Modelli numerici per la simulazione

esercizi e modelli

Carli Samuele

carlisamuele@csspace.net

www.csspace.net

2 febbraio 2023

Indice

1 Verifica proprietà di $\Delta(z_n y_n)$

Valgono:

$$\begin{aligned}
\Delta(z_n y_n) &= z_{n+1}y_{n+1} - z_n y_n \\
&= z_{n+1}y_{n+1} - z_n y_n \pm y_{n+1}z_n = y_{n+1}\Delta z_n + (\Delta y_n) z_n \\
&= z_{n+1}y_{n+1} - z_n y_n \pm y_n z_{n+1} = y_n \Delta z_n + (\Delta y_n) z_{n+1} \\
&= z_{n+1}y_{n+1} - z_n y_n \pm y_n z_{n+1} \pm y_{n+1} z_n \pm z_n y_n = z_n(\Delta y_n) + (\Delta z_n) y_n + (\Delta z_n)(\Delta y_n) \\
\Delta \frac{y_n}{z_n} &= \frac{y_{n+1}}{z_{n+1}} - \frac{y_n}{z_n} = \frac{z_n \Delta y_n - y_n \Delta z_n}{z_{n+1} z_n}
\end{aligned}$$

Per la seconda è necessario che $z_n \neq 0 \, \forall n$.

2 Relazione tra Δ e derivata

Si ha che $\frac{\Delta f_n}{h} = f'(x_n) + O(h)$. Infatti, su $J = \{x_n | x_n = x_0 + nh\}$, assumendo che f sia sufficientemente regolare e sviluppando in serie di Taylor, si ottiene direttamente che:

$$\begin{aligned}
f(x_{n+1}) &= f(x_n + h) = \sum_{i=0}^{\infty} \frac{f^{(n)}(x_n)}{n!} h^n = f(x_n) + f'(x_n)h + O(h^2) \\
\implies & \frac{f(x_{n+1}) - f(x_n)}{h} = f'(x_n) + O(h) = \frac{\Delta f_n}{h}
\end{aligned}$$

Con passaggi simili si ottiene anche che $\frac{\Delta^2 f_n}{h^2} = f''(x_n) + O(h)$. Osserviamo che:

$$\begin{aligned}
f(x_{n+2}) &= f(x_{n+1} + h) = f(x_{n+1}) + f'(x_{n+1})h + f''(x_{n+1})h^2 + O(h^3) \\
f(x_n) &= f(x_{n+1} - h) = f(x_{n+1}) - f'(x_{n+1})h + f''(x_{n+1})h^2 + O(h^3)
\end{aligned}$$

sommando membro a membro si ottiene:

$$\begin{aligned}
f(x_{n+2}) + f(x_n) &= 2f(x_{n+1}) + f''(x_{n+1})h^2 + O(h^3) \\
f(x_{n+2}) - 2f(x_{n+1}) + f(x_n) &= f''(x_{n+1})h^2 + O(h^3) \\
\frac{\Delta^2 f(x_n)}{h^2} &= f''(x_{n+1}) + O(h)
\end{aligned}$$

3 Calcolo di alcune serie notevoli

3.1 Calcolare $\sum_{i=1}^{n} i$

Osserviamo che $i = i^{(1)}$ e che:

$$\begin{aligned}
\Delta i^{(2)} &= (i+1)i - i(i-1) = 2i^{(1)} \\
\sum_{i=1}^{n} \Delta i^{(2)} &= 2 \sum_{i=1}^{n} i^{(1)} \\
(n+1)^{(2)} - 1^{(2)} &= 2 \sum_{i=1}^{n} i^{(1)}
\end{aligned}$$

ovvero:

$$\sum_{i=1}^{n} i^{(1)} = \frac{1}{2} i^{(2)} \Big|_1^{n+1}$$

e quindi:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n} i^{(1)} = \frac{(n+1)^{(2)} - 1^{(2)}}{2} = \frac{(n+1)n}{2}$$

in quanto $1^{(2)} = 1(1-1) = 0$

3.2 Calcolare $\sum_{i=1}^{n} i^2$

Ricordando che, definiti i numeri di Stirling di seconda specie:

$$\begin{aligned} S_i^{n+1} &= S_{i-1}^n + iS_i^n, \quad n = 2, ..., n \\ S_1^n = S_n^n &= 1, \quad n = 1, 2, ... \end{aligned}$$

si può dimostrare per induzione che:

$$x^n = \sum_{i=1}^{n} S_i^n x^{(i)}$$

e quindi:

$$\begin{aligned} \sum_{i=1}^{n} i^2 &= \sum_{i=1}^{n} (S_1^2 x^{(1)} + S_2^2 x^{(2)}) = \sum_{i=1}^{n} x^{(1)} + \sum_{i=1}^{n} x^{(2)} = \frac{1}{2} x^{(2)} \Big|_1^{n+1} + \frac{1}{3} x^{(3)} \Big|_1^{n+1} \\ &= \frac{(n+1)n - 0}{2} + \frac{(n+1)n(n-1) - 0}{3} = \frac{(n+1)n(2n+1)}{6} \end{aligned}$$

3.3 Calcolare $\sum_{i=1}^{n} i^3$

Usando lo stesso metodo dell'esempio precendente:

$$\begin{aligned} \sum_{i=1}^{n} i^3 &= \sum_{i=1}^{n} (S_1^3 x^{(1)} + S_2^3 x^{(2)} + S_3^3 x^{(3)}) = \sum_{i=1}^{n} x^{(1)} + 3 \sum_{i=1}^{n} x^{(2)} + \sum_{i=1}^{n} x^{(3)} \\ &= \frac{1}{2} x^{(2)} \Big|_1^{n+1} + 3 \frac{1}{3} x^{(3)} \Big|_1^{n+1} + \frac{1}{4} x^{(4)} \Big|_1^{n+1} = \frac{2n(n+1) + 4(n-1)n(n+1) + (n-2)(n-1)n(n+1)}{4} \\ &= \frac{(n^2+n)^2}{4} = \frac{n^2(n+1)^2}{4} \end{aligned}$$

3.4 Calcolare la somma dei primi n dispari

$$\sum_{i=1}^{n} (2i-1) = 2 \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = n(n+1) - n = n^2$$

applicando i risultati ricavati in 3.1.

4 Differenze divise

Data una funzione f definita su un dominio discreto $\{x_k = x_0 + kh\}$, $k = 0, ..., N$, si definiscono le differenze divise in maniera ricorsiva come:

$$\begin{aligned} f[x_k] &= f(x_k) \\ f[x_k, ..., x_{k+j}] &= \frac{f[x_{k+1}, ..., x_{k+j}] - f[x_k, ..., x_{k+j-1}]}{x_{k+j} - x_k} \end{aligned}$$

4.1 Relazione con l'operatore Δ

Si ha che:

$$\frac{\Delta^k f(x_n)}{h^k} = k! f[x_n, ..., x_{n+k}] = f^{(k)}(\xi_n), \quad \xi_n \in [x_n, x_{n+k}] \tag{4.3}$$

Dimostriamolo per induzione. Osserviamo che nel caso base $k = 0$:

$$\frac{\Delta^0 f(x_n)}{h^0} = f(x_n) = f^{(0)}(x_n), \quad x_n \in [x_n, x_n]$$

Svolgendo il caso $k = 1$:

$$\frac{\Delta f(x_n)}{h} = \frac{f(x_{n+1}) - f(x_n)}{h} = \frac{f[x_{n+1}] - f[x_n]}{x_{n+1} - x_n} = f[x_n, x_{n+1}] = \frac{f(x_{n+1}) - f(x_n)}{x_{n+1} - x_n} = {}^{\dagger} f^{(1)}(\xi_n)$$

con $\xi_n \in (x_n, x_{n+1})$ per il teorema di Lagrange [†] ed in quanto $x_{n+1} = x_n + h$ per definizione nel dominio discreto su cui stiamo operando.

Nel caso $k = 2$ abbiamo:

$$\begin{aligned}
\frac{\Delta^2 f(x_n)}{h^2} &= \frac{\Delta(f(x_{n+1}) - f(x_n))}{h^2} = \frac{(f(x_{n+2}) - f(x_{n+1})) - (f(x_{n+1}) - f(x_n))}{h^2} \\
&= \frac{\frac{f(x_{n+2}) - f(x_{n+1})}{x_{n+2} - x_{n+1}} - \frac{f(x_{n+1}) - f(x_n)}{x_{n+1} - x_n}}{h} = \frac{f[x_{n+1}, x_{n+2}] - f[x_n, x_{n+1}]}{h} \\
&= 2\frac{f[x_{n+1}, x_{n+2}] - f[x_n, x_{n+1}]}{2h} = 2\frac{f[x_{n+1}, x_{n+2}] - f[x_n, x_{n+1}]}{x_{n+2} - x_n} = 2f[x_n, x_{n+1}, x_{n+2}]
\end{aligned}$$

utilizzando il risultato del caso $k = 1$.
Assumendo vero il caso k-esimo (4.3), nel caso $(k+1)$-esimo si ottiene:

$$\begin{aligned}
\frac{\Delta^{k+1} f(x_n)}{h^{k+1}} &= \frac{\Delta\Delta^k f(x_n)}{h^{k+1}} = \frac{\Delta\frac{\Delta^k f(x_n)}{h^k}}{h} = \frac{\Delta(k! f[x_n, ..., x_{n+k}])}{h} = \frac{k! f[x_{n+1}, ..., x_{n+k+1}] - k! f[x_n, ..., x_{n+k}]}{h} \\
&= \frac{k!(k+1)(f[x_{n+1}, ..., x_{n+k+1}] - f[x_n, ..., x_{n+k}])}{(k+1)h} = \frac{(k+1)!(f[x_{n+1}, ..., x_{n+k+1}] - f[x_n, ..., x_{n+k}])}{(x_{n+k+1} - x_n)} \\
&= (k+1)! f[x_n, ..., x_{n+k+1}]
\end{aligned}$$

ancora tenendo conto che $x_{n+k} - x_n = kh$. C.v.d.

Dimostriamo per induzione anche che:

$$\frac{\Delta^k f(x_n)}{h^k} = f^{(k)}(\xi_n), \quad \xi_n \in [x_n, x_{n+k}]$$

Il caso base $k = 1$ come abbiamo visto prima è direttamente dimostrato dal teorema di Lagrange[†]:

$$\frac{\Delta f(x_n)}{h} = \frac{f(x_{n+1}) - f(x_n)}{h} = \frac{f(x_{n+1}) - f(x_n)}{x_{n+1} - x_n} = f^{(k)}(\xi_n), \quad \xi_n \in [x_n, x_{n+k}]$$

assumendo quindi vero il caso k si ottiene:

$$\frac{\Delta^{k+1} f(x_n)}{h^{k+1}} = \frac{\Delta\Delta^k f(x_n)}{hh^k} = \frac{\Delta f^{(k)}(\xi_n)}{h} = \frac{f^{(k)}(\xi_n + h) - f^{(k)}(\xi_n)}{h} = f^{(k+1)}(\nu)$$

in cui ancora per il teorema di Lagrange, $\nu \in [x_n, x_{n+k+1}]$ in quanto $\xi_n \in [x_n, x_{n+k}]$ e $E\xi_n = \xi_n + h \in [x_{n+1}, x_{n+k+1}]$ ed ovviamente $h = (\xi_n + h) - \xi_n$. C.v.d.
Le due equivalenze in (4.3) quindi valgono entrambe.

[†]detto anche del valor medio o dell'incremento finito. Se $f : [a, b] \to \mathbb{R}$ è una funzione continua nell'intervallo chiuso $[a, b]$ e derivabile nell'intervallo aperto (a, b), allora esiste almeno un punto $c \in (a, b)$ per cui $f'(c) = \frac{f(b) - f(a)}{b - a}$

5 Forma chiusa del determinante di una matrice di Frobenius di ordine k

Dimostriamo che vale:

$$p(\lambda) = det(\lambda I - F_k) = \sum_{i=0}^{k} p_i \lambda^{k-i}, \quad F_k = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & \ddots & 0 & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & 1 \\ -p_k & -p_{k-1} & \dots & \dots & -p_1 \end{pmatrix}, \quad p_0 \equiv 1$$

Chiamiamo $D^{(k)} = \lambda I - F_k$:

$$D^{(k)} = \begin{pmatrix} \lambda & -1 & 0 & \dots & 0 \\ 0 & \lambda & \ddots & 0 & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \lambda & -1 \\ p_k & p_{k-1} & \dots & p_2 & \lambda + p_1 \end{pmatrix}$$

Calcoliamone il determinante usando lo sviluppo di Laplace per colonne:

$$\det(D^{(k)}) = \sum_{i=1}^{k} [d_{ij}(-1)^{i+j} \det(D_{ij}^{(k)})]$$

dove denotiamo con $D_{ij}^{(k)}$ il minore principale di $D^{(k)}$, ovvero quest'ultima a cui sono state tolte la riga i-esima e la colonna k-esima. Applicandolo alla prima colonna, soltanto il primo e l'ultimo termine della sommatoria rimangono (in quanto gli intermedi sono tutti nulli):

$$\det(D^{(k)}) = \sum_{i=1}^{k} [d_{i1}(-1)^{i+1} \det(D_{i1}^{(k)})] = \lambda(-1)^2 \det(D_{11}^{(k)}) + p_k(-1)^{k+1} \det(D_{k1}^{k})$$

Osserviamo che il minore principale $D_{k1}^{(k)}$ ha la forma:

$$D_{k1}^{(k)} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ \lambda & -1 & 0 & 0 \\ 0 & \ddots & \ddots & \\ 0 & 0 & \lambda & -1 \end{pmatrix} \in \mathcal{M}^{(k-1)\times(k-1)}$$

per la quale si ottiene immediatamente $\det(D_{k1}^{(k)}) = (-1)^{k-1}$.
Osserviamo anche che $D_{11}^{(k)} = D^{(k-1)}$, ovvero il minore a cui togliamo la prima riga e la prima colonna è una matrice con la stessa struttura di quella di partenza ma dimensioni ridotte di 1.

Abbiamo quindi:

$$\det(D^{(k)}) = \lambda(-1)^2 \det(D_{11}^{(k)}) + p_k(-1)^{k+1}(-1)^{k-1} = \lambda \det(D^{(k-1)}) + p_k$$

Possiamo quindi applicare gli stessi ragionamenti per $D^{(k-1)}, D^{(k-2)}, ..., D^{(3)}$, e una volta calcolato esplicitamente il determinante di $D^{(2)}$:

$$\det D^{(2)} = \det \begin{pmatrix} \lambda & 1 \\ p_2 & \lambda + p_1 \end{pmatrix} = \lambda(\lambda + p1) + p_2$$

otteniamo l'espressione telescopica:

$$\begin{aligned}\det(D^{(k)}) &= \lambda \det(D^{(k-1)}) + p_k = \lambda(\lambda(...\lambda(\lambda + p1) + p_2) + ...) + p_{k-1}) + p_k \\ &= \lambda^{k+1}(p_0) + \lambda^k p_1 + ... + p_k = \sum_{i=0}^{k} p_i \lambda^{k-i}\end{aligned}$$

ricordando che $p_0 = 1$. c.v.d.

6 Metodo di Horner per valutare un polinomio

Dimostriamo che la soluzione y_n dell'equazione alle differenze:

$$y_i = x y_{i-1} + b_i, \quad i = 1, ..., n \quad y_0 = b_0 \quad x, b_1, ..., b_n \text{noti} \tag{6.1}$$

è in realtà la valutazione nel punto x del polinomio:

$$p(x) = \sum_{i=0}^{n} b_i x^{n-i}$$

Ricordiamo che la soluzione di $y_{n+1} = p_n y_n + g_n$ si ricava portandosi nel caso $z_{n+1} = z_n + q_n$ dividendo entrambi i membri per $P(n+1) = \prod_{i=0}^{n} p_i$:

$$\frac{y_n}{P(n+1)} = \frac{p_n y_n}{P(n+1)} + \frac{g_n}{P_{n+1}} \implies \frac{y_n}{P(n+1)} = \frac{y_n}{P(n)} + \frac{g_n}{P_{n+1}} \implies z_{n+1} = z_n + q_n$$

quindi passando alle sommatorie e infine riesplicitando si ottiene:

$$y_n = y_0 \prod_{i=0}^{n-1} p_i + \sum_{i=0}^{n-1} \left(g_i \prod_{j=i+1}^{n-1} p_j \right)$$

Applicando la soluzione al caso (6.1), $p_n = x \, \forall n$ e $g_i = b_{i+1}$, quindi:

$$y_n = y_0 \prod_{i=0}^{n-1} x + \sum_{i=0}^{n-1} \left(b_{i+1} \prod_{j=i+1}^{n-1} x \right) = y_0 x^n + \sum_{i=0}^{n-1} b_{i+1} x^{n-1-i} = \sum_{i=0}^{n} b_i x^{n-i}$$

in quanto $y_0 = b_0$.
Osserviamo che il calcolo del polinomio $p(x)$ richiederebbe n somme, $n+1$ moltiplicazioni e $n+1$ elevamenti a potenza. Il metodo di Horner invece richiede n passi, ognuno dei quali esegue soltanto una moltiplicazione e un'addizione, per un totale di $2n$ operazioni elementari.

7 Numeri di Fibonacci

La successione di Fibonacci è definita dall'equazione:

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = F_1 = 1$$

che equivale all'equazione omogenea lineare di secondo grado:

$$L y_n = 0 = \sum_{i=0}^{k} p_i y_{n+k-i} = y_{n+2} - y_{n+1} - y_n$$

In quanto i coefficienti sono costanti, possiamo costruire una base per lo spazio delle soluzioni utilizzando le radici del polinomio caratteristico associato. In particolare, la soluzione dell'equazione omogenea è del tipo:

$$y_n = \sum_{i=1}^{k} \alpha_i f_n^{(i)}, \quad f_n^{(i)} = z_i^n, \quad p(z_i) = 0 = z_i^2 - z_i - 1, \quad i = 1, ..., k \tag{7.1}$$

ed in quanto $p(z)$ è di secondo grado, $k = 2$ ovvero ci sono soltanto due radici del polinomio caratteristico ovvero due soluzioni linearmente indipendenti dell'equazione omogenea $Ly_n = 0$.
Le radici sono dunque:

$$z_1 = \frac{1+\sqrt{5}}{2}, \quad z_2 = \frac{1-\sqrt{5}}{2}$$

Troviamo i coefficienti $\boldsymbol{\alpha} = (\alpha_1, \alpha_2)^T$ imponendo le condizioni iniziali $\boldsymbol{y} = (y_0, y_1)^T = (1, 1)$ ovvero risolvendo il sistema $C(0)\boldsymbol{\alpha} = \boldsymbol{y}$:

$$\begin{pmatrix} z_1^0 & z_2^0 \\ z_1^1 & z_2^1 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \implies \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} z_1/\sqrt{5} \\ -z_2/\sqrt{5} \end{pmatrix}$$

Sostituendo in (7.1) [p.7] si ottiene:

$$y_n = \sum_{i=1}^{k} \alpha_i f_n^{(i)} = \frac{z_1^{n+1}}{\sqrt{5}} - \frac{z_2^{n+1}}{\sqrt{5}}$$

Osservazioni

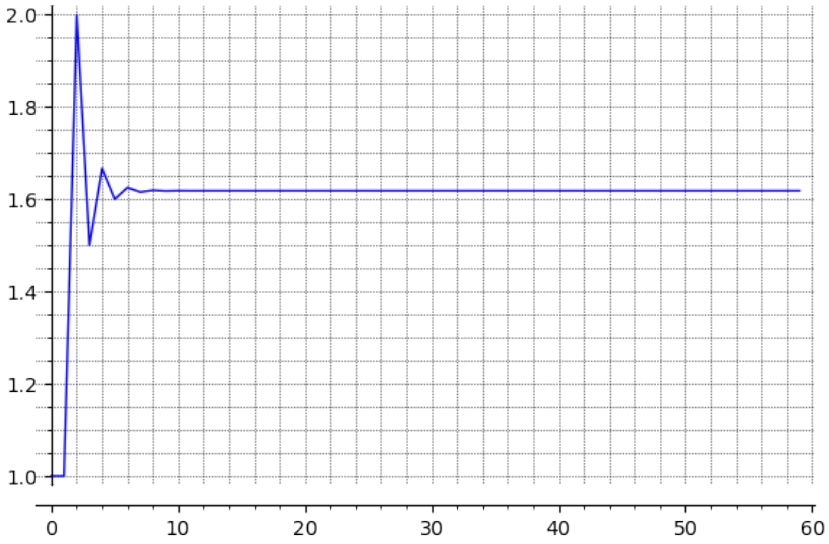
Se $\alpha_1 \neq 0$ possiamo calcolare il limite del rapporto tra due iterazioni successive:

$$\lim_{n\to\infty} \frac{y_{n+1}}{y_n} = \lim_{n\to\infty} \frac{\alpha_1 z_1^{n+2} - \alpha_2 z_2^{n+2}}{\alpha_1 z_1^{n+1} - \alpha_2 z_2^{n+1}} = z_1$$

in quanto $z_2 < 1$.

Verifichiamo che il rapporto tra due iterazioni converge velocemente a $z_1 = \frac{1+\sqrt{5}}{2} \approx 1.61803398874989$

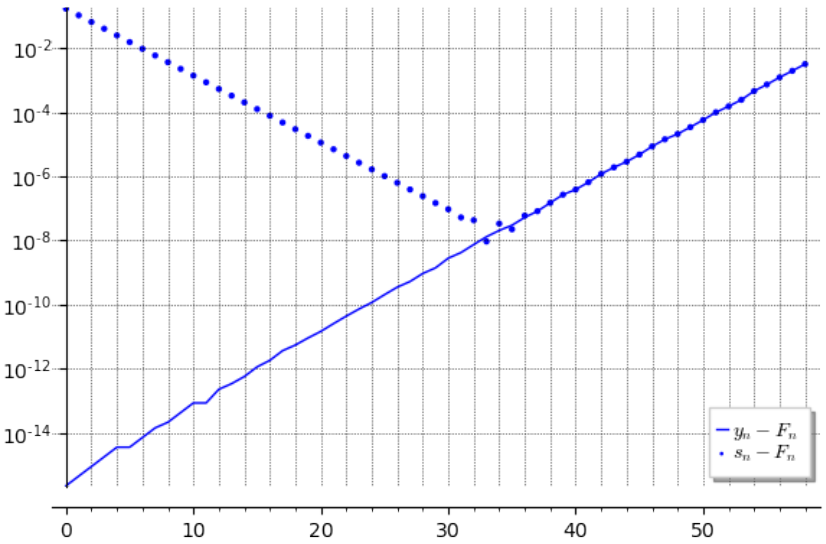
```
L = 60
fib = [(1,1), (1,1)] # [(fib_i, fib_i/fib_{i-1}), ...]
for i in range(2,L):
    f_i = fib[i-1][0] + fib[i-2][0]
    rapporto = (1.0*f_i/fib[i-1][0])
    fib.append((f_i,rapporto))
list_plot([y[1] for y in fib], plotjoined=True, gridlines='minor')

```



Inoltre, in quanto $z_2 < 1$, ci si aspetta che $F_n \approx \frac{z_1^{n+1}}{\sqrt{5}}$ per $n >> 0$.

Chiamando $y_n = \left(\frac{z_1^{n+1}}{\sqrt{5}} - \frac{z_2^{n+1}}{\sqrt{5}}\right)$ e $s_n = \frac{z_1^{n+1}}{\sqrt{5}}$, grafichiamo gli errori $|y_n - F_n|$ e $|s_n - F_n|$:

```
z_1 = (1+sqrt(5))/2.
z_2 = (1-sqrt(5))/2.
y = lambda n: (z_1**(n+1)-z_2**(n+1))/sqrt(5)
s = lambda n: z_1**(n+1)/sqrt(5)

fibdiff = [ abs( ( y(n) - fib[n][0] ) ) for n in range(0,L)]
fibdiff_approx = [ abs( ( s(n) - fib[n][0] ) ) for n in range(0,L)]
P = list_plot(fibdiff[1:], plotjoined=True, gridlines='minor', scale='semilogy', legend_label='$y_n-F_n$')
P += list_plot(fibdiff_approx[1:], plotjoined=False, gridlines='minor', scale='semilogy',
↪  legend_label='$s_n-F_n$')
P.show()
```



Osserviamo che l'errore della soluzione approssimata s_n si riduce con il ridursi di $|z_2^{n+1}|$ ad ogni passo fino a che $|z_2^{n+1}|$ non si annulla al raggiungimento della precisione di macchina, cioè quando $|\frac{z_2^n}{\sqrt{5}}| \approx |u|$:

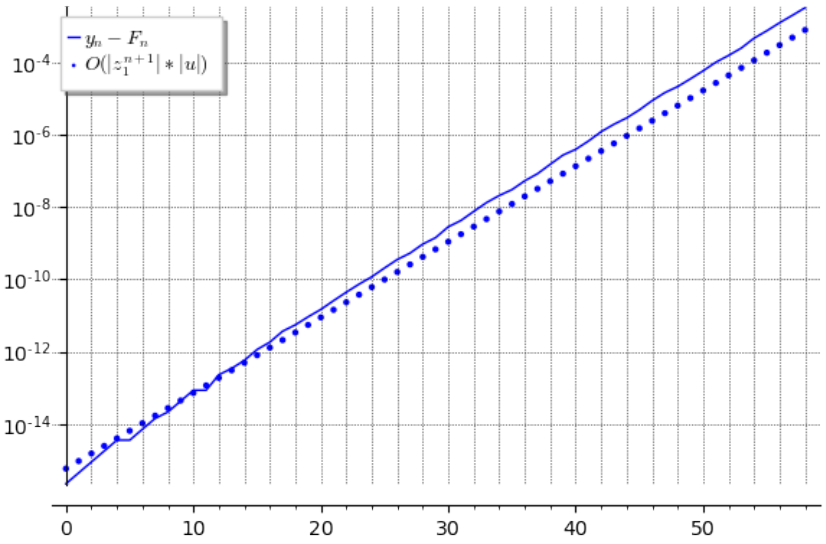
$$\frac{|z_2^n|}{\sqrt{5}} \approx |u| \Rightarrow n \approx \frac{log(|u|)\sqrt{5}}{log(|z_2|)} \approx 38$$

Da quel punto, il comportamento delle due soluzioni coincide.

L'errore assoluto cresce comunque infinitamente a causa dell'approssimazione delle rappresentazioni di $z_1, z_2, \sqrt{5}$, e ci aspettiamo che sia $O(|z_1^{n+1}| * |u|)$.

Approssimando $e_n = O(|z_1^{n+1}| * |u|)$ con $|z_1^{n+1}|2^{-52}$ si ottiene:

```
fibdiff = [ abs( ( y(n) - fib[n][0] ) ) for n in range(0,L)]
Oz_1u = [ abs(z_1**(n+1) * 2**-52) for n in range(0,L)]
P2 = list_plot(fibdiff[1:], plotjoined=True, gridlines='minor', scale='semilogy', legend_label='$y_n-F_n$')
P2 += list_plot(Oz_1u[1:], plotjoined=False, gridlines='minor', scale='semilogy',
↪  legend_label='$O(|z_1^{n+1}| * |u|)$')
P2.show()
```



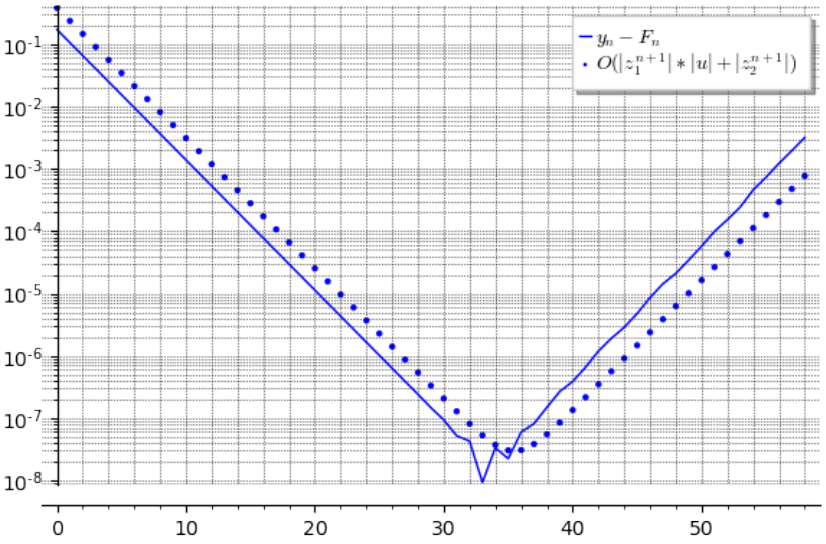
ovvero l'errore si comporta esattamente come ci si aspetterebbe.

L'errore della soluzione approssimata invece ha un termine in più che tende a zero con le iterazione, in particolare ci si aspetta:

$$e_n = O(|z_1^{n+1}| * |u| + |z_2|^{n+1}) \approx |z_1^{n+1}|2^{-52} + |z_2^{n+1}|$$

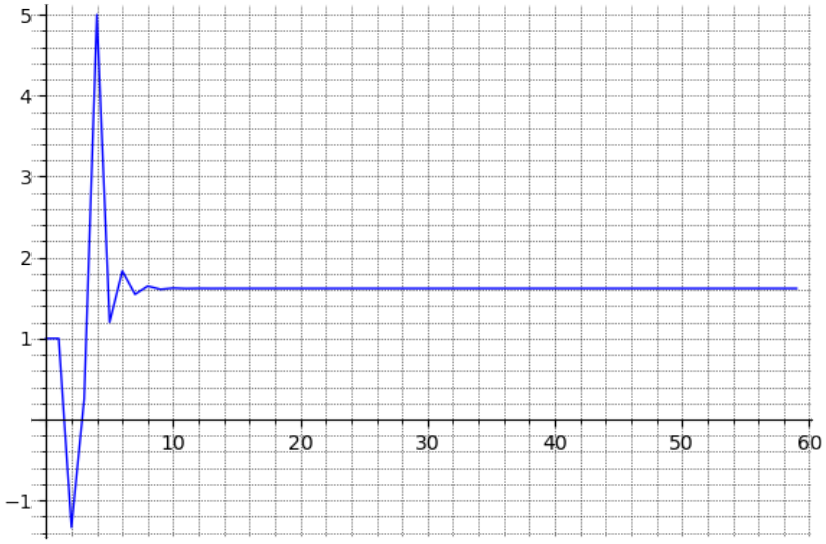
Grafichiamo:

```
fibdiff = [ abs( ( y(n) - fib[n][0] ) ) for n in range(0,L)]
Oz_1z_2u = [ abs(z_1**(n+1) * 2**-52 ) + abs(z_2**(n+1)) for n in range(0,L)]
P3 = list_plot(fibdiff_approx[1:], plotjoined=True, gridlines='minor', scale='semilogy',
↪  legend_label='$y_n-F_n$')
P3 += list_plot(Oz_1z_2u[1:], plotjoined=False, gridlines='minor', scale='semilogy',
↪  legend_label='$O(|z_1^{n+1}| * |u| + |z_2^{n+1}|)$')
P3.show()
```

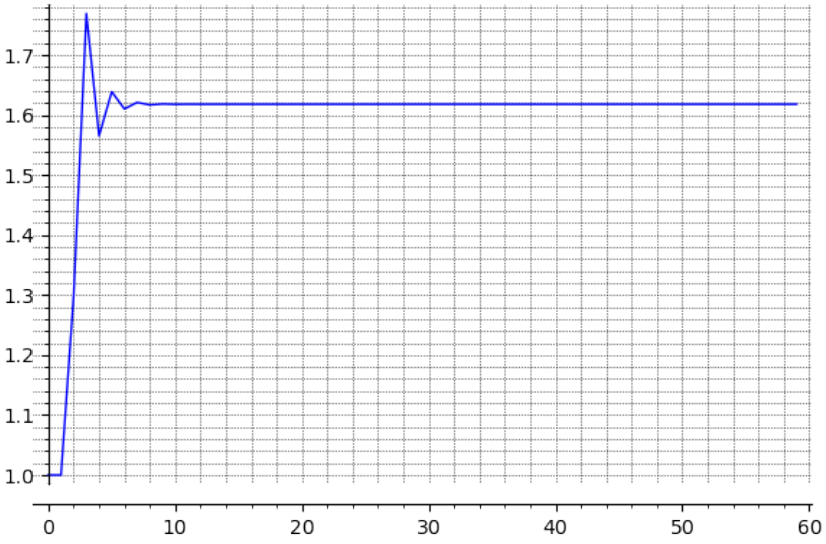


Modificando le condizioni iniziali l'andamento del rapporto cambia ma la convergenza verso z_1 rimane se $\alpha_1 \neq 0$:

```
fib = [(-0.7,1), (0.3,1)]
for i in range(2,L):
    f_i = fib[i-1][0] + fib[i-2][0]
    rapporto = (1.0*f_i/fib[i-1][0])
    fib.append((f_i,rapporto))
list_plot([y[1] for y in fib], plotjoined=True, gridlines='minor')
```




```
fib = [(3,1), (10,1)]
for i in range(2,L):
    f_i = fib[i-1][0] + fib[i-2][0]
    rapporto = (1.0*f_i/fib[i-1][0])
    fib.append((f_i,rapporto))
list_plot([y[1] for y in fib], plotjoined=True, gridlines='minor')
```



8 Il bit fatale

Calcoliamo la soluzione di

$$y_n = y_{n-1} + y_{n-2}, \quad y_0 = 1, y_1 = \frac{1-\sqrt{5}}{2}$$

Essendo la stessa equazione dei numeri di fibonacci (7) con condizioni iniziale diverse, ci aspettiamo che la soluzione sia:

$$y_n = \sum_{i=1}^{2} \alpha_i z_i^n, \quad z_1 = \frac{1+\sqrt{5}}{2}, \quad z_2 = \frac{1-\sqrt{5}}{2}$$

e imponendo $y_0 = 1,\ y_1 = z_2$ si ottiene:

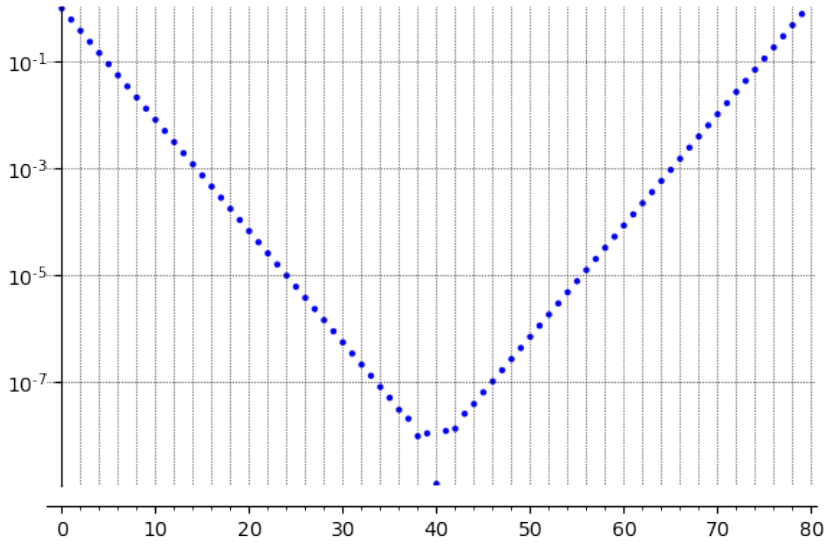
$$\begin{pmatrix} z_1^0 & z_2^0 \\ z_1^1 & z_2^1 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} 1 \\ z_2 \end{pmatrix} \implies \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

ovvero $y_n = z_2^n$; in quanto $z_2 < 1$ ci aspettiamo che al limite:

$$\lim_{i\to\infty} y_n = \lim_{i\to\infty} z_2^n = 0$$

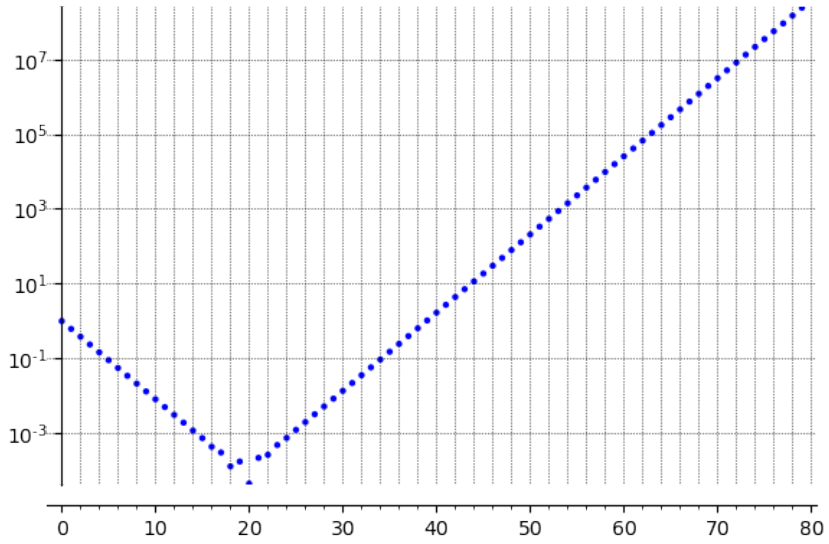
Usando doppia precisione (float64) si ottiene:

```
from numpy import float64
y = [1, float64((1-sqrt(5))/2)]
for i in range(2,80):
    y.append(float64(y[i-1]+y[i-2]))
list_plot([abs(x) for x in y], scale='semilogy', gridlines='minor')
```



Ed in precisione singola invece (float32):

```
from numpy import float32
y = [1, float32((1-sqrt(5))/2)]
for i in range(2,80):
    y.append(float32(y[i-1]+y[i-2]))
list_plot([abs(x) for x in y], scale='semilogy', gridlines='minor')
```



Questo avviene a causa degli errori di macchina nell'approssimazione di z_2. Consideriamo l'equazione dell'errore:

$$Le_n = 0, \quad e_n = y_n - \bar{y}_n$$

in cui y_n è la soluzione calcolata a partire da un'approssimazione di z_2 mentre invece $\bar{y}_n$ è la soluzione esatta.
In generale, avremo $e_0 = 0$ ma $e_1 \approx u$, indicando con u la precisione di macchina. La soluzione dell'equazione è quindi come prima:

$$e_n = \sum_{i=1}^{2} \alpha_i z_i^n, \quad z_1 = \frac{1+\sqrt{5}}{2}, \quad z_2 = \frac{1-\sqrt{5}}{2}$$

ma i coefficienti α_i valgono:

$$\begin{pmatrix} z_1^0 & z_2^0 \\ z_1^1 & z_2^1 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} 0 \\ u \end{pmatrix} \implies \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \approx \begin{pmatrix} u \\ u \end{pmatrix}$$

ovvero si ha:

$$e_n \approx u z_1^n + u z_2^n$$

ed in quanto $z_1 > 1$, ci si aspetta che l'errore diverga col tendere di n all'infinito.
Si ha quindi che la soluzione calcolata in realtà vale:

$$y_n = \bar{y_n} + e_n = z_2^n + u z_1^n + u z_2^n = (1+u) z_2^n + u z_1^n \approx z_2^n + u z_1^n$$

ci aspettiamo quindi che l'errore inizi a divergere non appena $u|z_1|^n \approx |z_2|^n$, ovvero:

$$n = \frac{-\log(|u|)}{\log(|z_1|) - \log(|z_2|)} \quad n_{64} = \frac{-log(2^{-52})}{\log(|z_1|) - \log(|z_2|)} \approx 37, \quad n_{32} = \frac{-log(2^{-23})}{\log(|z_1|) - \log(|z_2|)} \approx 17$$

in aritmetica a doppia e singola precisione che hanno rispettivamente 52 e 23 bit di mantissa nella rappresentazione binaria. In effetti, nei grafici si rispecchia esattamente questo comportamento.

9 Ordine di convergenza del metodo delle secanti

Sia $f(x) : \mathbb{R} \to \mathbb{R}$ sufficientemente regolare, e sia $\bar{x}$ tale che $f(\bar{x}) = 0$ una radice semplice di f.
Date due approssimazioni di $\bar{x}$: x_0, x_1, il metodo delle secanti segue l'equazione:

$$x_{n+1} = x_n - \frac{f(x_n)}{f[x_{n-1}, x_n]} = x_n - f(x_n)\frac{(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

Definiamo quindi l'errore al passo n-esimo:

$$e_n = x_n - \bar{x}$$

e ricordiamo la definizione di ordine di convergenza di un metodo:

Definizione DEF9.1: Ordine di convergenza
Un metodo ha ordine di convergenza p se vale:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^p} = c < \infty$$

in cui c è detta *costante asintotica dell'errore*. Se $p = 1$ il metodo ha convergenza lineare.

