*Scritti da Gabriel: gli appunti seguono l’ordine di tutte le lezioni.*

*Per eventuali completamenti anche di procedimenti di calcolo, si segua il file riassunto già presente su MEGA, da cui sono estrapolate le immagini presenti (oltre che dalle stesse lezioni del prof o da appunti miei).*

***02/03/2022: Lezione 1 e 2 - Numeri e floating point***

Parliamo dei numeri con segno. In particolare per un numero si intende una parte con segno, una parte intera ed una parte frazionaria, solitamente realizzata con una codifica binaria.

Le cifre e gli indici corrispondono alle potenze, positive e negative.

Noi ci focalizzeremo sulla base *b=10,* nonostante siano rimaste altre basi come la base 60 (sessagesimale).

All’interno del calcolatore si usa invece la base 2, corrispondente ai 2 possibili stati assunti da una macchina.

In merito ad un numero sappiamo essere composto da una parte intera e una serie frazionaria, definita come serie convergente. Ciò è visibile come:

Immagine che contiene testo, orologio

Descrizione generata automaticamente

La serie della parte frazionaria (che sta tra 0 ed 1) è convergente e ciò si dimostra per confronto per serie a termini non negativi, da cui viene maggiorata.

Genericamente tutto si riduce alla serie geometrica di ragione .

In generale la successione delle somme parziali con diverge con a > 1, mentre per a < 1 la serie della parte frazionaria è maggiorata da una serie geometrica convergente e quindi converge.

Per vedere che la parte frazionaria sta effettivamente tra 0 ed 1, ciò viene visto in base 10 si considera come esempio 0.99 periodico in base 10. Usando la serie geometrica e raccogliendo la cifra massima si dimostra che si ottiene proprio 1.

Alcune osservazioni:

* *i numeri irrazionali hanno parte frazionaria infinita in qualsiasi base*, dato che i numeri con parte frazionaria finita in una base sono necessariamente razionali, perché somma della parte intera che è numero naturale e di una combinazione lineare;
* i numeri razionali possono avere una base finita od infinita, per esempio:

= (0,100)3  (caso finito) Ma in base 10 diventa: (0,333…)10 = 1/3 (caso infinito)

Fondamentale nel calcolo numerico è il concetto di *errore* e in particolare il *troncamento* dei numeri, tagliando *n* cifre nella parte frazionaria ma mantenendo tutta la parte intera.

Introduciamo quindi due concetti chiave: *l’approssimazione* e l’errore stesso.

In merito al troncamento abbiamo:

Immagine che contiene testo, orologio

Descrizione generata automaticamente

La quantità errore è il modulo della distanza tra due reali, *a* ed  *̃a*, (quindi una differenza, vista come errore assoluto). Quindi:

Immagine che contiene orologio, calibro

Descrizione generata automaticamente

Vediamo di stimarla ed approssimarla correttamente.

L’errore di troncamento non è altro che il resto della serie geometrica per avere una stima (usando i conti di dell’effettivo errore di troncamento generabile dai calcoli. Si nota quindi che esso non supera , dove il *b* varia a seconda delle cifre utilizzate.

Attenzione: l’errore è stato solo stimato, che è una situazione tipica del calcolo numerico.

Quante cifre devo prendere dopo la virgola per non superare una certa soglia di tolleranza lo determina la risoluzione di una disuguaglianza logaritmica, che porta come risultato

Il fatto di avere una stima permette di controllare l’errore, che dovrà necessariamente essere ≤ ε, che sarebbe la soglia di tolleranza.

La tecnica generalmente più usata è *l’arrotondamento*, che corrisponde alla serie infinita, considerando l’arrotondamento per *difetto* (quindi prima cifra va da 0 a 4 nella base 10) oppure per *eccesso* (da 5 a 9 base 10). Noi ci limitiamo a vedere basi pari. L’esempio è:

Immagine che contiene testo

Descrizione generata automaticamente

Il massimo errore nel caso di basi pari è metà del massimo errore di troncamento ad n cifre, quindi:

Immagine che contiene orologio, antenna, calibro

Descrizione generata automaticamente

Ad esempio nel caso del numero 0,44:

* nel caso del troncamento tutti i numeri dell’intervallo [0.44 e 0.449) tendono a 0.45
* nel caso dell’arrotondamento approssima tutti i numeri dell’intervallo (0.435,0.445) seguendo le regole di prima dell’arrotondamento per eccesso/per difetto.

In generale è meglio l’arrotondamento perché considera un intervallo più ampio e in generale è effettivamente migliore.

***Lezione 2: Floating point***

Un numero viene scritto come numero con segno, mantissa (vista come serie da 1 ad inf delle cifre) moltiplicato per una potenza della base.

L’esponente può essere positivo, nullo o negativo. Si adotta la convenzione che la prima cifra dopo la virgola *d1* sia diversa da 0 in maniera tale da non avere infinite rappresentazioni dello stesso numero.

Questa è la classica scrittura *floating point*, utile ad introdurre il concetto di *precisione macchina* ma anche di definizione dell’insieme dei *reali macchina*.

Per esempio:

diventa oppure:

Semplicemente si sposta la virgola con un opportuno spostamento della base.

La mantissa sta in 0 ed 1 (con 0 escluso), avendo in particolare la mantissa compressa tra 0.1 ad 1 (entrambi inclusi).

In generale un numero reale ha mantissa infinita, perché comprendiamo anche gli irrazionali; i numeri con mantissa finita sono invece i razionali (un esempio può essere 1/3 di prima con base 3 e base 10)

A questo punto si è in grado di definire *l’insieme dei reali-macchina* che, in un sistema di calcolo, lavora un insieme di numeri con una quantità finita di cifre di mantissa (definito come *d*) e un esponente che varia in un intervallo finito di numeri (*p* che dipende dalla sua base *b*). Il fatto che il numero di cifre di mantissa sia finito e l’intervallo di esponenti sia finito permette la memorizzazione utilizzando sequenze di bit in base 2.

A questa lezione segue modello di rappresentazione a 64 bit dei reali-macchina.

Ora vogliamo stimare l’errore in cui l’approssimazione avviene per arrotondamento della mantissa al numero di cifre disponibili. Quindi:

Immagine che contiene testo

Descrizione generata automaticamente

Come detto, l’errore massimo sarà , da cui sappiamo che non supera ,

Subito si nota che l’errore dipende da *p*, cioè dall’ordine di grandezza del numero.

