Titus Petersson

Utforska programmering med matematik

Att finna bästa närmevärde på pi har intresserat matematiker i hundratals år. När vi låter datorer göra jobbet uppstår fel som beror på avrundning av närmevärden. Hur bra är programmet? Hur snabbt blir det fel? Det är frågor som undersöks i den här artikeln där matematiken används för att närmare studera hur pålitliga datorerna är.

Kunskaper i programmering kan användas som verktyg för att underlätta räknandet och för att utforska matematik och matematiska modeller numeriskt eller grafiskt. Men det är också möjligt att vända på steken: Kunskaper i matematik kan användas som verktyg för att underlätta respektive utforska programmering. Exempel på detta är att utforska begreppen avkortning och snabbhet för att jämföra algoritmers begränsningar och effektivitet.

I Nämnaren 2021:4 redovisar Kilhamn, Bråting och Rolandsson resultatet från en studie av matematiklektioner om i vilken utsträckning matematikinnehållet är synligt i skolans programmeringsundervisning. De kategoriserade sina resultat i fyra nivåer, som i korthet är:

1. Enbart programmering, alltså utan koppling till matematikämnet.
2. Matematik som kontext för programmering, exempelvis att programmera ett spel för multiplikationsträning.
3. Programmering som verktyg för att effektivisera beräkningar, alltså främst att låta datorn göra det som annars hade motsvarat mödosamt handräknande.
4. Programmering som verktyg för att utforska matematik, alltså att utforska matematiska begrepp eller samband.

Ett resultat i deras studie var att under lektioner med programmering var det vanligare att matematiken hade en undanskymd roll (nivå 1–2) jämfört med där matematiken hade en fördjupande roll (nivå 3–4).

På isländska heter dator tölvu (töl + völva = tal + schaman, völva har sitt ursprung i fornnordiska som en titel för kvinnlig schaman) och matematik heter stærðfræði (stærð + fræði = storlek + teori). Det går förstås att använda datorn för att förutsäga tal. Men det går också att använda teorier om storlek hos beräkningar för att undersöka talens representation och algoritmers effektivitet. Med andra ord, det går att vända på steken och istället utforska programmering med matematik. Dessutom, den tredje nivån i denna hierarki är att effektivisera beräkningar, men det handlar inte bara om att slippa göra dem för hand utan att också om att använda matematik för att programmera effektivare algoritmer. Låt oss undersöka några exempel på detta.

Avkortning i datorers talrepresentation

Tabell 1 visar att det räcker med förkunskaper motsvarande decimaltal, subtraktion, grundläggande programmering och helst även tiopotenser för att undersöka egenskapen avkortning. Följande exempel börjar dock i gymnasiematematiken då derivata är ett naturligt tillfälle där avkortning uppstår vid numerisk derivering. Exemplet använder felsökning som princip för att identifiera att felet uppstår i själva subtraktionen i täljaren.
Derivatans definition innebär att undersöka differenskvoten

$$\frac{(f(x+h)-f(x))}{h} \text{ då } h \text{ går mot noll.}$$

Det verkar passa utmärkt för numerisk derivering och vi skriver ett program för detta som beräknar den numeriska derivatan för $f(x)=x^2$ för olika h och redovisar detta i kolumn 3 i tabell 1. För denna funktion blir differenskvotens algebraiska värde

$$\frac{(f(x+h)-f(x))}{h} = 2x+h.$$

Tabell 1 visar att den numeriska och den algebraiska differenskvotens värde ser ut att bli just densamma till och med potensen (-8), men därefter blir de numeriska värdena sämre av någon anledning för att till och med bli noll vid potensen (-16).

