# TESINA DI CALCOLO NUMERICO 2

FEDERICO PALA

*Prof. Giuseppe Rodriguez*

SOMMARIO. In questa tesina si vogliono analizzare i vari algoritmi per la risoluzione numerica di equazioni e sistemi di equazioni non lineari. In particolare verranno esposti e testati i metodi di bisezione, Newton, corde, secanti, Broyden, Gauss-Seidel e Newton-Jacobi.

## 1. INTRODUZIONE

Un'equazione non lineare può essere scritta nella forma

$$f(x) = 0$$

La funzione può avere una o più radici, ma supponiamo che ci sia un metodo che ci consenta di trovare un intervallo al cui interno vi sia una sola radice.

I metodi numerici utilizzati per la risoluzione di tali equazioni sono iterativi, quindi del tipo:

$$x_{k+1} = \psi(x_k) \quad \text{con } x_0 \text{ il punto iniziale}$$

Si vuole quindi costruire una successione

$$x_0\ x_1\ x_2\ \ldots\ x_k\ \rightarrow\ \alpha$$

dove $f(\alpha) = 0$.

La convergenza dipende oltre che dal metodo utilizzato anche dal punto iniziale scelto. Se denotiamo l'errore al passo $k$ con $e_k = x_k - x$, un metodo converge se

$$\lim_{k\to\infty} |e_k| = 0$$

è importante stabilire un ordine di convergenza $p$ tale per cui:

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^p} = C$$

con $C$ una costante. Si tratta di una forma indeterminata del tipo $\frac{0}{0}$: se si riesce a trovare una costante $p$ tale per cui gli ordini di infinitesimo si bilanciano allora il metodo è di ordine $p$. Per grandi valori di $k$:

$$|e_{k+1}| \approx C|e_k|^p$$

raggiunto un valore $k > \overline{k}$ avremo che $|e_k| < 1$

- se $p > 1$, $C$ può essere una costante qualsiasi (l'errore si riduce più velocemente all'aumentare dell'ordine);
- se $p = 1$, l'errore non si riduce grazie a $p$, quindi dev'essere $C < 1$;
- se $p = 1$ e $C = 0$ il metodo si dice *superlineare*.

*Key words and phrases.* Equazioni non lineari, Sistemi di equazioni non lineari, Metodo di bisezione, Newton, corde, secanti, Broyden, Gauss-Seidel, Newton-Jacobi.

## 2. Un'applicazione nell'analisi dei sistemi dinamici

Un problema classico nell'analisi dei sistemi è quello della stabilità dei sistemi retroazionati. Consideriamo la funzione di trasferimento ad anello aperto, che è una funzione complessa di variabile complessa $s = \sigma + j\omega$:

$$G(s) = \frac{10(1+s)e^{-2s}}{s(1+5s)}$$

per trovare la risposta armonica del sistema si pone $s = j\omega$ ottenendo

$$G(j\omega) = \frac{10(1+j\omega)e^{-2j\omega}}{s(1+5j\omega)}$$

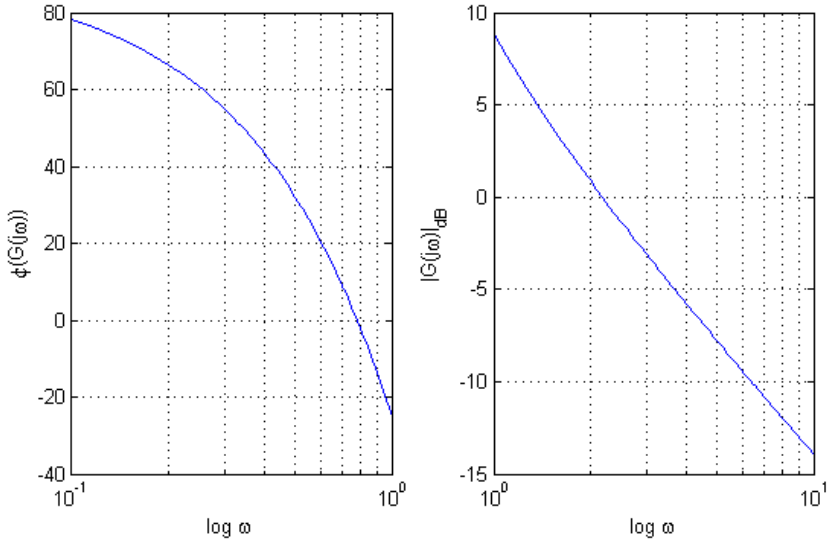
una proprietà importante di questa funzione è la pulsazione per cui la sua fase tocca $-\pi$ e quella per cui il modulo in decibel tocca lo zero:

$$\phi = \arctan\omega - 2\omega\frac{180}{\pi} - 90 - \arctan\frac{\omega}{0.2} + 180 = 0$$

$$|G_p(j\omega)|_{dB} = 20\log 10 + 20\log\sqrt{1+\omega^2} - 20\log\omega - 20\log\sqrt{1+\left(\frac{\omega}{0.2}\right)^2} = 0$$

```
function y = eqfase(x)
y=atan(x)-2*x*180/pi-90-atan(x/0.2)+180;

function y = eqmodulo(x)
y = 20*log10(10)+20*log10(sqrt(1+x.^2))-20*log10(x)
    -20*log10(sqrt(1+(x/0.2).^2));
```



Tramite questi due parametri si riescono a ricavare i margini di fase e di guadagno che saranno utili alla valutazione della stabilità del sistema con retroazione unitaria. Queste due equazioni possono essere utilizzate come esempio per l'applicazione dei metodi per la risoluzione di equazioni non lineari.

## 3. Metodo di bisezione

Quello di bisezione è un metodo molto semplice per trovare gli zeri di una funzione $f(x)$. E' necessario prima di tutto fissare un intervallo $[a, b]$ all'interno del quale vi sia una sola soluzione.

La presenza di uno zero implica un cambio di segno, per cui si può procedere calcolando la funzione al centro $c$ di tale intervallo, e verificando quale tra le due metà $[a, c]$ e $[c, b]$ contenga ancora un cambiamento di segno. Questa metà conterrà ancora la radice, per cui il metodo può essere applicato ricorsivamente a questo nuovo intervallo.

Il procedimento prosegue fino a che l'intervallo preso in considerazione diventa così stretto da approssimare la radice con la precisione desiderata. Per vedere se c'è un cambiamento di segno nell'intervallo $[a, b]$ basta verificare che $f(a)f(b) < 0$. Si dimostra che l'ordine del metodo è $p = 1$ e $C = \frac{1}{2}$. Una possibile implementazione è la seguente:

```
% funfcn è la funzione da analizzare:
% può essere passata come funzione matlab
% oppure in formato testuale
% [a,b] è l'intervallo preso in considerazione
% tao definisce il criterio di stop
% Nmax è il massimo numero di iterazioni
function [res k] = bisezione(funfcn,a,b,tao,Nmax)

f = fcnchk(funfcn);
fa = f(a); fb = f(b);
if(sign(fa)*sign(fb)>=0)
    error('zero o un numero pari di radici rilevate')
end
c = (a+b)/2; fc = f(c);
k = 0;

while (abs(b-a)>=tao) && (k<=Nmax) && (fc~=0)
    k = k + 1;
    if(sign(fa)*sign(fc)<0)
        b = c;
    else
        a = c; fa = fc;
    end
    c = (a+b)/2; fc = f(c);
end

res = c;
```

Si noti che l'onere computazionale consiste sostanzialmente nella valutazione ad ogni passo della funzione in un punto. Come criterio di stop si è considerato anche il caso in cui la funzione nel punto medio sia nulla. Ciò è stato fatto perchè in caso contrario avremmo già la soluzione e al passo successivo sarebbe stato segnalato un errore. Infine, per evitare il possibile underflow nell'operazione $f(a) * f(b)$, sono stati considerati i loro segni separatamente.

