TANA23 Matematiska algoritmer och modeller

Laboration 1. Felnalys och funktionsberäkning

1 Introduktion

Om vi vill använda en dator för att utföra matematiska beräkningar så måste vi acceptera beräkningsfel eftersom datorn arbetar med begränsad noggranhet. Syftet med denna laboration är att bli bekant med problem som uppstår på grund av att datorn inte kan räkna helt exakt.

Redovisning sker genom att en kort rapport skrivs där svaret på alla frågor finns med. Program skrivna i Python skall bifogas. Bifoga även grafer. Den färdiga rapporten skall skickas med epost till `fredrik.berntsson@liu.se`. Var noga med att skriva *TANA23 Lab1* i rubriken.

2 Datorns talsystem

Ett *flyttalssystem* kan beskrivas med fyra heltal, (β, t, L, U), där β är basen, t är antalet siffror i bråkdelen, och L och U är den minsta respektive största exponenten som är tillåten. Då tal x lagras i ett flyttalssystem görs ett relativt fel som är begränsat av

$$\left|\frac{\Delta x}{x}\right| \leq \frac{1}{2} \cdot \beta^{-t} = \mu,$$

där μ kallas *avrundningsenhet*, eller *maskinnoggranheten*. Avrundningsenheten är alltså en övre gräns för det relativa felet då ett tal x lagras i talsystemet (β, t, L, U).

Uppgift 2.1 Vi vet att basen är $\beta = 2$. Experimentera med olika värden på n och beräkna $1 + 2^{-n}$. Hitta det värde på n som gör att 2^{-n} avrundas bort och resultatet blir exakt 1. Vad är μ och t?

Uppgift 2.2 Ge en övre gräns för det *absoluta felet* felet när talet $x = \sqrt{2}$ lagras på datorn. Hur många korrekta decimaler har vi i datorns approximation av $\sqrt{2}$? Hur många *signifikanta siffror* har vi?

Uppgift 2.3 Ge även en övre gräns för både *absolut* och *relativt fel* då talet $x = 100 + \mathrm{e}^2$ lagras på datorn. Ange även hur många korrekta decimaler samt signifikanta siffror approximationen har.

Uppgift 2.4 Vi vet att flyttalssystemet använder 64 bitar för att larga tal. En av bitarna är teckenbit och t bitar går åt för att lagra bråkdelen. Använd tidigare resultat för att avgöra hur många bitar som används för att lagra exponentdelen? Beräkna uttrycket 2^n, för lämpliga värden på n och hitta den största tillåtna exponenten i talsystemet. Hitta även det största möjliga tal x som ingår i talsystemet.

3 Numerisk beräkning av ett gränsvärde

I detta kapitel skall vi studera beräkning av ett gränsvärde. Det är känt att

$$\lim_{x\to 0} f_1(x) = 1, \qquad \text{där} \qquad f_1(x) = \frac{\mathrm{e}^x - 1}{x}.$$

Vi skall försöka beräkna gränsvärdet i Python. För att undersöka noggranheten i beräkningen så skapar vi en x vektor som innehåller en stor mängd tal, mellan 10^{-16} och 1 och beräknar $f_1(x)$. Följande Python rader utför beräkningarna:

```
>>> import numpy as np
>>> x=np.linspace(0,16,161)
>>> x=10**-x
>> f1=(np.exp(x)-1)/x;
```

Vi plottar sedan felen $|f_1(x) - 1|$ i log-log skala genom att skriva

```
>>> import matplotlib.pyplot as pp
>>> pp.loglog(x,np.abs(f1-1))
>>> pp.show()
```

Uppgift 3.1 För vilket värde fås det minsta felet? Vad händer för mycket små x-värden?

Vi skall nu göra en teoretisk analys av felet i ovanstående beräkning.

Uppgift 3.2 Utnyttja att exponentialfunktionen kan skrivas som en serie

$$\mathrm{e}^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \ldots,$$

för att uppskatta *trunkeringsfelet* $R_T = |f_1(x) - 1|$. Detta är vad grafen ovan skulle visa om alla beräkningar utförts exakt på datorn.

Uppgift 3.3 Antag nu att alla beräkningar utförs med ett (`-`,`/`,`exp`, etc) utförs med ett relativt fel som är högst avrundningsenheten μ. Genomför en beräkningsfelsanalys som visar hur stort fel som görs då $f_1(x)$ beräknas på datorn. Hitta alltså en gräns för $|R_X| = |\mathrm{fl}[f_1(x)] - f_1(x)|$. Redovisa dina beräkningar och det färdiga uttrycket. Vad orsakar den dåliga noggranheten för små värden på x?

Uppgift 3.4 Plotta det totala felet $R_{TOT} = R_T + R_X$ i samma graf som det verkliga felet. Hur väl stämmer de bägge kurvorna överens?

Analysen ovan visar att det fel som görs vid beräkning av e^x orsakar ett stort fel i resultatet på grund av *kancellation*. Vi kan påverka detta genom att införa exakt samma fel på en annan plats i uträkningen. Gör vi det ändras motsvarande derivata under beräkningsfelsanalysen.

Uppgift 3.5 Undersök noggranheten hos det matematiskt ekvivalenta uttrycket

$$f_2(x) = \frac{\mathrm{e}^x - 1}{\ln(\mathrm{e}^x)}.$$

Utför samma experiment som ovan och plotta felet $|f_1(x) - 1|$ i log-log skala. Vad händer nu?