Numeri grandi in modulo avranno errori grandi, numeri piccoli in modulo avranno errori piccoli.

Allontandosi oppure avvicinandosi agli estremi, è accettabile un errore variabile con l’ordine di grandezza del numero? Questo succede se ci spostiamo dall’*errore assoluto*, quindi dati due numeri *a* e  *̃a* in modulo, *all’errore relativo*, definito invece come il rapporto dell’errore assoluto per il modulo di *a*. In generale l’errore relativo è il più importante in campo sperimentale e viene infatti espresso in percentuale.

Quindi:

Immagine che contiene testo

Descrizione generata automaticamente

Per questo motivo andremo ad eseguire una stima da sotto, con

Di fatto viene poi eseguito un calcolo in cui, usando la tecnica di arrotondamento, mettiamo a frazione la stima da sopra e da sotto, si calcola il massimo errore relativo di arrotondamento possibile, cosiddetta precisione di macchina, che sarebbe . Esso dipende solo da *b* e da *t.*

Un esempio del concetto del floating point sono *overflow* ed *underflow*, arrotondando a seconda della situazione a t cifre di mantissa. Nell’intervallo di rappresentazione ogni “tacchetta” nella rappresentazione è un intorno di precisione e, unendo tutti questi intorni, ogni numero se non è numero macchina viene arrotondato per difetto.

***07/03/2022: Lezione 3 - Struttura del sistema floating point e distribuzione***

Trattiamo l’insieme dei reali macchina e la rappresentazione dei numeri, discutendo poi un modello di rappresentazione a 64 bit in MATLAB. In questo caso trattiamo una base *b* generica, prima di passare al modello a 64 bit. I *reali macchina* sono dei numeri con una quantità finita di cifre di mantissa e con un intervallo discreto di esponenti a disposizione che sposta la virgola a destra o a sinistra dando l’intervallo massimo e minimo dei numeri in modulo. Avendo *L* < 0 e *U* > 0 (estremi inferiore e superiore dell’intervallo), il numero rappresentato sarà nell’intervallo *L, L+1, … , -1, 0, 1, 2, …. U-1, U*.

Cercheremo quindi di capire:

* quanti sono i reali-macchina?
* quali reali sono approssimabili per arrotondamento con i reali macchina?
* come sono distribuiti nell’asse reale i reali-macchina?



I reali macchina sono l’insieme finito di tacche sull’asse reale, considerando lo 0, anch’esso reale macchina, i numeri stanno *nell’unione di due intervalli simmetrici*.

I due intervalli sono in particolare:

Immagine che contiene testo, antenna

Descrizione generata automaticamente

Questi intervalli nel continuo sono i reali approssimabili tramite i reali-macchina, sapendo che l’errore relativo sarà <= a. I numeri al di fuori di questi due intervalli sono troppo grandi oppure troppo piccoli per poter essere rappresentati e parliamo di overflow (intorno di ∞) ed underflow (intorno di 0).

In questi casi il numero rappresentato è fuori dall’intervallo rappresentabile e, nei vecchi linguaggi, questi errori potevano determinare l’arresto dello stato di calcolo del programma, dando degli errori oppure visualizzando *Inf* (*infinito)* oppure *NaN (Not a Number)*.

La struttura di un reale macchina è data da: *μ* = segno \* mantissa \*

Per cercare di prevenire underflow/overflow, conviene calcolare min e max .

Il *minimo numero positivo* si otterrà *moltiplicando* la *minima mantissa* per la *potenza della base con il minimo esponente*, quindi:

Immagine che contiene testo

Descrizione generata automaticamente

E quindi:



Invece il *massimo numero positivo* è ottenuto *moltiplicando* la *mantissa massima* per la *potenza della base col massimo esponente*, quindi:

Immagine che contiene testo

Descrizione generata automaticamente

avendo le t cifre uguali alla cifra massima, cioè b - 1.

Andiamo a calcolare la serie geometrica di ragione :

Immagine che contiene testo, orologio

Descrizione generata automaticamente



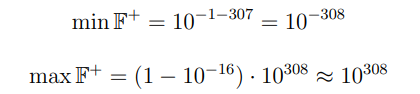
e con un po’ di passaggi si ottiene:

Immagine che contiene testo

Descrizione generata automaticamenteA questo punto, avremo che l’esponente massimo e l’estremo vero e proprio sono dati da:

Ad esempio, in base *10* con *16* cifre di mantissa, avendo esponenti minimo e massimo

, avremo quindi intervalli con estremi grandissimi e piccolissimi:



Importante ricordarsi che la rappresentazione avviene per arrotondamento e gli unici numeri rappresentabili dal calcolatore sono proprio i *reali macchina* (numero di tacchette intuitivamente). La *cardinalità* (quindi il numero degli elementi) è data dal numero delle possibili mantisse moltiplicato per il numero dei possibili esponenti. Quindi:



che sarà uguale a:



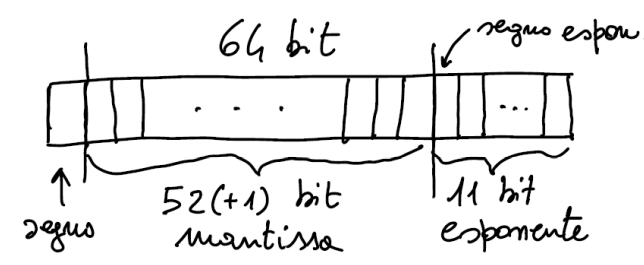
avendo *b* scelte per ogni cifra e *b-1* scelte per la prima cifra di mantissa. Andando fuori dagli intervalli detti siamo fuori dall’intervallo di rappresentazione.

Nel modello simil-Matlab di prima si ottiene una cardinalità data da un numero nell’ordine di .

Nel caso del modello simil-Matlab, in quanto i numeri rappresentati non sono in base 2. Si sfrutta lo standard IEEE dedicando ai numeri una sequenza di 64 bit.

Qui avremo 1 bit riservato al segno, 52 (+1, quindi come fossero 53, perché la prima cifra di mantissa deve essere non nulla) bit alla mantissa ed 11 bit dedicati all’esponente.