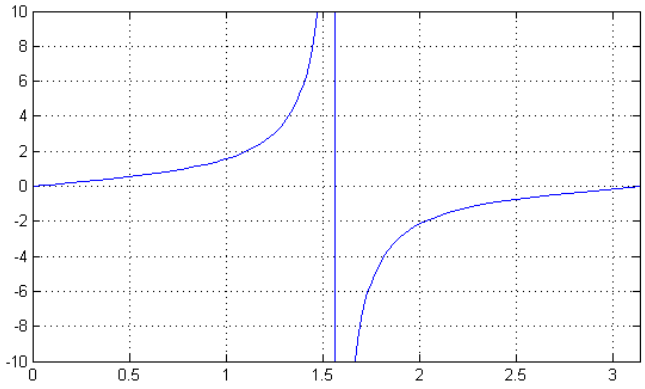
Proviamo ad applicare il metodo alle equazioni del problema della sezione 2:

```
[res1 k1] = bisezione(@eqfase,0.1,1,1e-8,100);
[res2 k2] = bisezione(@eqmodulo,1,10,1e-8,100);
```

Con la prima equazione il metodo converge a $0,7797$[1] in 27 passi, mentre la seconda converge a 2.1896 in 30 passi. Allargando gli intervalli presi in considerazione di due decadi a destra e due decadi a sinistra il numero di iterazioni necessarie diventa rispettivamente 34 e 37.

Un'altra caratteristica del metodo è che converge anche ai punti di singolarità. Ad esempio nel caso della tangente, anche se nell'intervallo $\left[-\frac{\pi}{4}, \frac{3}{4}\pi\right]$ non sono presenti degli zeri, converge a $\frac{\pi}{2}$:

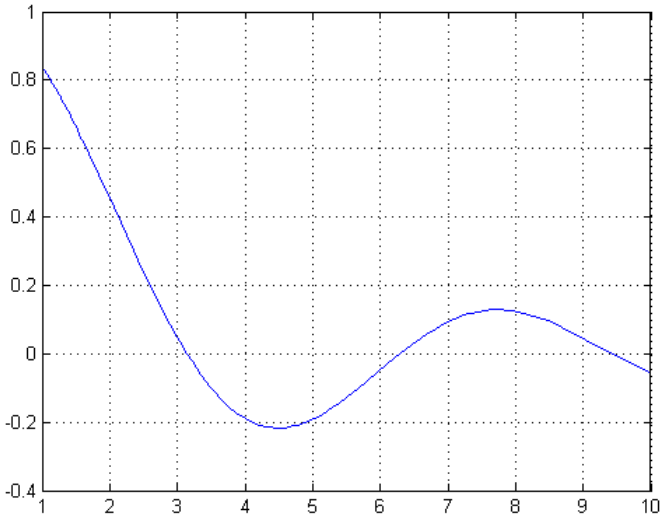
```
[res k] = bisezione('tan(x)',pi/4,pi/2+pi/4,1e-8,100);
```




Il metodo converge a 1.5708 in 28 passi. Alla prima iterazione l'algoritmo si ritrova a fare la tangente di $\frac{\pi}{2}$ che in matlab, essendo il pi greco arrotondato, non da infinito ma un numero molto grande: $1.6331 \cdot 10^{16}$. A questo punto va a considerare l'intervallo a destra visto che il risultato è positivo e dunque all'aumentare del numero di iterazioni convergerà a $\frac{\pi}{2}$ da destra, ovvero per valori decrescenti di x.

Vediamo infine cosa succede se è presente più di uno zero con la funzione $\frac{\sin x}{x}$. Se consideriamo ad esempio l'intervallo [1 10] saranno presenti tre radici:

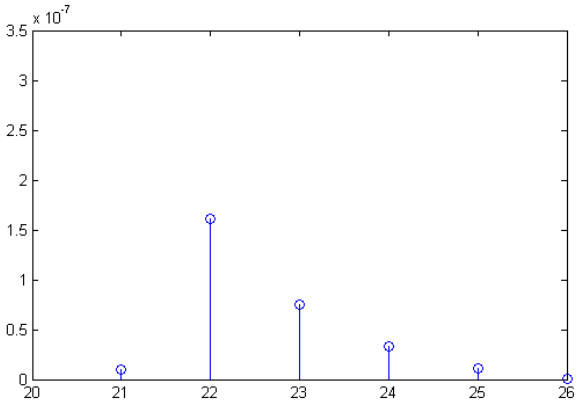
[1]per brevità, in questa relazione evito di scrivere tutte le cifre significative




Una volta che l'intervallo viene diviso in due, il cambio di segno sta nella parte sinistra mentre le altre due radici sono mascherate nella parte destra. Di conseguenza il metodo convergerà alla radice in $x = \pi$. Se invece l'intervallo considerato fosse $[-3\ 10]$ il metodo avrebbe restituito la radice in $x = 3\pi$. In entrambi i casi il metodo converge in 30 passi se richiediamo una precisione di $10^{-8}$ Proviamo infine a vedere cosa succede all'errore:

```
for i = 1:53
    [res(i) k] = bisezione('sin(x)/x',1,10,1e-15,i);
    err(i)=(res(i)-pi)/pi;
end
stem(err);
```

Lasciando perdere il grafico nella sua interezza, consideriamo la seguente parte:



Si può notare che l'errore non necessariamente diminuisce col numero di iterazioni, in altre parole non è vero che $|e_{k+1}| < |e_k|$ per qualsiasi k. Analizzando i dati più nello specifico, si può notare che ci vogliono circa 3 iterazioni per aumentare la precisione di una cifra decimale.

In conclusione possiamo dire che il metodo non è molto efficiente dal momento che il metodo seguente convergerà molto più in fretta. Nonostante ciò la sua semplicità, stabilità e la possibilità di studiare funzioni discontinue (quindi robustezza) lo rendono comunque interessante.

## 4. Metodo di Newton

Il metodo di Newton, come quello di bisezione, richiede una sola radice nell'intervallo preso in considerazione. Si parte da un punto $x_0$ abbastanza vicino alla radice. Poi si considera la retta tangente della funzione in questo punto e si prende $x_1$ pari alla sua intercetta con l'asse $x$. A questo punto il procedimento prosegue ricavando la retta tangente alla funzione corrispondente al punto $x_1$ e chiamando $x_2$ l'intercetta con l'asse $x$. La procedura continua poi iterativamente fino a convergere.

