Laboratorio di Analisi Numerica

Federica Piras matr. 29486
federica_piras@virgilio.it

Stefano Angioni matr. 30153
stefano_angioni@tin.it

Introduzione

Questo lavoro rappresenta la conclusione del seminario su *"Modelli e metodi di supporto alle decisioni"*, svolto nei mesi di febbraio-marzo presso la facoltà di Ingegneria dell'università di Cagliari.
Avendo già seguito il corso di Metodi della Ricerca Operativa, la nostra attenzione nello svolgere questo lavoro si è spostata sulla parte di analisi numerica svolta in aula da prof. Rodriguez durante l'ultima lezione. Con l'ausilio di Matlab abbiamo cercato di studiare in quale modo si possano propagare gli errori nella risoluzione di problemi diffusissimi, quali i sistemi lineari di equazioni. Infatti, al crescere della dimensione dei problemi, il solo fatto di avere necessità di memorizzare i dati su un numero finito di bit, comporta errori di arrotondamento o troncamento che possono diventare tanto importanti da rendere, nella pratica, le soluzioni ottenute inutilizzabili. Senza considerare il fatto che spesso oltre agli errori "di macchina" ve ne sono altri, che spesso sono trascurabili, ma in particolari situazioni assumono proporzioni tali da dovere studiare metodi particolari per la loro eliminazione; tra questi ricordiamo gli errori di approssimazione del modello fisico, o l'incertezza intrinseca nei dati sperimentali.

Consideriamo ora un sistema lineare di n equazioni in n incognite, ovvero quello che si suole dire un sistema quadrato, nella sua forma generale:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases} \quad \text{o, in notazione matriciale, } \mathbf{Ax} = \mathbf{b}$$

Ricercarne la soluzione significa determinare una n-upla $\left(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\right)$ tale che $\mathbf{A}\overline{\mathbf{x}} = \mathbf{b}$, cioè n numeri che sostituiti all'interno delle equazioni le verifichino tutte e contemporaneamente. Questo nella teoria. In realtà, essendo impossibile per un essere umano risolvere sistemi di dimensione grande (ma non troppo, anche $n = 6$ è una bella sfida!) in tempi ragionevoli, ci si appoggia a dei software particolari che sfruttano la potenza di calcolo dei moderni elaboratori elettronici. Tuttavia, il prezzo da pagare per la velocità di calcolo è il fatto che un computer non esegue i calcoli esattamente, e di conseguenza introduce un errore che è utile sapere stimare, e questa stima è proprio lo scopo di questo nostro lavoro.

Metodi di lavoro

Abbiamo scritto alcuni semplici programmi nel linguaggio Matlab, che ci mostreranno come al crescere delle dimensioni, al variare dei parametri, e dello stesso metodo di risoluzione dei sistemi, possono presentarsi situazioni abbastanza varie.

In sostanza il metodo utilizzato è questo:

- Creare una matrice di coefficienti, nella fattispecie abbiamo utilizzato alcune matrici predefinite nel software;
- Determinare un termine noto moltiplicando la matrice per una *n*-upla determinata;
- Risolvere il sistema lineare con questo termine noto.
- Valutare lo scostamento

Ciò che sembra un procedimento alquanto stupido in realtà permette di valutare quanto la soluzione numerica determinata da Matlab si discosti dalla soluzione vera, nota a priori. Il metro di misura di tale scostamento sarà la norma del vettore-differenza, ovvero, detto $\tilde{\mathbf{x}}$ la soluzione di Matlab, e **x** la soluzione esatta, si avrà:

$$\delta x = \left\| \tilde{\mathbf{x}} - \mathbf{x} \right\|.$$

I metodi di risoluzione saranno:

- Divisione a sinistra, ovvero quello standard per chi utilizza Matlab; il comando è `x = A\b`
- Moltiplicazione per l'inversa: `x = inv(A) * b`

