ma questa è la definizione di matrice diagonale dominante per colonne e questo termina la prova. □

Proof of $\Leftarrow$. Con argomento simmetrico si dimostra anche l'implicazione inversa. □

Exercise 3.2.5 (3.10)**.** *Se $A = LDL^T$ allora A è sdp.*

Dimostrazione. Devo dimostrare che A soddisfa la definizione sdp:

- $A = LDL^T = (L^T)^T DL^T = LDL^T = A^T$, quindi A è simmetrica.
- $\forall \mathbf{x} \in \mathbb{R}^n, \mathbf{x} \neq \mathbf{0}$ allora deve valere:

  $$\mathbf{x}^T A\mathbf{x} = \mathbf{x}^T LDL^T \mathbf{x} > 0$$

  Costruisco il vettore $\mathbf{y} = L^T\mathbf{x}$, di conseguenza $\mathbf{y}^T = \mathbf{x}^T(L^T)^T = \mathbf{x}^T L$. Quindi posso riscrivere $\mathbf{y}^T D\mathbf{y} > 0$. Rappresento il prodotto $\mathbf{y}^T(D\mathbf{y})$:

$$\mathbf{y}^T D\mathbf{y} = \begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix} \begin{bmatrix} d_{11} & & & & \\ & d_{22} & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & d_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix} \begin{bmatrix} y_1 d_{11} \\ \vdots \\ y_n d_{nn} \end{bmatrix} = \sum_{i=1}^{n} y_i^2 d_{ii} \geqslant 0$$

  In quanto per ipotesi $d_{ii} > 0$. Per avere la somma uguale a 0, si dovrebbe avere $\mathbf{y} = \mathbf{0}$. Ma questo non è possibile in quanto $\mathbf{y} = L^T\mathbf{x}$ con L non singolare e $\mathbf{x} \neq \mathbf{0}$ per definizione di sdp. Quindi si ha:

  $$\mathbf{y}^T D\mathbf{y} = \sum_{i=1}^{n} y_i^2 d_{ii} > 0$$

  questo è quanto richiesto dalla definizione di sdp, quindi A è sdp e questo termina la prova.

□

Exercise 3.2.6 (3.11). *Se A è non singolare allora A^TA e AA^T sono sdp.*

Dimostrazione. Devo verifica che $B = A^TA$ sia sdp quindi:

- $B = A^TA = A^T(A^T)^T = A^TA = B^T$, quindi B è simmetrica
- $\forall x \in \mathbb{R}^n, x \neq 0$ allora deve valere:

  $$x^TBx = x^TA^TAx > 0$$

  Costruisco il vettore $y = Ax$, di conseguenza $y^T = x^TA^T$. Quindi posso riscrivere $y^Ty > 0$ e rappresento:

  $$y^Ty = \begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} y_i^2 \geqslant 0$$

  Per avere la somma uguale a 0, si dovrebbe avere $y = 0$. Ma questo non è possibile in quanto $y = Ax$ con A non singolare, quindi A non è la matrice nulla, e $x \neq 0$ per definizione di sdp. Quindi si ha:

  $$\sum_{i=1}^{n} y_i^2 > 0$$

  questo è quanto richiesta dalla definizione di sdp, quindi B è sdp.

  Dato che ho dimostrato che B è simmetrica allora anche B^T è sdp e questo termina la prova.

□

Exercise 3.2.7 (3.12). *Se $A \in \mathbb{R}^{m\times n}$, con $m \geqslant n = \mathrm{rank}(A)$ allora A^TA è sdp.*

Dimostrazione. Devo dimostrare che $B = A^TA$ è sdp.

- B è simmetrica per la prova della simmetria dell'esercizio 3.2.6.
- $\forall x \in \mathbb{R}^n, x \neq 0$ allora posso costruire un vettore (come fatto nell'esercizio precedente) $y = Ax, y \in \mathbb{R}^m$ ed il rispettivo $y^T = x^TA^T, y^T \in \mathbb{R}^{1\times m}$ e costruire il prodotto richiesto dalla definizione

  $$\sum_{i=1}^{m} y_i^2 \geqslant 0$$

  Dato che in A ci sono $m-n$ linearmente dipendenti, allora alcuni termini della sommatoria y_i possono essere nulli. Ma per le ipotesi, $n = \mathrm{rank}(A)$, quindi esiste un minore principale $\det(A_k) \neq 0$. Per come è costruito y allora $\exists i \in \mathbb{N} : y_i \neq 0$ perchè $\det(A_k) \neq 0$ per ipotesi e $x \neq 0$ per la definizione di sdp. Quindi la somma è strettamente positiva e questo termina la prova.

□

Exercise 3.2.8. *Sia* $S \in \mathbb{R}^{n\times n}$. *Se* $\det(S) \neq 0$ *allora* $A = SS^T$ *è sdp.*

Dimostrazione. Devo dimostrare che A soddisfa le due richieste della definizione di matrice sdp.

- $A = SS^T = (S^T)^T S^T = SS^T = A^T$, quindi A è simmetrica.
- $\forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{x} \neq \boldsymbol{0}$ allora

  $$\boldsymbol{x}^T A \boldsymbol{x} = \boldsymbol{x}^T SS^T \boldsymbol{x} > 0$$

  Costruisco il vettore $\boldsymbol{y} = S^T \boldsymbol{x}$, di conseguenza $\boldsymbol{y}^T = \boldsymbol{x}^T S$. Quindi posso riscrivere $\boldsymbol{y}^T \boldsymbol{y} > 0$. Svolgendo il prodotto riga-colonna vale:

  $$\sum_{i=1}^{n} y_i^2 > 0$$

  Questa somma non può essere zero in quanto per come è costruito $\boldsymbol{y}$, per ipotesi S è non singolare e dato che $S = S^T$, anche S^T è non singolare, quindi S^T non annulla il prodotto. Rimane da controllare il vettore $\boldsymbol{x}$, ma per costruzione $\boldsymbol{x} \neq \boldsymbol{0}$, quindi la somma è positiva e questo termina la prova che $A = SS^T$ è sdp.

□

Observation 3.2.2. *Posso utilizzare l'asserto del precedente esercizio per costruire una matrice sdp partendo da una matrice non singolare.*

Exercise 3.2.9 (3.13a)**.** *Se* $A \in \mathbb{R}^{n\times n}$ *allora* $A = \frac{1}{2}(A + A^T) + \frac{1}{2}(A - A^T) = A_s + A_a$

Dimostrazione.

$$\frac{1}{2}\left(\begin{bmatrix} a_{11} & \cdots & \cdots & \cdots & a_{1n} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix} + \begin{bmatrix} a_{11} & \cdots & \cdots & \cdots & a_{n1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{1n} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix}\right) =$$

$$= \frac{1}{2}\begin{bmatrix} 2a_{11} & \cdots & \cdots & \cdots & a_{1n} + a_{n1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1} + a_{1n} & \cdots & \cdots & \cdots & 2a_{nn} \end{bmatrix} = A^s$$

La matrice A_s è una matrice simmetrica.

Analogamente per la differenza:

$$\frac{1}{2}\left(\begin{bmatrix} a_{11} & \cdots & \cdots & \cdots & a_{1n} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix} - \begin{bmatrix} a_{11} & \cdots & \cdots & \cdots & a_{n1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{1n} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix}\right) =$$

$$= \frac{1}{2}\begin{bmatrix} 0 & \cdots & \cdots & \cdots & a_{1n} - a_{n1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1} - a_{1n} & \cdots & \cdots & \cdots & 0 \end{bmatrix} = A^a$$

Verifico che $A_s + A_a = A$:

$$A_s + A_a = \frac{1}{2}\begin{bmatrix} 2a_{11} & \cdots & \cdots & \cdots & a_{1n} + a_{n1} + a_{1n} - a_{n1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1} + a_{1n} + a_{n1} - a_{1n} & \cdots & \cdots & \cdots & 2a_{nn} \end{bmatrix} =$$

$$= \frac{1}{2}\begin{bmatrix} 2a_{11} & \cdots & \cdots & \cdots & 2a_{1n} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ 2a_{n1} & \cdots & \cdots & \cdots & 2a_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & \cdots & \cdots & a_{1n} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix} = A$$

In generale vale $a'_{ij} = a_{ij} + a_{ji} + a_{ij} - a_{ji} = 2a_{ij}$. L'uguaglianza è verificata e questo termina la prova. □

Exercise 3.2.10 (3.13b). *Se* $A \in \mathbb{R}^{n\times n}$ *e* $\forall \mathbf{x} \in \mathbb{R}^n$ *allora* $\mathbf{x}^T A\mathbf{x} = \mathbf{x}^T A_s \mathbf{x}$

Dimostrazione. Per la parte 3.13a vale che $A = A_s + A_a$, quindi posso sostituire ed applicare la proprietà distributiva del prodotto:

$$\mathbf{x}^T A\mathbf{x} = \mathbf{x}^T (A_s + A_a)\mathbf{x} = \mathbf{x}^T A_s \mathbf{x} + \mathbf{x}^T A_a \mathbf{x}$$

Posso ridurre l'argomento che si vuole dimostrare ($\mathbf{x}^T A\mathbf{x} = \mathbf{x}^T A_s \mathbf{x}$), considerandolo vero, a dimostrare che necessariamente deve valere $0 = \mathbf{x}^T A_a \mathbf{x}$.

Per definizione di A_a vale $\frac{1}{2}(A - A^T) = A_a$, che posso rappresentare:

$$\frac{1}{2}\begin{bmatrix} 0 & a_{12}-a_{21} & \cdots & \cdots & a_{1n}-a_{n1} \\ a_{21}-a_{12} & 0 & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1}-a_{1n} & \cdots & \cdots & \cdots & 0 \end{bmatrix} = A^a$$

Fisso un vettore $\mathbf{x} \in \mathbb{R}^n$ e rappresento il prodotto $\mathbf{x}^T A_a \mathbf{x}$:

$$\begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix} \begin{bmatrix} 0 & a_{12}-a_{21} & \cdots & \cdots & a_{1n}-a_{n1} \\ a_{21}-a_{12} & 0 & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1}-a_{1n} & \cdots & \cdots & \cdots & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = 2A^a$$

Sviluppo il prodotto matrice-vettore colonna:

$$\begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix} \begin{bmatrix} 0x_1 + (a_{12}-a_{21})x_2 + (a_{13}-a_{31})x_3 + \cdots + (a_{1n}-a_{n1})x_n \\ (a_{21}-a_{12})x_1 + 0x_2 + (a_{23}-a_{32})x_3 + \cdots + (a_{2n}-a_{n2})x_n \\ \vdots \\ (a_{n1}-a_{1n})x_1 + (a_{n2}-a_{2n})x_2 + (a_{n3}-a_{3n})x_3 + \cdots + 0x_n \end{bmatrix} = 2A^a$$

Sviluppo il prodotto vettore riga-vettore colonna ottenendo uno scalare:

$$\begin{gathered} 0x_1x_1 + (a_{12}-a_{21})x_2x_1 + (a_{13}-a_{31})x_3x_1 + \cdots + (a_{1n}-a_{n1})x_nx_1 + \\ +(a_{21}-a_{12})x_1x_2 + 0x_2x_2 + (a_{23}-a_{32})x_3x_2 + \cdots + (a_{2n}-a_{n2})x_nx_2 + \\ \vdots \\ +(a_{n1}-a_{1n})x_1x_n + (a_{n2}-a_{2n})x_2x_n + (a_{n3}-a_{3n})x_3x_n + \cdots + 0x_nx_n \end{gathered} = 2A^a$$

Omettendo gli elementi che contengono i fattori $ax_ix_i = 0$ ottengo:

$$0 = \begin{gathered} (a_{12}-a_{21})x_2x_1 + (a_{13}-a_{31})x_3x_1 + \cdots + (a_{1n}-a_{n1})x_nx_1 + \\ +(a_{21}-a_{12})x_1x_2 + (a_{23}-a_{32})x_3x_2 + \cdots + (a_{2n}-a_{n2})x_nx_2 + \\ \vdots \\ +(a_{n1}-a_{1n})x_1x_n + (a_{n2}-a_{2n})x_2x_n + (a_{n3}-a_{3n})x_3x_n + \cdots + (a_{n,n-1}-a_{n-1,n})x_{n-1}x_n \end{gathered} = 2A^a$$

Ma $(a_{ij}-a_{ji})x_jx_i = -(a_{ji}-a_{ij})x_ix_j$ quindi tutti gli elementi della somma precedente si eliminano a coppie. Per completezza dimostro che il numero di elementi della precedente somma è pari.

Gli elementi della penultima matrice (comprendendo anche i prodotti nulli) è pari al prodotto cartesiano n^2, sottraggo gli elementi nulli, ovvero la diagonale, quindi gli elementi non nulli sono $n^2 - n$. Devo dimostrare quindi $n^2 - n = 2k$, con $k \in \mathbb{N}$. Divido per 2 entrambi i membri ottenendo $\frac{n(n-1)}{2} = k$. Il primo membro è la somma dei primi $n-1$ numeri naturali, quindi $\sum_{i=1}^{n-1} i = k$. Ma dato che l'indice superiore della somma è finito ($n - 1 < \infty$) allora la somma ammette un valore finito $k \in \mathbb{N}$, l'uguaglianza è verificata e questo conclude la prova. □

Exercise 3.2.11 (3.16, 3.17)**.** *Per i testi degli esercizi consultare il libro di testo.*

Vedere il codice LDLmethod.

Exercise 3.2.12 (3.18)**.** *Per i testi degli esercizi consultare il libro di testo.*

```
octave:43> A = [
> 1 1 1 1
> 1 2 2 2
> 1 2 1 1
> 1 2 1 2];
octave:45> [L,U,xs] = LDLmethod(A, [1;1;1;1])
error: A non sdp.
```

Exercise 3.2.13. *Usare le fattorizzazioni* LU, LDL^T *alla matrice:*

$$\begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

```
octave:49> A = [1 -1 0 0; -1 2 -1 0; 0 -1 2 -1; 0 0 -1 2]
A =
   1  -1   0   0
  -1   2  -1   0
   0  -1   2  -1
   0   0  -1   2
octave:50> [L,U,xs] = LUmethod(A, [1;1;1;1])
L =
   1   0   0   0
  -1   1   0   0
   0  -1   1   0
   0   0  -1   1
U =
```

```
    1   -1    0    0
    0    1   -1    0
    0    0    1   -1
    0    0    0    1
xs =
   10
    9
    7
    4
octave:51> [L,D,xs] = LDLmethod(A, [1;1;1;1])
L =
    1    0    0    0
   -1    1    0    0
    0   -1    1    0
    0    0   -1    1
D =
    1    0    0    0
    0    1    0    0
    0    0    1    0
    0    0    0    1
xs =
   10
    9
    7
    4
```

Exercise 3.2.14 (3.19). $a_{ii}^{(i)} \in A^{(i+1)} \in \mathbb{R}^{n\times n}$, $a_{ii}^{(i)} \neq 0, \forall i \in 1,\dots,n$ *se* $\det(A) \neq 0$

Dimostrazione. Rappresento la matrice $A^{(i+1)}$:

$$A^{(i+1)} = \begin{bmatrix} a_{k_1 1}^{(1)} & \cdots & \cdots & \cdots & \cdots & a_{k_1 n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & & a_{k_{i-1},i-1}^{(i-1)} & \cdots & \cdots & a_{k_{i-1},n}^{(i-1)} \\ \vdots & & 0 & a_{ii}^{(i)} & \cdots & a_{in}^{(i)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{ni}^{(i)} & \cdots & a_{nn}^{(i)} \end{bmatrix} = \{ \text{ ragionando a blocchi } \} = \left[\begin{array}{c|c} A_{i-1} & B \\ \hline 0 & C \end{array}\right]$$

Adesso posso applicare la funzione ad entrambi i membri più esterni dell'uguaglianza, ottengo quindi

$$\det(A^{(i+1)}) = \det(A_{i-1})\det(C) - 0\det(B) = \det(A_{i-1})\det(C)$$

Dato che ho potuto raggiungere il passo i-esimo della fattorizzazione, allora significa che $a_{k_j j} \neq 0$, con $j \in \{1, \ldots, i-1\}$. Osservo che A_{i-1} è tringolare inferiore quindi, per l'esercizio 3.1.5, vale $\det(A_{i-1}) = a^{(1)}_{k_1 1} \cdots a^{(i-1)}_{k_{i-1} i-1} \neq 0$. Ma per ipotesi A è non singolare quindi vale $\det(A^{(i+1)}) \neq 0$, in quanto sono state i trasformazioni reversibili (basta considerare le L^{-1}), ovvero rendono le due matrici equivalenti. Per questo si ha necessariamente che $\det(C) \neq 0$, ovvero lo colonne della matrice C sono linearmente indipendenti e quindi

$$a^{(i)}_{k_i i} = \max_{k \geqslant i} |a^{(i)}_{ki}| \neq 0$$

in altre parole la colonna $A^{(i+1)} \boldsymbol{e}_i \neq \mathbf{0}$. □

3.3 AFTER PARTIAL PIVOTING

Exercise 3.3.1 (3.21, 3.22)**.** *Per i testi degli esercizi consultare il libro di testo.*

Vedere il codice PALUmethod.

Exercise 3.3.2 (3.23)**.** *Per il testo dell'esercizio consultare il libro di testo.*

- applicazione al sistema dell'esercizio 3.3.3: si osserva che usando il pivoting vengono trovate le soluzioni esatte.

```
octave:25> A
A =
   2.22044604925031e-16   1.00000000000000e+00
   1.00000000000000e+00   0.00000000000000e+00
octave:26> b
b =
   1.00000000000000e+00
   2.50000000000000e-01
octave:27> [L, U, p, unknowns] = PALUmethod(A, b)
L =
   1.00000000000000e+00   0.00000000000000e+00
   2.22044604925031e-16   1.00000000000000e+00
U =
   1.00000000000000e+00   0.00000000000000e+00
   0.00000000000000e+00   1.00000000000000e+00
p =
   2.00000000000000e+00   1.00000000000000e+00
unknowns =
   2.50000000000000e-01
   1.00000000000000e+00
```

- applicazione al primo sistema dell'esercizio 3.3.3:

```
octave:42> A = [
1 0 0 0 0 0 0 0 0 0;
100 1 0 0 0 0 0 0 0 0;
0 100 1 0 0 0 0 0 0 0;
0 0 100 1 0 0 0 0 0 0;
0 0 0 100 1 0 0 0 0 0;
0 0 0 0 100 1 0 0 0 0;
0 0 0 0 0 100 1 0 0 0;
0 0 0 0 0 0 100 1 0 0;
0 0 0 0 0 0 0 100 1 0;
0 0 0 0 0 0 0 0 100 1];
octave:43> x = [1 101*ones(1,9)]';
octave:45> [L, U, p, unknowns] = PALUmethod(A, x);
octave:46> format
octave:47> p
p =
    2    3    4    5    6    7    8    9   10    1
octave:50> format long e
octave:51> unknowns'
ans =
 Columns 1 through 4:
   1.00000000000000e+00   1.00000000000000e+00
      9.99999999999925e-01   1.00000000000755e+00
 Columns 5 through 8:
   9.99999999245411e-01   1.00000007545890e+00
      9.99992454110141e-01   1.00075458898588e+00
 Columns 9 and 10:
   9.24541101412386e-01   8.54588985876136e+00
```

- applicazione al secondo sistema dell'esercizio 3.3.3:

```
octave:42> A = [
1 0 0 0 0 0 0 0 0 0;
100 1 0 0 0 0 0 0 0 0;
0 100 1 0 0 0 0 0 0 0;
0 0 100 1 0 0 0 0 0 0;
0 0 0 100 1 0 0 0 0 0;
0 0 0 0 100 1 0 0 0 0;
0 0 0 0 0 100 1 0 0 0;
0 0 0 0 0 0 100 1 0 0;
0 0 0 0 0 0 0 100 1 0;
0 0 0 0 0 0 0 0 100 1];
```

```
octave:43> x = [1 101*ones(1,9)]';
octave:54> y = .1*x;
octave:55> [L, U, p, unknowns] = PALUmethod(A, y);
octave:56> format
octave:57> p
p =
    2    3    4    5    6    7    8    9   10    1
octave:58> format long e
octave:59> unknowns'
ans =
 Columns 1 through 4:
   1.000000000000000e-01   1.000000000000001e-01
      9.999999999985500e-02   1.000000000145040e-01
 Columns 5 through 8:
   9.999999854963830e-02   1.000001450361730e-01
      9.998549638267910e-02   1.014503617320920e-01
 Columns 9 and 10:
  -4.503617320921390e-02   1.460361732092140e+01
```

- applicazione al sistema $B\mathbf{x} = \mathbf{f}$:

```
octave:59> format
octave:60> B = [0 1 1; 1 0 1; 2 3 4]
B =
   0   1   1
   1   0   1
   2   3   4
octave:61> f = [1;0;3];
octave:62> [L, U, p, unknowns] = PALUmethod(B, f);
octave:63> L
L =

   1.000000000000000e+00   0.000000000000000e+00
      0.000000000000000e+00
   5.000000000000000e-01   1.000000000000000e+00
      0.000000000000000e+00
   0.000000000000000e+00  -6.666666666666670e-01
      1.000000000000000e+00

octave:64> U
U =
   2.000000000000000e+00   3.000000000000000e+00
      4.000000000000000e+00
```

```
    0.00000000000000e+00   -1.50000000000000e+00
       -1.00000000000000e+00
    0.00000000000000e+00    0.00000000000000e+00
       3.33333333333333e-01
octave:66> p
p =
   3   2   1
octave:67> format long e
octave:68> unknowns
unknowns =
    0.00000000000000e+00
    1.00000000000000e+00
    0.00000000000000e+00
octave:72> B*unknowns - f
ans =
    0
    0
    0
```

Exercise 3.3.3 (3.24). *Per il testo dell'esercizio consultare il libro di testo.*

```
octave:8> A = [eps 1; 1 0], b = [1; 1/4]
A =
   2.22044604925031e-16   1.00000000000000e+00
   1.00000000000000e+00   0.00000000000000e+00
b =
   1.00000000000000e+00
   2.50000000000000e-01
octave:9> A\b
ans =
   2.50000000000000e-01
   1.00000000000000e+00
octave:10> L = [1 0; 1/eps 1], U = [eps 1; 0 -1/eps]
L =
   1.00000000000000e+00   0.00000000000000e+00
   4.50359962737050e+15   1.00000000000000e+00
U =
   2.22044604925031e-16   1.00000000000000e+00
   0.00000000000000e+00  -4.50359962737050e+15
octave:11> (L*U) - A
ans =
   0.00000000000000e+00   0.00000000000000e+00
```

```
   0.00000000000000e+00   0.00000000000000e+00
octave:12> U\(L\b)
ans =
   0.00000000000000e+00
   1.00000000000000e+00
```

La riga *octave:9* restituisce la soluzione esatta del sistema, ovvero il vettore $\mathbf{x} = \begin{bmatrix} .25 \\ 1 \end{bmatrix}$.

La riga *octave:11* dimostra che $A = LU$ quindi $0 = LU - A$, le matrici di fattorizzazione sono corrette.

La riga *octave:12* dimostra che il metodo di fattorizzazione LU senza pivoting risulta essere mal condizionato, infatti la prima componente $\tilde{x}_1 = 0 \neq .25 = x_1$.

Exercise 3.3.4 (3.25). *Per il testo dell'esercizio consultare il libro di testo.*

Per calcolare $k_\infty(A)$ è necessario calcolare l'inversa:

$$\begin{bmatrix} 1 & & & & & \\ -100 & 1 & & & & \\ 100^2 & -100 & 1 & & & \\ -100^4 & 100^2 & -100 & 1 & & \\ \vdots & & & & \ddots & \\ -100^{16} & 100^{14} & -100^{12} & \cdots & \cdots & 1 \end{bmatrix} = A^{-1}$$

Questo l'output del codice:

```
octave:35> cond(A, Inf)
warning: inverse: matrix singular to machine precision,
   rcond = 9.80198e-21
ans =  1.0202e+20
```

Exercise 3.3.5 (3.26). *Per il testo dell'esercizio consultare il libro di testo.*

Questo il codice che verifica che i vettori $\mathbf{x}, \mathbf{y}$ sono soluzioni rispettivamente delle equazioni $A\mathbf{x} = \mathbf{b}, A\mathbf{y} = \mathbf{c}$:

```
octave:41> A
A =

     1     0     0     0     0     0     0     0     0
           0
   100     1     0     0     0     0     0     0     0
         0
     0   100     1     0     0     0     0     0     0
           0
```

```
      0      0    100      1      0      0      0      0      0
             0
      0      0      0    100      1      0      0      0      0
             0
      0      0      0      0    100      1      0      0      0
             0
      0      0      0      0      0    100      1      0      0
             0
      0      0      0      0      0      0    100      1      0
             0
      0      0      0      0      0      0      0    100      1
             0
      0      0      0      0      0      0      0      0    100
             1

octave:42> b = [1;101;101;101;101;101;101;101;101;101];
octave:43> x = [1;1;1;1;1;1;1;1;1;1];
octave:44> A*x - b
ans =
   0
   0
   0
   0
   0
   0
   0
   0
   0
   0
octave:45> c = .1*b;
octave:46> y = [.1;.1;.1;.1;.1;.1;.1;.1;.1;.1];
octave:47> A*y - c
ans =
   0
   0
   0
   0
   0
   0
   0
   0
   0
   0
```

Implemento le istruzioni:

```
octave:50> b = [1  101*ones(1,9)]';
octave:51> x(1) = b(1);
octave:52> for i=2:10, x(i) = b(i) - 100*x(i-1); end
octave:53> x = x(:)
x =
   1
   1
   1
   1
   1
   1
   1
   1
   1
   1
octave:54> c = .1*[1  101*ones(1,9)]';
octave:55> y(1) = c(1);
octave:56> for i=2:10, y(i) = c(i) - 100*y(i-1); end
octave:57> y = y(:)
y =
    0.100000
    0.100000
    0.100000
    0.100000
    0.100000
    0.100000
    0.099986
    0.101407
   -0.040702
   14.170153
octave:62> (y-yexact) ./ yexact
ans =
     0.00000
     0.00000
    -0.00000
     0.00000
    -0.00000
     0.00000
    -0.00014
     0.01407
```

```
  −1.40702
 140.70153
```

Si osserva che il vettore $\mathbf{y}$, soluzione esatta del secondo sistema, non è uguale al vettore approssimato dal secondo set di istruzioni: il comando *octave:62* calcola il vettore degli errori relativi che si commettono e si nota che per l'ultima componente si ha un errore molto significativo. Questo è dovuto dal fatto che le componenti del vettore dei termini noti $\mathbf{c}$ non è possibile rappresentarle correttamente in macchina (0.1 ha rappresentazione infinita periodica in binario). Questi errori di rappresentazione si sommano nelle 9 iterazioni che vengono fatte dal ciclo for, risultando significative per la decima componente.

Exercise 3.3.6 (3.28)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Si vuole dimostrare:

$$\beta = -\frac{v_1}{\alpha} = \frac{2}{\hat{\mathbf{v}}^T\hat{\mathbf{v}}}, \quad \text{con} \quad \hat{\mathbf{v}} = \frac{\mathbf{v}}{v_1}$$

- Verifico la concordanza dei segni:

  $$\frac{2}{\hat{\mathbf{v}}^T\hat{\mathbf{v}}} > 0$$

  $\hat{\mathbf{v}}^T\hat{\mathbf{v}} > 0$ per definizione di norma euclidea. Deve quindi valere:

  $$\frac{v_1}{\alpha} < 0$$

  posso sostituire $v_1 = z_1 - \alpha$ e $\alpha = -\text{sign}(z_1)\|\mathbf{z}\|_2$ e $\text{sign}(a) = \begin{cases} 1 & \text{if} \quad a \geqslant 0 \\ -1 & \text{if} \quad a < 0 \end{cases}$

  ottenendo:

  $$\frac{z_1}{\alpha} - 1 < 0 \Rightarrow \frac{z_1}{\alpha} < 1 \Rightarrow \frac{\text{sign}(z_1)|z_1|}{-\text{sign}(z_1)\|\mathbf{z}\|_2} < 1 \Rightarrow \frac{|z_1|}{\|\mathbf{z}\|_2} > -1$$

  Ma questo è vero in quanto $|z_1| > 0$ per definizione di $|\cdot|$ e $\|\mathbf{z}\|_2 > 0$ per definizione di norma euclidea, quindi $\frac{|z_1|}{\|\mathbf{z}\|_2} > 0 > -1$. I segni concordano.

- dimostro l'uguaglianza dei due membri:

  Per il primo membro vale:

$$-\frac{v_1}{\alpha} = -\frac{\text{sign}(z_1)|z_1| - (-\text{sign}(z_1)\|\mathbf{z}\|_2)}{-\text{sign}(z_1)\|\mathbf{z}\|_2} = -\frac{\text{sign}(z_1)(|z_1| + \|\mathbf{z}\|_2)}{-\text{sign}(z_1)\|\mathbf{z}\|_2} = \frac{|z_1| + \|\mathbf{z}\|_2}{\|\mathbf{z}\|_2}$$

Per il secondo membro vale:

$$\hat{\mathbf{v}}^T\hat{\mathbf{v}} = \frac{\mathbf{v}^T\mathbf{v}}{v_1^2} = \frac{(\mathbf{z}^T - \alpha\mathbf{e}_i^T)(\mathbf{z} - \alpha\mathbf{e}_i)}{v_1^2} = \frac{\mathbf{z}^T\mathbf{z} - \alpha\mathbf{z}^T\mathbf{e}_i - \alpha\mathbf{e}_i^T\mathbf{z} + \alpha^2\mathbf{e}_i^T\mathbf{e}_i}{v_1^2} =$$
$$= \frac{\alpha^2 - \alpha z_1 - \alpha z_1 + \alpha^2}{v_1^2} = \frac{2\|\mathbf{z}\|_2^2 - 2\alpha z_1}{v_1^2} = \frac{2\|\mathbf{z}\|_2^2 - 2(-\text{sign}(z_1)\|\mathbf{z}\|_2\text{sign}(z_1)|z_1|)}{v_1^2} =$$
$$= \frac{2\|\mathbf{z}\|_2^2 + 2\|\mathbf{z}\|_2|z_1|}{v_1^2} = \frac{2\|\mathbf{z}\|_2(\|\mathbf{z}\|_2 + |z_1|)}{v_1^2}$$

Ma vale che $v_1 = z_1 - \alpha = \text{sign}(z_1) - (-\text{sign}(z_1)\|\mathbf{z}\|_2) = \text{sign}(z_1)(|z_1| + \|\mathbf{z}\|_2)$, per cui vale
$v_1^2 = (|z_1| + \|\mathbf{z}\|_2)^2$, quindi posso sostituire e semplificare ottenendo:

$$\hat{\mathbf{v}}^T\hat{\mathbf{v}} = \frac{2\|\mathbf{z}\|_2}{\|\mathbf{z}\|_2 + |z_1|}$$

Posso invertire potendo riscrivere il secondo membro come:

$$\frac{2}{\hat{\mathbf{v}}^T\hat{\mathbf{v}}} = \frac{\|\mathbf{z}\|_2 + |z_1|}{\|\mathbf{z}\|_2} = -\frac{v_1}{\alpha} = \beta$$

e questo termina la prova.

□

Exercise 3.3.7 (3.29, 3.30). *Per i testi degli esercizi consultare il libro di testo.*

Vedere il codice QRmethod.

Exercise 3.3.8 (3.31). *Per il testo dell'esercizio consultare il libro di testo.*

Questa l'implementazione:

```
octave:37> A = [3 2 1; 1 2 3; 1 2 1; 2 1 2]
A =
   3   2   1
   1   2   3
   1   2   1
   2   1   2
octave:38> b = [10*ones(1,4)]'
b =
   10
   10
   10
   10
octave:39> [houseHolderVectors, Rhat, R, Q, g1, g2,
   unknowns, residue] =QRmethod(A,b)
```

```
houseHolderVectors =
   1.00000   0.00000   0.00000
   0.14550   1.00000   0.00000
   0.14550   0.40559   1.00000
   0.29099  -0.15589  -0.47436
Rhat =
  -3.87298  -3.09839  -2.84019
   0.00000  -1.84391  -1.73544
   0.00000   0.00000   1.98030
   0.00000   0.00000   0.00000
R =
  -3.87298  -3.09839  -2.84019
   0.00000  -1.84391  -1.73544
   0.00000   0.00000   1.98030
Q =
  -0.77460   0.21693  -0.41586  -0.42426
  -0.25820  -0.65079   0.57429  -0.42426
  -0.25820  -0.65079  -0.43566   0.56569
  -0.51640   0.32540   0.55448   0.56569
g1 =
  -18.0739
   -7.5926
    2.7724
g2 =  2.8284
unknowns =
   1.4000
   2.8000
   1.4000
residue =
   1.2000
   1.2000
  -1.6000
  -1.6000
octave:40> R*unknowns -g1
ans =
   0.0000e+00
  -8.8818e-16
   0.0000e+00
octave:41> norm(R*unknowns -g1 )^2 + norm(g2 )^2
ans =  8.0000
octave:42> r = Q*Rhat*unknowns - b
r =
   1.2000
```

```
    1.2000
   -1.6000
   -1.6000
octave:43> norm(r)^2
ans =  8.0000
octave:44> Q*Rhat
ans =
    3.00000   2.00000   1.00000
    1.00000   2.00000   3.00000
    1.00000   2.00000   1.00000
    2.00000   1.00000   2.00000
octave:45> r/norm(r)
ans =
    0.42426
    0.42426
   -0.56569
   -0.56569
```

Exercise 3.3.9 (3.32). *Per il testo dell'esercizio consultare il libro di testo.*

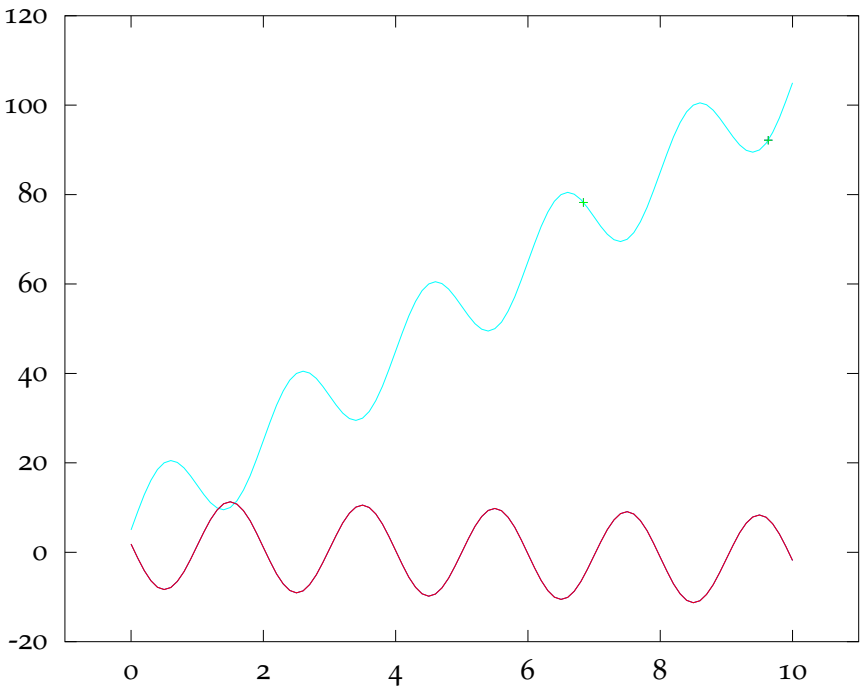
Implemento lanciando la funzione functionExercise332:

```
octave:13> [unknowns, givenUnknowns] = functionExercise332
    (0)
unknowns =
 Columns 1 through 3:
   9.63228005156231e+00   9.96322800515623e+00
      9.98161400257811e+00
   6.83859974218842e+00   5.18385997421884e+00
      5.09192998710942e+00
 Columns 4 through 6:
   9.99632280051562e+00   9.99816140025781e+00
      9.99963228005156e+00
   5.01838599742188e+00   5.00919299871094e+00
      5.00183859974219e+00
 Columns 7 and 8:
   9.99981614002578e+00   9.99996322800516e+00
   5.00091929987110e+00   5.00018385997422e+00

givenUnknowns =
 Columns 1 through 3:
   9.63228005156232e+00   9.96322800515623e+00
      9.98161400257811e+00
```

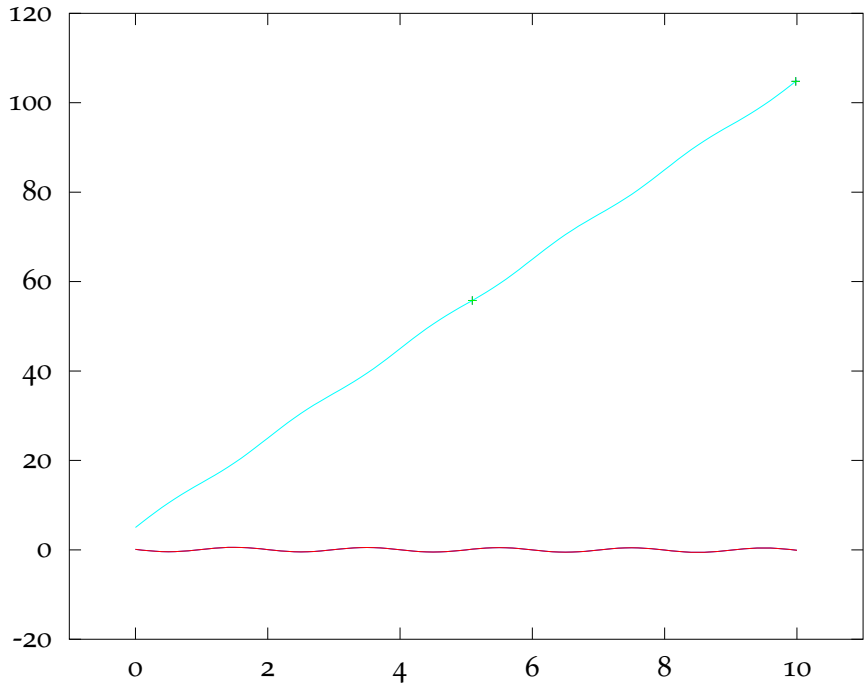
```
   6.83859974218843e+00   5.18385997421885e+00
      5.09192998710943e+00
Columns 4 through 6:
   9.99632280051562e+00   9.99816140025781e+00
      9.99963228005156e+00
   5.01838599742189e+00   5.00919299871094e+00
      5.00183859974219e+00
Columns 7 and 8:
   9.99981614002578e+00   9.99996322800515e+00
   5.00091929987110e+00   5.00018385997422e+00
```

Riporto quattro grafici relativi a quattro diversi valori di gamma. I grafici in blu sono relativi alla soluzione calcolata con l'implementazione QRmethod, mentre i grafici verdi (a obiettivo) e rossi sono invece relativi al codice dato nell'esercizio.