Per trovare la tangente della funzione in un punto bisogna conoscerne la derivata. Questo può essere un problema, specie nel caso in cui la funzione è misurata e non si conosce analiticamente. Il metodo iterativo è il seguente:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Si può dimostrare che l'ordine è $p = 2$ se la radice è semplice. In questo caso si tratta però di convergenza locale, e non globale: il metodo di Newton converge se prendiamo il punto iniziale abbastanza vicino alla radice. Nel caso di radici multiple, cioé quando $f'(\alpha) = 0$, diventa $p = 1$ e il metodo dunque rallenta. Come criterio di stop è stato utilizzato quello di Cauchy assieme al vincolo sul massimo numero di iterazioni. Il costo computazionale della procedura sta nel calcolo della funzione in due punti a ogni passo e nella necessaria conoscenza della derivata. Una possibile implementazione dell'algoritmo è la seguente:

```
function [res k] = newton(funfcn,derfcn,x0,tao,Nmax)

f = fcnchk(funfcn); d = fcnchk(derfcn);
if(f(x0)==0)
    k = 0; res = x0; return;
end
k = 1;
xold = x0; x = xold-f(xold)/d(xold);
while (abs(x-xold)>=tao*x) && (k<=Nmax)
    k = k + 1;
    xold = x; x = xold-f(xold)/d(xold);
end

res = x;
```

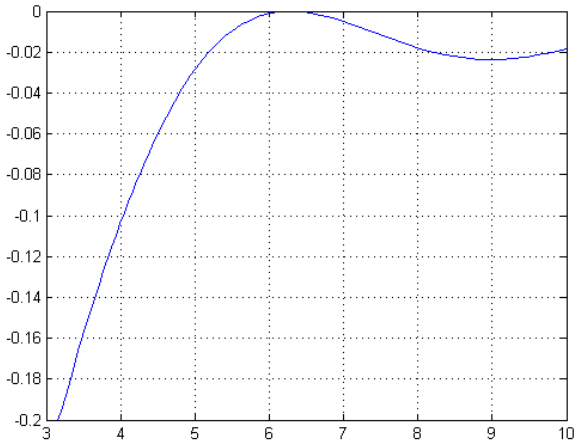
Si può verificare che le equazioni non lineari dell'applicazione sui sistemi lineari vengono risolte con Newton in un numero molto minore di passi:

```
[res1 k1] = newton(@eqfase,@derfase,0.5,10e-8,100)
[res2 k2] = newton(@eqmodulo,@dermodulo,2,10e-8,100)

res1 = 0.7797
k1 = 6

res2 = 2.1896
k2 = 4
```

Proviamo a considerare ad esempio la funzione $\frac{\cos x - 1}{x^2}$ nell'intervallo [3 10] che contiene una radice doppia in $2\pi$ e dunque dovrebbe convergere più lentamente ($p = 1$):



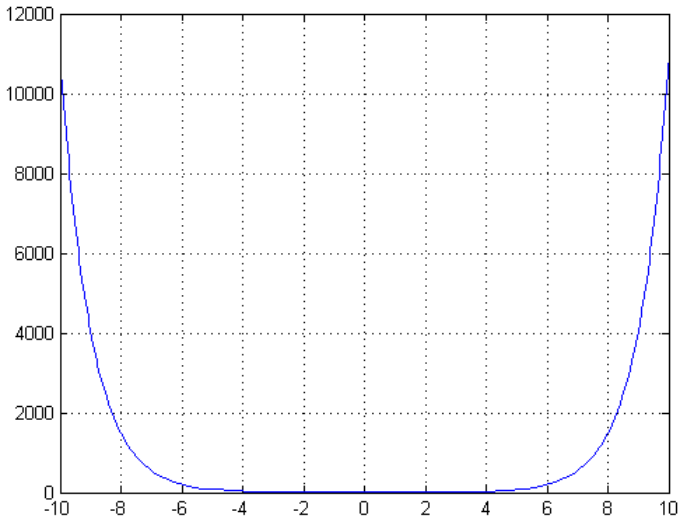

```
[res k] = newton(@(x)(cos(x)-1)/x^2,@(x)(1-sin(x)-cos(x))/x^2,3,10e-8,100)

res = 6.2832

k = 28
```

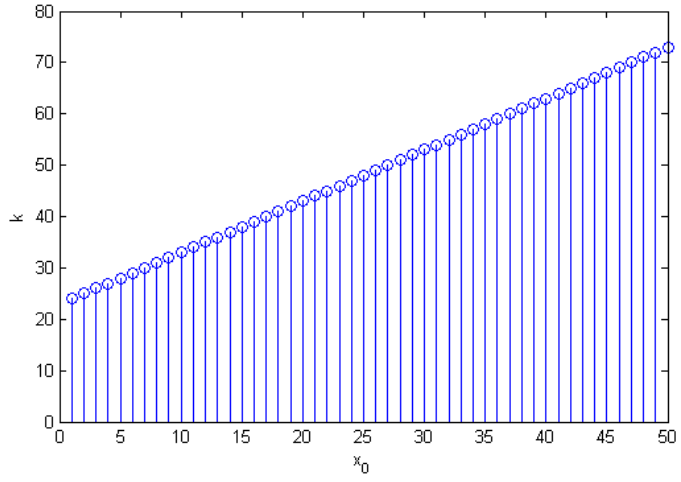
Il metodo converge alla radice commettendo un errore di $\frac{2\pi - \text{res}}{2\pi} = 1.1687 \cdot 10^{-8}$. Bisogna stare attenti a quale radice converge perchè l'algoritmo non prende in considerazione un intervallo ma solo il punto iniziale. Se infatti prendessimo come punto iniziale $x_0 = 1.4$ otterremmo la radice in 0, mentre con 1.5 otterremmo quella in $-4\pi$. Ciò vuol dire che il metodo non trova necessariamente la radice più vicina al punto iniziale.

Proviamo ora ad analizzare la funzione $\cosh(x) - 1$ che ha una radice con molteplicità doppia in zero e quindi anche questo dovrebbe essere lento a convergere:




Fissiamo come criterio di stop $10^{-8}$ e vediamo come varia il numero di iterazioni al variare del punto iniziale:

```
for i=1:710
   [res k(i)] = newton(@(x)(cosh(x)-1),@(x)sinh(x),i,10e-8,10000);
end
stem(k)
```



Partendo da 1, sono necessarie 24 iterazioni. All'aumentare di un'unità il punto iniziale, curiosamente è sempre necessaria un'iterazione in più. Questo avviene fino a che non diventa $x_0 = 711$ e matlab approssima la funzione con infinito. In corrispondenza di $x_0 = 710$ sono necessarie 733 iterazioni. Se ci avviciniamo a zero, ad esempio con $x_0 = 0.01$ sono necessarie ancora 17 iterazioni.