Entrambi questi metodi sfruttano le proprietà della fattorizzazione **PA** = **LU** che permette (a meno della matrice **P** detta di permutazione) di esprimere una qualsiasi matrice quadrata **A** che sia non singolare, sotto forma di prodotto di altre due matrici quadrate **L** ed **U** che sono rispettivamente triangolari inferiore (Lower) e superiore (Upper). Tale fattorizzazione consente, ad esempio, di evitare l'onere computazionale del calcolo dei determinanti per l'applicazione del metodo di Cramer, e riconduce la soluzione di un sistema quadrato nella risoluzione in cascata di due sistemi triangolari, infatti, essendo **A=LU**:

$$\mathbf{Ax} = \mathbf{b} \Rightarrow \begin{cases} \mathbf{Ly} = \mathbf{b} \\ \mathbf{Ux} = \mathbf{y} \end{cases}$$

E si vede che il primo sistema (triangolare inferiore) fornisce il termine noto per il secondo (triangolare superiore). Essendo poi la complessità computazionale di ciascuno dell'ordine di $O(1/2n^2)$, è comunque conveniente in termini di calcolo, rispetto alla crescita fattoriale del metodo di Cramer, la cui unica applicazione pratica è forse quella di essere un potente strumento teorico per le dimostrazioni..

Ora, analoghe considerazioni consentono di dimostrare che anche il calcolo della matrice inversa di **A** può essere effettuato rapidamente mediante tale fattorizzazione, al pari del suo determinante, e naturalmente un software come Matlab, che non si nega niente, ha nelle sue librerie l'implementazione degli algoritmi che operano tale fattorizzazione; in particolare, utilizza il cosiddetto *metodo di Gauss con pivoting totale*, che con una leggera variante rispetto al metodo standard (in parole molto povere si aggiunge una ricerca di massimo all'interno della matrice e poi si opera uno scambio di righe e colonne) consente di estendere l'applicabilità del metodo di Gauss a ogni matrice invertibile, e in più può ridurre la propagazione degli errori.

Accanto a tali algoritmi evoluti (nel senso che sono stati implementati da gente esperta) abbiamo testato il comportamento di una versione più "casalinga", ovvero realizzata da noi. Questa è composta di un'implementazione del l'algoritmo di Gauss **senz** pivoting per la triangolarizzazione del sistema, e da un algoritmo di backward substitution (sostituzione a ritroso) per la sua risoluzione. Niente di particolarmente complicato, quindi, eppure i risultati sono abbastanza sorprendenti… Buona lettura

Le matrici

Rappresentano le "cavie" del nostro esperimento. Abbiamo scelto la matrice di Hilbert, quella di Pascal e la Random, ma Matlab ne mette a disposizione molte altre.

Matrice di Hilbert

La matrice di Hilbert è una particolare matrice che ha sulle anti-diagonali i reciproci dei numeri interi, ossia. A esempio se diamo a Matlab l'istruzione `A = hilb(6)`, questa sarà l'uscita:

```
A =

    1.0000    0.5000    0.3333    0.2500    0.2000    0.1667
    0.5000    0.3333    0.2500    0.2000    0.1667    0.1429
    0.3333    0.2500    0.2000    0.1667    0.1429    0.1250
    0.2500    0.2000    0.1667    0.1429    0.1250    0.1111
    0.2000    0.1667    0.1429    0.1250    0.1111    0.1000
    0.1667    0.1429    0.1250    0.1111    0.1000    0.0909
```

Questa matrice è fortemente mal condizionata, ossia amplifica molto gli errori al crescere della dimensione del sistema.

Matrice di Pascal

La seconda matrice è quella detta di Pascal, una particolare matrice che contiene al suo interno un triangolo di Tartaglia, ossia i coefficienti binomiali. Ad esempio `A= pascal(6)` produrrà come effetto:

```
A =

     1     1     1     1     1     1
     1     2     3     4     5     6
     1     3     6    10    15    21
     1     4    10    20    35    56
     1     5    15    35    70   126
     1     6    21    56   126   252
```

Essendo una matrice di interi, non crea particolari problemi quando il sistema ha termine noto anch'esso intero, ma si vede che le cifre crescono rapidamente, e questo può dar luogo a problemi di overflow, ovvero i numeri da memorizzare nel calcolatore possono aver bisogno di più cifre (bit) di quante il calcolatore stesso disponga.