Calcoliamo l'equazione dell'errore:

$$\begin{aligned}
e_{n+1} &= x_{n+1} - \bar{x} = x_n - \frac{f(x_n)}{f[x_{n-1}, x_n]} - \bar{x} = {}^{\dagger} e_n - \frac{f(x_n) - f(\bar{x})}{f[x_{n-1}, x_n]} \\
&= e_n - \frac{(x_n - \bar{x}) f[\bar{x}, x_n]}{f[x_{n-1}, x_n]} = e_n\left(1 - \frac{f[\bar{x}, x_n]}{f[x_{n-1}, x_n]}\right) = \frac{e_n}{f[x_{n-1}, x_n]}(f[x_{n-1}, x_n] - f[\bar{x}, x_n]) \\
&= \frac{e_n}{f[x_{n-1}, x_n]}(f[x_{n-1}, x_n] - f[x_n, \bar{x}]) = \frac{e_n(x_{n-1} - \bar{x})}{f[x_{n-1}, x_n]} \frac{(f[x_{n-1}, x_n] - f[x_n, \bar{x}])}{(x_{n-1} - \bar{x})} \\
&= \frac{e_n e_{n-1} f[x_{n-1}, x_n, \bar{x}]}{f[x_{n-1}, x_n]}
\end{aligned}$$

[†] $f(\bar{x}) = 0$

Assumendo che il metodo sia convergente, ovvero che:

$$\lim_{n\to\infty} x_n = \bar{x} \iff \lim_{n\to\infty} e_n = 0$$

e che f sia almeno $\mathcal{C}^{(2)}$ in un intorno di $\bar{x}$, possiamo calcolare:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n e_{n-1}|} = \lim_{n\to\infty} \frac{|f[x_{n-1}, x_n, \bar{x}]|}{|f[x_{n-1}, x_n]|} = \frac{|f''(\bar{x})|}{2|f'(\bar{x})|} = M$$

infatti per quanto argomentato sulle differenze divise in 4, $f[x_1, ..., x_k] = f^{(k)}(\xi)/k!$, $\xi \in [x_1, x_k]$, ed in questo caso siamo al limite in cui $x_n \to \bar{x}$ per cui $\xi \in [x_{n-1}, \bar{x}] \to [\bar{x}, \bar{x}]$.
Abbiamo quindi che per n sufficientemente grande valgono:

$$\begin{aligned} |e_{n+1}| &= M|e_n||e_{n-1}| \\ M|e_{n+1}| &= M|e_n|M|e_{n-1}| \\ \log(M|e_{n+1}|) &= log(M|e_n|M|e_{n-1}|) = \log(M|e_n|) + \log(M|e_{n-1}|) \end{aligned}$$

dalla quale, ponendo $y_n = \log(M|e_n|)$ si ottiene l'equazione:

$$y_{n+1} = y_n + y_{n-1}$$

abbiamo quindi la stessa equazione omogenea della sequenza di fibonacci vista in 7, che ha polinomio caratteristico $p(z) = z^2 - z - 1$ e quindi soluzione:

$$y_n = \alpha_1 z_1^n + \alpha_2 z_2^n, \quad z_1 = \frac{1+\sqrt{5}}{2}, \quad z_2 = \frac{1-\sqrt{5}}{2}$$

per la quale abbiamo visto che se $\alpha_1 \neq 0$, per n sufficientemente grandi $\frac{|y_{n+1}|}{|y_n|} \approx z_1 = 1.618...$. Abbiamo quindi:

$$\lim_{n\to\infty} \frac{|y_{n+1}|}{|y_n|} = z_1 = \lim_{n\to\infty} \frac{\log(M|e_{n+1}|)}{\log(M|e_n|)} \implies \log(M|e_{n+1}|) \approx z_1 \log(M|e_n|)$$

e quindi:

$$\begin{aligned} \log(M|e_{n+1}|) &\approx z_1 \log(M|e_n|) \approx \log(M^{z_1}|e_n|^{z_1}) \\ M|e_{n+1}| &\approx M^{z_1}|e_n|^{z_1} \end{aligned}$$

da cui infine:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^{z_1}} \approx M^{z_1 - 1}$$

ed in quanto M è una costante finita visto che abbiamo supposto f sufficientemente regolare, l'ordine del metodo è $p = z_1 = 1.618...$, ovvero il metodo e' superlineare.

Verifichiamo l'ordine di convergenza su qualche esempio. Usiamo un'implementazione ricorsiva del metodo delle secanti:

```
1   import numpy as np
2   import pandas as pd
3
4   def secanti(f, x0, x1, tol=10^-15, yield_args=True):
5       fx0 = f(x0)
6       fx1 = f(x1)
7       x2 = x1 - (fx1 * (x1-x0))/(fx1 - fx0)
8       if yield_args:
9           yield x0
10          yield x1
11      yield x2
12      if abs(x2-x1) > tol:
13          yield from secanti(f, x1, x2, tol, yield_args=False)
```

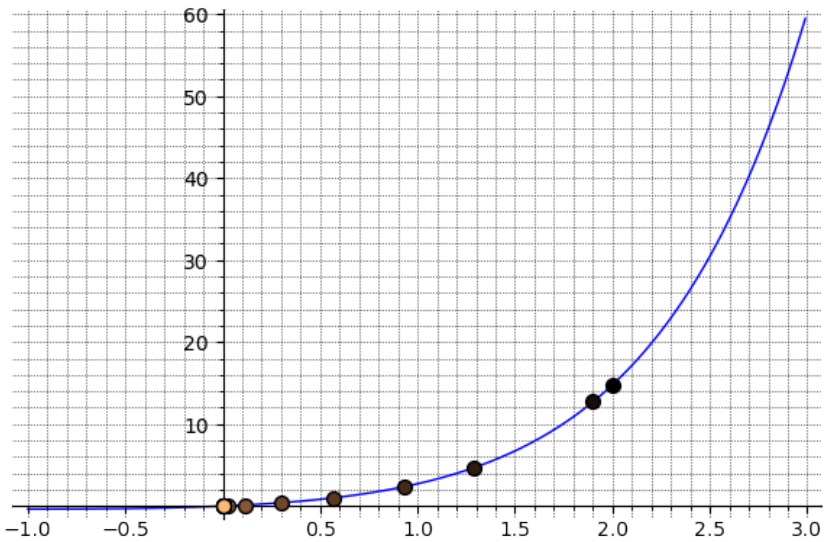
Dato il vettore $\mathbf{x} = (x_1, ..., x_n)^T$ delle approssimazioni successive della radice di $f(x)$ ottenute dal metodo, ed il valore della soluzione esatta $\bar{x}$, calcoliamo il vettore degli errori $\mathbf{e} = (e_0, ..., e_n)^T = |\mathbf{x} - \bar{x}|$.

Applicando il metodo delle secanti a $f(x) = xe^x$, con $x_0 = 2$ e $x_1 = 1.9$ si ottiene:

```
f = lambda x: float(x*exp(x))
x = list(secanti(f, 2.0, 1.9))
pd.DataFrame(x)
```

	0
0	2.00000000000000
1	1.90000000000000
2	1.28777187646923
3	0.932127183796176
4	0.566086060505791
5	0.299767090935616
6	0.117938854294964
7	0.0291795932820642
8	0.00320364703781081
9	0.0000919865342931649
10	2.94207350908546e-7
11	2.70618659102019e-11
12	7.96179870689832e-18
13	2.15460716226400e-28

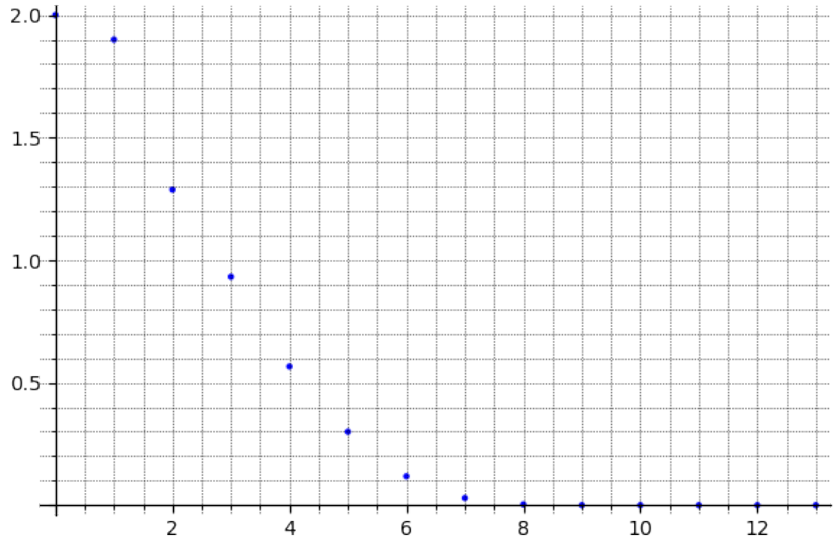
```
def plot_secanti(func, points):
    min_x = min(points)
    max_x = max(points)
    r = (max_x - min_x) / 2.
    base = np.arange(min_x - r, max_x + r, 0.01)
    f_points = [func(x) for x in base]
    p = list_plot( list(zip(*[base, f_points])), plotjoined=True, gridlines='minor')
    for i,x in enumerate(points):
        p += scatter_plot([(x,func(x))], marker='o', facecolor=list(colormaps.copper(i/(len(points)))[0:3]))
    p.show()
plot_secanti(f, x)
```



Per $f(x) = xe^x$, la radice è in $x = 0$. Vediamo che in effetti i punti si allineano in maniera quasi esatta dopo poche iterazioni.

```
def plot_error(x, barx):
    x = np.array(x)
    e = abs(x-barx)
    p = list_plot([(i,e[i]) for i in range(len(e))], gridlines='minor')
```

```
    p.show()
plot_error(x,0)
```

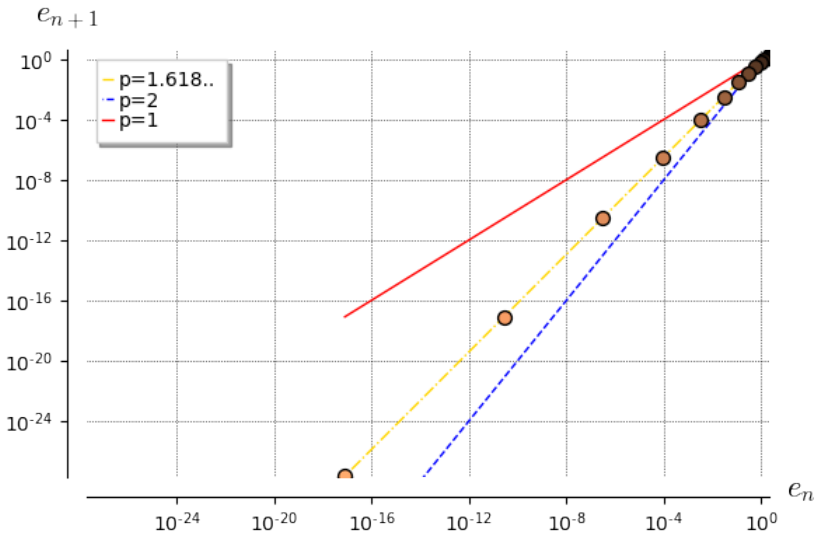


Facciamo il grafico di e_{n+1} rispetto ad e_n (punti di colore via via più chiaro al crescere di n), e contemporaneamente, il grafico di $e_n^{1.618}$ rispetto a e_n (retta gialla).

Se l'ordine di convergenza del metodo è veramente $p = z_1 \approx 1.618$, ci aspettiamo che i punti tendano ad allinearsi sulla retta in quanto si dovrebbe avere $e_{n+1} \approx e_n^p$.

Per riferimento, aggiungiamo anche una linea rossa che rappresenta la convergenza lineare ($|e_{n+1}| = |e^n|$) e una blu che delinea la convergenza quadratica ($|e_{n+1}| = |e^n|^2$)

```
def plot_convergence(x, barx):
    x = np.array(x)
    e = abs(x-barx)
    minval = min(x for x in e if x>0)
    # Plot reference lines for golden, linear and quadratic convergence
    p = list_plot(list(zip(*[e[0:-1], e[0:-1]^1.618])), scale='loglog', plotjoined=True, linestyle='-.',
                  color='gold', xmin=minval, ymin=minval, axes_labels=['$e_n$', '$e_{n+1}$'],
                  gridlines='minor', legend_label='p=1.618..')
    p += list_plot(list(zip(*[e[0:-1], e[0:-1]^2])), scale='loglog', plotjoined=True, linestyle='--',
                   color='blue', xmin=minval, ymin=minval, legend_label='p=2')
    p += list_plot(list(zip(*[e[0:-1], e[0:-1]])), scale='loglog', plotjoined=True, linestyle='-',
                   color='red', xmin=minval, ymin=minval, legend_label='p=1')
    # Plot errors
    for i in range(e.size-1):
        p += scatter_plot([(e[i],e[i+1])], marker='o', facecolor=list(colormaps.copper(i/(e.size))[0:3]))

    p.show()

plot_convergence(x, 0)
```

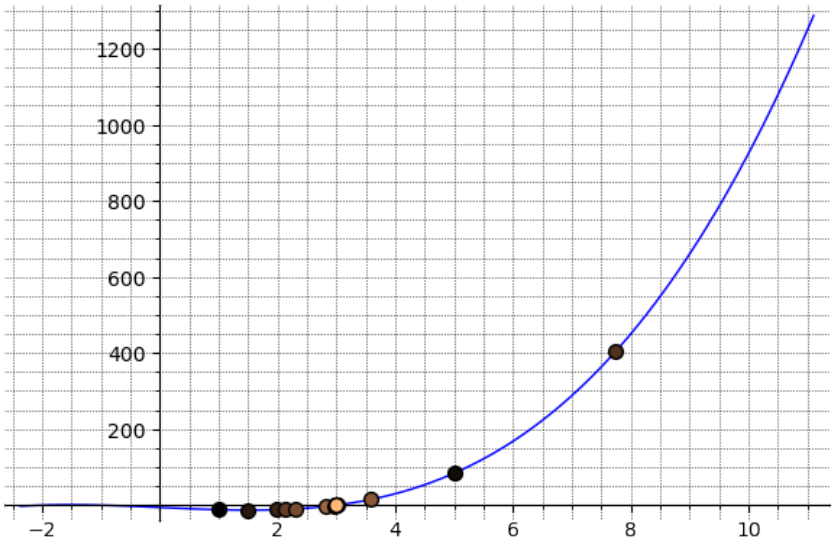


Applichiamo il metodo delle secanti alla funzione polinomiale $f(x) = (x+2)(x+1)(x-3)$, che evidentemente ha uno zero in $x = 3$. Utilizzando $x_0 = 1, x_1 = 5$ come approssimazioni iniziali si ottiene:

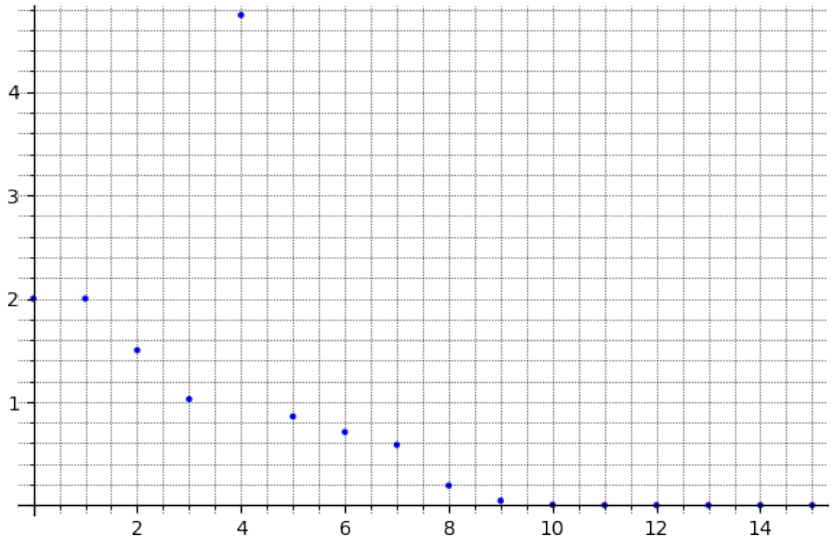
```
f = lambda x: float((x+2)*(x+1)*(x-3))
x = list(secanti(f, 1, 5.))
pd.DataFrame(x)
```

	0
0	1
1	5.00000000000000
2	1.50000000000000
3	1.97297297297297
4	7.74381026843888
5	2.14113466435299
6	2.29183965522379
7	3.58377704398299
8	2.80997632265162
9	2.95623378532361
10	3.00406333014903
11	2.99991888445032
12	2.99999985188026
13	3.00000000000541
14	3.00000000000000
15	3.00000000000000

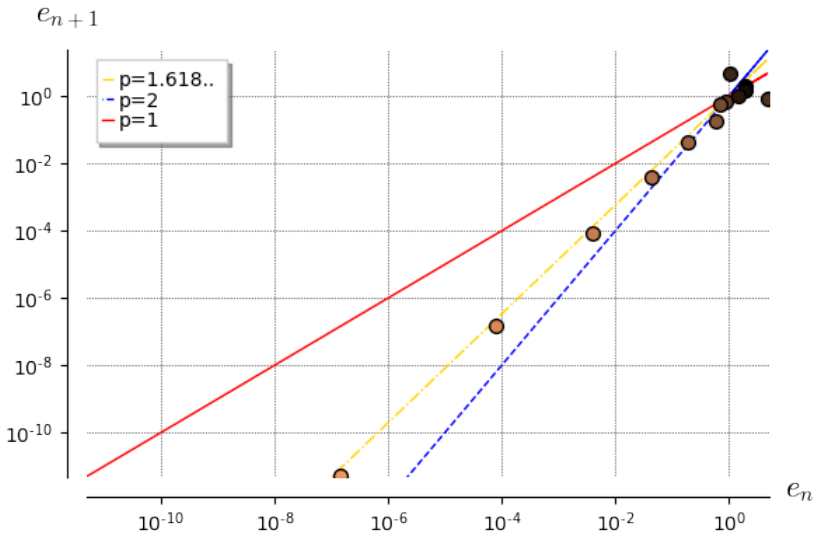
```
plot_secanti(f, x)
```




```
plot_error(x, 3)
```




```
plot_convergence(x, 3)
```



Nelle prime iterazioni il metodo sembra quasi non convergere, ma si stabilizza velocemente.

Notiamo che la pendenza della retta che congiunge i punti sufficientemente allineati delle ultime iterazioni è parallela alla retta gialla (e quindi l'ordine è circa $p = 1.618$) ma traslata verso il basso, quindi in questo caso la costante M^{z_1-1} deve essere minore di 1.

In effetti, si ha che:

$$f(x) = (x+2) * (x+1) * (x-3) = x^3 - 7x - 6, \quad f'(x) = 3x^2 - 7, \quad f''(x) = 6x$$

e quindi:

$$M = \lim_{i \to \infty} \frac{|e_{n+1}|}{|e_n e_{n-1}|} = \frac{|f''(\bar{x})|}{2f'(\bar{x})} = \frac{18}{20} < 1$$

10 Piano di ammortamento di un mutuo

Sia C il capitale per il quale si contrae un mutuo da estinguere in N rate, con un tasso di interesse i fissato per il periodo della rata. Sia y_n il debito residuo dopo il pagamento della rata n-esima ed r l'importo della rata. Si ha:

$$y_n = (1+i)y_{n-1} - r, \quad y_0 = C, \quad n = 1, ..., N \tag{10.1}$$

L'obiettivo della rata r è di estinguere il mutuo, quindi ha senso imporre che dopo N rate il debito residuo si sia annullato: imponiamo $y_N = 0$ e calcoliamo il valore della rata di conseguenza.
La (10.1) è un'equazione alle differenze di primo grado a coefficienti costanti non omogenea. Il polinomio caratteristico dell'equazione omogenea $y_n - (1+i)y_{n-1} = 0$ è:

$$p(z) = z - (1+i)$$

che essendo di primo grado ha una sola radice $z = 1 + i$. La soluzione generale di (10.1) è dunque:

$$y_n = \bar{y} + \alpha z^n$$

in cui $\bar{y}$ è una soluzione particolare dell'equazione non omogenea e α deve essere ricavato dalle condizioni iniziali.
In questo caso, possiamo cercare una soluzione di (10.1), y_c, che sia costante:

$$y_c - (1+i)y_c = -r \Longrightarrow y_c = \frac{r}{i}$$

Imponendo la condizione iniziale $y_0 = C$ ricaviamo α:

$$y_0 = C = \bar{y} + \alpha(1+i)^0 \Longrightarrow \alpha = C - \bar{y} = C - \frac{r}{i}$$

pertanto la soluzione è:

$$y_n = \frac{r}{i} + \left(C - \frac{r}{i}\right)(1+i)^n = C(1+i)^n + \frac{r}{i}(1 - (1+i)^n)$$

Imponendo che il mutuo sia estinto in N rate, possiamo ricavare il valore della rata in funzione del capitale iniziale e del tasso d'interesse:

$$y_N = 0 = C(i+1)^N + \frac{r}{i}(1-(i+1)^N) \quad \Longrightarrow \quad r = \frac{-iC(i+1)^N}{1-(i+1)^N} = C\frac{i}{1-(i+1)^{-N}} = \frac{C}{f}a$$

in cui abbiamo anche definito f, il *fattore di ammortamento*.
Osserviamo che ad ogni rata, iy_n rappresenta la *quota interessi* e $r - iy_n$ la *quota capitale* del mutuo. Al crescere di n verso N ci si aspetta quindi che la quota interessi tenda a zero in favore della quota capitale.

Implementiamo il calcolo del mutuo tramite un iteratore:

```
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]=8,4

def mutuo(C, i, r):
    # Iteratore che restituisce per ogn passo la tripla: (Capitale rimanente, quota interessi, quota capitale)
    yield (C, 0, 0)
    while C > 10**-3:
        C_last = C
        C = C*(1+i)-r
        yield (C, C_last*i, r-C_last*i)


def rata(C, i, N):
    return C * i  / (1-(i+1)^-N)
```

Esempio di mutuo al 5 di interessi su un capitale di 100000 euro da estinguere in 10 rate:

```
N = 10
C = 100000
i = 0.05
r = rata(C,i,N)
print("Rata calcolata: ", r )
andamento_mutuo = list(mutuo(C,i,r))
pd.DataFrame(andamento_mutuo, columns=['Capitale', 'q. interessi', 'q. capitale'])
```

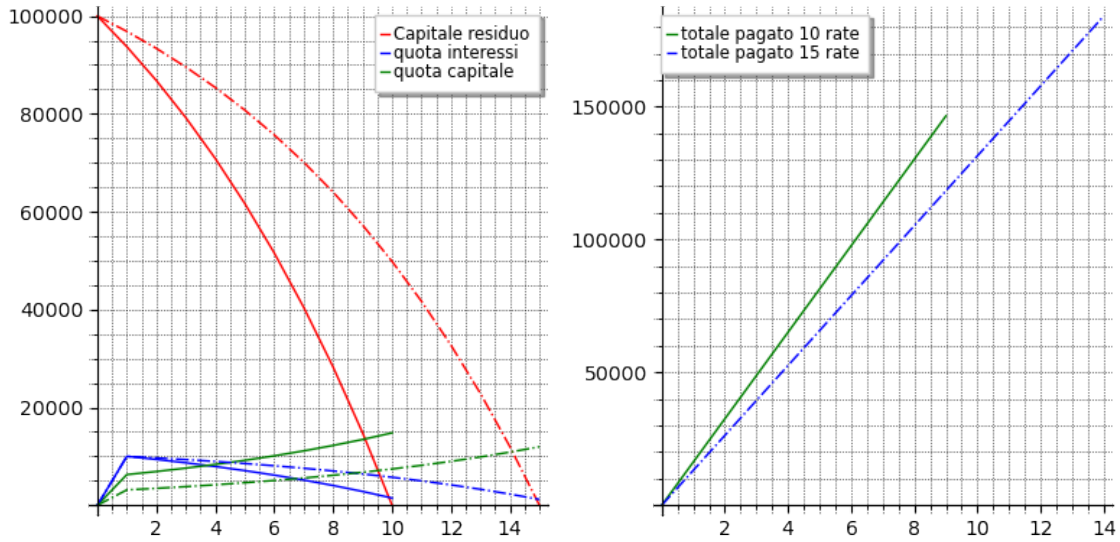
```
Rata calcolata:  12950.4574965457
```

	Capitale	q. interessi	q. capitale
0	100000	0	0
1	92049.5425034543	5000.00000000000	7950.45749654566
2	83701.5621320814	4602.47712517272	8347.98037137294
3	74936.1827421398	4185.07810660407	8765.37938994159
4	65732.5343827012	3746.80913710699	9203.64835943867
5	56068.7036052906	3286.62671913506	9663.83077741060
6	45921.6812890094	2803.43518026453	10147.0223162811
7	35267.3078569143	2296.08406445047	10654.3734320952
8	24080.2157532143	1763.36539284571	11187.0921036999
9	12333.7690443294	1204.01078766072	11746.4467088849
10	1.74622982740402e-10	616.688452216468	12333.7690443292

Che differenza c'è tre estinguere lo stesso mutuo in 10 o in 20 rate a parità di interessi?

```
N = 10
i = 0.1
r = rata(C,i,N)
andamento_mutuo = list(mutuo(C,i,r))
cap_residuo, q_inter, q_cap = zip(*andamento_mutuo)
p = list_plot(cap_residuo, plotjoined=True, color='red', legend_label='Capitale residuo', gridlines='minor')
```

```
p += list_plot(q_inter, plotjoined=True, color='blue', legend_label="quota interessi")
p += list_plot(q_cap, plotjoined=True, color='green', legend_label="quota capitale")
lp = list_plot([r*n for n in range(N)],plotjoined=True, color='green', legend_label=f'totale pagato {N}
↪ rate')

N = 15
i = 0.1
r = rata(C,i,N)
andamento_mutuo = list(mutuo(C,i,r))
cap_residuo, q_inter, q_cap = zip(*andamento_mutuo)
p += list_plot(cap_residuo, plotjoined=True, color='red', linestyle='-.')
p += list_plot(q_inter, plotjoined=True, color='blue', linestyle='-.')
p += list_plot(q_cap, plotjoined=True, color='green', linestyle='-.')
lp += list_plot([r*n for n in range(N)],plotjoined=True, color='blue', linestyle='-.',
                gridlines='minor', legend_label=f"totale pagato {N} rate")

p.set_legend_options(font_size='small')
lp.set_legend_options(font_size='small')
g = graphics_array([p,lp])

g.show()
```

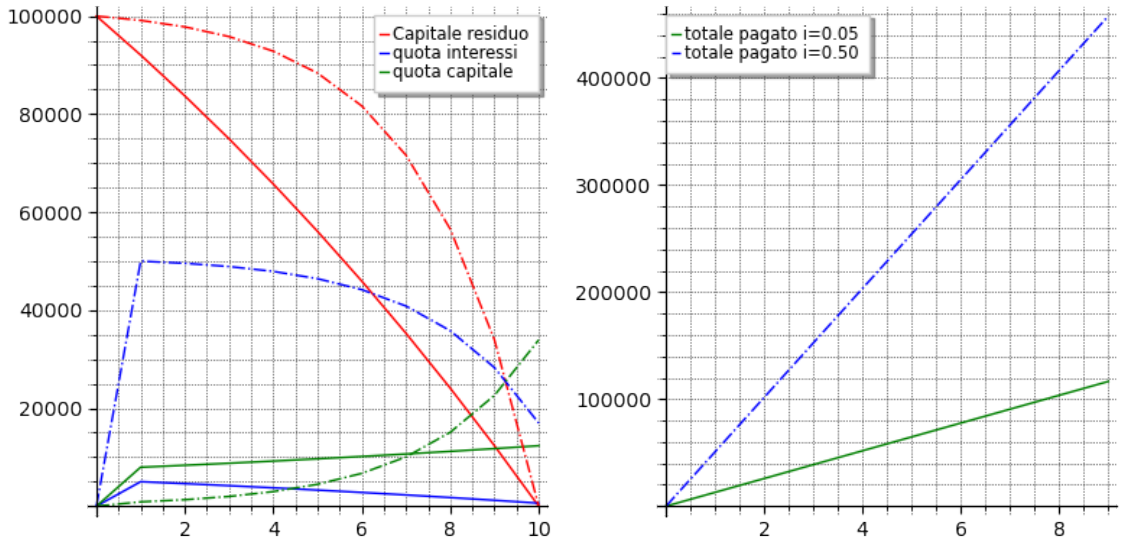


Come ci si aspettava, la quota capitale rimpiazza progressivamente la quota interessi fino all'estinzione del mutuo. Con l'aumento del numero delle rate (e quindi della durata del mutuo, sotto l'ipotesi della rata annuale), il grafico di destra evidenzia che il valore della singola rata diminuisce (la linea trattegiata ha una pendenza minore) ma il totale pagato è maggiore in quanto ci sono più interessi da pagare.

Al variare del tasso di interesse invece:

```
N = 10
i = 0.05
r = rata(C,i,N)
andamento_mutuo = list(mutuo(C,i,r))
cap_residuo, q_inter, q_cap = zip(*andamento_mutuo)
p = list_plot(cap_residuo, plotjoined=True, color='red', legend_label='Capitale residuo',
↪ gridlines='minor')
p += list_plot(q_inter, plotjoined=True, color='blue', legend_label="quota interessi")
p += list_plot(q_cap, plotjoined=True, color='green', legend_label="quota capitale")
lp = list_plot([r*n for n in range(N)],plotjoined=True, color='green', legend_label=f'totale pagato
↪ i={i:.2f}')

N = 10
i = 0.5
r = rata(C,i,N)
```

```
andamento_mutuo = list(mutuo(C,i,r))
cap_residuo, q_inter, q_cap = zip(*andamento_mutuo)
p += list_plot(cap_residuo, plotjoined=True, color='red', linestyle='-.')
p += list_plot(q_inter, plotjoined=True, color='blue', linestyle='-.')
p += list_plot(q_cap, plotjoined=True, color='green', linestyle='-.')
lp += list_plot([r*n for n in range(N)],plotjoined=True, color='blue', linestyle='-.',
               gridlines='minor', legend_label=f'totale pagato i={i:.2f}')

p.set_legend_options(font_size='small')
lp.set_legend_options(font_size='small')
g = graphics_array([p,lp])
g.show()
```



Il tasso di interesse più alto (linee tratteggiate) comporta una quota interessi totale più alta, che si riflette sull'importo della rata e sul totale pagato.

Osserviamo che nel grafico di sinistra l'area sottesa alle curve verdi della quota capitale è costante ed equivale al capitale iniziale; al variare del numero delle rate o del tasso di interesse (ma a parità di capitale iniziale), cambia soltanto l'area sottesa alle curve blu (la quota interessi) e quella sotto alla curva rossa (il totale pagato).

Il caso limite $N \to \infty$

Se il capitale non venisse mai ripagato, ovvero si avesse un numo infinito di rate, potremmo essere nel caso di un canone di locazione, ad esempio di un bene immobile. Osserviamo che la rata del canone, secondo il nostro modello, dovrebbe essere:

$$r = \lim_{N\to\infty} C\frac{i(i+1)^N}{1-(i+1)^N} = iC$$

ovvero il "giusto" canone di affitto sarebbe la quota interessi sul valore del bene affittato.

11 Modello del cobweb

Studiamo la dinamica del prezzo di un bene in un mercato libero. Discretizzando il tempo, assumiamo che la produzione e la vendita dei beni avvenga in cicli; questa ipotesi è conforme alla realtà dei fatti, in quanto spesso i beni vengono fabbricati e immessi sul mercato in serie limitate ed eventualmente ripetute in base alla domanda.
Assumendo che domanda e offerta siano funzioni lineari del prezzo, definiamo:

p_n : prezzo al tempo n

D_n : domanda al tempo n in funzione del prezzo : $g(p_n) = -ap_n + d_0$

S_n : offerta al tempo n in funzione del prezzo : $f(p_{n-1}) = bp_{n-1} + s_0$ (11.1)

in cui $s_0, d_0 > 0$ rappresentano l'offerta e la domanda a prezzo nullo, e i parametri $a, b > 0$. Avendo discretizzato il tempo la quantità offerta al tempo n corrisponde al ciclo di produzione avviato al tempo $n-1$, che ovviamente dipende dal prezzo al tempo $n-1$; ovviamente sotto l'ipotesi che la produzione richieda esattamente un ciclo di tempo.

Imponendo che il mercato sia bilanciato, ovvero che $S_n = D_n$ per ogni n, si ottiene:

$$S_n = D_n \implies bp_{n-1} + s_0 = -ap_n + d_0 \implies ap_n + bp_{n-1} + s_0 - d_0 = 0 \tag{11.2}$$

Ovvero abbiamo un'equazione lineare non omogenea a coefficienti costanti.
Il polinomio caratteristico dell'equazione omogenea è $p(z) = az + b$, che ha una sola radice: $z = -b/a$.
La soluzione dell'equazione non omogenea è quindi: $p_n = \bar{p_n} + \alpha(\frac{-b}{a})^n$, in cui $\bar{p}_n$ è una soluzione particolare dell'equazione non omogenea.

Cerchiamo, se esiste, una soluzione costante $\bar{p}$ per l'equazione non omogenea:

$$a\bar{p} + b\bar{p} + s_0 - d_0 = 0 \implies \bar{p} = \frac{d_0 - s_0}{a + b}$$

La soluzione costante esiste e quindi abbiamo la soluzione generica dell'equazione:

$$p_n = \frac{d_0 - s_0}{a + b} + \alpha \left(\frac{-b}{a}\right)^n$$

in cui α andrà determinato dal prezzo iniziale del bene p_0; in particolare:

$$p_0 = \frac{d_0 - s_0}{a + b} + \alpha(\frac{-b}{a})^0 \implies \alpha = p_0 - \frac{d_0 - s_0}{a + b}$$

ovvero la soluzione è:

$$p_n = \frac{d_0 - s_0}{a + b} + \left(p_0 - \frac{d_0 - s_0}{a + b}\right)\left(\frac{-b}{a}\right)^n = \bar{p} + (p_0 - \bar{p})\left(\frac{-b}{a}\right)^n$$

Osserviamo che l'unico termine che dipende da n nella soluzione è $\left(\frac{-b}{a}\right)^n = c^n$. A regime, ovvero al limite per $n \to \infty$, la soluzione:

- tende al prezzo di equilibrio se $|b| < |a|$ ovvero $c^n \to 0$ (ovvero $p(z)$ è un polinomio di Schur)
- cresce verso l'infinito se $c^n \to \infty$. Osserviamo però che $c < 0$, e quindi il prezzo oscillerà e diventerà negativo ad ogni ciclo di indice dispari, a partire dal momento in cui supera $\bar{p}$ in valore assoluto.
- oscilla tra $2\bar{p} - p_0$ e p_0 se $b = a$, nella sequenza infinita $2\bar{p} - p_0, p_0, 2\bar{p} - p_0, p_0, \ldots$

Implementiamo la ricorrenza $p_n = \dfrac{-bp_{n-1} + d_0 - s_0}{a}$ che abbiamo ottenuto in (11.2):

```
def cobweb(p0, d0, s0, a, b, N):
    p = p0
    yield p
    for i in range(N):
        p = (-b*p + d0 - s0)/a
        yield p
```

Per graficare l'andamento del prezzo, mostriamo in verde la domanda in funzione del prezzo $g(p)$, ed in blu l'offerta $f(p)$.

Ad ogni passo n corrisponde una coppia di frecce (di colore via via più chiaro al crescere di n). La prima freccia collega i punti $(p_n, g(p_n)), (p_n, g(p_{n+1}))$ e la seconda collega $(p_n, g(p_{n+1})), (p_{n+1}, g(p_{n+1}))$, ricordiamo che abbiamo imposto $g(p_n) = f(p_n)$ per ogni n, ovvero che la domanda eguagli l'offerta.

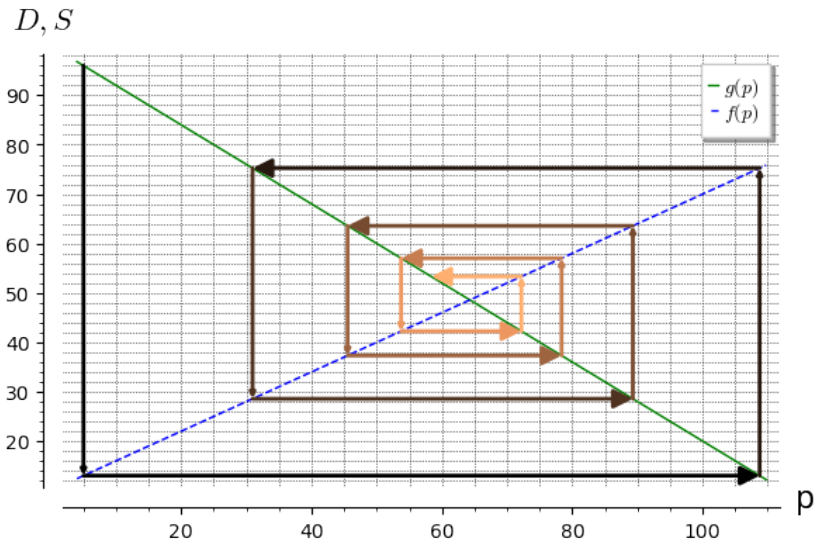
Il prezzo al tempo p_{n+1} può essere letto come la proiezione del prezzo al tempo p_n sulla retta dell'offerta, la quale determina l'offerta al tempo $n+1$. Essendo imposta l'uguaglianza tra domanda e offerta, la proiezione dell'offerta al tempo $n+1$ sulla retta della domanda determina quindi il prezzo.

```
def plot_cobweb(d0,s0,a,b,p):
    minp, maxp = min(p)-1, max(p)+1
    dom_off = lambda p: -a*p+d0
    pl = plot(dom_off, color='green', xmin=minp, xmax=maxp, axes_labels=['p', '$D,S$'], gridlines='minor',
    ↪  legend_label='$g(p)$')
```

```
    pl += plot(lambda p: b*p+s0, color='blue', xmin=minp, xmax=maxp, legend_label='$f(p)$', linestyle='--')
    for i in range(len(p)-1):
        pl += arrow((p[i],dom_off(p[i])), (p[i],dom_off(p[i+1])),
        ↪  rgbcolor=list(colormaps.copper(i/(len(p)-1))[0:3]), arrowsize=1 )
        pl += arrow((p[i],dom_off(p[i+1])), (p[i+1],dom_off(p[i+1])),
        ↪  rgbcolor=list(colormaps.copper(i/(len(p)-1))[0:3]), arrowsize=4)
    pl.show()
```

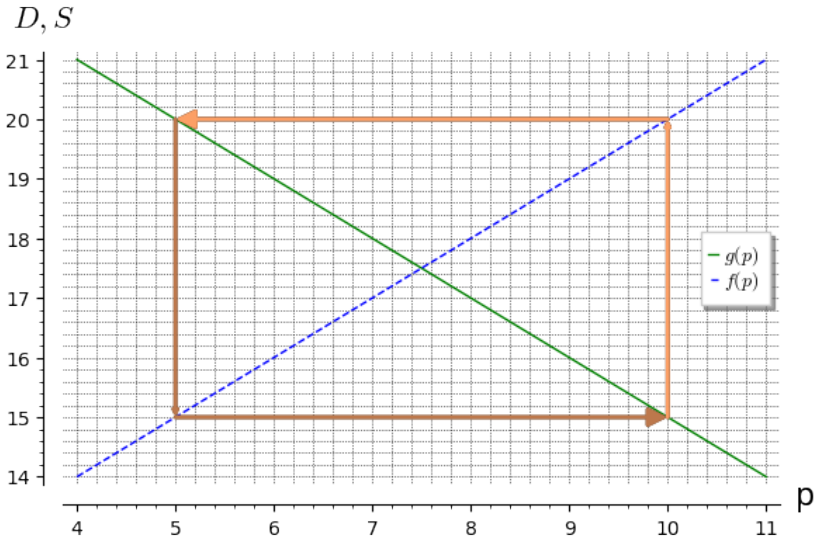
Nel caso in cui $|b| < |a|$, come ci aspettavamo l'andamento è stabile verso il prezzo di equilibrio.

```
p0 = 5.
d0 = 100.
s0 = 10.
a = 0.8
b = 0.6
N = 8
p = list(cobweb(p0, d0, s0, a, b, N))
plot_cobweb(d0, s0, a, b, p)

```



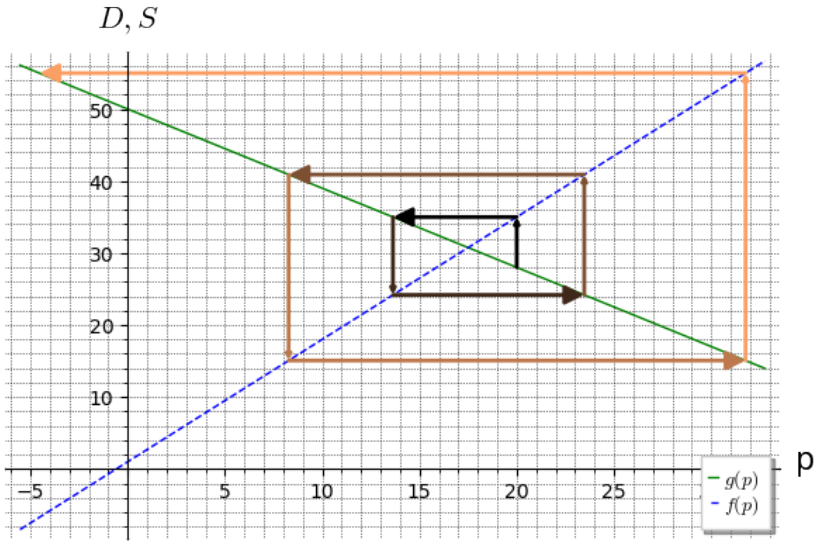
Nel caso in cui $|b| = |a|$, il prezzo oscilla tra $p0$ e $2\bar{p} - p_0$:

```
p0 = 10.
d0 = 25.
s0 = 10.
a = 1
b = 1
N = 5
p = list(cobweb(p0, d0, s0, a, b, N))
plot_cobweb(d0, s0, a, b, p)

```



Nel caso in cui $|b| > |a|$, l'equilibrio è instabile e il prezzo oscilla crescendo in valore assoluto. Le oscillazioni portano velocemente il prezzo a diventare negativo; questo può essere letto come una tendenza alla sovraproduzione quando il prezzo sale, che porta alla bancarotta.

```
p0 = 20.
d0 = 50.
s0 = 1.
a = 1.1
b = 1.7
N = 5
p = list(cobweb(p0, d0, s0, a, b, N))
plot_cobweb(d0, s0, a, b, p)

```



I parametri a e b possono anche essere interpretati come la velocità di risposta della domanda e dell'offerta. Se il produttore (l'offerta) risponde troppo velocemente al cambiamento di prezzo ($|b| > |a|$), il mercato diventa instabile.