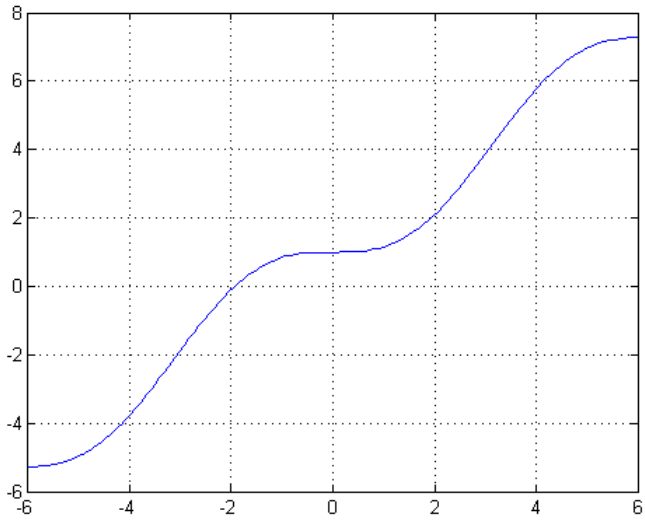
Vediamo qualche caso in cui il metodo di Newton fallisce. Iniziamo con la funzione $1 + x - \sin(x)$ che ha una sola radice ed è semplice:

```
[res k] = newton(@(x)1+x-sin(x),@(x)1-cos(x),0,10e-8,100)

res = NaN

k = 2
```

Questo avviene perchè la funzione in 0 vale 1, mentre la derivata si annulla. Di conseguenza l'incremento $\frac{f(x)}{f'(x)}$ non risulta definito. Questo è un caso limite, ma da problemi anche se la derivata non è proprio nulla, ma piccolissima: ad esempio $1 + (1 + 10^{-6})x - \sin x$ sarà molto lenta a convergere. Ad esempio ci vogliono 130 passi se ci fermiamo a $10^{-8}$.




Se partiamo invece da un punto diverso, ad esempio da 1, convergerà tranquillamente a 1.9346 in 6 passi.

Infine consideriamo il caso della funzione $\sqrt{|x|}$. Se partiamo dal punto $x_0 = 1$ succede qualcosa di particolare: al primo passo otteniamo $1 - \frac{1}{\frac{1}{2}} = -1$ mentre al secondo $-1 - \frac{1}{-\frac{1}{2}} = 1$. Di conseguenza entriamo in un ciclo infinito e il metodo non converge.

In conclusione possiamo dire che quello di Newton è un metodo molto potente e che in generale la convergenza è quadratica. Però ci sono anche parecchi problemi;

prima di tutto la necessità di calcolare in maniera analitica la derivata: non sempre è possibile farlo. Il metodo dipende molto dal punto iniziale, quindi dobbiamo avere una buona informazione a priori su dove si trovi la soluzione. C'è da dire inoltre che diventa lento nel caso di radici multiple e che la derivata deve comportarsi bene nell'intorno della radice.

## 5. Metodo delle corde

Questo è uno dei metodi quasi Newton. In pratica il funzionamento è lo stesso del metodo di Newton, ma al posto della derivata ci mettiamo una costante:

$$x_{k+1} = x_k - \frac{f(x_k)}{C}$$

Ad esempio C può essere fissata col valore della derivata nel primo punto. Il metodo funziona molto male rispetto agli altri; se scegliessimo una costante $C$ opportuna il metodo diverrebbe di ordine 1 quindi tanto vale usare il metodo di bisezione. Il metodo può essere però interessante per la sua applicabilità al caso multidimensionale.

```
% n=0 calcola la derivata solo una volta
% n=1 calcola la derivata sempre
% n=k calcola la derivata ogni k iterazioni
function [res k] = corde(funfcn,derfcn,x0,n,tao,Nmax)

f = fcnchk(funfcn); d = fcnchk(derfcn);
if(f(x0)==0)
    k = 0; res = x0; return;
end
k = 1;
c = d(x0);
xold = x0; x = xold-f(xold)/c;
while (abs(x-xold)>=tao) && (k<=Nmax)
    k = k + 1;
    xold = x;
    if(n~=0)
        if(mod(k-1,n)==0) c = d(xold); end
    end
    x = xold-f(xold)/c;
end

res = x;
```

In questo algoritmo si richiede per comodità l'espressione analitica della derivata, ma in realtà non è necessario: si potrebbe utilizzare una funzione che restituisce solamente la derivata nel punto richiesto.

Iniziamo a testare il metodo con le equazioni sul modulo e la fase della funzione trasferimento. Proviamo a imporre che $C$ sia pari alla derivata nel punto iniziale e di non aggiornarla:

```
[res k] = corde(@eqfase,@derfase,0.5,0,10e-8,100)

res = 0.7797
k = 6

[res k] = corde(@eqmodulo,@dermodulo,2,0,10e-8,100)

res = 2.1896
k = 8
```

Il metodo converge rispettivamente in 6 e 8 passi. Se invece $n = 1$ il metodo diventa tale e quale a quello di Newton. Aumentando la frequenza di aggiornamento si ottiene che la prima equazione richiede sempre 6 passi mentre la seconda ne richiede 5 ($n = 2, 3$) 6 ($n = 4$) 7 ($n = 5$) 8 ($n = 6, 7$). Una funzione più complicata, già studiata col metodo di Newton è la seguente: $f(x) = \frac{\cos x - 1}{x^2}$:

```
[res k] =
 corde(@(x)(cos(x)-1)/x^2,@(x)(1-sin(x)-cos(x))/x^2,3,0,10e-8,20000)

res = 6.2819

k = 12690
```

Oltre al fatto che è necessario fare un mucchio di iterazioni, l'errore è considerevolmente maggiore. Infatti mentre col metodo di Newton l'errore assoluto era di $7.3431 \cdot 10^{-8}$, in questo caso invece è di 0.0013: c'è già un errore alla terza cifra decimale! Si possono ottenere risultati migliori utilizzando un n diverso da zero. All'aumentare di n il metodo converge sempre in più passi mentre l'errore ha un comporamento più strano:

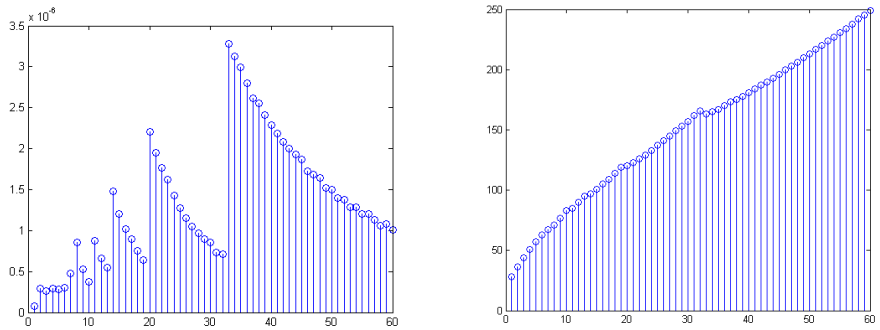



diagramma errore/n e k/n

## 6. Metodo delle secanti

Come il metodo delle corde, anche questo è un metodo di quasi Newton. In questo caso la derivata viene approssimata col rapporto incrementale:

$$x_{k+1} = x_k - \frac{f(x_k)}{m_k}$$

$$m_k = \frac{f(x_k) - f(x_{k+1})}{x_k - x_{k+1}}$$

```
function [res k] = secanti(funfcn,a,b,tao,Nmax)

f = fcnchk(funfcn);
k = 0;

fa = f(a); fb = f(b);
if(fa==0)
    res = a;
    return
end
if(fb==0)
    res = b;
    return
end
k = 1;
x = b - ((b-a)/(fb-fa))*fb;
while (abs(x-b)>=tao) && (k<=Nmax)
    a = b; b = x;
    fa = fb;
    fb = f(b);
    x = b-((b-a)/(fb-fa))*fb;
    k = k + 1;
end

res = x;
k = k-1;
```