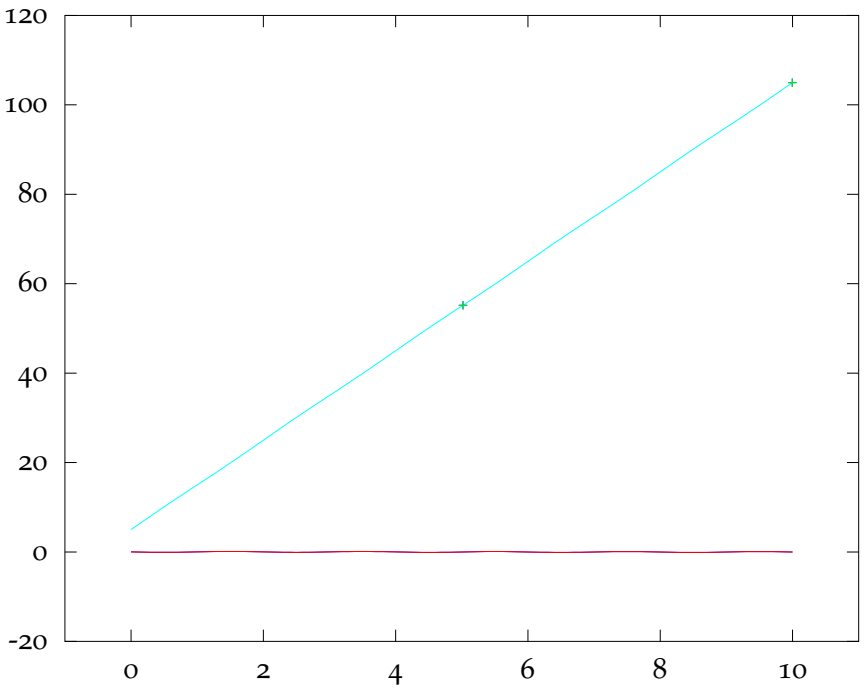
gamma = 10:



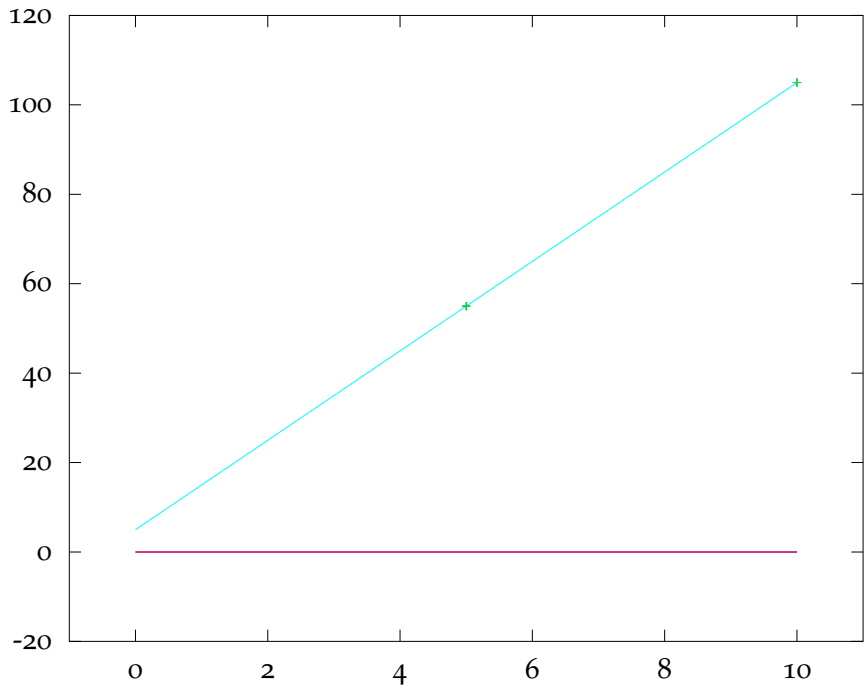
gamma = 5e − 1:



$gamma = 1e-1$:



$gamma = 1e-3$:



Per $gamma \to \infty$ si raggiunge la soluzione ottima del sistema $A\mathbf{x} = \mathbf{b} + \mathbf{r}$ con $\mathbf{r} = \mathbf{0}$.

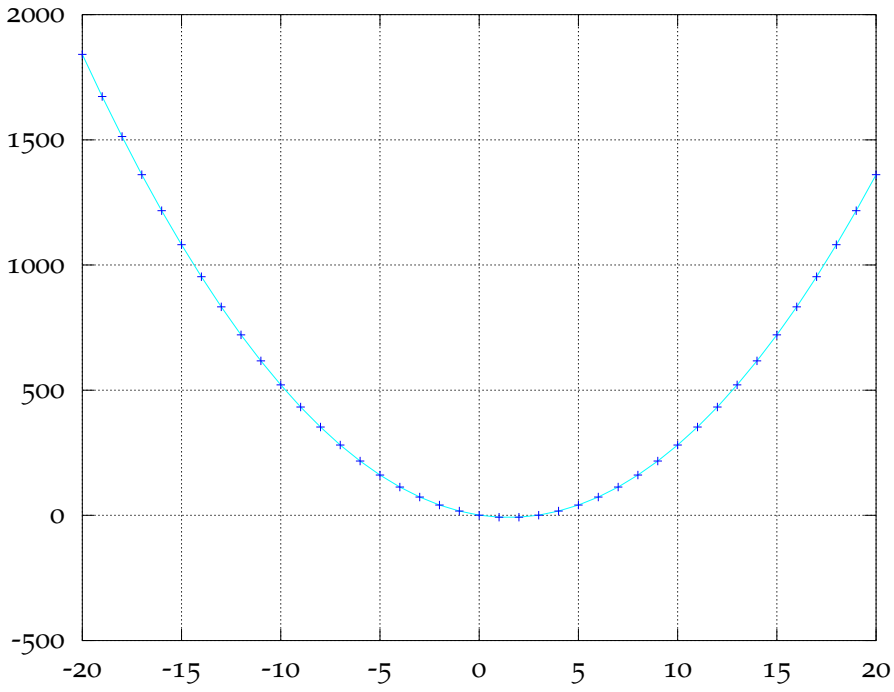
4

APPROSSIMAZIONE FUNZIONI

4.1 INTERPOLAZIONE POLINOMIALE

Exercise 4.1.1 (4.1). *Trovare un polinomio* $p(x)$ *che interpola la funzione* $f(x) = 4x^2 - 12x + 1$ *nei punti di ascissa* $x_i = i$, *con* $i \in \{0, \ldots, 4\}$.

Per l'implementazione vedere il codice Exercise 4.1 on textbook e per la sua esecuzione lanciare lo script Script for Exercise 4.1 on textbook.



In cyan è reppresentata la curva della funzione f, mentre con i simboli + sono rappresentati i punti interpolati dal polinomio p.

Exercise 4.1.2 (4.2). *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. L'algoritmo riportato implementa questo schema (utilizzo gli indici nella notazione usata nella formulazione matematica, quindi sono zero-based):

$$\begin{aligned} p^{(0)}(x) &= a_n \\ p^{(i+1)}(x) &= p^{(i)}x + a_{n-i} \end{aligned}$$

con $p^{(i)}$ indico il valore di p all'i-esimo passo dei esecuzione. Se considero il valore di p ad un generico passo i di esecuzione si osserva che ha questa struttura:

$$\begin{aligned} p^{(i)}(x) &= (a_n x^{i-1} + a_{n-1}x^{i-2} + \ldots + a_{n-i+1})x + a_{n-i} \\ &= a_n x^i + a_{n-1}x^{i-1} + \ldots + a_{n-i+1}x + a_{n-i} \end{aligned}$$

Il polinomio $p^{(i)}$ è di grado i, quindi saturando l'indice i arrivando a calcolare p^n, si ottiene il polinomio:

$$p(x) = p^{(n)}(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$

Ovvero quello che si chiede nel problema. □

Exercise 4.1.3 (4.3)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Dimostro i punti del *Lemma 4.1*:

1. questa è la forma di un generico polinomio k della base di Lagrange di grado n:

$$L_{kn}(x) = \frac{(x-x_0)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}$$

   Se valuto il polinomio $L_{kn}(x_i)$ in una generica ascissa di interpolazione x_i posso avere due casi:

   - se $i \neq k$, allora $x_i \in \{x_0, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n\}$ ma questo fa si che uno dei fattori del numeratore abbia la forma $(x_i - x_i) = 0$ che annulla la produttoria:

$$L_{kn}(x_{i\neq k}) = \frac{(x_i-x_0)\cdots(x_i-x_{k-1})(x_i-x_{k+1})\cdots(x_i-x_i)\cdots(x_i-x_n)}{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)} = 0$$

     Osservazione: ho inserito il termine che si annulla più a destra, ma il posizionamento è da ritenersi non influente in quanto per il prodotto l'ordine dei fattori non influisce ed inoltre è dipendente da i.

   - se $i = k$ allora ottengo

$$L_{kn}(x_{i=k}) = \frac{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)} = 1$$

2. i polinomi della base di Lagrange hanno grado esatto n in quanto per definizione del generico polinomio k si ha:

$$L_{kn}(x) = \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j}$$

l'indice della produttoria "corre" da 0 a n, indicizzando $n+1$ posizioni, ma la k-esima posizione viene saltata, quindi vengono eseguiti n prodotti, di conseguenza il termine di grado massimo è x^n. Inoltre posso dividera la produttoria in questo modo:

$$L_{kn}(x) = \frac{\prod_{j=0;j\neq k}^{n} x - x_j}{\prod_{j=0;j\neq k}^{n} x_k - x_j}$$

la quantità al denominatore non dipende dall'ascissa x in input, quindi si può sviluppare il prodotto al numeratore, ottenendo un polinomio di grado n, con coefficiente uguale a 1, poi moltiplicare per la costante al denominatore, ottenendo il termine di grado massimo:

$$a_n x^n = \frac{1}{\prod_{j=0;j\neq k}^{n} x_k - x_j} x^n$$

3. per dimostrare che i polinomi $L_{kn}(x)$ formano una base per $\prod_n$ allora devo dimostrare:

$$\alpha_0 L_{0n}(x) + \ldots + \alpha_n L_{nn}(x) = 0 \Leftrightarrow \alpha_0 = \ldots = \alpha_n = 0$$

Il verso $\Leftarrow$ è ovvio. Per il verso $\Rightarrow$ invece valuto la combinazione lineare dei polinomi in una generica ascissa di interpolazione x_i ottenendo:

$$\alpha_0 L_{0n}(x_i) + \ldots + \alpha_n L_{nn}(x_i) = 0$$

Per il punto 1 i termini $\alpha_k L_{kn}(x_i) = 0$ se $i \neq k$, quindi rimane il solo elemento $\alpha_i L_{in}(x_i)$. Ma, sempre per il punto 1 vale $L_{in}(x_i) = 1$, quindi:

$$0 = \alpha_i L_{in}(x_i) = \alpha_i$$

Ripetendo questo ragionamento per ogni ascissa di interpolazione si ottiene $\alpha_0 = \ldots = \alpha_n = 0$ e il verso risulta dimostrato.

□

Exercise 4.1.4 (4.4)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Dimostro i punti del *Lemma 4.2*:

1. $w_{k+1}(x) = (x - x_k)w_k(x) = (x - x_k)(x - x_{k-1})w_{k-1}(x) = \ldots = (x - x_k)\cdots(x - x_0) = \prod_{j=0}^{k}(x - x_j)$

2. come si vede dal punto 1, valutare $w_k(x)$ richiede eseguire k prodotti, per cui si ottiene un polinomio di grado k, quindi $w_k(x) \in \prod_k$

3. valutare il polinomio w_k in una delle ascisse di interpolazione x_i con $i < k$, ovvero $w_k(x_i)$ si ha che uno dei fattori di $w_k(x_i) = (x_i - x_{k-1}) \cdots (x_i - x_i) \cdots (x_i - x_0)$ si annulli, annullando di conseguenza tutta la produttoria, quindi vale $i < k \Rightarrow w_k(x_i) = 0$.

4. per dimostrare che i polinomi $w_k(x)$ formano una base per $\prod_k$ allora devo dimostrare:

   $$\alpha_0 w_0(x) + \ldots + \alpha_k w_k(x) = 0 \Leftrightarrow \alpha_0 = \ldots = \alpha_k = 0$$

   Sviluppo per chiarezza la precedente combinazione lineare:

   $$\alpha_0 + \alpha_1(x - x_0) + \alpha_2(x - x_1)(x - x_0) \ldots + \alpha_k(x - x_{k-1}) \cdots (x - x_0) = 0$$

   Il verso $\Leftarrow$ è ovvio. Per il verso $\Rightarrow$ invece valuto la combinazione lineare dei polinomi in una generica ascissa di interpolazione $x_i \in \{x_0, \ldots, x_k\}$ ottenendo:

   $$\alpha_0 w_0(x_i) + \ldots + \alpha_k w_k(x_i) = 0$$

   Posso ragionare distinguendo due casi:

   - se $i = k$ allora $\forall j \in \{0, k\} : w_j(x_i) \neq 0$, in quanto per ipotesi le ascisse di interpolazione sono tutte distinte. Per ipotesi di questo verso, la combinazione lineare degli elementi della base deve annullarsi, quindi affinche l'ipotesi venga rispettata deve valere $\alpha_0 = \ldots = \alpha_k = 0$
   - se $i < k$ allora tutti i termini della forma $w_j(x_i) = 0$ se $i < j$. Possono però rimanere dei termini per cui $w_r(x_i) \neq 0$ se $i \geqslant r$. Ma per questi termini posso ragionare come fatto nel caso di $i < k$, ottenendo che $\alpha_0 = \ldots = \alpha_r = 0$.

   Ripetendo questo ragionamento per ogni ascissa di interpolazione si ottiene $\alpha_0 = \ldots = \alpha_k = 0$ e il verso risulta dimostrato.

□

Exercise 4.1.5 (4.5). *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Dimostro i punti del teorema in ordine di come vengono richiesti dall'*esercizio 4.5*:

1. chiamo $w(x) = \alpha f(x) + \beta g(x)$, posso riscrivere:

$$\begin{aligned} w(x)[x_0, \dots, x_r] &= \sum_{j=0}^{r} \frac{w(x_j)}{\prod_{l=0;l\neq j}^{r} (x_j - x_l)} = \sum_{j=0}^{r} \frac{\alpha f(x_j) + \beta g(x_j)}{\prod_{l=0;l\neq j}^{r} (x_j - x_l)} = \\ &= \sum_{j=0}^{r} \frac{\alpha f(x_j)}{\prod_{l=0;l\neq j}^{r} (x_j - x_l)} + \sum_{j=0}^{r} \frac{\beta g(x_j)}{\prod_{l=0;l\neq j}^{r} (x_j - x_l)} = \\ &= \alpha \sum_{j=0}^{r} \frac{f(x_j)}{\prod_{l=0;l\neq j}^{r} (x_j - x_l)} + \beta \sum_{j=0}^{r} \frac{g(x_j)}{\prod_{l=0;l\neq j}^{r} (x_j - x_l)} = \\ &= \alpha f(x)[x_0, \dots, x_r] + \beta g(x)[x_0, \dots, x_r] \end{aligned}$$

2. esiste una matrice di permutazione $P \in \mathbb{R}^{(r+1)\times(r+1)}$ tale che permette di costruire un vettore $\mathbf{i}$:

$$\mathbf{i} = \begin{bmatrix} i_0 \\ \vdots \\ i_r \end{bmatrix} = P \begin{bmatrix} 0 \\ \vdots \\ r \end{bmatrix}$$

   in modo, usando la simmetria dell'operatore $\sum$ e dell'operatore $\prod$, da verificare $f(x)[x_0, \dots, x_r] = f(x)[x_{i_0}, \dots, x_{i_r}]$

3. sia $f \in \prod_k$ la funzione da approssimare e si assuma che questa sia definita con il seguente polinomio:

$$f(x) = \sum_{i=0}^{k} a_k x^k$$

   Analizzo $f[x_0, \dots, x_r]$ distinguendo due casi:

   - se $k = r$ allora

$$f[x_0, \dots, x_r] = f[x_0, \dots, x_k] = \sum_{j=0}^{k} \frac{f_j}{\prod_{l=0;l\neq j}^{k} (x_j - x_l)}$$

     Sia $p_r \in \prod_r$ il polinomio interpolante la funzione f. Per ipotesi del problema, f è un polinomio e stiamo analizzando per $k = r$, quindi $p_{k=r}$ e f hanno lo stesso grado. Inoltre, per l'unicità del polinomio interpolante, è vero che i coefficienti dei termini x^i devono coincidere, di conseguenza vale quanto detto anche per il termine principale:

$$f[x_0, \dots, x_k]x^k = a_k x^k \Rightarrow f[x_0, \dots, x_k] = a_k$$

   - se $k < r$: sia $p_r \in \prod_r$ il polinomio interpolante la funzione f. Per ipotesi del problema, f è un polinomio e stiamo analizzando per $k <$

r, quindi p_r ha un grado maggiore di f. Per l'unicità del polinomio interpolante, essendo f un polinomio (interpolante se stesso), allora p_r deve coincidere con il polinomio f. Per definizione, p_r ha questa forma:

$$p_r(x) = \sum_{j=0}^{r} f[x_0, \ldots, x_j]\omega_j(x)$$

ma tutti i termini $f[x_0, \ldots, x_i]\omega_i(x) = 0, k < i \leqslant r$ devono annullarsi in ordine di avere p_k coincidente con f. Affinchè si annullino è strettamente necessario che si annulli $f[x_0, \ldots, x_i] = 0$ in quanto $\omega_i \in \prod_r$ non può annullarsi perchè è un vettore della base di $\prod_r$. Se fosse vero che $\omega_i(x) = 0$ allora si produrrebbe un assurdo in quanto per ipotesi $p_k \in \prod_r$ ma questo spazio non sarebbe di grado r (ma di grado più basso).

4. devo dimostrare che se $f \in \mathcal{C}^{(r)}$ allora $\exists \xi \in [\min_i\{x_i\}, \ldots, \max_i\{x_i\}]$ tale che:

$$f[x_0, \ldots, x_r] = \frac{f^{(r)}(\xi)}{r!}$$

Sia $p_r \in \prod_r$ il polinomio interpolante, chiamo $e(x) = f(x) - p_r(x)$, con $e \in \mathcal{C}^{(r)}$. Si osserva che per costruzione del polinomio e, vale $e(x_i) = 0, \forall x_i \in \{x_0, \ldots, x_r\}$ insieme delle ascisse di interpolazione.

Per il teorema di Rolle $\exists \xi_0^{(1)} : e'(\xi_0^{(1)}) = 0$. Astraendo ottengo:

$$x_0 < \xi_0^{(1)} < x_1 < \xi_1^{(1)} < x_2 < \ldots < x_{r-1} < \xi_{r-1}^{(1)} < x_r \quad \wedge \quad \forall i \in \{0, \ldots, r-1\} : e'(\xi_i^{(1)}) = 0$$

Posso derivare r volte perchè $e \in \mathcal{C}^{(r)}$ ottenendo:

$$\begin{aligned}
&\forall i \in \{0, \ldots, r-1\} : e^{(1)}(\xi_i^{(1)}) = 0 \\
&\forall i \in \{0, \ldots, r-1\} : e^{(2)}(\xi_i^{(2)}) = 0 \\
&\qquad\qquad\vdots \\
&\forall i \in \{0, \ldots, r-1\} : e^{(r)}(\xi_i^{(r)}) = 0
\end{aligned}$$

Posso astrarre sui $\xi_i^{(k)}$ sia sull'indice i della posizione, sia sull'indice k dell'ordine della derivata (in quanto in tutti la derivata r-esima di e si annulla) e considerare un generico ξ.

Derivo r volte l'equazione $e(x) = f(x) - p_r(x)$ ottenendo $e^{(r)}(x) = f^{(r)}(x) - p_r^{(r)}(x)$ e valuto in ξ:

$$0 = e^{(r)}(\xi) = f^{(r)}(\xi) - p_r^{(r)}(\xi)$$

Per definizione p_r ha questa struttura:

$$p_r(x) = f[x_0]\omega_0(x) + f[x_0, x_1]\omega_1(x) + \ldots + f[x_0, \ldots, x_k]\omega_k(x) + \ldots + f[x_0, \ldots, x_r]\omega_r(x) \quad , k < r$$

Se derivo r volte, ottengo che i termini $f[x_0, \ldots, x_k]\omega_k^{(r)}(x)$ con $k < r$ si annullano per definizione dell'operatore derivata. Rimane solo il termine $f[x_0, \ldots, x_r]\omega_r(x)$. Ma anche per questo termine possiamo applicare lo stesso ragionamento, ovvero se si considera i termini generati da $\omega_r(x) = x^r + \alpha_{r-1}x^{r-1} + \ldots + \alpha_1 x + \alpha_0$, quando si deriva r volte si ottiene:

$$\omega_r^{(r)} = \frac{\vartheta^{(r)}(x^r)}{\vartheta x} + \frac{\vartheta^{(r)}(\alpha_{r-1}x^{r-1})}{\vartheta x} + \ldots + \frac{\vartheta^{(r)}(\alpha_i x^i)}{\vartheta x} + \ldots + \frac{\vartheta^{(r)}(\alpha_1 x)}{\vartheta x} + \frac{\vartheta^{(r)}(\alpha_0)}{\vartheta x} = \frac{\vartheta^{(r)}(x^r)}{\vartheta x} = r$$

$$\text{con } i < r \Rightarrow \frac{\vartheta^{(r)}(\alpha_i x^i)}{\vartheta x} = 0$$

Quindi rimane l'unico termine $\alpha_r x^r$, che derivato r volte produce $\frac{\vartheta^{(r)}(x^r)}{\vartheta x} = r!$. Osservazione: dato che ω_r produce un polinomio monico, e si considera solo il termine principale, allora $\alpha_r = 1$. Riassumendo:

$$0 = f^{(r)}(\xi) - p_r^{(r)}(\xi) = f^{(r)}(\xi) - f[x_0, \ldots, x_r]r!$$
$$f[x_0, \ldots, x_r] = \frac{f^{(r)}(\xi)}{r!}$$

5. Prova per induzione su r.
   - *Base*:
   $$f[x_0, x_1] = \frac{f_0}{x_0 - x_1} + \frac{f_1}{x_1 - x_0} = \frac{f_1 - f_0}{x_1 - x_0} = \frac{f[x_1] - f[x_0]}{x_1 - x_0}$$
   - *Induction hypothesis*: suppongo vero l'asserto per r e dimostro per $r + 1$
   - *Induction step*: devo far vedere che vale la seguente:
   $$f[x_0, \ldots, x_{r+1}] = \frac{f[x_1, \ldots, x_{r+1}] - f[x_0, \ldots, x_r]}{x_{r+1} - x_0} \tag{4.1}$$
   Studio i termini che compaiono al numeratore:
   $$\sum_{j=1}^{r+1} \frac{f_j}{\prod_{l=1; l\neq j}^{r+1}(x_j - x_l)} - \sum_{j=0}^{r} \frac{f_j}{\prod_{l=0; l\neq j}^{r}(x_j - x_l)} =$$
   Metto in evidenza la sommatoria, con indice $j \in \{1, \ldots, r\} = J$; l'insieme J è l'intersezione degli insiemi sui cui "corrono" entrambi gli indici delle due sommatorie che sto fattorizzando:

$$= \sum_{j=1}^{r} f_j \left(\frac{1}{\prod_{l=1; l\neq j}^{r+1}(x_j - x_l)} - \frac{1}{\prod_{l=0; l\neq j}^{r}(x_j - x_l)} \right) + \frac{f_{r+1}}{\prod_{l=1}^{r}(x_{r+1} - x_l)} - \frac{f_0}{\prod_{l=1}^{r}(x_0 - x_l)}$$

(4.2)

Studio il primo addendo della 4.2:

$$\sum_{j=1}^{r} f_j \left(\frac{1}{\prod_{l=1;l\neq j}^{r+1} (x_j - x_l)} - \frac{1}{\prod_{l=0;l\neq j}^{r} (x_j - x_l)} \right) = \sum_{j=1}^{r} \frac{f_j}{\prod_{l=1;l\neq j}^{r} (x_j - x_l)} \left(\frac{1}{x_j - x_{r+1}} - \frac{1}{x_j - x_0} \right)$$
$$= \sum_{j=1}^{r} \frac{f_j}{\prod_{l=1;l\neq j}^{r} (x_j - x_l)} \left(\frac{(x_j - x_0) - (x_j - x_{r+1})}{(x_j - x_{r+1})(x_j - x_0)} \right) = \sum_{j=1}^{r} \frac{f_j (x_{r+1} - x_0)}{\prod_{l=0;l\neq j}^{r+1} (x_j - x_l)}$$

Come si nota nella 4.1, questa quantità deve essere raggruppata per $x_{r+1} - x_0$, quindi:

$$\sum_{j=1}^{r} \frac{f_j (x_{r+1} - x_0)}{\prod_{l=0;l\neq j}^{r+1} (x_j - x_l)} \left(\frac{1}{x_{r+1} - x_0} \right) = \sum_{j=1}^{r} \frac{f_j}{\prod_{l=0;l\neq j}^{r+1} (x_j - x_l)}$$

Componendo questi passi si ottiene:

$$\sum_{j=1}^{r} f_j \left(\frac{1}{\prod_{l=1;l\neq j}^{r+1} (x_j - x_l)} - \frac{1}{\prod_{l=0;l\neq j}^{r} (x_j - x_l)} \right) \left(\frac{1}{x_{r+1} - x_0} \right) = \sum_{j=1}^{r} \frac{f_j}{\prod_{l=0;l\neq j}^{r+1} (x_j - x_l)} \tag{4.3}$$

Studio il secondo addendo della 4.2 applicando il raggruppamento $x_{r+1} - x_0$ come primo passo; come secondo passo "sincronizzo" l'indice della produttoria per essere "compatibile" con l'indice della produttoria che appare nel denomitore della 4.3:

$$\frac{f_{r+1}}{\prod_{l=1}^{r} (x_{r+1} - x_l)} \left(\frac{1}{x_{r+1} - x_0} \right) = \frac{f_{r+1}}{\prod_{l=0}^{r} (x_{r+1} - x_l)} = \frac{f_{r+1}}{\prod_{l=0;l\neq r+1}^{r+1} (x_{r+1} - x_l)}$$

Ho ottenuto il $(r+1)$-esimo termine della sommatoria principale, quindi sommo questo risultato alla 4.3:

$$\sum_{j=1}^{r} \frac{f_j}{\prod_{l=0;l\neq j}^{r+1} (x_j - x_l)} + \frac{f_{r+1}}{\prod_{l=0;l\neq r+1}^{r+1} (x_{r+1} - x_l)} = \sum_{j=1}^{r+1} \frac{f_j}{\prod_{l=0;l\neq j}^{r+1} (x_j - x_l)} \tag{4.4}$$

Studio il terzo addendo della 4.2 applicando il raggruppamento $x_{r+1} - x_0$:

$$\frac{f_0}{\prod_{l=1}^{r} (x_0 - x_l)} \left(\frac{1}{x_{r+1} - x_0} \right)$$

e sommo algebricamente alla 4.4:

$$\sum_{j=1}^{r+1} \frac{f_j}{\prod_{l=0;l\neq j}^{r+1}(x_j - x_l)} - \frac{f_0}{\prod_{l=1}^{r}(x_0 - x_l)}\left(\frac{1}{x_{r+1} - x_0}\right) =$$
$$= \sum_{j=1}^{r+1} \frac{f_j}{\prod_{l=0;l\neq j}^{r+1}(x_j - x_l)} + \frac{f_0}{\prod_{l=1}^{r}(x_0 - x_l)}\left(\frac{1}{x_0 - x_{r+1}}\right) =$$

"sincronizzo" l'indice della produttoria per essere "compatibile" con l'indice della produttoria che appare nel denomitore del primo termine:

$$= \sum_{j=1}^{r+1} \frac{f_j}{\prod_{l=0;l\neq j}^{r+1}(x_j - x_l)} + \frac{f_0}{\prod_{l=0;l\neq 0}^{r+1}(x_0 - x_l)} =$$

Con il secondo termine ho ottenuto lo (0)-esimo termine della sommatoria principale, quindi posso includerlo nella sommatoria principale modificando il range su cui "corre" l'indice j:

$$= \sum_{j=0}^{r+1} \frac{f_j}{\prod_{l=0;l\neq j}^{r+1}(x_j - x_l)} = f[x_0, \ldots, x_{r+1}] \tag{4.5}$$

questo completa il passo induttivo.

□

Exercise 4.1.6 (4.6)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Vedi il codice Exercise 4.6 on textbook - Differenze divise.

Exercise 4.1.7 (4.7)**.** *Per il testo dell'esercizio consultare il libro di testo.*

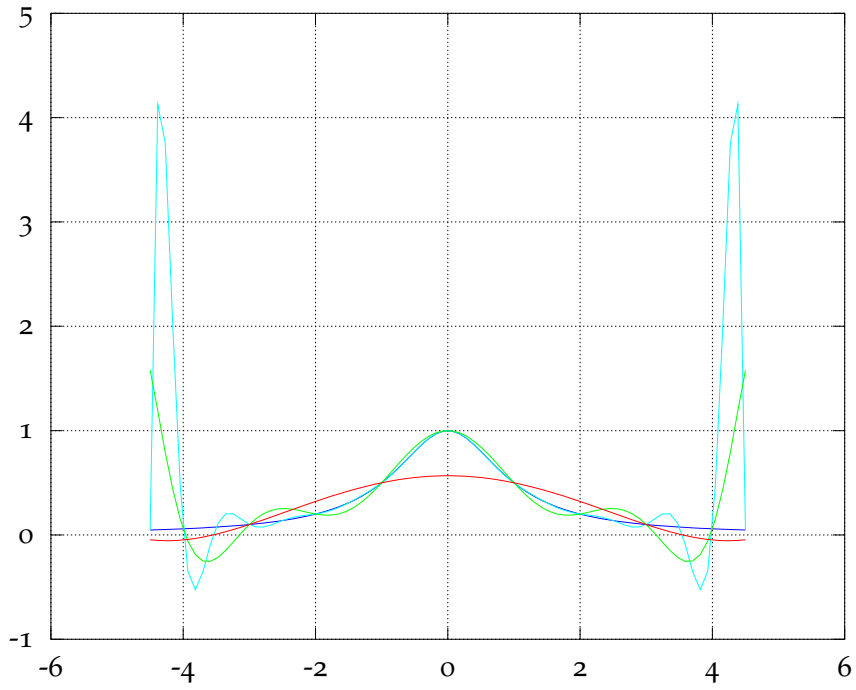
Vedi il codice Exercise 4.7 on textbook - Horner generalizzato. Per generare i risultati che sto per descrivere, lanciare la funzione definita in Exercise 4.7 on textbook - Function

Ho considerato la funzione:

$$f(x) = \frac{1}{1 + x^2}$$

fissando come ascisse di intepolazione $x \in linspace(-3, 3, 30) = \S$, con $linspace$ intendo la funzione di *Octave*.

Il seguente grafico mostra l'interpolazione della funzione f utilizzando l'algoritmo di *Horner generalizzato* (implementato in Exercise 4.7 on textbook - Horner generalizzato) in ascisse $\hat{x} \in [-5 : 0.5 : 5], [-5 : 1 : 5], [-5 : 2 : 5]$, corrispondenti rispettivamente alle curve cyan, verde, rosso. In blu la funzione reale f.



Dimostro che l'algoritmo Exercise 4.7 on textbook - Horner generalizzato valuta il polinomio interpolante nella forma di Newton $p_r(x) \in \prod_r$ in un punto di ascissa x:

Dimostrazione. L'algoritmo riportato implementa questo schema (utilizzo gli indici nella notazione usata nella formulazione matematica, quindi sono zero-based):

$$\begin{aligned} p^{(0)}(x) &= f[x_0, \ldots, x_r] \\ p^{(i+1)}(x) &= p^{(i)}(x - x_{r-i}) + f[x_0, \ldots, x_{r-i}] \end{aligned}$$

con $p^{(i)}$ indico il valore di p all'i-esimo passo dei esecuzione. Se considero il valore di p ad un generico passo i di esecuzione si osserva che ha questa struttura:

$$\begin{aligned} p^{(i)}(x) = \big(&f[x_0, \ldots, x_r](x - x_{r-1}) \cdots (x - x_{r-i+1}) + f[x_0, \ldots, x_{r-1}](x - x_{r-2}) \cdots (x - x_{r-i+1}) + \ldots + \\ &+ f[x_0, \ldots, x_{r-i+1}](x - x_{r-i+1})\big)(x - x_{r-i}) + f[x_0, \ldots, x_{r-i}] \end{aligned}$$

Il polinomio $p^{(i)}$ è di grado i, quindi saturando l'indice i arrivando a calcolare $p^{(r)}$, si ottiene il polinomio:

$$p_r(x) = p^{(r)}(x) = f[x_0, \ldots, x_r]\omega_r(x) + f[x_0, \ldots, x_{r-1}]\omega_{r-1}(x) + \ldots + f[x_0]$$

Ovvero quello che si chiede nel problema. □

Exercise 4.1.8 (4.8). *Per il testo dell'esercizio consultare il libro di testo.*

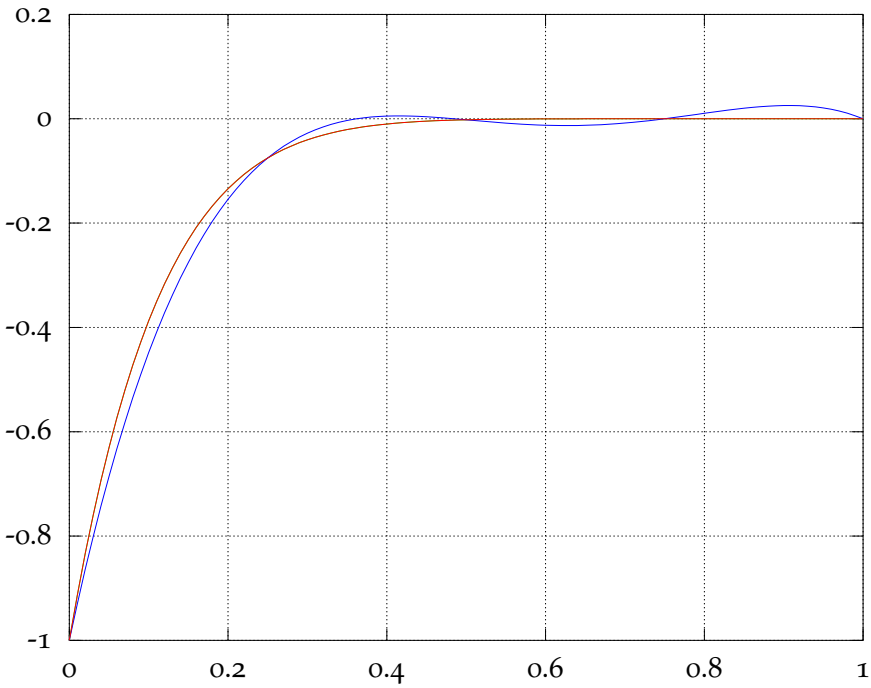
Vedi il codice Exercise 4.8 on textbook - Differenze divise di Hermite.

Exercise 4.1.9 (4.9). *Per il testo dell'esercizio consultare il libro di testo.*

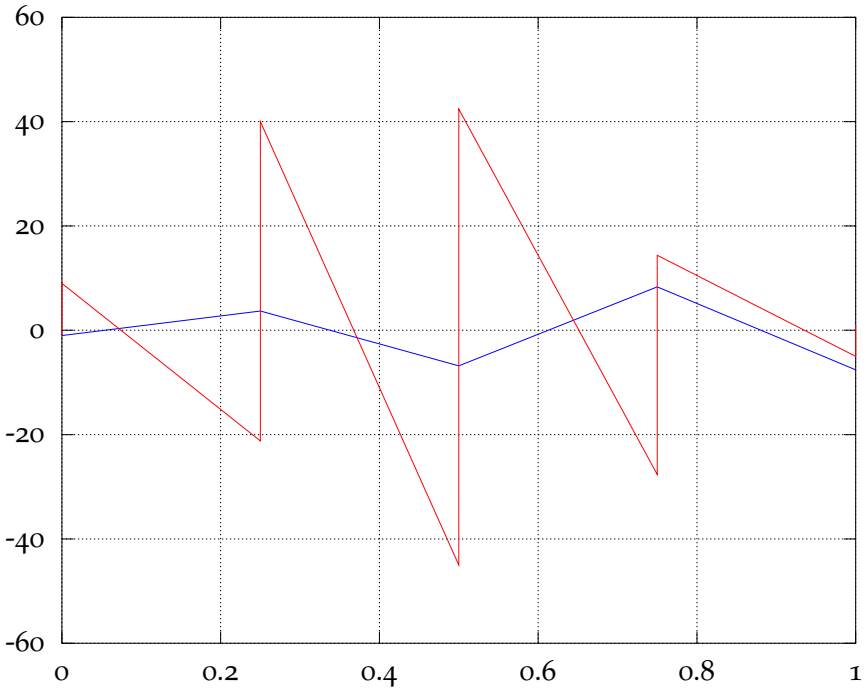
Per l'implementazione del metodo di *Hermite* vedere il codice Hermite engine.

Per generare i risultati che sto per descrivere, lanciare la funzione definita in Exercise 4.9 on textbook. Nei prossimi grafici le curve in colore *rosso* sono custruite usando un polinomio interpolante nella forma di *Hermite*, mentre quelle in *blu* sono custruite usando un polinomio interpolante nella forma di *Newton*.