Graficamente questa è la situazione:



In ciascuna delle caselle possiamo rappresentare gli stati dei bit 0/1 (i segni sono dati da “+” o “-“ scegliendo uno dei due) e la precisione di macchina con un ordine di grandezza espresso da , in cui gli errori saranno a questa quantità. La precisione macchina è data da:

Immagine che contiene testo, orologio, screenshot

Descrizione generata automaticamente

Per il range degli esponenti, invece, avremo che il numero massimo (immagine sotto) e relativi estremi (immagine successiva) sono:

Immagine che contiene orologio

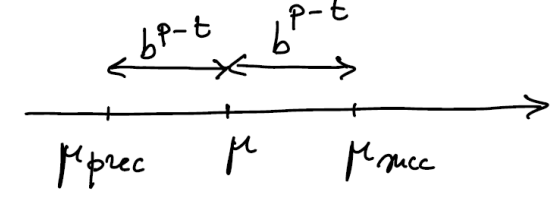
Descrizione generata automaticamente

Immagine che contiene testo

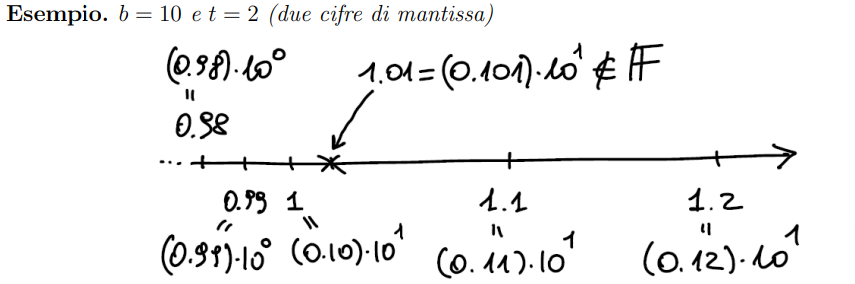
Descrizione generata automaticamente

Attenzione che potremmo avere comportamenti strani/imprevisti; questo perché stiamo lavorando in base 2, mentre appunto stiamo rappresentando tutto in base 10. L’interfaccia dei numeri è data da **F** (10, 16, -307, 308), avendo l’ordine di grandezza dei parametri chiave corretto.

Resta da analizzare come sono distribuiti i reali macchina, sapendo che questi numeri sono a *densità variabile* e *non sono distribuiti uniformemente*, avendo i numeri piccoli più vicini, mentre i numeri grandi più distanti tra di loro (si vede quindi che la densità varia).



Questo si può capire calcolando la distanza tra reali-macchina consecutivi (con lo stesso esponente). La distanza è e i numeri differiscono del minimo possibile, in particolar modo di 1 nella t-esima cifra di mantissa. Questa distanza è assoluta; a noi interessa invece una distanza relativa, sapendo che varia (grazie a *p*, quindi l’ordine di grandezza del numero in base *b*) quando si passa per una potenza della base.



*Chi è il reale-macchina successivo ad 1?* Pensando a due cifre decimali, si potrebbe rispondere 1.01. In realtà non va bene, perché 1.01 ha 3 cifre di mantissa e in questo esempio t=2.

Il reale macchina successivo ad 1 è in realtà *1.1*. Arrotondando i numeri da 0.99, 1, 1.1, avremo graficamente:

Immagine che contiene testo, antenna

Descrizione generata automaticamente

I reali macchina hanno come centro un intorno di approssimazione a *t* cifre di mantissa, avendo un errore assoluto <= *.*

Immagine che contiene testo, antenna

Descrizione generata automaticamente

I numeri vengono arrotondati per eccesso o per difetto da una parte o dall’altra. Passando per le potenze della base gli intorni diventano asimmetrici.

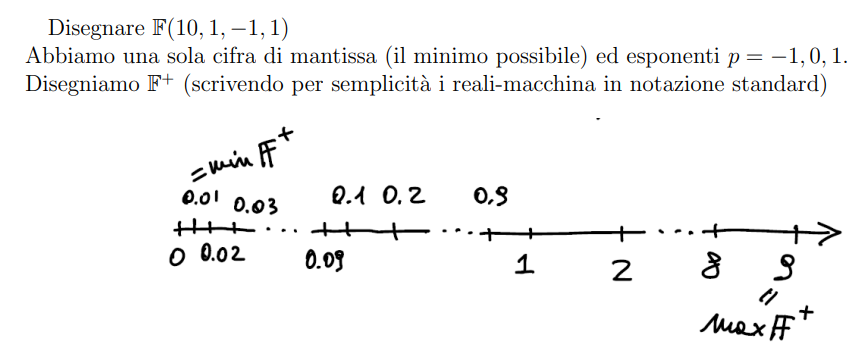
L’intorno è asimmetrico e, a sx o a dx, si allunga a seconda di un fattore *b*. Tutti i reali di questi intorni vengono approssimati dal reale-macchina centro dell’intorno per arrotondamento a *t* cifre di mantissa. Nel caso di errore relativo sappiamo che non supererà la precisione di macchina ε, sapendo che è indipendente dal parametro *p*.

L’unione di tutti questi intervalli che coprono il continuo degli intervalli di rappresentazione per arrotondamento (partendo dal discreto) è il seguente:

Immagine che contiene antenna

Descrizione generata automaticamente

Abbiamo quindi due esempi conclusivi:



Avendo una sola cifra, l’intervallo rappresentato è molto limitato e “povero”, dove la precisione macchina arriva al massimo fino al 50% e si ha un range di esponenti molto piccolo, che parte dai centesimi.

Un esempio più preciso può essere l’intervallo **F** (10, 2, -2, 2) per esercizio (facendo opportuni zoom sui vari ordini di grandezza). Riporto l’esempio molto semplice (nella parte finale dell’immagine avanza appunto di 2 partendo da 1, nel caso non si leggesse):

Immagine che contiene testo, lavagnabianca

Descrizione generata automaticamente

Questo è un intervallo più ricco del precedente, calcolando i giusti valori di max e di min di errori e di arrotondamento vero.

In questo caso avendo due cifre di mantissa, partendo dai millesimi, avrò i decimillesimi, poi ai centesimi coi millesimi, i decimi coi centesimi, ecc.

Questo esempio dovrebbe far capire la ricchezza del sistema floating-point a 64 bit (precisione doppia).

Analogamente esiste la precisione singola (32 bit) o precisione quadrupla (128 bit).

La precisione viene scelta dal linguaggio, permettendo di scegliere tra singola, doppia, quadrupla e solitamente vengono implementate a livello hardware (circuiti elettronici). Eventualmente è possibile aumentare il livello di precisione, implementando l’operazione aritmetica non più a livello hardware ma software, usando un algoritmo e quindi cresce il costo dell’operazione (risultando pesante in caso di grandi quantità di dati).