Eftersom uttrycken $f_1(x)$ och $f_2(x)$ är matematiskt ekvivalenta är trunkeruingsfelet detsamma. Däremot ändras beräkningsfelsanalysen. Detta eftersom e^x nu förekommer på två ställen och eftersom det är samma uträkning kommer vi att få exakt samma fel vid bägge tillfällena.

Uppgift 3.6 Genomför samma beräkningsfelsanalys som ovan för uttrycket $f_2(x)$. Visar beräkningsfelsanalysen att problemet med kancellation är löst?

Tips Detta är en rätt klurig uppgift. Titta på kvotregeln för beräkning av derivator. När uttrycket för R_X skall uppskattas gäller det att skriva om det på ett sätt som är enkelt att uppskatta med hjälp av serierepresentationen för e^x.

4 Beräkning av exponentialfunktionen

Standard funktionen e^x kan beräknas med hjälp av serierepresentationen,

$$\mathrm{e}^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{k=0}^{\infty} t_k, \qquad \text{där} \quad t_k = \frac{x^k}{k!}.$$

I denna övning skall vi implementera en funktion som utför beräkningen på ett så effektivt, och noggrannt, sätt som möjligt i Python. Eftersom Python inte kan lagra hur stora reella tal som helst behöver vi endast beräkna e^x för $0 \leq x \leq 709.7$. För större x-värden bör istället `Inf` returneras. Varför vi inte behöver betrakta negativa x-värden återkommer vi till.

Den enklaste tänkbara implementationen av beräkningen ovan är följande

```
def min_exp( x ):
    S=0.0
    for k in range(0,100):
    Tk=x**k/math.factorial(k)
         S=S+Tk
    return S
```

Denna implementation innehåller en mängd problem som måste åtgärdas. Exempelvis kan man inte vara säker på att summan konvergerat efter $n = 100$ termer. Vi skulle kunna använda en `while` loop och kontrollera konvergensen men då skulle beräkningen ta olika tid för olika x-värden och förutsägbarhet är önskvärt.

Uppgift 4.1 Det första vi skall undersöka hur bra koden ovan faktiskt är. Använd `linspace` för att skapa en vektor med 1000 jämt utspridda tal mellan -10 och 10. Använd funktionen `min_exp()`

för att beräkna en approximation av exponential funktionen för samtliga dessa x-värden. Jämför ditt resultat med funktionen `exp()` från `Numpy`. Plotta absolutbeloppet av skillnaden mellan dessa beräkninar. Kan du se någon skillnad mellan resultatet för positiva och negetiva x-värden? Plotta både de *absoluta* och *relativa* felen. Ge en förklaring. **Redovisa** graferna.

Uppgift 4.2 Det första vi skall åtgärda är problemen för stora negativa x-värden- Vi utnyttjar att $\mathrm{e}^{-x} = 1/\mathrm{e}^{x}$. Vi kan alltså först beräkna exponentialfunktionen fär ett positivt x och sedan utföra en division. Modifera funktionen `min_exp` så att detta utförs. Upprepa jämförelsen mellan `min_exp` och funktionen `exp()` från `Numpy` för att verifiera att problemet är löst. **Redovisa** återigen grafen.

Uppgift 4.3 Nu tar vi alltid med $n = 100$ termer i summan oavsett om det behövs eller inte. Det är givetvis otillfredställande. Vi vet att serien konvergerar långsammare för större värden på x. Vi vill därför begränsa oss till att beräkna e^{x} för $0 < x < 1$. Det kan vi åstakomma genom att utnyttja $\mathrm{e}^{a+b} = \mathrm{e}^{a}\mathrm{e}^{b}$. Om a är ett heltal kan e^{a} beräknas genom att talkonstanten e multipliceras ett antal gånger med sig själv. Vi kan alltså beräkna alltså e^{x} genom att dela upp x i heltalsdel och bråkdel. Eftersom summan konvergerar hytsat snabbt kan vi nu välja antalet termer så att

$$\frac{t_n}{\mathrm{e}^{1}} = \frac{1^{n}}{n!\mathrm{e}^{1}} \leq \mu$$

då kommer nästa term avrundas bort och inte påverka summan. Pröva dig fram och hitta ett lämpligt värde på n. För sedan in de ändringar som krävs i din Python kod. Upprepa igen noggranhetsjämförelsen med `Numpy` och **redovisa** grafen. Noggranheten bör vara ungefär samma som tidigare.

Uppgift 4.4 Det sista vi skall ändra är att termerna måste beräknas effektivare. Då vi beräknar termen t_k måste vi beräkna x^{k}. Eftersom vi beräknat x^{k-1} då förra termen beräknades kräver detta enbart *en* ny multiplikation med x och inte kst. Beskriv hur termen t_k kan beräknas givet den tidigare termen t_{k-1}. Implementera sedan detta sätt att beräkna termerna i ditt program. Tänk på att $t_0 = 1$. Kontrollera att vi fortfarande har samma noggranhet som tidigare version. **Redovisa** genom att bifoga den slutgiltiga versionen av koden.

Tips Detta har även fördelen att både x^{k} och $n!$ kan bli så stora att de inte kan lagras i datorns talsystem men trots det kan kvoten $x^{k}/n!$ lagras. Koden blir alltså säkrare att använda med denna ändring.