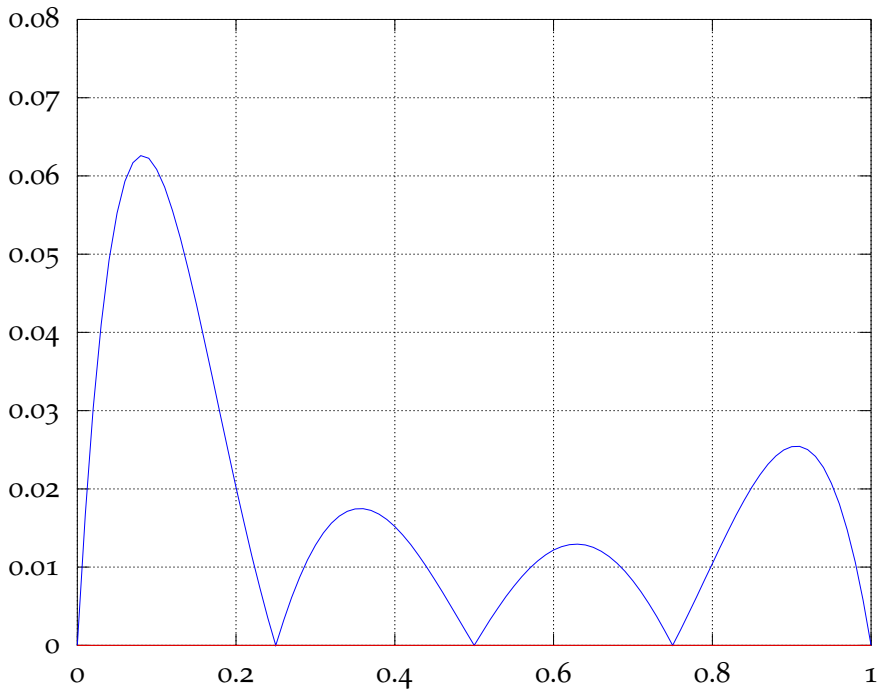
In questo primo plot mostro l'interpolazione della curva reale (in *verde*), la quale è totalmente nascosta dalla curva ottenuta con polinomio interpolante nella forma di *Hermite*:



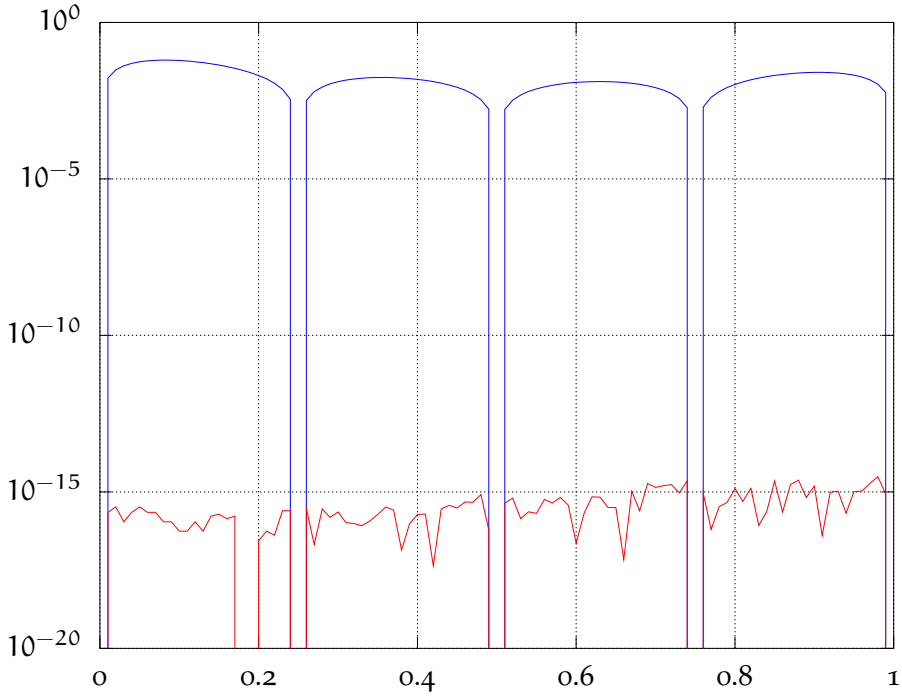
Il prossimo plot rappresenta le differenze divise calcolate per costruire i polinomi interpolanti:



Il prossimo plot mostra gli errori che si commette nelle due approssimazioni, usando entrambi gli assi con scala lineare:



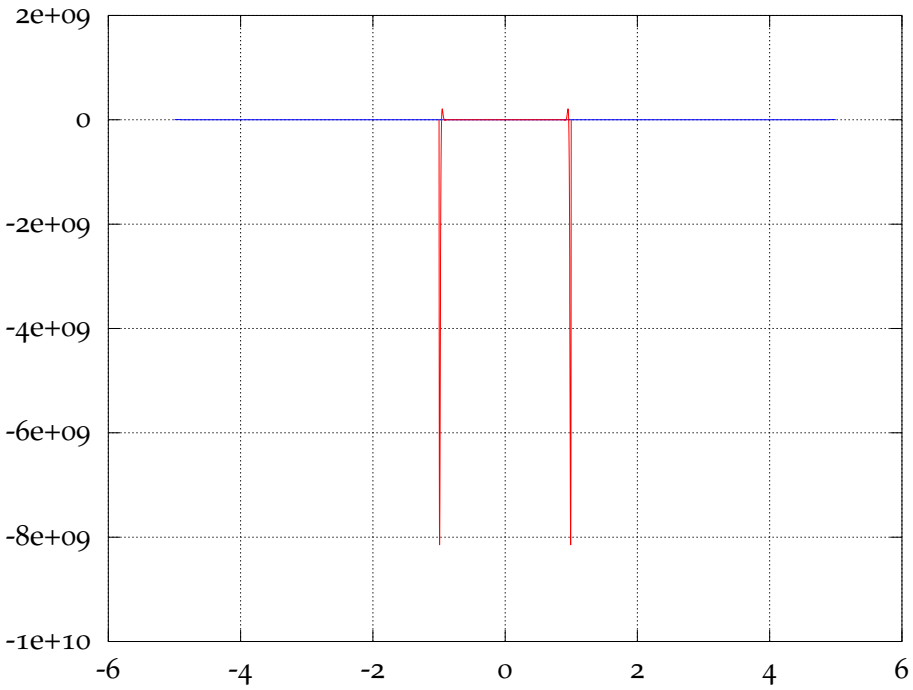
Il prossimo plot mostra gli errori che si commette nelle due approssimazioni, usando l'asse delle ordinate con scala logaritmica, in modo da evidenziare che anche usando lo schema di *Hermite* si commettono errori, molto piccoli, ma che non era possibile apprezzarli nel plot precedente:



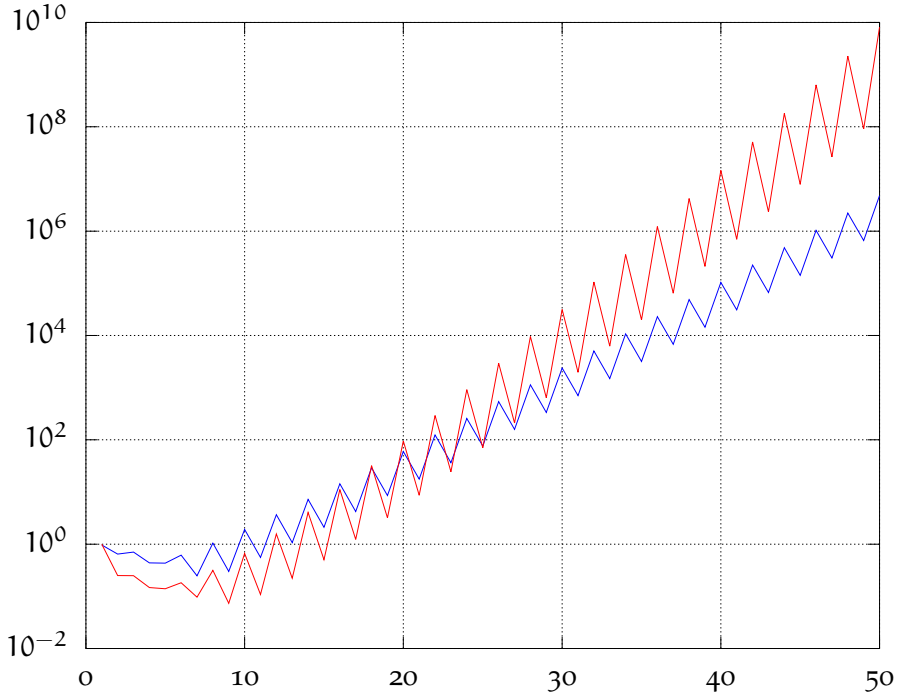
Exercise 4.1.10 (4.11). *Per il testo dell'esercizio consultare il libro di testo.*

Per il codice che implementa le richieste dell'esercizio e produce i seguenti risultati vedere Exercise 4.11 on textbook.

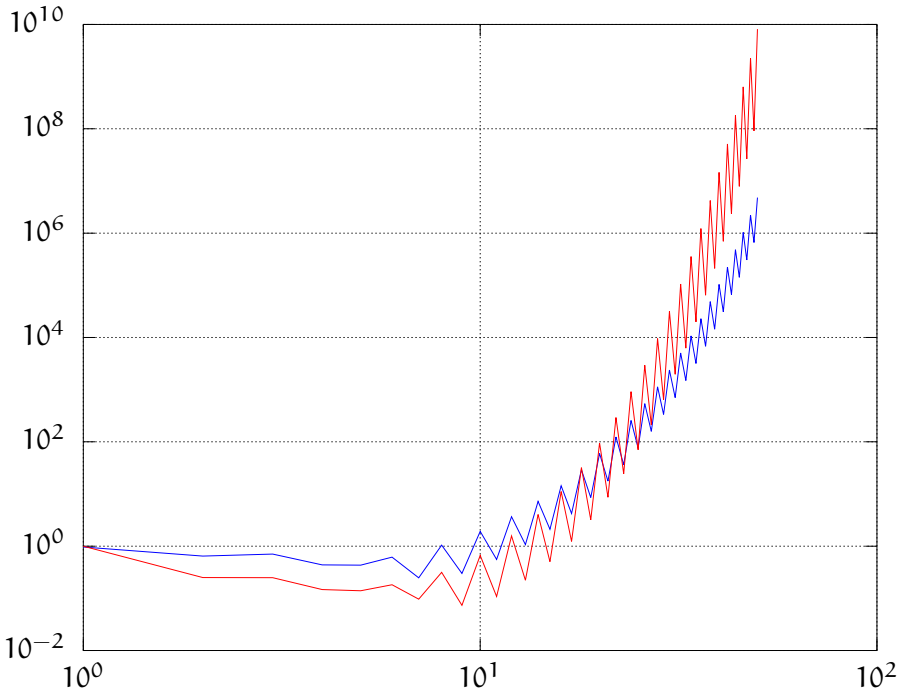
Riporto il plot delle approssimazioni ottenute considerando il massimo numero di ascisse a disposizione:



Nei seguenti grafici in *rosso* è rappresentata la funzione di *Bernstein*, mentre in *blu* è rappresentata la funzione di *Runge*. Questo primo grafico ha l'asse delle ordinate costruito in modo logaritmico:



Riporto lo stesso dataset di errori per le due curve, costruendo entrambi gli assi in modo logaritmico:



Per entrambi i grafici, le valutazioni della rispettiva funzione, del polinomio interpolante e la costruzione del polinomio interpolante sono riferiti ai rispettivi dominii $[a, b]$ definiti nel testo dell'esercizio.

Possiamo notare che l'errore diverge all'aumentare del numero di ascisse di interpolazione e, confrontando gli errori ottenuti per le due funzioni, possiamo dire il problema di interpolare la funzione di *Bernstein* è maggiormente malcondizionato del problema di interpolare la funzione di *Runge*.

Exercise 4.1.11 (4.12)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Dimostro i due versi distinti di:

$$x \in [-1, 1] \Leftrightarrow \tilde{x} = \tilde{x}(x) \in [a, b] \quad \text{con } \tilde{x} = \tilde{x}(x) = \frac{a+b}{2} + \frac{b-a}{2}x$$

supponendo senza perdere di generalità $a \leqslant b$:

- ($\Rightarrow$) Quello che voglio utilizzare è $\tilde{x} = \tilde{x}(x)$
  - se $x = -1$:

    $$\tilde{x}(x) = \frac{a+b}{2} - \frac{b-a}{2} = \frac{a+b-b+a}{2} = a \in [a, b]$$

    vero perchè $a \leqslant b$ per ipotesi.

– se $x = 1$:

$$\tilde{x}(x) = \frac{a+b}{2} + \frac{b-a}{2} = \frac{a+b+b-a}{2} = b \in [a,b]$$

vero perchè $a \leqslant b$ per ipotesi.

– se $x = 0$:

$$\tilde{x}(x) = \frac{a+b}{2}$$

per ipotesi $a \leqslant b$ allora $\exists \epsilon \geqslant 0 : b = a + \epsilon$, quindi ottengo:

$$\tilde{x}(x) = \frac{a+b}{2} = \frac{2a+\epsilon}{2} = a + \frac{\epsilon}{2}$$

Ma $a + \frac{\epsilon}{2} \geqslant a$ perchè $\frac{\epsilon}{2} \geqslant 0$ in quanto $\epsilon \geqslant 0$ per ipotesi.

Ma $a + \frac{\epsilon}{2} \leqslant b$ perchè $b = a + \epsilon$ implica $a + \frac{\epsilon}{2} \leqslant a + \epsilon$, vero perchè $\frac{\epsilon}{2} \leqslant \epsilon$ in quanto $\epsilon \geqslant 0$ per ipotesi. Quindi $\tilde{x}(x) \in [a,b]$.

- ($\Leftarrow$) Quello che voglio utilizzare è $x = x(\tilde{x})$. Dalle ipotesi segue che:

$$\begin{aligned}
\tilde{x} &= \frac{a+b}{2} + \frac{b-a}{2}x \\
2\tilde{x} &= a + b + (b-a)x \\
2\tilde{x} &= a + (b-a) + a + (b-a)x \\
2(\tilde{x}-a) &= (b-a)(x+1) \\
\frac{2(\tilde{x}-a)}{b-a} &= x+1 \\
\frac{2(\tilde{x}-a)}{b-a} - 1 &= x
\end{aligned}$$

Se $\tilde{x} = \min\{[a,b]\} = a \Rightarrow x = -1$.

Se $\tilde{x} = \max\{[a,b]\} = a \Rightarrow x = \frac{2(b-a)}{(b-a)} - 1 = 1$

$x(\tilde{x}) = 0$ per $2\tilde{x} - 2a + a - b = 0$ quindi per $\tilde{x} = \frac{b+a}{2}$. Inoltre $x'(\tilde{x}) = \frac{2}{b-a}$, la derivata prima è costante, non ci sono punti di min o max locali. Quindi $x \in [-1,1]$.

□

Exercise 4.1.12. *Se* $f \in \mathcal{C}^{(1)}([a,b])$ *allora* f *è* Lipschtziana *con* $L = \|f^{(1)}\|$.

Dimostrazione. Per ipotesi f è continua e derivabile in $[a, b]$, quindi posso applicare il teorema di Lagrange ottenendo

$$\begin{aligned}
\frac{f(b)-f(a)}{b-a} &= f'(\xi) \quad \xi \in [a,b] \\
\left|\frac{f(b)-f(a)}{b-a}\right| &= |f'(\xi)| \\
\frac{|f(b)-f(a)|}{|b-a|} &= |f'(\xi)| \\
|f(b)-f(a)| &= |f'(\xi)| \cdot |b-a| \\
\max_{x\in[a,b]}\{|f(b)-f(a)|\} &= \max_{x\in[a,b]}\{|f'(\xi)| \cdot |b-a|\} \\
\max_{x\in[a,b]}\{|f(b)-f(a)|\} &\leqslant \max_{x\in[a,b]}\{|f'(\xi)|\} \max_{x\in[a,b]}\{|b-a|\} \\
|f(b)-f(a)| &\leqslant \max_{x\in[a,b]}\{|f'(\xi)|\}|b-a| \\
|f(b)-f(a)| &\leqslant \|f'\| \cdot |b-a|
\end{aligned}$$

ovvero la definizione di funzione Lipstichziana con $L = \|f'\|$. □

Exercise 4.1.13. *Sia* $f \in \mathcal{C}^{(1)}([a,b])$ *una funzione* Lipschtziana. *Allora* $w(f;h) \leqslant Lh$, *con* $h > 0$.

Dimostrazione. Dalla definizione di f *Lipschtziana* segue che

$$\begin{aligned}
|f(x)-f(y)| &\leqslant L|x-y| \\
\max_{x,y\in[a,b]}\{|f(x)-f(y)|\} &\leqslant \max_{x,y\in[a,b]}\{L|x-y|\} \quad \text{ho applicato la funzione max ad entramb} \\
\max_{x,y\in[a,b]}\{|f(x)-f(y)|\} &\leqslant L \max_{x,y\in[a,b]}\{|x-y|\} \\
\max_{x,y\in[a,b]}\{|f(x)-f(y)| : |x-y| < h\} &\leqslant L \max_{x,y\in[a,b]}\{|x-y| : |x-y| < h\} \quad \text{impongo vincolo sulla distar} \\
w(f;h) &\leqslant Lh
\end{aligned}$$

□

Exercise 4.1.14. *Dimostrare che al crescere di* n *il termine* $w(f; \frac{b-a}{n}) \to 0$.

Dimostrazione. Per ipotesi segue che $h = \frac{b-a}{n} \to 0$.
Posso usare le definizione di $w(f;h) = \sup\{|f(x)-f(y)| : \exists h > 0 : |x-y| < h\}$.
Istanzio per il nostro caso di $h \to 0$ ottenendo $w(f;h) = \sup\{|f(x)-f(y)| : h \to 0\}$.
Supponendo f continua allora vale $\lim_{x\to y} f(x) = f(y)$.
Compongo questi risultati:

$$\lim_{h\to 0} w(f;h) = \lim_{h\to 0} \sup\{|f(x)-f(y)| : |x-y| = h\} = \lim_{x\to y} \sup\{|f(x)-f(y)|\} = \lim_{x\to y} \sup\{|f(y)-f(y)|\}$$

□

Exercise 4.1.15. *Dimostrare le proprietà dei polinomi di* Chebyshev *di pagina 92.*

Dimostrazione. Dimostro per punti:

1. $T_k(x)$ ha grado esatto k

   PROOF Prova per induzione su k.

   BASE per $k = 0$ si ha per la definizione dei polinomi di *Chebyshev*, $T_0(x) = 1$, essendo una costante implica $\deg(T_0(x)) = 0$, la base è vera.

   INDUCTION HP suppongo vero che $\deg(T_{k-1}(x)) = k-1$ e $\deg(T_{k-2}(x)) = k-2$

   INDUCTION STEP dimostro per k:

   dalla definizione del k-esimo polinomio: $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$
   Applico la funzione deg ad entrambi i membri: $\deg(T_k(x)) = \deg(2xT_{k-1}(x) - T_{k-2}(x))$
   La funzione deg applicata ad una somma di polinomi produce:

$$\begin{aligned}\deg(2xT_{k-1}(x) - T_{k-2}(x)) &= \max\{\deg(2xT_{k-1}(x)), \deg(T_{k-2}(x))\} = \\ &= \max\{\deg(2x) + \deg(T_{k-1}(x)), \deg(T_{k-2}(x))\}\end{aligned}$$

   Ma per ipotesi induttiva: $\deg(T_{k-1}(x)) = k-1 > k-2 = \deg(T_{k-2}(x))$, quindi:

$$\max\{\deg(2x) + \deg(T_{k-1}(x)), \deg(T_{k-2}(x))\} = \deg(2x) + \deg(T_{k-1}(x)) = 1 + (k-1) = k$$

2. Il coefficiente principale di $T_k(x)$, in simboli $cp(T_k(x))$, soddisfa $cp(T_k(x)) = 2^{k-1}, k \geqslant 1$

   PROOF Prova per induzione su k.

   BASE per $k = 1$ si ha per la definizione dei polinomi di *Chebyshev*, $T_1(x) = x$, quindi $cp(x) = 1 = 2^{1-1} = 1$, la base è vera.

   INDUCTION HP suppongo vero che $cp(T_{k-1}(x)) = 2^{k-2}$, $cp(T_{k-2}(x)) = 2^{k-3}$

   INDUCTION STEP dimostro per k:

   dalla definizione del k-esimo polinomio: $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$
   Applico la funzione cp ad entrambi i membri: $cp(T_k(x)) = cp(2xT_{k-1}(x) - T_{k-2}(x))$
   La funzione cp applicata ad una somma di polinomi produce:

$$cp(2xT_{k-1}(x) - T_{k-2}(x)) = \max\{cp(2xT_{k-1}(x)), cp(T_{k-2}(x))\}$$

   Ma per ipotesi induttiva: $\deg(T_{k-1}(x)) = k-1 > k-2 = \deg(T_{k-2}(x))$, quindi:

$$\max\{cp(2xT_{k-1}(x)), cp(T_{k-2}(x))\} = cp(2xT_{k-1}(x)) = cp(2x)cp(T_{k-1}(x)) = 2 \cdot 2^{k-2} = 2^{k-1}$$

3. la famiglia di polinomi costruita con questo schema ricorsivo

$$\hat{T}_0(x) = T_0(x) = 1 \quad \hat{T}_1(x) = T_1(x) = x \quad \hat{T}_k(x) = 2^{1-k}T_k(x)$$

è una famiglia di polinomi monici.

PROOF Usando il precedente punto di questo esercizio vale che per il k-esimo polinomio $T_k(x)$ si ha $cp(T_k(x)) = 2^{k-1}$.
Dalla definizione dei polinomi di questa nuova famiglia

$$cp(\hat{T}_k(x)) = 2^{1-k}cp(T_k(x)) = 2^{1-k} \cdot 2^{k-1} = 1$$

e quindi i $\hat{T}_k(x)$ sono polinomi monici.

4. ponendo $x = \cos\theta$ con $\theta \in [0, \pi]$, si ottiene $T_k(x) = T_k(\cos\theta) = \cos k\theta$ con $k = 0, 1, \ldots$

PROOF Prova per induzione su k.

BASE per $k = 0$ si ha per la definizione dei polinomi $T_0(x) = 1$, mentre usando la sostituzione $x = \cos\theta$, si ha $T_0(x) = T_0(\cos\theta) = \cos 0 \cdot \theta = 1$, la base è vera.

INDUCTION HP suppongo vero che $T_k(x) = T_k(\cos\theta) = \cos k\theta$ e $T_{k-1}(x) = T_{k-1}(\cos\theta) = \cos(k-1)\theta = \cos(k\theta - \theta)$

INDUCTION STEP dimostro per $k+1$:
dalla definizione del $k+1$-esimo polinomio: $T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x)$
utilizzo l'ipotesi induttiva per il secondo membro e ottengo:

$$\begin{aligned} T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x) &= 2\cos\theta\cos k\theta - \cos(k\theta - \theta) = \\ &= 2\cos\theta\cos k\theta - (\cos k\theta\cos\theta + \sin k\theta\sin\theta) \\ &= \cos\theta\cos k\theta - \sin\theta\sin k\theta = \cos(k+1)\theta \end{aligned}$$

□

Exercise 4.1.16 (4.13). *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Dimostro per punti:

- gli zeri di $T_k(x)$ sono tra loro tutti distinti e sono dati da:

$$x_i^{(k)} = \cos\left(\frac{(2i+1)\pi}{2k}\right)$$

PROOF Sto cercando gli zeri in $[-1,1]$ di conseguenza $\theta \in [0,\pi]$.
Usando le quarta proprietà dell'esercizio 4.1.15 posso studiare $T_k(\cos\theta) = 0$.
Questo implica studiare $\cos k\theta = 0$, vero quando $k\theta = \frac{\pi}{2} + j\pi$ con $j \geqslant 0$
Devo porre vincoli sul precedente indice j, in quanto sto supponendo che $\theta \in [0,\pi]$. Quindi $j = 0,\dots,k-1$ (se si includesse anche k allora, per $j = k$, si avrebbe $\theta = \frac{\pi}{2k} + \frac{k}{k}\pi = \frac{\pi}{2k} + \pi$ e si avrebbe $\theta \in [0, \frac{\pi}{2k} + \pi] \supset [0,\pi]$ contro le nostre richieste).

Questa scelta di valori implica che si avranno k zeri ed inoltre tutti distinti tra loro in quanto non si effettua più di un "giro" sulla circonferenza unitaria (in quanto il termine $\frac{j}{k}\pi < \pi$ per ogni j scelto precedentemente).

- i valori estremi di $T_k(x)$ sono assunti nei punti

$$\xi_i^{(k)} = \cos\left(\frac{i}{k}\pi\right) \quad i = 0,\dots,k$$

PROOF Sto cercando i punti estremi in $[-1,1]$ di conseguenza $\theta \in [0,\pi]$.
Usando le quarta proprietà dell'esercizio 4.1.15 posso studiare $T_k(\cos\theta) = \cos k\theta$.
Questo implica studiare $\cos k\theta = \pm 1$, vero quando $k\theta = j\pi$ con $j = 0,\dots,k$, manipolando $\theta = \frac{j}{k}\pi$.
Dato che abbiamo usato la sostituzione $x = \cos\theta$ si ottiene, cambiando variabile solo per chiarezza e non usare x sia per gli zeri che per gli estremi, $\xi_j^{(k)} = \cos\left(\frac{j}{k}\pi\right)$.
Valutando il polinomio $T_k(x)$ nei punti $\xi_i^{(k)}$, si ottiene:

$$T_k(\xi_i^{(k)}) = T_k\left(\cos\left(\frac{i}{k}\pi\right)\right) = \cos\left(k\frac{i}{k}\pi\right) = \cos(i\pi) = (-1)^i \quad i = 0,\dots,k$$

Inoltre, dalla precedente equazione si osserva che $\|T_k\| = 1$

- Per $k = 0, 1, \dots$ vale

$$\|\hat{T}_k\| = \min_{p \in \prod_n} \|p\|$$

PROOF Supponiamo per assurdo che esiste un polinomio monico $p \in \prod_k$ di grado k tale che $\|p\| = \delta < 2^{1-k} = \|\hat{T}_k\|$ (per il punto precedente vale $\|T_k\| = 1$), in parole: abbiamo supposto (per assurdo) che $\hat{T}_k(x)$ non sia di minima norma.
Considero un nuovo polinomio $(\hat{T}_k - p)(x)$. Osserviamo che $\deg((\hat{T}_k - p)(x)) = k-1$, in quanto sia $\hat{T}_k(x)$ che p sono monici e dello stesso grado k, annullando così il coefficiente del termine principale.

Considero i punti in cui $T_k(x)$ assume valori massimo, ovvero per $\xi_i^{(k)} = \cos\left(\frac{i}{k}\pi\right)$, con $i = 0, \dots, k$.
Studiando il segno dei polinomi T_k e $\hat{T}_k$ nei punti $\xi_i^{(k)}$ si ottiene:

$$\text{sign}(\hat{T}_k(\xi_i^{(k)})) = \text{sign}(T_k(\xi_i^{(k)})) = (-1)^i \quad i = 0, \dots, k$$

Ma anche $(\hat{T}_k - p)(\xi_i^{(k)}) = (-1)^i$, ovvero $(\hat{T}_k - p)$ cambia di segno $k+1$ volte. Dato che $(\hat{T}_k - p)$ è una funzione continua, allora considerando due ascisse di massimo consecutive $\xi_i^{(k)} = (-1)^i \neq (-1)^{i+1} = \xi_{i+1}^{(k)}$, ovvero il polinomio assume segno discorde tra due ascisse consecutive di massimo. Per questo motivo per ogni coppia di ascisse di massimo distinte esisterà una radice (quindi si avranno k radici).
Dato che abbiamo $k+1$ ascisse $\xi_i^{(k)}$, allora la funzione $(\hat{T}_k - p)$ si annullerà k volte.
Ma affinchè questo sia vero, e da come abbiamo trovato, $(\hat{T}_k - p) \in \prod_{k-1}$, deve valere che $(\hat{T}_k - p) = 0$, ovvero sia il vettore identicamente nullo. Ma per costruzione abbiamo che $\hat{T}_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$, considerando il caso base $k = 0$ implica $\hat{T}_0(x) = 1$. Ma per ipotesi di assurdo deve valere $\|p\| < \|\hat{T}_0\| = 1$, quindi il polinomio $(\hat{T}_k - p)$ non può essere il polinomio nullo e questo produce una contraddizione. Abbiamo quindi dimostrato che $\hat{T}_k$ è di minima norma.

Se $(\hat{T}_k - p)$ non fosse un polinomio identicamente nullo allora avrebbe almeno *un* e al massimo $k-2$ termini della forma $a_u x^u$: $u < k$ perchè il grado massimo è al massimo $k-1$, e avrebbe al massimo $k-2$ termini contenenti x_u, escludendo il termine $a_y x^0 = a_y$. Ma dato che abbiamo al massimo $k-2$ termini non è possibile annullare il polinomio in k radici.

□

Observation 4.1.1 (Obiettivo delle ascisse di Chebyshev)**.** *Posso riassumere in poche righe l'obiettivo delle ascisse di* Chebyshev*. Sappiamo che l'errore di approssimazione viene maggiorato seguento questa disuguaglianza:*

$$\|e\| \leqslant \alpha(1 + \Lambda_n)w(f : h)$$

in cui dobbiamo controllare l'andamento dei due termini $\Lambda_n, w(f : h)$:

- *per quanto riguarda Λ_n sappiamo che ha due comportamenti:*

$$\Lambda_n \sim \frac{2^{n+1}}{\log n}$$

dove il numeratore diverge più velocemente del denominatore, facendo divergere Λ_n.

È anche vero che

$$\Lambda_n \sim \log n \Rightarrow \lim_{n\to\infty} \frac{\log n}{n} = 0$$

e questo è il caso a cui vogliamo ricondurci, scegliendo in modo opportuno le ascisse di interpolazione (escludendo il modo equidistante, in quanto abbiamo già visto che implica la divergenza).

Usando le ascisse di Chebyshev *si ottiene* $\Lambda_n \sim \frac{2}{\pi} \log n$*, dello stesso ordine di quello che vorremmo avere.*

- *se si assume la* $f \in \mathcal{C}^{(n+1)}$ *sufficientemente regolare allora*

$$\|e\| \leqslant \frac{\|f^{(n+1)}\|}{(n+1)!}\|w_{n+1}\|$$

L'unico termine che può dare problemi è $\|w_{n+1}\|$*, in quanto dipendente dalle ascisse di interpolazione (con ascisse equidistanti produce una successione* $\{\Lambda_n\}$ *che diverge in modo esponenziale). Per questo motivo si vogliono scegliere delle ascisse che minimizzano* $\|w_{n+1}\|$*.*

Usando le ascisse di Chebyshev *si ottiene* $w_{n+1}(x) = \hat{T}_{n+1}(x)$ *e* $\|w_{n+1}\| = 2^{-n}$*, minimizzata in quanto*

$$\|w_{n+1}\| = \|\hat{T}_k\| = \min_{\gamma \in \prod_k'} \|\gamma\| = 2^{1-(n+1)} = 2^{-n}$$

con $\hat{T}_k$ *il polinomio con minima norma come enunciato nel* teorema 4.9 *del libro di testo.*

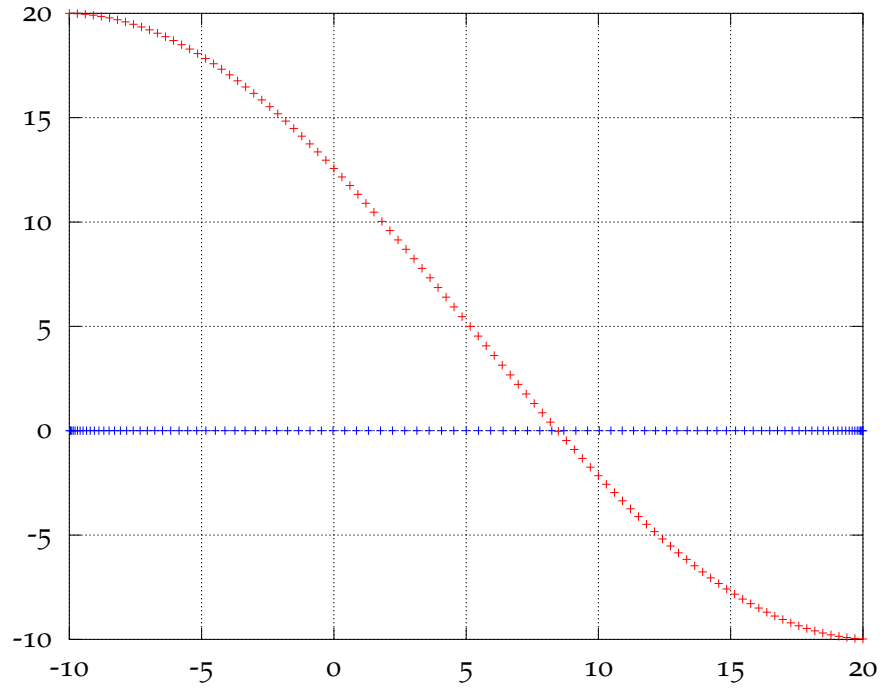
Exercise 4.1.17 (4.14)**.** *Per il testo dell'esercizio consultare il libro di testo.*

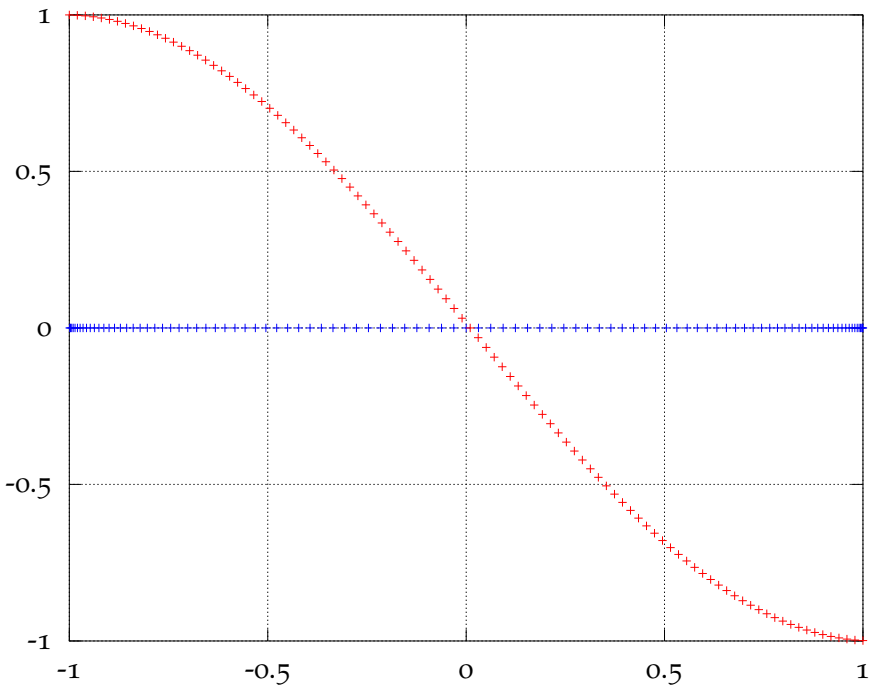
Per utilizzare lo schema di *Chebyshev* in un intervallo generico $[a, b]$ si utilizzano le riscritture date nell'esercizio 4.1.11.

Exercise 4.1.18. *Rappresentare* 100 *ascisse generate dal motore BuildChebyshevAscisse Maker sull'intervallo* $[-10, 20]$ *e sull'intervallo* $[-1, 1]$*.*

Per il codice che implementa le richieste dell'esercizio e produce i seguenti risultati vedere Chebyshev Ascisse Example.

Nei seguenti grafici in *rosso* è rappresentata la "proiezione" delle ascisse generate ed è in accordo con quanto ci aspettiamo, seguono un coseno. In *blu* invece le ascisse sono disposte sul proprio asse, vediamo che non sono equidistanti.

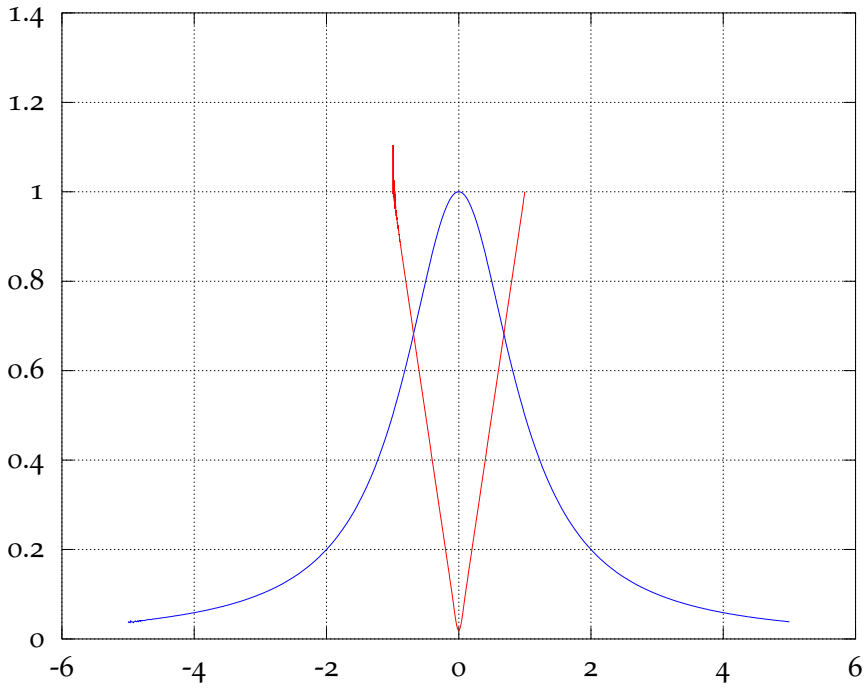

20
15
10
5
0
-5
-10
-10
-5
0
5
10
15
20


1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1

Exercise 4.1.19 (4.15). *Per il testo dell'esercizio consultare il libro di testo.*

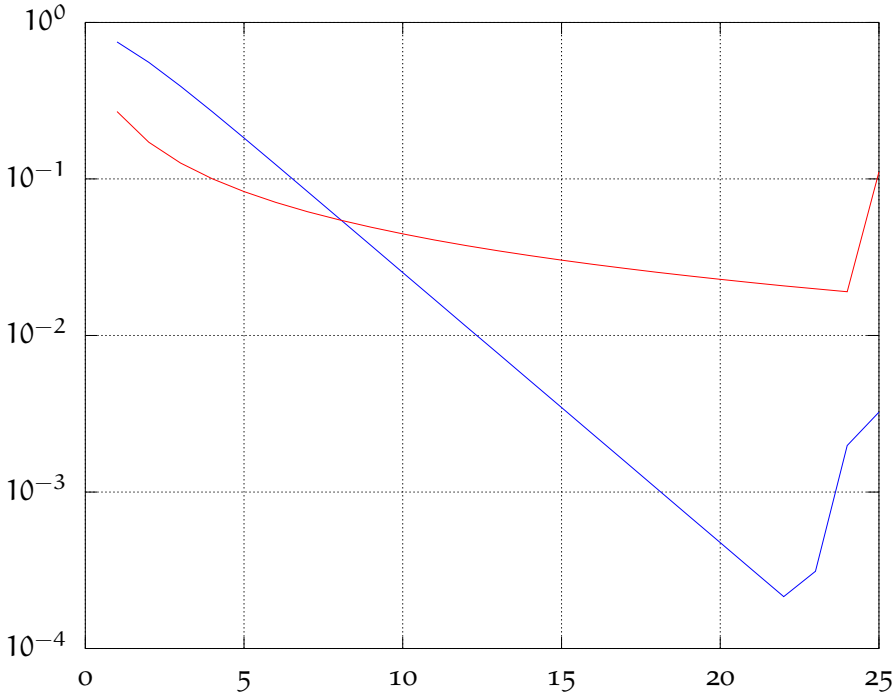
Per il codice che implementa le richieste dell'esercizio e produce i seguenti risultati vedere Exercise 4.15 on textbook.