Matrice Random

La terza matrice presa in esame è la Random, ovvero una matrice di numeri casuali nell'intervallo tra 0 e 1 generati dal software con probabilità uniforme. Come esempio `A = rand(6)` ci dà

```
A =

    0.1043    0.9121    0.3993    0.0923    0.7971    0.5626
    0.2142    0.0779    0.3010    0.8884    0.3794    0.1572
    0.3424    0.0632    0.6101    0.4545    0.1366    0.7966
    0.4801    0.3181    0.5501    0.7332    0.2654    0.8807
    0.6975    0.4880    0.2504    0.6564    0.2470    0.3657
    0.6085    0.1223    0.6712    0.3348    0.7127    0.4451
```

La matrice aleatoria messa a disposizione da Matlab è solitamente ben condizionata, per cui ci aspettiamo che l'errore propagato sia sempre molto basso.

I grafici

Passiamo ora ad analizzare i risultati dei nostri esperimenti. In ognuno dei grafici dell'errore sono presenti delle righe di diverso colore, ciascuna di esse rappresenta l'andamento dell'errore in corrispondenza di un particolare termine noto. E' stato interessante notare come in alcuni casi l'andamento dell'errore è abbastanza uniforme al variare del termine noto, mentre in altri casi i comportamenti sono decisamente imprevedibili, e questo non soltanto al variare della matrice del sistema (questo era prevedibile), ma nel passare da un algoritmo ad un altro! Questa è stata un'ulteriore conferma che **un algoritmo non vale un altro.**

I primi tre grafici che prendiamo in esame sono quelli riferiti alla matrice di Hilbert:

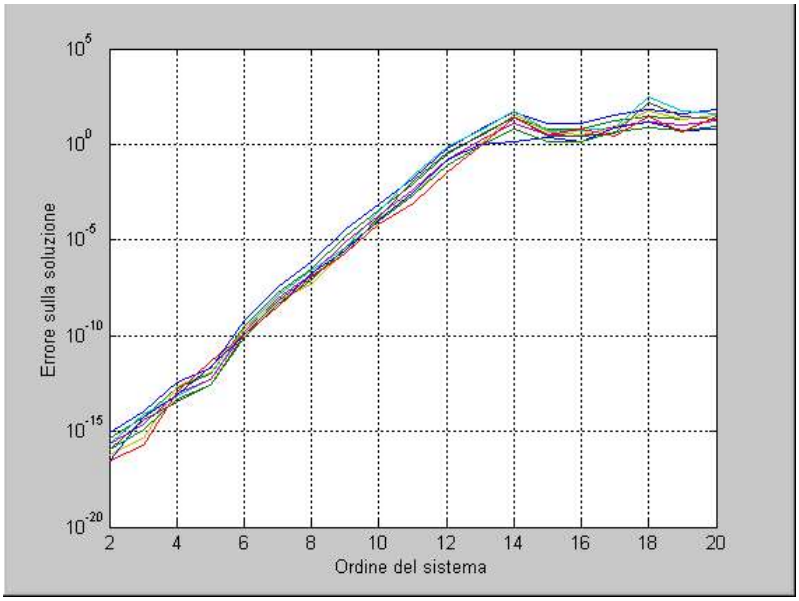

Figura 1. Hilbert con divisione a sinistra

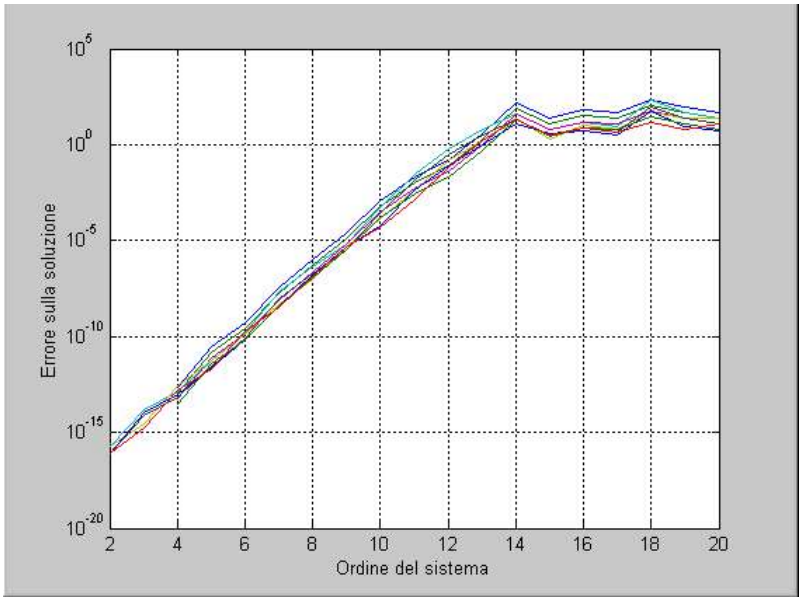

Figura 2. Hilbert con moltiplicazione per l'inversa

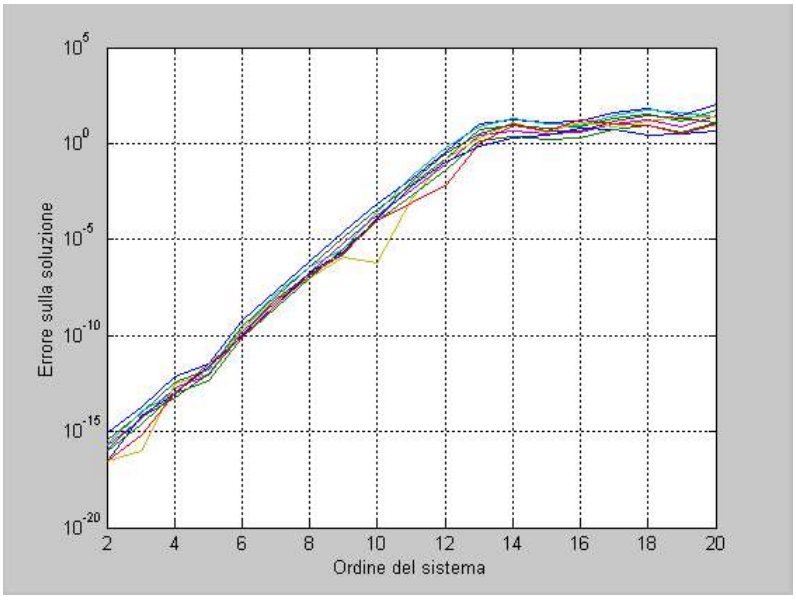

Figura 3. Hilbert con Gauss + backward substitution

Come già avevamo accennato, la matrice di Hilbert è una matrice abbastanza rognosa, in quanto, come si può vedere, l'errore introdotto cresce esponenzialmente con la dimensione del sistema (le ordinate sono logaritmiche!) e già per dimensioni piccole, come quelle prese in esame da noi l'errore introdotto è dell'ordine di 10^3. Naturalmente dire 1000 non significa niente, se non si conoscono gli ordini di grandezza dei dati in gioco… Beh, la matrice contiene numeri tutti minori o al più uguali all'unità, e le soluzioni note che abbiamo preso in esame erano 1, 1/2, 1/3, …, 1/9, e infatti ogni linea di diverso colore nel grafico si riferisce ad una soluzione particolare. La

saturazione che sembra raggiungere la curva è probabilmente dovuta a effetti numerici, teoricamente non può che crescere. Infine, gli andamenti sono abbastanza simili per tutti e tre i metodi di risoluzione, ma il primo (divisione a sinistra) è risultato nettamente il più veloce.

Analizziamo ora come si comporta la matrice di Pascal:

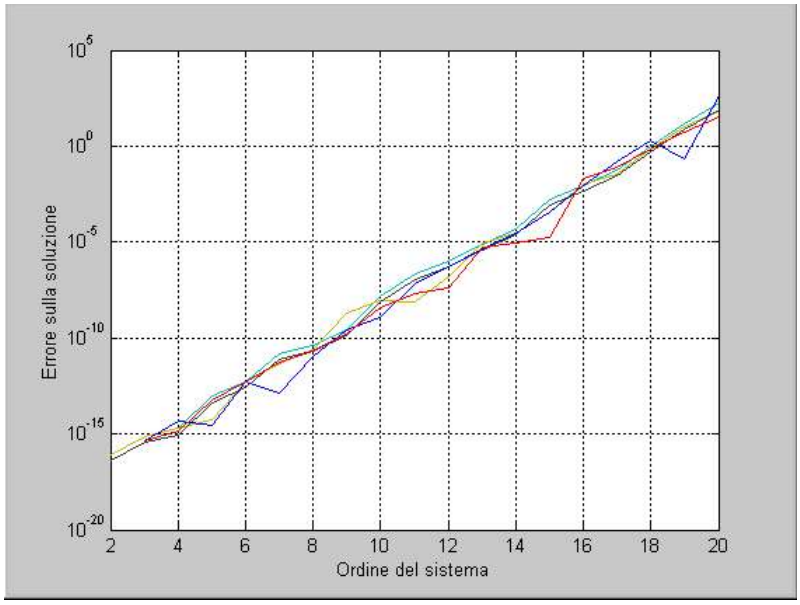

Figura 4. Pascal con divisione a sinistra

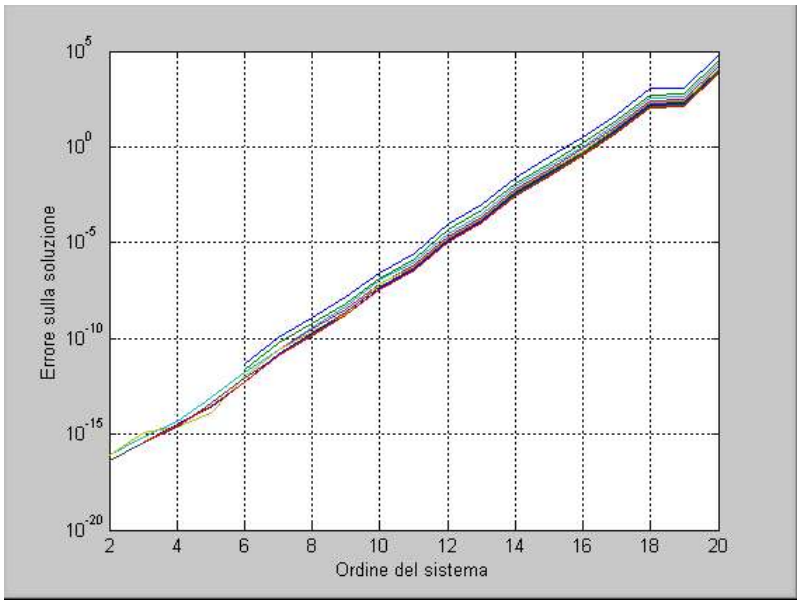

Figura 5. Pascal con moltiplicazione per l'inversa

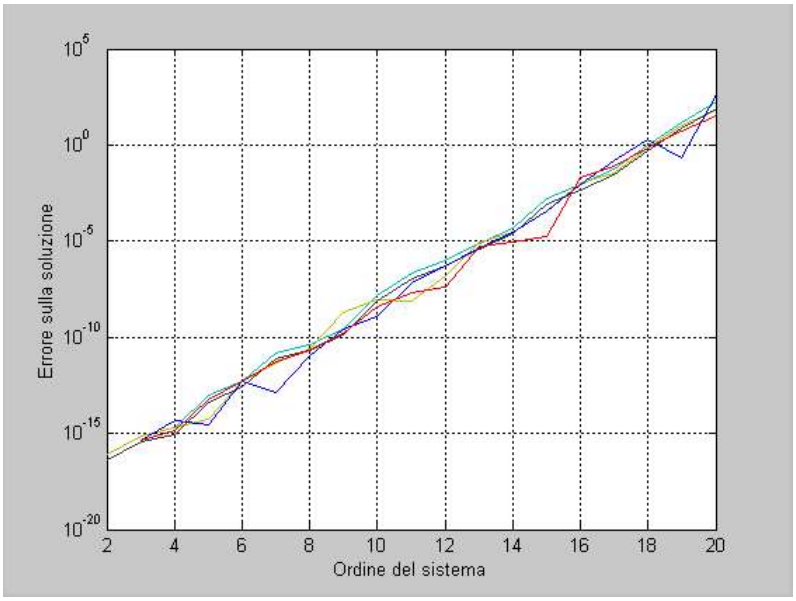

Figura 6. Pascal con Gauss + backward substitution

Se possibile è ancora peggiore della Hilbert… Infatti la crescita è ancora esponenziale, ma non presenta alcuno smorzamento, visto che i coefficienti binomiali non possono che crescere. E' da notare come gli algoritmi che in qualche modo utilizzano il metodo di Gauss, sia implicitamente come quello standard di Matlab, sia esplicitamente come quello scritto da noi, abbiano andamenti dell'errore abbastanza irregolari, mentre quello che fa uso dell'inversa si comporti in modo molto più uniforme. Se potessimo fare uno zoom, e Matlab ce lo permette, vedremmo che in corrispondenza delle soluzione ottenute come potenze negative di 2, ovvero 1, 1/2, 1/4, 1/8, la comparsa di errore è ritardata. Questo è dovuto al fatto che tali numeri possono essere rappresentati senza introdurre errore nella base di numerazione binaria adottata dai calcolatori, ma la crescita dei coefficienti della matrice porta comunque ad errori, dovuti, come accennato, al fenomeno dell'overflow.

Infine i grafici della matrice Random:

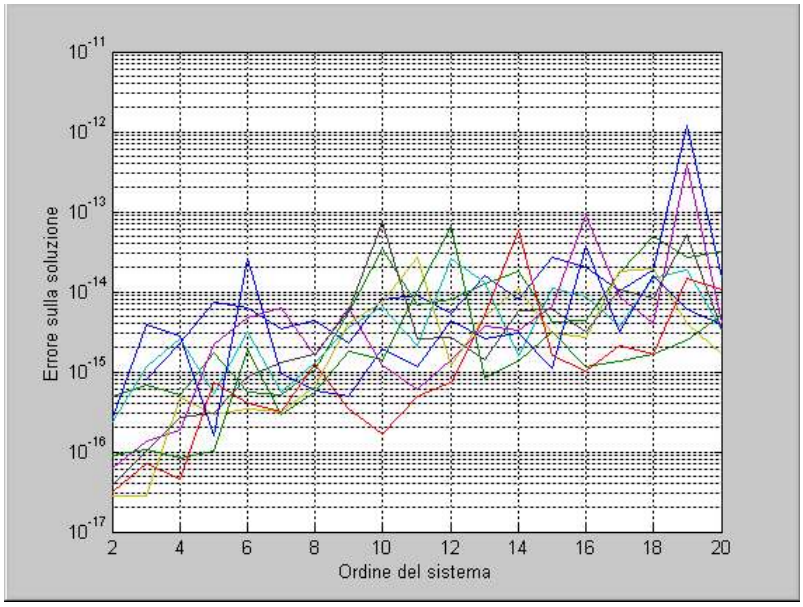

Figura 7. Random con divisione a sinistra

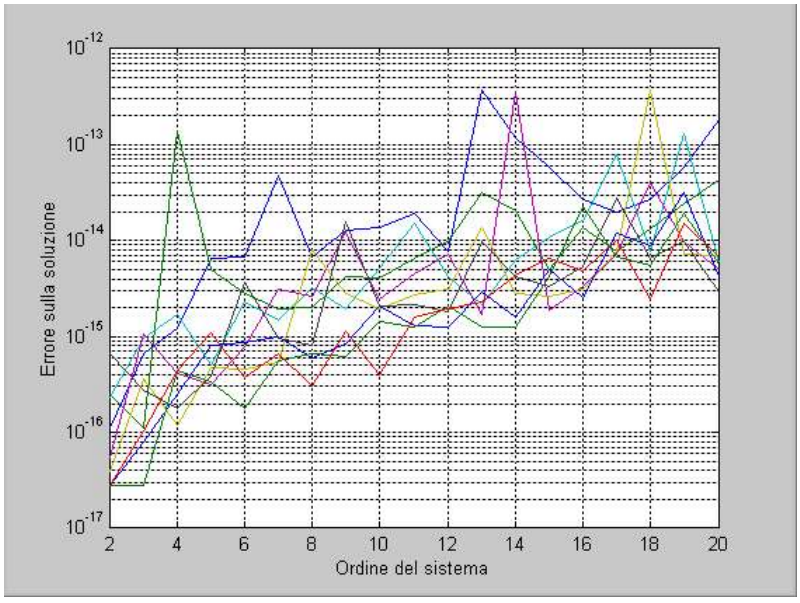

Figura 8. Random con moltiplicazione per l'inversa

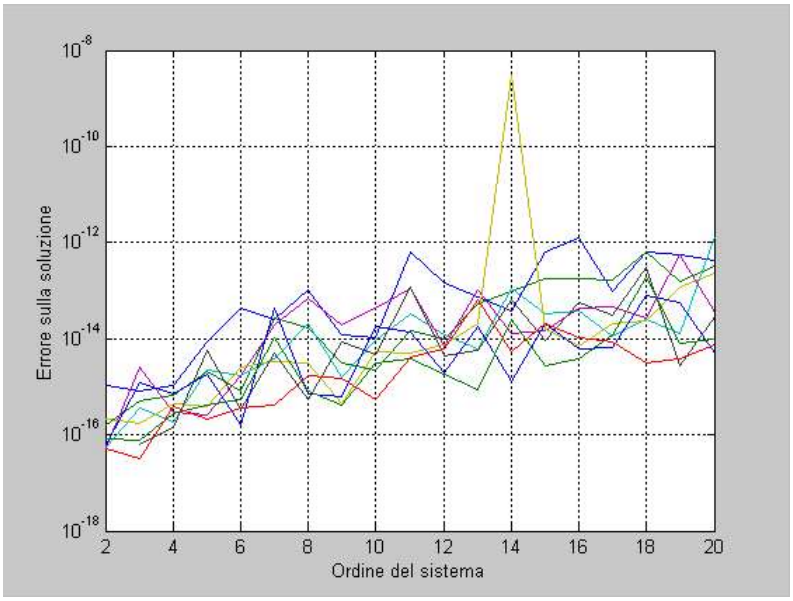

Figura 9. Random con Gauss + backward substitution

Come previsto, la nostra matrice casuale si comporta decisamente bene, mantenendo gli errori decisamente contenuti, mentre gli andamenti risultano alquanto bizzarri, proprio a causa della natura aleatoria degli elementi della matrice. In conclusione, è emerso che particolari matrici possono portare ad errori enormi sulla soluzione di sistemi apparentemente "piccoli", e che comunque l'algoritmo implementato come divisione a sinistra è quello che, in condizioni normali, fornisce le prestazioni migliori, per via della sua ridotta complessità computazionale.