11.1 Estensione

Il modello del cobweb può essere migliorato basando il calcolo dell'offerta su un prezzo di mercato previsto, chiamiamolo f_n, anziché sul prezzo al tempo $n-1$ come in (11.1) [p.22]. Stimiamo il prezzo previsto con l'espressione:

$$f_n = p_{n-1} + \rho(p_{n-1} - p_{n-2})$$

ovvero la previsione tiene conto del trend di cambiamento del prezzo. Le nuove funzioni per domanda e offerta sono quindi:

$$D_n \text{ : domanda al tempo } n = g(p_n) = -ap_n + d_0$$

$$S_n \text{ : offerta al tempo } n = f(p_{n-1}, p_{n-2}) = bf_n + s_0 = b\left(p_{n-1} + \rho(p_{n-1} - p_{n-2})\right) + s_0$$

Come prima, otteniamo l'equazione del modello eguagliando domanda e offerta:

$$S_n = D_n \Longrightarrow b\left(p_{n-1} + \rho(p_{n-1} - p_{n-2})\right) + s_0 = -ap_n + d_0$$

$$ap_n + b(1+\rho)p_{n-1} - b\rho p_{n-2} = d_0 - s_0 \tag{11.3}$$

ovvero abbiamo un'equazione non omogenea di secondo grado. Il polinomio caratteristico associato è:

$$p(z) = az^2 + b(1+\rho)z - b\rho$$

La soluzione generica dell'equazione omogenea associata alla (11.3), $y_n = \sum_{i=0}^{2} \alpha_i z_i^n$, sarà asintoticamente stabile soltanto se $p(z)$ è di Schur, ovvero se $|z_i| < 1$.
Per il primo criterio di Schur deve valere per i coefficienti $|p_0| > |p_k|$, ovvero $|a| > |b\rho|$, ed il polinomio ridotto $p^{(1)}(z)$ dev'essere di Schur anch'esso. In questo caso quindi:

$$p^{(1)}(z) = \frac{\bar{p_0}p(z) - p_k q(z)}{z} = \frac{ap(z) + b\rho q(z)}{z} = \frac{(a^2z^2 + ab(1+\rho)z + ab\rho) + b\rho(b\rho z^2 + b(1+\rho)z + a)}{z}$$

$$= \frac{(a^2 - (b\rho)^2)z^2 + (ab(1+\rho) + b^2\rho(1+\rho))z + ab\rho - ab\rho}{z} = (a^2 - b^2\rho^2)z + b(1+\rho)(a+b\rho)$$

ovvero devono valere:

$$\frac{b|\rho|}{a} < 1, \quad |z| = \left|\frac{b(1+\rho)(a+b\rho)}{(a^2 - b^2\rho^2)}\right| = \left|\frac{b(1+\rho)}{(a-b\rho)}\right| = \left|\frac{b(1+\rho)}{a(1-\frac{b}{a}\rho)}\right| < 1$$

Osserviamo che se $\rho = 0$, la condizione di stabilità ridiventa $|a| > |b|$ come nel caso precedente.
Assumendo $a, b > 0$ e imponendo $\frac{b|\rho|}{a} < 1$, ci sono soltanto due casi da analizzare:

- se $1 + \rho > 0$, semplificando si ottiene: $\frac{b}{a}(1+2\rho) < 1$. Si deve quindi avere $\frac{b}{a} < (max\{\rho, 1+2\rho\})^{-1}$ affinché il metodo sia asintoticamente stabile
- se $1 + \rho < 0$ allora $\rho < -1$ e semplificando si ottiene la condizione $\frac{b|\rho|}{a} < 1$ che abbiamo già imposto. In quanto $|\rho| > 1$, la condizione è però più restrittiva rispetto a quella del modello semplice visto in precedenza, $\frac{b}{a} < 1$

Implementiamo il metodo del cobweb esteso tramite la ricorrenza (11.3):

$$p_n = \frac{-b(1+\rho)p_{n-1} + b\rho p_{n-2} + d_0 - s_0}{a}$$

```
def cobweb_esteso(p0, p1, d0, s0, a, b, rho, N):
    yield p0
    yield p1
    for i in range(N):
        p2 = (-b*(1+rho)*p1 +b*rho*p0 + d0 - s0)/a
        yield p2
        p0=p1
        p1=p2

```

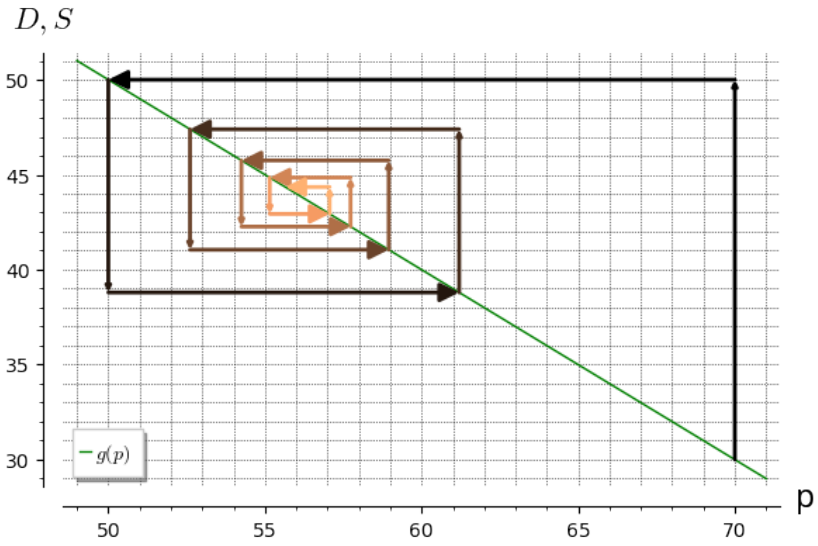
Per graficare l'andamento del prezzo, mostriamo in verde la domanda in funzione del prezzo $g(p)$. A differenza del caso semplice, adesso l'offerta non è più rappresentabile con una retta in quanto è una funzione dei due prezzi precedenti.

Ad ogni passo n corrisponde una coppia di frecce (di colore via via più chiaro al crescere di n). La prima freccia collega i punti $(p_n, g(p_n)), (p_n, g(p_{n+1}))$ e la seconda collega $(p_n, g(p_{n+1})), (p_{n+1}, g(p_{n+1}))$, ricordiamo che abbiamo imposto $g(p_n) = f(p_n)$ per ogni n, ovvero che la domanda eguagli l'offerta.

```
import numpy as np
def plot_cobweb(d0,s0,a,b,rho,p):
    minp, maxp = min(p)-1, max(p)+1
    dom = lambda p: -a*p+d0
    pl = plot(dom, color='green', xmin=minp, xmax=maxp, axes_labels=['p', '$D,S$'], gridlines='minor',
    ↪  legend_label='$g(p)$')
    for i in range(len(p)-1):
        pl += arrow((p[i],dom(p[i])), (p[i],dom(p[i+1])),
        ↪  rgbcolor=list(colormaps.copper(i/(len(p)-1))[0:3]), arrowsize=1 )
        pl += arrow((p[i],dom(p[i+1])), (p[i+1],dom(p[i+1])),
        ↪  rgbcolor=list(colormaps.copper(i/(len(p)-1))[0:3]), arrowsize=4)
    pl.show()
```

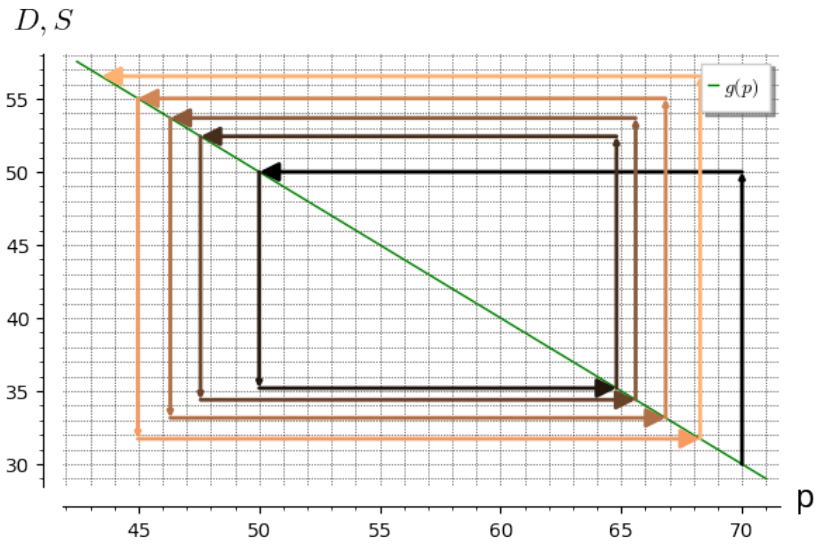
Nel caso in cui $\frac{b}{a}|\rho| < 1$, come ci aspettavamo l'andamento è stabile verso il prezzo di equilibrio.

```
p0 = 70.
p1 = 50.
rho = 0.1
d0 = 100.
s0 = 10.
a = 1.
b = 0.6
N = 8
p = list(cobweb_esteso(p0, p1, d0, s0, a, b, rho, N))
plot_cobweb(d0, s0, a, b, rho, p)

```



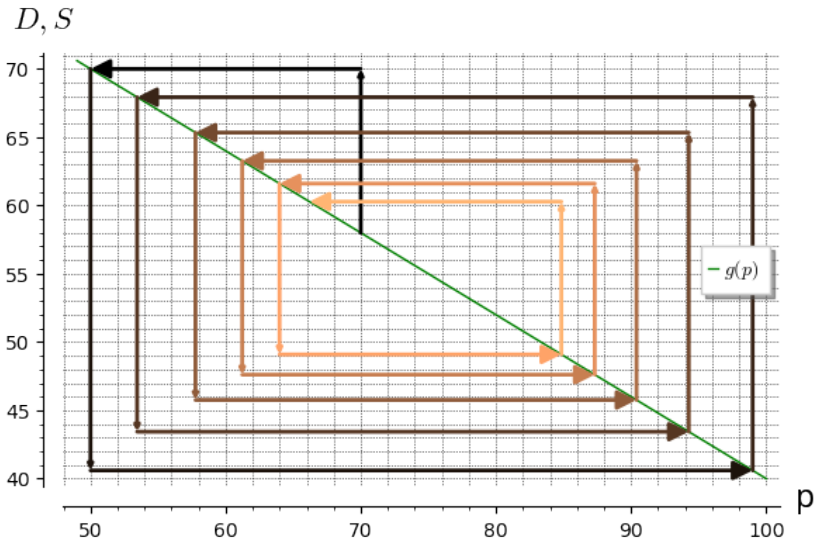
Aumentando ρ fino ad avere $\frac{b}{a} > (max\{\rho, 1+2\rho\})^{-1}$ (in questo caso, $0.6/1$), si perde la stabilità.

```
p0 = 70.
p1 = 50.
rho = 0.4
d0 = 100.
s0 = 10.
a = 1.
b = 0.6
N = 8
p = list(cobweb_esteso(p0, p1, d0, s0, a, b, rho, N))
plot_cobweb(d0, s0, a, b, rho, p)

```



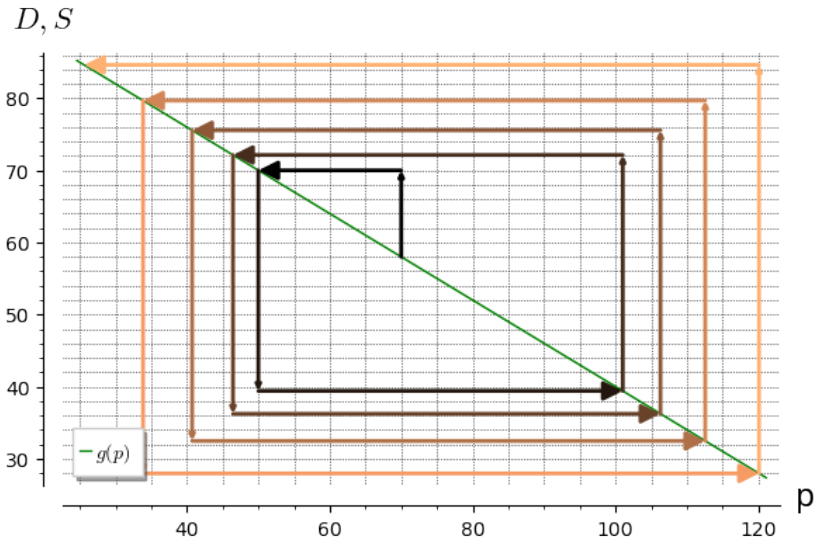
Se $a = b$, il sistema converge se $-1 < \rho < 0$ e diverge altrimenti:

```
p0 = 70.
p1 = 50.
rho = -0.05
d0 = 100.
s0 = 10.
a = 0.6
b = 0.6
N = 10
p = list(cobweb_esteso(p0, p1, d0, s0, a, b, rho, N))
plot_cobweb(d0, s0, a, b, rho, p)

```

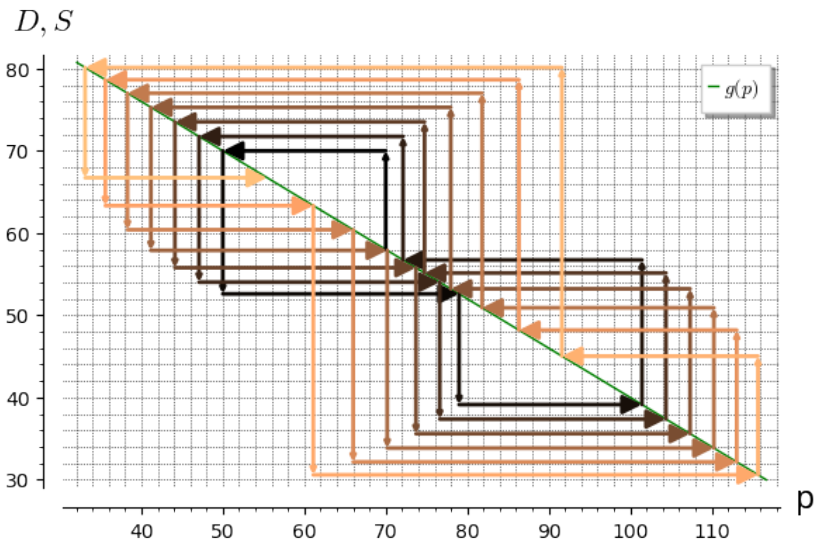



```
p0 = 70.
p1 = 50.
rho = 0.05
d0 = 100.
s0 = 10.
a = 0.6
b = 0.6
N = 8
p = list(cobweb_esteso(p0, p1, d0, s0, a, b, rho, N))
```

```
plot_cobweb(d0, s0, a, b, rho, p)

```




```
p0 = 70.
p1 = 50.
rho = -1.05
d0 = 100.
s0 = 10.
a = 0.6
b = 0.6
N = 25
p = list(cobweb_esteso(p0, p1, d0, s0, a, b, rho, N))
plot_cobweb(d0, s0, a, b, rho, p)

```

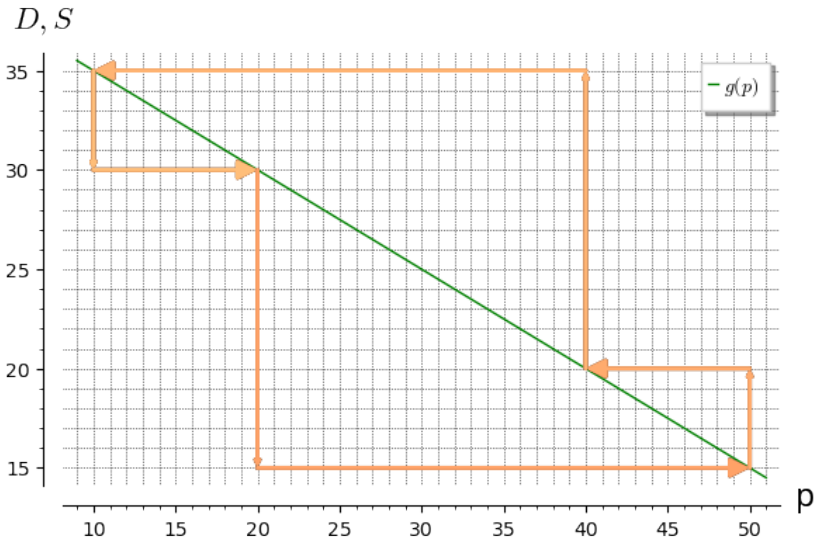


Se $a = b$ e $\rho = -1$, otteniamo un sistema oscillante a quattro stati. Infatti:

$$p_n = \frac{-b(1+\rho)p_{n-1} + b\rho p_{n-2} + d_0 - s_0}{a} = -p_{n-2} + \frac{d0 - s0}{a} = -p_{n-2} + c$$

Si ha quindi la successione $\{p_0, p_1, -p_0 + c, -p_1 + c, -(-p_0 + c) + c = p_0, -(-p_1 + c) + c = p_1, -p_0 + c, -p_1 + c, p_0, p_1, ...\}$

```
p0 = 10.
p1 = 20.
rho = -1
d0 = 40.
s0 = 10.
a = 0.5
b = 0.5
N = 20
p = list(cobweb_esteso(p0, p1, d0, s0, a, b, rho, N))
plot_cobweb(d0, s0, a, b, rho, p)

```



12 Economia di una nazione

Siano Y_n il prodotto interdo lordo, I_n gli investimenti, C_n i consumi e G_n le spese di governo.
In generale si ha:

$$Y_n = I_n + C_n + G_n$$

Assumiamo che i consumi siano una frazione della ricchezza al tempo precedente:

$$C_n = \alpha Y_{n-1}, \quad 0 < \alpha < 1$$

e che gli investimenti siano proporzionali all'andamento dei consumi:

$$I_n = \rho(C_n - C_{n-1}), \quad \rho > 0$$

Assumiamo anche che le spese di governo siano costanti $G_n \equiv G > 0$. Si ha quindi:

$$Y_n = \rho(C_n - C_{n-1}) + C_n + G = \rho\alpha(Y_{n-1} - Y_{n-2}) + \alpha Y_{n-1} + G$$

ovvero:

$$Y_n - \alpha(1+\rho)Y_{n-1} + \alpha\rho Y_{n-2} = G$$

che è un'equazione alle differenze lineare di secondo grado a coefficienti costanti.
Date $f_n^{(1)}, f_n^{(2)}$ due soluzioni linearmente indipendenti dell'equazione omogenea associata e $\bar{Y}_n$ una soluzione particolare dell'equazione non omogenea, si ha:

$$Y_n = c_1 f_n^{(1)} + c_2 f_n^{(1)} + \bar{Y}_n$$

Essendo i coefficienti costanti possiamo costruire le $f_n^{(i)}$ utilizzando le radici del polinomio caratteristico associato:

$$p(z) = z^2 - \alpha(1+\rho)z + \alpha\rho \quad \Longrightarrow z_{1/2} = \frac{\alpha(1+\rho) \pm \sqrt{\alpha^2(1+\rho)^2 - 4\alpha\rho}}{2}$$

e quindi:

$$Y_n = c_1 z_1^n + c_2 z_2^n + \bar{Y}_n$$

in cui c_1, c_2 possono essere univocamente determinati dalle condizioni iniziali Y_0, Y_1.
Osserviamo che esiste un punto fisso di equilibrio dell'equazione non omogenea:

$$Y - \alpha(1+\rho)Y - \alpha\rho Y - G = 0 \Longrightarrow Y = \frac{G}{1-\alpha}, \quad \alpha \neq 1$$

e quindi abbiamo anche $\bar{Y}_n \equiv \bar{Y} = Y$.

La soluzione Y_n è asintoticamente stabile se $p(z)$ è un polinomio di Schur, ovvero devono valere:

- $|p_0| > |p_k|$, quindi $1 > |\alpha\rho|$ ovvero $\rho < \alpha^{-1}$
- il polinomio ridotto $(1+\alpha\rho)z - \alpha(1+\rho)$ di Schur ovvero $\alpha < 1$

Usiamo una semplice implementazione per osservare il comportamento del modello:

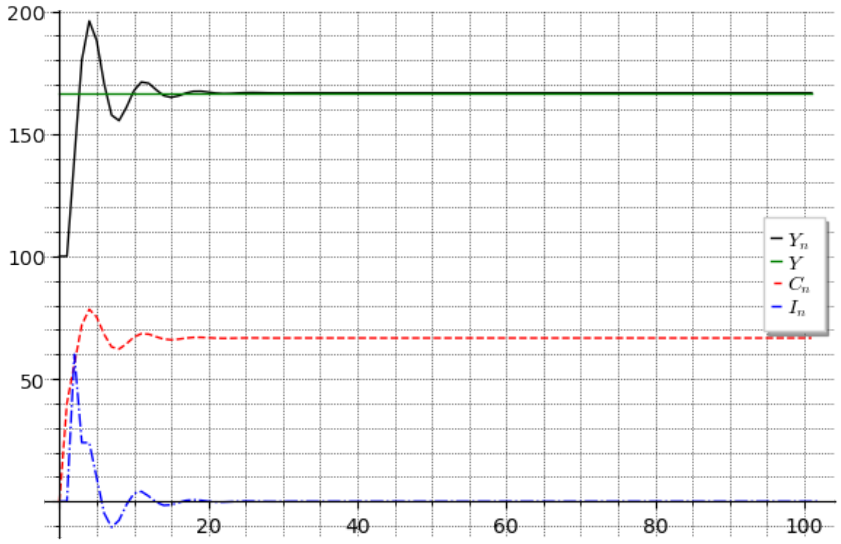
```
def economia(Y0, Y1, rho, alpha, G, yield_args=True, iterations=100):
        yield Y0
        yield Y1
        for i in range(iterations):
            Y = alpha*(1+rho)*Y1 - alpha*rho*Y0+G
            yield Y
            Y0 = Y1
            Y1 = Y
```

Disegnamo in verde la retta che rappresenta la soluzione costante; in nero l'andamento del PIL Y_n, in rosso i consumi C_n e in blu gli investimenti I_n.

```
import numpy as np
def plot_economia(rho,alpha,G,Y):
    base = list(range(len(Y)))
    Y = np.array(Y)
    C = np.concatenate(([0], alpha * Y[1:]))
    I = np.concatenate(([0,0], rho * (C[1:]-C[0:-1])))
    barY = G/(1-alpha)
    pl = list_plot(Y, plotjoined=True, color='black', gridlines='minor', legend_label='$Y_n$')
    pl += list_plot([barY]*len(Y), plotjoined=True, color='green', legend_label="$Y$")
    pl += list_plot(C, plotjoined=True, color='red', legend_label='$C_n$', linestyle='--')
    pl += list_plot(I, plotjoined=True, color='blue', legend_label='$I_n$', linestyle='-.')
    pl.show()
```

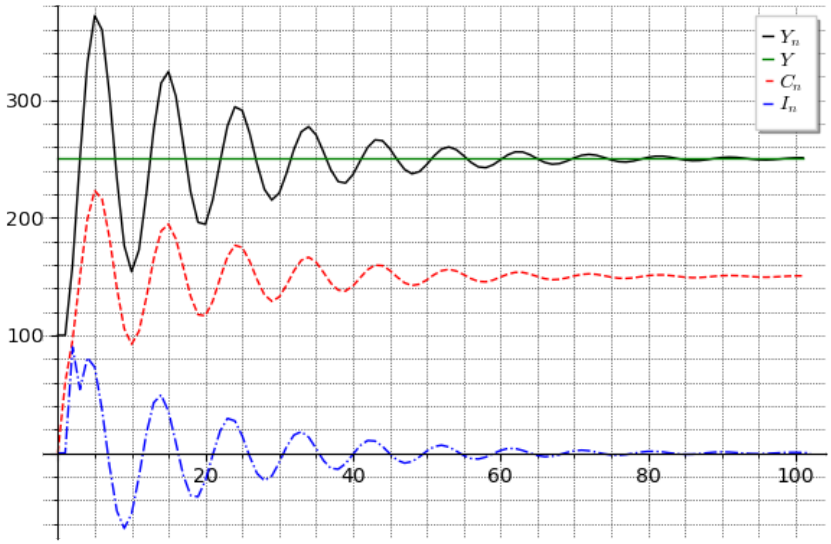
Con $\rho < \alpha^{-}1$ e $\alpha < 1$ il sistema è asintoticamente stabile:

```
Y0=100
Y1=100
rho=1.5
alpha=0.4
G=100
Y = list(economia(Y0,Y1,rho,alpha,G))
plot_economia(rho,alpha,G,Y)
```



Al crescere di α, la convergenza rallenta:

```
Y0=100
Y1=100
rho=1.5
alpha=0.6
G=100
Y = list(economia(Y0,Y1,rho,alpha,G))
plot_economia(rho,alpha,G,Y)
```



Quando $\rho = \alpha^{-1}$, il polinomio caratteristico $p(z)$ diventa di Von Neumann, e la soluzione è stabile ma non asintoticamente stabile. Si ha:

$$p(z) = z^2 - \alpha(1+\rho)z + \alpha\rho \quad \Longrightarrow z^2 - (\alpha+1)z + 1$$

per il quale vale il secondo criterio di Schur ovvero $p^{(1)}(z) \equiv 0$ e $p'(z)$ è di Schur. In quanto $p_0 = 1, p_1 = -(\alpha+1), p_2 = 1$ sono tutti numeri reali e $p_0 = p_2$:

$$q(z) = \sum_{i=0}^{k} \bar{p_i} z^i = p_2 z^2 + p_1 z + p_0 = p(z)$$

ed avendo anche $p(0) = q(0) = 1$, si ottiene:

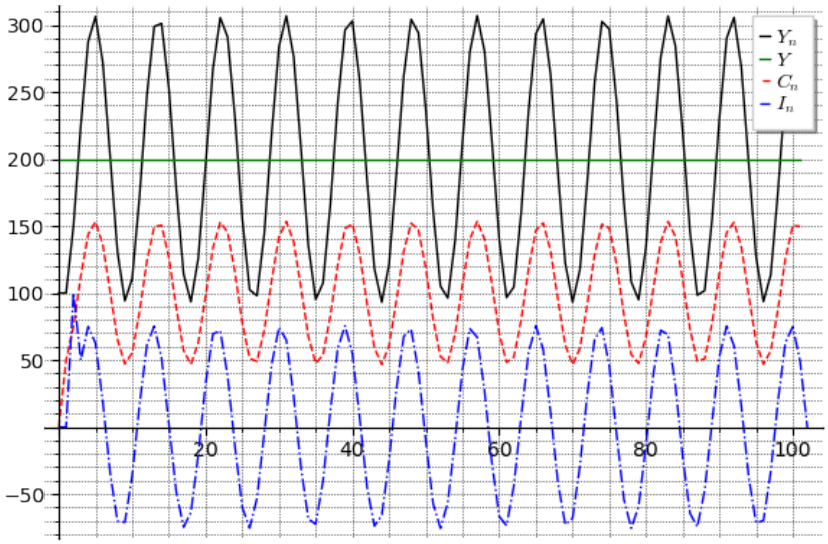
$$p^{(1)}(z) = \frac{q(0)p(z) - p(0)q(z)}{z} = \frac{p(z) - p(z)}{z} = 0$$

ed inoltre:

$$p'(z) = 2z - (\alpha + 1)$$

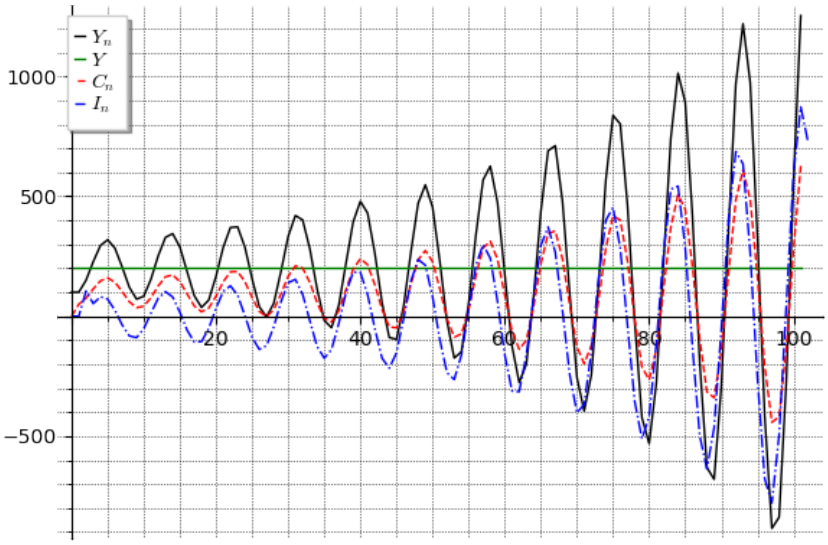
e quindi $p(z)$ è di Von Neumann se $|2| > |\alpha + 1|$ ovvero se $\alpha < 1$ e $\rho = \alpha^{-1}$.

```
Y0=100
Y1=100
rho=2
alpha=0.5
G=100
Y = list(economia(Y0,Y1,rho,alpha,G,iterations=100))
Y
plot_economia(rho,alpha,G,Y)
```



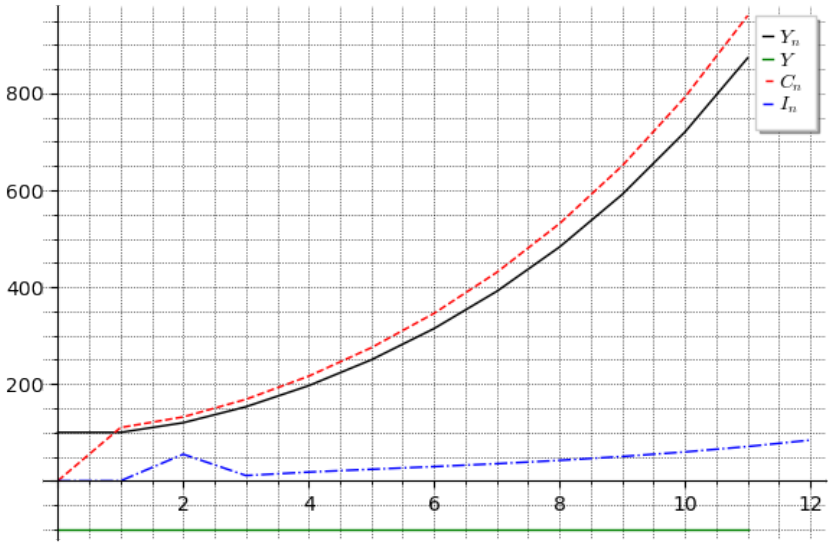
Quando $\rho > \alpha^{-1}$, $p(z)$ non è di Schur e il sistema è instabile. Ovviamenete, non appena il PIL (Y_n) diventa negativo, si avrebbe la bancarotta.

```
Y0=100
Y1=100
rho=2.1
alpha=0.5
G=100
Y = list(economia(Y0,Y1,rho,alpha,G))
plot_economia(rho,alpha,G,Y)
```



Con $\alpha > 1$ si avrebbe una crescita esponenziale infinita. Si avrebbero anche consumi più alti rispetto al PIL, che ovviamente nel mondo reale equivale alla bancarotta... a meno che non si usufrisca di prestiti senza l'intenzione di poi restituire il denaro:

```
Y0=100
Y1=100
rho=0.5
alpha=1.1
G=10
Y = list(economia(Y0,Y1,rho,alpha,G, iterations=10))
plot_economia(rho,alpha,G,Y)
```



13 Linear Multistep Formula Methods

Approssimiamo la soluzione ad un problema di Cauchy:

$$y'(t) = f(t, y(t)), \quad t \in [t_0, T], \quad y(t_0) = y_0$$

su un dominio discreto uniforme:

$$t_n = t_0 + nh, \quad n = 0, ..., N, \quad h = \frac{T - t_0}{N}$$

attraverso il metodo LMF a k passi definito dall'equazione:

$$\sum_{i=0}^{k} \alpha_i y_{n+i} = h \sum_{i=0}^{k} \beta_i f(t_{n+i}, y_{n+i}) \tag{13.1}$$

I polinomi:

$$\rho(z) = \sum_{i=0}^{k} \alpha_i z^k, \quad \sigma(z) = \sum_{i=0}^{k} \beta_i z^k$$

detti rispettivamente primo e secondo polinomio caratteristico, permettono di rappresentare il metodo LMF in forma compatta come:

$$LMF(\rho, \sigma): \quad \rho(E)y_n - h\sigma(E)f_n = 0$$

in cui $f_n = f(t_n, y_n)$.

La (13.1) è definita ad ogni passo a meno di una costante moltiplicativa. Possiamo quindi porre $\alpha_k = 1$ senza perdita di generalità. Esplicitiamo i termini incogniti:

$$y_{n+k} - h\beta_k f(t_k, y_{n+k}) = -\sum_{i=0}^{k-1} \alpha_i y_{n+i} + h \sum_{i=0}^{k-1} \beta_i f(t_{n+i}, y_{n+i}) \tag{13.2}$$

Ad ogni passo abbiamo quindi bisogno di k condizioni iniziali $y_n, ..., y_{n+k-1}$, e, se il metodo è esplicito ovvero $\beta_k \neq 0$, dobbiamo anche risolvere l'equazione in y_{n+k}:

$$y_{n+k} - h\beta_k f(t_{n+k}, y_{n+k}) - g_n = 0$$

che in generale è non lineare, e deve essere risolta iterativamente a meno di una tolleranza con un metodo numerico.
In particolare, osserviamo che per 'avviare' il metodo, abbiamo bisogno di k condizioni iniziali (a differenza della singola condizione iniziale che definisce il problema di Cauchy).

Ordine del metodo

Supponiamo di conoscere la funzione $\hat{y}(t)$ soluzione esatta del problema. Indichiamo con $\hat{y}_n$ le sue valutazioni nei punti della mesh t_n e poniamo $\hat{f}_n = f(t_n, \hat{y}(t_n)) = \hat{y}'(t_n)$. Applicando il metodo, ad ogni passo otterremo un errore:

$$\sum_{i=0}^{k} \alpha_i \hat{y}_{n+i} - h \sum_{i=0}^{k} \beta_i \hat{f}_{n+i} \equiv \tau_{n+k} \tag{13.3}$$

Se $\tau_n = O(h^{p+1})$ per ogni $n > n_0 + k$, il metodo LMF ha ordine p. Se $p \geq 1$ il metodo si dice consistente.
Sviluppando la $\hat{y}(t)$ in serie di taylor in (13.3) e imponendo $\tau_{n+k} = O(h^{p+1})$, si ottengono le condizioni di ordine:

$$\sum_{i=0}^{k} \alpha_i = 0, \qquad \sum_{i=0}^{k} (i^j \alpha_i - j i^{(j-1)} \beta_i) = 0,\ j = 1, ..., p \tag{13.4}$$

Possiamo quindi calcolare i coefficienti α_i, β_i imponendo le condizioni di ordine in un unico sistema lineare:

$$\begin{pmatrix} 1 & \cdots & 1 & 0 & \cdots & 0 \\ 0^1 & \cdots & k^1 & -1 \cdot 0^0 & \cdots & -1 \cdot k^0 \\ 0^2 & \cdots & k^2 & -2 \cdot 0^1 & \cdots & -2 \cdot k^1 \\ 0^3 & \cdots & k^3 & -3 \cdot 0^2 & \cdots & -3 \cdot k^2 \\ & \vdots & & & \vdots & \\ 0^p & \cdots & k^p & -p \cdot 0^{p-1} & \cdots & -p \cdot k^{p-1} \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_k \\ \beta_0 \\ \vdots \\ \beta_k \end{pmatrix} = M_p \mathbf{c} = \mathbf{0} \tag{13.5}$$

Per costruzione, la matrice $M_p \in \mathbb{R}^{p+1 \times 2(k+1)}$ è una istanza di matrice di *Vandermonde confluente*, e si può dimostrare essere nonsingolare quando $p = 2k + 1$.
Imponiamo la condizione $\alpha_k = 1$ rimuovendo la $k-esima$ colonna di M_p e quindi portandone il valore nel vettore noto. Otteniamo il sistema:

$$\begin{pmatrix} 1 & \cdots & 1 & 0 & \cdots & 0 \\ 0^1 & \cdots & (k-1)^1 & -1 \cdot 0^0 & \cdots & -1 \cdot k^0 \\ 0^2 & \cdots & (k-1)^2 & -2 \cdot 0^1 & \cdots & -2 \cdot k^1 \\ 0^3 & \cdots & (k-1)k^3 & -3 \cdot 0^2 & \cdots & -3 \cdot k^2 \\ & \vdots & & & \vdots & \\ 0^p & \cdots & (k-1)^p & -p \cdot 0^{p-1} & \cdots & -p \cdot k^{p-1} \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_{k-1} \\ \beta_0 \\ \vdots \\ \beta_k \end{pmatrix} = M_{p\alpha} \mathbf{c}_\alpha = \begin{pmatrix} -1 \\ -k^1 \\ \vdots \\ -k^p \end{pmatrix} = \mathbf{b}$$

in cui $M_{p\alpha} \in \mathbb{R}^{p+1 \times 2k+1}$. Il metodo LMF implicito ha ordine massimo quando $p = 2k$, esiste ed è unico in quanto in quel caso $M_{p\alpha}$ è nonsingolare.

Imponiamo anche che il metodo sia esplicito, cioè $\beta_k = 0$ togliendo l'ultima colonna di $M_{p\alpha}$, ottenendo il sistema:

$$\begin{pmatrix} 1 & \cdots & 1 & 0 & \cdots & 0 \\ 0^1 & \cdots & (k-1)^1 & -1\cdot 0^0 & \cdots & -1\cdot(k-1)a^0 \\ 0^2 & \cdots & (k-1)^2 & -2\cdot 0^1 & \cdots & -2\cdot(k-1)^1 \\ 0^3 & \cdots & (k-1)k^3 & -3\cdot 0^2 & \cdots & -3\cdot(k-1)^2 \\ & \vdots & & & \vdots & \\ 0^p & \cdots & (k-1)^p & -p\cdot 0^{p-1} & \cdots & -p\cdot(k-1)^{p-1} \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_{k-1} \\ \beta_0 \\ \vdots \\ \beta_{k-1} \end{pmatrix} = M_{p\alpha\beta}\mathbf{c}_{\alpha\beta} = \begin{pmatrix} -1 \\ -k^1 \\ \vdots \\ -k^p \end{pmatrix} = \mathbf{b}$$

in cui $M_{p\alpha\beta} \in \mathbb{R}^{p+1\times 2k}$. Il metodo LMF esplicito ha ordine massimo quando $p = 2k-1$, esiste ed è unico in quanto in quel caso $M_{p\alpha\beta}$ è nonsingolare.

Per calcolare numericamente i coefficienti del metodo LMF, sia esso implicito o esplicito, a k passi di ordine massimo, costruiamo la matrice M_p, il vettore $\mathbf{b}$, quindi calcoliamo $\mathbf{c} = \mathbf{M_p^{-1}b}$.

Il vettore finale dei coefficienti $(\alpha_0, ..., \alpha_k, \beta_0, ..., \beta_k)$ si ottiene infine riaggiungendo α_k e β_k nelle rispettive posizioni in $\mathbf{c}$.

```python
import numpy as np

def LMF(k, type='implicit'):

    if type == 'implicit':
        p = 2*k
    elif type == 'explicit':
        p = 2*k-1
    else:
        raise Exception('Type must be either implicit or explicit')

    #Left half of the confluent vandermonde matrix (first k columns)
    v = np.array(range(k))
    MpL = np.array([v**i for i in range(0,p+1)])

    #Right half of the matrix (last k+1 (or k if explicit) colunms)
    vsize = k + (1 if type == 'implicit' else 0)
    v = np.array(range(vsize))
    MpR = np.array([np.zeros(vsize)]+[-i*v**(i-1) for i in range(1,p+1)], dtype=int)

    #Concatenate left and right part to obtain M_p
    M = np.concatenate((MpL,MpR), axis=1)

    #b vector of the known terms [-1,-k^1,...,-k^p]
    b = -np.array([1]+[k^i for i in range(1,p+1)])

    #Solve the system
    c = np.linalg.solve(M,b)

    #Add back the missing a_k coefficient (=1) and b_k (=0) if necessary
    c = np.concatenate((c[0:k],[1],c[k:],[0] if type == 'explicit' else []))

    return c
```

Calcoliamo i coefficienti dei metodi dei primi ordini, limitando la precisione visualizzata a pochi decimali per agevolare la lettura:

```python
np.set_printoptions(precision=3, suppress=True)
for k in range(1,6):
    c = LMF(k, type='implicit')
```

```
        print('k=%s' % k)
        print('a0,...,ak: ', c[0:k+1])
        print('b0,...,bk: ', c[k+1:])
        print()

```

```
k=1
a0,{\ldots},ak:  [-1.  1.]
b0,{\ldots},bk:  [0.5 0.5]

k=2
a0,{\ldots},ak:  [-1.  0.  1.]
b0,{\ldots},bk:  [0.333 1.333 0.333]

k=3
a0,{\ldots},ak:  [-1.    -2.455  2.455  1.   ]
b0,{\ldots},bk:  [0.273 2.455 2.455 0.273]

k=4
a0,{\ldots},ak:  [-1.  -6.4 -0.   6.4  1. ]
b0,{\ldots},bk:  [0.24 3.84 8.64 3.84 0.24]

k=5
a0,{\ldots},ak:  [ -1.    -11.861 -14.599  14.599  11.861   1.   ]
b0,{\ldots},bk:  [ 0.219  5.474 21.898 21.898  5.474  0.219]
```

Calcoliamo i coefficienti dei metodi espliciti dei primi ordini:

```
for k in range(1,6):
    c = LMF(k, type='explicit')
    print('k=%s' % k)
    print('a0,...,ak: ', c[0:k+1])
    print('b0,...,bk: ', c[k+1:])
    print()
```

```
k=1
a0,{\ldots},ak:  [-1.  1.]
b0,{\ldots},bk:  [1. 0.]

k=2
a0,{\ldots},ak:  [-5.  4.  1.]
b0,{\ldots},bk:  [2. 4. 0.]

k=3
a0,{\ldots},ak:  [-10.  -9.  18.   1.]
b0,{\ldots},bk:  [ 3. 18.  9.  0.]

k=4
a0,{\ldots},ak:  [-15.667 -64.     36.     42.667   1.   ]
b0,{\ldots},bk:  [ 4. 48. 72. 16.  0.]

k=5
a0,{\ldots},ak:  [ -21.833 -191.667 -100.     233.333   79.167    1.   ]
b0,{\ldots},bk:  [  5. 100. 300. 200.  25.   0.]
```

Convergenza e Stabilità

Abbiamo visto che la soluzione discreta y_n e la soluzione esatta $\hat{y}_n = \hat{y}(t_n)$ soddisfano rispettivamente:

$$\sum_{i=0}^{k} \alpha_i y_{n+i} - h\sum_{i=0}^{k} \beta_i f_{n+i} = 0, \qquad \sum_{i=0}^{k} \alpha_i \hat{y}_{n+i} - h\sum_{i=0}^{k} \beta_i \hat{f}_{n+i} \equiv \tau_{n+k} = O(h^{p+1})$$

da cui ha senso richiedere che, al limite per $h \to 0$, per la differenza membro a membro delle precedenti valga:

$$\sum_{i=0}^{k} \alpha_i (y_{n+i} - \hat{y}_{n+i}) - h\sum_{i=0}^{k} \beta_i (f_{n+i} - \hat{f}_{n+i}) = \sum_{i=0}^{k} \alpha_i e_{n+i} - h\sum_{i=0}^{k} \beta_i (f_{n+i} - \hat{f}_{n+i}) = -\tau_{n+k} \to_{h\to 0} 0$$

Se le condizioni di consistenza sono imposte, in quanto $\lim_{n\to\infty} \tau_{n+k} = 0$, ha senso studiare:

$$\sum_{i=0}^{k} \alpha_i e_{n+i} = \rho(E) e_n = 0$$

Quest'ultima è un'equazione alle differenze a coefficienti costanti che ha $\rho(z)$ come polinomio caratteristico, e pertanto è asintoticamente stabile se ρ è un polinomio di Schur, stabile se è di VonNeumann, e instabile altrimenti.
Ricordiamo che le condizioni di ordine (13.4) [p.35] includono sempre $\sum_{i=0}^{k} \alpha_i = 0$, il che equivale a richiedere $\rho(1) = 0$ ovvero $\rho(z)$ non può essere un polinomio di Schur se il metodo è consistente.