Tendenzialmente l’errore di misura è molto più “realistico”, arrivando a , o misure simili; chiaramente è un esempio ideale. Operazioni come la sottrazione può far perdere gran parte della precisione, unita ad altre operazzioni di calcolo eseguite. Ttutto questo studio conduce alla *stabilità* degli algoritmi numerici, considerando appunto la propagazione di errori in algoritmi instabili.

***09/03/2022: Lezione 4 - Operazioni aritmetiche con numeri approssimati***

Tutti i calcoli e le varie operazioni sono sempre affetti da errori. Dato un insieme di reali-macchina il modo in cui il processore realizza l’operazione tra due reali rappresentabili x, y è indicata come si vede sotto (caso moltiplicazione tra due reali rappresentabili *x* ed *y*):

Immagine che contiene testo

Descrizione generata automaticamente

In generale per le *operazioni macchina* si ha l’arrotondamento dei due reali, viene eseguita l’operazione tra gli arrotondamenti e il risultato viene a sua volta arrotondato.

Le operazioni macchina quindi sono così esprimibili:

Immagine che contiene testo

Descrizione generata automaticamente

L’esempio viene fatto con la moltiplicazione con **F** (10, 16, -307, 308) in cui poniamo:



Con le limitazioni degli esponenti reali si ha un overflow e:

Immagine che contiene testo

Descrizione generata automaticamente

Infatti si verifica che la proprietà associativa (anche la distributiva) e similmente la commutativa possono generare errori di arrotondamento (dati dal rimescolamento degli operandi).

Sempre con il sistema **F** (10, 16, -307, 308) si esegue:



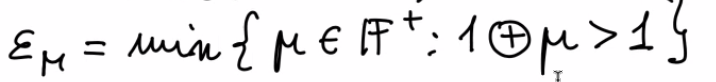
Non esiste inoltre più un unico elemento neutro nell’addizione, avendo 17 cifre significative ed eseguendo l’arrotondamento, che porta quindi ad 1.

Nel caso invece:



In questo caso invece le cifre significative sono 16 e non vengono fatti particolari arrotondamenti.

Si può quindi dare una seconda caratterizzazione della precisione di macchina (da noi non dimostrata):



Dato che non si può superare la precisione di macchina, l’errore chiave da stimare è l’errore relativo:

Immagine che contiene testo

Descrizione generata automaticamente

supponendo *x* ed *y* entrambi non nulli, tutto in funzione degli errori relativi sui dati εx ed εy:

Immagine che contiene testo, tavolo

Descrizione generata automaticamente

Nel caso dei dati approssimati quindi diremo *stabile* un’operazione aritmetica per cui l’errore sul risultato ha lo stesso ordine di grandezza dell’errore (massimo) sui dati.

Iniziamo con la *moltiplicazione:*



Usando la stessa tecnica di stima che si usa per dimostrare che il limite del prodotto di due successioni o funzioni è il prodotto dei limiti, aggiungendo e togliendo a numeratore ad esempio ˜xy.

Immagine che contiene testo, lavagnabianca

Descrizione generata automaticamente

Usiamo quindi la *disuguaglianza triangolare* per operare correttamente questa stima otteniamo:

Immagine che contiene testo, orologio, antenna

Descrizione generata automaticamente

La moltiplicazione è stabile, perché l’errore relativo sul risultato è maggiorato da una quantità che è dello stesso ordine di errore sui dati. Siccome , qualitativamente:

Immagine che contiene testo, orologio

Descrizione generata automaticamente



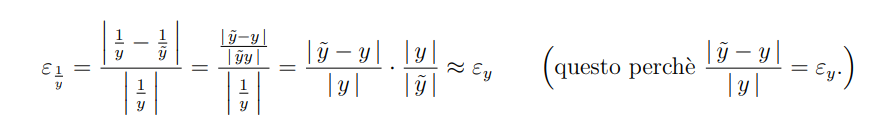
Per esprimere questo fatto possiamo usare la notazione:

La stima quindi, grazie anche alla *disuguaglianza triangolare* diventa:

Immagine che contiene testo, orologio

Descrizione generata automaticamente

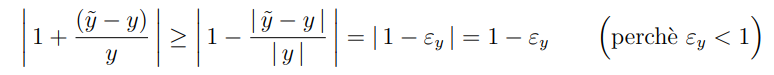
Passiamo ora alla *divisione*, essendo che è la moltiplicazione del reciproco, basta analizzare la stabilità dell’ordine di reciproco. Quindi:

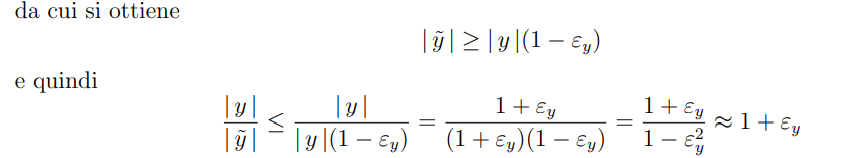


Assumendo che l’errore relativo sia minore di 1, situazione ragionevole. Usando la stima da sotto nella disuguaglianza triangolare (usandola nel senso opposto, cioè):



avremo:





L’ordine del reciproco ha stabilità, in quanto anche qui maggiorato dall’errore su “y”, considerando che l’errore relativo è <= 1. Parliamo quindi di *addizione* e di *sottrazione*.

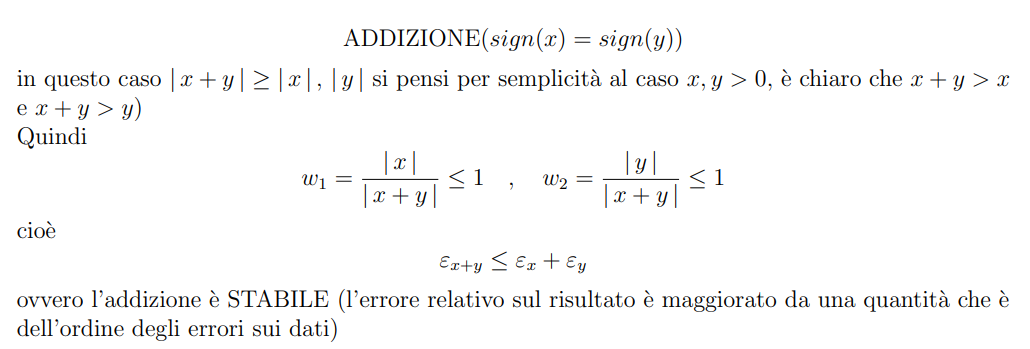
Andremo quindi ad analizzare gli errori sui dati della somma algebrica “x + y”.