När en matematisk övning blir ett matematiskt problem föreslog Pólya en bland matematiker väl beprövad strategi: *byt mot ett enklare problem* och se om det hjälper till att förstå vad som händer. Denna strategi passar även för felsökning. Den kanske enklaste funktionen, som inte är en konstant, är proportionaliteten $f(x)=x$. Dess algebraiska differenskvot är

$$\frac{(f(x+h)-f(x))}{h} = 1, \text{ alltså exakt 1.}$$

Kolumn 4 i tabell 1 visar att vid potensen (-14) går något fel även med denna ytterst enkla differenskvot. För att undersöka om det har något med divisionen med små värden på h att göra låter vi kolumn 5 visa endast täljaren $(1+h)-1$ i differenskvoten till $f(x)=x$ och finner att divisionen med h inte är problemet.

Tabell 1. Numerisk derivata genom differenskvoter i punkten $x=1$.

Potens	h	$\frac{(1+h)^2-1^2}{h}$	$\frac{(1+h)-1}{h}$	$(1+h)-1$
0	1E+00	3,0000000E+00	1,000E+00	1,000E+00
-2	1E-02	2,0100000E+00	1,000E+00	1,000E-02
-4	1E-04	2,0001000E+00	1,000E+00	1,000E-04
-6	1E-06	2,0000010E+00	1,000E+00	1,000E-06
-8	1E-08	2,0000000E+00	1,000E+00	1,000E-08
-10	1E-10	2,0000002E+00	1,000E+00	1,000E-10
-12	1E-12	2,0001778E+00	1,000E+00	1,000E-12
-14	1E-14	1,9984014E+00	9,992E-01	9,992E-15
-16	1E-16	0,0000000E+00	0,000E+00	0,000E+00

Slutsatsen från tabell 1 blir att det måste vara i själva differensen $(1+h)-1$ i täljaren som något går fel. Detta fenomen beror på att numerisk beräkning i datorer använder ett ändligt antal siffror som beror på operativsystemets ordlängd (bitar). Precis som bråket $1/3 \approx 0{,}333$ måste avrundas i basen 10 så måste decimaltal i digitala verktyg avrundas. Från tabell 1 blir det tydligt att vid potensen $10^{(-14)}$ börjar avrundningsfelet märkas tydligt och vid potensen $10^{(-16)}$ lagras talet $(1+h)$ helt och hållet som 1 och inte alls som $(1+h)$. Fenomenet kallas avkortning (engelska: cancellation).

Det är helt enkelt så att datorn avrundar $(1+h) \approx 1$ när antalet siffror inte får plats i datorns ordlängd. Alltså, om datorn har plats för fyra siffror inklusive den inledande ettan, så kommer alla siffror efter det att avkortas, vilket innebär att datorn inte tar med de understrukna siffrorna i talet 1,000<u>01</u>.

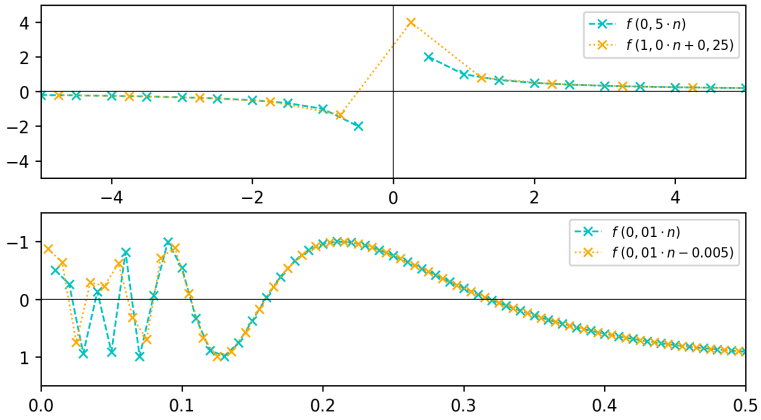
Det går bra att göra detta experiment med olika programvaror och olika operativsystem. Det kan visa sig att med en kombination av datorprogram och operativsystem sker avkortningen vid $10^{(-16)}$ medan avkortningen för en annan kombination av datorprogram och operativsystem sker vid $10^{(-17)}$.