Definizione DEF13.1: $0-$stabilità
Un metodo LMF si dice $0-$stabile se $\rho(z)$ è un polinomio di Von Neumann.

Definizione DEF13.2: Convergenza
Un metodo LMF si dice *convergente* se vale:

$$\lim_{N\to\infty} \max_{n} |e_n| = 0$$

Si può dimostrare che se un metodo LMF è consistente ($p \geq 1$) e $0-$stabile, ovvero se $\tau_n = O(h^{p+1})$ per $n = k, ..., N$ con $\rho(z)$ di Von Neumann, e se le condizioni iniziali $y_0, ..., y_k$ sono tali che $|e_i| = O(h^r)$, $i = 0, ..., k$ allora il metodo è convergente.

Teorema TH13.1: Prima barriera di Dahlquist
L'ordine massimo di un metodo LMF a k passi $0-$stabile e' $k+1$ se k è dispari, e $k+2$ se k è pari.

Metodi di Adams

Questa classe di metodi hanno polinomio caratteristico $\rho(z) = z^{k-1}(z-1) = z^k - z^{k-1}$. Essendo le uniche radici 0 con molteplicità $k-1$ e 1 con molteplicità unitaria, $\rho(z)$ è di Von Neumann ed i metodi sono 0-stabili e consistenti, e quindi anche convergenti.

Adams-Bashforth: metodi di Adams espliciti

Imponiamo le condizioni implicate da $\rho(z)$: $\alpha_k = -\alpha_{k-1} = 1$, $\alpha_{k-2} = ... = \alpha_0 = 0$ ed in quanto il metodo è esplicito, anche $\beta_k = 0$. Il sistema (13.5) [p.35]:

$$\begin{pmatrix} 1 & \cdots & 1 & 0 & \cdots & 0 \\ 0^1 & \cdots & k^1 & -1\cdot 0^0 & \cdots & -1\cdot k^0 \\ 0^2 & \cdots & k^2 & -2\cdot 0^1 & \cdots & -2\cdot k^1 \\ 0^3 & \cdots & k^3 & -3\cdot 0^2 & \cdots & -3\cdot k^2 \\ & \vdots & & & \vdots & \\ 0^p & \cdots & k^p & -p\cdot 0^{p-1} & \cdots & -p\cdot k^{p-1} \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_k \\ \beta_0 \\ \vdots \\ \beta_k \end{pmatrix} = \mathbf{0}$$

si riduce a:

$$\begin{pmatrix} -1\cdot 0^0 & \cdots & -1\cdot(k-1)^0 \\ -2\cdot 0^1 & \cdots & -2\cdot(k-1)^1 \\ -3\cdot 0^2 & \cdots & -3\cdot(k-1)^2 \\ & \vdots & \\ -p\cdot 0^{p-1} & \cdots & -p\cdot(k-1)^{p-1} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_{k-1} \end{pmatrix} = \begin{pmatrix} -k^1+(k-1)^1 \\ \vdots \\ -k^p+(k-1)^p \end{pmatrix}$$

in cui è evidente anche che l'ordine massimo $p = k$. Ogni riga del sistema appena descritto corrisponde all'equazione:

$$\sum_{i=0}^{k-1} -ji^{(j-1)}\beta_i = -k^j + (k-1)^j, \quad j = 1, ..., k$$

ovvero:

$$\sum_{i=0}^{k-1} i^{(j-1)}\beta_i = \frac{k^j-(k-1)^j}{j}, \quad j = 1, ..., k \tag{13.6}$$

Adams-Moulton: metodi di Adams impliciti

Imponiamo le stesse condizioni viste per i metodi espliciti: $\alpha_k = -\alpha_{k-1} = 1$, $\alpha_0, ..., \alpha_{k-2} = 0$ ma in quanto il metodo è implicito, lasciamo libero il parametro β_k. Il sistema quindi diventa:

$$\begin{pmatrix} -1\cdot 0^0 & \cdots & -1\cdot k^0 \\ -2\cdot 0^1 & \cdots & -2\cdot k^1 \\ -3\cdot 0^2 & \cdots & -3\cdot k^2 \\ & \vdots & \\ -p\cdot 0^{p-1} & \cdots & -p\cdot k^{p-1} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_k \end{pmatrix} = \begin{pmatrix} -k^1+(k-1)^1 \\ \vdots \\ -k^p+(k-1)^p \end{pmatrix}$$

in cui è evidente anche che l'ordine massimo $p = k+1$. Ogni riga del sistema appena descritto corrisponde all'equazione:

$$\sum_{i=0}^{k} -ji^{(j-1)}\beta_i = -k^j + (k-1)^j, \quad j = 1, ..., k$$

ovvero:

$$\sum_{i=0}^{k} i^{(j-1)}\beta_i = \frac{k^j-(k-1)^j}{j}, \quad j = 1, ..., k+1 \tag{13.7}$$

Calcoliamo i coefficienti dei metodi di Adams, impliciti ed espliciti implementando i sistemi (13.6) e (13.7) (che differiscono soltanto per il limite superiore).
Oltre a calcolare i coefficienti numericamente, li calcoliamo usando il sistema CAS Sage per preservare le frazioni e mostrare i coefficienti esatti, utili a fini didattici.

```
import numpy as np

def LMF_Adams(k, type='implicit'):
    if type=='implicit':
        p = k+1
    elif type=='explicit':
        p = k
    else:
        raise Exception('Type must be either implicit or explicit')

    # Numerical solution using numpy
```

```
    M = np.vander(range(0,p), increasing=True).transpose()
    b = np.array([(k**j-(k-1)**j)/j for j in range(1,p+1)])
    c = np.linalg.solve(M,b)

    # c = [a_0,...,a_k,b_0,...,b_k]
    c = np.concatenate(([0]*(k-1)+[-1,1],c,[0] if type == 'explicit' else []))

    # Rational solution using sage
    A = matrix(M)
    v = vector([(k**j-(k-1)**j)/j for j in range(1,p+1)])
    c_frac = A.solve_right(v)

    return c,c_frac
```

```
np.set_printoptions(precision=3, suppress=True)
for k in range(1,7):
    c,cf = LMF_Adams(k, 'explicit')
    print('k=%s:' % k)
    print('a0,...,ak: ', c[0:k+1])
    print('b0,...,bk: ', c[k+1:])
    print(cf)
    print()
```

```
k=1:
a0,{\ldots},ak:  [-1.  1.]
b0,{\ldots},bk:  [1. 0.]
(1)

k=2:
a0,{\ldots},ak:  [ 0. -1.  1.]
b0,{\ldots},bk:  [-0.5  1.5  0. ]
(-1/2, 3/2)

k=3:
a0,{\ldots},ak:  [ 0.  0. -1.  1.]
b0,{\ldots},bk:  [ 0.417 -1.333  1.917  0.   ]
(5/12, -4/3, 23/12)

k=4:
a0,{\ldots},ak:  [ 0.  0.  0. -1.  1.]
b0,{\ldots},bk:  [-0.375  1.542 -2.458  2.292  0.   ]
(-3/8, 37/24, -59/24, 55/24)

k=5:
a0,{\ldots},ak:  [ 0.  0.  0.  0. -1.  1.]
b0,{\ldots},bk:  [ 0.349 -1.769  3.633 -3.853  2.64   0.   ]
(251/720, -637/360, 109/30, -1387/360, 1901/720)

k=6:
a0,{\ldots},ak:  [ 0.  0.  0.  0.  0. -1.  1.]
b0,{\ldots},bk:  [-0.33   1.998 -5.068  6.932 -5.502  2.97   0.   ]
(-95/288, 959/480, -3649/720, 4991/720, -2641/480, 4277/1440)
```

```
for k in range(1,7):
    c,cf = LMF_Adams(k, 'implicit')
    print('k=%s:' % k)
    print('a0,...,ak: ', c[0:k+1])
    print('b0,...,bk: ', c[k+1:])
    print(cf)
```

```
7       print()
8
```

```
k=1:
a0,{\ldots},ak:  [-1.  1.]
b0,{\ldots},bk:  [0.5 0.5]
(1/2, 1/2)

k=2:
a0,{\ldots},ak:  [ 0. -1.  1.]
b0,{\ldots},bk:  [-0.083  0.667  0.417]
(-1/12, 2/3, 5/12)

k=3:
a0,{\ldots},ak:  [ 0.  0. -1.  1.]
b0,{\ldots},bk:  [ 0.042 -0.208  0.792  0.375]
(1/24, -5/24, 19/24, 3/8)

k=4:
a0,{\ldots},ak:  [ 0.  0.  0. -1.  1.]
b0,{\ldots},bk:  [-0.026  0.147 -0.367  0.897  0.349]
(-19/720, 53/360, -11/30, 323/360, 251/720)

k=5:
a0,{\ldots},ak:  [ 0.  0.  0.  0. -1.  1.]
b0,{\ldots},bk:  [ 0.019 -0.12   0.335 -0.554  0.991  0.33 ]
(3/160, -173/1440, 241/720, -133/240, 1427/1440, 95/288)

k=6:
a0,{\ldots},ak:  [ 0.  0.  0.  0.  0. -1.  1.]
b0,{\ldots},bk:  [-0.014  0.104 -0.334  0.62  -0.768  1.077  0.316]
(-863/60480, 263/2520, -6737/20160, 586/945, -15487/20160, 2713/2520, 19087/60480)
```

Formule di Newton-Cotes

Questa classe di metodi sono del tipo:

$$y_{n+k} - y_n = h \sum_{i=0}^{k} \beta_i f_{n+i}$$

ovvero il polinomio caratteristico è $\rho(z) = z^k - 1$ che è di Von Neumann in quanto, applicando il secondo criterio di Schur:

$$p(z) = \rho(z) = \sum_{i=0}^{k} p_i z^{k-i} = z^k - 1, \qquad q(z) = \sum_{i=0}^{k} \bar{p}_i z^i = z^k - 1$$

in quanto $\bar{1} = 1$ e quindi:

$$p^{(1)}(z) = \frac{q(0)p(z) - p(0)q(z)}{z} = \frac{1(z^k - 1) - 1(z^k - 1)}{z} \equiv 0$$

e $p'(z) = kz^{k-1}$ è di Schur in quanto ha solo radici in 0.

Imponendo $\alpha_k = -\alpha_0 = 1$, $\alpha_1, ..., \alpha_{k-1} = 0$, il sistema (13.5) [p.35] diventa:

$$\begin{pmatrix} -1\cdot 0^0 & \cdots & -1\cdot k^0 \\ -2\cdot 0^1 & \cdots & -2\cdot k^1 \\ -3\cdot 0^2 & \cdots & -3\cdot k^2 \\ & \vdots & \\ -p\cdot 0^{p-1} & \cdots & -p\cdot k^{p-1} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_k \end{pmatrix} = \begin{pmatrix} -k^1 \\ \vdots \\ -k^p \end{pmatrix}$$

ovvero

$$\begin{pmatrix} 0^0 & \cdots & k^0 \\ 0^1 & \cdots & k^1 \\ 0^2 & \cdots & k^2 \\ & \vdots & \\ 0^{p-1} & \cdots & k^{p-1} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_k \end{pmatrix} = \begin{pmatrix} k^1/1 \\ \vdots \\ k^p/p \end{pmatrix}$$

e quindi di ordine $p = k + 1$ nel caso implicito, e:

$$\begin{pmatrix} 0^0 & \cdots & (k-1)^0 \\ 0^1 & \cdots & (k-1)^1 \\ 0^2 & \cdots & (k-1)^2 \\ & \vdots & \\ 0^{p-1} & \cdots & (k-1)^{p-1} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_{k-1} \end{pmatrix} = \begin{pmatrix} k^1/1 \\ \vdots \\ k^p/p \end{pmatrix}$$

e quindi di ordine $p = k$ nel caso esplicito.

```python
import numpy as np

def LMF_NewtonCotes(k, type='implicit'):
    if type=='implicit':
        p = k+1
    elif type=='explicit':
        p = k
    else:
        raise Exception('Type must be either implicit or explicit')

    # Numerical solution using numpy
    M = np.vander(range(0,p), increasing=True).transpose()
    b = np.array([k**j/j for j in range(1,p+1)])
    c = np.linalg.solve(M,b)

    # c = [a_0,...,a_k,b_0,...,b_k]
    c = np.concatenate(([-1]+[0]*(k-1)+[1],c,[0] if type == 'explicit' else []))

    # Rational solution using sage
    A = matrix(M)
    v = vector([QQ(k**j/j) for j in range(1,p+1)])
    c_frac = A.solve_right(v)

    return c,c_frac
```

```
np.set_printoptions(precision=3, suppress=True)
for k in range(1,7):
    c,cf = LMF_NewtonCotes(k, 'implicit')
    print('k=%s:' % k)
    print('a0,...,ak: ', c[0:k+1])
    print('b0,...,bk: ', c[k+1:])
    print(cf)
    print()
```

```
k=1:
a0,{\ldots},ak:  [-1.  1.]
b0,{\ldots},bk:  [0.5 0.5]
(1/2, 1/2)

k=2:
a0,{\ldots},ak:  [-1.  0.  1.]
b0,{\ldots},bk:  [0.333 1.333 0.333]
(1/3, 4/3, 1/3)

k=3:
a0,{\ldots},ak:  [-1.  0.  0.  1.]
b0,{\ldots},bk:  [0.375 1.125 1.125 0.375]
(3/8, 9/8, 9/8, 3/8)

k=4:
a0,{\ldots},ak:  [-1.  0.  0.  0.  1.]
b0,{\ldots},bk:  [0.311 1.422 0.533 1.422 0.311]
(14/45, 64/45, 8/15, 64/45, 14/45)

k=5:
a0,{\ldots},ak:  [-1.  0.  0.  0.  0.  1.]
b0,{\ldots},bk:  [0.33  1.302 0.868 0.868 1.302 0.33 ]
(95/288, 125/96, 125/144, 125/144, 125/96, 95/288)

k=6:
a0,{\ldots},ak:  [-1.  0.  0.  0.  0.  0.  1.]
b0,{\ldots},bk:  [0.293 1.543 0.193 1.943 0.193 1.543 0.293]
(41/140, 54/35, 27/140, 68/35, 27/140, 54/35, 41/140)
```

```
for k in range(1,7):
    c,cf = LMF_NewtonCotes(k, 'explicit')
    print('k=%s:' % k)
    print('a0,...,ak: ', c[0:k+1])
    print('b0,...,bk: ', c[k+1:])
    print(cf)
    print()
```

```
k=1:
a0,{\ldots},ak:  [-1.  1.]
b0,{\ldots},bk:  [1. 0.]
(1)

k=2:
a0,{\ldots},ak:  [-1.  0.  1.]
b0,{\ldots},bk:  [0. 2. 0.]
(0, 2)

k=3:
a0,{\ldots},ak:  [-1.  0.  0.  1.]
b0,{\ldots},bk:  [0.75 0.   2.25 0.  ]
(3/4, 0, 9/4)
```

```
k=4:
a0,{\ldots},ak:  [-1.  0.  0.  0.  1.]
b0,{\ldots},bk:  [ 0.     2.667 -1.333  2.667  0.   ]
(0, 8/3, -4/3, 8/3)

k=5:
a0,{\ldots},ak:  [-1.  0.  0.  0.  0.  1.]
b0,{\ldots},bk:  [ 0.66  -0.347  4.167 -2.431  2.951  0.   ]
(95/144, -25/72, 25/6, -175/72, 425/144)

k=6:
a0,{\ldots},ak:  [-1.  0.  0.  0.  0.  0.  1.]
b0,{\ldots},bk:  [-0.   3.3 -4.2  7.8 -4.2  3.3  0. ]
(0, 33/10, -21/5, 39/5, -21/5, 33/10)
```

Backwards Differentiation Formulae

Questa classe di metodi, al contrario di quelli visti precedentemente, impone che $\sigma(z) = z^k$, ovvero:

$$\sum_{i=0}^{k} \alpha_i y_{n+i} = h f_{n+k}$$

ed i metodi risulsanti sono ovviamente tutti impliciti. Imponendo $\beta_k = 1$, $\beta_0, ..., \beta_{k-1} = 0$ il sistema (13.5) [p.35] diventa:

$$\begin{pmatrix} 1 & \cdots & 1 \\ 0^1 & \cdots & k^1 \\ 0^2 & \cdots & k^2 \\ 0^3 & \cdots & k^3 \\ & \vdots & \\ 0^p & \cdots & k^p \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_k \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \cdot k^0 \\ 2 \cdot k^1 \\ \vdots \\ p \cdot k^{p-1} \end{pmatrix}$$

da cui deduciamo che anche in questo caso, $p = k$. Calcoliamo i coefficienti:

```python
import numpy as np

def LMF_BDF(k):
    p = k+1

    # Numerical solution using numpy
    M = np.vander(range(0,p), increasing=True).transpose()
    b = np.array([0]+[j*k**(j-1) for j in range(1,p)])
    c = np.linalg.solve(M,b)

    # c = [a_0,...,a_k,b_0,...,b_k]
    c = np.concatenate((c,[0]*(k),[1]))

    # Rational solution using sage
    A = matrix(M)
    v = vector([0]+[j*k**(j-1) for j in range(1,p)])
    c_frac = A.solve_right(v)

    return c,c_frac
```

```python
np.set_printoptions(precision=3, suppress=True)
for k in range(1,10):
    c,cf = LMF_BDF(k)
    print('k=%s:' % k)
    print('a0,...,ak: ', c[0:k+1])
    print(cf)
    print('b0,...,bk: ', c[k+1:])
    print()
```

```
k=1:
a0,{\ldots},ak:  [-1.  1.]
(-1, 1)
b0,{\ldots},bk:  [0. 1.]

k=2:
a0,{\ldots},ak:  [ 0.5 -2.   1.5]
(1/2, -2, 3/2)
b0,{\ldots},bk:  [0. 0. 1.]

k=3:
a0,{\ldots},ak:  [-0.333  1.5   -3.     1.833]
(-1/3, 3/2, -3, 11/6)
b0,{\ldots},bk:  [0. 0. 0. 1.]

k=4:
a0,{\ldots},ak:  [ 0.25  -1.333  3.    -4.     2.083]
(1/4, -4/3, 3, -4, 25/12)
b0,{\ldots},bk:  [0. 0. 0. 0. 1.]

k=5:
a0,{\ldots},ak:  [-0.2    1.25  -3.333  5.    -5.     2.283]
(-1/5, 5/4, -10/3, 5, -5, 137/60)
b0,{\ldots},bk:  [0. 0. 0. 0. 0. 1.]

k=6:
a0,{\ldots},ak:  [ 0.167 -1.2    3.75  -6.667  7.5   -6.     2.45 ]
(1/6, -6/5, 15/4, -20/3, 15/2, -6, 49/20)
b0,{\ldots},bk:  [0. 0. 0. 0. 0. 0. 1.]

k=7:
a0,{\ldots},ak:  [ -0.143   1.167  -4.2     8.75  -11.667  10.5    -7.      2.593]
(-1/7, 7/6, -21/5, 35/4, -35/3, 21/2, -7, 363/140)
b0,{\ldots},bk:  [0. 0. 0. 0. 0. 0. 0. 1.]

k=8:
a0,{\ldots},ak:  [  0.125  -1.143   4.667 -11.2    17.5   -18.667  14.     -8.      2.718]
(1/8, -8/7, 14/3, -56/5, 35/2, -56/3, 14, -8, 761/280)
b0,{\ldots},bk:  [0. 0. 0. 0. 0. 0. 0. 0. 1.]

k=9:
a0,{\ldots},ak:  [ -0.111   1.125  -5.143  14.    -25.2    31.5   -28.     18.     -9.
   2.829]
(-1/9, 9/8, -36/7, 14, -126/5, 63/2, -28, 18, -9, 7129/2520)
b0,{\ldots},bk:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

Implementiamo i criteri di Schur per decidere se $p(z) = \sum_i^k p_i z^{k-i}$ sia di Schur o di Von Neumann.

Calcoliamo il polinomio ridotto usando la formula:

$$p^{(1)}(z) = \sum_{i=0}^{k-1} (\bar{p_0} p_i - p_k \bar{p_{k-i}}) z^{k-1-i}$$

e quindi implementiamo ricorsivamente il primo criterio di Schur, per cui $p(z)$ è di Schur se $|p_0| > |p_k|$ e $p^{(1)}(z)$ è di Schur, ed il secondo criterio per cui $p(z)$ è di VonNeumann se almeno una delle seguenti vale:

- $|p_0| > |p_k|$ e $p^{(1)}(z)$ è di VonNeumann
- $p^{(1)}(z) \equiv 0$ e $p'(z)$ è Schur

```
from numpy import conjugate as bar

def reduced_polynomial(c):
    # c such that p(z) = c[0]z^k+...+c[k]z^0 -> c[i] = bar(c[0])c[i] - c[k]bar(c[k-i])
    k = len(c)-1
    return [(bar(c[0])*c[i] - c[k]*bar(c[k-i]))  for i in range(0,k)]

def derivative_polynomial(c):
    #c such that p(z) = c[0]z^k + ... + c[k]z^0 -> c' = [k*c[0], (k-1)c[1], ...]
    k = len(c)-1
    return [(k-i)*c[i] for i in range(0,k)]

def check_schur(c):
    # c such that p(z) = c[0]z^k+...+c[k]z^0
    k = len(c)-1
    if k==0:
        return True

    if not abs(c[0])>abs(c[k]):
        return False

    return check_schur(reduced_polynomial(c))

def check_vonneumann(c):
    # c such that p(z) = c[0]z^k+...+c[k]z^0
    k = len(c)-1
    if k==0:
        return True
    first_crit = abs(c[0])>abs(c[k]) and check_vonneumann(reduced_polynomial(c))
    second_crit = not(any(reduced_polynomial(c))) and check_schur(derivative_polynomial(c))

    return first_crit or second_crit

```

Verifichiamo quali BDF hanno $\rho(z)$ di VonNeumann, ovvero quali siano $0-$stabili oltre ad essere consistenti (tutti lo sono per costruzione)

```
for k in range(1,11):
    c,cf = LMF_BDF(k)
    cf = cf[::-1]
    c = c[0:k+1][::-1]
    print('k = %s:\nrho =\n%s' % (k, '  +\n'.join( ['  '+str(cf[i])+' z^'+str(k-i) for i in range(k+1)] ) ))
    print('is VonNeumann: %s' % check_vonneumann(cf))
    print('is VonNeumann (numeric): %s' % check_vonneumann(c))
    print()
```

```
k = 1:
rho =
  1 z\^{}1  +
  -1 z\^{}0
is VonNeumann: True
is VonNeumann (numeric): True

k = 2:
rho =
  3/2 z\^{}2  +
```

```
  -2 z\^{}1  +
  1/2 z\^{}0
is VonNeumann: True
is VonNeumann (numeric): True

k = 3:
rho =
  11/6 z\^{}3  +
  -3 z\^{}2  +
  3/2 z\^{}1  +
  -1/3 z\^{}0
is VonNeumann: True
is VonNeumann (numeric): True

k = 4:
rho =
  25/12 z\^{}4  +
  -4 z\^{}3  +
  3 z\^{}2  +
  -4/3 z\^{}1  +
  1/4 z\^{}0
is VonNeumann: True
is VonNeumann (numeric): True

k = 5:
rho =
  137/60 z\^{}5  +
  -5 z\^{}4  +
  5 z\^{}3  +
  -10/3 z\^{}2  +
  5/4 z\^{}1  +
  -1/5 z\^{}0
is VonNeumann: True
is VonNeumann (numeric): True

k = 6:
rho =
  49/20 z\^{}6  +
  -6 z\^{}5  +
  15/2 z\^{}4  +
  -20/3 z\^{}3  +
  15/4 z\^{}2  +
  -6/5 z\^{}1  +
  1/6 z\^{}0
is VonNeumann: True
is VonNeumann (numeric): True

k = 7:
rho =
  363/140 z\^{}7  +
  -7 z\^{}6  +
  21/2 z\^{}5  +
  -35/3 z\^{}4  +
  35/4 z\^{}3  +
  -21/5 z\^{}2  +
  7/6 z\^{}1  +
  -1/7 z\^{}0
is VonNeumann: False
is VonNeumann (numeric): False

k = 8:
rho =
  761/280 z\^{}8  +
  -8 z\^{}7  +
  14 z\^{}6  +
```

```
  -56/3 z\^{}5  +
  35/2 z\^{}4  +
  -56/5 z\^{}3  +
  14/3 z\^{}2  +
  -8/7 z\^{}1  +
  1/8 z\^{}0
is VonNeumann: False
is VonNeumann (numeric): False

k = 9:
rho =
  7129/2520 z\^{}9  +
  -9 z\^{}8  +
  18 z\^{}7  +
  -28 z\^{}6  +
  63/2 z\^{}5  +
  -126/5 z\^{}4  +
  14 z\^{}3  +
  -36/7 z\^{}2  +
  9/8 z\^{}1  +
  -1/9 z\^{}0
is VonNeumann: False
is VonNeumann (numeric): False

k = 10:
rho =
  7381/2520 z\^{}10  +
  -10 z\^{}9  +
  45/2 z\^{}8  +
  -40 z\^{}7  +
  105/2 z\^{}6  +
  -252/5 z\^{}5  +
  35 z\^{}4  +
  -120/7 z\^{}3  +
  45/8 z\^{}2  +
  -10/9 z\^{}1  +
  1/10 z\^{}0
is VonNeumann: False
is VonNeumann (numeric): False
```

Le BDF con più di 6 passi non sono quindi 0-stabili.

Implementazione ed analisi del costo computazionale

Assumiamo di avere come parametri:

- il numero di passi del metodo k
- gli estremi della mesh t_0, T ed il numero di passi N
- il tipo e quindi i coefficienti del metodo α_i, β_i, $i = 0, ..., k$,
- la funzione $f(t, y(t)) = y'(t)$
- le k condizioni iniziali $y_{t_0}, ..., y_{t_{0+k-1}}$

assumiamo anche che $y(t)$ sia un vettore in $\mathbb{R}^s$, e quindi in generale:

$$y(t) \in \mathbb{R}^s, \quad f(t, y(t)) = \begin{pmatrix} f_1(t, y_1(t), ..., y_s(t)) \\ \vdots \\ f_s(t, y_1(t), ..., y_s(t)) \end{pmatrix} \in \mathbb{R}^s$$

Ad ogni passo del metodo calcoliamo y_{n+k} mediante la (13.2) [p.34]:

$$\alpha_k y_{n+k} - h\beta_k f(t_k, y_{n+k}) = -\sum_{i=0}^{k-1} \alpha_i y_{n+i} + h\sum_{i=0}^{k-1} \beta_i f(t_{n+i}, y_{n+i}), \quad n = 0, ..., N-k \tag{13.10}$$

in cui usiamo la notazione $n = t_0 + nh$ (e quindi $y_{n+k} = y(t_0 + nh)$) per brevità.
Se $\beta_k = 0$, ovvero il metodo è esplicito, il costo computazionale di ogni passo è costante: $O(k)$. In particolare, saranno necessarie ad ogni passo:

- $2k+1$ moltiplicazioni vettoriali: k nella prima sommatoria e k+1 nella seconda. Essendo tutte moltiplicazioni per una costanto, richiedono ogniuna s moltiplicazioni floating point.
- $2(k-1)+1 = 2k-1$ somme vettoriali: $k-1$ somme in ciascuna sommatoria più la somma finale; essendo somme tra vettori, richiedono s somme ogniuna.
- Assumendo un'implementazione che mantenga in memoria le ultime $k-1$ valutazioni di f, dovremo valutare f soltanto una volta su y_{n+k-1} ad ogni passo, fatto salvo il primo passo in cui ne dovremo calcolare $k-1$.

In totale abbiamo quindi $(N-k+1)(2k+1)s$ moltiplicazioni, $(N-k+1)(2k-1)s$ somme e $(N-k)+(k-1) = N-1$ valutazioni di funzioni.
Se invece $\beta_k \neq 0$, al costo appena menzionato si aggiunge il costo di risolvere l'equazione, generalmente non lineare:

$$y_{n+k} - h\beta_k f(t_{n+k}, y_{n+k}) - g_n = G_n(y_{n+k}) = 0 \tag{13.11}$$

in cui g_n è la parte destra della (13.10).
La soluzione $y = y_{n+k}$ della (13.11) esiste per h sufficientemente piccolo. Supponendo di essere in questa condizione, risolviamo numericamente l'equazione.

Usando il metodo di Newton, indicando con y^l il valore di y all'iterazione l-esima, calcoliamo y^{l+1} nello spazio $y, G(y)$ attraverso l'approssimazione:

$$G(y) = G'(y^l)(y - y^l) + G(y^l)$$

dalla quale, imponendo $G(y) = 0$ si ottiene:

$$G'_n(y^l)(y^{l+1} - y^l) = -G_n(y^l) \Longrightarrow y^{l+1} = y^l - [G'_n(y^l)]^{-1} G_n(y^l)$$

Chiamando $\delta^l = (y^{l+1} - y^l)$ il metodo consiste nel risolvere $G_n(y_{n+k}) = 0$ iterando su l:

$$\delta^l = -[G'_n(y^l_{n+k})]^{-1} G_n(y^l_{n+k})$$

$$y^{l+1}_{n+k} = y^l_{n+k} + \delta^l$$

in cui prendiamo come punto di partenza l'ultimo punto noto: $y^0_{n+k} = y_{n+k-1}$. Osserviamo però che questo metodo richiede l'inversione, e quindi la fattorizzazione di $G_n(y^l)$ ad ogni passo.

Per ovviare a questo problema possiamo usare invece il metodo delle corde, che procede come il metodo di Newton ma utilizza soltanto l'approssimazone della derivata nel punto di partenza senza mai ricalcolarla. Chiamando J^0_n lo Jacobiano di $f(t_{n+k}, y)$ calcolato nell'ultimo punto noto $y^0_{n+k} = y_{n+k-1}$, il metodo consiste ancora nel risolvere $G_n(y_{n+k}) = 0$ iterando su l:

$$G'_n(y^0_{n+k})\delta^l = (I - h\beta_k J^0_n)\delta^l = -G_n(y^l_{n+k}) \Longrightarrow \delta^l = -[(I - h\beta_k J^0_n)]^{-1} G_n(y^l_{n+k})$$

$$y^{l+1}_{n+k} = y^l_{n+k} + \delta^l$$

Come criterio di arresto, fissata una tolleranza tol, usiamo:

$$\|\delta^l ./ (1 + |y^l_{n+k}|)\| \leq \text{tol}$$

in cui ./ indica la divisione termine a termine. Osserviamo che la componente i-esima del vettore $\delta^l ./ (1 + |y^l_{n+k}|$, rappresenta un errore assoluto se $|y^l_{n+k}| \approx 0$, e un errore relativo tanto più $|y^l_{n+k}| >> 0$. I questo modo il controllo di tolleranza è concettualmente consistente rispetto al numero di cifre significativa della soluzione. Richiederemo anche che tol sia più stringete dell'accuratezza richiesta per il metodo LMF.

Raggruppiamo in un'unica funzione quanto visto fin'ora per calcolare i coefficienti di un generico metodo LMF dati il numero di passi, il tipo di formula desiderata (Adams, BDF, NewtonCotes) e, quando rilevante, specificando se il metodo debba essere implicito o esplicito:

```python
import numpy as np

def LMF_coeffs(k, method='Adams', kind='explicit'):
    """
    Given the number of steps k, the method ('Adams', 'NewtonCotes', 'BDF')
    and the kind ('implicit', 'explicit'),
    returns the coefficients vector [a_0,...,a_k,b_0,...,b_k] and the order p of the method
    """
    assert method in ['Adams', 'NewtonCotes', 'BDF'], 'Wrong method selected'
    assert kind in ['implicit', 'explicit'], 'Methods can only be implicit or explicit'

    if kind == 'implicit' or method == 'BDF':
        p = k+1
    else:
        p = k

    M = np.vander(range(0,p), increasing=True).transpose()

    if method == 'Adams':
        b = np.array([(k**j-(k-1)**j)/j for j in range(1,p+1)])
        c = np.linalg.solve(M,b)
        c = np.concatenate(([0]*(k-1)+[-1,1],c,[0] if kind == 'explicit' else []))

    elif method == 'NewtonCotes':
        b = np.array([k**j/j for j in range(1,p+1)])
        c = np.linalg.solve(M,b)
        c = np.concatenate(([-1]+[0]*(k-1)+[1],c,[0] if kind == 'explicit' else []))

    elif method == 'BDF':
        b = np.array([0]+[j*k**(j-1) for j in range(1,p)])
        c = np.linalg.solve(M,b)
        c = np.concatenate((c,[0]*(k),[1]))

    return c,p


```

Avremo bisogno di calcolare lo Jacobiano di $f(t,y)$ rispetto al vettore y. Supponendo $f(t,y) : (\mathbb{R},\mathbb{R}^m) \to \mathbb{R}^m$, definiamo lo Jacobiano rispetto a y come:

$$J(t) = \left(\frac{\partial f(t,y)}{\partial y_1}, \cdots, \frac{\partial f(t,y)}{\partial y_n}\right)$$

e, dato un incremento h, lo approssimiamo numericamente con:

$$J(t,h) = \left((f(t,(x_1+h,...,x_m)^T) - f(t,(x_1,...,x_m)^T))./h, \cdots, (f(t,(x_1,...,x_m+h)^T) - f(t,(x_1,...,x_m))^T)./h\right)$$

```python
eps = np.finfo(float).eps
def num_jacobian(f, t0, y0, h=eps*2):
    m = len(y0)
    Jf = np.zeros([m,m])
    h_inv = 1./h
    for j in range(m):
        y1 = np.array(y0)
        y1[j] += h
        Jf[:,j] = (f(t0,y1)-f(t0,y0))*h_inv
    return Jf
```

Implementiamo adesso il metodo LMF.

Il problema di Cauchy prevede una sola condizione iniziale, ma per avviare il metodo ne servono k: $y_0, ..., y_{k-1}$. In questa implementazione le condizioni iniziali possono essere fornite al metodo dall'esterno, oppure se viene fornito soltanto y_0, usiamo

una stima delle condizioni iniziali calcolandole iterativamente sulla mesh a partire y_0 e $f(t, y(t))$ e usando il passo h implicato dai parametri del metodo secondo la formula:

$$y_{n+1} = hf(t_n, y_n) + y_n, \quad n = 0, ..., k-2$$

I $t_n, ..., t_{n+k}$; $y_n, ..., y_{n+k-1}$; $f_n, ..., f_{n+k-1}$ sono tenuti ciascuno in un buffer di k elementi che scorre in avanti di uno ad ogni ciclo. Se il metodo è implicito, applichiamo il metodo delle corde come descritto prima per risolvere l'equazione.

La tolleranza per la soluzione dell'equazione implicita in questa implementazione è scelta compatibile con l'ordine del metodo, ovvero h^{p+1}; in caso si forniscano condizioni iniziali con un approssimazione peggiore, potrebbe essere conveniente aumentare la tolleranza per migliorare le prestazioni del metodo.