Immagine che contiene testo

Descrizione generata automaticamente

Attenzione che i pesi dipendono “x” e da “y”, ma non dipendono dagli errori, avendo inoltre ragionato con una somma pesata degli errori sui dati con pesi ω1 ed ω2.

Per quanto riguarda l’addizione avremo (considerando che nell’addizione i pesi sono <= 1):



Nel caso invece della *sottrazione* avremo che |x + y| < |y|, quindi max {w1, w2} > 1, quindi la sottrazione può farci perdere precisione, ma quanta?

Può succedere che |x| e |y| siano molti vicini in termini relativi, cioè che |x + y| << |x|, |y| ed in queste situazioni i pesi w1, w2 sono >> 1 e la sottrazione diventa instabile.

(Ricordo che si intende “molto maggiore/molto più grande con “>>” e “molto minore/molto più piccolo” con “<<”)

I casi instabili quindi non sono quelli in cui |x+y| è “piccolo”, ma quelli in cui è piccolo rispetto a |x|, |y|.

Ad esempio, sono analoghi:

Immagine che contiene testo

Descrizione generata automaticamente

Detto a parole, due numeri dell’ordine delle unità che distano di qualche milionesimo sono altrettanto vicini, in termini relativi, a due numeri dell’ordine del milione che distano di qualche unità.

Da questo si può notare che la sottrazione è potenzialmente instabile (questo non accade sempre comunque). Se |x|ed |y| sono *distanti* in termini relativi, la sottrazione perde *poca* precisione; se invece gli stessi sono *vicini* si perderà *molta* precisione (definito anche come *cancellazione numerica*) e ci può portare ad un algoritmo instabile.

Si fronteggia in 2 modi:

1. cercare di riscrivere espressioni ed algoritmi in modo tale da evitare sottrazioni instabili (ad esempio nella formula risolutiva delle equazioni di secondo grado, aggirabile riscrivendo la formula con una semplice manipolazione algebrica);
2. aumentando la precisione in funzione della grandezza di w1 e di w2.

In campo sperimentale, quindi, questo aumenta la precisione, andando ad un sistema floating point a precisione estesa. Si consideri questo esempio per fissare le idee:

Immagine che contiene testo

Descrizione generata automaticamente

Alcuni commenti:

non è un numero macchina, in base 2, e con avrò 51 cifre significative, ma ne ho 53 a disposizione. Dato che si moltiplica 0, va bene e non si fa influenzare dall’arrotondamento.

Nel caso invece di 1 + 2^(-53) arrotonderei ad 1 ed il risultato verrebbe 0, in quanto, similmente a prima, avrei bisogno di 54 cifre significative.

***14/03/2022: Lezione 5 – Esempi di instabilità della sottrazione***

Nella scorsa lezione abbiamo analizzato la risposta delle operazioni aritmetiche agli errori sui dati. In particolare, dati x ed y in R, ed ~x circa x , ~y circa y, l’errore può essere espresso dalla disuguaglianza:

Immagine che contiene testo

Descrizione generata automaticamente

I pesi w1 e w2 sono maggiori di 0 i quali possono dipendere da x,y (ma non dagli errori). Nel caso delle altre operazioni (moltiplicazione, divisione, addizione), w1 e w2 sono circa 1 oppure <= 1, dunque sono considerate *stabili*.

Nel caso della sottrazione i pesi:

Immagine che contiene testo, orologio

Descrizione generata automaticamente

Se avessimo possono essere invece grandi. Questo accade in particolare se ed sono vicini in termini relativi, rendendo l’operazione di sottrazione instabile e distruggendo completamente il risultato rendendolo privo di significato (quando w1, w2 > {1/εx , 1/εy} ), attendendosi un errore relativo > 100%.

Noi ci focalizzeremo su errori relativi, lavorando su sistemi floating point in base 10.

Alla luce di queste considerazioni, seguono esempi che dimostrano l’instabilità della sottrazione.

Esempio 1

Si consideri **F** (10, 4, L, U) (rappresentando con L ed U numeri che rappresentano ciò che ci interessa),

con x = ed y =

Allora (espressi entrambi come , arrotondamento a 4 cifre floating point) ed

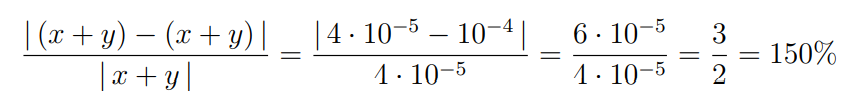
~y = .

Eseguendo l’operazione macchina è:

Immagine che contiene testo, antenna

Descrizione generata automaticamente

Invece e l’errore relativo sarà:



Si vede come avremo una perdita di precisione di ben 3 ordini di grandezza. Il problema è dai numeri scelti, dato che sono estremamente vicini tra di loro, dato anche che |x|, |y| che sono circa per ordine di grandezza.

Infatti, se calcoliamo i pesi:

Immagine che contiene testo, orologio

Descrizione generata automaticamente

dove analogamente anche w2 è circa 2500.

I fattori di amplificazione degli errori sono nell’ordine di e spiegano perché l’errore superi il 100%: si considera inoltre che questi fattori siano > 1/εr. Se avessimo avuto una cifra di mantissa in più non occorreva neppure arrotondare.

Esempio 2

Considerando invece **F**(10,8,L,U) con ed εr =

Vogliamo calcolare in floating point la somma algebrica , dove (si guardi soprattutto l’ordine di grandezza più che i numeri in sé):

a = , b = e c =

Il risultato della somma sarà:

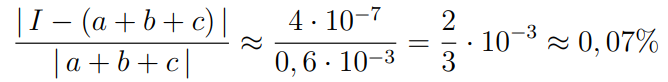
Tuttavia non vale la proprietà associativa, come si vede da:

Immagine che contiene testo

Descrizione generata automaticamente

Si verifica inoltre che *I* è diverso da *II* e si verifica che l’arrotondamento vero di *II* è .

Il risultato è quindi il meglio ottenibile da un’aritmetica floating point, con l’errore relativo di *I* dato da:



La perdita di precisione perde ben 4 ordini di grandezza, con un fenomeno di cancellazione di cifre significative, rispetto ad εr = .

Ora calcolo l’ordine di grandezza dei pesi:

Immagine che contiene testo, orologio, calibro

Descrizione generata automaticamente

e anche qui analogamente w2 è circa 50000.