Rispetto al metodo delle corde richiede un punto in più, inoltre ad ogni passo va calcolato un rapporto. Ci si guadagna però in una più alta velocità di convergenza in quanto il metodo è superlineare. Si dimostra che l'ordine risulta $p = \frac{1+\sqrt{5}}{2} \approx 1.618$ che corrisponde alla sezione aurea. Come primo test si può provare a risolvere il problema della sezione 2:

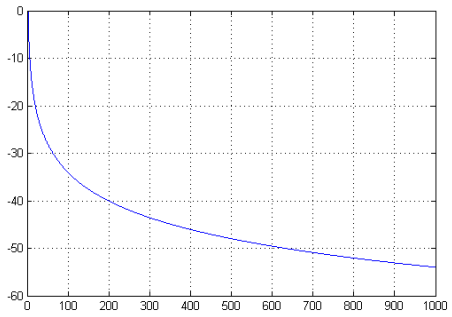
```
[res1 k1] = secanti(@eqfase,0.001,100,1e-8,100)
res1 = 0.7797
k1 = 4
```

```
[res2 k2] = secanti(@eqmodulo,0.01,1000,1e-8,100)
res2 = 3.4171e+132 -1.9587e+132i
k2 = 101
```

Mentre il modulo converge bene in 4 passi, il secondo non converge. Aumentando il numero di iterazioni addirittura tende a restituire NaN. Il problema in questo caso è l'intervallo preso in considerazione. Considerando un intervallo più ristretto infatti:

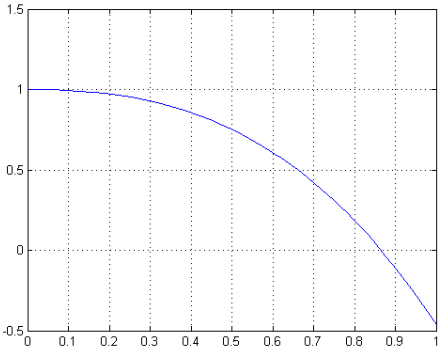
```
[res2 k2] = secanti(@eqmodulo,1,10,1e-8,100)
res2 = 2.1896 - 0.0000i
k2 = 12
```

Ci si può rendere conto del problema visualizzando la funzione in scala lineare:




Alla prima iterazione la soluzione va a 597.1068. All'itearazione successiva invece va a $-3.8556 \cdot 10^3$. In questo punto la funzione non è valutata nel suo campo di definizione: Matlab restituirà una soluzione complessa, che tende a divergere invalidando tutto l'algoritmo. Se invece l'intervallo venisse ristretto, alla seconda iterazione si uscirebbe di poco dal campo di definizione e il metodo riuscirebbe a convergere. Considerando infine l'intervallo da 2 a 10 la soluzione ad ogni iterazione non esce dal campo di definizione e il metodo converge in 8 passi.

Come ultimo esempio si può considerare l'equazione: $\cos(x) - x^3$:




Se come criterio di stop viene utilizzato l'eps di Matlab, il metodo converge a 0.8655 in 7 passi. In tabella è visualizzato l'errore assoluto rispetto a questo valore:

| errore | k |
|---|---|
| 0.0241 | 1 |
| $-0.0049$ | 2 |
| $1.1573 \cdot 10^{-4}$ | 3 |
| $5.4645 \cdot 10^{-7}$ | 4 |
| $-6.1397 \cdot 10^{-11}$ | 5 |
| 0 | 6 |
| 0 | 7 |

Si può vedere che la precisione ad ogni iterazione aumenta molto velocemente.

## 7. Confronto fra i metodi

Consideriamo prima di tutto l'equazione sulla fase. Come soluzione esatta consideriamo quella che risulta dall'applicazione di Newton con la precisione massima possibile:

```
[soluzione k] = newton(@eqfase,@derfase,0.7,2*eps,100)

res = 0.7797
k = 12

eqfase(res)

ans = 2.8422e-014
```

Questo è lo stesso valore che restituisce la funzione 'fzero' di Matlab. Proviamo quindi ad applicare tutti i metodi:

```
for i = 1:4
[res(1,i) k(1,i)] = bisezione(@eqfase,0.001,100,1e-8,i);
[res(2,i) k(2,i)] = newton(@eqfase,@derfase,0.5,10e-8,i);
[res(3,i) k(3,i)] = corde(@eqfase,@derfase,0.5,0,10e-8,i);
[res(4,i) k(4,i)] = corde(@eqfase,@derfase,0.5,2,10e-8,i);
[res(5,i) k(5,i)] = secanti(@eqfase,0.001,100,10e-8,i);
end

err = (abs(res-soluzione));
```

Ci siamo fermati così presto perché sappiamo già che il metodo più veloce converge in quattro iterazioni. La seguente tabella mostra l'errore assoluto:

| Metodo | k=1 | k=2 | k=3 | k=4 |
|---|---|---|---|---|
| Bisezione | 1.1721e+001 | 5.4713e+000 | 2.3463e+000 | 7.8382e-001 |
| Newton | 3.9816e-004 | 1.5318e-005 | 5.8934e-007 | 2.2674e-008 |
| Corde n=0 | 3.8930e-004 | 1.4634e-005 | 5.5013e-007 | 2.0681e-008 |
| Corde n=2 | 3.8930e-004 | 1.4977e-005 | 5.7621e-007 | 2.2169e-008 |
| Secanti | 1.5283e-005 | 5.1715e-011 | 2.2204e-016 | 2.2204e-016 |

Il metodo di Bisezione e delle Secanti, data l'ampiezza dell'intervallo, è stato un po' penalizzato; ci si rende conto però che i metodi di Newton e delle Secanti lavorano meglio rispetto a quello delle Corde e di Bisezione. Per quanto riguarda questa funzione vince di gran lunga il metodo delle secanti: converge più velocemente e con maggiore precisione. Proviamo a fare lo stesso con l'equazione per il modulo:

| Metodo | k=1 | k=2 | k=3 | k=4 |
|---|---|---|---|---|
| Bisezione | 0.0646 | 0.4979 | 0.2167 | 0.0760 |
| Newton | 3.0786e-005 | 2.6647e-010 | 0 | 0 |
| Corde n=0 | 1.0846e-003 | 1.1509e-004 | 1.2240e-005 | 1.3021e-006 |
| Corde n=2 | 1.0846e-003 | 3.3081e-007 | 2.0172e-010 | 8.8818e-016 |
| Secanti | 2.9917 | 1.9184 | 1.6112 | 0.6962 |

In questo caso il migliore è invece il metodo di Newton mentre il metodo delle secanti è quello che si comporta peggio.