```
def LMF(f,t0,T,N,Y0,k=2,method='Adams',kind='explicit',maxiters=500,tol=False):
    """
    f: (t,y)->R^s
    t0,T: beginning and end of integration interval
    N: number of steps
    Y0: a list of initial k initial conditions, or a list containing just one vector
        (in such case the others will be estimated using f)
    k: method steps
    method: method name ('Adams', 'NewtonCotes', 'BDF')
    kind: ('implicit', 'explicit')
    maxiters: implicit solver max iterations
    tol: error tolerance for implicit solver guard condition

    returns:
    sequence Y0,Y_1,...,Y_(N)
    """

    h = (T-t0)/float(N) # Step size
    s = len(Y0[0]) # Input vector size

    c,p = LMF_coeffs(k,method,kind)


    if not tol:
        tol = h**(p+1)

    a = c[0:k+1] #alpha coefficients
    b = c[k+1:] #beta coefficients

    Yn = np.zeros([k,s]) # sliding window on Y_n,...,Y_n+k-1
    Fn = np.zeros([k,s]) # sliding window on F_n,...,F_n+k-1 to avoid recomputations
    t = np.array([t0+(n*h) for n in range(k+1)]) #sliding window on t_n,...,t_n+k

    Yn[0] = Y0[0] # Initial condition

    if len(Y0) == 1:
        # We were provided only 1 initial conditions vector.
        # Estimate the remaining initial conditions starting from Y0 and using f(t,y(t))
        for n in range(0,k-1):
            Fn[n] = f(t[n], Yn[n])
            Yn[n+1] = h * Fn[n] + Yn[n]
        Fn[k-1] = f(t[k-1], Yn[k-1])
    else:
        for n in range(0,k):
            Yn[n] = Y0[n]
            Fn[n] = f(t[n], Yn[n])

    #results will hold the couples (t_n, y_n), n=0,...,N
    results = [(t[i],np.array(Yn[i])) for i in range(0,k)]

    for n in range(0,N-k+1): #n = 0,...,N-k

        Ysum = sum(a[i]*Yn[i] for i in range(0,k))
```

```
        Bsum = sum(b[i]*Fn[i] for i in range(0,k))

        Y =  (-Ysum + h*Bsum)/a[k]

        # If the method is explicit Y is already Y_n+k, otherwise it's g_n and we need to solve the equation
        if b[k] != 0:
            Yl = Yn[-1] #We use the last known value as starting point
            gn = Y
            Gn = lambda yl: yl - h*b[k]*f(t[-1],yl)/a[k] - gn #Gn(yl) will be computed at avery iteration
            Jn0 = num_jacobian(f, t[-1], Yl, h)
            M = np.linalg.inv(np.identity(s)-h*b[k]*Jn0) #The (I-hb[k]Jn0)^-1 can be computed only once
            dl = -M.dot(Gn(Yl))
            iter_count = 0
            while (np.linalg.norm(dl / (1+abs(Yl))) > tol and iter_count < maxiters) or iter_count == 0:
                Yl = Yl+dl
                dl = -M.dot(Gn(Yl))
                iter_count+=1
            Y = Yl

        Yn[0:-1] = Yn[1:]
        Yn[k-1] = Y
        Fn[0:-1] = Fn[1:]
        Fn[k-1] = f(t[k], Y)
        results.append((t[k],Y))

        #t is t_n,...,t_n+k, and here becomes t_n+1,...,t_n+1+k
        t[0:-1] = t[1:]
        t[-1] = t0+(n+1+k)*h

    return results
```

Confrontiamo il comportamento di alcuni metodi sull'equazione logistica:

$$y'(t) = 10y(t) - 2y^2(t), \quad y(0) = 1, \quad t \geq 0$$

che ha soluzione esplicita:

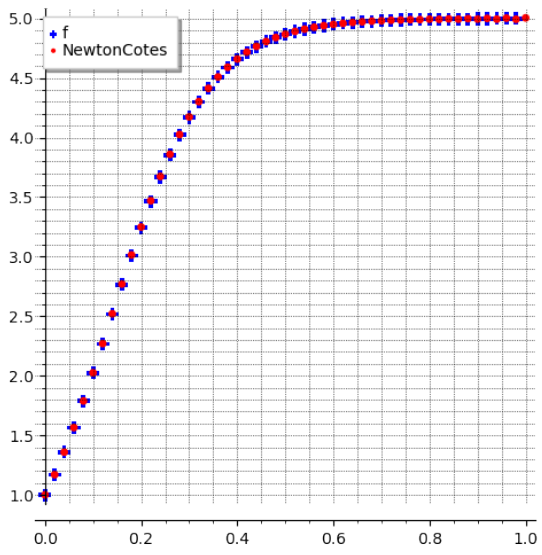
$$y(t) = \frac{5}{1 + 4e^{-10t}}$$

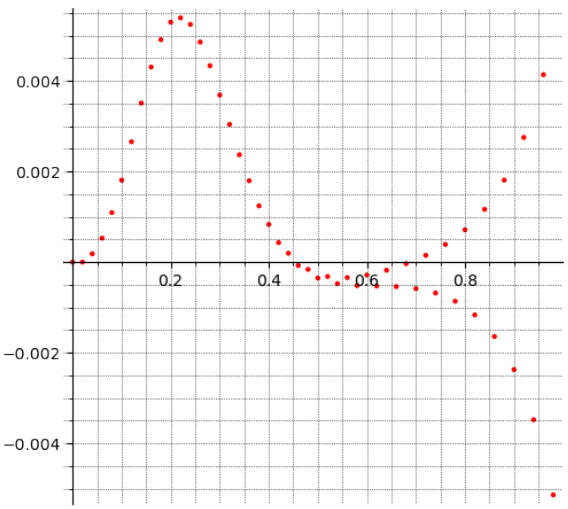
Per prima cosa, osserviamo il comportamento sull'equazione logistica del metodo mid-point (NewtonCotes di ordine 2) e dei metodi di Adams-Bashforth a $1, 2, 3$ passi.

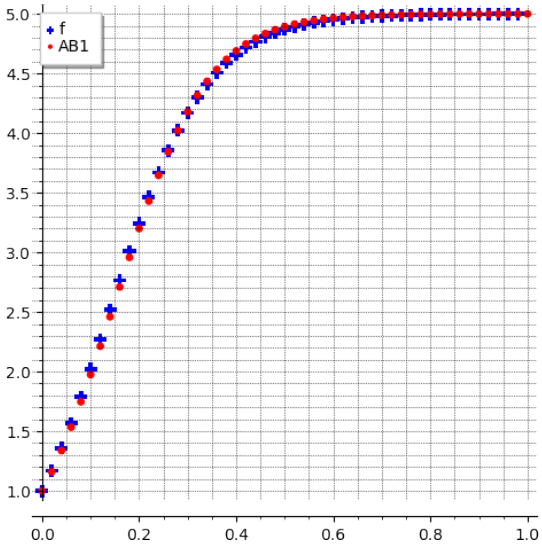
Osserviamo che l'errore del metodo mid-point ad un certo punto inizia ad oscillare; discuteremo questo fatto in seguito.

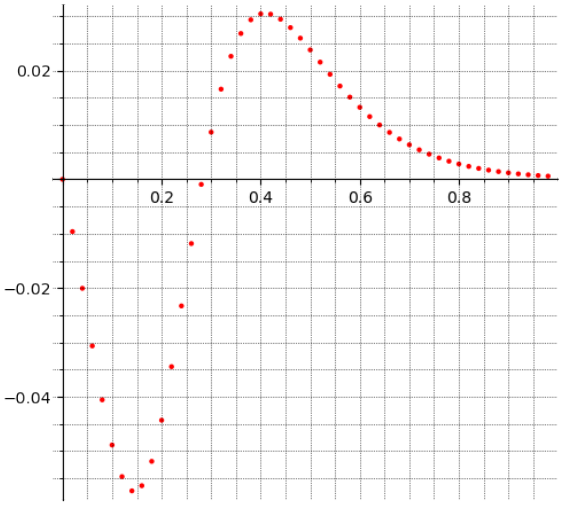
```
%run LMF.ipynb
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]=10,5


from collections import defaultdict
res = defaultdict(dict)


t0 = 0.
t1 = 1.

f = lambda t,y: 10*y-2*y**2 #f'(t)
ft = lambda t: float(5./(1+4*np.exp(-10*t))) #true value, f(t)
ic = lambda steps: list(np.array([ft(n*(1/steps))]) for n in range(5)) #initial conditions
steps = 50


res['NewtonCotes']['method'] = LMF(f, t0, t1, steps, ic(steps),k=2, method='NewtonCotes', kind='explicit')
d = res['NewtonCotes']['method']
```

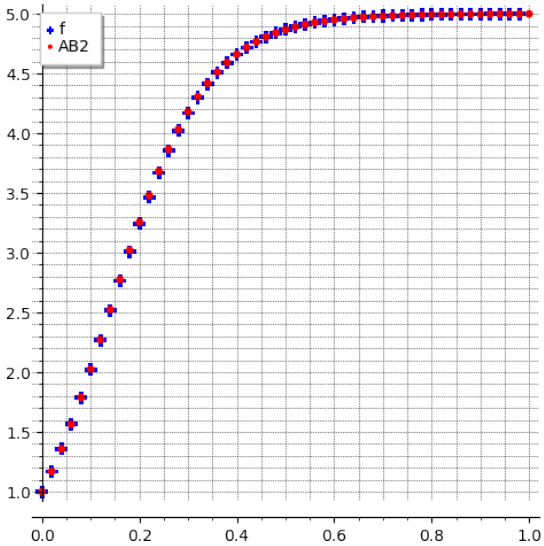
```
res['NewtonCotes']['error'] = [(d[i][0], (d[i][1]-ft(d[i][0]))) for i in range(steps)]

res['AB1']['method'] = LMF(f, t0, t1, steps, ic(steps),k=1, method='Adams', kind='explicit')
d = res['AB1']['method']
res['AB1']['error'] = [(d[i][0], (d[i][1]-ft(d[i][0]))) for i in range(steps)]

res['AB2']['method'] = LMF(f, t0, t1, steps, ic(steps),k=2, method='Adams', kind='explicit')
d = res['AB2']['method']
res['AB2']['error'] = [(d[i][0], (d[i][1]-ft(d[i][0]))) for i in range(steps)]

res['AB3']['method'] = LMF(f, t0, t1, steps, ic(steps),k=3, method='Adams', kind='explicit')
d = res['AB3']['method']
res['AB3']['error'] = [(d[i][0], (d[i][1]-ft(d[i][0]))) for i in range(steps)]

res['BDF1']['method'] = LMF(f, t0, t1, steps, ic(steps),k=1, method='BDF', kind='explicit')
d = res['BDF1']['method']
res['BDF1']['error'] = [(d[i][0], (d[i][1]-ft(d[i][0]))) for i in range(steps)]

res['BDF2']['method'] = LMF(f, t0, t1, steps, ic(steps),k=2, method='BDF', kind='explicit')
d = res['BDF2']['method']
res['BDF2']['error'] = [(d[i][0], (d[i][1]-ft(d[i][0]))) for i in range(steps)]

res['BDF3']['method'] = LMF(f, t0, t1, steps, ic(steps),k=3, method='BDF', kind='explicit')
d = res['BDF3']['method']
res['BDF3']['error'] = [(d[i][0], (d[i][1]-ft(d[i][0]))) for i in range(steps)]


for key,v in res.items():
    p = list_plot([(x,ft(x)) for x in np.arange(t0,t1,1./steps)], color='blue', marker='P',
                  size=64, legend_label='f', gridlines='minor')
    p += list_plot(res[key]['method'], color='red', legend_label=key, size=24)
    e = list_plot(res[key]['error'], color='red', gridlines='minor')
    g = graphics_array([p,e])
    g.show()

```

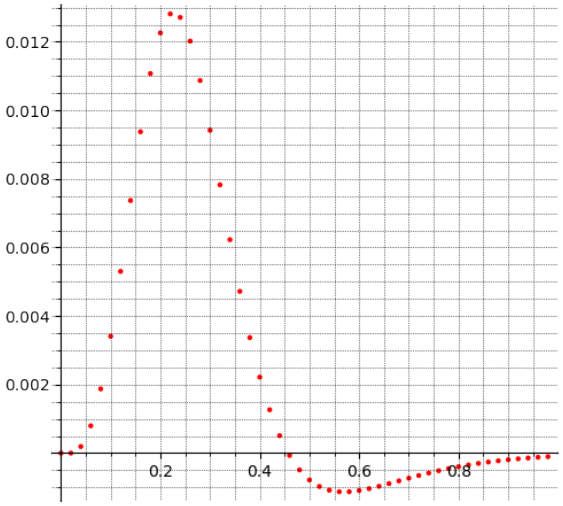


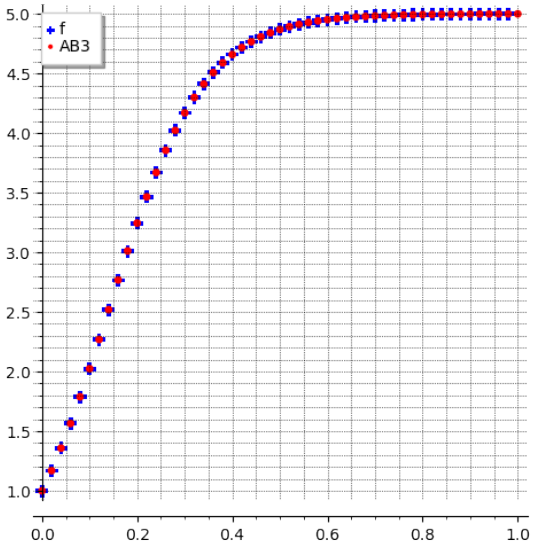


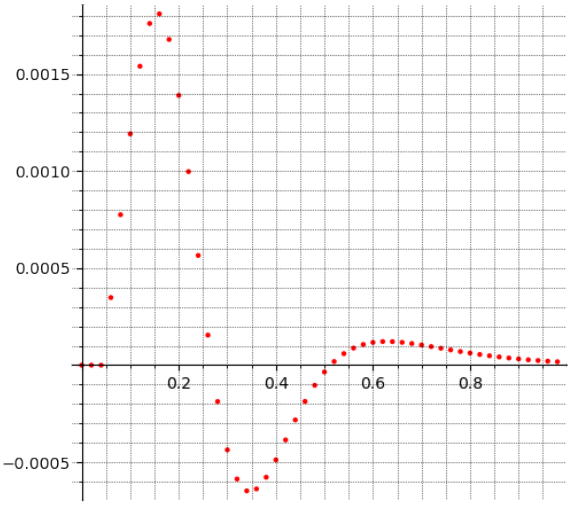

f
AB1
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0
0.2
0.4
0.6
0.8
1.0

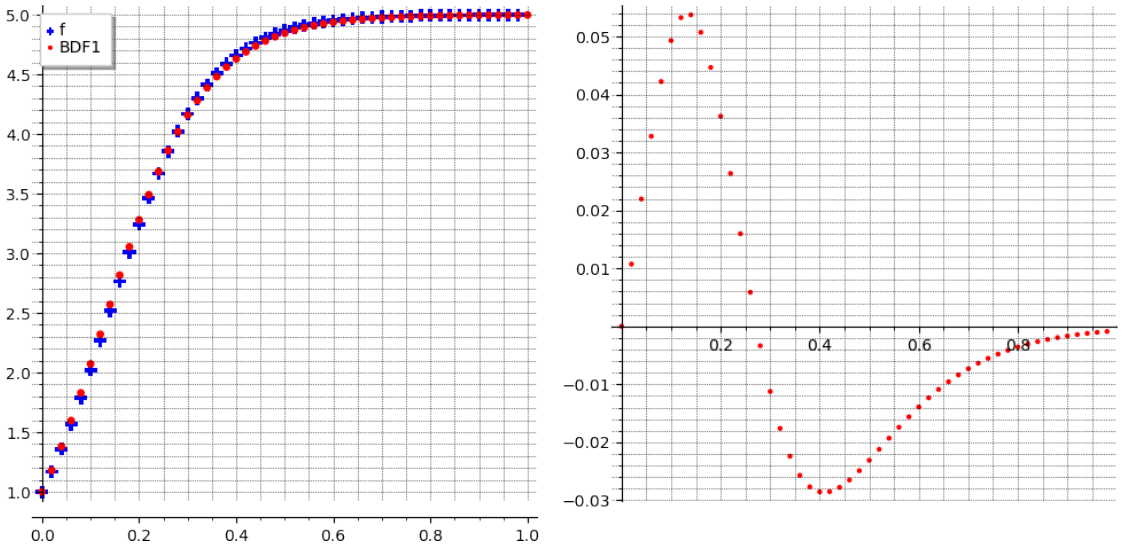

0.02
−0.02
−0.04
0.2
0.4
0.6
0.8

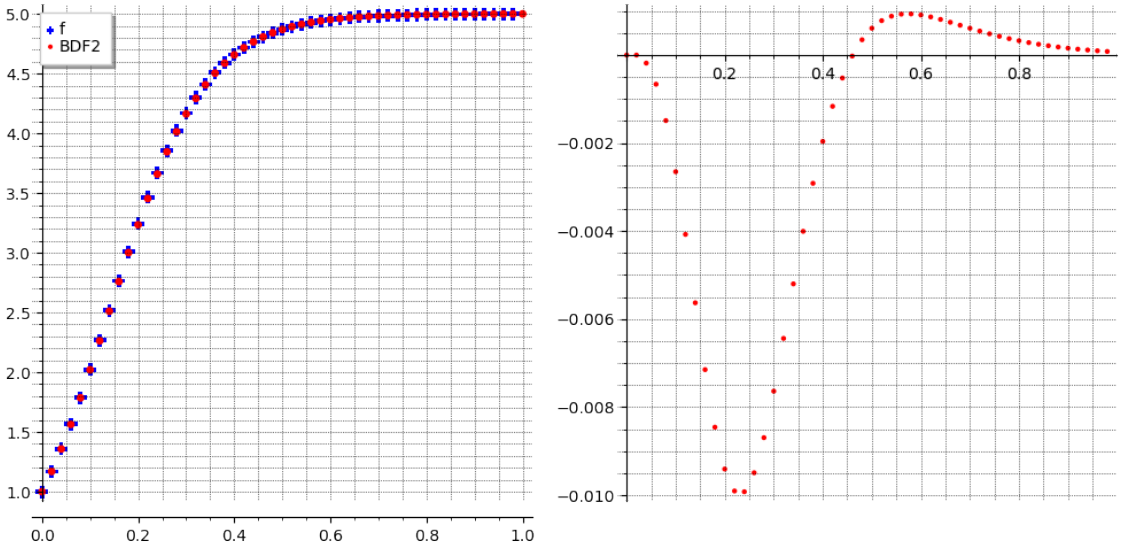

f
AB2
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0
0.2
0.4
0.6
0.8
1.0

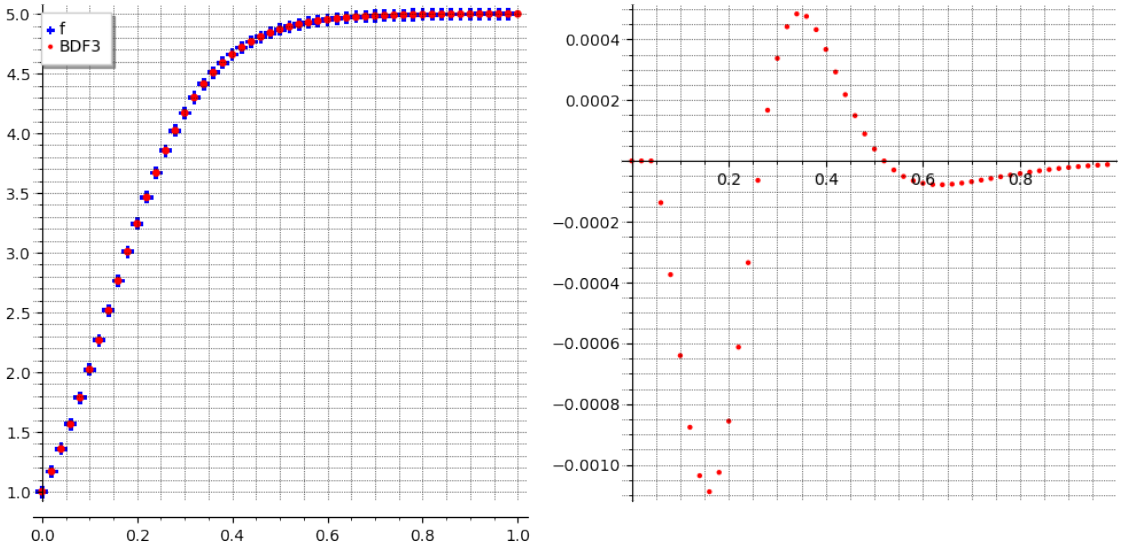

0.012
0.010
0.008
0.006
0.004
0.002
0.2
0.4
0.6
0.8


f
AB3
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0
0.2
0.4
0.6
0.8
1.0


0.0015
0.0010
0.0005
−0.0005
0.2
0.4
0.6
0.8

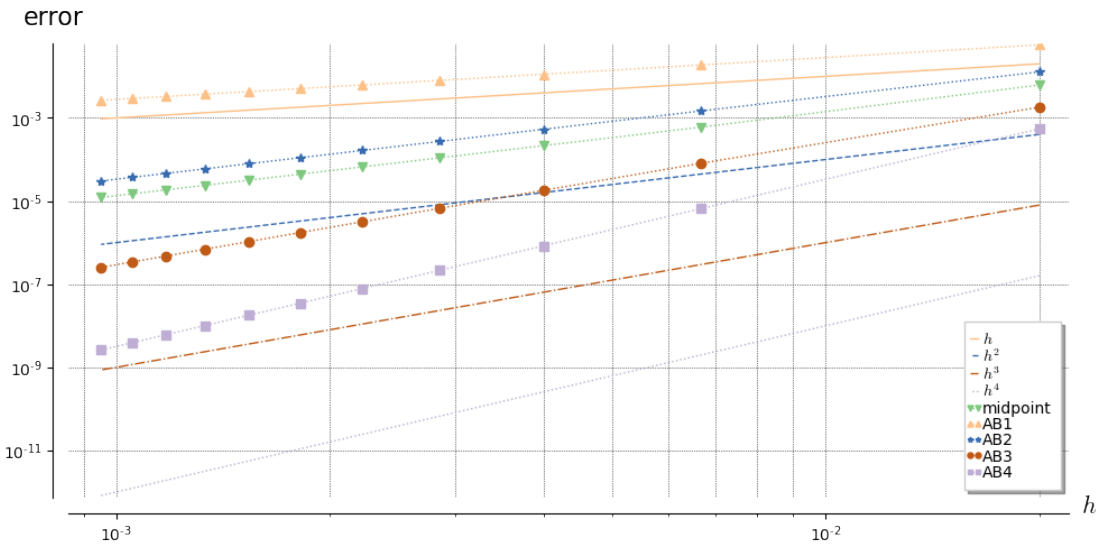






Grafichiamo adesso l'errore massimo dei metodi mid-point e di Adams-Bashfort, applicati all'equazione logistica in $[0, 1]$, al variare del numero di passi da 100 a 1000. Le linee tratteggiate descrivono l'andamento dei vari ordini di convergenza, da 1 a 4. Osserviamo che i metodi di Adams hanno in effetti ordine pari al numero di passi (l'andamento degli errori è esattamente parallelo a quello della rispettiva retta in blu).

```
%run LMF.ipynb

f = lambda t,y: 10*y-2*y**2
ft = lambda t: float(5./(1+4*np.exp(-10*t)))

minstep,maxstep,step = 50,1050,100
ic = lambda steps: list(np.array([ft(n*(1./steps))]) for n in range(10))

res = dict()
res['midpoint'] = [LMF(f, 0, 1, steps, ic(steps),k=2, method='NewtonCotes', kind='explicit')
                   for steps in range(minstep, maxstep+step, step)]
res['AB1'] = [LMF(f, 0, 1, steps, ic(steps),k=1, method='Adams', kind='explicit')
              for steps in range(minstep, maxstep+step, step)]
res['AB2'] = [LMF(f, 0, 1, steps, ic(steps),k=2, method='Adams', kind='explicit')
              for steps in range(minstep, maxstep+step, step)]
res['AB3'] = [LMF(f, 0, 1, steps, ic(steps),k=3, method='Adams', kind='explicit')
              for steps in range(minstep, maxstep+step, step)]
res['AB4'] = [LMF(f, 0, 1, steps, ic(steps),k=4, method='Adams', kind='explicit')
              for steps in range(minstep, maxstep+step, step)]



for method, steps in res.items():
    res[method] = np.array([ max([abs(p[1][0]-ft(p[0])) for p in points ]) for points in steps])

h = 1/np.array(range(minstep,maxstep+step,step))
p = list_plot([(x,x) for x in h], scale='loglog', plotjoined=True, linestyle='-', gridlines='minor',
              axes_labels=['$h$','error'], legend_label='$h$', color=list(colormaps.Accent(1/4))[0:3])
p += list_plot([(x,x**2) for x in h], plotjoined=True,linestyle='--',
               legend_label='$h^2$', color=list(colormaps.Accent(2/4))[0:3])
p += list_plot([(x,x**3) for x in h], plotjoined=True,linestyle='-.',
               legend_label='$h^3$', color=list(colormaps.Accent(3/4))[0:3])
p += list_plot([(x,x**4) for x in h], plotjoined=True,linestyle=':',
               legend_label='$h^4$', color=list(colormaps.Accent(4/4))[0:3])

markers = ['v', '^', '*', 'o', 's']

for i,v in enumerate(res.items()):
    key,value = v
    p += list_plot([(h[i], value[i]) for i in range(len(h))], marker=markers[i],
                   plotjoined=True,linestyle=':', color=list(colormaps.Accent(i/4))[0:3],
                   legend_label=key)

p.show()
```



Stabilità per h fissato

Studiamo il comportamento del metodo LMF $\rho(E)y_n - h\sigma(E)f_n$ sull'*equazione test*, di cui conosciamo la soluzione esplicita:

$$y'(t) = \lambda y(t), \quad y(0) = y_0, \quad Re(\lambda) < 0, \qquad \Longrightarrow y(t) = y_0 e^{\lambda(t-t_0)}$$

per la quale vale:

$$\lim_{t\to\infty} y_0 e^{\lambda(t-t_0)} = 0$$

Se anche per la soluzione discreta y_n:

$$\rho(E)y_n - h\sigma(E)f_n = \rho(E)y_n - h\sigma(E)\lambda y_n = (\rho(E) - h\lambda\sigma(E))y_n = 0 \tag{13.12}$$

si avesse la stabilità asintotica, ovvero $\lim_{n\to\infty} y_n = 0$, allora anche l'equazione dell'errore:

$$(\rho(E) - h\lambda\sigma(E))e_n = \tau_{n+k}$$

sarebbe tale che $\lim_{n\to\infty} e_n = 0$, in quanto $e_n = y_n - \hat{y}_n$, $(\rho(E) - h\lambda\sigma(E))\hat{y}_n = \tau_{n+k}$ e $\lim_{n\to\infty} \hat{y}_n = 0$ (in cui $\hat{y}_n$ rappresenta la soluzione esatta).
La stabilità asintotica si ha quando polinomio caratteristico della (13.12), ovvero:

$$\pi(z, h\lambda) = \pi(z, q) = \rho(z) - q\sigma(z)$$

è di Schur.

Definizione DEF13.3: Regione di assoluta stabilità

$$\mathcal{D} = \{q \in \mathbb{C} | \pi(z, q) \text{ è di Schur}\}$$

Definizione DEF13.4: A-stabilità
Un metodo LMF(ρ, σ) è

- A-stabile se $\mathbb{C}^- \subseteq \mathcal{D}$
- Perfettamente A-stabile se $\mathcal{D} \equiv \mathbb{C}^-$

Teorema TH13.2: seconda barriera di Dahlquist
Non esistono metodi LMF espliciti che siano A-stabili; l'ordine massimo di un metodo LMF A-stabile è 2.

Boundary Locus

Definiamo il boundary locus come la curva Γ del piano complesso luogo dei punti in cui $\pi(z, q)$ ha almeno una radice di modulo unitario. Nel piano complesso tutti e soli i punti sulla circonferenza unitaria hanno modulo 1, ovvero i punti $e^{i\theta} = cos(\theta) + i\sin(\theta)$, $0 \leq \theta \leq 2\pi$. Cerchiamo quindi i q per i quali vale:

$$\pi(z, q) = \rho(z) - q\sigma(z) = 0, \quad z = e^{i\theta}, 0 \leq \theta \leq 2\pi$$

Definizione DEF13.5: Boundary Locus

$$\Gamma = \left\{ q(\theta) = \frac{\rho(e^{i\theta})}{\sigma(e^{i\theta})} : 0 \leq \theta \leq 2\pi \right\}$$

Grafichiamo il boundary locus e quindi le regioni di stabilità asintotica dei metodi che abbiamo visto fino ad ora.
Rappresentiamo in blu la curva Γ. Campioniamo poi q ad intervalli regolari e controlliamo se $\pi(z, q)$ sia di Schur (verde) oppure no (rosso).

Per fare il campionamento dobbiamo testare π con i criteri di Schur; i coefficienti di $\pi(z, q)$ sono:

$$\pi(z, x+iy) = \rho(z) - q\sigma(z) = \sum_{i=0}^{k} \alpha_i z^i - \sum_{i=0}^{k} q\beta_i z^i = \sum_{i=0}^{k} (\alpha_i - q\beta_i) z^i$$

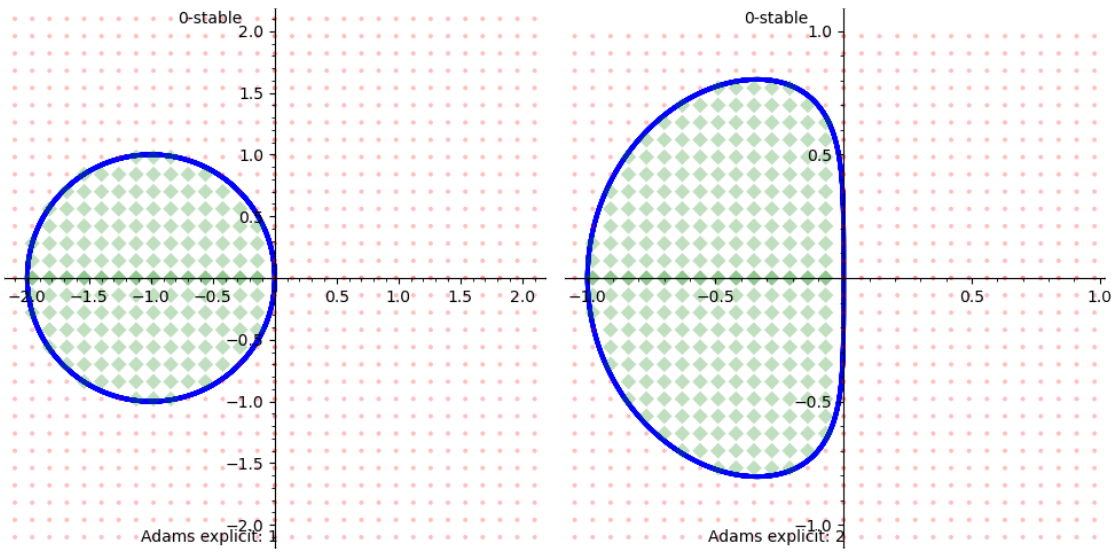
```
%run LMF.ipynb

from numpy import conjugate as bar
from numpy import polyval
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]=10,5


def reduced_polynomial(c):
    # c such that p(z) = c[0]z^k+...+c[k]z^0 -> c[i] = bar(c[0])c[i] - c[k]bar(c[k-i])
    k = len(c)-1
    return [(bar(c[0])*c[i] - c[k]*bar(c[k-i]))  for i in range(0,k)]

def derivative_polynomial(c):
    #c such that p(z) = c[0]z^k + ... + c[k]z^0 -> c' = [k*c[0], (k-1)c[1], ...]
    k = len(c)-1
    return [(k-i)*c[i] for i in range(0,k)]

def check_schur(c):
    # c such that p(z) = c[0]z^k+...+c[k]z^0
    k = len(c)-1
    if k==0:
        return True

    if not abs(c[0])>abs(c[k]):
        return False

    return check_schur(reduced_polynomial(c))

def check_vonneumann(c):
    # c such that p(z) = c[0]z^k+...+c[k]z^0
    k = len(c)-1
    if k==0:
        return True
    first_crit = abs(c[0])>abs(c[k]) and check_vonneumann(reduced_polynomial(c))
    second_crit = not(any(reduced_polynomial(c))) and check_schur(derivative_polynomial(c))

    return first_crit or second_crit


def plot_boundary_locus(k, method, kind):
    c = LMF_coeffs(k,method,kind)[0]
    a = c[0:k+1][::-1] #alpha_k,...,alpha_0
    b = c[k+1:][::-1] #beta_k,...,beta_0

    # The cast to complex is necessary for sage list_plot to work correctly
    q = lambda theta: complex(
        sum(a[i] * theta**(k-i) for i in range(k+1)) / sum(b[i] * theta**(k-i) for i in range(k+1)))

```

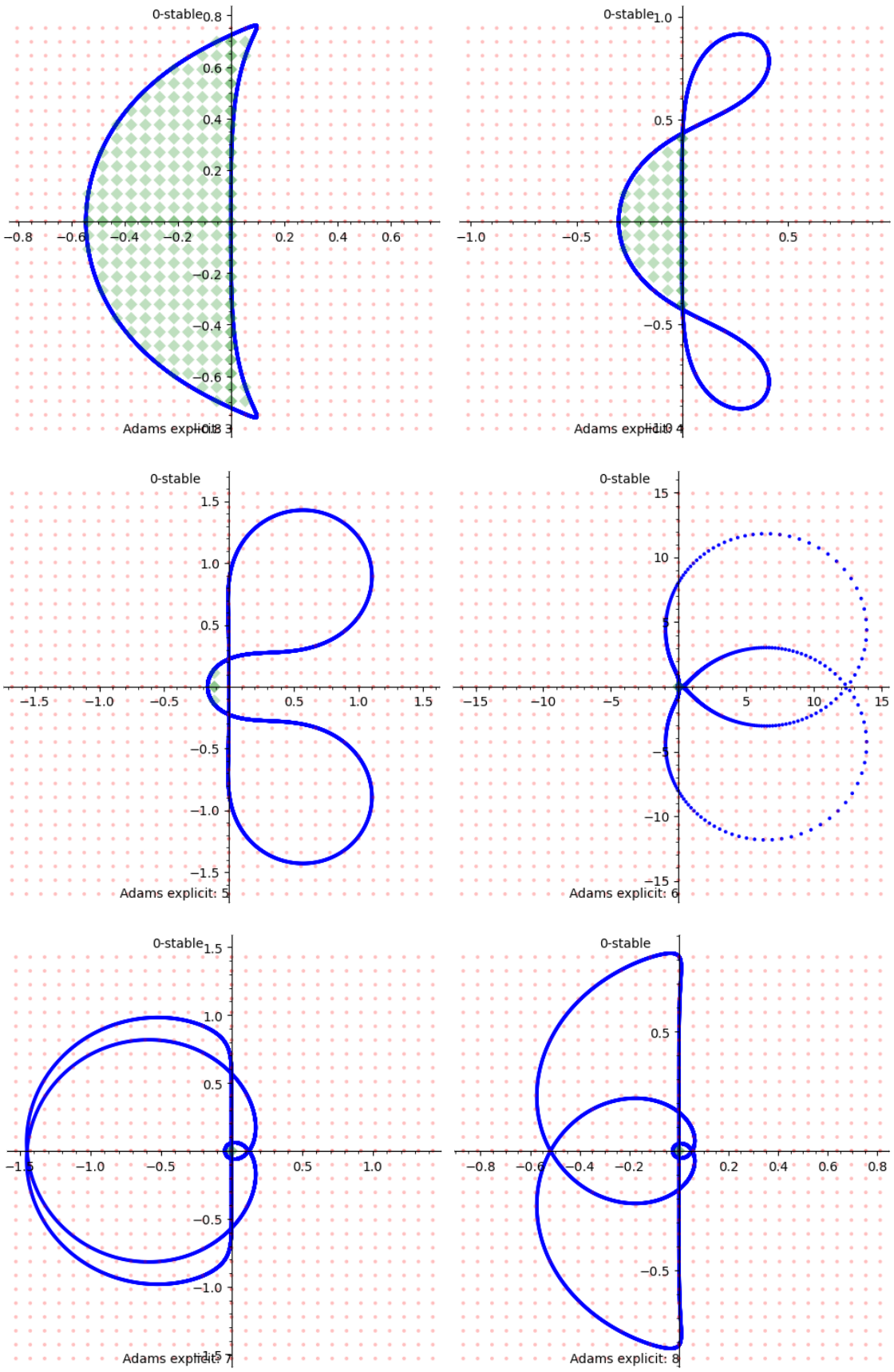
```
    points = list(q(exp(1j*theta)) for theta in np.arange(0,2*np.pi,0.001))

    p = list_plot(points, plotjoined=False, size=8)

    mm = max(map(abs, points))*1.05 #max value+margin for plotting
    sample_step = mm/15.
    for x in list(np.arange(-mm,mm, sample_step))+[0]:
        for y in list(np.arange(-mm,mm,sample_step))+[0]:
            coeffs = [ a[i]-(x+I*y)*b[i] for i in range(k+1)]
            color = 'green' if check_schur(coeffs) else 'red'
            marker = 'D' if color=='green' else 'o'
            size = 48 if color=='green' else 8
            p += list_plot([[x,y]], color=color, size=size, marker=marker, alpha=0.25)

    p+=text(method+' '+kind+': '+str(k), [-mm/4,-mm], color='black')
    coeffs = [a[i] for i in range(k+1)]
    if check_vonneumann(coeffs):
        p+=text("0-stable", [-mm/4,mm], color='black')
    return p
```

Osserviamo innanzitutto che nel punto $q = (0, i0)$ $\pi(z, q)$ non è di Schur ma può essere di VonNeumann, a causa delle condizioni di consistenza.

Vediamo le regioni di assoluta stabilità dei metodi di Adams espliciti di ordine $1, .., 8$. Osserviamo empricamente che i metodi di Adams-Bashfort hanno una regione di stabilità asintotica limitata che si restringe all'aumentare del grado.

```
p=[]
for i in range(1,9):
    p.append(plot_boundary_locus(i,'Adams', 'explicit'))

g = graphics_array(p[0:2])
g.show()
g = graphics_array(p[2:4])
g.show()
g = graphics_array(p[4:6])
g.show()
g = graphics_array(p[6:8])
g.show()
```

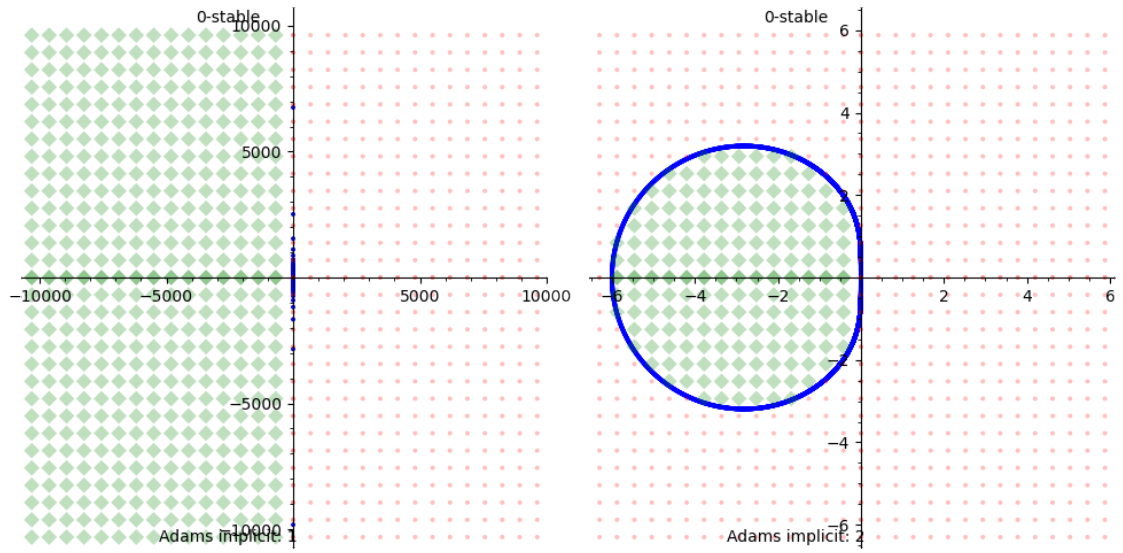


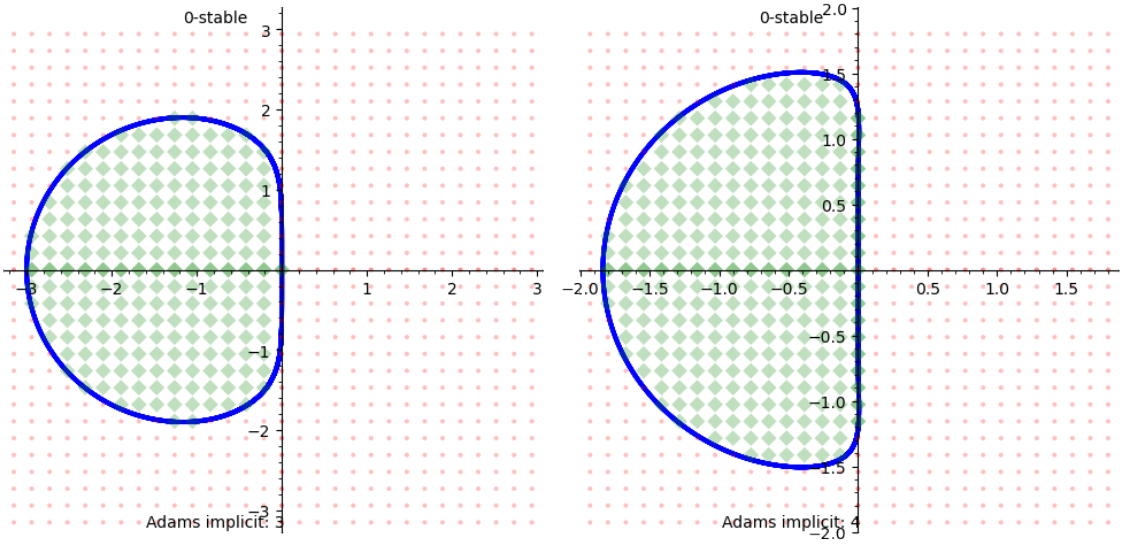


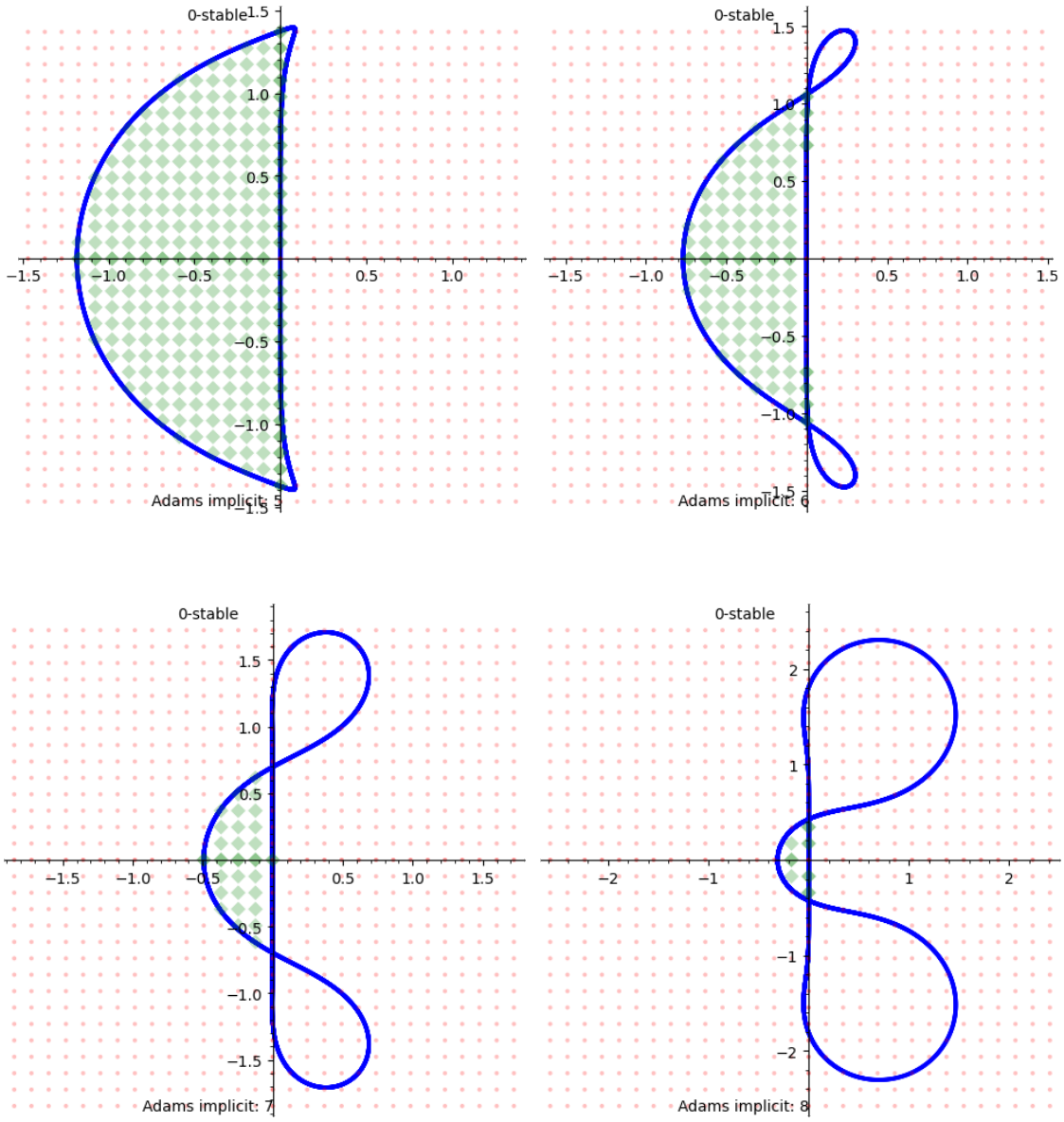
Per quanto riguarda i metodi di Adams-Moulton, il primo grado, che coincide con il metodo dei trapezi è perfettamente A-stabile. Tutti gli altri hanno una regione di stabilità asintotica limitata.