Riporto il plot delle approssimazioni ottenute considerando il massimo numero di ascisse a disposizione:

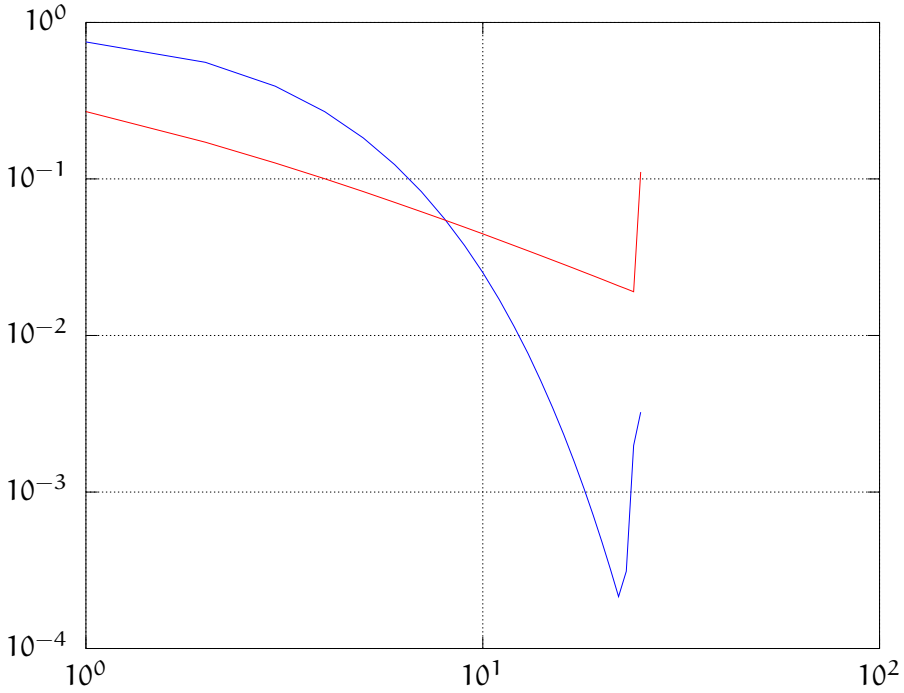


Rispetto al plot dell'esercizio Interpolazione Polinomiale utilizzando le ascisse di Chebyshev siamo riusciti ad avere una rappresentazione delle due curve, cosa che nell'altro plot non è possibile apprezzare. In particolare vicino agli estremi dei rispettivi intervalli (dove si commette il maggior errore) con questa scelta di ascisse è invece possibile avere una buona approssimazione.

Nei seguenti grafici riporto il plot degli errori: in *rosso* è rappresentata la funzione di *Bernstein*, mentre in *blu* è rappresentata la funzione di *Runge*. Questo primo grafico ha l'asse delle ordinate costruito in modo logaritmico:



Riporto lo stesso dataset di errori per le due curve, costruendo entrambi gli assi in modo logaritmico:



Per entrambi i grafici, le valutazioni della rispettiva funzione, del polinomio interpolante e la costruzione del polinomio interpolante sono riferiti ai rispettivi dominii $[a, b]$ definiti nel testo dell'esercizio.

Possiamo notare che gli errori iniziare a crescere dal momento che si utilizza un numero di ascisse superiore a 20, questo include anche errori di rappresentazione (soprattutto nel calcolo delle differenze divise vi possono essere fenomi di cancellazione numerica).

4.2 SPLINE E MINIMI QUADRATI

Exercise 4.2.1 (4.16). *Per il testo dell'esercizio consultare il libro di testo.*

Stiamo costruendo un polinomietto della spline cubica relativo all'i-esimo intervallo $[x_{i-1}, x_i]$. Il sistema che otteniamo utilizzando lo schema *naturale* usa queste condizioni:

$$\left(\forall i \in \{1, \ldots, n-1\}[m_i = s_3^{(2)}(x_i)]\right) \wedge m_0 = 0 \wedge m_n = 0$$

Rappresentiamo in forma matriciale il modello che ci permette di ricavare le incognite m_k:

$$\begin{bmatrix} 2 & \xi_1 & & & & \\ \varphi_2 & 2 & \xi_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \varphi_{n-2} & 2 & \xi_{n-2} & \\ & & & \varphi_{n-1} & 2 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_{n-2} \\ m_{n-1} \end{bmatrix} = 6 \begin{bmatrix} f[x_0, x_1, x_2] \\ f[x_1, x_2, x_3] \\ \vdots \\ f[x_{n-3}, x_{n-2}, x_{n-1}] \\ f[x_{n-2}, x_{n-1}, x_n] \end{bmatrix}$$

Posso fattorizzare la matrice dei coefficienti in questo modo:

$$L = \begin{bmatrix} 1 & & & & \\ l_2 & 1 & & & \\ & \ddots & \ddots & & \\ & & l_{n-2} & 1 & \\ & & & l_{n-1} & 1 \end{bmatrix} \qquad U = \begin{bmatrix} u_1 & \xi_1 & & & \\ & u_2 & \xi_2 & & \\ & & \ddots & \ddots & \\ & & & u_{n-2} & \xi_{n-2} \\ & & & & u_{n-1} \end{bmatrix}$$

Adesso considero il prodotto dell'i-esima riga di L con la $(i-1)$-esima colonna di U:

$$\begin{array}{c} \begin{matrix} i-2 & i-1 & i \\ \downarrow & \downarrow & \downarrow \end{matrix} \\ e_i^T L = \begin{bmatrix} 0 & l_i & 1 \end{bmatrix} \end{array} \qquad U e_{i-1} = \begin{bmatrix} \\ \\ \xi_{i-2} \\ u_{i-1} \\ 0 \\ \\ \\ \\ \end{bmatrix} \begin{matrix} \\ \\ \leftarrow i-2 \\ \leftarrow i-1 \\ \leftarrow i \\ \\ \\ \\ \\ \end{matrix}$$

$$(e_i^T L)(U e_{i-1}) = l_i u_{i-1}$$

Ma dato che la matrice dei coefficienti è diagonale dominante, allora è fattorizzabile LU, quindi deve valere:

$$(e_i^T A e_{i-1}) = \varphi_i = l_i u_{i-1}$$

Adesso considero il prodotto dell'i-esima riga di L con la i-esima colonna di U:

$$\begin{array}{c} \begin{matrix} i-1 & i \\ \downarrow & \downarrow \end{matrix} \\ e_i^T L = \begin{bmatrix} l_i & 1 \end{bmatrix} \end{array} \qquad U e_i = \begin{bmatrix} \\ \\ \xi_{i-1} \\ u_i \\ \\ \\ \\ \end{bmatrix} \begin{matrix} \\ \\ \leftarrow i-1 \\ \leftarrow i \\ \\ \\ \\ \end{matrix}$$

$$(e_i^T L)(U e_i) = l_i \xi_{i-1} + u_i$$

Usando ancora la proprietà di diagonale dominanza deve valere:

$$(\boldsymbol{e}_i^T A \boldsymbol{e}_i) = 2 = l_i \xi_{i-1} + u_i$$

Rimane da far vedere che la matrice dei coefficienti (la chiamo A) è diagonale dominante.

Dimostrazione. Dimostro che sia per righe che per colonne vale l'asserto.

- *per righe* Per definizione vale $\varphi_i + \xi_i = 1$.
  Osservando la struttura della matrice vediamo che le righe $r_k = \boldsymbol{e}_k^T A, \forall k \in \{2, \dots, n-2\}$ contengono esattamente tre elementi, mentre la riga $\boldsymbol{e}_1^T A$ e la riga $\boldsymbol{e}_{n-1}^T A$ ne contengono esattamente due.
  Ma questo insieme di elementi che consideriamo contiene l'elemento diagonale, e i vari φ_k, ξ_k. Posso impostare:

  $$\forall i \in \{1, \dots, n-1\} : \left[A_{ii} = 2 > \sum_{j=1, j\neq i}^{n-1} |A_{ij}| = \varphi_i + \xi_i = 1 \right]$$

  e la condizione di diagonale dominanza per righe viene soddisfatta.

- *per colonne* Per definizione vale $\varphi_i + \xi_i = 1$.
  Osservando la struttura della matrice vediamo che le colonne $c_k = A\boldsymbol{e}_k, \forall k \in \{2, \dots, n-2\}$ contengono esattamente tre elementi, mentre la colonna $A\boldsymbol{e}_1$ e la colonna $A\boldsymbol{e}_{n-1}$ ne contengono esattamente due.
  Ma questo insieme di elementi che consideriamo contiene l'elemento diagonale, e i vari φ_k, ξ_k. Posso impostare:

  $$\forall j \in \{1, \dots, n-1\} : \left[A_{jj} = 2 > \sum_{i=1, i\neq j}^{n-1} |A_{ij}| = \xi_i + \varphi_{i+2} = 1 \right]$$

  e la condizione di diagonale dominanza per colonne viene soddisfatta.

□

Exercise 4.2.2 (4.17)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Riporto la forma matriciale *(4.59)* del libro di testo che ci permette di ricavare le incognite m_k:

$$\begin{bmatrix} 1 & 0 & & & & & & \\ \varphi_1 & 2-\varphi_1 & \xi_1-\varphi_1 & & & & & \\ & \varphi_2 & 2 & \xi_2 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & \varphi_{n-2} & 2 & \xi_{n-2} & & \\ & & & & \varphi_{n-1}-\xi_{n-1} & 2-\xi_{n-1} & \xi_{n-1} & \\ & & & & & 0 & 1 & \end{bmatrix} \begin{bmatrix} m_0+m_1+m_2 \\ m_1 \\ m_2 \\ \vdots \\ m_{n-2} \\ m_{n-1} \\ m_{n-2}+m_{n-1}+m_n \end{bmatrix} = 6 \begin{bmatrix} f[x_0,x \\ f[x_0,x \\ f[x_1,x \\ \\ f[x_{n-3},x_n \\ f[x_{n-2},x \\ f[x_{n-2},x \end{bmatrix}$$

Posso fattorizzare la matrice dei coefficienti in questo modo:

$$L = \begin{bmatrix} 1 & & & & \\ l_2 & 1 & & & \\ & \ddots & \ddots & & \\ & & l_n & 1 & \\ & & & l_{n+1} & 1 \end{bmatrix} \qquad U = \begin{bmatrix} u_1 & 0 & & & & \\ & u_2 & \xi_1 & & & \\ & & u_3 & \xi_2 & & \\ & & & \ddots & \ddots & \\ & & & & u_n & \xi_{n-1} \\ & & & & & u_{n+1} \end{bmatrix}$$

Adesso considero il prodotto dell'i-esima riga di L con la $(i-1)$-esima colonna di U:

$$e_i^T L = \begin{bmatrix} & i-2 & i-1 & i & \\ & \downarrow & \downarrow & \downarrow & \\ & 0 & l_i & 1 & \end{bmatrix} \qquad Ue_{i-1} = \begin{bmatrix} \\ \\ \xi_{i-3} \\ u_{i-1} \\ 0 \\ \\ \\ \end{bmatrix} \begin{matrix} \\ \\ \leftarrow i-2 \\ \leftarrow i-1 \\ \leftarrow i \\ \\ \\ \end{matrix}$$

$$(e_i^T L)(Ue_{i-1}) = l_i u_{i-1}$$

Ma dato che la matrice dei coefficienti è diagonale dominante, allora è fattorizzabile LU, quindi deve valere:

$$(e_i^T A e_{i-1}) = \varphi_{i-1} = l_i u_{i-1}$$

Adesso considero il prodotto dell'i-esima riga di L con la i-esima colonna di U:

$$e_i^T L = \begin{bmatrix} & i-1 & i & \\ & \downarrow & \downarrow & \\ & l_i & 1 & \end{bmatrix} \qquad Ue_i = \begin{bmatrix} \\ \\ \\ \xi_{i-1} \\ u_i \\ \\ \\ \end{bmatrix} \begin{matrix} \\ \\ \\ \leftarrow i-1 \\ \leftarrow i \\ \\ \\ \end{matrix}$$

$$(e_i^T L)(Ue_i) = l_i \xi_{i-1} + u_i$$

Usando ancora la proprietà di diagonale dominanza deve valere:

$$(e_i^T A e_i) = 2 = l_i \xi_{i-1} + u_i$$

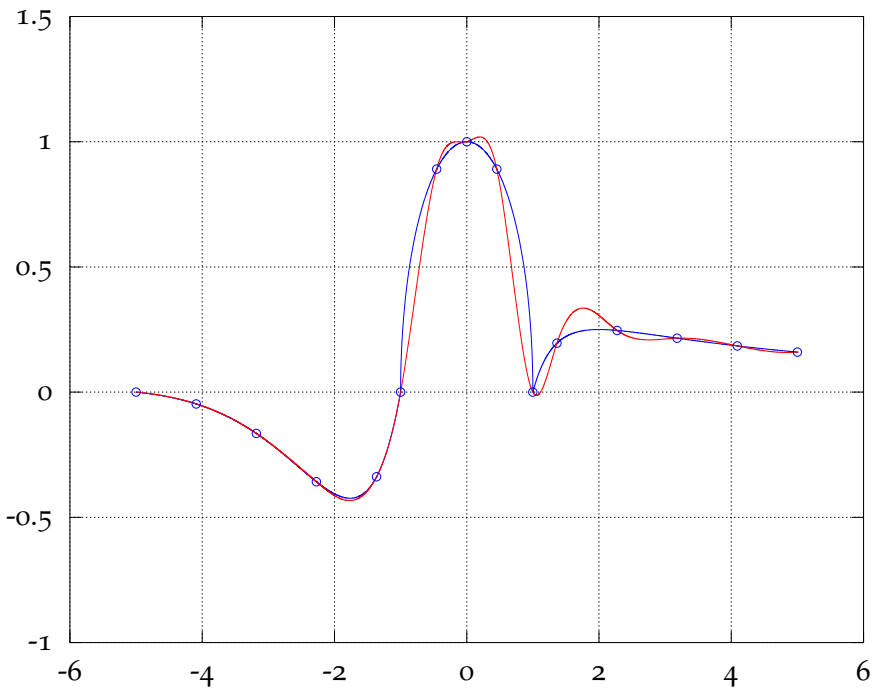
prestando attenzione per la prima, seconda, penultima e ultima riga. Una volta calcolati $m_i, i \in 1, \ldots, n-1$ sarà possibile calcolare anche m_0, m_n risolvendo rispettivamente:

$$m_0 + m_1 + m_2 = 6f[x_0, x_1, x_2]$$
$$m_{n-2} + m_{n-1} + m_n = 6f[x_{n-2}, x_{n-1}, x_n]$$

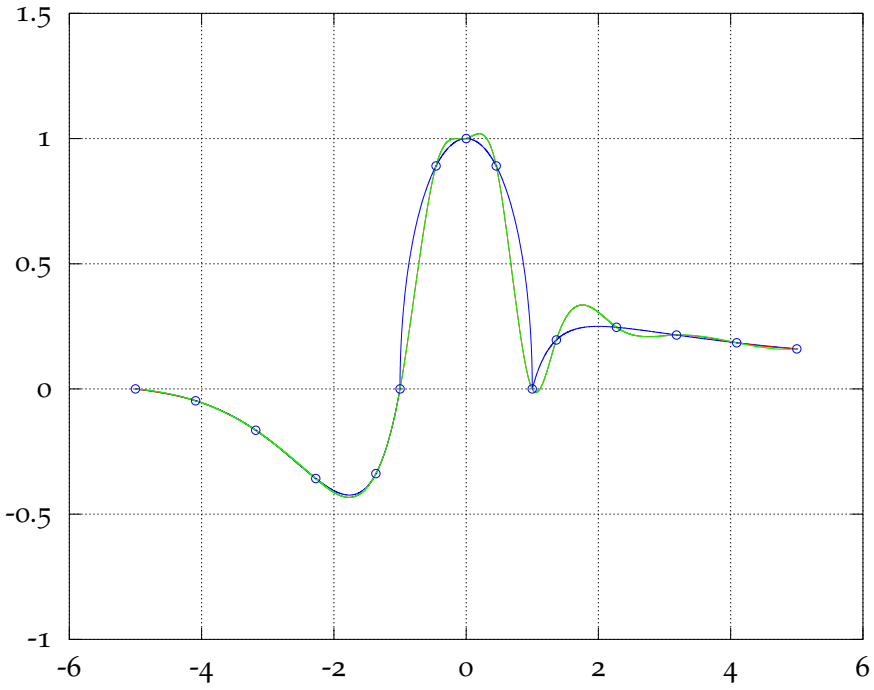
Exercise 4.2.3. *Provare ad interpolare la seguente funzione, in una partizione composta da 12 ascisse di interpolazione:*

$$f(x) = \begin{cases} (x+5)(x+1)e^x & \text{se } x \leqslant -1 \\ \sqrt{1-x^2} & \text{se } -1 < x \leqslant 1 \\ \frac{x-1}{x^2} & \text{se } 1 < x \end{cases}$$

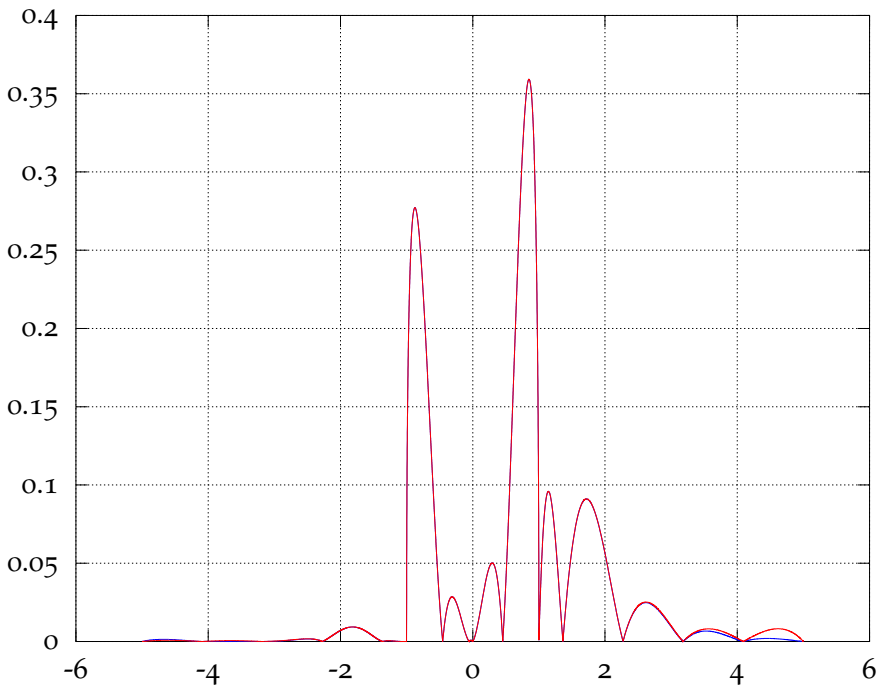
Per l'implementazione del codice di questo esercizio vedere il codice Spline stress. Il motore *Octave* crea la seguente spline (in rosso):



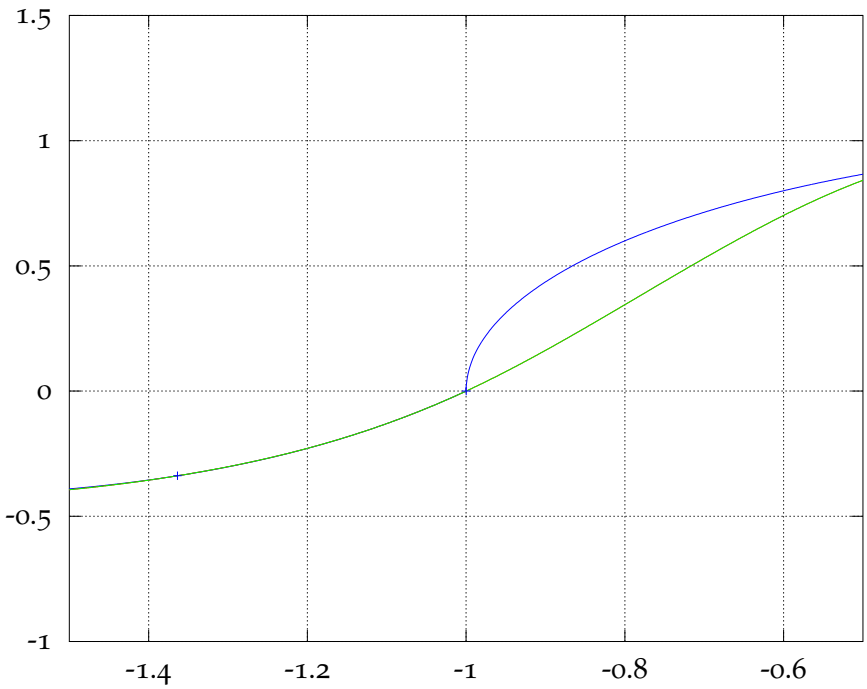
Nel seguente plot invece riporto il mio output, rappresentando due interpolazioni con spline cubiche, in verde utilizzando lo schema *not-a-knot*, mentre in rosso lo schema *naturale*.



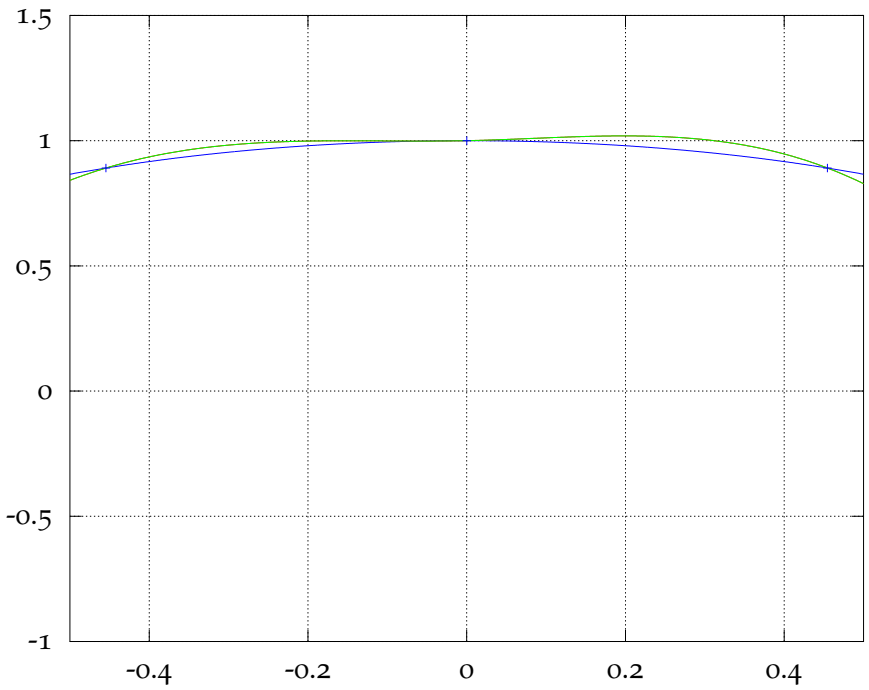
Nel seguente plot rappresento il modello degli errori delle precedenti interpolazioni con spline cubiche, in rosso riferito allo schema *not-a-knot*, mentre in blu riferito allo schema *naturale*:



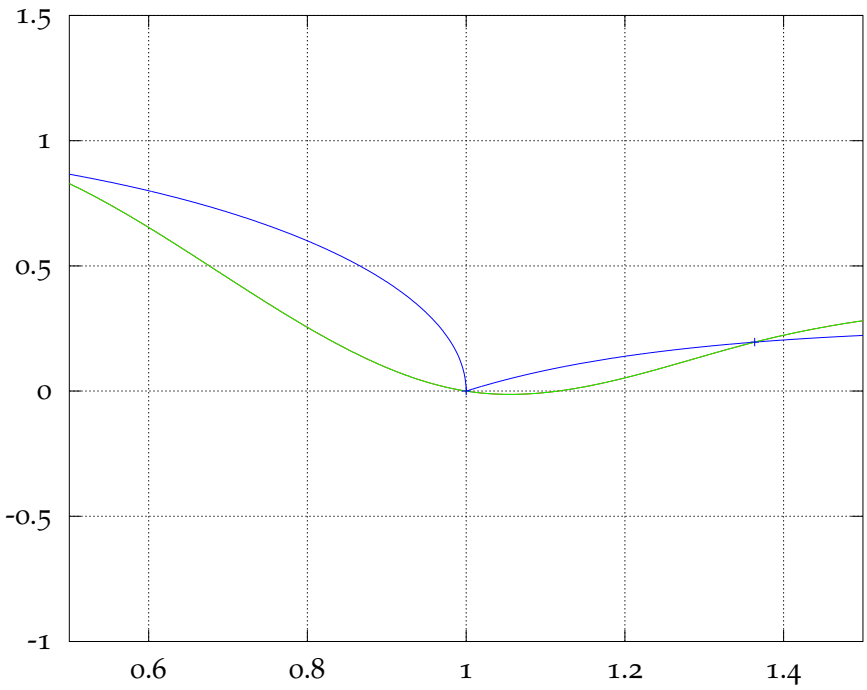
Riporto uno zoom per ognuno dei punti di irregolarità: In $x = -1$:



In $x = 0$:

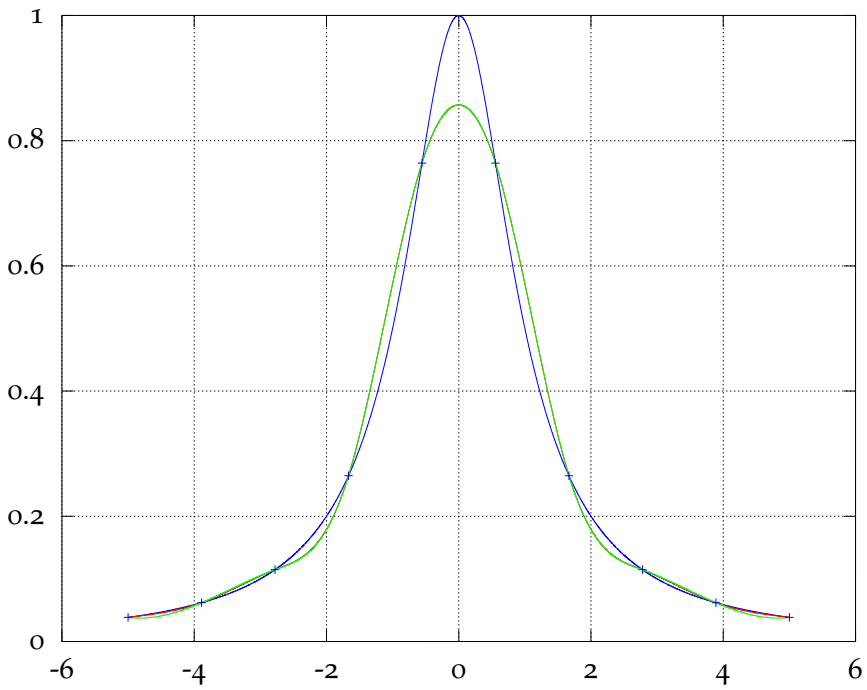


In $x = 1$:

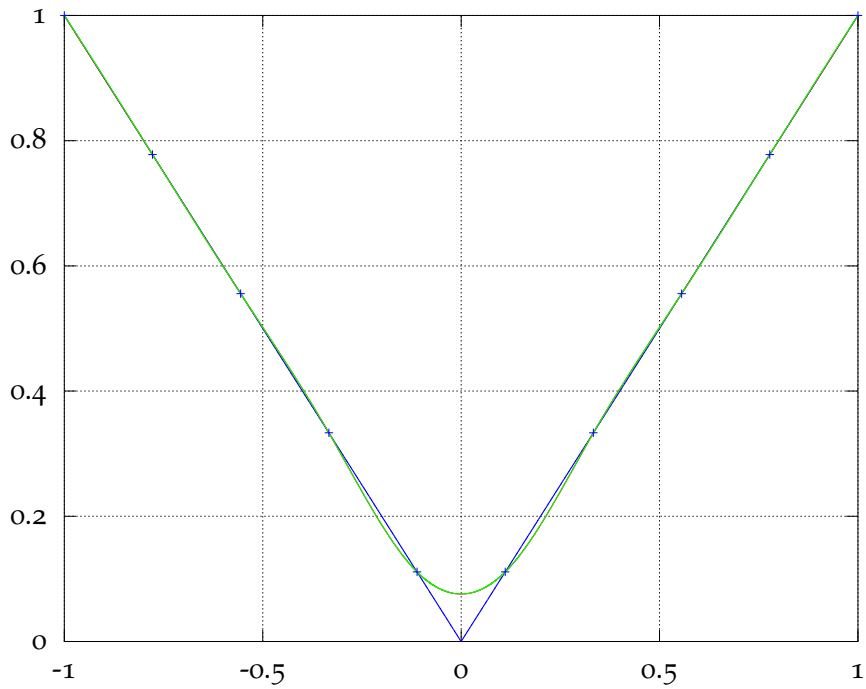


Exercise 4.2.4 (4.19)**.** *Per il testo dell'esercizio consultare il libro di testo.*

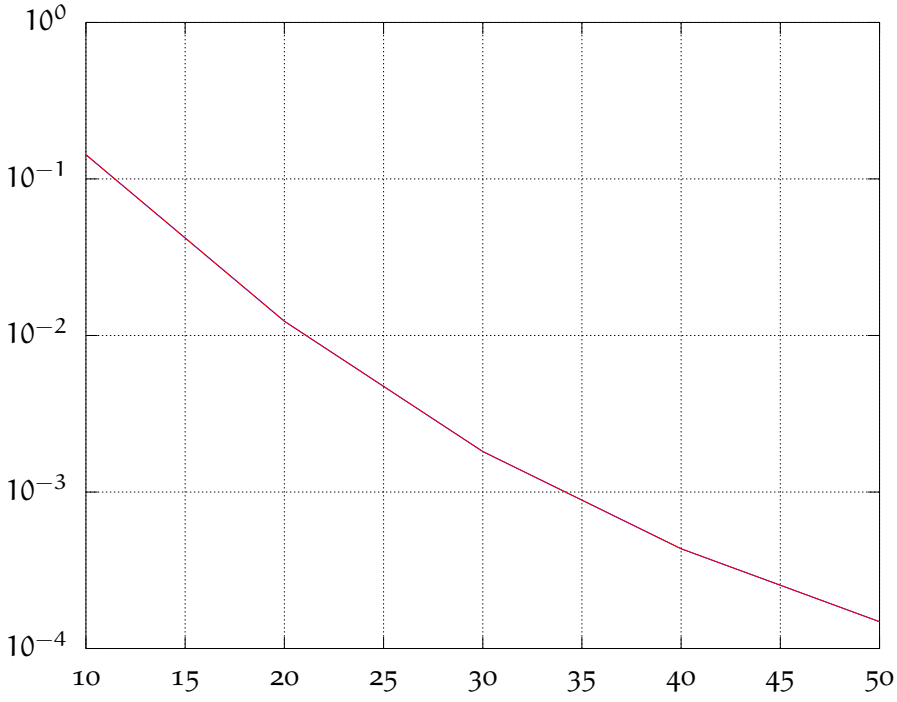
Riporto il plot delle interpolazioni della funzione di *Runge*, in verde l'interpolazione eseguita con lo schema *not-a-knot*, in rosso l'interpolazione eseguita con lo schema *normale*, in blu la curva originale. Questo plot è stato generato con il codice Exercise 4.19 - Runge interpolationPlotter usando il motore Exercise 4.19 - Runge interpolation.



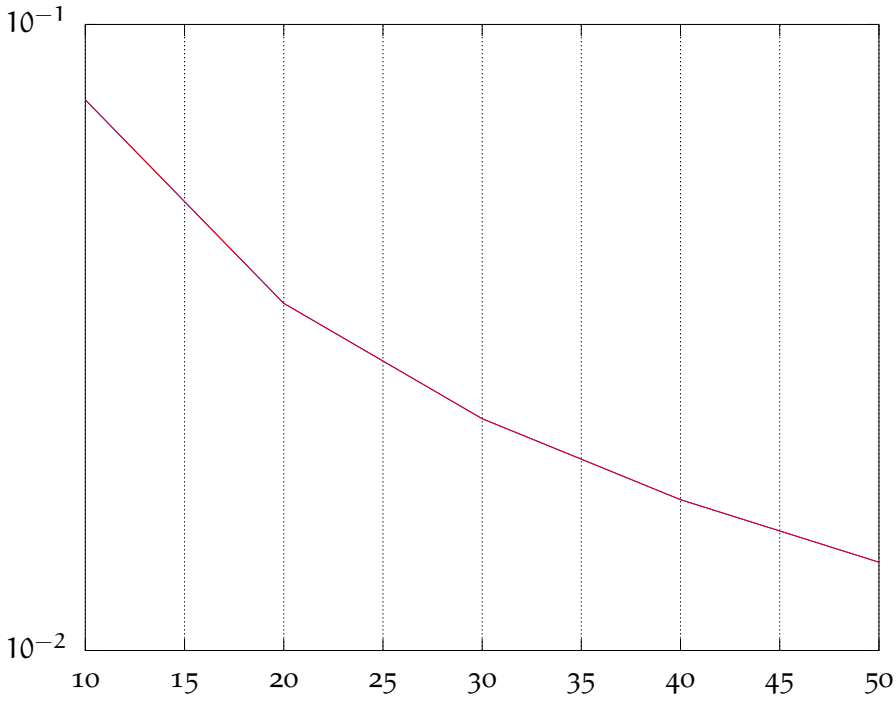
Riporto il plot delle interpolazioni della funzione di *Bernstein*, in verde l'interpolazione eseguita con lo schema *not-a-knot*, in rosso l'interpolazione eseguita con lo schema *normale*, in blu la curva originale. Questo plot è stato generato con il codice Exercise 4.19 - Bernstein interpolation Plotter usando il motore Exercise 4.19 - Bernstein interpolation.



Riporto il plot degli errori che si commettono nelle interpolazioni della funzione di *Runge*, in rosso l'errore dell'interpolazione eseguita con lo schema *not-a-knot*, in blu l'interpolazione eseguita con lo schema *normale*. Questo plot è stato generato con il codice Exercise 4.19 on textbook

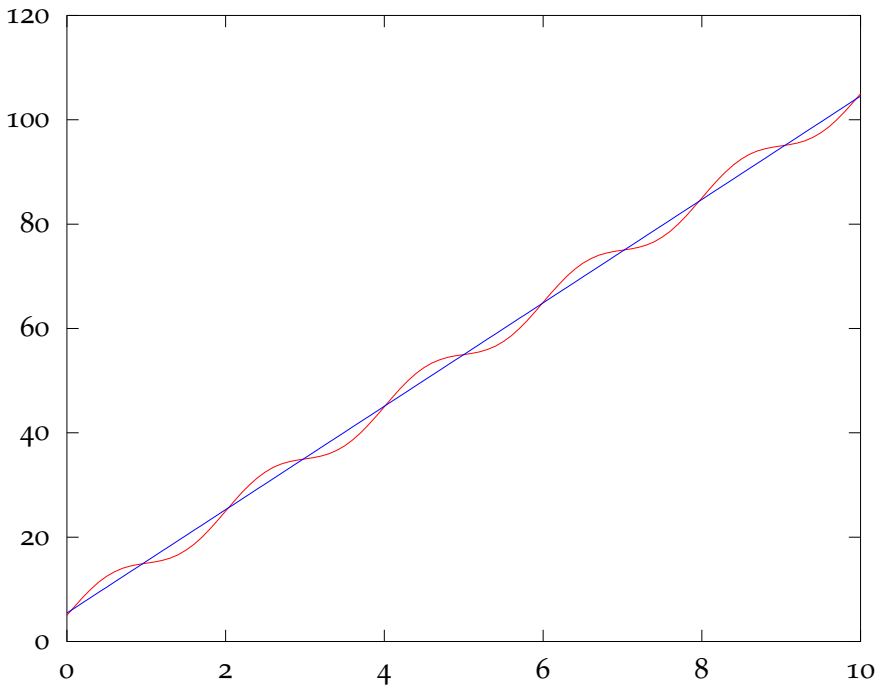


Riporto il plot degli errori che si commettono nelle interpolazioni della funzione di *Bernstein*, in rosso l'errore dell'interpolazione eseguita con lo schema *not-a-knot*, in blu l'interpolazione eseguita con lo schema *normale*. Questo plot è stato generato con il codice Exercise 4.19 on textbook



Exercise 4.2.5 (4.21). *Per il testo dell'esercizio consultare il libro di testo.*

Per il codice che implementa l'esercizio vedere Exercise 4.21 on textbook. Il seguente plot renderizza in rosso la curva originale, mentre in blu la retta ai minimi quadrati con equazione $f(x) = 9.9081x + 5.4596$, con la variabile gamma $= 2.5$:



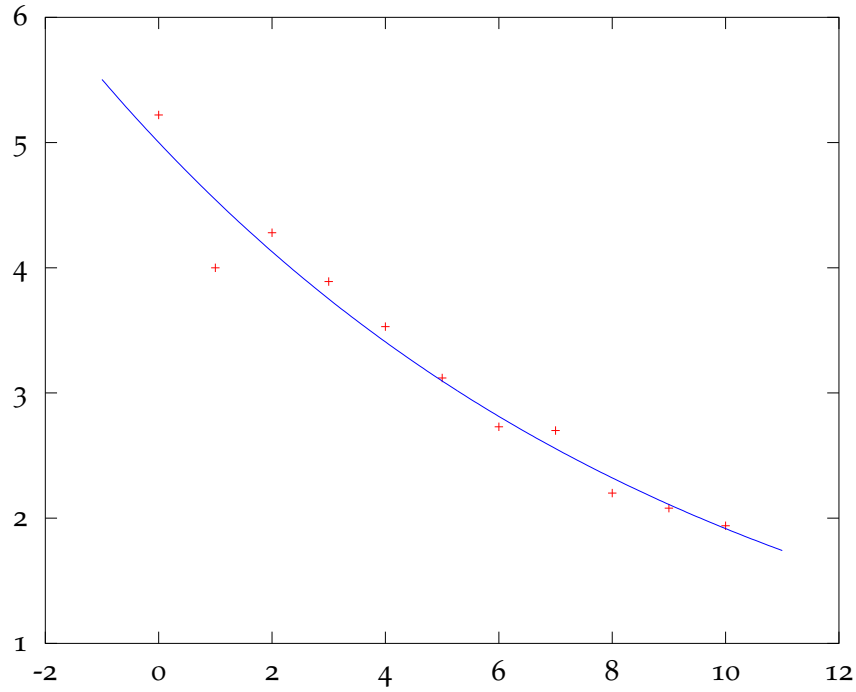
Exercise 4.2.6 (4.22). *Per il testo dell'esercizio consultare il libro di testo.*

Trasformo il modello esponenziale in una polinomiale con i seguenti passaggi (l'idea è quella di passare ai logaritmi entrambi i membri e successivamente fare dei cambi di variabile):

$$
\begin{aligned}
y &= \alpha e^{-\lambda t} \\
\log y &= \log \alpha e^{-\lambda t} \\
\log y &= \log \alpha + \log e^{-\lambda t} \\
\log y &= \log \alpha - \lambda t \\
\rho &= \beta - \lambda t \quad \text{se } \rho = \log y \wedge \beta = \log \alpha
\end{aligned}
$$

Adesso che ci siamo riportati ad un modello polinomiale posso implementare il codice Exercise 4.22 on textbook.