Nell’espressione di calcolo *II non ci sta perdita di precisione*: la spiegazionesta nel fatto che i numeri a, b, c entrano con 8 cifre significative e non occorre arrotondarli, cioè εa = εb = εc = 0 quindi la sottrazione “b+c” non perde precisione (invece ne perderebbe se b oppure c fossero arrotondati).

Nota a margine, abbiamo usato 1 cifra significativa perché ci interessa non tanto l’accuratezza del risultato, quanto piuttosto l’ordine di grandezza.

*In generale in una sottrazione basta che uno dei due dati possegga errore e si perde precisione.*

Esempio 3

Si consideri la funzione seguente (dove f(x) coincide (≡) con 1):

Immagine che contiene orologio

Descrizione generata automaticamente

dove il Matlab esegue le operazioni in questo modo, avendo poi:

Immagine che contiene testo, lavagnabianca

Descrizione generata automaticamente

dove l’errore relativo nel calcolo è maggiore dell’11%, con la differenza tra 1 e 1.11.

Anche qui, 1 e sono estremamente vicini, avendo inoltre 1 che non viene arrotondato, mentre

viene arrotondato.

Questi piccolissimi errori di arrotondamento vengono però amplificati dal peso w1 nella sottrazione

,

Calcoliamo poi:

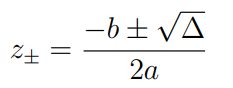
Immagine che contiene testo, oggetto, orologio, calibro

Descrizione generata automaticamente

Nel caso invece della sottrazione con è sempre dell’ordine di e quindi il risultato sarà esattamente uguale ad 1, perché sia che sono entrambi reali-macchina in base 2, pertanto non vengono arrotondati.

dunque ε1+x= 0.

Esempio 4

Consideriamo un’equazione di secondo grado del tipo

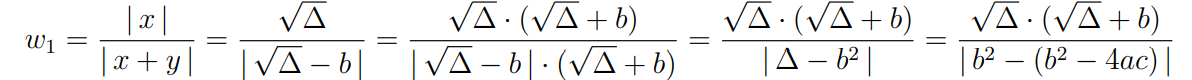
, dove sappiamo che le soluzioni si calcolano come:

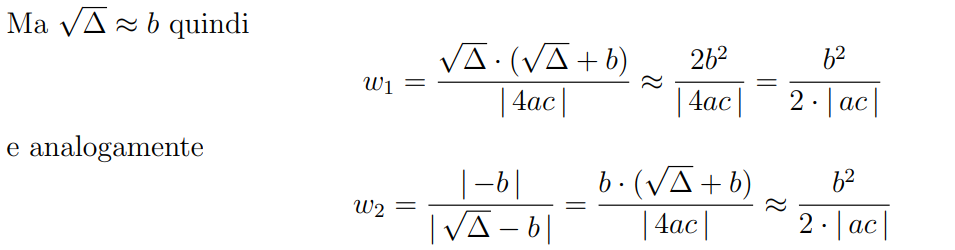
*Attenzione*: in aritmetica floating point la radice viene sempre arrotondata, dato che in essa vi è sempre un errore. Quest’ultimo sempre dal peso nel caso della sottrazione.

Prendiamo in oltre per semplicità *b* > 0, con la sottrazione il primo dato sicuramente affetto da un errore al massimo dell’ordine della precisione di macchina, considerando che nella sottrazione si perde precisione nel caso seguente:



Posti quindi x come e , avremo:





Possiamo quindi perdere molta precisione, ma successivamente basterà arrotondare la sottrazione. Quindi appunto la soluzione subisce fortissimi arrotondamenti e perdita di precisione a causa del rapporto che può diventare molto grande.

Ulteriore esempio

, con un , arrotondato poi a (avendo ). Quindi viene calcolato . Le soluzioni portano quindi ad un possibile errore relativo del 100%.

Per effetto dei coefficienti di amplificazione:

Immagine che contiene testo

Descrizione generata automaticamente

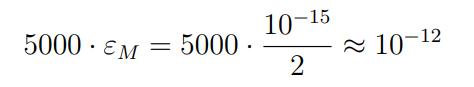
Consideriamo poi un ulteriore esempio:

Immagine che contiene testo

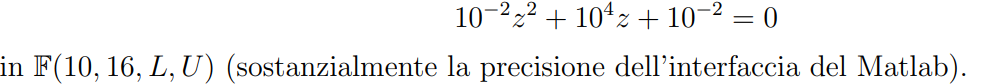
Descrizione generata automaticamente

Si noti che non è del tutto corretto, ma viene poi calcolato in maniera relativa, come al solito, per rendere effettivamente il calcolo utile.

Commettiamo un errore dello 0.01%, errore che potrebbe essere comunque accettabile in molti contesti applicativi, ma è 2000 volte più grande della precisione di macchina (poteva esserlo fino a 5000 volte, ma va tenuto presente che la precisione di macchina è una soglia massima e che gli errori relativi sono di solito più piccoli di essa). Se avessimo lavorato con t = 16 cifre di mantissa, ci saremmo aspettati un’amplificazione degli errori di arrotondamento fino a:



Facciamo un altro esempio (dove succede il disastro, cit. prof) considerando:



con un errore relativo totale, addirittura del 100%.

Immagine che contiene testo

Descrizione generata automaticamente

Si vede quindi che la formula delle equazioni di secondo grado sia molto instabile. Invece:

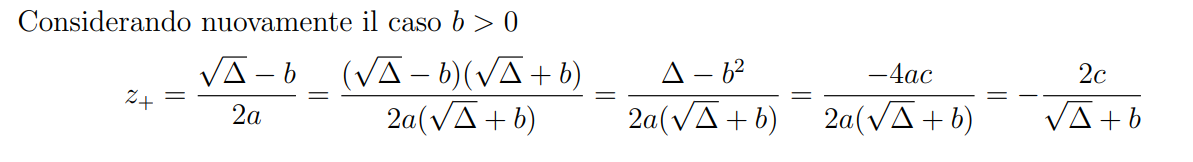


Immagine che contiene testo, orologio

Descrizione generata automaticamente

usando una formula stabile eliminando la sottrazione:

avendo poi una formula stabilizzata (non sempre è possibile eliminare la sottrazione, tuttavia si combinano altre formule (addizioni/divisione), ottenendo la stabilizzazione dell’operazione:

Immagine che contiene testo

Descrizione generata automaticamente

La perdita di precisione quindi non è completamente eliminabile, ma esiste un algoritmo grazie al quale l’errore relativo sulle due soluzioni con ≈ è dell’ordine di .