```
p=[]
for i in range(1,9):
```

```
        p.append(plot_boundary_locus(i,'Adams', 'implicit'))

    g = graphics_array(p[0:2])
    g.show()
    g = graphics_array(p[2:4])
    g.show()
    g = graphics_array(p[4:6])
    g.show()
    g = graphics_array(p[6:8])
    g.show()
```

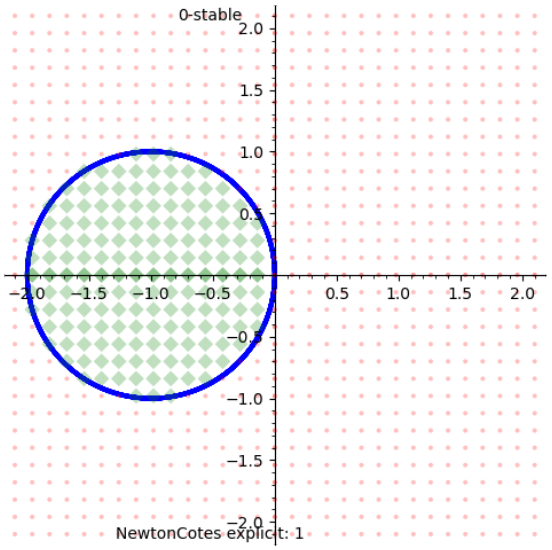


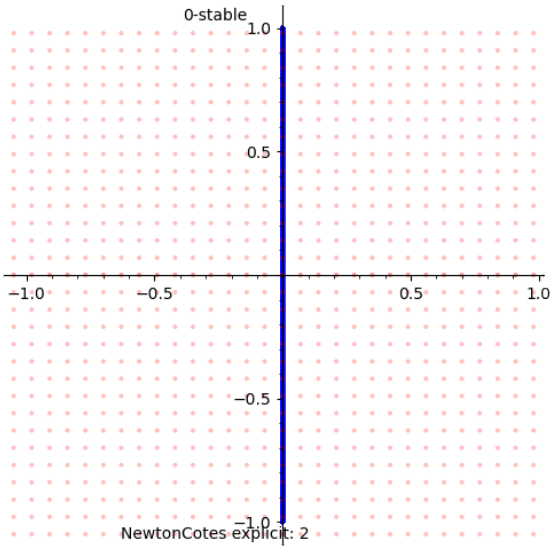


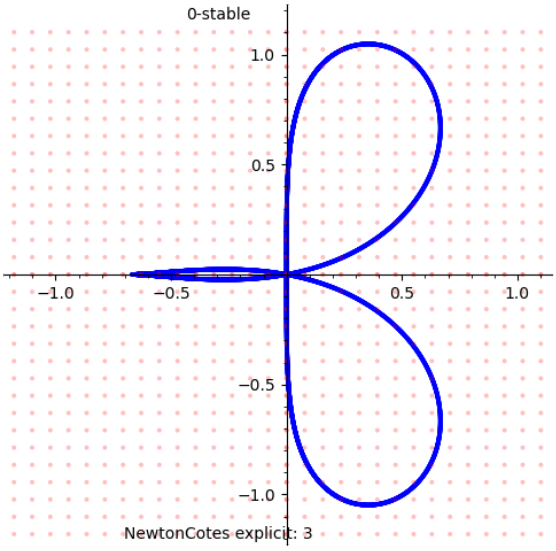


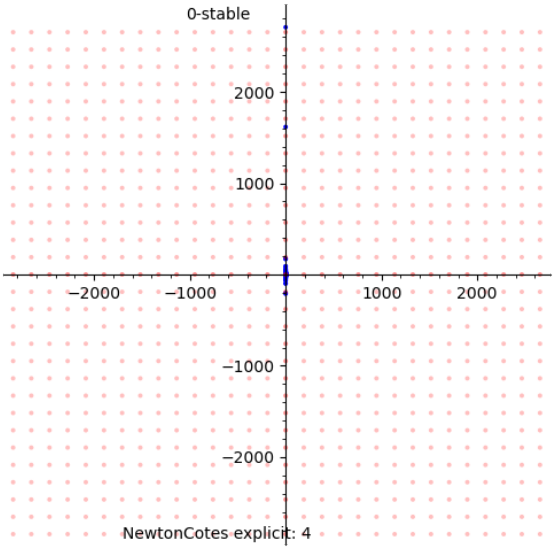
I metodi di Newton-Cotes espliciti sono 0-stabili, ma con $k > 2$ non hanno regioni di stabilità asintotica, a meno di un intervallo limitato sull'asse immaginario.

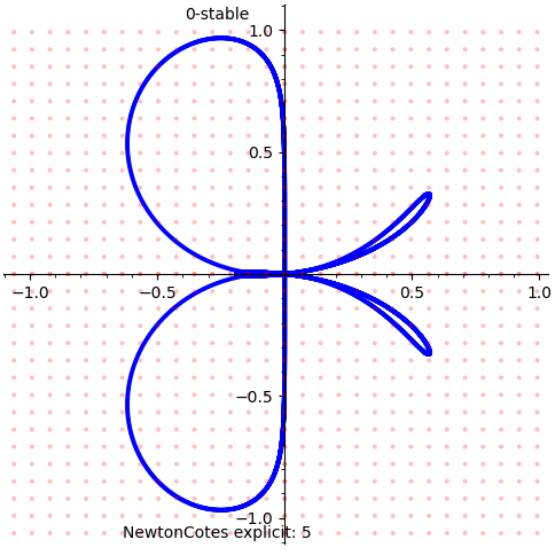
```
p=[]
for i in range(1,9):
    p.append(plot_boundary_locus(i,'NewtonCotes', 'explicit'))

g = graphics_array(p[0:2])
g.show()
g = graphics_array(p[2:4])
g.show()
g = graphics_array(p[4:6])
g.show()
g = graphics_array(p[6:8])
g.show()
```

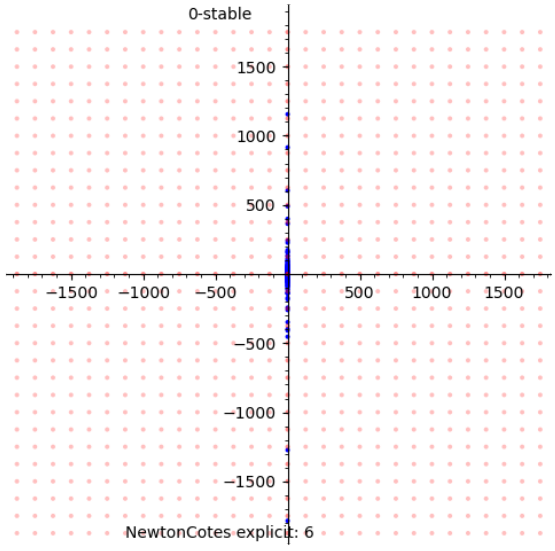

0-stable
NewtonCotes explicit: 1

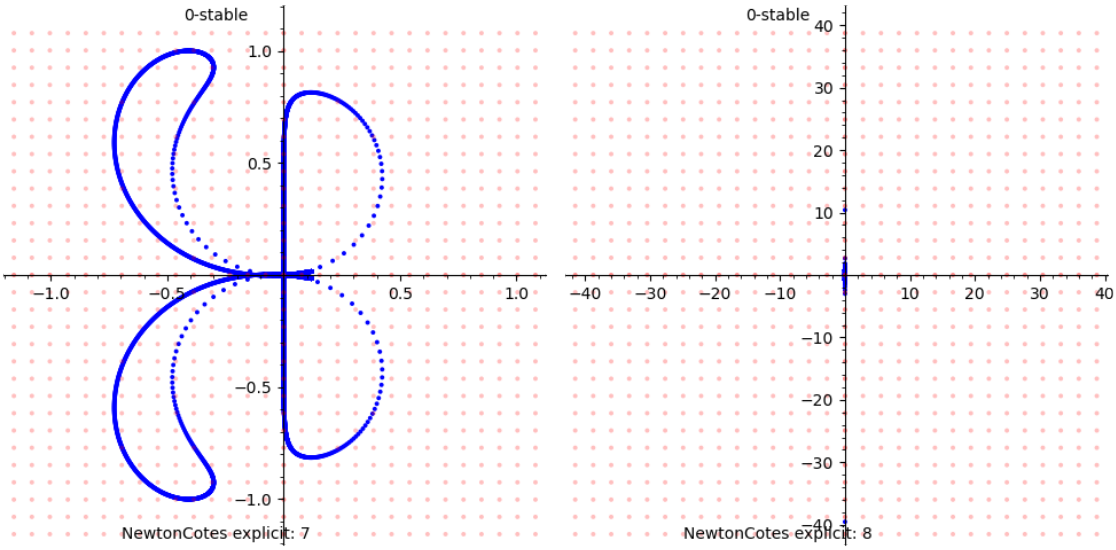

0-stable
NewtonCotes explicit: 2


0-stable
NewtonCotes explicit: 3


0-stable
NewtonCotes explicit: 4

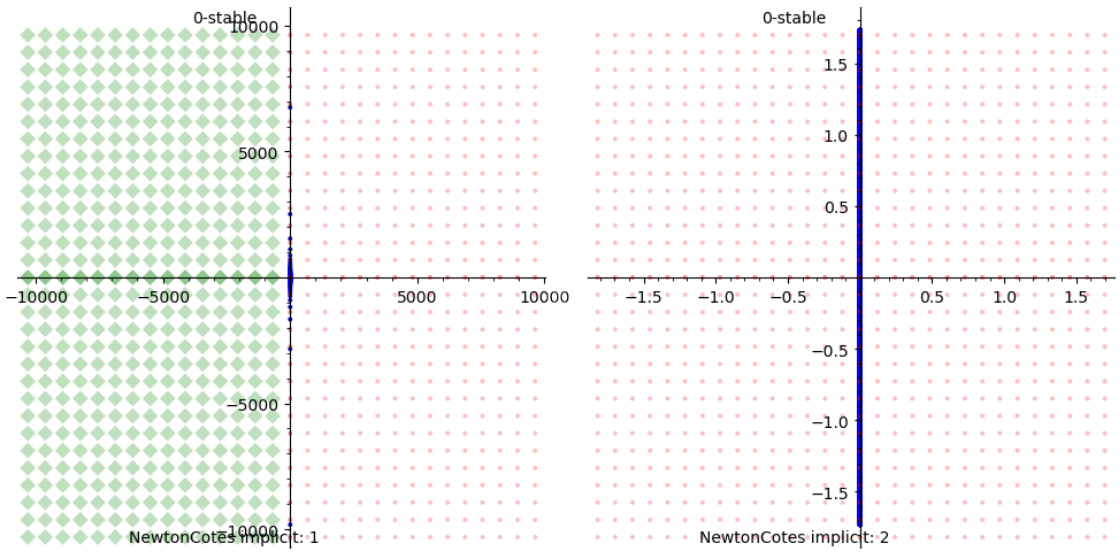

0-stable
NewtonCotes explicit: 5

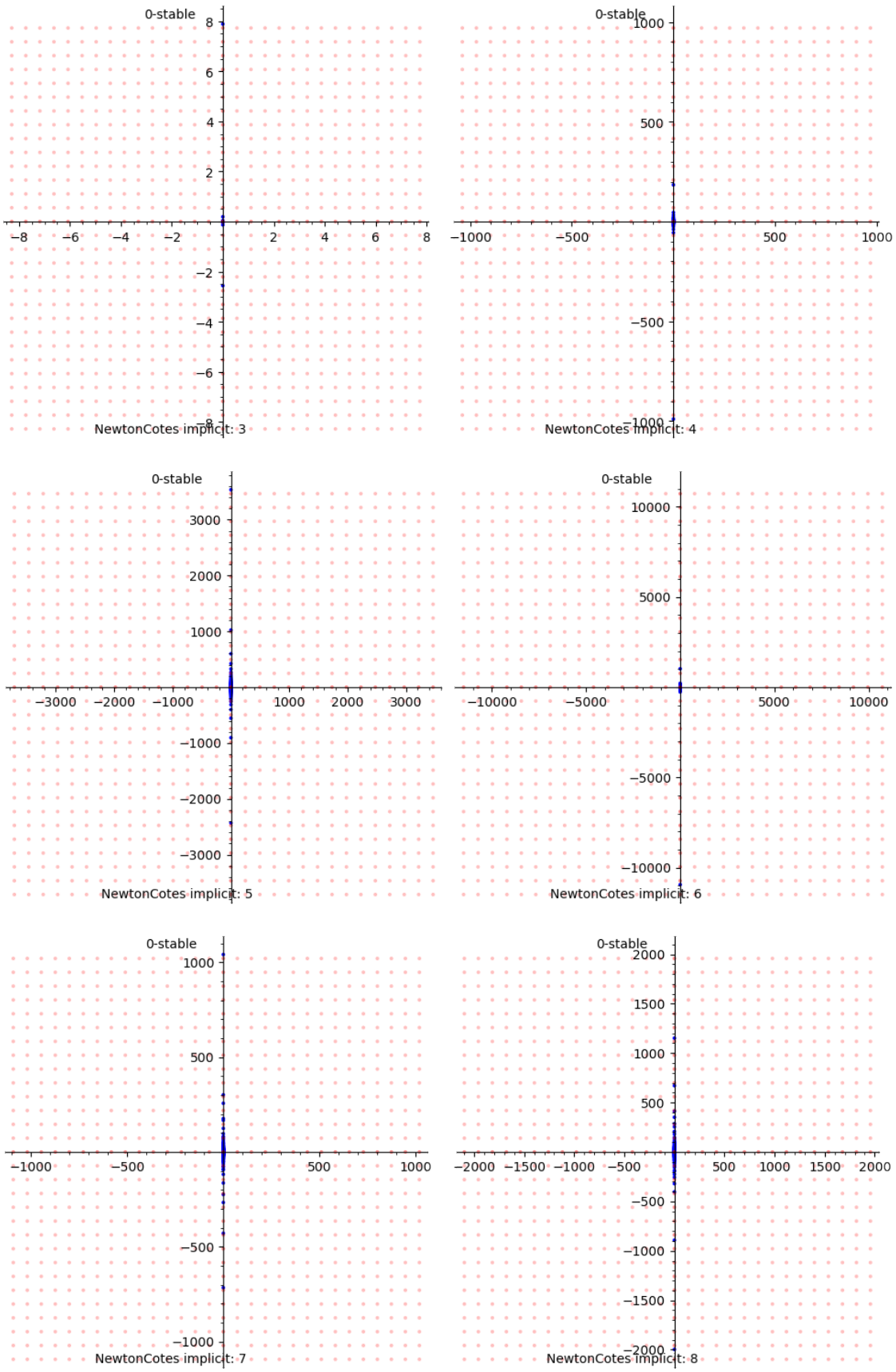

0-stable
NewtonCotes explicit: 6



I metodi di Newton-Cotes impliciti:

```
p=[]
for i in range(1,9):
    p.append(plot_boundary_locus(i,'NewtonCotes', 'implicit'))

g = graphics_array(p[0:2])
g.show()
g = graphics_array(p[2:4])
g.show()
g = graphics_array(p[4:6])
g.show()
g = graphics_array(p[6:8])
g.show()
```



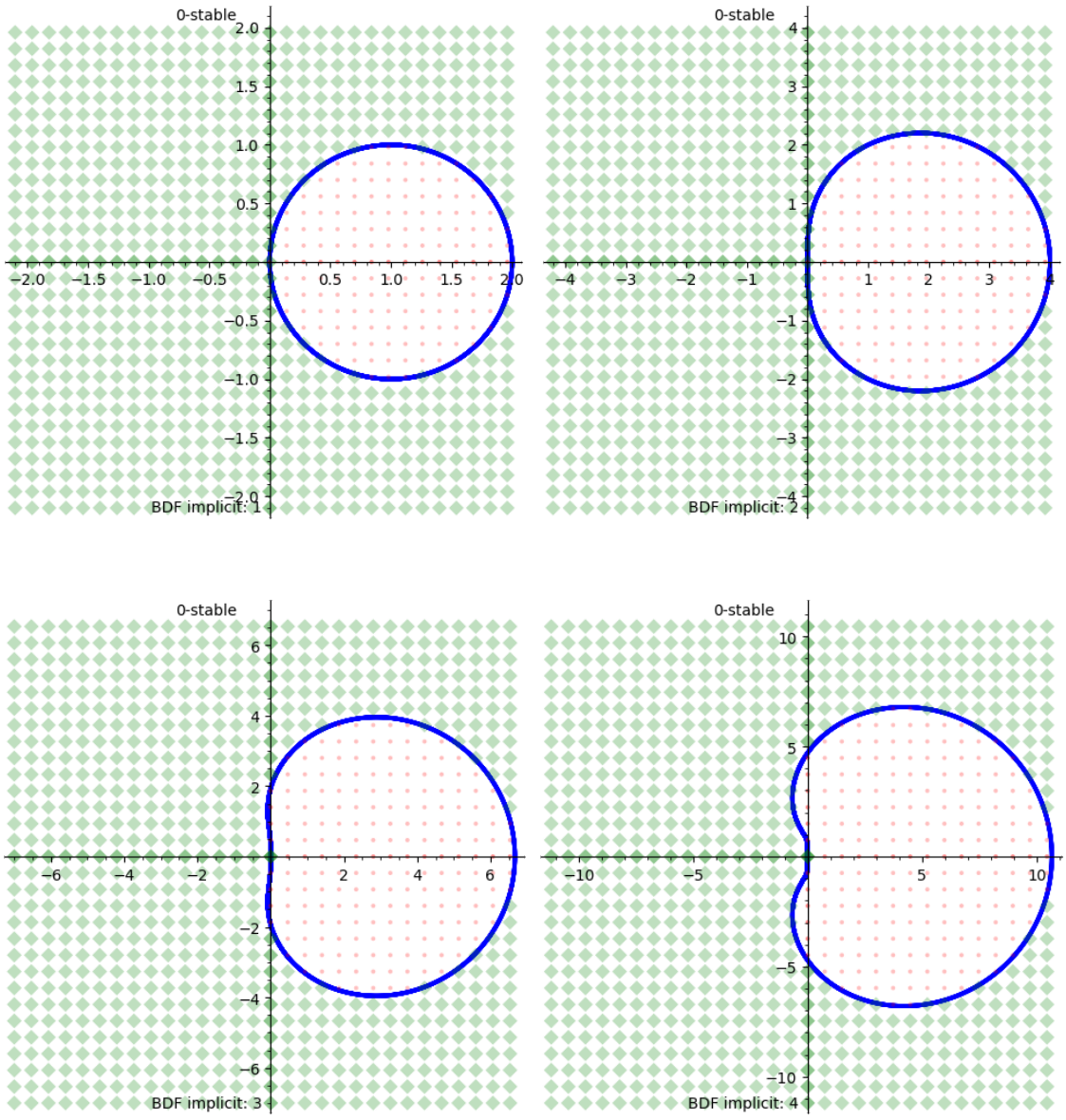


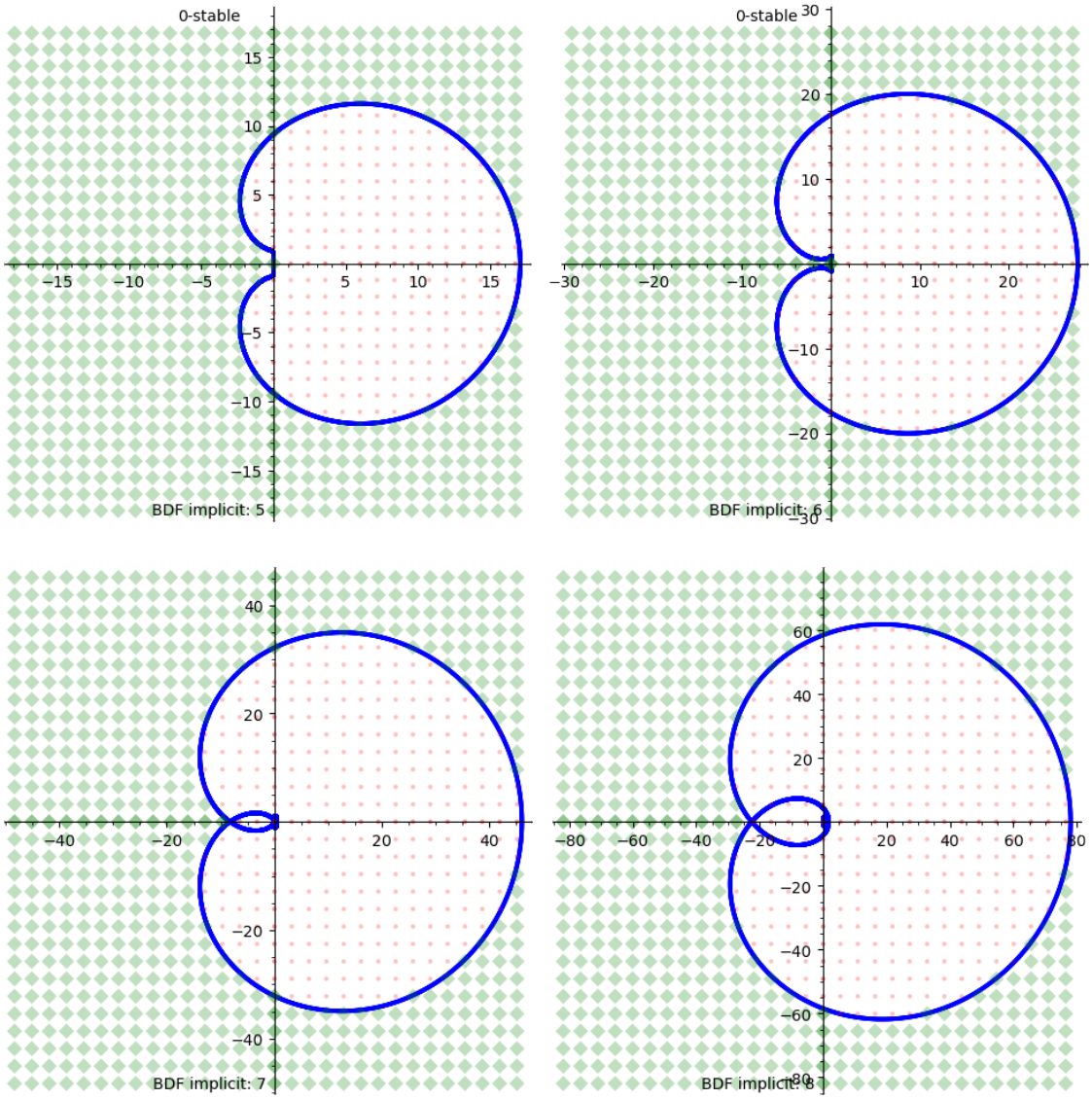
I metodi BDF sono quelli con la regione di stabilità più ampia, ma, come abbiamo dimostrato in precedenza, dal 7 grado smettono di essere 0-stabili.

```
p=[]
for i in range(1,9):
```

```
    p.append(plot_boundary_locus(i,'BDF', 'implicit'))

g = graphics_array(p[0:2])
g.show()
g = graphics_array(p[2:4])
g.show()
g = graphics_array(p[4:6])
g.show()
g = graphics_array(p[6:8])
g.show()
```

```
/tmp/ipykernel\_1803812/2529482878.py:12: RuntimeWarning: overflow encountered in cdouble\_scalars
  return [(bar(c[Integer(0)])*c[i] - c[k]*bar(c[k-i]))  for i in range(Integer(0),k)]
/tmp/ipykernel\_1803812/2529482878.py:12: RuntimeWarning: invalid value encountered in cdouble\_scalars
  return [(bar(c[Integer(0)])*c[i] - c[k]*bar(c[k-i]))  for i in range(Integer(0),k)]
```





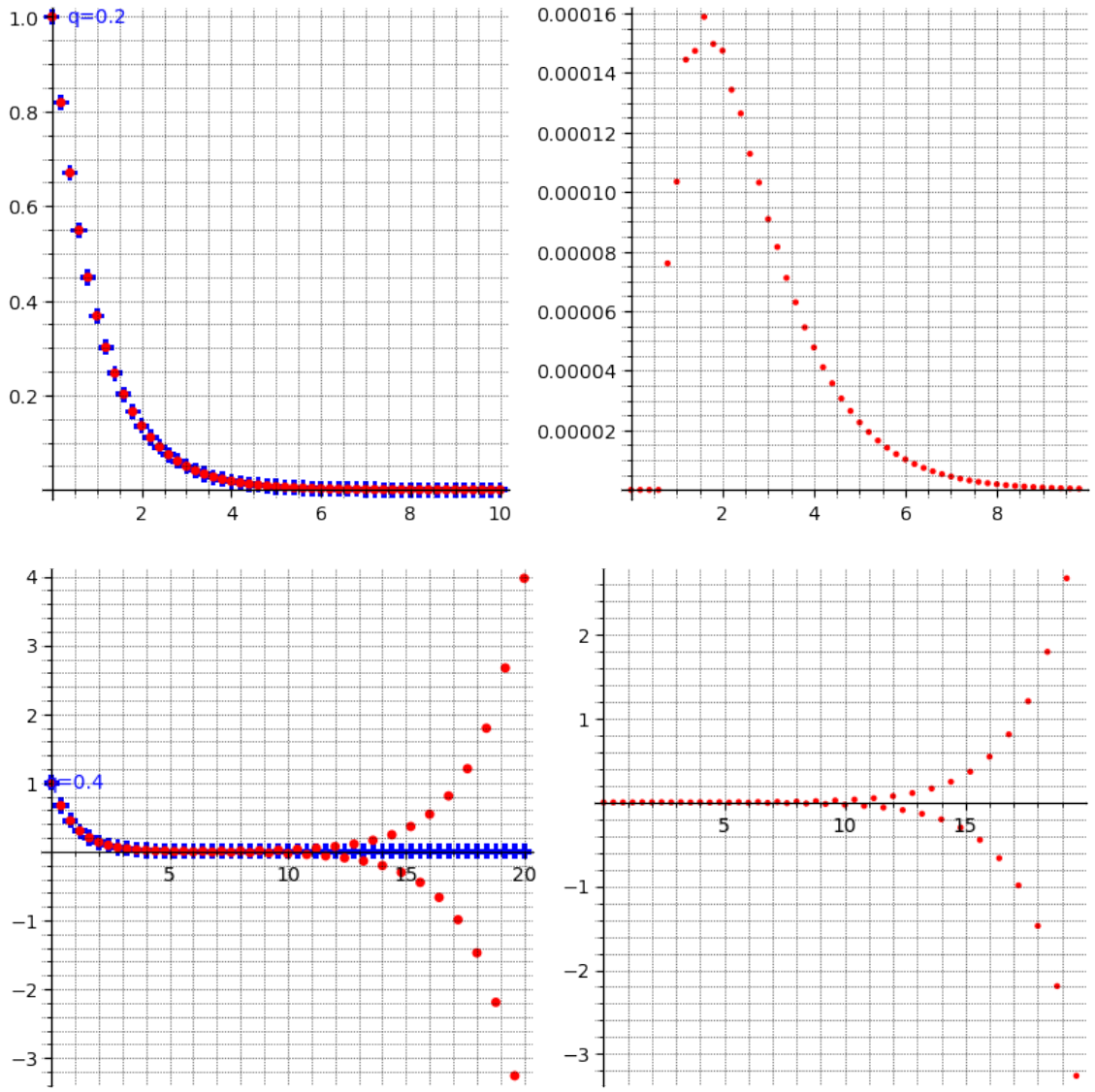
Alla luce delle regioni di stabilità individuate studiando il boundary locus, possiamo testare l'effettivo comportamento di alcuni metodi sull'equazione test $y' = \lambda y$.

```
%run LMF.ipynb
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]=8,4

lam = -1

def compare(l,q,method,kind,k,label=None, tol=False):
    h = abs(q/l)
    steps = 50
    T = h*steps

    f = lambda t,y: l*y
    ft = lambda t: float(exp(l*t))

    ic = lambda steps: list(np.array([ft(n*h)]) for n in range(10)) #initial conditions

    d = LMF(f, 0, T, steps, ic(steps),k, method=method, kind=kind,  tol=tol)

    err = [(d[i][0], (d[i][1]-ft(d[i][0]))) for i in range(steps)]

    p = list_plot([(x,ft(x)) for x in np.arange(0,T+h,h)], color='blue', marker='P', size=64,
                  gridlines='minor')
```

```
23
24      p += list_plot(d, color='red', size=24)
25
26      e = list_plot(err, color='red', gridlines='minor')
27
28      if label:
29          p+=text(label,[1,1])
30
31      g = graphics_array([p,e])
32
33      g.show()
```

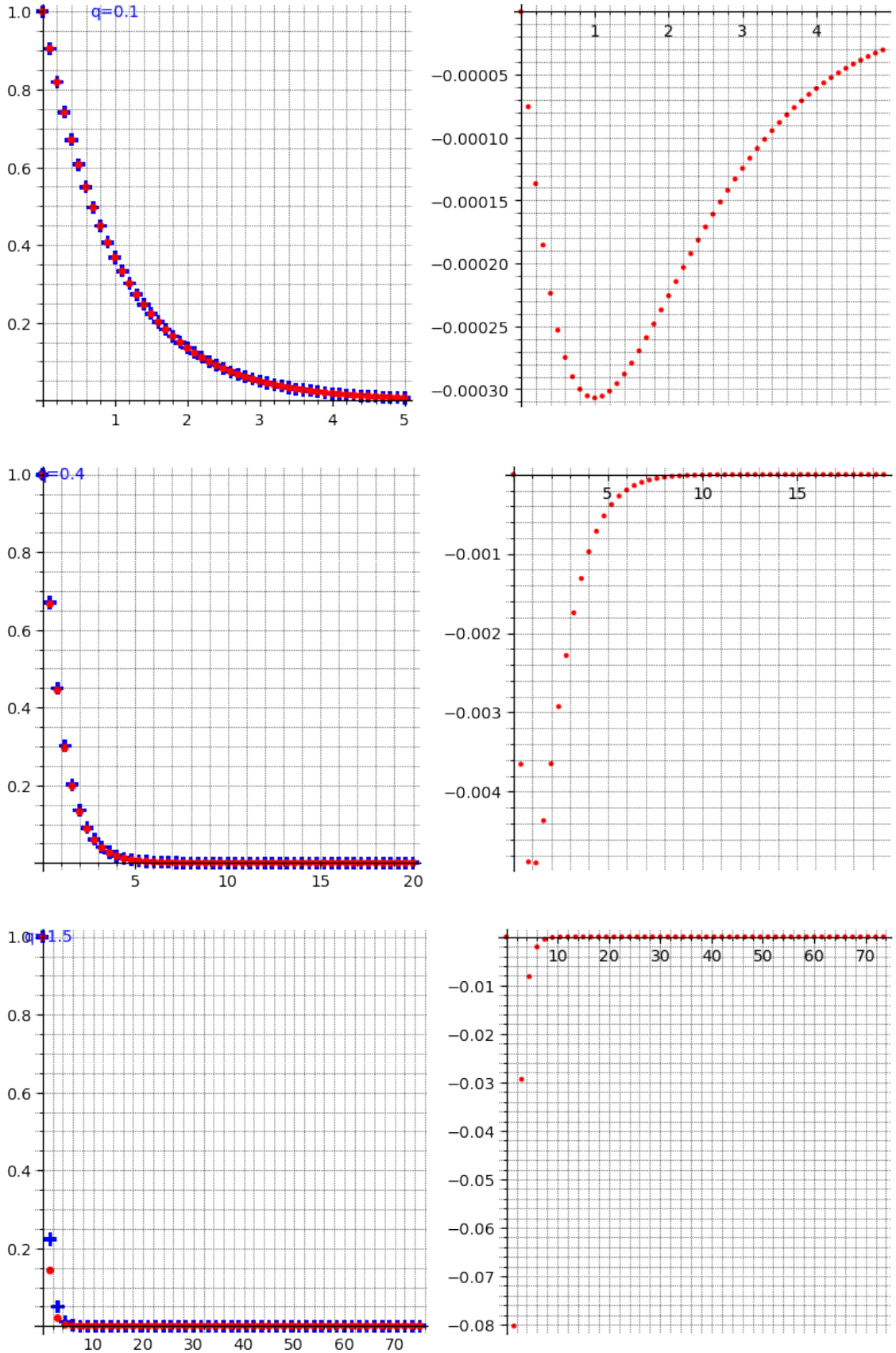
Confrontiamo i metodi di Adams espliciti di ordine 4: quando scegliamo il passo per avere $q = 0.2$ siamo all'interno della zona di assoluta stabilità, ma raddoppiando il passo, con $q = 0.4$ il metodo diverge.

```
1   compare(lam,0.2,'Adams','explicit',4,label='q=0.2')
2   compare(lam,0.4,'Adams','explicit',4,label='q=0.4')
```



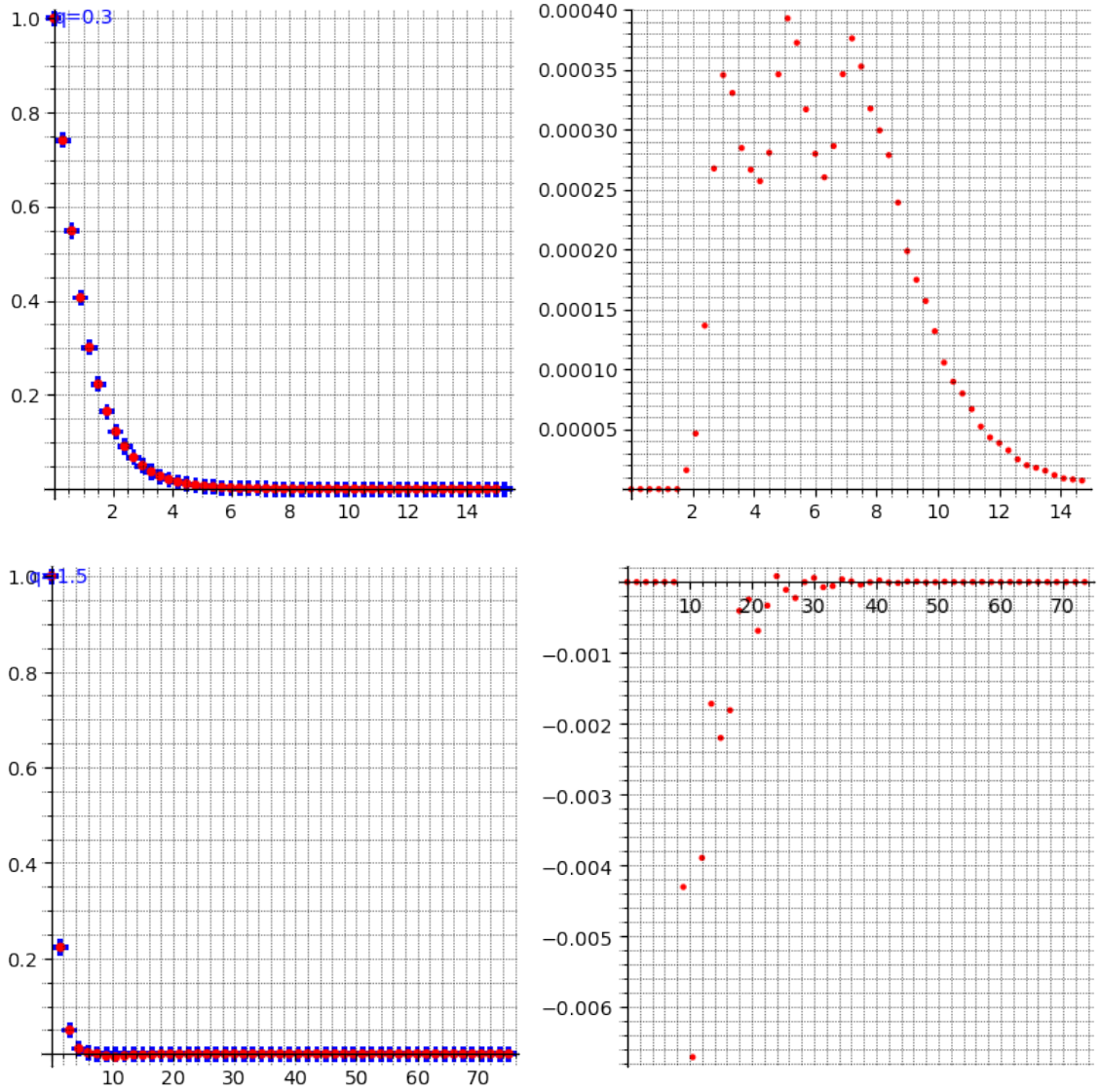
Il metodo dei trapezi (Newton-Cotes implicito di ordine 1) è A-stabile, e converge anche quando $h > 1$.

```
1   compare(lam,0.1,'NewtonCotes','implicit',1,label='q=0.1')
2   compare(lam,0.4,'NewtonCotes','implicit',1,label='q=0.4', tol=10**-15)
3   compare(lam,1.5,'NewtonCotes','implicit',1,label='q=1.5', tol=10**-15)
```



Le BDF di ordine 6 non sono A-stabili, ma hanno comunque una regione di stabilità asintotica che si estende all'infinito quando $Re(\lambda) < 0$. Se però $Im(\lambda) \neq 0$, h dovrà essere sufficientemente grande da mantenere il metodo fuori dalla zona di instabilità.