La retta ai minimi quadrati per il modello polinomiale ricavata ha equazione $\rho = -0.095896x + 1.609602$, pertanto passando agli esponenti entrambi i membri (per annullare il cambio di variabile $y \to \rho$) si ottiene il seguente plot, dove gli obiettivi in rosso rappresentano i dati sperimentali, mentre la curva in blu è la curva ai minimi quadrati che possiamo ottenere su una partizione generata con `linspace(-1, 11, 200)'`:


6
5
4
3
2
1
-2
0
2
4
6
8
10
12

5 FORMULE DI QUADRATURA

5.1 FORMULE COMPOSITE

Exercise 5.1.1 (5.1). *Per il testo dell'esercizio consultare il libro di testo.*

Per la *(5.2)* del libro di testo vale $\kappa = b - a = e^{21} \gg 1$, quindi il problema risulta molto mal condizionato.

Exercise 5.1.2 (5.2). *Per il testo dell'esercizio consultare il libro di testo.*

Proof of Coefficienti formula trapezi. Dalla definizione della formula generica di *Newton-Cotes* istanzio per il caso dei trapezi:

$$I_1(f) = (b-a)\sum_{k=0}^{1} c_{k1} f_k$$

$$c_{01} = \int_0^1 \prod_{j=0, j\neq 0}^{1} \frac{t-j}{0-j} dt = \int_0^1 \frac{t-1}{0-1} dt = -\int_0^1 t dt + \int_0^1 dt = -\left.\frac{t^2}{2}\right|_0^1 + t|_0^1 = -\frac{1}{2} + 1 = \frac{1}{2}$$

$$c_{11} = \int_0^1 \prod_{j=0, j\neq 1}^{1} \frac{t-j}{1-j} dt = \int_0^1 \frac{t-0}{1-0} dt = \int_0^1 t dt = \left.\frac{t^2}{2}\right|_0^1 = \frac{1}{2}$$

□

Proof of Coefficienti formula Simpson. Dalla definizione della formula generica di *Newton-Cotes* istanzio per il caso di Simpson:

$$I_2(f) = \frac{b-a}{2}\sum_{k=0}^{2} c_{k2} f_k$$

$$c_{02} = \int_0^2 \prod_{j=0, j\neq 0}^{2} \frac{t-j}{0-j} dt = \int_0^2 \frac{(t-1)(t-2)}{(0-1)(0-2)} dt = \frac{1}{2}\int_0^2 t^2 - 3t + 2 dt =$$

$$= \frac{1}{2}\left(\int_0^2 t^2 dt - 3\int_0^2 t dt + 2\int_0^2 dt\right) = \frac{1}{2}\left(\left.\frac{t^3}{3}\right|_0^2 - 3\left.\frac{t^2}{2}\right|_0^2 + 2\,t|_0^2\right) =$$

$$= \frac{1}{2}\left(\frac{2^3}{3} - \frac{3}{2}2^2 + 2^2\right) = \frac{1}{2}2^2\left(\frac{2}{3} - \frac{3}{2} + 1\right) = 2\frac{4-9+6}{6} = \frac{1}{3}$$

$$c_{12} = \int_0^2 \prod_{j=0, j\neq 1}^{2} \frac{t-j}{1-j} dt = \int_0^2 \frac{(t-0)(t-2)}{(1-0)(1-2)} dt = -\int_0^2 t^2 - 2t dt =$$

$$= -\int_0^2 t^2 dt + 2\int_0^2 t dt = -\left.\frac{t^3}{3}\right|_0^2 + 2\left.\frac{t^2}{2}\right|_0^2 = -\frac{2^3}{3} + 2^2 = 2^2\left(1 - \frac{2}{3}\right) = \frac{4}{3}$$

$$c_{22} = \int_0^2 \prod_{j=0, j\neq 2}^{2} \frac{t-j}{2-j} dt = \int_0^2 \frac{(t-0)(t-1)}{(2-0)(2-1)} dt = \frac{1}{2}\int_0^2 t^2 - t dt =$$
$$= \frac{1}{2}\left(\int_0^2 t^2 dt - \int_0^2 t dt\right) = \frac{1}{2}\left(\frac{t^3}{3}\Big|_0^2 - \frac{t^2}{2}\Big|_0^2\right) = \frac{1}{2}\left(\frac{2^3}{3} - \frac{2^2}{2}\right) = \left(\frac{2^2}{3} - 1\right) = \frac{1}{3}$$

□

Exercise 5.1.3 (5.3). *Per il testo dell'esercizio consultare il libro di testo.*

Dimostrazione. Parto dalla definizione dell'*errore di quadratura*:

$$E_n(f) = \nu_n \frac{f^{(n+k)}(\xi)}{(n+k)!}\left(\frac{b-a}{n}\right)^{n+k+1}$$

Per il *metodo dei trapezi* si fissa $n = 1$, che implica la scelta di $k = 1$:

$$E_1(f) = \nu_1 \frac{f^{(2)}(\xi)}{2}(b-a)^3$$

Ricavo ν_1:

$$\nu_1 = \int_0^1 \prod_{j=0}^{1} (t-j) dt = \int_0^1 (t-0)(t-1) dt = \int_0^1 t^2 - t dt =$$
$$= \left(\int_0^1 t^2 dt - \int_0^1 t dt\right) = \left(\frac{t^3}{3}\Big|_0^1 - \frac{t^2}{2}\Big|_0^1\right) = \frac{1}{3} - \frac{1}{2} = -\frac{1}{6}$$

Andando a sostituire nel metodo si verifica l'uguaglianza esposta nel testo *(5.10)*:

$$E_1(f) = -\frac{1}{6}\frac{f^{(2)}(\xi)}{2}(b-a)^3 = -\frac{f^{(2)}(\xi)}{12}(b-a)^3$$

Per il *metodo dei Simpson* si fissa $n = 2$, che implica la scelta di $k = 2$:

$$E_2(f) = \nu_2 \frac{f^{(4)}(\xi)}{4!}\left(\frac{b-a}{2}\right)^5$$

Ricavo ν_2:

$$\nu_2 = \int_0^2 t \prod_{j=0}^{2} (t-j) dt = \int_0^2 t(t-0)(t-1)(t-2) dt = \int_0^2 t^2(t-1)(t-2) dt =$$
$$= \int_0^2 t^2(t^2 - 3t + 2) dt = \int_0^2 t^4 dt - 3\int_0^2 t^3 dt + 2\int_0^2 t^2 dt = \frac{t^5}{5}\Big|_0^2 - 3\,\frac{t^4}{4}\Big|_0^2 + 2\,\frac{t^3}{3}\Big|_0^2 =$$
$$= \frac{2}{5}2^4 - \frac{3}{4}2^4 + \frac{1}{3}2^4 = \left(\frac{24 - 45 + 20}{60}\right)2^4 = -\frac{1}{2^2 \cdot 3 \cdot 5}2^4 = -\frac{2^2}{3 \cdot 5}$$

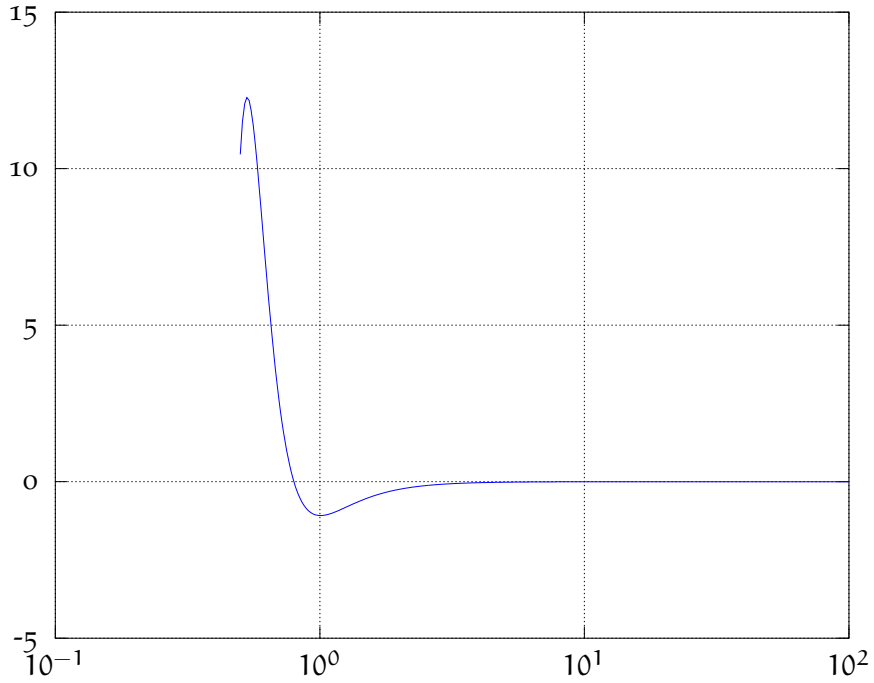
Andando a sostituire nel metodo si verifica l'uguaglianza esposta nel testo *(5.11)*:

$$E_2(f) = -\frac{2^2}{3 \cdot 5} \frac{f^{(4)}(\xi)}{2^2 \cdot 3 \cdot 2} \left(\frac{b-a}{2}\right)^5 = -\frac{f^{(4)}(\xi)}{90} \left(\frac{b-a}{2}\right)^5$$

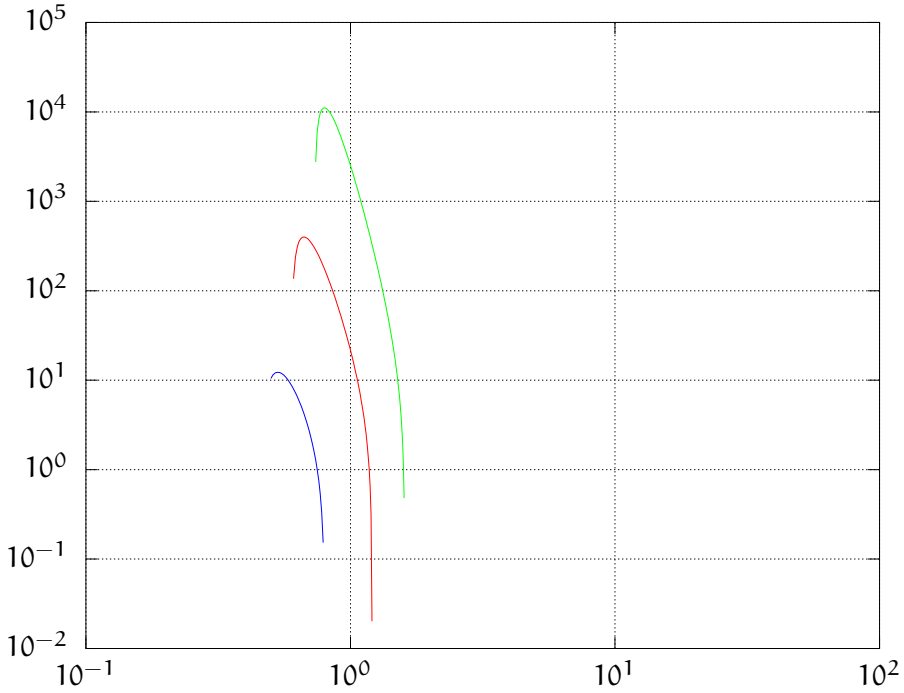
□

Exercise 5.1.4. *Rappresentare la funzione integranda* 5.17 *esposta nel libro di testo.*

Rappresento la funzione utilizzando l'asse delle ascisse in modo logaritmico.



Rappresento la derivata seconda (in rosso) e la derivata quarta (in verde), questo plot utilizza per entrambi gli assi una scala logaritmica:



I precedenti plot si possono ricavare utilizzando il codice Plotter della funzione (5.17).

Exercise 5.1.5 (5.4)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Vedere il codice Exercise 5.4 on textbook - Trapezi composita.

Exercise 5.1.6 (5.5)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Vedere il codice Exercise 5.5 on textbook - Simpson composita.

Exercise 5.1.7 (5.6)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Vedere il codice Adaptive Trapezi.

Exercise 5.1.8 (5.7)**.** *Per il testo dell'esercizio consultare il libro di testo.*

Vedere il codice Adaptive Simpson.

Exercise 5.1.9 (5.9, 5.10 tabelle per schemi compositi)**.** *Per i testi degli esercizi consultare il libro di testo.*

Il seguente output è generato usando il codice Solver exercise 5.9 on textbook.

```
[trapeziCompositeResultTable, simpsonCompositeResultTable]
    = exercise59solver
trapeziCompositeResultTable =
```

```
3
    1.0000e+03   6.6401e-01  -1.7771e+00
    2.0000e+03   7.3077e-01  -4.4427e-01
    3.0000e+03   7.4507e-01  -1.9745e-01
    4.0000e+03   7.5020e-01  -1.1107e-01
8   5.0000e+03   7.5260e-01  -7.1083e-02
    6.0000e+03   7.5391e-01  -4.9363e-02
    7.0000e+03   7.5470e-01  -3.6267e-02
    8.0000e+03   7.5522e-01  -2.7767e-02
    9.0000e+03   7.5557e-01  -2.1939e-02
13  1.0000e+04   7.5582e-01  -1.7771e-02

simpsonCompositeResultTable =

    1.0000e+03   7.0132e-01  -1.3618e-01
18  2.0000e+03   7.5303e-01  -8.5114e-03
    3.0000e+03   7.5617e-01  -1.6813e-03
    4.0000e+03   7.5668e-01  -5.3196e-04
    5.0000e+03   7.5681e-01  -2.1789e-04
    6.0000e+03   7.5686e-01  -1.0508e-04
23  7.0000e+03   7.5688e-01  -5.6719e-05
    8.0000e+03   7.5689e-01  -3.3248e-05
    9.0000e+03   7.5689e-01  -2.0756e-05
    1.0000e+04   7.5690e-01  -1.3618e-05
```

nelle precedenti matrici si ha valorizzato nella prima colonna il numero di sottointervalli, nella seconda
l'approssimazione dell'integrale e nella terza l'errore che si commette.

Exercise 5.1.10 (5.9, 5.10 tabelle per schemi adattivi)**.** *Per i testi degli esercizi consultare il libro di testo.*

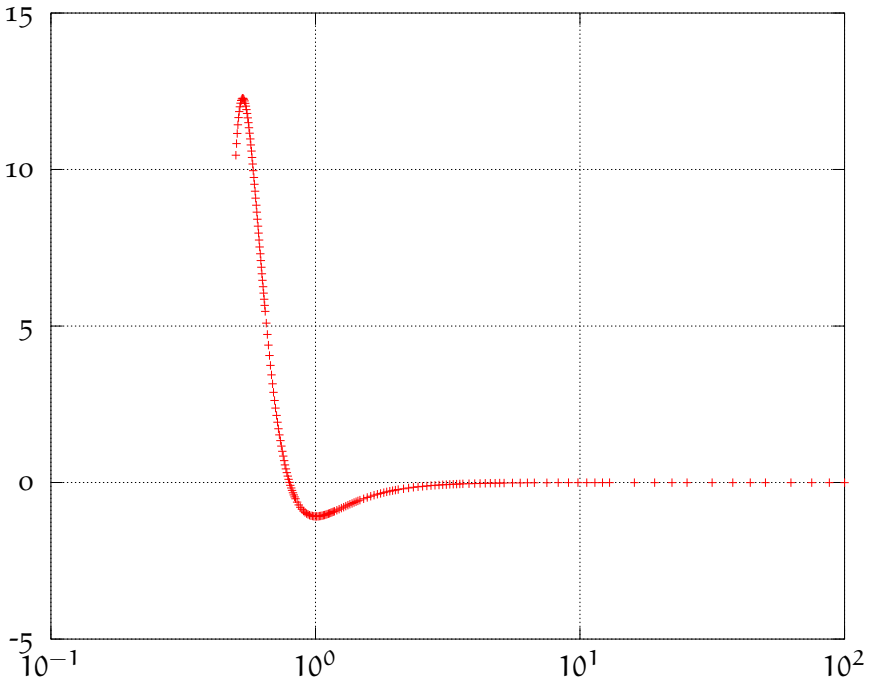
Il seguente output è generato usando il codice Solver exercise 5.10 on textbook.

```
octave:2> [trapeziAdaptiveResultTable,
   simpsonAdaptiveResultTable] = exercise510solver
trapeziAdaptiveResultTable =

4   1.0000e-01   5.8198e-03   1.5900e+02
    1.0000e-02   1.1881e-03   4.7100e+02
    1.0000e-03   2.9774e-04   1.5670e+03
    1.0000e-04   5.8566e-05   4.8510e+03
    1.0000e-05   5.5259e-06   1.4823e+04
9

simpsonAdaptiveResultTable =
```

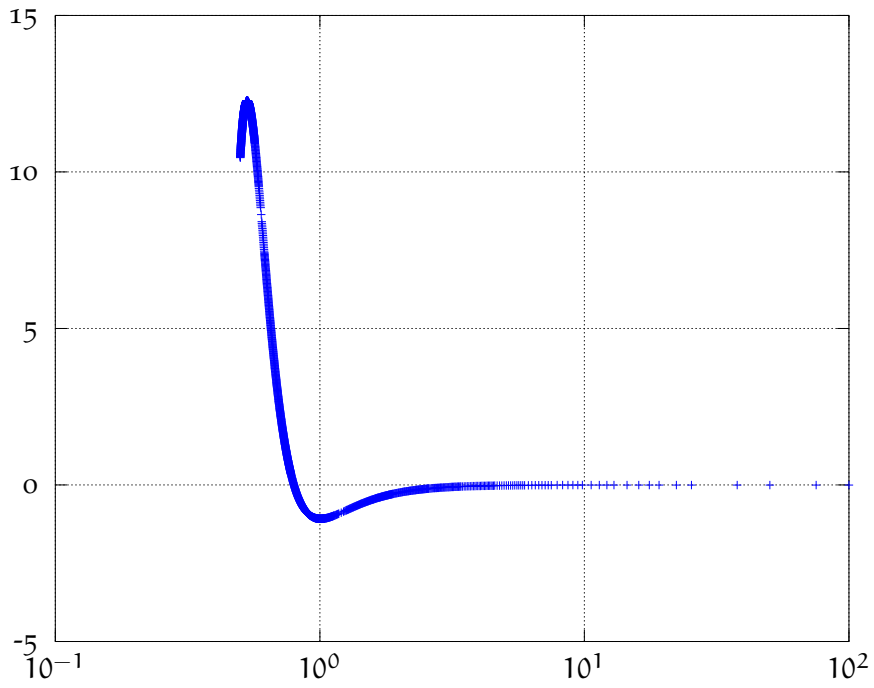
```
14  1.0000e-01    5.4037e-04    4.9000e+01
    1.0000e-02    2.1963e-04    6.5000e+01
    1.0000e-03    4.8899e-05    9.3000e+01
    1.0000e-04    1.2482e-05    1.8100e+02
    1.0000e-05    3.3642e-06    3.0900e+02
```

nelle precedenti matrici si ha valorizzato nella prima colonna la tollerenza richiesta, nella seconda l'errore che si commette e nella terza il numero di punti necessari per raggiungere la precisione richiesta.

Rappresento i punti sulla funzione integranda utilizzati dal metodo adattivo di Simpson per raggiungere una precisione di 10^{-4}:



Rappresento i punti sulla funzione integranda utilizzati dal metodo adattivo dei Trapezi per raggiungere una precisione di 10^{-3}:


15
10
5
0
-5
10^{-1}
10^{0}
10^{1}
10^{2}

6 SORGENTI OCTAVE

6.1 METODO ITERATIVO

```
## This program capture an iterative method that converges
    to sqrt(2).
## It take two initial step to start the creation of the
   approximation
## succession's values.
## It display at each step the value of the created
   succession value and
## show the difference with the previous one.
## The least operation that is done is to show the
   distance from the
## Octave value of sqrt(2).
## The parameters order's is important: the first is the
   first step,
## the second is the second step, be careful before swap
   their position!

function iterative (val1, val2)
  format long e
  next = val2;
  while (abs(sqrt(2) - next) > 1e-12)
    next = ((val1 * val2) + 2) / (val1 + val2)
    disp (strcat("Difference with last step: ", num2str(
       abs(val2 - next), 15)))

    val1 = val2;
    val2 = next;
  end

  disp (strcat("Difference with sqrt(2): ", num2str(abs(
     sqrt(2) - next), 15)))

  format
endfunction
```

6.2 UTILITY FUNCTIONS

6.2.1 *invokeDelegate*

```
## invokeDelegate
##
## Questo oggetto matematico incapsula tutte le funzioni che verranno utilizzate
## nelle applicazioni descritte nell'elaborato per quanto riguardo i metodi
## iterativi.
## Per definizione della gestione delle funzioni di Octave e' possibile invocare
## solo la prima funzione di questo file, quindi i client di questo oggetto
## matematico riescono ad usufruire solo della funzione "invokeDelegate".
## Per evitare di costruire un file per ogni funzione che viene usata dalle
## applicazioni, ho costruito questo oggetto come un "invocatore" di funzioni mono-
## parametro, in modo da inserire in questo file tutte le funzioni di cui ho
## bisogno nelle applicazioni dei vari metodi. Questo oggetto rappresenta un
## entry-point per le funzioni successive alla sua dichiarazione, le quali vengono
## nascoste da invokeDelegate.
##
## Input:
##  - function_name: nome della funzione che si vuole invocare, in formato stringa
##    deve essere una tra quelle elencate dopo la funzione "invokeDelegate".
##    Ognuna di queste funzioni deve avere un solo parametro
##  - argument: argomento a cui applicare la funzione "function_name", puo'
##    essere di qualsiasi tipo, basta che la relativa funzione riesca a lavorarlo
## Output:
```

```
##  - ret: array di oggetti, ognuno dei quali e' l'
    applicazione della funzione
##    "function_name" all'i-esimo elemento dell'array "
    argument" passato
##    come parametro.

function [ ret ] = invokeDelegate (function_name, argument
    )
  ret = [];
  # ho implementato questo ciclo perche la feval non era
     trasparente come
  # una funzione scalare, ovvero non funzionava quando '
     argument' viene passato
  # come array.
  # questo ciclo mi serve quando uso comandi come questo
  # plot(xSingleZero, ySingleZero, "c", ascisse,
     invokeDelegate('singleZero', ascisse), "b+", prepX,
     prepY, "r")
  # che utilizzano direttamente questo oggetto passando un
     vettore di argomenti (ascisse nell'esempio)
  for i = 1:length(argument)
    value = feval(function_name, argument(i));
    ret = [ret value]; # colleziono ogni valore computato
  end
endfunction

function [ ret ] = singleZero (x)
  ret = (x - 3)*(x + 4) - 3;
endfunction

function [ ret ] = singleZeroDerivative (x)
  ret = (x + 4) + (x - 3);
endfunction

function [ ret ] = functionWithNoRealZero (x)
  ret = x^(2) + x + 1;
endfunction

function [ ret ] = functionWithNoRealZeroDerivative (x)
  ret = 2*x + 1;
endfunction

function [ ret ] = functionNewtonRecursion (x)
```

```
  ret = x^(3) - 5 * x;
endfunction

function [ ret ] = functionNewtonRecursionDerivative (x)
  ret = 3 * x^(2) - 5;
endfunction

# funzione relativa all'esercizio 2.5
function [ ret ] = function25first (x)
  ret = (x - 1)^(10);
endfunction

function [ ret ] = function25firstDerivative (x)
  ret = 10 * (x - 1)^(9);
endfunction

# funzione relativa all'esercizio 2.5
function [ ret ] = function25second (x)
  ret = (x - 1)^(10) * exp(x);
endfunction

# funzione relativa all'esercizio 2.5
function [ ret ] = function25secondDerivative (x)
  ret = exp(x) * (10 * (x - 1)^(9) + (x - 1)^(10));
endfunction

function [ ret ] = chordConvergenceFunction (x)
  ret = (x - 3)^(3) + x;
endfunction

function [ ret ] = chordConvergenceFunctionDerivative (x)
  ret = 3 * (x - 3)^(2) + 1;
endfunction

function [ ret ] = secantConvergenceFunction (x)
  ret = x^(2) - x;
endfunction

function [ ret ] = secantConvergenceFunctionDerivative (x)
  ret = 2 * x - 1;
endfunction

function [ ret ] = exercise27Function (x)
```

```
  ret = x - cos(x);
endfunction

function [ ret ] = exercise27FunctionDerivative (x)
  ret = 1 + sin(x);
endfunction
```

6.2.2 *prepareForPlottingMethodSegments*

```
## prepareForPlottingMethodSegments
## questa funzione ha il compito di restituire due nuovi vettori
## per effettuare la stampa dei segmenti che congiungono gli zeri
## calcolati dal metodo con la successiva valutazione.
## Duplicano ogni elemento del vettore delle ascisse in quanto
## per ogni x, avro' uno zero approssimato (che viene calcolato dal metodo)
## ed una valutazione di funzione in quel punto; per poter usare
## in modo trasparente la funzione di Octave plot, devo
## restituire questi due punti identici ma doppioni
## per poi congiungere con una spezzata da una ordinata zero
## (per riferirmi sempre all'asse delle ascisse) ed una coordinata uguale
## alla valutazione della funzione.
##
## Input:
##    - ascisse: vettore di ascisse che si vuole preparare per rappresentare
##      il comportamento del metodo (la curva rossa descritta nell'elaborato)
##    - f_name: nome della funzione che si vuole applicare ad ogni punto
##      contenuto nell'array ascisse. Se f_name rappresenta un nome di funzione
##      diverso da "invokeDelegate", ovvero rappresenta in modo esplicito una funzione,
##      allora passare il parametro "f_argument" = ""
```

```
##
##      Devo prestare attenzione se la funzione
##      da valutare e' "invokeDelegate", in quanto essendo per costruzione una
##      funzione, posso chiedere di usarla per prepare vettori per un grafico.
##      In questo caso, deve passare anche il nome della funzione che il delegato
##      dovra' invocare, introducendo il parametro f_argument.
##    - f_argument, rappresenta il nome della funzione che l'oggetto "invokeDelegate"
##      dovra' invocare, quindi sara' valorizzato solo nel caso si utilizzi
##      un delegato, quindi una rappresentazione implicita della funzione che si vuole
##      utilizzare.

function [ preparedXs, preparedYs ] = prepareForPlottingMethodSegments(ascisse, f_name, f_argument)
preparedXs = [];
preparedYs = [];
for i = 1:length(ascisse)
  # duplico l'oggetto ascisse(i) in modo da avere il punto (ascisse(i), 0) e
  # il punto (ascisse(i), feval(f_name, ascisse(i)) per poter disegnare una spezzata
  # con la funzione plot.
  preparedXs = [preparedXs ascisse(i)];
  preparedXs = [preparedXs ascisse(i)];

  # associo al primo dei due oggetti duplicati ascisse(i) il valore 0 (come descritto
  # nel commento precedente)
  preparedYs = [preparedYs 0];

  if strcmp(f_name, "invokeDelegate") == 1
    # se sto usando un delegato allora invoco il delgato, passando il nome
    # della funzione che il delegato (f_name == "invokeDelegate") deve invocare (f_argument)
```

```
    # con argomento ascisse(i) su cui "f_argument" verra'
      applicata.
    # Ho un doppio livello di indirezione: feval >
      invokeDelegate
    preparedYs = [preparedYs feval(f_name, f_argument,
      ascisse(i))];
  else
    # f_name rappresenta in modo diretto una funzione,
      quindi posso utilizzare
    # la feval nel modo classico, invocando f_name con
      argomento ascisse(i)
    preparedYs = [preparedYs feval(f_name, ascisse(i))];
  end
end
endfunction
```

6.2.3 *prepareForPlottingSecantMethodSegments*

```
## prepareForPlottingSecantMethodSegments
## questa funzione ha il compito di restituire due nuovi
   vettori
## per effettuare la stampa dei segmenti che congiungono
   gli zeri
## calcolati dal metodo delle secanti con la successiva
   valutazione.
##
## Input:
##    - ascisse: vettore di ascisse che si vuole preparare
    per rappresentare
##      il comportamento del metodo (la curva rossa
   descritta nell'elaborato)
##    - f_name: nome della funzione che si vuole applicare
    ad ogni punto
##      contenuto nell'array ascisse. Se f_name
   rappresenta un nome di funzione
##      diverso da "invokeDelegate", ovvero rappresenta in
    modo esplicito una funzione,
##      allora passare il parametro "f_argument" = ""
##
##      Devo prestare attenzione se la funzione
```

```
##      da valutare e' "invokeDelegate", in quanto essendo per costruzione una
##      funzione, posso chiedere di usarla per prepare vettori per un grafico.
##      In questo caso, deve passare anche il nome della funzione che il delegato
##      dovra' invocare, introducendo il parametro f_argument.
##    - f_argument, rappresenta il nome della funzione che l'oggetto "invokeDelegate"
##      dovra' invocare, quindi sara' valorizzato solo nel caso si utilizzi
##      un delegato, quindi una rappresentazione implicita della funzione che si vuole
##      utilizzare.

function [ preparedXs, preparedYs ] = prepareForPlottingSecantMethodSegments (ascisse, f_name, f_argument)
preparedXs = [];
preparedYs = [];

for i = 1:length(ascisse)

  # region: add the point with coordinate (ascisse(i), f(ascisse(i)) --------
  # graphically i go to the evaluation point
  preparedXs = [preparedXs ascisse(i)];

  if strcmp(f_name, "invokeDelegate") == 1
    # se sto usando un delegato allora invoco il delgato, passando il nome
    # della funzione che il delegato (f_name == "invokeDelegate") deve invocare (f_argument)
    # con argomento ascisse(i) su cui "f_argument" verra' applicata.
    # Ho un doppio livello di indirezione: feval > invokeDelegate
    y = feval(f_name, f_argument, ascisse(i));
  else
    # f_name rappresenta in modo diretto una funzione, quindi posso utilizzare
```

```
    # la feval nel modo classico, invocando f_name con
       argomento ascisse(i)
    y = feval(f_name, ascisse(i));
  end
  preparedYs = [preparedYs y];
  # endregion
     ------------------------------------------------------------------


  # region: add the point with coordinate (ascisse(i), 0)
     -------------------
  # graphically I go down the the ascissa line
  preparedXs = [preparedXs ascisse(i)];
  preparedYs = [preparedYs 0];
  # endregion
     ------------------------------------------------------------------


  # region: add the point with coordinate (ascisse(i), f(
     ascisse(i)) ---------
  # graphically I come back up to the evaluation point.
  preparedXs = [preparedXs ascisse(i)];
  preparedYs = [preparedYs y];
  # endregion
     ------------------------------------------------------------------


  # region: add the point with coordinate (ascisse(i+2),
     0) -----------------
  # graphically I go to the +2 element down to the ascissa
      line to "draw" the secante line
  if (length(ascisse) >= i + 2)
    preparedXs = [preparedXs ascisse(i + 2)];
    preparedYs = [preparedYs 0];
  end
  # endregion
     ------------------------------------------------------------------

end
endfunction
```

6.2.4 *errorMonitor*

```
## errorMonitor
## Questo oggetto matematico costruisce un vettore di fattori di amplificazione
## k, relativamente al vettore di valori passati come ingresso.
##
## I fattori di amplificazione sono calcolati tra valori adiacenti, per questo
## motivo vale questa legge length(array) = length(error) + 1.

function [ errors ] = errorMonitor (array)
errors = [];
for i = 1:length(array) - 1
  error = (abs(array(i)) + abs(array(i+1)))/abs(array(i) - array(i+1));
  errors = [errors error];
end
endfunction
```

6.3 METODO DI BISEZIONE

```
## bisectionMethod
##
## Questo oggetto matematico implementa il metodo di bisezione.
##
## Input:
##    f: nome della funzione di cui si vuole approssimare lo zero
##    a: estremo inferiore dell'intervallo di confidenza
##    b: estremo superiore dell'intervallo di confidenza
##    tolx: tolleranza accettata sull'approssimazione trovata
##
## Output:
##    x: approssimazione che rispetta la tolleranza richiesta
```

```
##    i: passi effettivamente eseguiti per ricavare l'
   approssimazione
##    imax: numero massimo di passi necessari per ricavare
    l'approssimazione
##    ascisse: vettore di ascisse calcolate durante i
   passi del metodo

function [ x,i,imax,ascisse ] = bisectionMethod (f,a,b,
   tolx)
ascisse = [];
fa = invokeDelegate(f,a);
fb = invokeDelegate(f,b);

x=(a+b)/2;
ascisse = [ascisse x];

fx = invokeDelegate(f,x);
imax = ceil(log2(b-a)-log2(tolx));
for i = 2:imax
  f1x = ((fb-fa)/(b-a));
  if abs(fx)<=tolx*abs(f1x);
    break
  elseif fa*fx<0
    b=x;
    fb=fx;
  else
    a=x;
    fa=fx;
  end
  x=(a+b)/2;
  ascisse = [ascisse x];
  fx=invokeDelegate(f,x);
end
i = i - 1;
endfunction
```

6.4 METODO DI NEWTON

Questa implementazione utilizza il criterio di arresto esposto nell' *Osservazione 2.3* nel caso di radici semplici.

Una osservazione sul numero di passi effettuati: per la mia implementazione vale $i = \mathtt{length}(\mathtt{ascisse}) + 2$ in quanto nel ciclo *while* colleziono $\mathtt{length}(\mathtt{ascisse})$

valori, più uno per il valore di innesco iniziale, più uno per l'ultimo valore appena usciti dal ciclo *while*.

L'implementazione di questo oggetto matematico permette di specificare il criterio di arresto da utilizzare. Sono disponibili due criteri, quello per *incremento* e per *residuo*. Ho introdotto la possibilità di scegliere fra due criteri di arresto in quanto se si utilizza l'implementazione proposta (criterio del *residuo*) si va in contro al fenomeno della cancellazione numerica, in quanto $x_{i+1} \approx x_i$ per $i \to \infty$ (vedere esercizio 1.3.16). Per questo motivo implemento un secondo criterio di arresto, per *residuo*, basandomi sull'idea dell'equazione (2.14) e sull'*Osservazione 2.3*, ottengo:

$$\frac{\frac{f(x_i)}{f'(x_i)}}{rtol_X|x_{i+1}| + tol_X} \leqslant 1$$

Questo criterio risulta essere più robusto dal punto di vista implementativo:

- usando il rapporto $\frac{f}{f'}$ si ha un'operazione di macchina sempre ben condizionata ($k = 2$ per l'equazione del testo (1.25)), mentre usando la differenza introdurrei nella valutazione del criterio di arresto un errore con un alto fattore di amplificazione.
- la somma al denominatore risulta sempre ben condizionata, però è importante eseguire la somma con l'ordine degli addendi scritto nella precedente equazione. Questo permette di non perdere cifre significative quando il primo operando è molto piccolo.

```
## newtonMethod
##
## Questo oggetto matematico implementa il metodo di
   Newton.
##
## Input:
##    f: nome della funzione di cui si vuole approssimare
   lo zero
##    f1: derivata della f
##    x0: punto di innesco del metodo
##    itmax: numero massimo di iterazioni per fermare la
   computazione nel caso il metodo non converge.
##    tolx: tolleranza assoluta accettata sull'
   approssimazione trovata
##    rtolx: tolleranza relativa accettata sull'
   approssimazione trovata
##    stopCriterion: nome del criterio che si vuole
   utilizzare criterio di arresto.
```

```
##
## Output:
##    x: approssimazione che rispetta la tolleranza
   richiesta
##    i: passi effettivamente eseguiti per ricavare l'
   approssimazione
##    ascisse: vettore di ascisse calcolate durante i
   passi del metodo
##    checkedTolXs: vettore di tolleranze che sono state
   controllate nel criterio di arresto.

function [x, i, ascisse, checkedTolXs]=newtonMethod(f,f1,
   x0,itmax,tolx,rtolx,stopCriterion)
i=0;
ascisse = [];
ascisse = [ascisse x0];
checkedTolXs = []; # array delle tolleranze controllate

fx = invokeDelegate(f,x0); # applico la funzione

if fx==0
  x=x0;
  return
end

f1x = invokeDelegate(f1,x0); # applico la derivata
if f1x==0
  error('La derivata prima ha assunto valore zero
     impossibile continuare')
end

x= x0-(fx/f1x);
tolXToCheck = feval(stopCriterion, x, x0, fx, f1x);

while (i<itmax) & (tolXToCheck/(rtolx*abs(x) + tolx)>1)
  checkedTolXs = [checkedTolXs tolXToCheck];
  i = i+1;
  x0 = x;
  ascisse = [ascisse x0]; # colleziono la nuova ascissa

  fx = invokeDelegate(f,x0);
  f1x = invokeDelegate(f1,x0);
```

```
  %Se la derivata vale zero non possiamo continuare:
  % controllo che non si abbia raggiunto una soluzione
  % rispettando le tolleranze richieste.
  if f1x==0
    if fx == 0
      disp('f(x) = f1(x) = 0');
      return
    elseif (feval(stopCriterion, x, x0, fx, f1x)/(rtolx*
       abs(x) + tolx)<=1)
      disp('f1(x) = 0 & stopCriterion reached');
      return
    end
    disp('La derivata prima ha assunto valore zero,
       impossibile continuare')
  end
  x = x0-fx/f1x;
  tolXToCheck = feval(stopCriterion, x, x0, fx, f1x);
end

# altrimenti l'ultima computazione di x (riga 63 ed
   eventualmente riga 38 nel caso
# in cui non ho nessuna iterazione del ciclo while) non
   vengono collezionate.
# Qui devo collezionare x e non x0, perche' x0 non viene
   aggiornato nell'ultima iterazione
ascisse = [ascisse x];

tolXToCheck = feval(stopCriterion, ascisse(length(ascisse)
   ), ascisse(length(ascisse) - 1), fx, f1x);

# colleziono l'ultima tolX che non e' stata aggiunta nel
   ciclo while
checkedTolXs = [checkedTolXs tolXToCheck];

if (tolXToCheck/(rtolx*abs(x) + tolx)>1)
  disp('Il metodo non converge.')
end

endfunction

function result = incrementCriterion (x, x0, fx0, f1x0)
  result = abs(x-x0);
endfunction
```

```
87 function result = residueCriterion (x, x0, fx0, f1x0)
     result = fx0/f1x0;
   endfunction
```

6.5 VARIANTI DEL METODO DI NEWTON

6.5.1 *Molteplicità dello zero nota*

Questa implementazione utilizza il criterio di arresto esposto nell' *Osservazione 2.3* nel caso di radici semplici.