Fenomenet avkortning har använts som humor i TV-serien Simpsons när Homer Simpson ställer upp ekvationen $3987^{12} + 4365^{12} = 4472^{12}$. Se kapitel 3 i boken *Räkna med Simpsons!*. Alla som känner till Fermats stora sats (bevisad av Andrew Wiles drygt 350 år efter att Fermat formulerade den) vet att denna likhet inte gäller. Men gör man beräkningarna på en miniräknare med högst nio gällande siffror ser det ut att vara en likhet just på grund av avkortning. För den alternativa beräkningen $(3987^{12} + 4365^{12})^{1/12} - 4472$ sker avkortningen istället vid ungefär 11 gällande siffror. Det passar därför bra att jämföra olika miniräknare och olika sätt att beräkna och inse att avkortning beror på både maskinvara och algoritm.

Konsten att plotta en funktion

I skolmatematiken kan man ibland fundera över varför man behöver studera funktioners egenskaper såsom lutning och nollställen algebraiskt när de är så lätt tillgängliga grafiskt med hjälp av digitala verktyg. Ett svar på en sådan fråga är att observera funktioner med x i nämnaren. Funktionen $f(x) = 1/x$ är välbekant för lärare och gymnasieelever, se figur 1. Men varför ger de blå och orangea kryssen olika kurvor? Detta beror på att de blå kryssen beräknar $f(0{,}5 \cdot n)$ medan de orangea beräknar $f(0{,}25 + 1 \cdot n)$. Det leder till att de orangea kryssen plottas som om kurvan vore kontinuerlig i $x = 0$ vilken den matematiska kurvan för $f(x) = 1/x$ inte är. Den blå kurvan försöker beräkna 1/0 men eftersom Python-kompilatorn returnerar ett kompilationsfel för $x = 0$ gör kurvan ett hopp där. Kalkylprogrammet Excel har inte denna felhantering utan sätter felaktivt $f(0) = 0$ i grafen, vilket ger en kontinuerlig kurva.

I grafen för $\sin(1/x)$ är det steglängden som ställer till det. Så länge funktionen ändrar sig långsamt $(0{,}1 < x)$ är detta inget problem men när x närmar sig noll växer $1/x$ snabbt vilket gör att perioderna i $\sin(1/x)$ blir så pass korta att steglängden 0,01 blir alldeles för lång för att rättvist avbilda funktionen. En elak övning för elever är att räkna antalet nollställen för funktionen $\sin(1/x)$ i ett intervall $[-10; 10]$. För att se detaljerna behöver de zooma in till $[-1; 1]$ och upptäcker då ännu fler nollställen och inser att de behöver zooma in ytterligare och hittar då ännu fler nollställen. En sådan övning kan lära eleverna att en plottad graf inte alltid är helt tillförlitlig. Man måste också skaffa sig viss förtrogenhet med de elementära funktionernas egenskaper.



Figur 1. Funktionerna f(x) = 1/x och f(x) = sin(1/x) plottade i Python (Numpy).

En praktisk tillämpning av detta är att betrakta grafritaren som en form av AD (analog-digital) omvandling där en 'analog' funktion plottas diskret för en serie av x-värden. Man måste plotta grafen med högre frekvens än funktionen svänger (denna slutsats kallas för samplingsteoremet). På samma sätt måste antenner för wifi och annat klara av en tydligt högre frekvens än vad själva wifi-signalen ligger på.

Piologi

Ett elevvänligt exempel på konvergens hos algoritmer är att beräkna rötter, vilket beskrivs utförligt i Nämnarenartikeln *Algoritmer + datastrukturer = program*. Där kan eleverna själva med hjälp av grundskolematematik formge och härleda olika algoritmer för rotberäkning och jämföra algoritmernas effektivitet.