***15/03/2022: Lezione 6 - Propagazione degli errori, condizionamento delle funzioni***

Ci interessa in questa lezione l’effetto degli errori di arrotondamento su una variabile generica e la costruzione di un algoritmo iterativo che sfrutta una successione convergente alla quantità approssimabile.

Qual è l’effetto sul valore di *f* della variabile indipendente *x*?

Consideriamo quindi una funzione dove I è un intervallo, con ~x ≈ x, x diverso da 0, con il solito errore relativo εx.

Immagine che contiene testo

Descrizione generata automaticamente

Il cosiddetto *cond* rappresenta *l’indice di condizionamento* di f in x ed è la quantità che misura la risposta della funzione ad errori e quantifica l’amplificazione dell’errore sulla variabile.

Abbiamo il primo esempio di *problema instabile*, definibile in modo empirico e non formale.

Cominciamo quindi dal considerare:

Immagine che contiene testo, orologio

Descrizione generata automaticamente

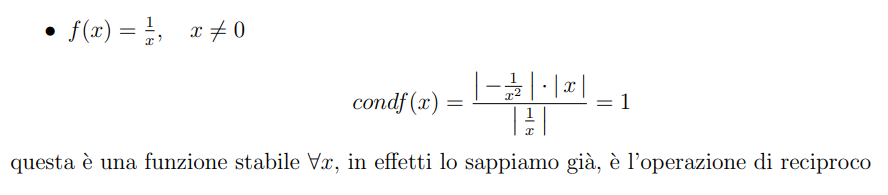
Questa è la funzione costante 1, che risponde in modo ottimale agli errori su *x*, dato che ovunque vale 1 e l’errore relativo vale 0.

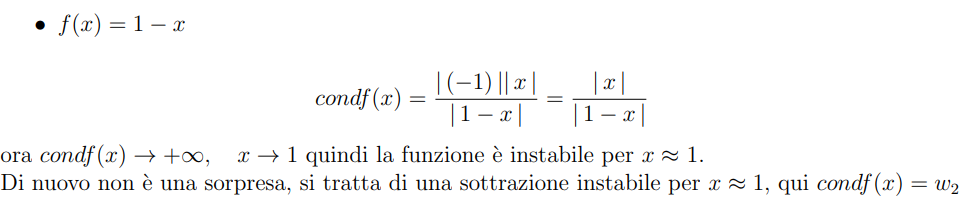
In aritmetica floating-point il calcolo è instabile se usiamo la formula scritta nel modo di prima.

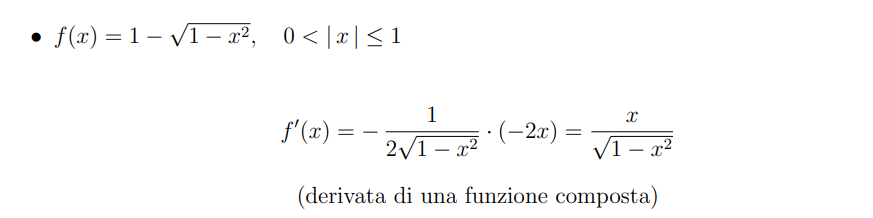
Per x 🡪 0, il peso w1 tende a +∞, a causa della piccolezza di |x|, crescendo la sottrazione ma avendo condizionamento ottimale.

Attenzione però: in sé la funzione è perfettamente stabile, tuttavia è l’algoritmo ad essere instabile.

Quello appena analizzato è un caso limite, ma ora diamo esempi di casi meno estremi:







dove il condizionamento (con qualche passaggio algebrico):

Immagine che contiene testo

Descrizione generata automaticamente

Avremo quindi che la funzione è ben condizionata, avendo che *condf(x)* 🡪2.

La formula di calcolo è quella che fa perdere precisione; sfrutteremo il solito trucco per rendere la funzione ed il calcolo stesso stabile, creandosi un altro algoritmo stabile.

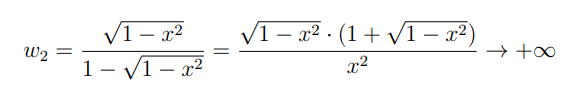
Ciò vale quando x è piccolo in modulo, quindi x ≈ 0.

Tuttavia il calcolo sull’espressione:



è *stabile* nella sottrazione *interna* alla radice, *instabile* invece nella sottrazione *esterna*.

Questo succede perché, per x 🡪 0, i numeri si allontanano in termini relativi e si vede da:



La formula quindi si stabilizza per x ≈ 0:

Immagine che contiene testo

Descrizione generata automaticamente

Questo succede perché tutte le operazioni coinvolte sono stabili e la funzione in sé strutturalmente risponde agli errori sulla variabile, che non dipende da come è scritta (quindi qual è l’algoritmo di calcolo) ma solo dall’indice di condizionamento *condf(x)*.

Dunque sta a noi scegliere un algoritmo stabile nel caso di una funzione ben condizionata e quindi errori piccoli, come arrotondamenti, possono dare grossi errori nei risultati.

Ci si aspetta quindi che se una funzione sia ben condizionata, *condf(x)* non sia grande.

Questo accade ad esempio:

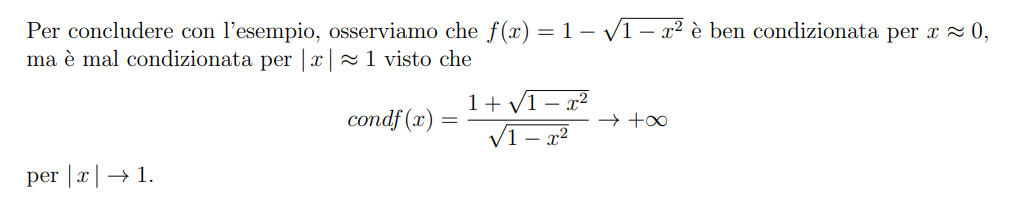
Immagine che contiene testo

Descrizione generata automaticamente

Viceversa, se una funzione è instabile ci aspettiamo che gli algoritmi siano a loro volta instabili.

Tutti questi concetti sono quindi empirici, perché ragioniamo in termini puramente astratti.

Nel caso sottostante ci si aspetta di avere errori, avendo una funzione in sé mal condizionata:



In questa seconda parte, abbiamo il calcolo di una successione che *converge* alla quantità che vogliamo approssimare tramite un *algoritmo iterativo*.