Utilizza il criterio di arresto per *incremento*.

```
1  ## newtonMethodMoltKnown
   ##
   ## Questo oggetto matematico implementa il metodo di
      Newton con molteplicita'
   ## dello zero nota.
   ##
6  ## Input:
   ##    f: nome della funzione di cui si vuole approssimare
      lo zero
   ##    f1: derivata della f
   ##    x0: punto di innesco del metodo
   ##    tolx: tolleranza assoluta accettata sull'
      approssimazione trovata
11 ##    rtolx: tolleranza relativa accettata sull'
      approssimazione trovata
   ##    m: molteplicita' dello zero
   ##
   ## Output:
   ##    x: approssimazione che rispetta la tolleranza
      richiesta
16 ##    i: passi effettivamente eseguiti per ricavare l'
      approssimazione
   ##    ascisse: vettore di ascisse calcolate durante i
      passi del metodo

   function [x, i, ascisse]=newtonMethodMoltKnown(f,f1,x0,
      itmax,tolx,rtolx, m)
   i=0;
21 ascisse = [];
```

```
ascisse = [ascisse x0];

fx = invokeDelegate(f,x0); # applico la funzione

if fx==0
  x=x0;
  return
end

f1x = invokeDelegate(f1,x0); # applico la derivata
if f1x==0
  error('La derivata prima ha assunto valore zero
     impossibile continuare')
end

x= x0-(m*fx/f1x);
while (i<itmax) & ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1)
  i = i+1;
  x0 = x;
  ascisse = [ascisse x0]; # colleziono la nuova ascissa

  fx = invokeDelegate(f,x0);
  f1x = invokeDelegate(f1,x0);

  %Se la derivata vale zero non possiamo continuare:
  % controllo che non si abbia raggiunto una soluzione
  % rispettando le tolleranze richieste.
  if f1x==0
    if fx == 0
      disp('f(x) = f1(x) = 0');
      return
    elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
      disp('f1(x) = 0 & stopCriterion reached');
      return
    end
    disp('La derivata prima ha assunto valore zero,
       impossibile continuare')
  end
  x = x0-(m*fx/f1x);
end

# altrimenti l'ultima computazione di x (riga 54 ed
   eventualmente riga 34 nel caso
```

```
# in cui non ho nessuna iterazione del ciclo while) non
   vengono collezionate.
# Qui devo collezionare x e non x0, perche' x0 non viene
   aggiornato nell'ultima iterazione
ascisse = [ascisse x];

if ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1)
  disp('Il metodo non converge.')
end

endfunction
```

6.5.2 *Molteplicità dello zero non nota - variante Aitken*

Questa implementazione utilizza il criterio di arresto esposto nell' *Osservazione 2.3* nel caso di radici semplici.

Utilizza il criterio di arresto per *incremento*.

```
## newtonMethodAitken
##
## Questo oggetto matematico implementa il metodo di
   Newton.
##
## Input:
##    f: nome della funzione di cui si vuole approssimare
   lo zero
##    f1: derivata della f
##    x0: punto di innesco del metodo
##    tolx: tolleranza assoluta accettata sull'
   approssimazione trovata
##    rtolx: tolleranza relativa accettata sull'
   approssimazione trovata
##
## Output:
##    x: approssimazione che rispetta la tolleranza
   richiesta
##    i: passi effettivamente eseguiti per ricavare l'
   approssimazione
##    ascisse: vettore di ascisse calcolate durante i
   passi del metodo
```

```
function [x, i, ascisse]=newtonMethodAitken(f,f1,x0,itmax,
    tolx,rtolx)
i=0;
ascisse = [];
ascisse = [ascisse x0];

fx = invokeDelegate(f,x0); # applico la funzione

if fx==0
  x=x0;
  return
end

f1x = invokeDelegate(f1,x0); # applico la derivata
if f1x==0
  error('La derivata prima ha assunto valore zero
     impossibile continuare')
end

x= x0-(fx/f1x);
go = 1;
while (i<itmax) & go
  i = i+1;
  x0 = x;
  ascisse = [ascisse x0]; # colleziono la nuova ascissa

  fx = invokeDelegate(f,x0);
  f1x = invokeDelegate(f1,x0);

  #Se la derivata vale zero non possiamo continuare:
  # controllo che non si abbia raggiunto una soluzione
  # rispettando le tolleranze richieste.
  if f1x==0
    if fx == 0
      disp('f(x) = f1(x) = 0');
      return
    elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
      disp('f1(x) = 0 & stopCriterion reached');
      return
    end
    disp('La derivata prima ha assunto valore zero,
       impossibile continuare')
  end
```

```
  x1 = x0-fx/f1x;
  fx = invokeDelegate(f,x1);
  f1x = invokeDelegate(f1,x1);
  if f1x==0
    if fx == 0
      return
    elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
      return
    end
    error('La derivata prima ha assunto valore zero,
       impossibile continuare!')
  end
  x = x1 - fx/f1x;
  # evito la cancellazione numero in quanto + e - sono mal
     condizionate
  t = ((x-2*x1)+x0);
  if t == 0
    if invokeDelegate(f,x) == 0
      return
    end
    error('Impossibile determinare la radice nella
       tolleranza desiderata')
  end
  x = (x*x0-x1^2)/t;
  go = ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1);
end

# altrimenti l'ultima computazione di x (riga 54 ed
   eventualmente riga 34 nel caso
# in cui non ho nessuna iterazione del ciclo while) non
   vengono collezionate.
# Qui devo collezionare x e non x0, perche' x0 non viene
   aggiornato nell'ultima iterazione
ascisse = [ascisse x];

if go
  disp('Il metodo non converge.')
end

endfunction
```

6.6 METODI QUASI-NEWTON

6.6.1 *Metodo delle corde*

Questa implementazione utilizza il criterio di arresto esposto nell' *Osservazione 2.3* nel caso ordine di convergenza $p = 1$, ovvero il criterio di arresto sarà dato da:

$$|x_{i+1} - x_i| \leqslant \frac{1-c}{c} tol_X$$

Ricavo in modo dinamico la costante c, usando le tre iterate più recenti:

$$c \approx \frac{|x_i - x_{i-1}|}{|x_{i-1} - x_{i-2}|}$$

```
## chordMethodLinearCriteria
##
## Questo oggetto matematico implementa il metodo delle corde.
## Questa implementazione utilizza il criterio di arresto lineare di
## pag. 36.
##
## Input:
##    f: nome della funzione di cui si vuole approssimare lo zero
##    f1: derivata della f
##    x0: punto di innesco del metodo
##    itmax: numero massimo di iterazioni per fermare la computazione nel caso il metodo non converge.
##    tolx: tolleranza assoluta accettata sull' approssimazione trovata
##    rtolx: tolleranza relativa accettata sull' approssimazione trovata
##
## Output:
##    x: approssimazione che rispetta la tolleranza richiesta
##    i: passi effettivamente eseguiti per ricavare l' approssimazione
##    ascisse: vettore di ascisse calcolate durante i passi del metodo

function [x, i, ascisse]=chordMethodLinearCriteria(f,f1,x0,itmax,tolx,rtolx)
```

```
ascisse = [];
ascisse = [ascisse x0]; # colleziono il punto di innesco

fx = invokeDelegate(f,x0); # applico la funzione
if fx==0
  x=x0;
  return
end
f1x = invokeDelegate(f1,x0); # applico la derivata
if f1x==0
  error('La derivata prima ha assunto valore zero nel
     punto di innesco, impossibile continuare')
end

x1= x0-(fx/f1x);
ascisse = [ascisse x1]; # colleziono la prima iterata
fx1 = invokeDelegate(f,x1); # applico la funzione
if fx1==0
  x=x1;
  return
end

x= x1-(fx1/f1x); # non colleziono qui in ascisse perche'
   viene fatto subito dopo il while se non ci sono
   iterazioni.

c = abs(x - x1) / abs(x1 - x0);

# while initialization
i=2; # ho gia calcolato due passi x1 ed x
# non importa resettare le due variabili x0 ed x1 in
   quanto vengono settate (solo x0)
# all'interno del ciclo
while (i<itmax) & ((abs(x-x0) / abs(tolx*((1-c)/c)))>1)
  i = i+1;
  x0 = x;
  ascisse = [ascisse x0]; # colleziono la nuova ascissa

  fx = invokeDelegate(f,x0);

  % qui non ha senso controllare se la derivata si annulla
      in quanto questo
```

```
  % controllo viene fatto alla riga 30.

  x = x0-(fx/f1x);

  # posso considerare ascisse(length(ascisse)) = x(i-1)
     perche' ancora non ho collezionato x
  c = abs(x - ascisse(length(ascisse))) / abs(ascisse(
     length(ascisse)) - ascisse(length(ascisse) - 1));
end

# altrimenti l'ultima computazione di x (riga 54 ed
   eventualmente riga 34 nel caso
# in cui non ho nessuna iterazione del ciclo while) non
   vengono collezionate.
# Qui devo collezionare x e non x0, perche' x0 non viene
   aggiornato nell'ultima iterazione
ascisse = [ascisse x];
c = abs(ascisse(length(ascisse)) - ascisse(length(ascisse)
    - 1)) / abs(ascisse(length(ascisse) - 1) - ascisse(
   length(ascisse) - 2));

if ((abs(x-x0) / abs(tolx*((1-c)/c)))>1)
  disp('Il metodo non converge.')
end

endfunction
```

Observation 6.6.1. *Questa implementazione non è molto robusta in quanto mancano molti controlli per garantire il significato di tutte le operazioni di divisione. Dovrebbero essere aggiunti. Se si prova ad eseguire questa applicazione:*

```
[x, i, ascisse] =
chordMethodLinearCriteria('functionNewtonRecursion','
   functionNewtonRecursionDerivative',1-1e-10,1e5, 1e-15,
   1e-15)
```

*si ottiene una serie infinita di messaggi "*Warning: division by zero*".*

Inoltre sarebbe interessante confrontare il criterio di arresto usato nell'implementazione con il criterio di arresto usato invece per l'implementazione del metodo Metodo di Newton.

6.6.2 *Metodo delle secanti*

Questa implementazione utilizza il criterio di arresto esposto nell' *Osservazione 2.3* nel caso di radici semplici.

Utilizza il criterio di arresto per *incremento*.

```
## secantMethod
##
## Questo oggetto matematico implementa il metodo delle secanti.
##
## Input:
##    f: nome della funzione di cui si vuole approssimare lo zero
##    f1: derivata della f
##    x0: punto di innesco del metodo
##    itmax: numero massimo di iterazioni per fermare la computazione nel caso il metodo non converge.
##    tolx: tolleranza assoluta accettata sull' approssimazione trovata
##    rtolx: tolleranza relativa accettata sull' approssimazione trovata
##
## Output:
##    x: approssimazione che rispetta la tolleranza richiesta
##    i: passi effettivamente eseguiti per ricavare l' approssimazione
##    ascisse: vettore di ascisse calcolate durante i passi del metodo
##    checkedTolXs: vettore di tolleranze che sono state controllate nel criterio di arresto.

function [x, i, ascisse, checkedTolXs]=secantMethod(f,f1,x0,itmax,tolx,rtolx)
# fisso il criterio di arresto in quanto questo metodo non calcola la derivata
# ad ogni passo come invece viene fatto nel metodo di Newton.
stopCriterion = "incrementCriterion";
i=0;
ascisse = [];
ascisse = [ascisse x0];
```

```
checkedTolXs = []; # array delle tolleranze controllate

fx = invokeDelegate(f,x0); # applico la funzione

if fx==0
  x=x0;
  return
end

f1x = invokeDelegate(f1,x0); # applico la derivata
if f1x==0
  error('La derivata prima ha assunto valore zero
     impossibile continuare')
end

x= x0-(fx/f1x);
tolXToCheck = feval(stopCriterion, x, x0, fx, f1x);

while (i<itmax) & (tolXToCheck/(rtolx*abs(x) + tolx)>1)
  checkedTolXs = [checkedTolXs tolXToCheck];
  i = i+1;
  fx0 = fx;
  ascisse = [ascisse x]; # colleziono la nuova ascissa

  fx = invokeDelegate(f,x);
  t = fx - fx0;

  %Se t vale zero non possiamo continuare in quanto la
     divisione non sarebbe definita:
  % controllo che non si abbia raggiunto una soluzione
  % rispettando le tolleranze richieste.
  if t==0
    if fx == 0
      disp('f(x) = t = o');
      return
    elseif (feval(stopCriterion, x, x0, fx, f1x)/(rtolx*
       abs(x) + tolx)<=1)
      disp('f1(x) = o & stopCriterion reached');
      return
    end
    disp('La derivata prima ha assunto valore zero,
       impossibile continuare')
  end
```

```
  x1 = ((fx* x0) - (fx0 * x))/t;
  x0 = x;
  x = x1;
  tolXToCheck = feval(stopCriterion, x, x0, fx, f1x);
end

# altrimenti l'ultima computazione di x (riga 63 ed
   eventualmente riga 38 nel caso
# in cui non ho nessuna iterazione del ciclo while) non
   vengono collezionate.
# Qui devo collezionare x e non x0, perche' x0 non viene
   aggiornato nell'ultima iterazione
ascisse = [ascisse x];

tolXToCheck = feval(stopCriterion, ascisse(length(ascisse)
   ), ascisse(length(ascisse) - 1), fx, f1x);

# colleziono l'ultima tolX che non e' stata aggiunta nel
   ciclo while
checkedTolXs = [checkedTolXs tolXToCheck];

if (tolXToCheck/(rtolx*abs(x) + tolx)>1)
  disp('Il metodo non converge.')
end

endfunction

function result = incrementCriterion (x, x0, fx0, f1x0)
  result = abs(x-x0);
endfunction

function result = residueCriterion (x, x0, fx0, f1x0)
  result = fx0/f1x0;
endfunction
```

6.7 SCRIPT

6.7.1 *Script eser 2.5*

```
# script per lanciare l'esecuzione dell'esercizio 2.5

```

```
disp('application of the Newton method')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
     cursor))))
  [x, i] = newtonMethod('function25first','
     function25firstDerivative', 10, 1e5, 10^(-cursor),
     10^(-cursor), 'incrementCriterion')
end
disp('')
disp('application of the second function')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
     cursor))))
  [x, i] = newtonMethod('function25second','
     function25secondDerivative', 10, 1e5, 10^(-cursor),
     10^(-cursor), 'incrementCriterion')
end
disp('
  ')

disp('application of the Newton method with multiplicity m
  ')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
     cursor))))
  [x, i] = newtonMethodMoltKnown('function25first','
     function25firstDerivative', 10, 1e5, 10^(-cursor),
     10^(-cursor), 10)
end
disp('')
disp('application of the second function')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
     cursor))))
  [x, i] = newtonMethodMoltKnown('function25second','
     function25secondDerivative', 10, 1e5, 10^(-cursor),
     10^(-cursor), 10)
end
disp('
  ')
```

```
disp('application of the Newton method Aitken version')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
      cursor))))
  [x, i] = newtonMethodAitken('function25first','
      function25firstDerivative', 10, 1e5, 10^(-cursor),
      10^(-cursor))
end
disp('')
disp('application of the second function')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
      cursor))))
  [x, i] = newtonMethodAitken('function25second','
      function25secondDerivative', 10, 1e5, 10^(-cursor),
      10^(-cursor))
end
disp('
    ------------------------------------------------------------------------
    ')
```

6.7.2 *Script eser 2.5 - Newton Stop Criteria Comparison*

```
## scriptExercise25NewtonComparison
# script per lanciare l'esecuzione della comparazione del
    solo metodo di Newton usando
# i due diversi criteri, relativamente all'esercizio 2.5

disp('application of the Newton method with increment stop
    criteria')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
      cursor))))
  [x, i] = newtonMethod('function25first','
      function25firstDerivative', 10, 1e5, 10^(-cursor),
      10^(-cursor), 'incrementCriterion')
end
disp('')
disp('application of the second function')
for cursor = 2:2:16
```

```
  disp(strcat('application with tolx = ', num2str(10^(-
     cursor))))
  [x, i] = newtonMethod('function25second','
     function25secondDerivative', 10, 1e5, 10^(-cursor),
     10^(-cursor), 'incrementCriterion')
end
disp('
  ----------------------------------------------------------------------
  ')

disp('application of the Newton method with residue stop
   criteria')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
     cursor))))
  [x, i] = newtonMethod('function25first','
     function25firstDerivative', 10, 1e5, 10^(-cursor),
     10^(-cursor), 'residueCriterion')
end
disp('')
disp('application of the second function')
for cursor = 2:2:16
  disp(strcat('application with tolx = ', num2str(10^(-
     cursor))))
  [x, i] = newtonMethod('function25second','
     function25secondDerivative', 10, 1e5, 10^(-cursor),
     10^(-cursor), 'residueCriterion')
end
disp('
  ----------------------------------------------------------------------
  ')
```

6.7.3 *Script eser 2.7*

```
# script per lanciare l'esecuzione dell'esercizio 2.7

disp('bisection method')
for cursor = 2:2:16
  disp(strcat('tol_x = ', num2str(10^(-cursor))));
  [x, i] = bisectionMethod('exercise27Function', 0, 1,
     10^(-cursor))
```

```
end

disp('-------------------------');

disp('Newton method')
for cursor = 2:2:16
  disp(strcat('tol_x = ', num2str(10^(-cursor))));
  [x, i] = newtonMethod('exercise27Function','
     exercise27FunctionDerivative', 0, 1e5, 10^(-cursor),
     10^(-cursor), 'incrementCriterion')
end

disp('-------------------------');

disp('chord method')
for cursor = 2:2:16
  disp(strcat('tol_x = ', num2str(10^(-cursor))));
  [x, i] = chordMethodLinearCriteria('exercise27Function',
     'exercise27FunctionDerivative', 0, 1e5, 10^(-cursor),
      10^(-cursor), 'residueCriterion')
end

disp('-------------------------');

disp('secant method')
for cursor = 2:2:16
  disp(strcat('tol_x = ', num2str(10^(-cursor))));
  [x, i] = secantMethod('exercise27Function','
     exercise27FunctionDerivative', 0, 1e5, 10^(-cursor),
     10^(-cursor))
end

disp('-------------------------');
```

6.8 FATTORIZZAZIONI

6.8.1 *triangularSystemSolver*

```
## triangularSystemSolver

```

```
function [ ret ] = triangularSystemSolver (A, b,
   typeOfMatrix)
ret = feval(typeOfMatrix, A, b);
end
endfunction

function [ unknowns ] = lowerSolveEngine(A, b)
  unknowns = b;
  n = length(unknowns);
  for j = 1:n
    unknowns(j) = unknowns(j) / A(j,j);
    for i = j+1:n
      unknowns(i) = unknowns(i) - A(i,j) * unknowns(j);
    end
  end
endfunction

function [ unknowns ] = upperSolveEngine(A, b)
  unknowns = b;
  n = length(unknowns);
  for j = n:-1:1
    unknowns(j) = unknowns(j) / A(j,j);
    for i = 1:j-1
      unknowns(i) = unknowns(i) - A(i,j) * unknowns(j);
    end
  end
endfunction
```

6.8.2 *normalizationEngine*

```
## normalizationEngine

function [ ret ] = normalizationEngine (narmalizationName,
    B, setUnaryDiagonal)
ret = feval(narmalizationName, B, setUnaryDiagonal);
endfunction


function [ normalized ] = normalizeLowerTriangular (B,
   setUnaryDiagonal)
  n = columns(B);
```

```
  normalized = B(1:n, 1:n);
  for i = 1:n
    for j = 1:n
      if i < j
        normalized(i, j) = 0;
      end
    end
  end

  if setUnaryDiagonal == 1
    for i = 1:n
      normalized(i,i) = 1;
    end
  end
endfunction

function [ normalized ] = normalizeUpperTriangular (B,
   setUnaryDiagonal)
  n = columns(B);
  normalized = B(1:n, 1:n);
  for i = 1:n
    for j = 1:n
      if i > j
        normalized(i, j) = 0;
      end
    end
  end

  if setUnaryDiagonal == 1
    for i = 1:n
      normalized(i,i) = 1;
    end
  end
endfunction
```

6.8.3 *LUmethod*

```
## LUmethod

function [L, U, unknowns] = LUmethod (A, b)
[L, U] = factorLU(A);
```

```
unknowns = triangularSystemSolver(U,
    triangularSystemSolver(L, b, "lowerSolveEngine"), "
    upperSolveEngine");
endfunction

function [L, U] = factorLU (A)
n = length(A);
for i = 1:n-1
  if A(i,i) == 0
    error('A non fattorizzabile LU');
  end

  A(i+1:n,i) = A(i+1:n,i)/A(i,i);
  A(i+1:n,i+1:n) = A(i+1:n, i+1:n) - A(i+1:n,i) * A(i, i
      +1:n);
end
L = normalizationEngine("normalizeLowerTriangular", A, 1);
U = normalizationEngine("normalizeUpperTriangular", A, 0);
endfunction
```

6.8.4 *LDLmethod*

Observation 6.8.1 (Struttura matriciale degli aggiornamenti del metodo al j-esimo passo di iterazione)**.**

$$\begin{bmatrix} \Xi & & & \\ & \ddots & & \\ & & \Xi & \\ \Theta & \cdots & \Theta & a^*_{jj} \\ \gamma & \cdots & \gamma & \Delta^* \\ \vdots & \vdots & & \vdots \\ \gamma & \cdots & \gamma & \Delta^* \end{bmatrix}$$

*Con Δ ho indicato elementi appartenti al vettore caratteristico, con Γ gli elementi della matrice che vengono letti per calcolare $l_{ij}, i \geqslant j$; con Θ gli elementi che uso per costruire il vettore di elementi $l_{jk}, k \in \{1, \ldots, j-1\}$; con Ξ gli elementi che leggo per costruire il vettore $d_k, k \in \{1, \ldots, j-1\}$ (il prodotto componente a componente $\boldsymbol{\Xi}.*\boldsymbol{\Theta} = \boldsymbol{\nu}$ (usando notazione Octave), $\boldsymbol{\nu}$ è il vettore che viene costruito nel codice nell'errata corrige; con * indico gli elementi che vengono aggiornati ad ogni passo del metodo.*

Risolvo in quest'ordine:

1. $\boldsymbol{\nu} \leftarrow \boldsymbol{\Xi}_{\cdot} * \boldsymbol{\Theta}$
2. $a_{jj} \leftarrow a_{jj} - \boldsymbol{\Theta}^{\mathsf{T}} * \boldsymbol{\nu}$
3. *nella prossima equazione* $\gamma \in \Gamma \in \mathbb{R}^{k \times k}$*:*

$$\boldsymbol{\Delta} \leftarrow \frac{\boldsymbol{\Delta} - \Gamma * \boldsymbol{\nu}}{a^{*}_{jj}}$$

Tutti i vettori sono da intendersi vettori colonna e la relazione $\leftarrow$*,* $(a, b) \in \leftarrow$*, significa assegno* b *ad* a*.*

```
## LDLmethod

function [L, D, unknowns] = LDLmethod (A, b)
[L, D] = factorLDL(A);
unknowns = triangularSystemSolver(L',
    triangularSystemSolver(L, triangularSystemSolver(D, b,
    "lowerSolveEngine"), "lowerSolveEngine"), "
    upperSolveEngine");
endfunction

function [ L, D ] = factorLDL (A)
n = length(A);

if A(1,1) <= 0
  error("A non sdp.");
end

A(2:n,1) = A(2:n,1) / A(1,1);

for j = 2:n
  v = ( A(j,1:j-1)' ) .* diag(A(1:j-1,1:j-1));
  A(j,j) = A(j,j) - A(j,1:j-1)*v;
  if A(j,j) <= 0
    error("A non sdp.");
  end
  A(j+1:n,j) = ( A(j+1:n,j) - A(j+1:n, 1:j-1)*v ) / A(j,j)
      ;
end
L = normalizationEngine("normalizeLowerTriangular", A, 1);

D = normalizationEngine("normalizeLowerTriangular", A, 0);
D = normalizationEngine("normalizeUpperTriangular", D, 0);
endfunction
```

6.8.5 *PALUmethod*

Observation 6.8.2 (Struttura matriciale degli aggiornamenti del metodo all'i-esimo passo di iterazione)**.**

$$\begin{bmatrix} & & & \\ & & & \\ & & & \\ a^*_{ii} & \Theta & \cdots & \Theta \\ \Delta^* & \Gamma^* & \cdots & \Gamma^* \\ \vdots & \vdots & & \vdots \\ \Delta^* & \Gamma^* & \cdots & \Gamma^* \end{bmatrix}$$

*Con Δ ho indicato elementi appartenti al vettore caratteristico di Gauss, con Γ elementi della matrice che vengono modificati dal passo di aggiornamento del metodo, con Θ gli elementi che uso per costruire la matrice da sottrarre a $I\boldsymbol{\nu}_i$ ($\boldsymbol{\nu}_i$ il vettore colonna che voglio annullare secondo il metodo) e con * indico gli elementi che vengono aggiornati ad ogni passo del metodo.*

Risolvo in quest'ordine:

1. *trovo l'elemento pivot e eventualmente scambio le righe, questo produce un eventuale aggiornamento di* a_{ii}
2. *normalizzo il vettore di Gauss:*

$$\boldsymbol{\Delta} \leftarrow \frac{\boldsymbol{\Delta}}{a^*_{ii}}$$

3. $a_{jj} \leftarrow a_{jj} - \boldsymbol{\Theta}^T * \boldsymbol{\nu}$
4. *nella prossima equazione* $\gamma \in \Gamma \in \mathbb{R}^{k\times k}$*:*

$$\Gamma \leftarrow \Gamma - \boldsymbol{\Delta}^* * \boldsymbol{\Theta}^T$$

Tutti i vettori sono da intendersi vettori colonna e la relazione $\leftarrow$*,* $(a, b) \in \leftarrow$*, significa assegno* b *ad* a*.*

```
## PALUmethod

function [L, U, p, unknowns] = PALUmethod (A, b)
[L, U, p] = factorPALU(A);

n = length(b);
clone = b;
```

```
for i = 1:n
  b(i) = clone(p(i));
end

unknowns = triangularSystemSolver(U,
   triangularSystemSolver(L, b, "lowerSolveEngine"), "
   upperSolveEngine");
endfunction

function [L, U, p] = factorPALU (A)
n = length(A);
p = [1:n];
for i = 1:n-1
  [mi, ki] = max(abs(A(i:n, i)));

  if mi == 0
    error("A singolare.");
  end

  ki = ki + i -1;
  if ki > i
    A([i ki], :) = A([ki i], :);
    p([i ki]) = p([ki i]);
  end

  A(i+1:n,i) = A(i+1:n,i)/A(i,i);
  A(i+1:n,i+1:n) = A(i+1:n, i+1:n) - A(i+1:n,i) * A(i, i
     +1:n);
end
L = normalizationEngine("normalizeLowerTriangular", A, 1);
U = normalizationEngine("normalizeUpperTriangular", A, 0);
endfunction
```

6.8.6 *QRmethod*

Observation 6.8.3 (Sull'uso e costruzione delle matrici di eliminazione H)**.** *Durante il metodo non vengono create e sviluppate in modo esplicito le matrici di eliminazione* H^i *per effettuare i prodotti* $H^i\boldsymbol{v}$*: vengono invece costruite in modo implicito, ovvero la computazione che viene effettuata dal metodo è direttamente la trasformazione ortogonale*

$$H\boldsymbol{v} \to \boldsymbol{v} - \frac{2}{z^T z} z(z^T \boldsymbol{v})$$

Observation 6.8.4 (Sul costo del prodotto scalare)**.** *Dati due vettori* $a, b \in \mathbb{R}^k$*, il prodotto scalare costa* $2k-1$ *operazioni, di cui* k *per i prodotti* $\forall i \in \{1, \ldots, k\}[a_i b_i]$*, e* $k-1$ *somme dei* k *prodotti.*

Observation 6.8.5 (Idee alla base del metodo)**.** *Queste sono le idee alla base del metodo:*

- $H(\beta z) = H(z), \forall \beta \in \mathbb{R}$*, ovvero la matrice di eliminazione non varia per multipli del vettore caratteristico* z *di Householder (permette quindi di scegliere* β *in modo da avere il vettore caratteristico con una struttura particolare,* $z_1 = 1$*).*
- *la seguente ugualianza permette di evitare il calcolo del prodotto scalare, risparmiando* $2m$ *operazioni:*

$$-\frac{z_1}{\alpha} = \frac{2}{z^T z}$$

  vedi esercizio 3.3.

- $v_1 = A(i,i) - \alpha$ *con* $\alpha A(i,i) < 0$*, permette di evitare il fenomeno della cancellazione numerica.*

Observation 6.8.6 (Sulla struttura delle matrici di eliminazione H_k)**.** *Ragionando a blocchi sulla struttura delle matrici di eliminazione vale:*

$$H_{i+1} = \left[\begin{array}{c|c} I_i & 0^T \\ \hline 0 & H^{(i+1)} \end{array}\right]$$

con $H^{(i+1)}$ *elementare di Householder relativa alla colonna* $i+1$ *di* $A^{(i)}$*:*

$$A^{(i)} e_{i+1} = \begin{bmatrix} a^{(i)}_{i+1,i+1} \\ \vdots \\ a^{(i)}_{m,i+1} \end{bmatrix}$$

Il precedente vettore è il vettore che si vuole rendere uguale a αe_{i+1}*.*

Al passo i-esimo ottengo:

$$A^{(i)} = H_i A^{(i-1)} = H_i \cdots H_1 A = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1i}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2i}^{(2)} & \cdots & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots & & & \vdots \\ & & & a_{i-1,i-1}^{(i-1)} & a_{i-1,i}^{(i-1)} & \cdots & a_{i-1,n}^{(i-1)} \\ & & & & & H^{(i)} A_{ii}^{(i-1)} & \end{bmatrix} =$$

$$= \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1i}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2i}^{(2)} & \cdots & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots & & & \vdots \\ & & & a_{ii}^{(i)} & a_{i,i+1}^{(i)} & \cdots & a_{in}^{(i)} \\ & & & & a_{i+1,i+1}^{(i)} & \cdots & a_{i+1,n}^{(i)} \\ & & & & \vdots & \ddots & \vdots \\ & & & & a_{m,i+1}^{(i)} & \cdots & a_{m,n}^{(i)} \end{bmatrix}$$

Attenzione: al primo passo $a_{11}^{(1)}$ viene aggiornato differentemente per quanto avviene per la fattorizzazione LU senza pivoting.

Observation 6.8.7 (Forma matriciale del passo di aggiornamento)**.**

$$\begin{bmatrix} a_{i+1,j}^{(i+1)} \\ \vdots \\ a_{m,j}^{(i+1)} \end{bmatrix} = \begin{bmatrix} a_{i+1,j}^{(i)} \\ \vdots \\ a_{m,j}^{(i)} \end{bmatrix} - \frac{2}{(z^{(i+1)})^T z^{(i+1)}} z^{(i+1)} \left((z^{(i+1)})^T \begin{bmatrix} a_{i+1,j}^{(i+1)} \\ \vdots \\ a_{m,j}^{(i+1)} \end{bmatrix} \right)$$

La precedente uguaglianza rappresenta in forma matriciale l'istruzione Octave *per l'aggiornamento della matrice (e implicitamente la costruzione della matrice triangolare superiore* R *(no* $\hat{R}$*)).*

Observation 6.8.8 (Struttura matriciale degli aggiornamenti del metodo all'i-esimo passo di iterazione)**.**

$$\begin{bmatrix} & & & \\ & & & \\ & & & \\ & a_{ii}^{*} & \Gamma^{*} & \cdots & \Gamma^{*} \\ & \Delta^{*} & \Gamma^{*} & \cdots & \Gamma^{*} \\ & \vdots & \vdots & & \vdots \\ & \Delta^{*} & \Gamma^{*} & \cdots & \Gamma^{*} \end{bmatrix}$$

*Con Δ ho indicato elementi appartenti al vettore caratteristico di Householder, con Γ elementi della matrice che vengono modificati dal passo di aggiornamento del metodo e con * indico gli elementi che vengono aggiornati ad ogni passo del metodo.*

Risolvo in quest'ordine:

1. *calcolo la norma α ed eventualmente cambio segno per evitare il fenomeno della cancellazione numerica nel calcolo di v_1, in modo da avere $A(i,i)\alpha < 0$*
2. $v_1 \leftarrow A(i,i) - \alpha$
3. $A(i,i) \leftarrow \alpha$
4. *normalizzo il vettore di Householder:*

$$\boldsymbol{\Delta} \leftarrow \frac{\boldsymbol{\Delta}}{v_1}$$

5. $\beta \leftarrow -\frac{v_1}{\alpha}$
6. *nella prossima equazione $\gamma \in \Gamma \in \mathbb{R}^{k\times k}$:*

$$\Gamma \leftarrow \Gamma - \beta \begin{bmatrix} 1 \\ \boldsymbol{\Delta}^* \end{bmatrix} \begin{bmatrix} 1 & (\boldsymbol{\Delta}^*)^T \end{bmatrix} \Gamma$$

Tutti i vettori sono da intendersi vettori colonna e la relazione $\leftarrow$, $(a,b) \in \leftarrow$, significa assegno b ad a.

```
## QRmethod

function [ hhvectors, Rhat, R, Q, g1, g2, unknowns,
    residue ] = QRmethod (A, b)
[ hhvectors, Rhat] = factorQR(A);
Q = buildQOrthogonalMatrix(hhvectors);
g = (Q') * b;

R = Rhat(1:columns(Rhat), 1:columns(Rhat));
g1 = g(1:columns(Rhat));
g2 = g(columns(Rhat)+1:rows(g));
unknowns = triangularSystemSolver(R, g1, "upperSolveEngine
    ");
residue = Q*Rhat*unknowns - b;
endfunction

function [houseHolderVectors, R] = factorQR (A)
```

```
n = columns(A);
m = rows(A);

for i = 1:n

  # alpha >= 0 per definizione di norma.
  alpha = norm(A(i:m, i));

  if alpha == 0
    error("rank(A) ~= n. A non ha rango massimo");
  end

  # A(i,i) e' il primo elemento della colonna i-esima.
  if A(i,i) > 0
    # inverto il segno di alpha in quanto alpha e'
       positivo per definizione
    # di norma, quindi per evitare il fenomeno della
       cancellazione numerica,
    # cambio di segno in quanto nella prossima istruzione
       cambio di nuovo
    # segno.
    alpha = -alpha;
  end

  # considero il primo elemento del vettore che voglio
     manipolare con la
  # trasformazione ortogonale.
  v1 = A(i,i) - alpha;

  # setto il primo elemento in base a quanto richiesto nel
     metodo di Householder
  # ovvero H * v = alpha * e_{1}
  A(i,i) = alpha;

  # costruisco il vettore caratteristico di Householder,
     normalizzando
  # solo le componenti con indice j > i, in quanto il
     primo elemento (settato
  # nell'istruzione precedente) non deve essere modificato
     in quanto sara'
  # un elemento (sulla diagonale) della matrice R.
  A(i+1:m, i) = A(i+1:m, i) / v1;
```

```
  # evito di calcolare il prodotto scalare, vedi testo per
      l'uso di questo trick.
  beta = -(v1/alpha);

  # A(i:m, i+1:n) = ... dato che ho gia modificato la
     colonna i, adesso aggiorno
  #   la restante parte della matrice, ovvero le colonne
     con indice j > i.
  # = A(i:m, i+1:n) parto dalle informazioni presenti
     nella matrice memorizzate
  # fino al passo (i-1)
  A(i:m, i+1:n) = A(i:m, i+1:n) - (beta * [1; A(i+1:m, i)
     ]) * ([1, A(i+1:m, i)'] * A(i:m, i+1:n));
    # moltiplico il coefficiente beta per il vettore di
       Householder, il quale
    # lo costruisco con [1; A(i+1:m, i)], perche essendo
       normalizzato, la sua
    # prima componente e' per costruzione 1 (oss: se
       considerassi A(i:m, i)
    # come vettore caratteristico commetterei un errore
       perche A(i,i) = alpha
    # per l'assegnazione fatta precedentemente.

    # finisco di effettuare il prodotto tra il vettore
       caratteristico trasposto
    # e il vettore che voglio manipolare (azzerando tutte
       le componenti tranne la
    # prima). Questo vettore e' A(i:m, i+1:n), infatti si
       usa come primo indice
    # A(i:...) invece di A(i+1:...) come invece si usa per
        il vettore caratteristico.

end
B = A;
houseHolderVectors = A;

# costruisco i vettori caratteristici partendo dalla
   matrice modificata A. I vettori
# caratteristici sono 0 fino alla (i-1)-esima componente,
   1 per la i-esima e
# valori computati dal precedente ciclo dalla componente (
   i+1) in poi.
for i = 1:n
```

```
  if i > 1
    houseHolderVectors(1:i-1, i) = 0;
  end

  houseHolderVectors(i, i) = 1;
end

# uso la matrice A modificata per estrapolare le informazioni che costruiscono
# la matrice triangolare superiore R.
R = normalizationEngine("normalizeUpperTriangular", A, 0);

# finisco di costruire la matrice R aggiungendo una matrice nulla in coda
R = [R; zeros(m-n,n)];
endfunction

function [ Q ] = buildQOrthogonalMatrix (houseHolderVectors)
m = rows(houseHolderVectors);
n = columns(houseHolderVectors);
Q = eye(m, m);
# costruisco la matrice ortogonale Q = (H_n * ... * H_1)^{T} =H_1 * ... * H_n
# perche H_j e' simmetrica
# per ogni vettore caratteristico ricavo la relativa matrice di eliminazione e moltiplico
# a sinistra.
for i = 1:n
  Q = Q * buildHElementaryMatrix(houseHolderVectors(1:m, i)); # it is important to multiply Q at left side
end
endfunction

# questo oggetto matematico computa la i-esima matrice di eliminazione in funzione
# del vettore caratteristico passato come input. La costruzione segue la definizione
# teorica enumciata nel testo.
function [ elementaryMatrix ] = buildHElementaryMatrix (houseHolderVector)
elementaryMatrix = eye(length(houseHolderVector)) - 2 * ((houseHolderVector * (houseHolderVector'))/((
```

```
    houseHolderVector') * houseHolderVector));
endfunction
```