```
compare(lam,0.3,'BDF','implicit',6,label='q=0.3')
compare(lam,1.5,'BDF','implicit',6,label='q=1.5')
```



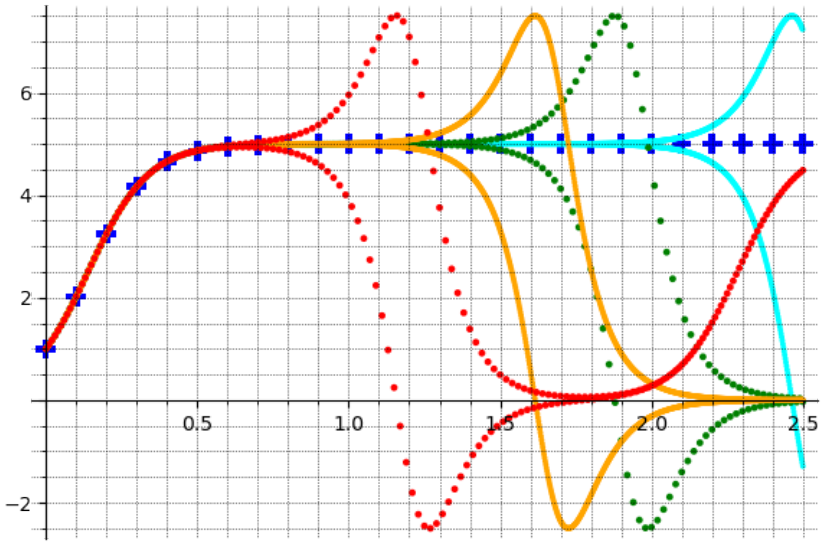
Osserviamo ancora il comportamento del metodo del mid-point sull'equazione logistica $y'(t) = 10y - 2y^2$. Il metodo, di ordine 2, è 0-stabile e consistente, e quindi convergente, ma abbiamo visto studiando il boundary locus che non è mai asintoticamente stabile (tranne il caso particolare di un segmento sull'asse immaginario).

Osserviamo, in corcordanza con l'analisi, che la soluzione non si stabilizza ma oscilla, peraltro in maniera periodica. In verde e in ciano vediamo il metodo con due diversi ordini di grandezza di h. Al diminuire di h, le oscillazioni si spostano in avanti (ovvero il metodo ha un comportamento stabile per più iterazioni), ma inizierà sempre ad oscillare.

In questo esempio vediamo anche l'importanta della precisione delle condizioni iniziali: in rosso e in arancio vediamo il comportamento dello stesso metodo con, come prima, 250 e 2500 passi sull'intervallo $[0, 2.5]$, in cui la seconda condizione iniziale però non è esatta ma è approssimata al primo grado con la formula $y_1 = y_0 + hf(t, y_0)$. La mancanza di precisione sulle condizioni iniziali arretra di molte iterazioni l'inizio delle oscillazioni.

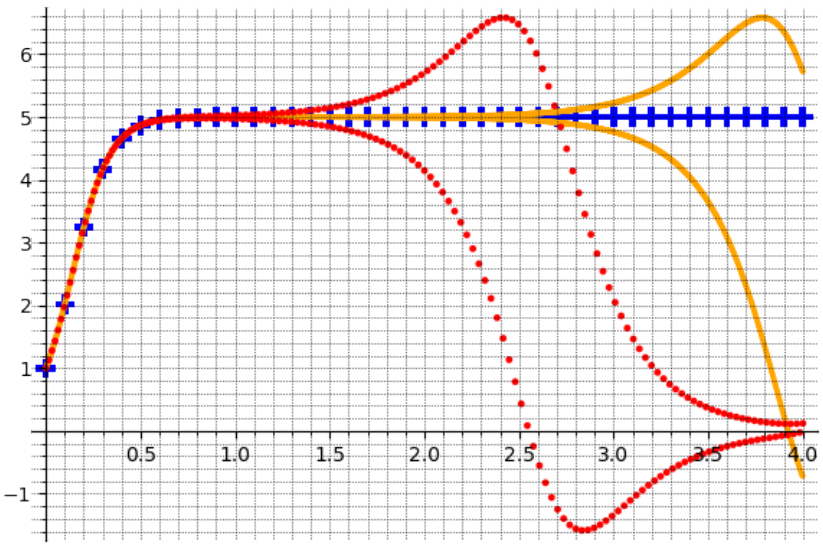
```
%run LMF.ipynb

f = lambda t,y: 10*y-2*y^2
data1 = LMF(f, 0, 2.5, 250, [np.array([1.])],k=2, method='NewtonCotes',kind='explicit')
data2 = LMF(f, 0, 2.5, 2500, [np.array([1.])],k=2, method='NewtonCotes',kind='explicit')

ic = [np.array([5/(1+4*exp(-10*x))]) for x in np.arange(0,2.5,0.01)]
data3 = LMF(f, 0, 2.5, 250, ic,k=2, method='NewtonCotes',kind='explicit')
```

```
9
10  ic = [np.array([5/(1+4*exp(-10*x))]) for x in np.arange(0,2.5,0.001)]
11  data4 = LMF(f, 0, 2.5, 2500, ic,k=2, method='NewtonCotes',kind='explicit')
12
13  p = list_plot([(x,5/(1+4*exp(-10*x))) for x in np.arange(0,2.6,0.1)], color='blue', marker='P', size=96,
14                gridlines='minor')
15  p += list_plot(data4,  color='cyan', size=8)
16  p += list_plot(data3,  color='green', size=12)
17  p += list_plot(data2,  color='orange', size=8)
18  p += list_plot(data1,  color='red', size=12)
19  p.show()
```



L'alter-ego implicito del metodo del mid-point, il metodo di Simpson, si comporta in maniera equivalente:

```
1   f = lambda t,y: 10*y-2*y^2
2   data1 = LMF(f, 0, 4, 250, [np.array([1.])],k=2, method='NewtonCotes',kind='implicit')
3   data2 = LMF(f, 0, 4, 2500, [np.array([1.])],k=2, method='NewtonCotes',kind='implicit')
4
5   p = list_plot([(x,5/(1+4*exp(-10*x))) for x in np.arange(0,4.1,0.1)], color='blue', marker='P', size=96,
6                 gridlines='minor')
7   p += list_plot(data2,  color='orange', size=8)
8   p += list_plot(data1,  color='red', size=12)
9   p.show()
```



14 Comportamento di un sistema dinamico lineare bidimensionale

14.1 Caso continuo

Esaminiamo le traiettorie delle soluzioni di un sistema dinamico continuo in due variabili:

$$\mathbf{y}'(t) = A\mathbf{y}(t) + \mathbf{b}, \quad A \in \mathbb{R}^{2\times 2}, \mathbf{b} \in \mathbb{R}^2, \mathbf{y} : \mathbb{R} \to \mathbb{R}^2$$

che ha soluzione generica:

$$\mathbf{y}(t) = \Phi_{(t,t_0)}\mathbf{y}(t_0) + \int_{t_0}^{t} \Phi_{(t,s)}\mathbf{b}(\mathbf{t})ds \tag{14.1}$$

in cui $\Phi_{(t,s)} = W(t)W^{-1}(s)$ è la matrice fondamentale del sistema e $W(t)$ una matrice nonsingolare soluzione del sistema omogeneo associato $W'(t) = A(t)W(t)$.

Si dimostra che se A e $\mathbf{b}$ sono costanti allora $\Phi_{(t,s)} = e^{A(t-s)}$ ed inoltre $\frac{\partial}{\partial t}e^{A(t-s)} = Ae^{A(t-s)}$. Assumendo che A sia nonsingolare, la soluzione (14.1) può essere semplificata:

$$\begin{aligned}
\mathbf{y}(t) &= e^{A(t-t_0)}\mathbf{y}(t_0) + \int_{t_0}^{t} e^{A(t-s)}\mathbf{b}ds \\
&= e^{A(t-t_0)}\mathbf{y}(t_0) + A^{-1}\int_{t_0}^{t} Ae^{A(t-s)}ds\mathbf{b} \\
&= e^{A(t-t_0)}\mathbf{y}(t_0) + A^{-1}e^{A(t-t_0)}\Big|_{t_0}^{t}\mathbf{b} \\
&= e^{A(t-t_0)}\mathbf{y}(t_0) + A^{-1}\mathbf{b} - e^{A(t-t_0)}A^{-1}\mathbf{b} \\
&= e^{A(t-t_0)}(\mathbf{y}(t_0) - A^{-1}\mathbf{b}) + A^{-1}\mathbf{b} \\
&= e^{A(t-t_0)}(\mathbf{y}(t_0) - \bar{\mathbf{y}}) + \bar{\mathbf{y}}
\end{aligned}$$

in cui $\bar{\mathbf{y}}$ è la soluzione costante e quindi un punto di equilibrio del sistema.

Ponendo $\tilde{\mathbf{y}}(t) = \mathbf{y}(t) + \bar{\mathbf{y}}$ otteniamo una soluzione che ha il punto di equilibrio nell'origine; per tale soluzione pertanto deve valere $\mathbf{0} = A\mathbf{0} + \mathbf{b}$ e quindi $\mathbf{b} = 0$; si ha quindi:

$$\tilde{\mathbf{y}}'(t) = A\tilde{\mathbf{y}}(t) \tag{14.2}$$

Casi

Esaminando (14.2), ovvero:

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

la matrice A ha due autovalori λ_1, λ_2 , per i quali si possono avere soltanto 3 casi distinti:

- $\lambda_1, \lambda_2 \in \mathbb{R}, \lambda_1 \neq \lambda_2$
- $\lambda_1, \lambda_2 \in \mathbb{R}, \lambda_1 = \lambda_2 = \lambda$
- $\lambda_1, \lambda_2 \in \mathbb{C}, \lambda_1 = \bar{\lambda}_2 = (\alpha \pm i\beta) \notin \mathbb{R}$

Esaminiamo il comportamento del sistema in ciascun caso.

Scegliamo delle condizioni iniziali casuali nell'intervallo $[-1, 1]$, che useremo per confrontare l'andamento delle soluzioni nei vari casi:

```
# Define colors and utilities
t,x,y = var('t, x, y')

initial_conditions = [(-1,-1), (-1,1), (1,-1), (1,1)] + [(random()*2-1,random()*2-1) for _ in range(20)]
```

```
conditions_number = len(initial_conditions)
colors = dict((ic,hue(i/conditions_number)) for i,ic in enumerate(initial_conditions))

import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]=3,3
```

Caso $\lambda_2 \neq \lambda_1$

In questo caso, A è diagonalizzabile tramite una trasformazione di similitudine reale. La matrice X che ha per colonne due autovettori relativi a λ_1 e λ_2 è tale che $AX = XJ$, in cui J è una matrice diagonale che ha per elementi i due autovettori.

Ponendo $\mathbf{z} = X^{-1}\mathbf{y}$, si ottiene:

$$\mathbf{z}' = X^{-1}A\mathbf{y} = X^{-1}AXX^{-1}\mathbf{y} = J\mathbf{z}$$

e quindi:

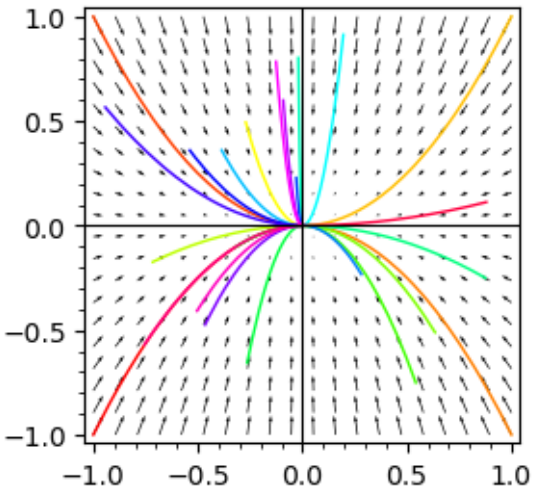
$$\begin{pmatrix} z_1' \\ z_2' \end{pmatrix} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

ovvero due equazioni indipendenti, che hanno soluzione $z(t) = z_0 e^{\lambda t}$ assumendo $t_0 = 0$.

$\lambda_2 < \lambda_1 < 0$: nodo stabile

In questo caso $\lim_{t\to\infty} e^{\lambda t} = 0$; le traiettorie delle soluzioni si muovono quindi verso il punto di equilibrio.

```
l1 = -1
l2 = -2
plots = []
plots.append(plot_vector_field((l1*x,l2*y), (x,-1,1), (y, -1,1)))

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((y0_1*e^(l1*t), y0_2*e^(l2*t)), (t,0,10.0), color=colors[(y0_1,y0_2)]))
show(sum(plots))
```

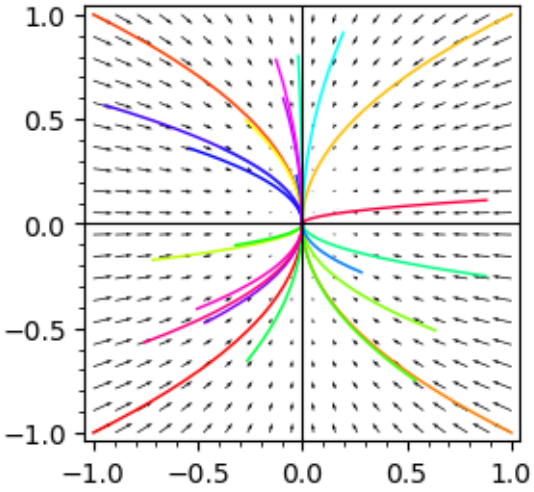


$\lambda_1 < \lambda_2 < 0$

Scambiando gli autovalori, il comportamento rimane ovviamente invariato ma ruotato rispetto agli assi.

```
l1 = -2
l2 = -1
plots = []
plots.append(plot_vector_field((l1*x,l2*y), (x,-1,1), (y, -1,1)))
```

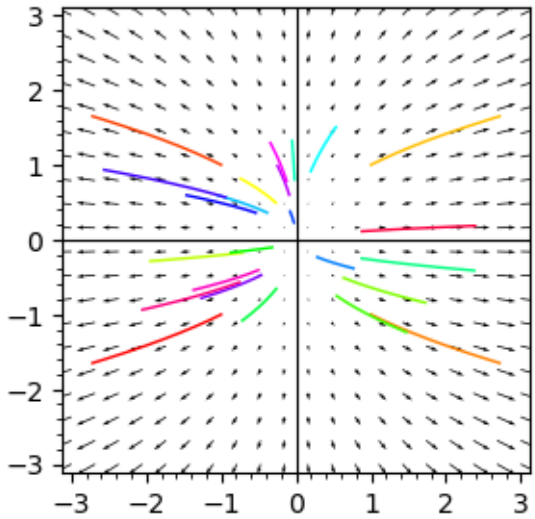
```
5
6   for y0_1,y0_2 in initial_conditions:
7       plots.append(parametric_plot((y0_1*e^(l1*t), y0_2*e^(l2*t)), (t,0,10.0), color=colors[(y0_1,y0_2)] ))
8   show(sum(plots))
```



$\lambda_1 > \lambda_2 > 0$ **nodo instabile**

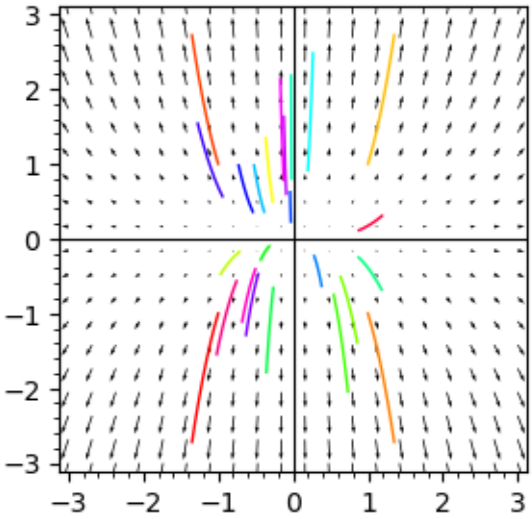
Se $\lambda > 0$ allora $\lim_{t\to\infty} e^{\lambda t} = \infty$; le traiettorie delle soluzioni si allontanano dal punto di equilibrio.

```
1   l1 = 1
2   l2 = 0.5
3   plots = []
4   plots.append(plot_vector_field((l1*x,l2*y), (x,-3,3), (y, -3,3), aspect_ratio=1))
5
6   for y0_1,y0_2 in initial_conditions:
7       plots.append(parametric_plot((y0_1*e^(l1*t), y0_2*e^(l2*t)), (t,0,1.0), color=colors[(y0_1,y0_2)] ))
8
9   show(sum(plots))
```



$\lambda_2 > \lambda_1 > 0$

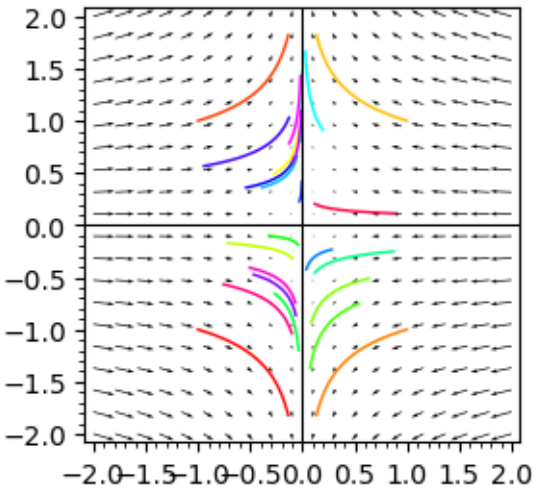
```
l1 = 0.3
l2 = 1
plots = []
plots.append(plot_vector_field((l1*x,l2*y), (x,-3,3), (y, -3,3), aspect_ratio=1))

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((y0_1*e^(l1*t), y0_2*e^(l2*t)), (t,0,1.0), color=colors[(y0_1,y0_2)] ))

show(sum(plots))
```



$\lambda_1 < 0 < \lambda_2$ **sella**

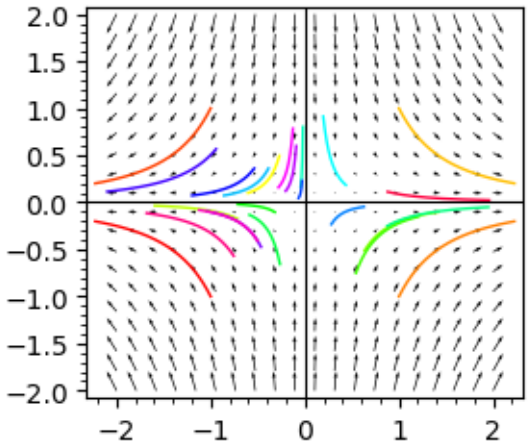
Quando uno degli autovalori è maggiore di zero e l'altro è minore, una delle soluzioni diverge mentre l'altra converge; si ha quindi una configurazione "a sella" lungo uno degli assi:

```
l1 = -1
l2 = 0.3

plots = []
plots.append(plot_vector_field((l1*x,l2*y), (x,-2,2), (y, -2,2), aspect_ratio=1))

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((y0_1*e^(l1*t), y0_2*e^(l2*t)), (t,0,2.0), color=colors[(y0_1,y0_2)] ))

show(sum(plots))
```



$\lambda_2 < 0 < \lambda_1$

```
l1 = 0.4
l2 = -0.8
plots = []
plots.append(plot_vector_field((l1*x,l2*y), (x,-2,2), (y, -2,2), aspect_ratio=1))

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((y0_1*e^(l1*t), y0_2*e^(l2*t)), (t,0,2.0), color=colors[(y0_1,y0_2)] ))

show(sum(plots))
```



Caso $\lambda_1 = \lambda_2 = \lambda$

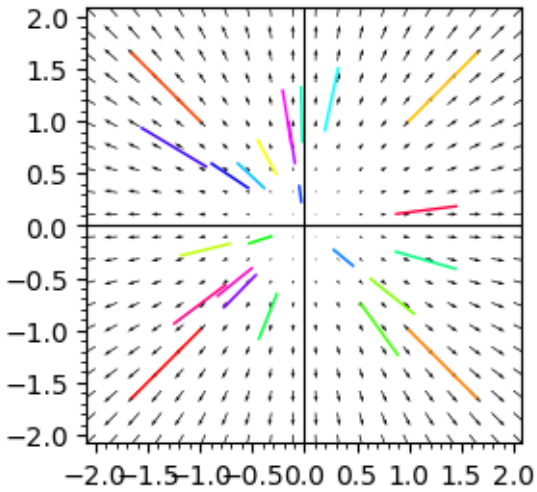
λ semisemplice > 0 stella instabile

Essendo λ semisemplice, ci sono due catene di Jordan linearmente indipendenti di lunghezza unitaria relative a questo autovalore. Trasformando come visto prima, si ottiene il sistema:

$$\begin{pmatrix} z_1' \\ z_2' \end{pmatrix} = \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

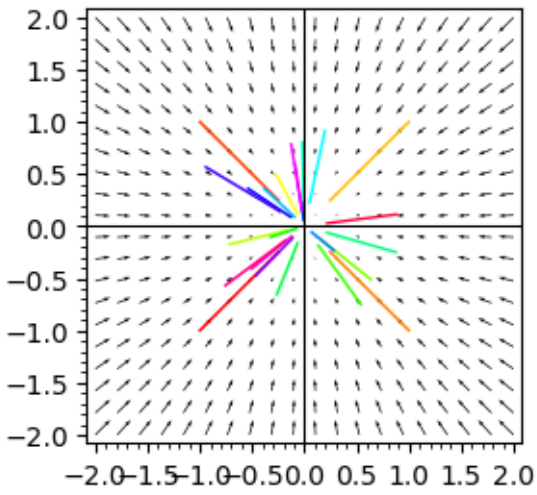
Le due equazioni pertanto sono indipendenti ma hanno esattamente lo stesso comportamento. Abbiamo quindi delle traiettorie a stella, convergenti o divergenti dal punto di equilibrio a seconda del valore di λ

```
l = 0.5
plots = []
plots.append(plot_vector_field((l*x,l*y), (x,-2,2), (y, -2,2), aspect_ratio=1))

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((y0_1*e^(l*t), y0_2*e^(l*t)), (t,0,1.0), color=colors[(y0_1,y0_2)] ))

show(sum(plots))
```



λ semisemplice < 0 stella stabile

```
l = -0.7
plots = []
plots.append(plot_vector_field((l*x,l*y), (x,-2,2), (y, -2,2), aspect_ratio=1))

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((y0_1*e^(l*t), y0_2*e^(l*t)), (t,0,2.0), color=colors[(y0_1,y0_2)] ))

show(sum(plots))
```



λ non semisemplice < 0 degenere stabile

Essendo λ non semisemplice, la forma di Jordan non disaccoppia più completamente le componenti, ma riflette la struttura a catena (che in questo caso, ha lunghezza massima 2).

$$\begin{pmatrix} z_1' \\ z_2' \end{pmatrix} = \begin{pmatrix} \lambda & 0 \\ 1 & \lambda \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

si hanno quindi le equazioni:

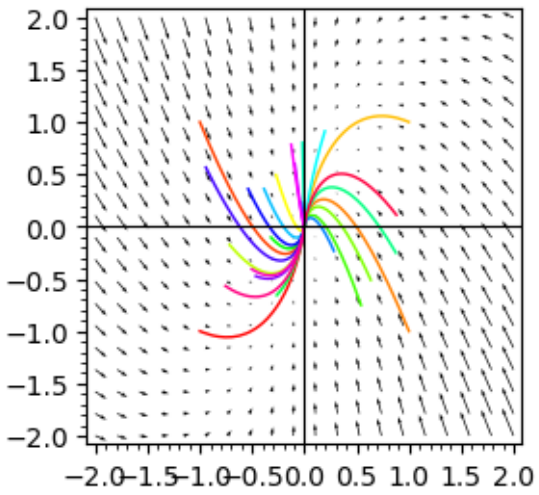
$$z_1'(t) = \lambda z_1(t), \quad z_2'(t) = z_1(t) + \lambda z_2(t)$$

che hanno soluzione:

$$z_1(t) = z_1(0)e^{\lambda t}, \quad z_2(t) = (z_1(0) + z_2(0)t)e^{\lambda t}$$

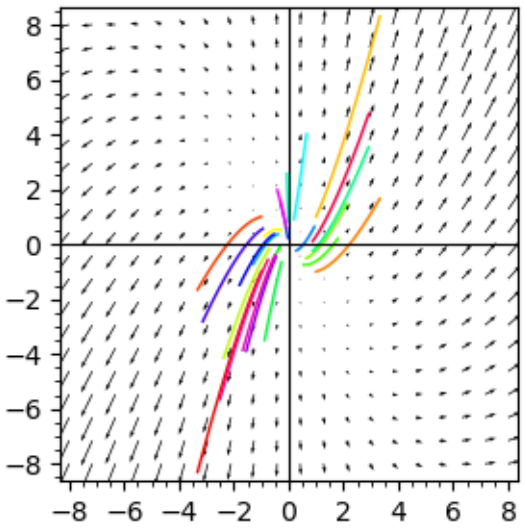
Se $\lambda < 0$ entrambe le soluzioni tendono al punto di equilibrio nell'origine, altrimenti divergono.

```
l = -0.7
plots = []
plots.append(plot_vector_field((l*x,x+l*y), (x,-2,2), (y, -2,2), aspect_ratio=1))

y_1 = lambda y0_1,y0_2,t: y0_1*e^(l*t)
y_2 = lambda y0_1,y0_2,t: (y0_1*t+y0_2)*e^(l*t)
for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((y_1(y0_1,y0_2,t), y_2(y0_1,y0_2,t)), (t,0,10.0),
    ↪  color=colors[(y0_1,y0_2)] ))

show(sum(plots))
```



λ non semisemplice > 0 degenere instabile

```
l = 0.8
plots = []
plots.append(plot_vector_field((l*x,x+l*y), (x,-8,8), (y, -8,8), aspect_ratio=1))

y_1 = lambda y0_1,y0_2,t: y0_1*e^(l*t)
y_2 = lambda y0_1,y0_2,t: (y0_1*t+y0_2)*e^(l*t)
for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((y_1(y0_1,y0_2,t), y_2(y0_1,y0_2,t)), (t,0,1.5), color=colors[(y0_1,y0_2)]
    ↪  ))

show(sum(plots))
```



Caso $\lambda_1 = \bar{\lambda_2} \to \lambda_{1/2} = (\alpha \pm i\beta) \in \mathbb{C}$

In questo caso la matrice di similitudine che diagonalizza A in forma di Jordan è a valori complessi; la diagonalizzazione pertanto non renderebbe lo studio delle soluzioni semplice come nei casi visti precendentemente.

Nel caso bidimensionale, si può dimostrare che una matrice con autovalori complessi coniugati è una matrice di scala-rotazione, ed esiste una matrice $T \in \mathbb{R}^{2\times 2}$ tale che:

$$T^{-1}AT = \begin{pmatrix} \alpha & -\beta \\ \beta & \alpha \end{pmatrix} = \hat{A}$$

che è la somma di una matrice di scala e una matrice di rotazione:

$$\alpha I + \beta \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \alpha J^0 + \beta J$$

Osserviamo che gli autovalori di J sono $\pm i$, e gli autovalori di $\hat{A}$ sono gli autovalori di J calcolati nel polinomio $p(z) = \alpha z^0 + \beta z^1$ in quanto $\hat{A}$ è un polinomio di J; essi sono pertanto $\alpha \pm i\beta$, ovvero $\hat{A}$ ha gli stessi autovalori di A.

Il sistema trasformato per similitudine è quindi:

$$\begin{cases} y_1' = \alpha y_1 - \beta y_2 \\ y_2' = \alpha y_2 + \beta y_1 \end{cases}$$

Rappresentando le componenti in coordinate polari, ovvero $y_1 = \rho\cos\theta$, $y_2 = \rho\sin\theta$ imponiamo l'equazione:

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} \rho'\cos\theta - \theta'\rho\sin\theta \\ \rho'\sin\theta + \theta'\rho\cos\theta \end{pmatrix} = \hat{A}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} \alpha\rho\cos\theta - \beta\rho\sin\theta \\ \beta\rho\cos\theta + \alpha\rho\sin\theta \end{pmatrix}$$

da cui, moltiplicando per y_1', y_2' rispettivamente per $\cos\theta, \sin\theta$ e sommando si ottengono:

$$\rho' = \alpha\rho$$

$$\theta' = \beta$$

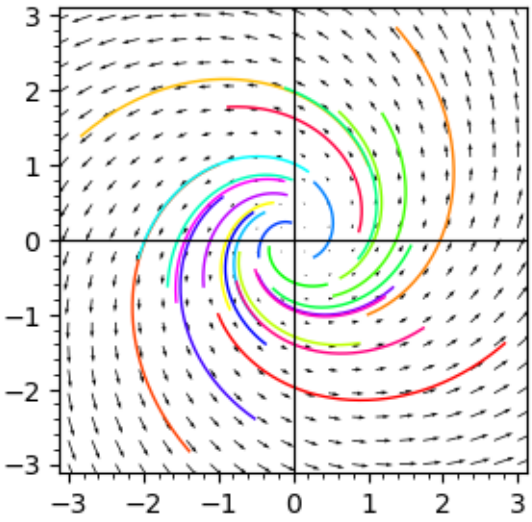
ovvero:

$$\rho(t) = \rho(0)e^{\alpha t}$$

$$\theta(t) = \theta(0) + \beta t$$

Le soluzioni hanno quindi delle traiettorie a spirale, che si avvicinano all'origine se $\alpha < 0$ o si allontanano altrimenti, o rimangono su un cerchio se $\alpha = 0$.

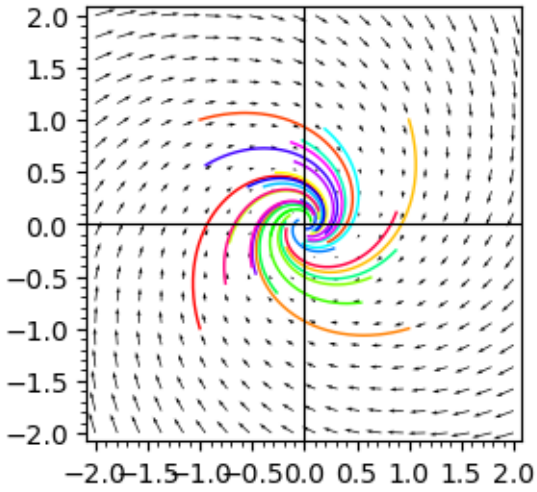
```
alpha = 0.8
beta = 1.9

plots = []
plots.append(plot_vector_field((alpha*x-beta*y,alpha*y+beta*x), (x,-3,3), (y, -3,3), aspect_ratio=1))

def cart_to_polar(x,y):
    rho = sqrt(x^2+y^2)
    theta = atan2(y,x)
    return rho, theta

def yf_1(y0_1,y0_2,t):
    rho_0, theta_0 = cart_to_polar(y0_1,y0_2)
    rho = rho_0*e^(alpha*t)
    theta = theta_0+beta*t
    return rho*cos(theta)

def yf_2(y0_1,y0_2,t):
    rho_0, theta_0 = cart_to_polar(y0_1,y0_2)
    rho = rho_0*e^(alpha*t)
    theta = theta_0+beta*t
    return rho*sin(theta)

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((yf_1(y0_1,y0_2,t),yf_2(y0_1,y0_2,t)), (t,0,1), color=colors[(y0_1,y0_2)]
    ↪  ))

show(sum(plots))
```



Caso $\lambda_1 = \bar{\lambda_2} \rightarrow \lambda_{1/2} = (\alpha \pm i\beta) \in \mathbb{C}$

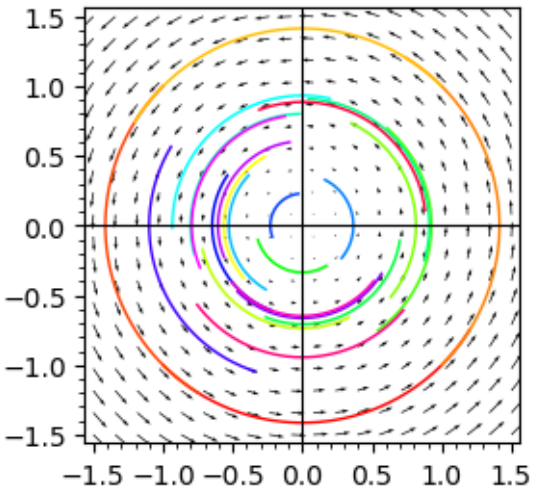
```
alpha = -0.8
beta = -1.5
```

```
plots = []
plots.append(plot_vector_field((alpha*x-beta*y,alpha*y+beta*x), (x,-2,2), (y, -2,2), aspect_ratio=1))

def cart_to_polar(x,y):
    rho = sqrt(x^2+y^2)
    theta = atan2(y,x)
    return rho, theta

def yf_1(y0_1,y0_2,t):
    rho_0, theta_0 = cart_to_polar(y0_1,y0_2)
    rho = rho_0*e^(alpha*t)
    theta = theta_0+beta*t
    return rho*cos(theta)

def yf_2(y0_1,y0_2,t):
    rho_0, theta_0 = cart_to_polar(y0_1,y0_2)
    rho = rho_0*e^(alpha*t)
    theta = theta_0+beta*t
    return rho*sin(theta)

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((yf_1(y0_1,y0_2,t),yf_2(y0_1,y0_2,t)), (t,0,2), color=colors[(y0_1,y0_2)]
    ↪  ))

show(sum(plots))
```




```
alpha = 0
beta = 0.6

plots = []
plots.append(plot_vector_field((alpha*x-beta*y,alpha*y+beta*x), (x,-1.5,1.5), (y, -1.5,1.5),
↪  aspect_ratio=1))

def cart_to_polar(x,y):
    rho = sqrt(x^2+y^2)
    theta = atan2(y,x)
    return rho, theta

def yf_1(y0_1,y0_2,t):
    rho_0, theta_0 = cart_to_polar(y0_1,y0_2)
    rho = rho_0*e^(alpha*t)
    theta = theta_0+beta*t
    return rho*cos(theta)

```

```
def yf_2(y0_1,y0_2,t):
    rho_0, theta_0 = cart_to_polar(y0_1,y0_2)
    rho = rho_0*e^(alpha*t)
    theta = theta_0+beta*t
    return rho*sin(theta)

for y0_1,y0_2 in initial_conditions:
    plots.append(parametric_plot((yf_1(y0_1,y0_2,t),yf_2(y0_1,y0_2,t)), (t,0,3), color=colors[(y0_1,y0_2)]
    ↪ ))

show(sum(plots))
```



14.2 Confronto tra metodi

Dato il sistema bidimensionale di equazioni differenziali lineari:

$$\mathbf{y}'(t) = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \mathbf{y}(t), \quad t \geq 0, \quad y_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

confrontare il ruolo dell'origine applicando il metodo di Eulero esplicito, Eulero implicito e quello dei trapezi.

Caso continuo

Gli autovalori di A sono le radici di $p(\lambda) = det(\lambda I - A) = \lambda^2 + 1$ ovvero $\lambda_{1/2} = \pm i$.
In quanto $\lambda_1 = \bar{\lambda_2}$ a parte reale nulla, come visto in precedenza le soluzioni sono stabili, ovvero dei cerchi intorno all'origine.
In questo caso si verifica facilmente che la soluzione è:

$$\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = \begin{pmatrix} cos(t) \\ -sin(t) \end{pmatrix}$$

infatti:

$$\begin{pmatrix} y_1'(t) \\ y_2'(t) \end{pmatrix} = A \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = \begin{pmatrix} -y_2(t) \\ y_1(t) \end{pmatrix} = \begin{pmatrix} sin(t) \\ cos(t) \end{pmatrix} = \frac{\partial}{\partial t} \begin{pmatrix} cos(t) \\ -sin(t) \end{pmatrix}$$

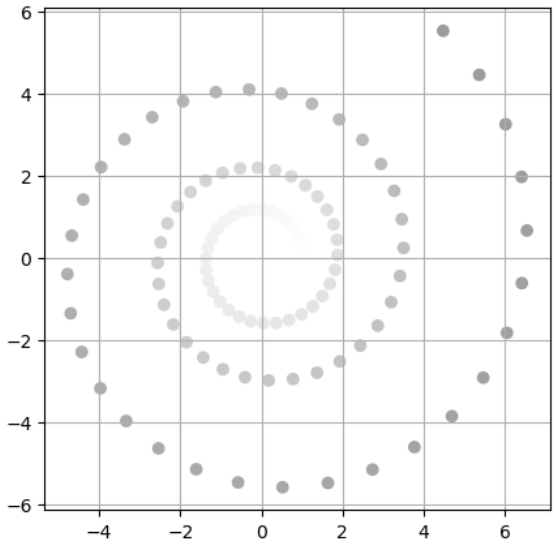
Metodo di Eulero esplicito: $y_{n+1} = y_n + hf_n$

In questo caso si ha:

$$y_{n+1} = y_n + hAy_n = (I + hA)y_n$$

In quanto $(I + hA) = p(A) = hA^1 + A^0$ e si dimostra che $\sigma(p(A)) = p(\sigma(A))$ (ovvero gli autovalori di un polinomio di matrice sono in corrispondenza biunivoca con i polinomi degli autovalori della matrice di partenza), si hanno $\lambda_{1/2} = p(\lambda_{1/2}) = 1 \pm ih$, che sono complessi coniugati ed hanno entrambi modulo > 1, quindi ci aspettiamo che la soluzione sia un fuoco instabile.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]=5,5
h=0.2
y = [vector([1,0])]
A = Matrix([[0,-1],[1,0]])
for i in range(100):
    y.append(y[-1] + h * A * y[-1])

cmap = plt.get_cmap('binary')
plt.scatter([p[0] for p in y], [p[1] for p in y], marker='o', color=cmap(range(len(y))))
plt.grid()

```



Metodo di Eulero implicito: $y_{n+1} = y_n + hf_{n+1}$

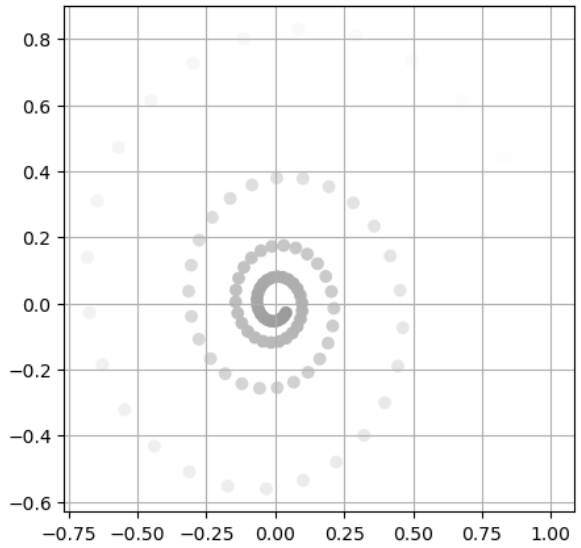
In questo caso si ha:

$$y_{n+1} = y_n + hAy_{n+1} = (I - hA)^{-1} y_n$$

Si ha $f(A) = (I - hA)^{-1}$, e si dimostra, analogamente al caso esplicito, che se gli autovalori hanno molteplicità unitaria, vale $\sigma(f(A)) = f(\sigma(A))$ e quindi $\lambda_{1/2} = (1 \pm hi)^{-1}$.

Questi hanno modulo < 1 e quindi ci aspettiamo che la soluzione sia un fuoco stabile.

```
h=0.25
y = [vector([1,0])]
A = Matrix([[0,-1],[1,0]])
B = (1 - h*A)^-1
for i in range(100):
    y.append(B * y[-1])

cmap = plt.get_cmap('binary')
plt.scatter([p[0] for p in y], [p[1] for p in y], marker='o', color=cmap(range(len(y))))
plt.grid()

```



Metodo dei trapezi: $y_{n+1} = y_n + \frac{h}{2}(f_n + f_{n+1})$

In questo caso si ha:

$$y_{n+1} = y_n + \frac{h}{2}(Ay_n + Ay_{n+1})$$

ovvero:

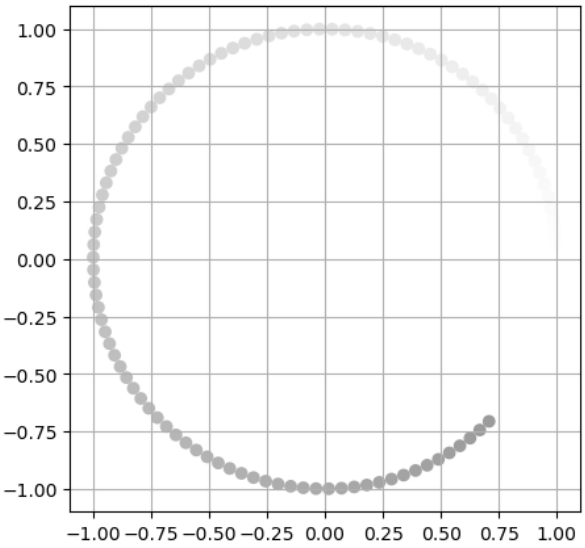
$$(1 - \frac{h}{2}A)y_{n+1} = (1 + \frac{h}{2}A)y_n$$

e quindi:

$$y_{n+1} = (1 - \frac{h}{2}A)^{-1}(1 + \frac{h}{2}A)y_n$$

Gli autovalori di $f(A) = (1 - \frac{h}{2}A)^{-1}(1 + \frac{h}{2}A)$ sono pertanto $\lambda_{1/2} = \frac{1 \pm ih/2}{1 \mp ih/2}$, i quali hanno entrambi modulo unitario e quindi ci aspettiamo che il sistema sia stabile, ma non asistoticamente.

```
h=0.055
y = [vector([1,0])]
A = Matrix([[0,-1],[1,0]])
B = (1 - h/2*A)^-1 * (1+h/2*A)
for i in range(100):
    y.append(B * y[-1])

cmap = plt.get_cmap('binary')
plt.scatter([p[0] for p in y], [p[1] for p in y], marker='o', color=cmap(range(len(y))))
plt.grid()

```



15 Modello preda-predatore

Consideriamo un ecosistema marino. Definiamo:

$$x(t) = \text{ numero delle prede}$$

$$y(t) = \text{ numero dei predatori}$$

Queste popolazioni sono regolate dal sistema di equazioni:

$$x'(t) = ax(t) - bx(t)y(t) \tag{15.1}$$

$$y'(t) = -cy(t) + dx(t)y(t)$$

in cui a, b sono rispettivamente il tasso di riproduzione e di predazione (e quindi diminuzione) delle prede, e c, d sono rispettivamente il tasso di mortalitá e di predazione (e quindi aumento) dei predatori. La predazione è supposta essere proporzionale alla probabilitá che due individui delle due popolazioni si incontrino, ed é perciò funzione del prodotto della cardinalitá delle due popolazioni. Manipolando il sistema si ottiene:

$$x' = x\,(a - by)$$

$$y' = -y\,(c - dx)$$

da cui si evince che ci sono due punti di equilibrio che annullano le derivate:

$$(0,0)^T, \quad \left(\frac{c}{d}, \frac{a}{b}\right)^T$$

Esprimendo (15.2) [p.93] come $\dot{s} = f(t,s)$ e linearizzando rispetto ad s nell'origine si ottiene:

$$\dot{s}(t) = f(t, s(t)) = f(t, \mathbf{0}) + J_f(t, \mathbf{0})s(t) + g(t, s(t))$$

in cui J_f è la matrice Jacobiana di f e g è il resto dello sviluppo di Taylor.

La matrice Jacobiana del sistema ha la forma:

$$J(x,y) = \begin{pmatrix} a - by & -bx \\ dy & -c + dx \end{pmatrix}$$

In particolare, nel punto di equilibrio $(0,0)^T$ si ha:

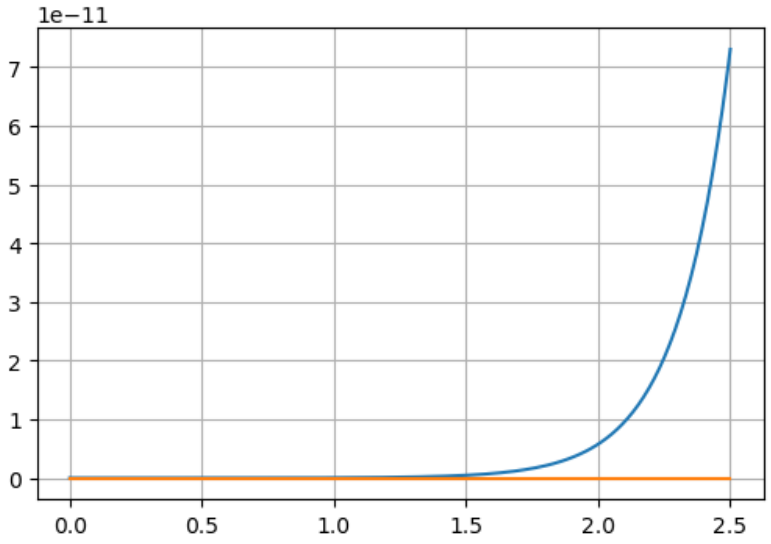
$$J(0,0) = \begin{pmatrix} a & 0 \\ 0 & -c \end{pmatrix}$$

i cui autovalori sono evidentemente $a, -c$ che non hanno entrambi parte reale negativa e pertanto il sistema (15.2) [p.93] non è stabile nell'origine ma ha una configurazione a sella.