Ett annat exempel är historiken om att beräkna talet π och här är det betydligt knepigare att härleda formlerna även om det är ganska lätt att jämföra deras effektivitet. Men först en fråga om varför det överhuvudtaget är meningsfullt att beräkna massor av decimaler i π. Jo, det finns två anledningar till detta. Den ena är att det är ett sätt att *mäta datorers snabbhet*. Precis som att maratonlöpare springer samma sträcka, så jämförs datorers snabbhet genom att göra beräkningar med samma algoritm på olika datorer.

Den andra är att datortillverkare låter en nyutvecklad processor beräkna π för att *undersöka om datorn fungerar korrekt*. En avvikelse från andra datorers värden på π avslöjar ett konstruktionsfel hos processorn eftersom talet π har den i detta fall användbara egenskapen att ha en slumpmässig decimalutveckling, vilket betyder att beräkningarna inte upprepar sig. Det innebär att datorns processor testas på otroligt många sätt vilket medför att sannolikheten att upptäcka ett konstruktionsfel är mycket hög.

Att jämföra effektivitet hos algoritmer

Att härleda formler för π är knepigt och den intresserade läsaren hänvisas till ProofWiki som är ett digitalt arkiv för matematiska bevis. Däremot är det någorlunda enkelt att programmera de färdiga algoritmerna och visa att de är olika snabba på att beräkna π. De tre vanliga formlerna är:

(1) Wallis produkt

$$\frac{2}{\pi} = \frac{(1\cdot 3)\cdot(3\cdot 5)\cdot(5\cdot 7)\cdot(7\cdot 9)\cdot\ldots}{(2\cdot 2)\cdot(4\cdot 4)\cdot(6\cdot 6)\cdot(8\cdot 8)\cdot\ldots}$$

(2) Vietas formel

$$\frac{2}{\pi} = (\sqrt{\frac{1}{2}})\cdot(\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}})\cdot(\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}})\cdot\ldots$$

(3) Chudnovksys ekvation

$$\frac{2}{\pi} = \frac{24}{(640320)^{3/2}}\left(\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!(13591409+545140134n)}{(n!)^3(3n)!(640320)^{(3n)}}\right)$$

Alla dessa algoritmer kräver förstås grundläggande kunskaper i programmering, eventuellt i kalkylblad för att enkelt få snygga tabeller. Förkunskaperna i matematik är en progression från att multiplicera och dividera heltal och eventuellt högstadiets konjugatregel för (1) samt rekursion och rotuttryck för (2) till gymnasiematematikens summatecken för (3).

Tabell 2 visar värdet för varje ny faktor i ekvation (1) och (2), respektive term för ekvation (3), för iteration n. Tabell 3 visar beräknat värde för $2/\pi$ =0,63661977… efter iteration n.

Tabell 2. Faktorer och termer i beräkningarna.

n=	Faktorer i Ekv. (1)	Faktorer i Ekv. (2)	Termer i Ekv. (3)
0	0,75	0,7071	0,636619772367593
1	0,9375	0,9239	-1,196213987731410E-14
2	0,9722	0,9808	6,238300782242480E-29
3	0,9843	0,9952	-3,486502983530300E-43
4	0,99	0,9988	2,026994255563500E-57

Tabell 3. Beräknat värde för $2/\pi$ =0,63661977… efter iteration n

n=	Ekv. (1)	Ekv. (2)	Ekv. (3)
0	0,75	0,7071	0,636619772367593
1	0,7031	0,6533	0,636619772367581
2	0,6836	0,6407	0,636619772367581
3	0,6729	0,6376	0,636619772367581
4	0,6662	0,6369	0,636619772367581

Wallis produkt (1)

John Wallis, engelsk matematiker 1616–1703

Wallis produkt kan programmeras på olika sätt. Betrakta dessa tre sätt att skriva faktorerna:

$$(5\cdot 7)/(6\cdot 6) = ((5\cdot 7)/6)/6 = (36-1)/36.$$

Det vänstra sättet innehåller två multiplikationer och en division. Det mellersta sättet innehåller en multiplikation och två divisioner. Det högra sättet använder konjugatregeln genom att $(n-1)(n+1)=(n^2-1)$ och behöver därför en multiplikation (för n^2), en subtraktion och en division. Dessutom kan man uppdatera $n^2 \to (n+1)^2$ med $2n+1$ genom kvadreringsregeln som därmed blir en addition (att multiplicera med 2 är enkelt med binära tal). Därför kräver den högra versionen något mindre räknearbete än de båda andra beräkningarna. Denna lilla skillnad går faktiskt att mäta i processortid om man gör en lång serie beräkningar med cirka 1 miljon faktorer. För en kort serie beräkningar kan dock tiden variera på grund av andra program som datorn kör i bakgrunden.