6.8.7 *functionExercise332*

```
# functionExercise332.m

function [ unknownsMatrix, givenUnknownsMatrix ] =
    functionExercise332 (printPlots)
gamma = [10 1 .5 .1 .05 .01 .005 .001];
x = linspace(0, 10, 101)';
A = [x x.^0 ];
unknownsMatrix = [];
residueMatrix = [];

givenUnknownsMatrix = [];
givenResidueMatrix = [];

for i = 1:length(gamma)
  y = evaluatingFunction(x, gamma(i));
  [houseHolderVectors, Rhat, R, Q, g1, g2, unknowns,
      residue] = QRmethod(A,y);
  unknownsMatrix = [unknownsMatrix unknowns];
  residueMatrix = [residueMatrix residue];

  # confronto con codice dato
  a = A\y;
  givenUnknownsMatrix = [givenUnknownsMatrix a];
  r = A * a - y;
  givenResidueMatrix = [givenResidueMatrix r];
end

if printPlots == 1

  residueRows = rows(residueMatrix);
  unknownsRows = rows(unknownsMatrix);

  gammaIndex = 1;
  plot(x, evaluatingFunction(x, gamma(gammaIndex)), "c", x
      , residueMatrix(1:residueRows, gammaIndex), x,
      givenResidueMatrix(1:residueRows, gammaIndex), "r",
```

```
    unknownsMatrix(1:unknownsRows,gammaIndex),
    evaluatingFunction(unknownsMatrix(1:unknownsRows,
    gammaIndex), gamma(gammaIndex)), "b+",
    givenUnknownsMatrix(1:unknownsRows,gammaIndex),
    evaluatingFunction(givenUnknownsMatrix(1:unknownsRows
    ,gammaIndex), gamma(gammaIndex)), "g+");
axis([-1, 11, -20, 120]);
print 'exer332gamma10.tex' '-dTex' '-S800, 600';

gammaIndex = 3;
plot(x, evaluatingFunction(x, gamma(gammaIndex)), "c", x
    , residueMatrix(1:residueRows, gammaIndex), x,
    givenResidueMatrix(1:residueRows, gammaIndex), "r",
    unknownsMatrix(1:unknownsRows,gammaIndex),
    evaluatingFunction(unknownsMatrix(1:unknownsRows,
    gammaIndex), gamma(gammaIndex)), "b+",
    givenUnknownsMatrix(1:unknownsRows,gammaIndex),
    evaluatingFunction(givenUnknownsMatrix(1:unknownsRows
    ,gammaIndex), gamma(gammaIndex)), "g+");
axis([-1, 11, -20, 120]);
print 'exer332gamma5e-1.tex' '-dTex' '-S800, 600';

gammaIndex = 4;
plot(x, evaluatingFunction(x, gamma(gammaIndex)), "c", x
    , residueMatrix(1:residueRows, gammaIndex), x,
    givenResidueMatrix(1:residueRows, gammaIndex), "r",
    unknownsMatrix(1:unknownsRows,gammaIndex),
    evaluatingFunction(unknownsMatrix(1:unknownsRows,
    gammaIndex), gamma(gammaIndex)), "b+",
    givenUnknownsMatrix(1:unknownsRows,gammaIndex),
    evaluatingFunction(givenUnknownsMatrix(1:unknownsRows
    ,gammaIndex), gamma(gammaIndex)), "g+");
axis([-1, 11, -20, 120]);
print 'exer332gammae-1.tex' '-dTex' '-S800, 600';

gammaIndex = 8;
plot(x, evaluatingFunction(x, gamma(gammaIndex)), "c", x
    , residueMatrix(1:residueRows, gammaIndex), x,
    givenResidueMatrix(1:residueRows, gammaIndex), "r",
    unknownsMatrix(1:unknownsRows,gammaIndex),
    evaluatingFunction(unknownsMatrix(1:unknownsRows,
    gammaIndex), gamma(gammaIndex)), "b+",
    givenUnknownsMatrix(1:unknownsRows,gammaIndex),
```

```
      evaluatingFunction(givenUnknownsMatrix(1:unknownsRows
      ,gammaIndex), gamma(gammaIndex)), "g+");
  axis([-1, 11, -20, 120]);
  print 'exer332gammae-3.tex' '-dTex' '-S800, 600';
end
endfunction

function [ ret ] = evaluatingFunction (x, gamma)
ret = 10*x + 5 + (sin(x*pi))*gamma;
endfunction
```

6.9 APPROSSIMAZIONE DI FUNZIONI

6.9.1 *Exercise 4.1 on textbook*

```
function [ ascisse, interpolatedYs, realYs ] = exercise41
   (ascisse)
xVector = [0 1 2 3 4]'; #costruisco vettore colonna delle
   ascisse da interpolare
ordinates = computesOrdinates(xVector);

# the following two statement are need to initialize the
# vectors realYs and interpolatedYs
interpolatedYs = ascisse;
realYs = ascisse;

# perform the computation and find the interpolated value
   and
# the effective value.
for i=1:length(ascisse)
  interpolatedYs(i) = interpolationPoly(ascisse(i),
     ordinates);
  realYs(i) = contextFunction(ascisse(i));
end

endfunction

function [ ordinates ] = computesOrdinates(xVector)
    ordinates = xVector;
    for i=1:5
      ordinates(i) = contextFunction(xVector(i));
```

```
    end
endfunction

function [result] = contextFunction(x)
  result = 4*x^(2) -(12*x) +1;
endfunction

function [result] = interpolationPoly(x, ordinates)
  result = ordinates(1)*baseVector04(x) + ...
    ordinates(2)*baseVector14(x) + ...
    ordinates(3)*baseVector24(x) + ...
    ordinates(4)*baseVector34(x) + ...
    ordinates(5)*baseVector44(x);
endfunction

# costruzione della base di Lagrange
function [result] = baseVector04(x)
  result = ((x-1)*(x-2)*(x-3)*(x-4))/((0-1)*(0-2)*(0-3)
     *(0-4));
endfunction

function [result] = baseVector14(x)
  result = ((x-0)*(x-2)*(x-3)*(x-4))/((1-0)*(1-2)*(1-3)
     *(1-4));
endfunction

function [result] = baseVector24(x)
  result = ((x-0)*(x-1)*(x-3)*(x-4))/((2-0)*(2-1)*(2-3)
     *(2-4));
endfunction

function [result] = baseVector34(x)
  result = ((x-0)*(x-1)*(x-2)*(x-4))/((3-0)*(3-1)*(3-2)
     *(3-4));
endfunction

function [result] = baseVector44(x)
  result = ((x-0)*(x-1)*(x-2)*(x-3))/((4-0)*(4-1)*(4-2)
     *(4-3));
endfunction
```

6.9.2 *Script for Exercise 4.1 on textbook*

```
# script for easy testing of exercise 4.1
ascisse = -20:1:20;
[ascisse, interpolatedYs, realYs] = exercise41(ascisse);
plot(ascisse, realYs, "c", ascisse, interpolatedYs, "b+");
grid;
print 'exercise41PlotOutput.tex' '-dTex' '-S800, 600';
```

6.9.3 *Exercise 4.6 on textbook - Differenze divise*

```
function [differenzeDiviseVector] = differenzeDiviseEngine
   (ascisseVector, functionValuesVector)
    differenzeDiviseVector = functionValuesVector;
    n = length(functionValuesVector) - 1;
    for i=1:n
        for j=n+1:-1:i+1
            differenzeDiviseVector(j) = (
               differenzeDiviseVector(j) - ...
                differenzeDiviseVector(j-1)) / (
                   ascisseVector(j) - ...
                ascisseVector(j-i));
        end
    end
endfunction
```

Il costo di questo algoritmo è dato dalla seguente relazione:

$$\sum_{i=1}^{n} 3(n-i+1) = 3\sum_{k=1}^{n} k = 3\frac{n(n+1)}{2} \in O(n^2)$$

6.9.4 *Exercise 4.7 on textbook - Horner generalizzato*

```
function [ordinateVector] = HornerGeneralizzato(
   ascisseToEvaluateVector, ...
    ascisseVector, differenzeDiviseVector)

    # initialize the vector ordinateVector to be a vector
       of the same length
    # of ascisseToEvaluateVector vector
    ordinateVector = ascisseToEvaluateVector;
```

```
    for i=1:length(ascisseToEvaluateVector)
        ordinateVector(i) = internal_HornerGeneralizzato
            (...
             ascisseToEvaluateVector(i), ascisseVector,
                differenzeDiviseVector);
    end

end

function [y] = internal_HornerGeneralizzato(x, ...
    ascisseVector, differenzeDiviseVector)
    # Assert: n = length(differenzeDiviseVector) = length(
       ascisseVector)
    n = length(differenzeDiviseVector);
    y = differenzeDiviseVector(n);
    for k=n-1:-1:1
        y = y * (x - ascisseVector(k)) +
            differenzeDiviseVector(k);
    end
end
```

Il costo di questo algoritmo è $3n \in O(n)$.

6.9.5 *Exercise 4.7 on textbook - Function for testing*

```
function [] = exercise47()
    toInterpolateAscisseVector = linspace(-4.5, 4.5,80)';

    interpolationOrdinateVector =
       toInterpolateAscisseVector;
    for i=1:length(toInterpolateAscisseVector)
        interpolationOrdinateVector(i) = realFunction(
           toInterpolateAscisseVector(i));
    end

    # linspace return a row vector, so I transpose in
       order to obtain a column
    # vector
    interpolationAscisseVector1 = -5:0.5:5;
    interpolatedFunctionValuesVector1 = exercise_Internal
       (...
```

```
        interpolationAscisseVector1,
            toInterpolateAscisseVector);

    # linspace return a row vector, so I transpose in
        order to obtain a column
    # vector
    interpolationAscisseVector2 = -5:1:5;
    interpolatedFunctionValuesVector2 = exercise_Internal
        (...
        interpolationAscisseVector2,
            toInterpolateAscisseVector);

    # linspace return a row vector, so I transpose in
        order to obtain a column
    # vector
    interpolationAscisseVector3 = -5:2:5;
    interpolatedFunctionValuesVector3 = exercise_Internal
        (...
        interpolationAscisseVector3,
            toInterpolateAscisseVector);

    # initialing the real function value
    #errorVector = toInterpolateAscisseVector;
    #for i=1:length(toInterpolateAscisseVector)
    #    errorVector = realFunction(
        toInterpolateAscisseVector(i)) ...
    #        - toInterpolateOrdinateVector(i);
    #end

    #scale = 1:1:length(interpolationAscisseVector);
    #semilogy(scale, errorVector, "b");
    #grid;
    #print 'exercise47-ErrorPlotOutput.tex' '-dTex' '-S800
        , 600';

    plot(toInterpolateAscisseVector,
        interpolationOrdinateVector, "b", ...
        toInterpolateAscisseVector,
            interpolatedFunctionValuesVector1, "c", ...
        toInterpolateAscisseVector,
            interpolatedFunctionValuesVector2, "g", ...
            toInterpolateAscisseVector,
                interpolatedFunctionValuesVector3, "r");
```

```
    grid;
    print 'exercise47-CorrectPlotOutput.tex' '-dTex' '-
       S800, 600';

endfunction

function [interpolatedFunctionValuesVector] =
   exercise_Internal(...
    interpolationAscisseVector, toInterpolateAscisseVector
       )

    interpolationOrdinateVector =
       interpolationAscisseVector;
    for i=1:length(interpolationAscisseVector)
        interpolationOrdinateVector(i) = realFunction(
           interpolationAscisseVector(i));
    end

    differenzeDiviseVector = differenzeDiviseEngine(
       interpolationAscisseVector, ...
        interpolationOrdinateVector);

    interpolatedFunctionValuesVector = HornerGeneralizzato
       (toInterpolateAscisseVector, ...
        interpolationAscisseVector, differenzeDiviseVector
           );
endfunction

function [y] = realFunction(x)
    y = 1 / (1 + x^(2));
endfunction
```

6.9.6 *Exercise 4.8 on textbook - Differenze divise di Hermite*

```
function [doubledAscisseVector, differenzeDiviseVector] =
   ...
    hermiteDifferenzeDiviseEngine(ascisseVector, ...
        functionValuesVector, firstDerivateValuesVector)

    # build two column vector, doubling the dimension of
       the original ones,
```

```
    # passed as parameters.
    doubledAscisseVector = ones(length(ascisseVector)*2,
       1);
    doubledFunctionValuesVector = ones(length(
       functionValuesVector)*2, 1);

    # setting up the vectors, doubling information and
       setting the first derivate
    # values when needed
    for i = 1:length(functionValuesVector)
        doubledAscisseVector(2*(i-1)+1) = ascisseVector(i)
           ;
        doubledAscisseVector(2*(i-1)+2) = ascisseVector(i)
           ;

        doubledFunctionValuesVector(2*(i-1)+1) =
           functionValuesVector(i);
        doubledFunctionValuesVector(2*(i-1)+2) =
           firstDerivateValuesVector(i);
    end

    differenzeDiviseVector =
       hermiteDifferenzeDiviseEngineInternal(...
        doubledAscisseVector, doubledFunctionValuesVector)
           ;
endfunction

function [differenzeDiviseVector] =
   hermiteDifferenzeDiviseEngineInternal ...
    (ascisseVector, functionValuesVector)

    # Assume that in ascisseVector thare are a duplicate
       foreach original ascissa
    n = length(ascisseVector) - 1;

    # setup the differenzeDiviseVector vector
    differenzeDiviseVector = functionValuesVector;

    # build the derivative of function f on odd positions
    for i=n:-2:3
        differenzeDiviseVector(i) = (
           differenzeDiviseVector(i) -
           differenzeDiviseVector(i-2)) ...
```

```
        / (ascisseVector(i) - ascisseVector(i-2));
    end

    for i=2:n
        for j=n+1:-1:i+1
            differenzeDiviseVector(j) = (
                differenzeDiviseVector(j) - ...
                 differenzeDiviseVector(j-1)) / (
                    ascisseVector(j) - ...
                 ascisseVector(j-i));
        end
    end
endfunction
```

6.9.7 *Hermite engine*

```
function [ordinateVector] = Hermite(
   ascisseToEvaluateVector, ...
    ascisseVector, differenzeDiviseVector)

    # initialize the vector ordinateVector to be a vector
       of the same length
    # of ascisseToEvaluateVector vector
    ordinateVector = ascisseToEvaluateVector;
    for i=1:length(ascisseToEvaluateVector)
        ordinateVector(i) = internal_Hermite(...
            ascisseToEvaluateVector(i), ascisseVector,
               differenzeDiviseVector);
    end

end

function [y] = internal_Hermite(x, ...
    ascisseVector, differenzeDiviseVector)
    # Assert: n = length(differenzeDiviseVector) = length(
       ascisseVector)
    n = length(differenzeDiviseVector);
    y = 0;
    polynomion = 1;

    for i=1:n
```

```
        y = y + differenzeDiviseVector(i)*polynomion;
        polynomion = polynomion * (x - ascisseVector(i));
    end
end
```

6.9.8 *Exercise 4.9 on textbook*

```
function [] = exercise49()

    # setup the vector with the ascisse where we want to
       evaluate
    toInterpolateAscisseVector = linspace(-0, 1, 101)';

    # initialize the two vectors for real function and
       first derivate values
    realFunctionValuesVector = toInterpolateAscisseVector;
    realFirstDerivateValuesVector =
       toInterpolateAscisseVector;

    # computes the effective values for real and first
       derivate function
    for i=1:length(toInterpolateAscisseVector)
        realFunctionValuesVector(i) = realFunction(
           toInterpolateAscisseVector(i));
        realFirstDerivateValuesVector(i) =
           realFunctionFirstDerivate(...
            toInterpolateAscisseVector(i));
    end

    # setup the vector with the interpolation ascisse, its
       function values and
    # first derivative values vectors
    interpolationAscisseVector = [0; 0.25; 0.5; 0.75; 1];
       #linspace(0,1,5);
    interpolationOrdinateVector =
       interpolationAscisseVector;
    firstDerivateValuesVector = interpolationAscisseVector
       ;

    for i=1:length(interpolationAscisseVector)
```

```
        interpolationOrdinateVector(i) = realFunction(
            interpolationAscisseVector(i));
        firstDerivateValuesVector(i) =
            realFunctionFirstDerivate(
            interpolationAscisseVector(i));
    end

    # Newton: differenze divise
    newtonDifferenzeDiviseVector = differenzeDiviseEngine(
        interpolationAscisseVector, ...
        interpolationOrdinateVector);

    # Hermite: differenze divise
    [doubledAscisseVector, hermiteDifferenzeDiviseVector]
       = hermiteDifferenzeDiviseEngine(...
        interpolationAscisseVector,
            interpolationOrdinateVector,
            firstDerivateValuesVector);

    # plot the differenze divise curve
    plot(interpolationAscisseVector,
        newtonDifferenzeDiviseVector, "b", ...
        doubledAscisseVector,
            hermiteDifferenzeDiviseVector, "r");
    grid;
    print 'exercise49-DifferenzeDivisePlotOutput.tex' '-
        dTex' '-S800, 600';

    # Newton: interpolation step.
    newtonToInterpolateOrdinateVector =
        HornerGeneralizzato(toInterpolateAscisseVector, ...
        interpolationAscisseVector,
            newtonDifferenzeDiviseVector);

    # Hermite: interpolation step.
    hermiteToInterpolateOrdinateVector = Hermite(
        toInterpolateAscisseVector, ...
        doubledAscisseVector,
            hermiteDifferenzeDiviseVector);

    plot(toInterpolateAscisseVector,
        realFunctionValuesVector, "g", ...
```

```
        toInterpolateAscisseVector,
           newtonToInterpolateOrdinateVector, "b", ...
        toInterpolateAscisseVector,
           hermiteToInterpolateOrdinateVector, "r");
    grid;
    print 'exercise49-normalPlotOutput.tex' '-dTex' '-S800
       , 600';

    plot(toInterpolateAscisseVector, abs(
       realFunctionValuesVector - ...
        newtonToInterpolateOrdinateVector), "b", ...
        toInterpolateAscisseVector, abs(
           realFunctionValuesVector - ...
        hermiteToInterpolateOrdinateVector), "r");
    grid;
    print 'exercise49-errorsPlotOutput.tex' '-dTex' '-S800
       , 600';

    semilogy(toInterpolateAscisseVector, abs(
       realFunctionValuesVector - ...
        newtonToInterpolateOrdinateVector), "b", ...
        toInterpolateAscisseVector, abs(
           realFunctionValuesVector - ...
        hermiteToInterpolateOrdinateVector), "r");
    grid;
    print 'exercise49-errorsSemilogYPlotOutput.tex' '-dTex
       ' '-S800, 600';

    #semilogy(toInterpolateAscisseVector,
       realFunctionValuesVector, "b", ...
    #    toInterpolateAscisseVector,
       realFirstDerivateValuesVector, "r", ...
    #    toInterpolateAscisseVector,
       newtonToInterpolateOrdinateVector, "c", ...
    #    toInterpolateAscisseVector,
       hermiteToInterpolateOrdinateVector, "g");
    #grid;
    #print 'exercise49-semilogyPlotOutput.tex' '-dTex' '-
       S800, 600';

endfunction

function [y] = realFunction(x)
```

```
    y = (x-1)^(9);
endfunction

function [y] = realFunctionFirstDerivate(x)
    y = 9*(x-1)^(8);
endfunction
```

6.9.9 *Common Code For Exercises 4.11 and 4.15 on textbook*

```
function [domain, rungeErrorVector, bernsteinErrorVector,
    ...
    rungeEvaluationIntervalOutput,
        rungeInterpolatedOrdinateVectorOutput,...
    bernsteinEvaluationIntervalOutput,
        bernsteinInterpolatedOrdinateVectorOutput] = ...
      exercises411415CommonFactor(errorIntervalVector,
          ascisseCreationFunctionName)

    # build the vectors for valuate the interpolation and
       study the error.
    evaluateSetDimension = 10^(3);
    rungeEvaluationInterval = linspace(-5, 5,
       evaluateSetDimension)';
    bernsteinEvaluationInterval = linspace(-1, 1,
       evaluateSetDimension)';

    # have the same dimension
    rungeEvaluatedFunctionValuesVector =
       rungeEvaluationInterval;
    bernsteinEvaluatedFunctionValuesVector =
       bernsteinEvaluationInterval;

     # build the real functions values for evaluate real
        functions
    rungeEvaluatedFunctionValuesVector = rungeRealFunction
       (...
      rungeEvaluationInterval);
    bernsteinEvaluatedFunctionValuesVector =
       bernsteinRealFunction(...
      bernsteinEvaluationInterval);

```

```
    # get the error vector dimension
    errorVectorDimension = length(errorIntervalVector);

    # initializing the vectors with a dummy value
    rungeErrorVector = ones(errorVectorDimension,1);
    bernsteinErrorVector = ones(errorVectorDimension,1);

    # start from 1 to have at least one ascisse for
       Bernstein interval domain.
    for i=1:errorVectorDimension

        # current number of ascisse to build the
           approximation
        # +1 because n is the rank, n+1 are the ascisse
        n = errorIntervalVector(i) + 1;

        # build the ascisse where I want to interpolate
           the functions
        # the limits {-5, 5, -1, 1} are fixed by
           definition of the exercise
        # fix the length of the set of interpolation
           ascisse
        # using the strategy that is passed as parameter
        rungeInterpolationAscisseVector = feval(
           ascisseCreationFunctionName, ...
            -5, 5, n)';
        bernsteinInterpolationAscisseVector = feval(
           ascisseCreationFunctionName, ...
            -1, 1, n)';

        # compute the function values on each
           interpolation ascisse
        rungeInterpolationOrdinateVector =
           rungeRealFunction(...
            rungeInterpolationAscisseVector);
        bernsteinInterpolationOrdinateVector =
           bernsteinRealFunction(...
            bernsteinInterpolationAscisseVector);

        # compute the differenze divise vectors for the
           two functions.
        rungeDifferenzeDiviseVector =
           differenzeDiviseEngine(...
```

```
            rungeInterpolationAscisseVector,
                rungeInterpolationOrdinateVector);
        bernsteinDifferenzeDiviseVector =
            differenzeDiviseEngine(...
             bernsteinInterpolationAscisseVector,
                bernsteinInterpolationOrdinateVector);

        # compute the polynomion and valuate on an
            interval
        rungeInterpolatedOrdinateVector =
            HornerGeneralizzato(...
             rungeEvaluationInterval, ...
             rungeInterpolationAscisseVector,
                rungeDifferenzeDiviseVector);
        bernsteinInterpolatedOrdinateVector =
            HornerGeneralizzato(...
             bernsteinEvaluationInterval, ...
             bernsteinInterpolationAscisseVector,
                bernsteinDifferenzeDiviseVector);

        if i == errorVectorDimension
          # saving the value for output and outside plot
              the approximated functions
            rungeEvaluationIntervalOutput =
                rungeEvaluationInterval;
            rungeInterpolatedOrdinateVectorOutput =
                rungeInterpolatedOrdinateVector;

            bernsteinEvaluationIntervalOutput =
                bernsteinEvaluationInterval;
            bernsteinInterpolatedOrdinateVectorOutput = ...
              bernsteinInterpolatedOrdinateVector;
        end

        # calculate the error
        rungeErrorVector(i) = max(abs(
            rungeEvaluatedFunctionValuesVector ...
              - rungeInterpolatedOrdinateVector));
        bernsteinErrorVector(i) = max(abs(
            bernsteinEvaluatedFunctionValuesVector ...
              - bernsteinInterpolatedOrdinateVector));

    end
```

```
    # costruisco il dominio su cui rappresentare gli
       errori commessi
    domain = [1:1:errorVectorDimension];

endfunction

function [y] = rungeRealFunction(x)
    y = 1./(1 .+ x.^(2));
endfunction

function [y] = bernsteinRealFunction(x)
    y = abs(x);
endfunction
```

6.9.10 *Exercise 4.11 on textbook*

```
function [] = exercise411()

    # the factored code allow me to specify the function
       that produce the
    # array of interpolation ascisse
    [domain, rungeErrorVector, bernsteinErrorVector, ...
    rungeEvaluationIntervalOutput,
       rungeInterpolatedOrdinateVectorOutput,...
    bernsteinEvaluationIntervalOutput,
       bernsteinInterpolatedOrdinateVectorOutput] = ...
      exercises411415CommonFactor([1:1:50], "linspace");

    # plot the output values sent by the common code
    plot(rungeEvaluationIntervalOutput, ...
      rungeInterpolatedOrdinateVectorOutput, "b", ...
      bernsteinEvaluationIntervalOutput, ...
      bernsteinInterpolatedOrdinateVectorOutput, "r");
    grid;
    print "exercise411-currentApproximationPlotOutput.tex"
        '-dTex' '-S800,600';

    # plot the errors
    semilogy(domain, rungeErrorVector, "b", ...
        domain, bernsteinErrorVector, "r");
```

```
    grid;
    print "exercise411-ErrorsSemilogYPlotOutput.tex" '-
       dTex' '-S800, 600';

    loglog(domain, rungeErrorVector, "b", ...
        domain, bernsteinErrorVector, "r");
    grid;
    print "exercise411-ErrorsLoglogPlotOutput.tex" '-dTex'
        '-S800, 600';

endfunction
```

6.9.11 *BuildChebyshevAscisse Maker*

```
function [chebyshevAscisse] = buildChebyshevAscisse(a, b,
   n)

    # use the (4.40) formula to build the following values
    length = n;
    chebyshevAscisse = ones(length, 1);
    for i = length:-1:1
        chebyshevAscisse(i) = cos((2*(i - 1) + 1)*pi /
           (2*(length+1)));

        chebyshevAscisse(i) = (a + b)/2 + ((b-a)/2)*
           chebyshevAscisse(i);
    end
endfunction
```

6.9.12 *Chebyshev Ascisse Example*

```
function [] = chebyshevAscissePlotter ()

  # build the ascisse vector
  ascisse = buildChebyshevAscisse(-10, 20, 100);
  ascisseForProjection = linspace(-10, 20, 100);

  # plot the errors
  plot(ascisse, zeros(length(ascisse),1) , "b+", ...
    ascisseForProjection, ascisse, "r+");
```

```
  grid;
  print "ChecyshevAscissePlotOutput.tex" '-dTex' '-S800,
     600';

   # build the ascisse vector
  ascisse = buildChebyshevAscisse(-1, 1, 100);
  ascisseForProjection = linspace(-1, 1, 100);

  # plot the errors
  plot(ascisse, zeros(length(ascisse),1) , "b+", ...
    ascisseForProjection, ascisse, "r+");
  grid;
  print "ChecyshevAscisseUnaryPlotOutput.tex" '-dTex' '-
     S800, 600';
endfunction
```

6.9.13 *Exercise 4.15 on textbook*

```
function [chebyshevAscisse] = exercise415()

   # the factored code allow me to specify the function
      that produce the
   # array of interpolation ascisse
   [domain, rungeErrorVector, bernsteinErrorVector, ...
   rungeEvaluationIntervalOutput,
      rungeInterpolatedOrdinateVectorOutput,...
   bernsteinEvaluationIntervalOutput,
      bernsteinInterpolatedOrdinateVectorOutput] = ...
     exercises411415CommonFactor([2:2:50], "
        buildChebyshevAscisse");

   # plot the output values sent by the common code
   plot(rungeEvaluationIntervalOutput, ...
     rungeInterpolatedOrdinateVectorOutput, "b", ...
     bernsteinEvaluationIntervalOutput, ...
     bernsteinInterpolatedOrdinateVectorOutput, "r");
   grid;
   print "exercise415-currentApproximationPlotOutput.tex"
       '-dTex' '-S800, 600';

   # plot the errors
```

```
    semilogy(domain, rungeErrorVector, "b", ...
        domain, bernsteinErrorVector, "r");
    grid;
    print "exercise415-ErrorsSemilogYPlotOutput.tex" '-
       dTex' '-S800, 600';

    loglog(domain, rungeErrorVector, "b", ...
        domain, bernsteinErrorVector, "r");
    grid;
    print "exercise415-ErrorsLoglogPlotOutput.tex" '-dTex'
        '-S800, 600';

endfunction
```

6.9.14 *triangularSystemSolver*

```
## triangularSystemSolver

function [ ret ] = triangularSystemSolver (A, b,
   typeOfMatrix)
ret = feval(typeOfMatrix, A, b);
end
endfunction

function [ unknowns ] = lowerSolveEngine(A, b)
  unknowns = b;
  n = length(unknowns);
  for j = 1:n
    unknowns(j) = unknowns(j) / A(j,j);
    for i = j+1:n
      unknowns(i) = unknowns(i) - A(i,j) * unknowns(j);
    end
  end
endfunction

function [ unknowns ] = upperSolveEngine(A, b)
  unknowns = b;
  n = length(unknowns);
  for j = n:-1:1
    unknowns(j) = unknowns(j) / A(j,j);
    for i = 1:j-1
```

```
      unknowns(i) = unknowns(i) - A(i,j) * unknowns(j);
    end
  end
endfunction
```

6.9.15 *tridiagonaleLUFactor*

```
function [lVector, uVector] = tridiagonaleLUFactor(matrix)

    # find the dimension of the matrix to build the two
       vector l and u
    dimension = rows(matrix);

    # build the returning vectors
    uVector = zeros(dimension, 1);
    lVector = zeros(dimension, 1);

    # initializing the firsts' elements
    uVector(1) = 2;
    lVector(1) = 0; # Does this assignment is right?

    for i=2:dimension
        lVector(i) = matrix(i, i-1) / uVector(i-1);
        uVector(i) = 2- (matrix(i-1, i) * lVector(i));
    end

endfunction
```

6.9.16 *hVarphiXiVectorsBuilder*

```
function [hVector, varPhiVector, xiVector] = ...
    hVarphiXiVectorsBuilder(interpolationAscisseVector)

    # compute the dimension of the new vectors to be built
    dimension = length(interpolationAscisseVector) - 1;

    # initialization of the column-vector h
    hVector = zeros(dimension, 1);
```

```
    # setting a dimension of minus one respect to hVector
       because at every step
    # we use two elements of hVector.
    varPhiXiDimension = dimension -1;
    varPhiVector = zeros(varPhiXiDimension, 1);
    xiVector = zeros(varPhiXiDimension, 1);

    # setting the hVector vector
    for i = 1:dimension
        hVector(i) = interpolationAscisseVector(i+1) -
           interpolationAscisseVector(i);
    end

    #hVector

    # setting the varPhiVector and xiVector vectors
    for i = 1:varPhiXiDimension
        varPhiVector(i) = hVector(i) / (hVector(i) +
           hVector(i+1));
        xiVector(i) = hVector(i+1) / (hVector(i) + hVector
           (i+1));
    end

endfunction
```

6.9.17 *triangularBidiagonalMatrixBuilder*

```
function [matrix] = triangularBidiagonalMatrixBuilder(...
    diagonalVector, bidiagonalVector, matrixType)

    # invoco la strategy che mi permette di costruire il
       tipo di matrice
    # che viene richiesto.
    matrix = feval(matrixType, diagonalVector,
       bidiagonalVector);

endfunction

function [matrix] = internal_LowerBidiagonalMatrix(
   diagonalVector, bidiagonalVector)
```

```
    # set the dimension refering to the "main" vector, ie.
        the diagonal vector.
    dimension = length(diagonalVector);

    # build the matrix, keeping the zero matrix as initial
        matrix value.
    matrix = zeros(dimension);

    for i = 1:dimension
        matrix(i, i) = diagonalVector(i);

        if (i-1) > 0
            # set the left hand side (column) of the
                diagonal item
            matrix(i, i-1) = bidiagonalVector(i);
        end
    end
endfunction

function [matrix] = internal_UpperBidiagonalMatrix(
   diagonalVector, bidiagonalVector)
    # set the dimension refering to the "main" vector, ie.
        the diagonal vector.
    dimension = length(diagonalVector);

    # build the matrix, keeping the zero matrix as initial
        matrix value.
    matrix = zeros(dimension);

    for i = 1:dimension
        matrix(i, i) = diagonalVector(i);

        if (i+1) < (dimension + 1)
            # for the upper type we set the right-adiacent
                item
            matrix(i, i + 1) = bidiagonalVector(i);
        end
    end
endfunction
```

6.9.18 *cubicSplainEngine*

```
function [hVector, varPhiVector, xiVector, lVector,
   uVector, lowerBiadiagonalMatrix, ...
    upperBiadiagonalMatrix, diffDiviseVector, mis,
       interpolatedValues] = ...
      cubicSplainEngine(interpolationAscisseVector,
         functionValuesVector, ...
        splainSchemeMatrixToFactorStrategyName,
           splainSchemeMisStrategyName, ...
        diffDiviseStrategyName, domain)

    # build the vectors
    [hVector, varPhiVector, xiVector] = ...
        hVarphiXiVectorsBuilder(interpolationAscisseVector
           );

    # build the matrix to factor, this costruction process
        depend on the parameter
    # splainSchemeMatrixToFactorStrategyName
    matrixToFactor = feval(
       splainSchemeMatrixToFactorStrategyName,
       varPhiVector, xiVector);

    # factor the matrix - works only with natural scheme
    #[lVector, uVector] = tridiagonaleLUFactor(
       matrixToFactor);

    # build the diagonal vector and build the lower trian.
        matrix
    #lowerDiagonalVector = ones(length(lVector), 1);
    #lowerBiadiagonalMatrix =
       triangularBidiagonalMatrixBuilder(...
    #    lowerDiagonalVector, lVector, '
       internal_LowerBidiagonalMatrix');

    # build the upper trian. matrix - check passed
    #upperBiadiagonalMatrix =
       triangularBidiagonalMatrixBuilder(...
    #    uVector, xiVector, '
       internal_UpperBidiagonalMatrix');
```

```
    # build the differenze divise vector and solve the
       system
    # invoking polymorphically the strategy to build the
       differenze divise vector
    diffDiviseVector = feval(diffDiviseStrategyName, ...
        interpolationAscisseVector, functionValuesVector);

    # factor LU - todo: use a more efficient factorization
        like above
    [L, U, p, unknowns] = PALUmethod (matrixToFactor,
       diffDiviseVector);

    # solve the system and find the mis
    mis = triangularSystemSolver(U, ...
        triangularSystemSolver(L, diffDiviseVector, ...
        "lowerSolveEngine"), "upperSolveEngine");

    # manage the results, adjusting the mis vector
    mis = feval(splainSchemeMisStrategyName, mis,
       diffDiviseVector);

    # initialization of the interpolated values vactor
    interpolatedValues = zeros(length(domain), 1);

    # caching the values of qi and ri, initialized with a
       dummy row
    chaceMatrix = [-1 -1];
    for index = 2:length(interpolationAscisseVector)

      ri = internal_RiComputer(index, mis,
         interpolationAscisseVector, ...
          functionValuesVector, hVector);

      qi = internal_QiComputer(index, mis,
         interpolationAscisseVector, ...
          functionValuesVector, hVector);

      chaceMatrix = [chaceMatrix; ri qi];
    end

    # cleaning up the first dummy row
    chaceMatrix = chaceMatrix(2:rows(chaceMatrix), :);
```

```
    for i = 1:length(domain)
63      for index = 2:length(interpolationAscisseVector)
            x = domain(i);
            if x >= interpolationAscisseVector(index-1) &&
                ...
                x <= interpolationAscisseVector(index)

68              ri = chaceMatrix(index-1, 1);

                qi = chaceMatrix(index-1, 2);

                interpolatedValues(i) = internal_Eval(x,
                    index, ...
73                  interpolationAscisseVector, mis, ri,
                        qi, hVector(index-1));

                break;
            end
        end
78  end

endfunction

function [diffDiviseVector] =
    internal_naturalBuildThreeDifferenzeDiviseVector(...
83   interpolationAscisseVector, functionValuesVector)

    # compute the dimension of the returning vector
    dimension = length(interpolationAscisseVector) - 2;

88  # initialize the vector
    diffDiviseVector = zeros(dimension, 1);

    for i = 1:dimension
        diffDiviseVector(i) = ...
93          (...
                internal_BuildDiffDivisaBetweenTwoAscisse
                    (...
                    interpolationAscisseVector(i+2),...
                    functionValuesVector(i+2), ...
                    interpolationAscisseVector(i+1), ...
98                  functionValuesVector(i+1)) - ...
```

```
                internal_BuildDiffDivisaBetweenTwoAscisse
                    (...
                     interpolationAscisseVector(i+1),...
                     functionValuesVector(i+1), ...
                     interpolationAscisseVector(i+0), ...
                     functionValuesVector(i+0)) ...
            ) / ...
            (interpolationAscisseVector(i+2) -
                interpolationAscisseVector(i+0));
    end
    diffDiviseVector = 6*diffDiviseVector;

endfunction

function [diffDiviseVector] =
   internal_notAKnotBuildThreeDifferenzeDiviseVector(...
    interpolationAscisseVector, functionValuesVector)

    [diffDiviseVector] =
       internal_naturalBuildThreeDifferenzeDiviseVector
       (...
      interpolationAscisseVector, functionValuesVector);

    # adjusting the vector according to linear system on
       page 99
    diffDiviseVector = [0; diffDiviseVector; 0];

endfunction

function [value] =
   internal_BuildDiffDivisaBetweenTwoAscisse(xi, fi, xi_1,
    fi_1)
    value = (fi - fi_1) / (xi - xi_1);
end

function [matrix] = normalSplainScheme_BuildMatrixToFactor
   (...
    varPhiVector, xiVector)

    dimension = length(varPhiVector);
    matrix = zeros(dimension);

    for i = 1:dimension
```

```
133
        # on the left (column) of the diagonal item
        if (i-1) > 0
          matrix(i, i-1) = varPhiVector(i);
        end
138
        # diagonal item, always fixed to 2
        matrix(i, i) = 2;

        # on the right (column) of the diagonal item
143     if (i+1) < (dimension + 1)
          matrix(i, i+1) = xiVector(i);
        end

    end
148
endfunction

function [mis] = normalSplainScheme_BuildMisVector(vector,
    diffDivise)

153 # We have to add two conditions to the n-1 already
       present
    mis = zeros(length(vector) + 2, 1);

    for i = 2:(length(mis)-1)
        mis(i) = vector(i-1);
158 end

endfunction

function [matrix] =
   notAKnotSplainScheme_BuildMatrixToFactor(...
163  varPhiVector, xiVector)

    dimension = length(varPhiVector) + 2;
    matrix = zeros(dimension);

168 # now we complete the setup according to (4.59) of
       textbook
    # setting the first row
    matrix(1,1) = xiVector(1);
    matrix(1,2) = -1;
```

```
    matrix(1,3) = varPhiVector(1);

    # we stop at dimension-2 for allow a post completion of the matrix on the
    # last two rows, starting from the third row, because the first two rows
    # are managed outside the loop
    for i = 2:(dimension - 1)
        # inside this loop we are free to check the index boundaries
        # because we are modifying a submatrix that doesn't share both the first
        # row and the first column.

        # on the left (column) of the diagonal item
        matrix(i, i-1) = varPhiVector(i-1);

        # diagonal item, always fixed to 2
        matrix(i, i) = 2;

        # on the right (column) of the diagonal item
        matrix(i, i+1) = xiVector(i-1);

    end

    varXiVectorLength = length(varPhiVector);

    # setting the last row
    matrix(dimension, dimension - 2) = xiVector(varXiVectorLength);
    matrix(dimension, dimension - 1) = -1;
    matrix(dimension, dimension) = varPhiVector(varXiVectorLength);

endfunction

function [mis] = notAKnotSplainScheme_BuildMisVector(vector, diffDivise)

    # nothign to adjust here, the vector that resolve the system is already okey.
    mis = vector;
```

```
endfunction

function [value] = internal_RiComputer(index, mis,
    interpolationAscisseVector, ...
    functionValuesVector, hVector)

    value = functionValuesVector(index-1) - ((hVector(
        index-1)^(2))/6)*mis(index-1);
endfunction

function [value] = internal_QiComputer(index, mis,
    interpolationAscisseVector, ...
    functionValuesVector, hVector)

    value = ((functionValuesVector(index) -
        functionValuesVector(index-1))/hVector(index-1))...
        - (hVector(index-1)/6)*(mis(index) - mis(index-1))
            ;
endfunction

function [value] = internal_Eval(ascissa, index, ...
    interpolationAscisseVector, mis, ri, qi, hi)

    a = (ascissa - interpolationAscisseVector(index -1))
        ^(3);
    b = (interpolationAscisseVector(index) - ascissa)^(3);
    c = (a * mis(index)) + (b * mis(index-1));
    d = c / (6*hi);
    e = ascissa - interpolationAscisseVector(index-1);
    value = d + (e * qi) + ri;
endfunction
```