Per concludere, è da notare come, in tutti i casi, è risultata veramente minima la differenza tra la nostra implementazione dell'algoritmo di Gauss e quella built-in di Matlab, almeno per quanto riguarda la propagazione degli errori, e questo può essere dovuto a varie cause, ad esempio l'estrema semplicità dell'algoritmo stesso, che non consente di migliorarlo più di tanto, oppure la particolare forma dei sistemi che abbiamo risolto (che non ha mai introdotto *breakdown* ovvero blocchi dell'algoritmo a causa di una divisione per zero). Tutto un altro discorso va fatto per quanto riguarda la velocità di esecuzione; infatti i primi due metodi sono già compilati e a disposizione di Matlab, per cui non si sente il peso dei *loop* **for** che necessariamente abbiamo dovuto scrivere. Tale differenza si è fatta sentire parecchio anche a causa della… vecchiaia del computer che abbiamo utilizzato per il lavoro… per cui dire (ad esempio) che il tempo di esecuzione è raddoppiato nel passare da un metodo all'altro può dire ben poco se i tempi in gioco sono nell'ordine dei millesimi di secondo, ma quando sono secondi…

Numeri di condizionamento

L'ultimo argomento di cui ci occupiamo sono i numeri di condizionamento. Il n.c. è definito in questa maniera:

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \left\|\mathbf{A}^{-1}\right\|$$

E rappresenta il massimo fattore di amplificazione dell'errore relativo sulla soluzione rispetto a quello sui dati. Esso dipende dalla particolare norma matriciale adottata, ma la sostanza non cambia, un numero di condizionamento alto è proprio di una matrice che amplifica gli errori. L'istruzione sotto Matlab è `x = cond(A,p)`, dove il parametro `p`, opzionale, indica quale particolare norma adottare. Il default è `p = 2`. Osserviamo l'andamento dei numeri di condizionamento per le nostre tre matrici:

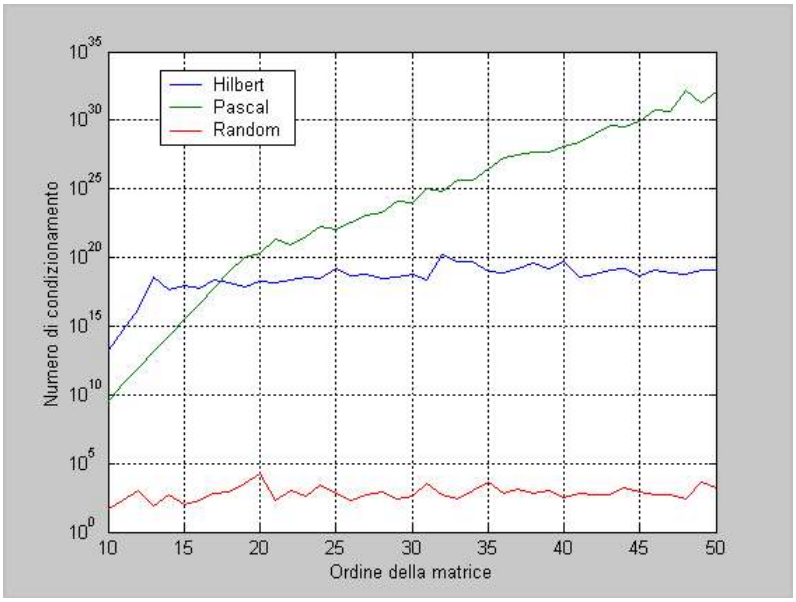

Figura 10. Numeri di condizionamento

Come si può vedere, il comportamento migliore è quello della matrice aleatoria, mentre le altre due divergono immediatamente, sebbene in qualche modo la matrice di Hilbert sia più "prevedibile" nel suo andamento. Questo però non ci consola, in quanto dal grafico emerge che in condizionamento è dell'ordine di 10^{16} !