## 8. Algoritmi per trovare le radici di un polinomio

Per trovare le radici di un polinomio $p_n(x) = 0$:

$$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = 0$$

innanzitutto lo rendiamo monico dividendo per $a_n$:

$$\frac{1}{a_n} p_n(x) = x^n + b_{n-1}x^{n-1} + \cdots + b_1 x + b_0 = 0$$

con $b_i = -\frac{a_i}{a_n}$. A questo possiamo associare una matrice detta 'matrice compagna del polinomio':

$$C_n = \begin{pmatrix} b_{n-1} & b_{n-2} & \dots & b_1 & b_0 \\ 1 & 0 & & & \\ & 1 & \ddots & & 0 \\ & & \ddots & \ddots & \\ & 0 & & 1 & 0 \end{pmatrix}$$

Calcolando il polinomio caratteristico di questa matrice:

$$\det(\lambda I - C_n) = k p_n(\lambda)$$

si ottiene a meno di una costante moltiplicativa il polinomio di partenza. Notando che la matrice è in forma di Hessenberg possiamo usare tutti i metodi per il calcolo degli autovalori al fine di trovare le radici del polinomio:

```
function res = radici(polinomio)

n = length(polinomio)-1;
C = [-polinomio(2:end)/polinomio(1);eye(n-1),zeros(n-1,1)]
res = eig(C);
```

```
return
```

Consideriamo il polinomio ciclotomico $\phi_5(z) = z^4 + z^3 + z^2 + z + 1$. L'algoritmo restituisce le radici $0.3090 \pm 0.9511i$ e $-0.8090 \pm 0.5878i$:

```
pol_cycl5 = [1 1 1 1 1];
zeri = radici(pol_cycl);
err = abs(zeri.^5-1)

err = 1.0e-014 * [0.3898 0.3898 0.0555 0.0555]
```

Oppure rispetto alle soluzioni analitiche:

$$\left\{ 1; \frac{u\sqrt{5}-1}{4} + v\sqrt{\frac{5+u\sqrt{5}}{8}}i : u, v \in [-1, 1] \right\}$$

l'errore commesso risulta $\{0.7772\ 0.1110\}10^{-15}$.

Consideriamo ora un caso con una radice reale per poter fare un confronto anche coi metodi per la risoluzioni di equazioni non lineari generiche:

$$p(x) = (x^2+1)(x^2+2)(x-3) = x^5 - 3x^4 + 3x^3 - 9x^2 + 2x - 6$$

```
poli=[1 -3 3 -9 2 -6]
zeri=radici(poli);
res(1)=zeri(1);

[res(2) k(2)] = bisezione(@polinomio,sqrt(2),3*pi,1e-8,1000);
[res(3) k(3)] = newton(@polinomio,@derpolinomio,0,10e-8,1000);
[res(4) k(4)] = corde(@polinomio,@derpolinomio,0,0,10e-8,1000);
[res(5) k(5)] = corde(@polinomio,@derpolinomio,0,2,10e-8,1000);
[res(6) k(6)] = secanti(@polinomio,sqrt(2),3*pi,10e-8,1000);

err = (res-3)/3

res = 3.0000    3.0000    3.0000    3.0000    3.0000    3.0000
k = 0    30   238   240   238    47
err = 1.0e-006 * [-0.0000   -0.0006   -0.1764   -0.1627   -0.1758   -0.0000]

err(1) = -1.1842e-015

err(6) = -1.9866e-013
```

Come intervalli sono stati usati dei numeri irrazionali in maniera da evitare particolari simmetrie. Il metodo con la matrice compagna risulta essere più preciso anche se come controparte è necessario calcolare gli autovalori di una matrice $5 \times 5$. Dopo questo, il metodo che funziona meglio è quello delle secanti. Bisogna comunque tenere in considerazione che probabilmente per gli altri algoritmi la scelta di un solo punto iniziale penalizza molto: infatti tendono a convergere in molti più passi.

Infine analizziamo il caso di un problema molto malcondizionato:

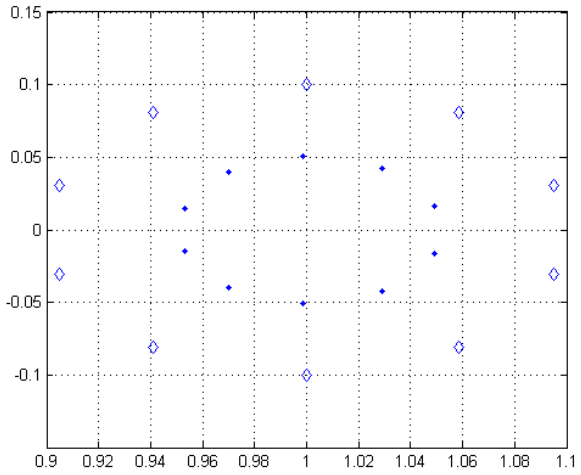
$p(x) = x^{10} - 10x^9 + 45x^8 - 120x^7 + 210x^6 - 252x^5 + 210x^4 - 120x^3 + 45x^2 - 10^x + 1$

Questo ha 10 radici coincidenti in 1. Il risultato è il seguente:

```
zeri =

   1.0490 +/- 0.0163i
   1.0293 +/- 0.0420i
   0.9988 +/- 0.0505i
   0.9700 +/- 0.0397i
   0.9530 +/- 0.0149i
```

Gli autovalori ruotano attorno alla soluzione. Questo è un problema tipico: essendo il determinante piccolo gli autovettori risultano vicini e la propagazione degli errori diventa disastrosa. Se ad esempio modifichiamo l'1 finale sommandoci $10^{-10}$ le soluzioni si spostano come si può vedere in figura (rombi):




l'errore aumenta di 0.05 circa: mezzo miliardo di volte l'errore introdotto sul termine di grado minore!

## 9. Sistemi di equazioni non lineari

Consideriamo il sistema quadrato:

$$\begin{cases} f_1(x_1, x_2, \dots, x_n) = 0 \\ f_2(x_1, x_2, \dots, x_n) = 0 \\ \vdots \\ f_n(x_1, x_2, \dots, x_n) = 0 \end{cases}$$

e supponiamo di avere una vaga idea di dove si trovi la soluzione. Il sistema può essere compattato come $F(x) = 0$ ponendo:

$$F(X) = \begin{bmatrix} f_1(x_1, x_2, \dots, x_n) \\ f_2(x_1, x_2, \dots, x_n) \\ \vdots \\ f_n(x_1, x_2, \dots, x_n) \end{bmatrix}$$

con $F : \Re^n \to \Re^n$ e $x \in \Re^n$.
Il metodo di bisezione non può funzionare, quindi proviamo il metodo di Newton. Facciamo uno sviluppo di Taylor troncando al primo ordine:

$$F(x)(x_0) + F'(x_0)(x - x_0)$$

dove $F'(x)$ è una matrice di funzioni composta dalle derivate parziali (jacobiano) e $x_0$ è il vettore di partenza. Da questa è possibile risalire alla funzione iterativa:

$$x_{k+1} = x_k - [F'(x_k)]^{-1} F(x_k)$$

Nella pratica non si inverte la matrice ma si risolve il sistema:

$$F'(x_k) h_k = -F(x_k)$$

con $h_k = x_{k+1} - x_k$ e $x_{k+1} = x_k + h_k$. Una possibile implementazione dell'algoritmo è la seguente:

```
function [x k] = funcNewton_sys(x0,fun,jac,t,Nmax)
fun = fcnchk(fun); jac = fcnchk(jac);
F = fun(x0);
if(any(F)==0) % verifica che x non sia già soluzione
    k = 0; x = x0; return;
end
k = 1; % prima iterazione
J = jac(x0);
xold = x0; x = x0-J\F;
while norm(x-xold)>t*norm(x) && Nmax>k % iterazioni successive
    F = fun(x); J = jac(x);
    xold = x; x = xold-J\F;
    k = k+1;
end
```

In generale ad ogni passo abbiamo da calcolare il valore di $n(n+1)$ funzioni ($n^2$ per lo jacobiano e n per la funzione) e risolvere un sistema lineare di dimensione n. Il metodo delle corde in questo caso, a differenza di quello monodimensionale, è molto utile; invece di calcolare lo jacobiano tutte le volte, lo calcolo solo una volta ogni tanto:

```
function [x k] = funcCorde_sys(x0,fun,jac,n,t,Nmax)
fun = fcnchk(fun); jac = fcnchk(jac);
F = fun(x0);
if(any(F)==0)
    k = 0; x = x0; return;
end
k = 1; % prima iterazione
C = jac(x0);
[L U] = lu(C);
xold = x0; y = -L\F; x= xold+U\y;
while (norm(x-xold)>=t*norm(x)) && (k<=Nmax) % iterazioni successive
    F = fun(x);
    if(n~=0) % aggiorna lo jacobiano ogni n iterazioni
        if(mod(k-1,n)==0)
            C = jac(x);
            [L U] = lu(C);
        end
    end
    xold = x;
    y = -L\F;
    x= xold+U\y;
    k = k + 1;
end
```

Solo ogni n passi si calcola lo jacobiano e si fa la fattorizzazione LU. Quest'ultima è molto conveniente perché la matrice del sistema rimane la stessa finché non si aggiorna lo jacobiano.

Newton Jacobi è un metodo che riduce la complessità computazionale. Al posto di usare tutto lo jacobiano, ne estraiamo la diagonale e a questo punto fare l'inversa è immediato:

$$F(x) = D(x) - L(x) - U(x)$$

dove $D$ è la diagonale di F, -L è la subdiagonale e -U la sopradiagonale.

$$C(x) = D'(x)$$

$$x_{k+1} = x_k - [C'(x_k)]^{-1} F(x_k)$$

```
function [x k] = funcNewtonJacobi_sys(x0,fun,jac,t,Nmax)
fun = fcnchk(fun); jac = fcnchk(jac);
F = fun(x0);
if(any(F)==0)
    k = 0; x = x0; return;
end
k = 1; % prima iterazione
C = diag(1./diag(jac(x0))); % inversa della diagonale dello jacobiano
xold = x0;
x = xold-C*F;
while norm(x-xold)>t*norm(x) && Nmax>k % iterazioni successive
    F = fun(x);
    C = diag(1./diag(jac(x)));
    xold = x;
    x= xold-C*F;
    k=k+1;
end
```

Gauss-Seidel funziona come il metodo di Newton-Jacobi ma al posto di considerare la diagonale dello jacobiano ne estrae il triangolo inferiore. Anche in questo caso l'inversione della matrice è un'operazione molto più semplice:

$$F(x) = D(x) - L(x) - U(x)$$

$$C(x) = (D(x) - L(x))'$$

$$x_{k+1} = x_k - [C'(x_k)]^{-1} F(x_k)$$

```
function [x k] = funcGaussSeidel_sys(x0,fun,jac,t,Nmax)
fun = fcnchk(fun); jac = fcnchk(jac);
F = fun(x0);
if(any(F)==0)
    k = 0; x = x0; return;
end
k=1; % prima iterazione
C = tril(jac(x0)); % triangolo inferiore dello jacobiano
xold = x0; x = xold-C\F;
while norm(x-xold)>t*norm(x) && Nmax>k % iterazioni successive
    F = fun(x);
    C = tril(jac(x));
    xold = x; x= xold-C\F;
    k=k+1;
end
```

Il metodo di Broyden è una generalizzazione a più dimensioni del metodo delle secanti. Lo jacobiano viene determinato attraverso l'equazione delle secanti (usando l'approssimazione alle differenze finite):

$$J_n(x_n - x_{n-1}) \cong F(x_n) - F(x_{n-1})$$

In più di una dimensione questo sistema è sottodeterminato. Il metodo di Broyden, tra le infinite soluzioni, prende quella che minimizza la norma di Frobenius $||J_n - J_{n-1}||_{\text{frob}}$. Per fare ciò, ad ogni passo aggiorna lo jacobiano secondo la formula:

$$J_n = J_{n-1} + \frac{\Delta F_n - J_{n-1}\Delta x_n}{||\Delta x_n||^2}\Delta x_n^T$$

e poi prosegue nella direzione di Newton per calcolare $x_{n+1}$. Siccome per iniziare occorrono due punti, il primo lo prendiamo tramite il metodo di Newton:

```
function [x k] = funcBroyden_sys(x0,fun,jac,t,Nmax)
fun = fcnchk(fun); jac = fcnchk(jac);
F = fun(x0);
if(any(F)==0)
    k = 0; x = x0; return;
end
k=1; % prima iterazione (come newton)
C = jac(x0);
xold = x0; x = xold-C\F;
while norm(x-xold)>t*norm(x) && Nmax>k % iterazioni successive
    Fold = F; F = fun(x);
    S = x-xold; Y = F-Fold; Cold = C;
    C = Cold + ((Y-Cold*S)/(S'*S))*S';
    xold=x; x= xold-C\F;
    k=k+1;
end
```

Vediamo ora di applicare tali algoritmi a qualche esempio. Consideriamo il sistema:

$$\begin{cases} f_1(x_1, x_2) = e^{x_1} + x_1x_2 - 1 = 0 \\ f_2(x_1, x_2) = \sin(x_1x_2) + x_1 + x_2 - 1 \end{cases}$$

che ha una sola soluzione $x = [0\ 1]$. Come punto iniziale scegliamo $x = [1\ 5]$, come precisione $t = 10^{-8}$ e per il metodo delle corde $n = 0$. In tabella è mostrato l'errore e il numero di iterazioni

| Metodo | x(1) | x(2) | errore | k |
|---|---|---|---|---|
| Newton | 1.0222e-016 | 1-1.1102e-016 | -1.1102e-016 | 7 |
| Corde | 5.6396e-009 | 1-1.1830e-008 | -1.1830e-008 | 46 |
| Newton-Jacobi | 4.9418e-017 | 1-1.1102e-016 | -1.1102e-016 | 6 |
| Gauss-Seidel | -6.5910e-018 | 1 | 0 | 5 |
| Broyden | -2.2481e-012 | 1+4.5666e-012 | 4.5666e-012 | 15 |

Il metodo di Gauss-Seidel si rivela il più performante assieme a quello di Jacobi che a spese di un'iterazione in più ci consente di valutare 3 elementi dello jacobiano ad ogni passo invece che 6. Anche Newton funziona molto bene, mentre il metodo delle corde e di Broyden richiedono molte più iterazioni. Questo è probabilmente dovuto al fatto che vengono usate meno informazioni sullo jacobiano.