För varje delfaktor gäller förstås att $(n-1)(n+1)/n^2 < 1$, se tabell 2. Det betyder att Wallis produkt närmar sig värdet $2/\pi$ ovanifrån. En datorkörning visar dock att det behövs över 100 iterationer för att få tre korrekta decimaler.

Ett bevis för att Wallis produkt (1) från år 1656 ger π finns på Proof Wiki. Ungefär lika långsam som Wallis produkt är Maclaurinutvecklingen av $\arctan(x)$ för $x = 1$, som ger π genom en summa av bråk. Den kräver över fem miljarder termer för att beräkna ett närmevärde till pi med tio decimalers noggrannhet.

Vietas formel (2)

François Viète, latinskt namn: Franciscus Vieta fransk matematiker 1540–1603

Vietas formel kräver en del matematiskt förarbete för att kunna programmeras. Tilldela A värdet $\sqrt{1/2} \rightarrow A$ och konstatera att varje ny faktor motsvarar uppdateringen av A med $\sqrt{1/2 + A/2} \rightarrow A$.

Numeriskt kan man se att faktorerna i Vietas formel närmar sig 1 underifrån, precis som faktorerna i Wallis produkt. Det betyder att för varje iteration närmar sig värdet $2/\pi$ ovanifrån och att den redan vid iterationen $n = 4$ har tre korrekta decimaler i $2/\pi$. Ett bevis för att Vietas formel från år 1579 ger π finns på Proof Wiki.

Chudnovksys ekvation (3)

Srinivasa Aiyangar Ramanujan indisk matematiker 1887–1920

David Chudnovksy född 1945 Gregory Chudnovksy född 1952 emigrerade från dåvarande Sovjetrepubliken Ukraina till USA och gjorde stora framsteg i sitt arbete med en hemmabyggd superdator under 1990-talet.

I sitt arbete med att beräkna π använde bröderna Chudnovksy ekvation (3) som är inspirerad av Ramanujans olika formler för π. Det imponerande med denna formel är att varje ny term i summan ger en drös med ytterligare korrekta decimaler i $2/\pi$. I tabell 2 skiljer det ungefär 14 tiopotenser för ekvation (3) för varje iteration.

Det betyder att ungefär lika många nya korrekta decimaler fastställs i varje beräkningssteg, alltså att ekvation (3) faktiskt ger 14 korrekta decimaler redan vid första termen. Så trots att varje ny term i ekvation (3) har betydligt högre beräkningskomplexitet än dem i ekvationerna (1) och (2), så blir det totalt sett mindre räknearbete eftersom de mycket snabbare närmar sig π. För att sådana här beräkningar ska fungera, krävs det att datorprogrammet kan hantera många decimaler och komma runt avkortningsproblemet (se tabell 1). Men det är en annan historia.

LITTERATUR

Blatner, David (1998). *π – det fantastiska talet*. Svenska förlaget.

Kilhamn, C., Bråting, K. & Rolandsson, L. (2021). *Programmering i skolmatematiken?* Nämnaren 2021:4.

Petersson, J. (2018). *Algoritmer + datastrukturer = program*. Nämnaren 2028:2.

Singh, S. (2014). *Räkna med Simpsons!* Leopard.

Wikström, F. (2014). *Funktionsteori*. Studentlitteratur.

Proofwiki. https://proofwiki.org/wiki/Main_Page