6.9.19 *Spline stress*

```
function [] = splineStress()

    intervalMagnitude = 5;
    [interpolationAscisseVector] =
        buildInterpolationPartition(intervalMagnitude);
```

```
    functionValuesVector = realFunction(
        interpolationAscisseVector, intervalMagnitude);

    # build the domain vectors containing the range where
        we want to interpolate
    # the real function using 10000 points for plotting.
    domain = linspace(-intervalMagnitude,
        intervalMagnitude, 10^(4))';
    realFunctionValuesVector = realFunction(domain,
        intervalMagnitude);

    # setting up the parameter to drive the cubic splain
        engine.
    splainSchemeMatrixToFactorStrategyName = '
        normalSplainScheme_BuildMatrixToFactor';
    splainSchemeMisStrategyName = '
        normalSplainScheme_BuildMisVector';
    diffDiviseStrategyName = '
        internal_naturalBuildThreeDifferenzeDiviseVector';

    # go!
    [hVector, varPhiVector, xiVector, lVector, uVector,
        lowerBiadiagonalMatrix, ...
         upperBiadiagonalMatrix, diffDiviseVector, mis,
            normalInterpolatedValues] = ...
         cubicSplainEngine(interpolationAscisseVector,
            functionValuesVector, ...
         splainSchemeMatrixToFactorStrategyName,
            splainSchemeMisStrategyName, ...
         diffDiviseStrategyName, domain);

    # setting up the parameter to drive the cubic splain
        engine.
    splainSchemeMatrixToFactorStrategyName = '
        notAKnotSplainScheme_BuildMatrixToFactor';
    splainSchemeMisStrategyName = '
        notAKnotSplainScheme_BuildMisVector';
    diffDiviseStrategyName = '
        internal_notAKnotBuildThreeDifferenzeDiviseVector';

    [hVector, varPhiVector, xiVector, lVector, uVector,
        lowerBiadiagonalMatrix, ...
```

```
    upperBiadiagonalMatrix, diffDiviseVector, mis,
        notAKnotInterpolatedValues] = ...
      cubicSplainEngine(interpolationAscisseVector,
          functionValuesVector, ...
      splainSchemeMatrixToFactorStrategyName,
          splainSchemeMisStrategyName, ...
      diffDiviseStrategyName, domain);

octaveSpline = spline(interpolationAscisseVector,
    functionValuesVector, domain);

plot(domain, realFunctionValuesVector, "b", ...
    domain, octaveSpline, "r", ...
    interpolationAscisseVector, functionValuesVector,
        "o");
grid;
print 'splineStress-octaveInterpolationPlotOutput.tex'
    '-dTex' '-S800, 600';

plot(domain, realFunctionValuesVector, "b", ...
    domain, normalInterpolatedValues, "r", ...
    domain, notAKnotInterpolatedValues, "g", ...
    interpolationAscisseVector, functionValuesVector,
        "o");
grid;
print 'splineStress-interpolationPlotOutput.tex' '-
    dTex' '-S800, 600';

plot(domain, realFunctionValuesVector, "b", ...
    domain, normalInterpolatedValues, "r", ...
    domain, notAKnotInterpolatedValues, "g", ...
    interpolationAscisseVector, functionValuesVector,
        "+");
axis([-1.5, -0.5]);
grid;
print 'splineStress-
    zoomMinusOneInterpolationPlotOutput.tex' '-dTex' '-
    S800, 600';

plot(domain, realFunctionValuesVector, "b", ...
    domain, normalInterpolatedValues, "r", ...
    domain, notAKnotInterpolatedValues, "g", ...
```

```
        interpolationAscisseVector, functionValuesVector,
            "+");
    axis([-0.5, 0.5]);
    grid;
    print 'splineStress-zoomZeroInterpolationPlotOutput.
        tex' '-dTex' '-S800, 600';

    plot(domain, realFunctionValuesVector, "b", ...
        domain, normalInterpolatedValues, "r", ...
        domain, notAKnotInterpolatedValues, "g", ...
        interpolationAscisseVector, functionValuesVector,
            "+");
    axis([0.5, 1.5]);
    grid;
    print 'splineStress-zoomOneInterpolationPlotOutput.tex
        ' '-dTex' '-S800, 600';

    semilogy(domain, abs(realFunctionValuesVector -
        normalInterpolatedValues), "b", ...
        domain, abs(realFunctionValuesVector -
            notAKnotInterpolatedValues), "r");
    grid;
    print 'splineStress-errorsPlotOutput.tex' '-dTex' '-
        S800, 600';

    plot(domain, abs(realFunctionValuesVector -
        normalInterpolatedValues), "b", ...
        domain, abs(realFunctionValuesVector -
            notAKnotInterpolatedValues), "r");
    grid;
    print 'splineStress-linearErrorsPlotOutput.tex' '-dTex
        ' '-S800, 600';


endfunction

function [interpolationAscisseVector] =
    buildInterpolationPartition(intervalMagnitude)

    # build the partition vector and its associated vector
        of interpolation conditions
    # using twelve interpolation ascisse
```

```
    interpolationAscisseVector = linspace(-
        intervalMagnitude, intervalMagnitude, 12)';

    # the following lines allow to add points in which we
        want to study the
    # non-regularity of the function
    interpolationAscisseVector = [
        interpolationAscisseVector; -1; 0; 1];
    interpolationAscisseVector = sort(
        interpolationAscisseVector);
endfunction

# capture the real model of the function
# here I can't be able to use optimized code for vector
    operation because
# I have a custom segment-defined function
function [value] = realFunction(ascissa, intervalMagnitude
    )
    value = zeros(length(ascissa),1);
    for i = 1:length(ascissa)
        if ascissa(i) < -1
            value(i) = (ascissa(i) + intervalMagnitude)*(
                ascissa(i) ...
                 +1)*exp(ascissa(i));
        elseif ascissa(i) <= 1
            value(i) = sqrt(1 - ascissa(i)^(2));
        else
            value(i) = (ascissa(i) -1)*1/(ascissa(i)^2);
        end
    end
endfunction
```

6.9.20 *Exercise 4.19 - Runge interpolation*

```
function [rungeEvaluationInterval,
    rungeEvaluatedFunctionValuesVector,...
    rungeInterpolationAscisseVector,
        rungeFunctionValuesVector,...
    rungeNormalInterpolatedValues,
        rungeNotAKnotInterpolatedValues] = ...
```

```
exercise419runge(evaluateSetDimension, numberOfAscisse)

rungeIntervalMagnitude = 5;
rungeEvaluationInterval = linspace(-rungeIntervalMagnitude, ...
    rungeIntervalMagnitude, evaluateSetDimension)';

# have the same dimension
rungeEvaluatedFunctionValuesVector = rungeEvaluationInterval;

 # build the real functions values for evaluate real functions
for i=1:evaluateSetDimension
    rungeEvaluatedFunctionValuesVector(i) = rungeRealFunction(...
        rungeEvaluationInterval(i));
end

# build the partition vector and its associated vector of interpolation conditions
rungeInterpolationAscisseVector = linspace(-rungeIntervalMagnitude, ...
    rungeIntervalMagnitude, numberOfAscisse)';
for i=1:length(rungeInterpolationAscisseVector)
    rungeFunctionValuesVector(i) = rungeRealFunction(...
        rungeInterpolationAscisseVector(i));
end

# setting up the parameter to drive the cubic splain engine.
splainSchemeMatrixToFactorStrategyName = 'normalSplainScheme_BuildMatrixToFactor';
splainSchemeMisStrategyName = 'normalSplainScheme_BuildMisVector';
diffDiviseStrategyName = 'internal_naturalBuildThreeDifferenzeDiviseVector';

# go!
[hVector, varPhiVector, xiVector, lVector, uVector, lowerBiadiagonalMatrix, ...
```

```
        upperBiadiagonalMatrix, diffDiviseVector, mis,
            rungeNormalInterpolatedValues] = ...
        cubicSplainEngine(rungeInterpolationAscisseVector,
             rungeFunctionValuesVector, ...
        splainSchemeMatrixToFactorStrategyName,
            splainSchemeMisStrategyName,...
        diffDiviseStrategyName, rungeEvaluationInterval);

    # setting up the parameter to drive the cubic splain
        engine.
    splainSchemeMatrixToFactorStrategyName = '
        notAKnotSplainScheme_BuildMatrixToFactor';
    splainSchemeMisStrategyName = '
        notAKnotSplainScheme_BuildMisVector';
    diffDiviseStrategyName = '
        internal_notAKnotBuildThreeDifferenzeDiviseVector';

    [hVector, varPhiVector, xiVector, lVector, uVector,
        lowerBiadiagonalMatrix, ...
        upperBiadiagonalMatrix, diffDiviseVector, mis,
            rungeNotAKnotInterpolatedValues] = ...
        cubicSplainEngine(rungeInterpolationAscisseVector,
             rungeFunctionValuesVector, ...
        splainSchemeMatrixToFactorStrategyName,
            splainSchemeMisStrategyName,...
        diffDiviseStrategyName, rungeEvaluationInterval);

endfunction

endfunction

function [y] = rungeRealFunction(x)
    y = 1/(1 + x^(2));
endfunction
```

6.9.21 *Exercise 4.19 - Runge interpolationPlotter*

```
function [] = exercise419rungePlotter()

    evaluateSetDimension = 10^(4);

```

```
    [rungeEvaluationInterval,
       rungeEvaluatedFunctionValuesVector,...
    rungeInterpolationAscisseVector,
       rungeFunctionValuesVector,...
    rungeNormalInterpolatedValues,
       rungeNotAKnotInterpolatedValues] = ...
        exercise419runge(evaluateSetDimension, 10);

    plot(rungeEvaluationInterval,
       rungeEvaluatedFunctionValuesVector, "b", ...
        rungeEvaluationInterval,
           rungeNormalInterpolatedValues, "r", ...
        rungeEvaluationInterval,
           rungeNotAKnotInterpolatedValues, "g", ...
        rungeInterpolationAscisseVector,
           rungeFunctionValuesVector, "+");
    grid;
    print 'exercise419-rungeInterpolationPlotOutput.tex' '
       -dTex' '-S800, 600';
endfunction
```

6.9.22 *Exercise 4.19 - Bernstein interpolation*

```
function [bernsteinEvaluationInterval,
   bernsteinEvaluatedFunctionValuesVector,...
    bernsteinInterpolationAscisseVector,
       bernsteinFunctionValuesVector,...
    bernsteinNormalInterpolatedValues,
       bernsteinNotAKnotInterpolatedValues] =...
    exercise419bernstein(evaluateSetDimension,
       numberOfAscisse)

    # build the vectors for valuate the interpolation and
       study the error.

    bernsteinIntervalMagnitude = 1;
    bernsteinEvaluationInterval = linspace(-
       bernsteinIntervalMagnitude, ...
        bernsteinIntervalMagnitude, evaluateSetDimension)
           ';
```

```
    # have the same dimension
    bernsteinEvaluatedFunctionValuesVector =
        bernsteinEvaluationInterval;

     # build the real functions values for evaluate real
        functions
    for i=1:evaluateSetDimension
        bernsteinEvaluatedFunctionValuesVector(i) =
            bernsteinRealFunction(...
             bernsteinEvaluationInterval(i));
    end

    # build the partition vector and its associated vector
        of interpolation conditions
    bernsteinInterpolationAscisseVector = linspace(-
        bernsteinIntervalMagnitude,...
         bernsteinIntervalMagnitude, numberOfAscisse)';
    for i=1:length(bernsteinInterpolationAscisseVector)
        bernsteinFunctionValuesVector(i) =
            bernsteinRealFunction(...
             bernsteinInterpolationAscisseVector(i));
    end

    # setting up the parameter to drive the cubic splain
        engine.
    splainSchemeMatrixToFactorStrategyName = '
        normalSplainScheme_BuildMatrixToFactor';
    splainSchemeMisStrategyName = '
        normalSplainScheme_BuildMisVector';
    diffDiviseStrategyName = '
        internal_naturalBuildThreeDifferenzeDiviseVector';

    # go!
    [hVector, varPhiVector, xiVector, lVector, uVector,
        lowerBiadiagonalMatrix, ...
         upperBiadiagonalMatrix, diffDiviseVector, mis,
            bernsteinNormalInterpolatedValues] = ...
         cubicSplainEngine(
            bernsteinInterpolationAscisseVector,
            bernsteinFunctionValuesVector, ...
         splainSchemeMatrixToFactorStrategyName,
            splainSchemeMisStrategyName,...
```

```
        diffDiviseStrategyName,
            bernsteinEvaluationInterval);

    # setting up the parameter to drive the cubic splain
        engine.
    splainSchemeMatrixToFactorStrategyName = '
        notAKnotSplainScheme_BuildMatrixToFactor';
    splainSchemeMisStrategyName = '
        notAKnotSplainScheme_BuildMisVector';
    diffDiviseStrategyName = '
        internal_notAKnotBuildThreeDifferenzeDiviseVector';

    [hVector, varPhiVector, xiVector, lVector, uVector,
        lowerBiadiagonalMatrix, ...
        upperBiadiagonalMatrix, diffDiviseVector, mis,
            bernsteinNotAKnotInterpolatedValues] = ...
        cubicSplainEngine(
            bernsteinInterpolationAscisseVector,
            bernsteinFunctionValuesVector, ...
        splainSchemeMatrixToFactorStrategyName,
            splainSchemeMisStrategyName,...
        diffDiviseStrategyName,
            bernsteinEvaluationInterval);

endfunction

endfunction

function [y] = bernsteinRealFunction(x)
    y = abs(x);
endfunction
```

6.9.23 *Exercise 4.19 - Bernstein interpolation Plotter*

```
function [] = exercise419bernsteinPlotter()

    evaluateSetDimension = 10^(4);

    [bernsteinEvaluationInterval,
        bernsteinEvaluatedFunctionValuesVector,...
```

```
    bernsteinInterpolationAscisseVector,
       bernsteinFunctionValuesVector,...
    bernsteinNormalInterpolatedValues,
       bernsteinNotAKnotInterpolatedValues] =...
        exercise419bernstein(evaluateSetDimension, 10);

    plot(bernsteinEvaluationInterval,
       bernsteinEvaluatedFunctionValuesVector, "b", ...
        bernsteinEvaluationInterval,
           bernsteinNormalInterpolatedValues, "r", ...
        bernsteinEvaluationInterval,
           bernsteinNotAKnotInterpolatedValues, "g", ...
        bernsteinInterpolationAscisseVector,
           bernsteinFunctionValuesVector, "+");
    grid;
    print 'exercise419-bernsteinInterpolationPlotOutput.
       tex' '-dTex' '-S800, 600';
endfunction
```

6.9.24 *Exercise 4.19 on textbook*

```
function [] = exercise419()

    evaluateSetDimension = 10^(4);

    # build the error vector
    errorVector = [10:10:50];
    errorVectorDimension = length(errorVector);

    normalRungeErrorVector = ones(errorVectorDimension,1);
    notAKnotRungeErrorVector = ones(errorVectorDimension
       ,1);

    normalBernsteinErrorVector = ones(errorVectorDimension
       ,1);
    notAKnotBernsteinErrorVector = ones(
       errorVectorDimension,1);

    # start from 1 to have at least one ascisse for
       Bernstein interval domain.
    for n=1:errorVectorDimension
```

```
        [rungeEvaluationInterval,
           rungeEvaluatedFunctionValuesVector,...
        rungeInterpolationAscisseVector,
           rungeFunctionValuesVector,...
        rungeNormalInterpolatedValues,
           rungeNotAKnotInterpolatedValues] = ...
            exercise419runge(evaluateSetDimension,
               errorVector(n));

        [bernsteinEvaluationInterval,
           bernsteinEvaluatedFunctionValuesVector,...
        bernsteinInterpolationAscisseVector,
           bernsteinFunctionValuesVector,...
        bernsteinNormalInterpolatedValues,
           bernsteinNotAKnotInterpolatedValues] =...
            exercise419bernstein(evaluateSetDimension,
               errorVector(n));

        # calculate the error
        normalRungeErrorVector(n) = max(abs(
           rungeEvaluatedFunctionValuesVector ...
             - rungeNormalInterpolatedValues));

        notAKnotRungeErrorVector(n) = max(abs(
           rungeEvaluatedFunctionValuesVector ...
             - rungeNotAKnotInterpolatedValues));

        normalBernsteinErrorVector(n) = max(abs(
           bernsteinEvaluatedFunctionValuesVector ...
             - bernsteinNormalInterpolatedValues));

        notAKnotBernsteinErrorVector(n) = max(abs(
           bernsteinEvaluatedFunctionValuesVector ...
             - bernsteinNotAKnotInterpolatedValues));

    end

    semilogy(errorVector, normalRungeErrorVector, "b", ...
        errorVector, notAKnotRungeErrorVector, "r");
    grid;
    print 'exercise419-rungeErrorsPlotOutput.tex' '-dTex'
       '-S800,600';
```

```
    semilogy(errorVector, normalBernsteinErrorVector, "b",
        ...
49      errorVector, notAKnotBernsteinErrorVector, "r");
    grid;
    print 'exercise419-bernsteinErrorsPlotOutput.tex' '-
       dTex' '-S800, 600';

endfunction
```

6.9.25 *Exercise 4.21 on textbook*

```
function [] = exercise421solver ()
2   gamma = 2.5;
  x = linspace(0, 10, 101)';
  A = [x x.^0 ];

  y = evaluatingFunction(x, gamma);
7   [houseHolderVectors, Rhat, R, Q, g1, g2, unknowns,
      residue] = QRmethod(A,y);
  unknowns
  minimalSquaresEvaluation = evaluatingMinimalSquaresLine(
     x, unknowns(1), unknowns(2));

  plot(x, y, "r", x, minimalSquaresEvaluation, "b");
12  print 'exercise421-PlotOutput.tex' '-dTex' '-S800, 600';

endfunction

function [ ret ] = evaluatingFunction (x, gamma)
17 ret = 10*x + 5 + (sin(x*pi))*gamma;
endfunction

function [ ret ] = evaluatingMinimalSquaresLine (x, alpha,
    beta)
ret = alpha*x + beta;
22 endfunction
```

6.9.26 *Exercise 4.22 on textbook*

È interessante in questo esercizio la potenza di riportarci ad un modello polinomiale, in quanto questo ci permette di costruire la matrice A di tipo *Vander-Monde* e la sua costruzione (molto semplice come si vede nel codice) dipende solo dalle ascisse relative ai valori sperimentali. Questo ci permette quindi da astrarre dalla forma (e dai segni) del modello polinomiale, in quanto di questo ci interessa solo un eventuale cambio di variabile eventualmente sulle ascisse (non il caso di questo esercizio) sia sulle ordinate (come nel caso di questo esercizio):

```
function [] = exercise422solver ()

  x = [0:1:10]';
  A = [x x.^0 ];

  # sperimental data
  y = [5.22 4.00 4.28 3.89 3.53 3.12 2.73 2.70 2.20 2.08
     1.94]';

  # in quanto ho effettuato il cambio di variabile \rho
  y = log(y);

  # applico la fattorizzazione QR
  [houseHolderVectors, Rhat, R, Q, g1, g2, unknowns,
     residue] = QRmethod(A,y);
  # mostro il valore dei coefficienti incogniti (indice
     piu' basso corrisponde al
  # coefficiente del termine di grado piu' alto)
  unknowns

  # costruisco una partizione fissa per poter costruire il
      plot
  ascisseForPlot = linspace(-1, 11, 200)';
  minimalSquaresEvaluation = evaluatingMinimalSquaresLine(
     ascisseForPlot,...
    unknowns(1), unknowns(2));

  # non c'e' bisogno di fare la singola operazione di exp
     sui due termini della
  # retta ai minimi quadrati, in quanto per le prop dell'
     esponenziale
  # exp(a+b) = exp(a)exp(b), quindi eseguendo su tutto il
```

```
    valore calcolato dovremo
  # risolvere il problema
  minimalSquaresEvaluation = exp(minimalSquaresEvaluation)
    ;

  #residuePoints = residue + exp(
    evaluatingMinimalSquaresLine(x,...
  #  unknowns(1), unknowns(2)));

  # plot time!
  plot(x, exp(y), "r+", ...
    #x, residuePoints , "g+",...
    ascisseForPlot, minimalSquaresEvaluation, "b");
  print 'exercise422-PlotOutput.tex' '-dTex' '-S800,600';

endfunction

function [ ret ] = evaluatingMinimalSquaresLine (x, alpha,
    beta)
ret = alpha*x + beta;
endfunction
```

6.10 FORMULE DI QUADRATURA

6.10.1 *Exercise 5.4 on textbook - Trapezi composita*

```
function [points, evaluatedFunctionValues, errore,
   integralValue] = ...
    trapeziComposita(fName, derSecondaName, intervalQty, a
      , b)

    # build the vector that keeps track about the point in
       which the function
    # will be evaluated
    points = linspace(a, b, intervalQty + 1);

    # setting up the vector for hosting the evaluation of
      f in each point
    # in points
    evaluatedFunctionValues = points;
```

```
    # compute a summation of the internal values, which
       are doubly counted
    valueSummation = 0;

    pointsLength = length(points);
    for i = 2:(pointsLength-1)

        # evaluate in each point
        currentEvaluation = feval(fName, points(i));
        evaluatedFunctionValues(i) = currentEvaluation;

        # add to the summation: observe that here we
           perform only one addition
        valueSummation = valueSummation +
           currentEvaluation;
    end

    # doubling the summation performed till now, instead
       of adding two copies
    # inside the for loop
    valueSummation = valueSummation * 2;

    # evaluating in the initial and final points too
    f0 = feval(fName, points(1));
    evaluatedFunctionValues(1) = f0;

    fn = feval(fName, points(pointsLength));
    evaluatedFunctionValues(pointsLength) = fn;

    valueSummation = valueSummation + f0 + fn;

    # compute the integral value
    integralValue = ((b-a)/(2*intervalQty)) *
       valueSummation;

    # compute the error
    somePointForErrorEvaluation = 1;
    errore = -(intervalQty/12)*feval(derSecondaName,
       somePointForErrorEvaluation) *...
        ((b-a)/intervalQty)^3;

endfunction
```

In questa implementazione si fissa $\xi = 1$ in quanto permette di annullare la metà dei termini della derivata seconda.

6.10.2 *Exercise 5.5 on textbook - Simpson composita*

```
function [points, evaluatedFunctionValues, errore,
    integralValue] = ...
    simpsonComposita(fName, derQuartaName, intervalQty, a,
        b)

    # build the vector that keeps track about the point in
        which the function
    # will be evaluated
    points = linspace(a, b, intervalQty + 1);

    # setting up the vector for hosting the evaluation of
       f in each point
    # in points
    evaluatedFunctionValues = points;

    # compute a summation of the internal values, which
       are doubly counted
    oddValueSummation = 0;
    evenValueSummation = 0;

    pointsLength = length(points);
    for i = 2:(intervalQty/2) # Do is right use
       intervalQty? Use pointsLength instead?
        # evaluate in each point
        # here instead of -1 we use -2 because Octave's
           vectors are 1-based
        currentEvaluation = feval(fName, points(2*i-2));

        # add to the summation: observe that here we
           perform only one addition
        oddValueSummation = oddValueSummation +
           currentEvaluation;
    end

    for i = 1:(intervalQty/2) # Do is right use
       intervalQty? Use pointsLength instead?
        # evaluate in each point
```

```
        # here instead of 0 we use -1 because Octave's
            vectors are 1-based
        currentEvaluation = feval(fName, points(2*i - 1));

        # add to the summation: observe that here we
            perform only one addition
        evenValueSummation = evenValueSummation +
            currentEvaluation;
    end

    for i = 1:pointsLength
        # evaluate in each point
        evaluatedFunctionValues(i) = feval(fName, points(i
            ));
    end

    # evaluating in the initial and final points too
    f0 = feval(fName, points(1));
    fn = feval(fName, points(pointsLength));

    valueSummation = 4 * oddValueSummation + 2 *
        evenValueSummation -(f0 + fn);

    # compute the integral value
    integralValue = ((b-a)/(3*intervalQty)) *
        valueSummation;

    # compute the error
    somePointForErrorEvaluation = 1;
    errore = -(intervalQty/180)*feval(derQuartaName,
        somePointForErrorEvaluation) *...
         ((b-a)/intervalQty)^5;

endfunction
```

In questa implementazione si fissa $\xi = 1$ in quanto permette di annullare la metà dei termini della derivata quarta.

6.10.3 *Funzione (5.17)*

```
function [value] = function517(ascissa)
  value = -2*ascissa^(-3)*cos(ascissa^(-2));
```

```
endfunction
```

6.10.4 *Derivata Seconda della funzione (5.17)*

```
function [value] = secondDer517(ascissa)
  value = -24*ascissa^(-5)*cos(ascissa^(-2)) + ...
    36*ascissa^(-7)*sin(ascissa^(-2)) + ...
    8*ascissa^(-9)*cos(ascissa^(-2));
endfunction
```

6.10.5 *Derivata Quarta della funzione (5.17)*

```
function [value] = fourthDer517(ascissa)
  value = -120*6*ascissa^(-7)*cos(ascissa^(-2)) + ...
    240*ascissa^(-9)*sin(ascissa^(-2)) + ...
    300*8*ascissa^(-9)*sin(ascissa^(-2)) + ...
    600*ascissa^(-11)*cos(ascissa^(-2)) + ...
    1440*ascissa^(-11)*cos(ascissa^(-2)) + ...
    -288*ascissa^(-13)*sin(ascissa^(-2)) + ...
    -16*12*ascissa^(-13)*sin(ascissa^(-2)) + ...
    -32*ascissa^(-15)*cos(ascissa^(-2));
endfunction
```

6.10.6 *Plotter della funzione (5.17)*

```
## this function allow the generation of the plot for the
   function 5.17 on textbook.
function [] = analizer517()
  ## creation of the range where we compute the function
     and it's derivates
  points = linspace(0.5, 100, 10001);

  ## initialization of the vectors that host the computed
     values.
  functionValues = points;
  secondDerValues = points;
  fourthDerValues = points;

  for i=1:length(points)
```

```
    currentPoint = points(i);
    functionValues(i) = function517(currentPoint);
    secondDerValues(i) = secondDer517(currentPoint);
    fourthDerValues(i) = fourthDer517(currentPoint);
  end

  ## now we can plot the results
  semilogx(
    points, functionValues, "b");
  grid;
  print 'analysisFunction517-PlotOutput.tex' '-dTex' '-
     S800, 600';

  ## now we can plot the results
  loglog(
    points, functionValues, "b", ...
    points, secondDerValues, "r", ...
    points, fourthDerValues, "g");
  grid;
  print 'analysisFunction517WithDers-PlotOutput.tex' '-
     dTex' '-S800, 600';


endfunction
```

6.10.7 *Solver exercise 5.9 on textbook*

```
function [trapeziCompositeResultTable,
   simpsonCompositeResultTable] = ...
  exercise59solver()

  b = 100;
  a = .5;

  ## the first row is only for initializing the result
     table
  trapeziCompositeResultTable = [-1 -1 -1];
  simpsonCompositeResultTable = [-1 -1 -1];
  for intervalQty=1000:1000:10000
    ######################### for trapezi method
       ########################
```

```
    [points, evaluatedFunctionValues, errore,
       integralValue] = ...
      trapeziComposita('function517', 'secondDer517',
         intervalQty, a, b);

    trapeziCompositeResultTable = [
       trapeziCompositeResultTable; ...
      intervalQty integralValue errore];

    ########################## for simpson method
       ##########################
    [points, evaluatedFunctionValues, errore,
       integralValue] = ...
      simpsonComposita('function517', 'fourthDer517',
         intervalQty, a, b);

    simpsonCompositeResultTable = [
       simpsonCompositeResultTable; ...
      intervalQty integralValue errore];
  end

  ########################## cleaning the results
     ##########################
  trapeziCompositeResultTable =
     trapeziCompositeResultTable(2:...
    rows(trapeziCompositeResultTable), :);

  simpsonCompositeResultTable =
     simpsonCompositeResultTable(2:...
    rows(simpsonCompositeResultTable), :);
endfunction
```

6.10.8 *Adaptive Trapezi*

```
function [i2, points, err] = adaptiveTrapezi(a, b,
   functionName,...
  tol, fa, fb)

  ## to enable recursion
  global points;
```

```
  ## compute the function value on the limits
  if nargin < 6
    points = [a, b];
    fa = feval(functionName, a);
    fb = feval(functionName, b);
  end

  h = b - a;
  newPoint = (a + b)/2;

  ## add the new point to the points collection
  points = [points, newPoint];

  ## evaluate in new computed point
  newPointEvaluation = feval(functionName, newPoint);

  ## compute without subintervals
  i1 = .5 * h * (fa + fb);

  ## compute with intervals
  i2 = .5 * (i1 + h * newPointEvaluation);

  ## according to (5.18)
  err = abs(i2 - i1)/3;

  if err > tol
    [i2left, points, errLeft] = adaptiveTrapezi(a,
       newPoint, functionName, ...
      tol/2, fa, newPointEvaluation);

    [i2right, points, errRight] = adaptiveTrapezi(newPoint
       , b, functionName, ...
      tol/2, newPointEvaluation, fb);

    i2 = i2left + i2right;

    err = errLeft + errRight;
  end

  ## se sono alla prima chiamata allora posso riordinare i
      punti
  if nargin < 6
    points = sort(points);
```

```
  end

48

endfunction
```

6.10.9 *Adaptive Simpson*

Per la seguente implementazione ho cercato di riusare il valore della variabile i1 per il calcolo della variabile i2. Spiego qua il motivo per cui si deve sottrarre il termine -(2/12)*h*fmiddle:

$$i_1 = \frac{1}{6}h(f_a + f_b) + \frac{4}{6}hf_{middle}$$
$$i_2 = \frac{4}{12}h(f_{left} + f_{right}) + \frac{2}{12}hf_{middle} + \frac{1}{12}h(f_a + f_b) = \frac{4}{12}h(f_{left} + f_{right}) + \frac{1}{2}i_1 + c$$

con $\frac{2}{12}hf_{middle} + \frac{1}{12}h(f_a + f_b) = \frac{1}{2}i_1 + c \Leftrightarrow c = -\frac{2}{12}hf_{middle}$

```
1  function [i2, points, err] = adaptiveSimpson(a, b,
     functionName,...
   tol, fa, fb, middle, fmiddle)

   ## to enable recursion
   global points;
6
   ## compute the function value on the limits
   if nargin < 8
     middle = (a + b)/2;
     points = [a, b, middle];
11   fa = feval(functionName, a);
     fb = feval(functionName, b);
     fmiddle = feval(functionName, middle);
   end

16 h = b - a;
   newPointLeft = (a + middle)/2;
   newPointRight = (middle + b)/2;

   ## add the new point to the points collection
21 points = [points, newPointLeft, newPointRight];

   ## evaluate in new computed point
   newPointLeftEvaluation = feval(functionName,
      newPointLeft);
```

```
  newPointRightEvaluation = feval(functionName,
     newPointRight);

## compute without subintervals
i1 = (1/6) * h * (fa + 4 * fmiddle + fb);

## multiply by .5 because the terms with even index are
   already managed in i1
## so to have (1/12) = (1/6)(1/2) required by the
   general equation (5.15).
## for the items with odd index we have to multiply by 4
    as required.
#i2 = (1/12) * h * (4 *(newPointLeftEvaluation +
   newPointRightEvaluation) + ...
#  (fa + 2 * fmiddle + fb));

i2 = (1/12) * h * 4 * (newPointLeftEvaluation +
   newPointRightEvaluation) + ...
  (1/2) * i1 - (2/12) * h * fmiddle;

## according to (5.19)
err = abs(i2 - i1)/15;

if err > tol && abs(b-a) > eps && tol > eps
  [i2left, points, errLeft] = adaptiveSimpson(a, middle,
      functionName, ...
    tol/2, fa, fmiddle, newPointLeft,
       newPointLeftEvaluation);

  [i2right, points, errRight] = adaptiveSimpson(middle,
     b, functionName, ...
    tol/2, fmiddle, fb, newPointRight,
       newPointRightEvaluation);

  i2 = i2left + i2right;

  err = errLeft + errRight;
end

## se sono alla prima chiamata allora posso riordinare i
    punti
if nargin < 8
  points = sort(points);
```

```
  end

endfunction
```

6.10.10 *Solver exercise 5.10 on textbook*

```
function [trapeziAdaptiveResultTable,
   simpsonAdaptiveResultTable] = ...
  exercise510solver()

  b = 100;
  a = .5;
  functionName = 'function517';

  ## the first row is only for initializing the result
     table
  trapeziAdaptiveResultTable = [-1 -1 -1];
  simpsonAdaptiveResultTable = [-1 -1 -1];

  ## initialize the data structures for store the
     information about the points
  ## considerated by the method and to be able to plot
     them
  lastTrapeziPoints = [];
  evaluatedLastTrapeziPoints = lastTrapeziPoints;

  lastSimpsonPoints = [];
  evaluatedLastSimpsonPoints = lastSimpsonPoints;

  tols=[10^(-1); 10^(-2); 10^(-3); 10^(-4); 10^(-5)];
  for i = 1:length(tols)
    ########################## for trapezi method
       ##########################
    [i2, points, err] = adaptiveTrapezi(a, b, functionName
       , tols(i));

    trapeziAdaptiveResultTable = [
       trapeziAdaptiveResultTable; ...
      tols(i) err length(points)];
```

```
    ## save the desired computed points to show them in a
       plot selecting at which
    ## tol we want to capture the points
    if i == 3
      lastTrapeziPoints = points;
    end

    ########################## for simpson method
       ##########################
    ## fix: infinite recursion :(
    [i2, points, err] = adaptiveSimpson(a, b, functionName
       , tols(i));

    simpsonAdaptiveResultTable = [
       simpsonAdaptiveResultTable; ...
      tols(i) err length(points)];

    ## save the desired computed points to show them in a
       plot selecting at which
    ## tol we want to capture the points
    if i == 4
      lastSimpsonPoints = points;
    end
  end

  ########################## cleaning the results for
     Trapezi ##################
  trapeziAdaptiveResultTable = trapeziAdaptiveResultTable
     (2:...
    rows(trapeziAdaptiveResultTable), :);

  for i = 1:length(lastTrapeziPoints)
    evaluatedLastTrapeziPoints(i) = feval(functionName,
       lastTrapeziPoints(i));
  end

  ########################## cleaning the results for
     Simpson ##################
  simpsonAdaptiveResultTable = simpsonAdaptiveResultTable
     (2:...
    rows(simpsonAdaptiveResultTable), :);

  for i = 1:length(lastSimpsonPoints)
```

```
    evaluatedLastSimpsonPoints(i) = feval(functionName,
      lastSimpsonPoints(i));
  end

  ## plotting
  semilogx(
    lastTrapeziPoints, evaluatedLastTrapeziPoints, "b+");
  grid;
  print 'exercise510-TrapeziAdaptivePlotOutput.tex' '-dTex
    ' '-S800,600';

  semilogx(
    lastSimpsonPoints, evaluatedLastSimpsonPoints, "r+");
  grid;
  print 'exercise510-SimpsonAdaptivePlotOutput.tex' '-dTex
    ' '-S800,600';
endfunction
```

6.11 HELPS FOR ?? SECTION

Riporto qui alcune righe di help per le funzioni che verranno utilizzate più spesso in questo capitolo:

```
octave:1> help poly
'poly' is a function from the file /usr/share/octave
  /3.2.3/m/polynomial/poly.m

 -- Function File:  poly (A)
     If A is a square N-by-N matrix, 'poly (A)' is the row
      vector of
     the coefficients of 'det (z * eye (N) - a)', the
        characteristic
     polynomial of A.  As an example we can use this to
        find the
     eigenvalues of A as the roots of 'poly (A)'.
          roots(poly(eye(3)))
          => 1.00000 - 0.00000i
          => 1.00000 + 0.00000i
     In real-life examples you should, however, use the '
        eig' function
     for computing eigenvalues.

     If X is a vector, 'poly (X)' is a vector of
        coefficients of the
     polynomial whose roots are the elements of X.  That
        is, of C is a
     polynomial, then the elements of 'D = roots (poly (C)
        )' are
     contained in C.  The vectors C and D are, however,
        not equal due
     to sorting and numerical errors.

     See also: eig, roots
```

```
octave:1> help ones
'ones' is a built-in function
```

```
 -- Built-in Function:  ones (X)
 -- Built-in Function:  ones (N, M)
 -- Built-in Function:  ones (N, M, K, ...)
 -- Built-in Function:  ones (..., CLASS)
     Return a matrix or N-dimensional array whose elements
         are all 1.
     The arguments are handled the same as the arguments
        for 'eye'.

     If you need to create a matrix whose values are all
        the same, you
     should use an expression like

           val_matrix = val * ones (n, m)
```

```
octave:1> help polyval
'polyval' is a function from the file /usr/share/octave
   /3.2.3/m/polynomial/polyval.m

 -- Function File: Y = polyval (P, X)
 -- Function File: Y = polyval (P, X, [], MU)
     Evaluate the polynomial at of the specified values
        for X.  When MU
     is present evaluate the polynomial for (X-MU(1))/MU
        (2).  If X is a
     vector or matrix, the polynomial is evaluated for
        each of the
     elements of X.

 -- Function File: [Y, DY] = polyval (P, X, S)
 -- Function File: [Y, DY] = polyval (P, X, S, MU)
     In addition to evaluating the polynomial, the second
        output
     represents the prediction interval, Y +/- DY, which
        contains at
     least 50% of the future predictions.  To calculate
        the prediction
     interval, the structured variable S, originating form
         'polyfit',
     must be present.
```

```
     See also: polyfit, polyvalm, poly, roots, conv,
        deconv, residue,
     filter, polyderiv, polyinteg
```

```
octave:14> help ceil
'ceil' is a built-in function

 -- Mapping Function:  ceil (X)
     Return the smallest integer not less than X.  This is
         equivalent to
     rounding towards positive infinity.  If X is complex,
         return 'ceil
     (real (X)) + ceil (imag (X)) * I'.
          ceil ([-2.7, 2.7])
            => -2  3

     See also: floor, round, fix
```