I problemi diventano molto più seri quando sono presenti più di due equazioni e più di una soluzione. Consideriamo ad esempio il sistema $4 \times 4$:

$$\begin{cases} f_1(x_1, x_2, x_3, x_4) = x_1^2 + 2x_2^2 + \cos(x_3) - x_4^2 = 0 \\ f_1(x_1, x_2, x_3, x_4) = 3x_1^2 + x_2^2 + \sin^2(x_3) - x_4^2 = 0 \\ f_1(x_1, x_2, x_3, x_4) = -2x_1^2 - x_2^2 - \cos(x_3) + x_4^2 = 0 \\ f_1(x_1, x_2, x_3, x_4) = -x_1^2 - x_2^2 - \cos^2(x_3) + x_4^2 - 1 = 0 \end{cases}$$

Che dovrebbe avere otto soluzioni $x_1 = [\pm 1 \ \pm 1 \ 0 \ \pm 2]$ se limitiamo le componenti tra -1 e 1. Spesso le soluzioni a cui gli algoritmi convergono vanno all'infinito oppure lo jacobiano (o chi per lui) tende ad essere singolare. Inoltre c'è il problema di trovarle tutte. Vediamo un po cosa succede con i vari metodi assegnando come punto iniziale un punto random:

```
t = 1e-8;
Nmax = 10000;
x0 = (rand(4,1)-0.5)*2;
```

Il metodo di Newton converge in 25,30 iterazioni mentre quello di Broyden in 40,50. Questi funzionano bene mentre Newton-Jacobi e Gauss-Seidel divergono sempre probabilmente per problemi di raggio spettrale. Viene di seguito illustrato qualche esempio di soluzione con l'applicazione del metodo di Newton:

| | | | |
|---|---|---|---|
| x = | x = | x = | x = |
| -1.000000000000000 | 1.000000000000000 | 1.000000000000000 | -1.000000000000000 |
| 1.000000000000000 | 1.000000000000000 | -1.000000000000000 | -1.000000000000000 |
| -0.000000013663823 | 0.000000019542294 | 0.000000017401714 | 0.000000020850833 |
| 2.000000000000000 | 2.000000000000000 | -2.000000000000000 | 2.000000000000000 |
| k = 26 | k = 25 | k = 25 | k = 26 |
| x = | x = | x = | x = |
| -1.000000000000000 | -1.000000000000000 | 1.000000000000000 | -1.000000000000000 |
| -1.000000000000000 | 1.000000000000000 | 1.000000000000000 | 1.000000000000000 |
| -0.000000017288264 | -0.000000011217008 | -0.000000012667746 | -0.000000013077558 |
| 2.000000000000000 | -2.000000000000000 | 2.000000000000000 | -2.000000000000000 |
| k = 26 | k = 26 | k = 26 | k = 26 |
| x = | x = | x = | x = |
| 1.000000000000000 | 1.000000000000000 | 1.000000000000000 | -1.000000000000000 |
| 1.000000000000000 | -1.000000000000000 | 1.000000000000000 | -1.000000000000000 |
| -0.000000021854155 | -0.000000013329068 | 0.000000017523266 | 0.000000020883700 |
| 2.000000000000000 | 2.000000000000000 | -2.000000000000000 | -2.000000000000000 |
| k = 24 | k = 26 | k = 26 | k = 24 |

Proviamo infine un sistema ancora più complesso che riguarda un problema di neurofisiologia. Il sistema non lineare che ne risulta è sparso:

$$\begin{cases} x_1^2 + x_3^2 - 1 = 0 \\ x_2^2 + x_4^2 - 1 = 0 \\ x_5 x_3^3 + x_6 x_4^3 = 0 \\ x_5 x_1^3 + x_6 x_2^3 = 0 \\ x_5 x_1 x_3^3 + x_6 x_4^2 x_2 = 0 \\ x_5 x_1^2 x_3 + x_6 x_2^2 x_4 = 0 \end{cases}$$

Come è facile aspettarsi, dall'analisi del sistema saranno predilette le soluzioni che implicano $x_5 = 0$ e $x_6 = 0$. Questo accade prevalentemente utilizzando il metodo di Newton. Col metodo di Broyden si riescono invece ad ottenere soluzioni diverse di cui si da qualche esempio:

| | X | F(X) |
|---|---|---|
| Soluzione 1 | -0.669502514428232 | -0.000017450989865 |
| | 0.571878809768603 | 0.000076238704981 |
| | 0.742798042663287 | -0.139642616115590 |
| | 0.820384583986454 | 0.152132260743036 |
| | -0.454374632382348 | 0.157143215727250 |
| | 0.084357295020868 | -0.128649381081078 |

| | X | F(X) |
|---|---|---|
| Soluzione 4 | 0.494842398835997 | -0.000242833997421 |
| | 0.713135251286559 | -0.000212322757018 |
| | -0.868843004412659 | -0.068346496845497 |
| | 0.700875017828028 | 0.201879087324627 |
| | 0.337253459411961 | 0.046066919748888 |
| | 0.443962955479702 | 0.086493820661914 |

| | X | F(X) |
|---|---|---|
| Soluzione 2 | -0.615256334059412 | -0.000003438232884 |
| | 0.489100982943492 | -0.000002246385918 |
| | -0.788324936283821 | 0.057527383481803 |
| | 0.872225877911101 | -0.007052228980999 |
| | 0.117362567990441 | 0.099875391863201 |
| | 0.173341726281330 | 0.001145895435999 |

| | X | F(X) |
|---|---|---|
| Soluzione 5 | -0.330753511638055 | -0.000000023739341 |
| | 0.330753502272579 | -0.000000023755858 |
| | 0.943717166739990 | -0.000000000739694 |
| | -0.943717170013646 | 0.000000000525469 |
| | -0.143069276967637 | -0.002371975589901 |
| | -0.143069274598671 | -0.000000001029810 |

| | X | F(X) |
|---|---|---|
| Soluzione 3 | -0.168468370723273 | 0.000293279119025 |
| | 0.719065657955925 | 0.001492700233521 |
| | 0.985855814602151 | 0.030941595465934 |
| | 0.696015287031782 | 0.014028975639259 |
| | 0.018928783964695 | 0.010173269409937 |
| | 0.037976340531182 | 0.014196497943126 |

dove sono state selezionate le soluzioni con componenti in valore assoluto minori di uno e non più piccole di $10^{-2}$. I metodi di Newton-Jacobi e Gauss-Seidel anche in questo caso non riescono a convergere.

## Riferimenti bibliografici

[1] Giuseppe Rodriguez, *Algoritmi Numerici*, Pitagora Editrice Bologna.

[2] Crina Grosan, Ajith Abraham: *A New Approach for Solving Nonlinear Equations Systems*, IEEE transactions on systems, man, and cybernetics part A: systems and humans, vol. 38, no. 3, may 2008.