In effetti, partendo da una perturbazione di $(0,0)^T$ e usando come coefficienti $a = 5, b = 2, c = 5, d = 2$, si ottiene l'andamento a sella in cui una componente della soluzione diverge mentre l'altra rimane stabile:

```
import numpy as np
from scipy.integrate import solve_ivp
from matplotlib import pyplot as plt

def f_prime_gen(a,b,c,d):
    def f_prime(t,s):
        return np.array(
        [
         s[0] * (a-b*s[1]),
         s[1] * (-c+d*s[0])
        ])
    return f_prime
```

```
eps = np.finfo(float).eps
s0 = np.array([0+eps,0+eps])
t0=0
T=2.5
fprime = f_prime_gen(5,2,5,2)
res = solve_ivp(fprime, (t0, T), s0, method='BDF', max_step=0.01)
plt.plot(res['t'], res['y'][0])
plt.plot(res['t'], res['y'][1])
plt.grid()
```



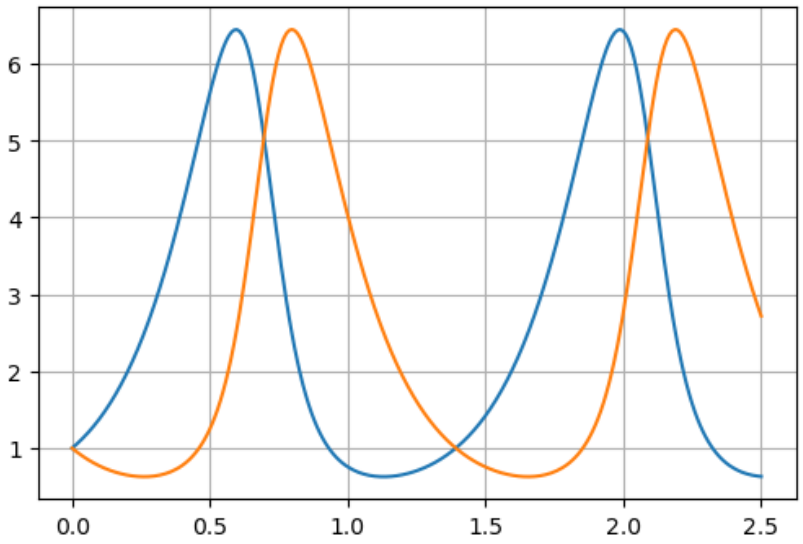
Esaminiamo adesso il comportamento del sistema nell'altro punto di equilibrio $\left(\frac{c}{d}, \frac{a}{b}\right)^T$.

La matrice Jacobiana diventa:

$$J\left(\frac{c}{d}, \frac{a}{b}\right) = \begin{pmatrix} 0 & -bc/d \\ da/b & 0 \end{pmatrix}$$

che ha polinomio caratteristico $p(\lambda) = \lambda^2 + ac$; i due autovalori sono pertanto $\lambda_{1/2} = \pm i\sqrt{ac}$. In questo caso gli autovalori sono complessi coniugati a parte reale nulle, e quindi questo punto di equilibrio è invece un centro.
Il teorema di Perron in questo caso non è applicabile in quanto la parte lineare non è uniformemente asintoticamente stabile; se la parte lineare però fosse dominante ci potremmo comunque aspettare un comportamento periodico in un intorno di questo punto di equilibrio.

```
s0 = np.array([1,1])
t0=0
T=2.5
fprime = f_prime_gen(5,2,5,2)
res = solve_ivp(fprime, (t0, T), s0, method='BDF', max_step=0.01)
plt.plot(res['t'], res['y'][0])
plt.plot(res['t'], res['y'][1])
plt.grid()
```



In questo caso particolare, è fortunatamente possibie studiare il piano delle fasi. Dividento membro a membro le due equazioni in (15.2) [p.93] si ottiene:

$$\frac{dy}{dx} = \frac{-y(c-dx)}{x(a-by)} \Rightarrow -\frac{dy}{y}(a-by) + \frac{dx}{x}(c-dx) = \left(\frac{a}{y} - b\right) dy + \left(\frac{c}{x} - d\right) dx = 0$$

Integrando membro a membro si ottiene:

$$(a\ln(y) - by) + (c\ln(x) - dx) = \kappa \Rightarrow a\ln(ye^{-\frac{b}{a}y}) + c\ln(xe^{-\frac{d}{c}x}) = \kappa$$

Possiamo adesso riparametrizzare ponendo:

$$\eta(x) = xe^{-\frac{d}{c}x}, \quad \xi(y) = ye^{-\frac{b}{a}y}$$

e quindi si ottiene:

$$a\ln(\xi(y)) + c\ln(\eta(x)) = \ln(\eta(x)^c\xi(y)^a) = \kappa$$

Esponenziando:

$$\eta(x)^c\xi(y)^a = e^\kappa > 0$$

In quanto però nel nostro modello $x, y \geq 0$, anche $\eta(x), \xi(y) \geq 0$. In particolare, entrambe η e ξ si annullanno nell'origine e all'infinito; in quanto le derivate:

$$\dot{\eta}(x) = e^{-\frac{d}{c}x}\left(1 - \frac{d}{c}x\right), \quad \dot{\xi}(y) = e^{-\frac{b}{a}y}\left(1 - \frac{b}{a}y\right)$$

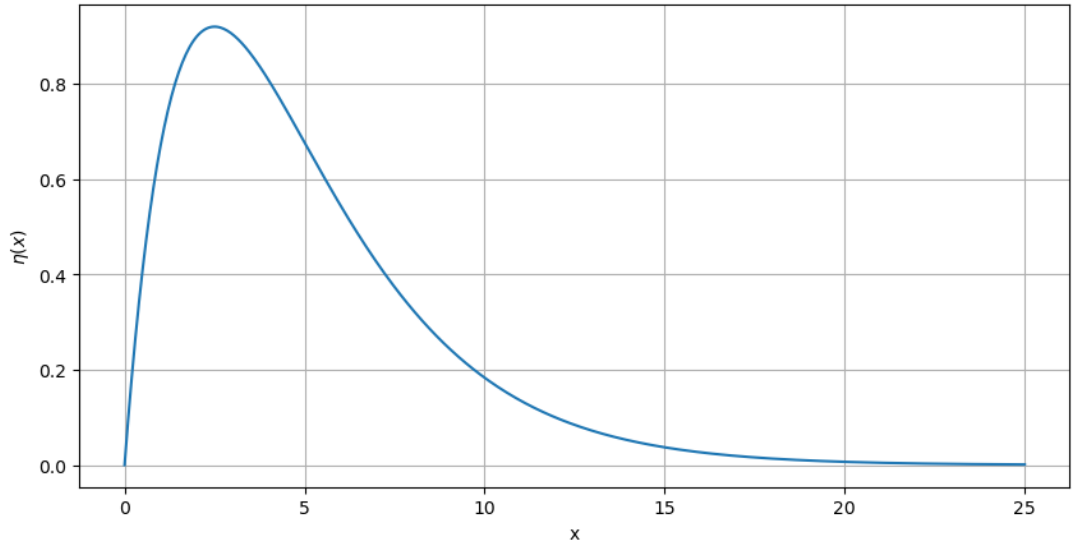
si annullano rispettivamente in $y = \frac{a}{b}$ e $x = \frac{c}{d}$, questi due saranno i rispettivi punti di massimo per le due funzioni. I valori di η e ξ sono quindi sempre compresi tra zero e i rispettivi massimi:

$$\max_y \xi(y) = \frac{a}{eb}, \quad \max_x \eta(x) = \frac{c}{ed}$$

Nello spazio η, ξ pertanto le soluzioni percorreranno orbite iperboliche secondo l'equazione $\eta^c\xi^a = K$.

Vediamo il grafico di $\eta(x)$, omologo a quello di $\xi(y)$:

```python
def xi(a,b,y):
    return y * np.exp(-(b/a)*y)

def eta(c,d,x):
    return x * np.exp(-(d/c)*x)

def xiprime(a,b,y):
    return np.exp(-(b/a)*y) * (1-(b/a)*y)

def etaprime(c,d,x):
    return np.exp(-(d/c)*x) * (1-(d/c)*x)

x = np.linspace(0,25,500)
y = eta(5,2,x)

plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('$\eta(x)$')
plt.grid()
```



Le traiettorie nello spazio η, ξ sono quindi iperboliche e limitate in entrambi gli assi: $\eta(x) \in [0, c/ed], \xi(y) \in [0, a/eb]$. Il punto $[0, 0]$ non fa parte della traiettoria iperbolica $\eta^c \xi^a = K$. In particolare, per $\eta(x)^c \xi(y)^a = K$ valgono entrambe:

$$\eta = (K\xi^{-a})^{1/c}, \quad \xi = (K\eta^{-c})^{1/a}$$

essendo esse funzioni monotone decrescenti rispettivamente di ξ e η, devono entrambe avere il minimo nel massimo valore dell'altra, pertanto:

$$\xi_{min} = (K\eta_{max}^{-c})^{1/a}, \quad \eta_{min} = (K\xi_{max}^{-a})^{1/c}$$

e come visto prima, in quanto sono entrambe funzioni di y e x rispettivamente, si hanno:

$$\xi_{max} = \frac{a}{eb}, \quad \eta_{max} = \frac{c}{ed}$$

Avere $\eta_{min}, \xi_{min} > 0$ implica che anche x, y hanno un rispettivo valore minimo e massimo che possono essere ricavati analiticamente invertendo le rispettive funzioni, come si evince dal grafico precedente.

Le traiettorie delle soluzioni pertanto scorrono su una porzione di iperbole nel piano η, ξ, e sono delle orbite chiuse nel piano x, y.

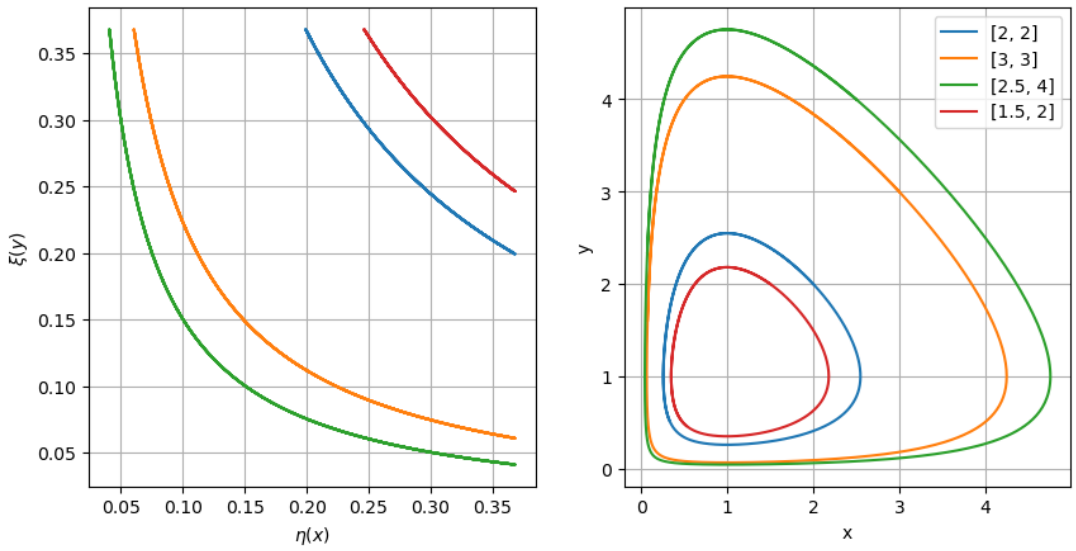
Al variare delle condizioni iniziali e a parita' di parametri a, b, c, d, le soluzioni sono una famiglia di iperboli $\eta^c \xi^a = K$ in cui K è univocamente determinato dalle condizioni iniziali, e si ricava direttamente per sostituzione:

$$K = \eta^c \xi^a = (x_0 e^{-\frac{d}{c} x_0})^c (y_0 e^{-\frac{b}{a} y_0})^a$$

Ponendo $a, b, c, d = 1$, ovvero imponendo un punto di equilibrio in $(1, 1)^T$, al variare del punto iniziale si ottengono le traiettorie:

```
plt.rcParams["figure.figsize"]=10,5
a,b,c,d = 1,1,1,1
s0 = np.array([1,1])
t0=0
T=10
fprime = f_prime_gen(a,b,c,d)
legend = list()
for s0 in [[2,2], [3,3], [2.5, 4],[1.5,2]]:
    res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
    plt.subplot(122)
    plt.plot(res['y'][0], res['y'][1])
    plt.subplot(121)
    plt.plot(eta(c,d,res['y'][0]), xi(a,b,res['y'][1]))
    legend.append(s0)

plt.subplot(121)
plt.grid()
plt.xlabel("$\eta(x)$")
plt.ylabel("$\\xi(y)$")
plt.subplot(122)
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.legend(legend)

```

```
<matplotlib.legend.Legend at 0x7f7c5bbe1ff0>
```

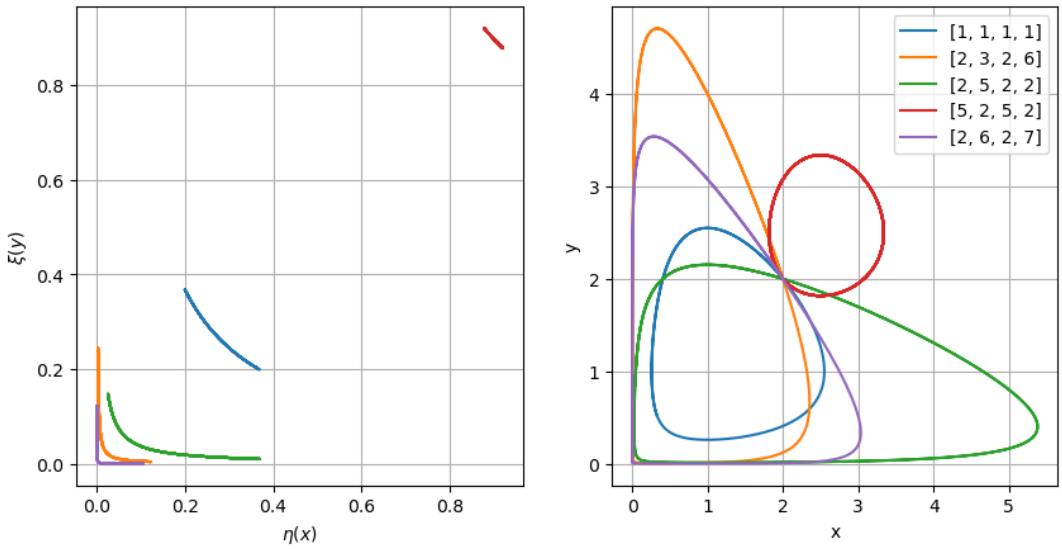


Lo stesso punto iniziale invece accomuna anche una famiglia di iperboli differenti per K e punto di equilibrio al variare dei parametri:

```
s0 = np.array([2,2])
t0=0
T=10

plt.rcParams["figure.figsize"]=10,5
legend = list()
for a,b,c,d in [[1,1,1,1], [2,3,2,6], [2,5,2,2], [5,2,5,2], [2,6,2,7]]:
    fprime = f_prime_gen(a,b,c,d)
    res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
    plt.subplot(122)
    plt.plot(res['y'][0], res['y'][1])
    plt.subplot(121)
    plt.plot(eta(c,d,res['y'][0]), xi(a,b,res['y'][1]))
```

```
        legend.append([a,b,c,d])


    plt.subplot(121)
    plt.grid()
    plt.xlabel("$\eta(x)$")
    plt.ylabel("$\\xi(y)$")
    plt.subplot(122)
    plt.grid()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(legend)

```

```
<matplotlib.legend.Legend at 0x7f7c5b49e890>
```



Abbiamo quindi che in un intorno del punto di equilibrio $\left(\frac{c}{d}, \frac{a}{b}\right)^T$, le traiettorie seguono un'orbita periodica; chiamiamo T il periodo dell'orbita.

I valori medi di prede e predatori in un generico periodo T, $\bar{x}, \bar{y}$, possono essere calcolati come segue:

$$\bar{x} = \frac{1}{T}\int_0^T x(t)dt, \quad \bar{y} = \frac{1}{T}\int_0^T y(t)dt,$$

Osserviamo che, in quanto il moto è periodico, $x(0) = x(T)$. Osserviamo che:

$$\frac{dx(t)}{dt} = x(t)(a - by(t)) \Rightarrow \frac{dx(t)}{x(t)} = (a - by(t))dt$$

Abbiamo quindi:

$$\frac{1}{T}\int_0^T \frac{x'(t)}{x(t)}dt = \frac{1}{T}\int_0^T (a - by(t))dt =$$

da cui:

$$\frac{1}{T}\ln x(t)|_0^T = \frac{at}{T}|_0^T - \frac{b}{T}\int_0^T y(t)dt$$

e quindi, ricordando che $x(0) = x(T)$:

$$\frac{1}{T}(ln(x(T)) - ln(x(0))) = 0 = a - \frac{b}{T}\int_0^T y(t)dt$$

Abbiamo quindi che:

$$\frac{1}{T}\int_0^T y(t)dt = \bar{y} = \frac{a}{b}$$

e con passaggi del tutto analogi si ricava che:

$$\frac{1}{T}\int_0^T x(t)dt = \bar{x} = \frac{c}{d}$$

Il punto di equilibrio $\left(\frac{c}{d}, \frac{a}{b}\right)^T$ è pertanto anche il valor medio attorno al quale le orbite si chiudono.

Ne consegue anche che il rapporto medio tra predatori e prede è dato da:

$$\frac{\bar{y}}{\bar{x}} = \frac{ad}{bc}$$

Cosa succede se introduciamo una perturbazione alle popolazioni prede e predatori, come ad esempio un'attività di pesca? Ipotizzando che la perturbazione riduca entrambe le popolazioni di una frazione ϵ:

$$x' = x\left(a(\epsilon) - by\right), \quad a(\epsilon) = a - \epsilon$$

$$y' = -y\left(c(\epsilon) - dx\right), \quad c(\epsilon) = c + \epsilon$$

Abbiamo quindi che il punto di equilibrio e il rapporto predatori/prede si spostano in funzione di ϵ:

$$p_e = \left(\frac{c(\epsilon)}{d}, \frac{a(\epsilon)}{b}\right)^T, \quad q(\epsilon) = \frac{\bar{y}}{\bar{x}} = \frac{a(\epsilon)d}{bc(\epsilon)} = \frac{a-\epsilon}{c+\epsilon}\frac{d}{b}$$

Osserviamo che la derivata del rapporto predatori/prede:

$$q'(\epsilon) = \frac{d}{b}\frac{-(c+\epsilon) - (a-\epsilon)}{(c+\epsilon)^2} = \frac{d}{b}\frac{-c-a}{(c+\epsilon)^2} \leq 0$$

è negativa, pertanto è una funzione decrescente. Il rapporto predatori/prede quindi tende a diminuire con l'incremento della pesca.

A riprova di questo fatto, nel grafico che segue si osserva chiaramente la diminuzione del rapporto medio predatori/prede all'aumentare di ϵ.

Quando ϵ inizia a diventare paragonabile al parametro a, la popolazione dei predatori si avvicina sempre piu' e per periodi sempre più lunghi all'estinzione; il rapporto predatori prede tenderà a zero:

$$\lim_{\epsilon \to a} q(\epsilon) = 0$$

Inoltre, la matrice Jacobiana nell'origine diventa:

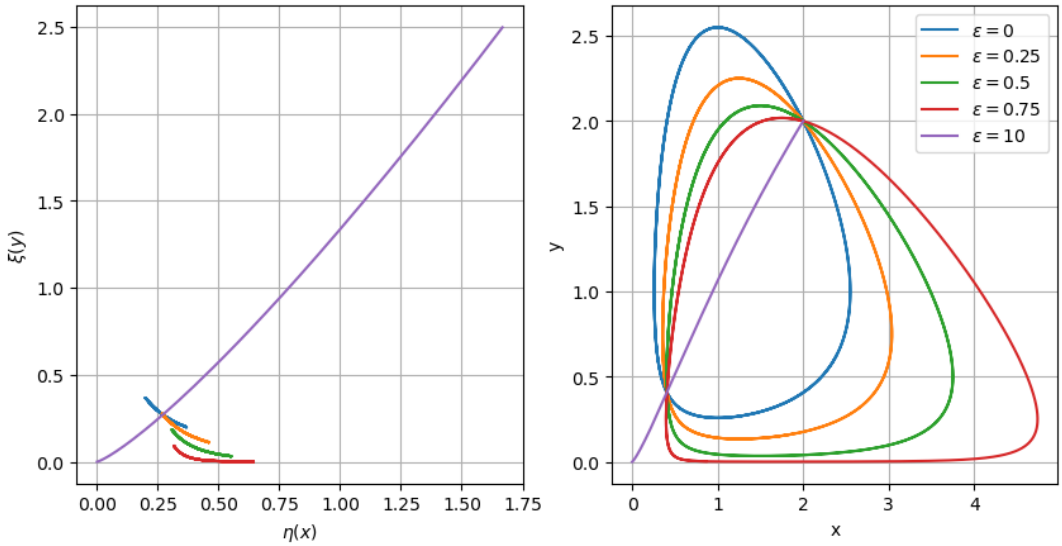
$$J(0,0) = \begin{pmatrix} a(\epsilon) & 0 \\ 0 & -c(\epsilon) \end{pmatrix} = \begin{pmatrix} a-\epsilon & 0 \\ 0 & -c-\epsilon \end{pmatrix}$$

ovvero, se $\epsilon > a$, l'origine, che era un punto di sella, diventa un nodo stabile ovvero entrambe le specie si estinguono.

```
def coeffs(eps):
    return 1-eps,1,1+eps,1
legend = list()
t0=0
T=20
a,b,c,d = coeffs(0)
fprime = f_prime_gen(a,b,c,d)
legend.append('$\epsilon=0$')
res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
# plt.plot(res['y'][0], res['y'][1])
plt.subplot(122)
plt.plot(res['y'][0], res['y'][1])
plt.subplot(121)
plt.plot(eta(c,d,res['y'][0]), xi(a,b,res['y'][1]))

```

```
a,b,c,d = coeffs(0.25)
fprime = f_prime_gen(a,b,c,d)
legend.append('$\epsilon=0.25$')
res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
# plt.plot(res['y'][0], res['y'][1])
plt.subplot(122)
plt.plot(res['y'][0], res['y'][1])
plt.subplot(121)
plt.plot(eta(c,d,res['y'][0]), xi(a,b,res['y'][1]))

a,b,c,d = coeffs(0.5)
fprime = f_prime_gen(a,b,c,d)
legend.append('$\epsilon=0.5$')
res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
# plt.plot(res['y'][0], res['y'][1])
plt.subplot(122)
plt.plot(res['y'][0], res['y'][1])
plt.subplot(121)
plt.plot(eta(c,d,res['y'][0]), xi(a,b,res['y'][1]))

a,b,c,d = coeffs(0.75)
fprime = f_prime_gen(a,b,c,d)
legend.append('$\epsilon=0.75$')
res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
# plt.plot(res['y'][0], res['y'][1])
plt.subplot(122)
plt.plot(res['y'][0], res['y'][1])
plt.subplot(121)
plt.plot(eta(c,d,res['y'][0]), xi(a,b,res['y'][1]))

a,b,c,d = coeffs(10)
fprime = f_prime_gen(a,b,c,d)
legend.append('$\epsilon=10$')
res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
# plt.plot(res['y'][0], res['y'][1])
plt.subplot(122)
plt.plot(res['y'][0], res['y'][1])
plt.subplot(121)
plt.plot(eta(c,d,res['y'][0]), xi(a,b,res['y'][1]))

plt.subplot(121)
plt.grid()
plt.xlabel("$\eta(x)$")
plt.ylabel("$\\xi(y)$")
plt.subplot(122)
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.legend(legend)
```

```
<matplotlib.legend.Legend at 0x7f7c5994faf0>
```



15.1 Interpretazione economica

Lo stesso modello può essere applicato alla dinamica del mercato del lavoro, interpretando con:

$$x(t) = \text{ forza lavoro disponibile (disoccupati)}$$

$$y(t) = \text{ capitale di investimento}$$

Abbiamo quindi che anche la dinamica di occupazione investimenti seguirà orbite chiuse intorno ai rispettivi valori medi che sono determinati dai parametri del sistema; in quanto il punto iniziale dell'orbita determina l'ampiezza delle oscillazioni, sarebbe opportuno che gli interventi legislativi fossero indirizzati a spostare occupazione e investimenti verso i rispettivi valori medi per stabilizzare il mercato dell'occupazione.

Il modello può essere adattato a simulare un mercato globalizzato. In particolare, consideriamo il caso in cui gli investimenti siano spendibili nel mercato del lavoro locale oppure nel mercato remoto di paesi emergenti:

$$x_l(t) = \text{ forza lavoro disponibile paesi sviluppati (locale)}$$

$$x_r(t) = \text{ forza lavoro disponibile paesi emergenti (remoto)}$$

$$y(t) = \text{ capitale di investimento}$$

Il sistema di equazioni diventa:

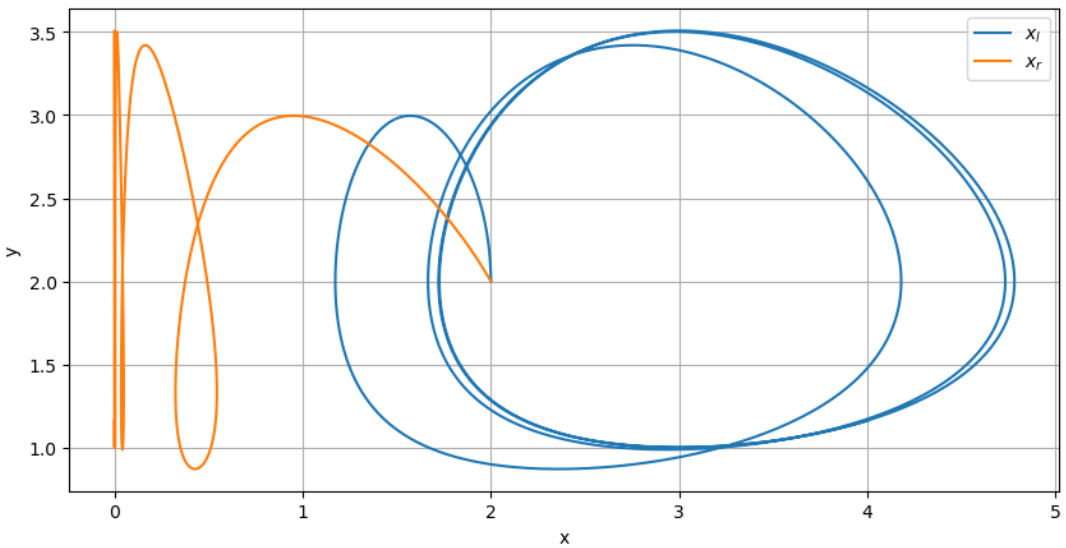
$$\begin{aligned} x_l'(t) &= a_l x_l(t) - b_l x_l(t) y(t) = x_l(t)(a_l - b_l y(t)) \\ x_r'(t) &= a_r x_l(t) - b_r x_r(t) y(t) = x_r(t)(a_r - b_r y(t)) \\ y'(t) &= -cy(t) + dx(t)y(t) = -y(t)(c - d_l x_l(t) - d_r x_r(t)) \end{aligned} \tag{15.2}$$

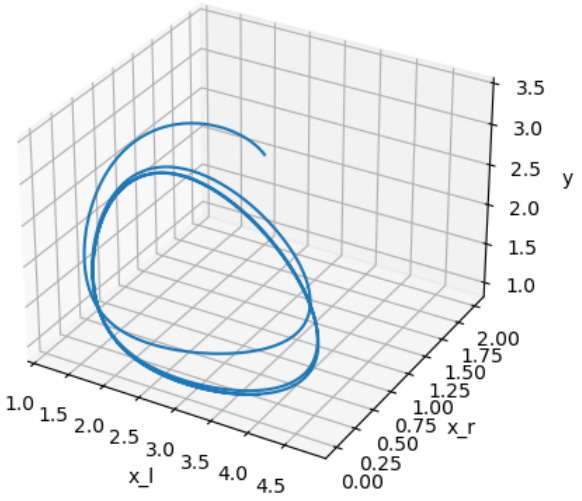
in cui adesso i coefficienti a_l, a_r continuano ad essere i parametri di riproduzione, e possiamo anche assumere $a_l \approx a_r$; c è rappresenta l'erosione del capitale dovuta al pagamento del lavoro. b_l, b_r rappresentano la facilità di impiego: al crescere di b aumenta la probabilità di connettere investimenti e lavoratori, creando impiego e quindi riducendo la forza lavoro disponibile. d_l, d_r infine rappresentano i rendimenti sul capitale del personale impiegato. In generale possiamo quindi aspettarci che $b_r < b_l$ in quanto i lavoratori dei paesi emergenti, avendo meno diritti, sono più facilmente impiegabili (meno burocrazia etc). Allo stesso tempo, ci possiamo aspettare che $d_r > d_l$ in quanto i guadagni sugli investimenti sono normalmente maggiori nei paesi emergenti, a fronte ovviamente di stipendi e diritti minori.

```
def f_prime_gen(al,ar,c, bl, br, dl, dr):
    def f_prime(t,s):
        return np.array(
        [
```

```
            s[0] * (al-bl*s[2]),
            s[1] * (ar-br*s[2]),
            s[2] * (-c+dl*s[0]+dr*s[1])
           ])
    return f_prime

al, ar, c = 2,2,3
bl, br = 1,1.5
dl, dr = 1,1.5
fprime = f_prime_gen(al,ar,c, bl, br, dl, dr)

t0=0
T=10
s0 = [2,2,2]

res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
plt.plot(res['y'][0], res['y'][2])
plt.plot(res['y'][1], res['y'][2])
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['$x_l$', '$x_r$'])

plt.figure()
ax=plt.axes(projection='3d')
ax.plot3D(res['y'][0], res['y'][1], res['y'][2])
plt.xlabel('x_l')
plt.ylabel('x_r')
ax.set_zlabel('y')

```

```
Text(0.5, 0, 'y')
```





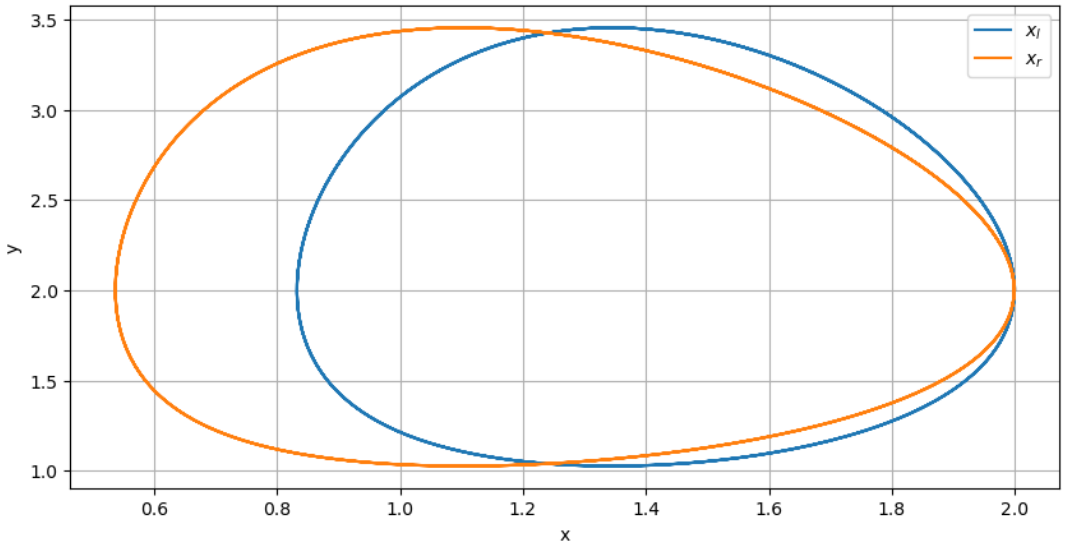
Dall'andamento della simulazione si evince che, grazie alle condizioni migliori per gli investitori nei paesi emergenti, la forza lavoro di questi ultimi viene velocemente saturata e tale rimane; nei paesi sviluppati invece si tende ad assestarsi sul normale ciclo periodico e quindi il problema della disoccupazione rimane.

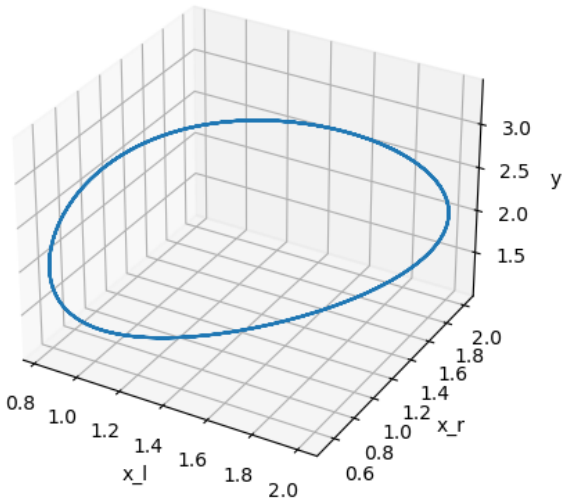
Se si avesse $a_r >> a_l$, ovvero l'aumento della popolazione del paese emergente ffosse sufficentemente più veloce di quello del paese sviluppato, allora si avrebbero oscillazioni nell'occupazione anche nei paesi emergenti.

Questa condizione non è ovviamente sostenibile, in quanto è naturale che il tasso di riproduzione tenda a scendere nei paesi poveri con l'aumentare della ricchezza; la conseguente diminuzione di a_r riporterebbe pertanto il sistema nella condizione precedente di saturazione della forza lavoro nel paese emergente.

```
al, ar, c = 2,3,3
bl, br = 1,1.5
dl, dr = 1,1.5
fprime = f_prime_gen(al,ar,c, bl, br, dl, dr)

t0=0
T=10
s0 = [2,2,2]

res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
plt.plot(res['y'][0], res['y'][2])
plt.plot(res['y'][1], res['y'][2])
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['$x_l$', '$x_r$'])

plt.figure()
ax=plt.axes(projection='3d')
ax.plot3D(res['y'][0], res['y'][1], res['y'][2])
plt.xlabel('x_l')
plt.ylabel('x_r')
ax.set_zlabel('y')
```

```
Text(0.5, 0, 'y')
```



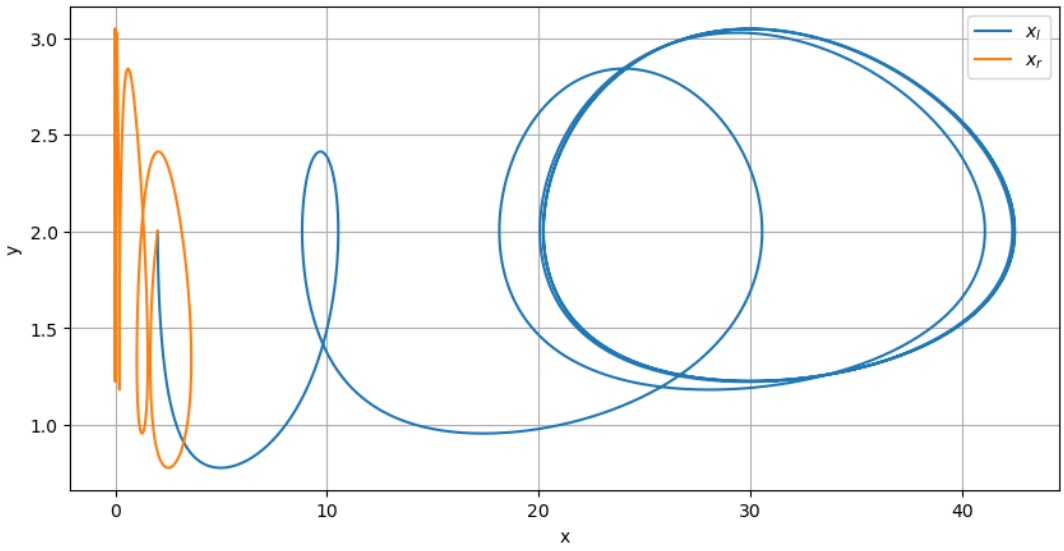


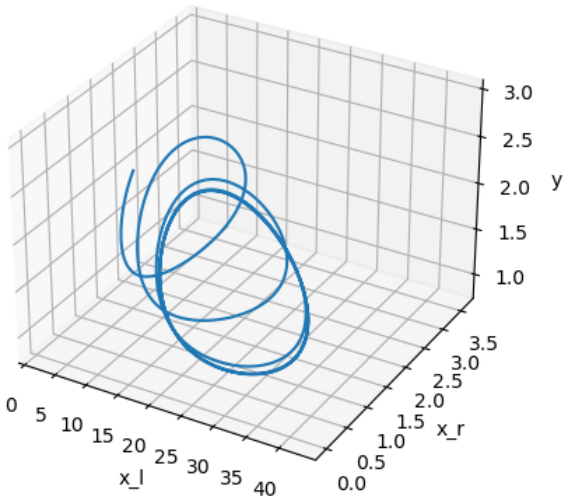
Quando il margine di profitto d si abbassa, come succede in effetti nei paesi sviluppati all'umentare del welfare, il numero medio di disoccupati tende ad aumentare:

```
al, ar, c = 2,2,3
bl, br = 1,1.5
dl, dr = 0.1,1
fprime = f_prime_gen(al,ar,c, bl, br, dl, dr)

t0=0
T=15
s0 = [2,2,2]

res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
plt.plot(res['y'][0], res['y'][2])
plt.plot(res['y'][1], res['y'][2])
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['$x_l$', '$x_r$'])

plt.figure()
ax=plt.axes(projection='3d')
ax.plot3D(res['y'][0], res['y'][1], res['y'][2])
plt.xlabel('x_l')
```

```
    plt.ylabel('x_r')
    ax.set_zlabel('y')
```

```
Text(0.5, 0, 'y')
```



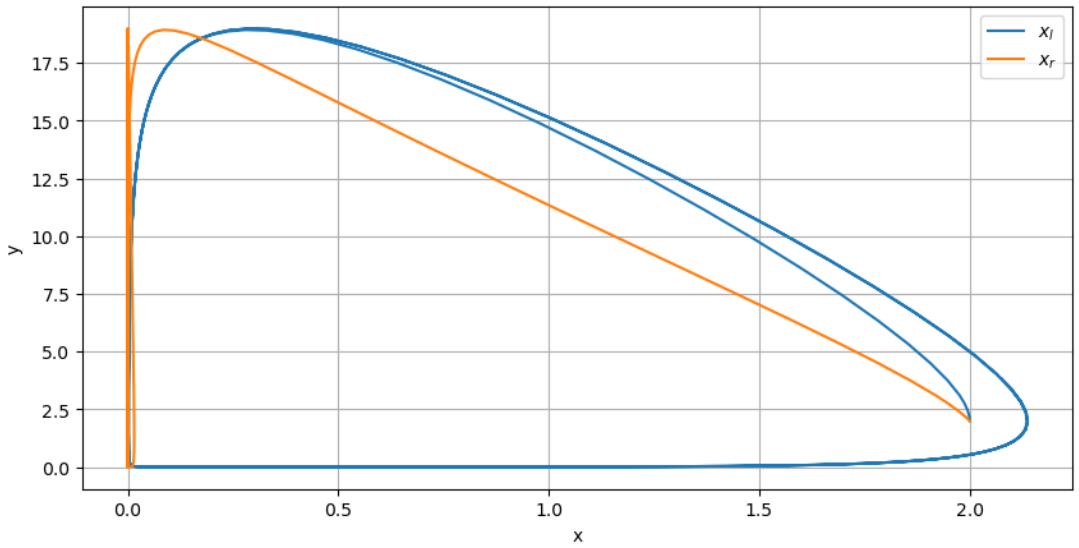


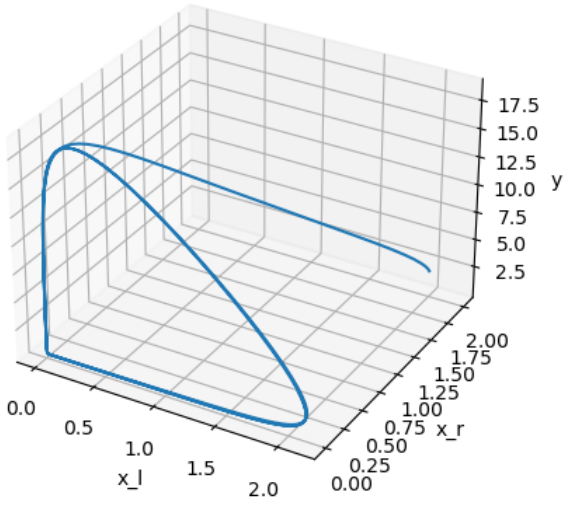
Anche se il margine di profitto nei paesi sviluppati fosse molto piu' grande di quello dei paesi emergenti ($d_l >> d_r$, cosa evidentemente impossibile), la facilità di impiego ($b_r > b_l$) in questi ultimi implica comunque la saturazione del mercato del lavoro, lasciando quindi la stessa dinamica ciclica nei paesi sviluppati.

```
al, ar, c = 2,2,3
bl, br = 1,1.5
dl, dr = 10,1
fprime = f_prime_gen(al,ar,c, bl, br, dl, dr)

t0=0
T=15
s0 = [2,2,2]

res = solve_ivp(fprime, (t0, T), np.array(s0), method='BDF', max_step=0.01)
plt.plot(res['y'][0], res['y'][2])
plt.plot(res['y'][1], res['y'][2])
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
```

```
16  plt.legend(['$x_l$', '$x_r$'])
17
18  plt.figure()
19  ax=plt.axes(projection='3d')
20  ax.plot3D(res['y'][0], res['y'][1], res['y'][2])
21  plt.xlabel('x_l')
22  plt.ylabel('x_r')
23  ax.set_zlabel('y')
```

```
Text(0.5, 0, 'y')
```





Riferimenti bibliografici

[1] Luigi Brugnano, Dispense per il corso di modelli numerici per la simulazione

[2] Donato Trigiante, Dispense per il corso di metodi di approssimazione

[3] V. Lakshmikantham, D. Trigiante, Theory of difference equations: numerical methods and applications

[4] Ronald E. Mickens, Difference equations: Theory, applications and advanced topics

[5] David Poole, Linear algebra: a modern introduction

[6] Meyer C. D., Matrix analysis and applied linear algebra

[7] Gilbert Strang, Linear algebra and its applications

[8] Nicholas J. Higham, Functions of Matrices, theory and computation

[9] Saber Elaydi, An introduction to difference equations

[10] A. Spitzbart, A Generalization of Hermite's Interpolation Formula, The American Mathematical Monthly, Vol. 67, No, 1 (Jan. 1960) pp. 42-46

[11] I. Gohberg, P. Lancaster, L. Rodman, Matrix Polynomials, Society of Industrial and Applied Mathemathics

[12] Carl de Boor Divided Differences Surveys in Approximation Theory 1 (2005) 46–69 arXiv:math/0502036 [math.CA]