Affrontiamo quindi il calcolo di π in un sistema floating-point a 64 bit con un errore della precisione di macchina (avendo 15 cifre corrette nell’interfaccia in base 10).

Immagine che contiene testo, orologio

Descrizione generata automaticamenteConsideriamo quindi due successioni convergenti a π.

La prima si può ottenere dal fatto che la cosiddetta serie armonica ha come somma:

Immagine che contiene testo

Descrizione generata automaticamente

Detta poi la somma come , avremo:

Il calcolo della serie anche qui è stabile perché coinvolge solo operazioni stabili, ma la successione sotto radice è inutilizzabile per calcolare π con grande precisione, dato che la successione in sé converge lentamente.

Per il criterio di confronto integrale con serie:

Immagine che contiene testo, orologio

Descrizione generata automaticamente

schematizzato poi da:

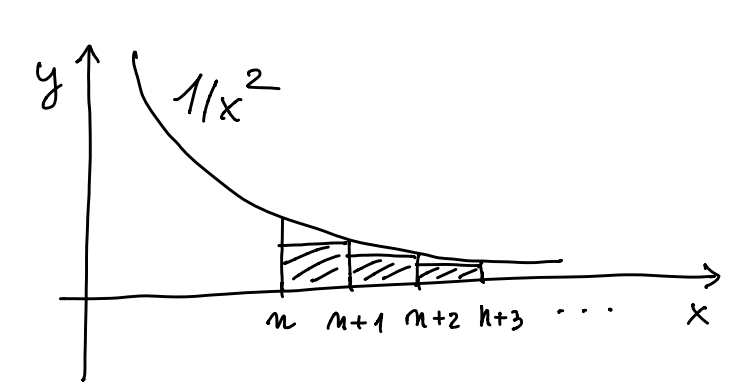


Immagine che contiene orologio, calibro

Descrizione generata automaticamente

Avremo quindi una stima apprrossimata con:

Per approssimare quindi correttamente l’errore in precisione macchina, dovremmo sommare .

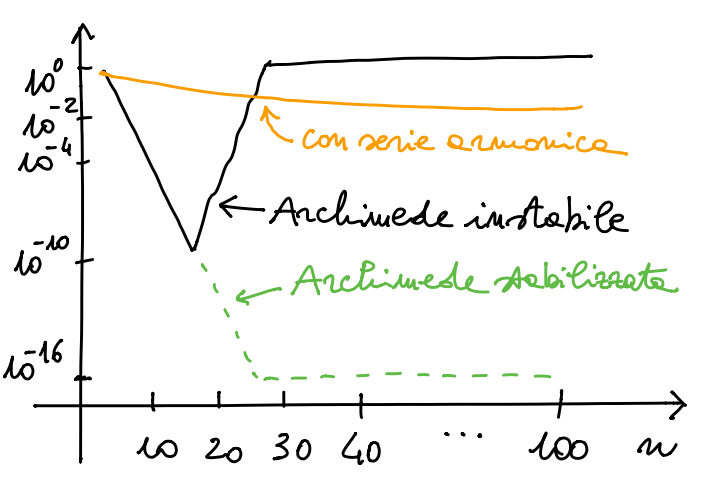
Nel caso invece di una successione definita per ricorrenza, cosiddetta “successione di Archimede”, avremo che il limite della successione per x 🡪 ∞ tende a π.

Mostriamo quindi com’è fatta questa successione:

Immagine che contiene testo, lavagnabianca

Descrizione generata automaticamente

Calcolando in Matlab l’errore relativo in scala logaritmica e ciclando in maniera regolare, ottiene in maniera grafica:



L’errore decresce linearmente fino a circa dopodiché comincia a crescere, dato che la successione con lo scherma descritto comincia ad allontanarsi da π.

L’errore quindi decade rapidamente in scala logaritmica, avendo poi un decadimento esponenziale, come si vede qui:

Immagine che contiene testo

Descrizione generata automaticamente

Dunque l’algoritmo è instabile e questa instabilità, da un certo *n* in poi, distrugge la convergenza della successione.

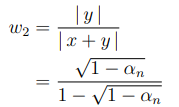
Di fatto l’instabilità proviene dalle 2 sottrazioni nell’operazione:



Di fatto la sottrazione 1 – an è stabile perché an è un infinitesimo.

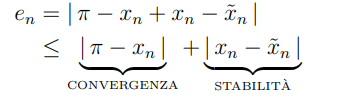
L’altra sottrazione (questa sopra dallo screen) diventa sempre più instabile al crescere di n, avendo una rapida convergenza ad 1.

Il peso w2 di questa sottrazione sarà sicuramente affetto da errore (dato dalla radice a numeratore), in cui:



che sarà circa 🡪

La stima dell’errore è dato da una combinazione della convergenza e la stabilità in questo modo:





e l’errore è legato alla convergenza teorica, avendo:

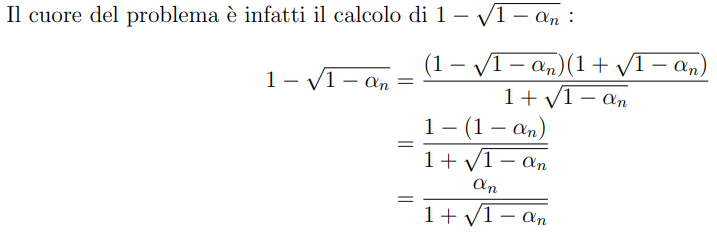
L’errore quindi è legato alla stabilità dell’algoritmo che implementa il metodo.

Immagine che contiene testo

Descrizione generata automaticamente

Ad un certo punto l’errore diventa costante; ciò accade quando an è molto piccolo, la successione è uguale a 0 e l’errore è del 100%.

Inoltre:



Dunque l’iterazione viene scritta (due formule coincidenti, ma in FP la seconda è stabile, la prima fortemente instabile):

Immagine che contiene testo

Descrizione generata automaticamente

Nel grafico degli errori presente sopra la successione stabiilizzata ha un errore coincidente con quella della successione, finché in quest’ultima si innesca l’instabilità distruttiva, stabilizzandosi poi sotto l’ordine della precisione di macchina, dato che si comincia a dominare sugli errori di arrotondamento.

Questa cosa detta si vede in termini formali dalla stima sottostante (dove ci aspettiamo che per *n* abbastanza grande, ci aspettiamo che il secondo addendo dell’ultima somma sia dominante nell’errore:

