KAPITTEL 2

Tall og datamaskiner

I dette kapitlet skal vi se på heltall og reelle tall og hvordan disse representeres på datamaskin. Heltall skaper ingen store problemer for datamaskiner annet enn at vi må begrense størrelsen på dem. Reelle tall skaper derimot er mer problematiske fordi vi ikke bare må begrense størrelsen, men også antall siffer. Dette kan gi opphav til avrundingsfeil som i ekstreme tilfeller kan gjøre at resultatet av numeriske beregninger blir fullstendig feil selv om beregningene rent matematisk er riktige. Vi diskuterer også to ulike måter å måle feil på, og avslutter med noen generelle kommentarer om objektorientert programmering og matematikk.

2.1 Naturlige, hele, rasjonale, reelle og komplekse tall

Fra *Kalkulus* vet vi at tallene kan deles inn i forskjellige klasser. Den minste klassen er de *naturlige tallene* som vi bruker når vi teller,

$$\mathbb{N} = \{1, 2, 3, 4, 5, 6, \ldots\}.$$

Tar vi med 0 og de negative tallene får vi mengden av *hele tall*,

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

Det neste steget kommer når vi begynner å dele opp ting i mindre deler, da trenger vi brøkene eller de *rasjonale tallene* som vi betegner med $\mathbb{Q}$. Dette er tall på formen

$$\frac{p}{q}$$

der p og q er hele tall med den begrensningen at q må være ulik 0. Setter vi $q = 1$ ser vi at alle de hele tallene er en delmengde av brøkene.

Som vi vet fra skolen, kan de rasjonale tallene skrives som desimaltall. For eksempel

er[1]

$$\begin{aligned} \frac{1}{2} &= 0.5, \\ \frac{3}{16} &= 0.1875, \\ \frac{97}{25} &= 3.88. \end{aligned} \tag{2.1}$$

For de fleste rasjonale tall får vi uendelig mange desimaler når vi skriver dem som desimaltall,

$$\begin{aligned} \frac{1}{3} &= 0.333333333\cdots \\ \frac{3}{7} &= 0.42857142857142\cdots \\ -\frac{10}{13} &= -0.7692307692307692307\cdots \\ \frac{11}{17} &= 0.647058823529411764705882352941\cdots \end{aligned} \tag{2.2}$$

I eksemplene over må vi altså ha uendelig mange desimaler, men heldigvis er det et system, sifrene gjentar seg etter en stund. Dette gjelder generelt: Når rasjonale tall skrives som desimaltall får vi enten et endelig antall desimaler slik som i (2.1) eller en endelig sekvens av desimaler som gjentar seg uendelig mange ganger.

I dagliglivet kommer vi langt med de rasjonale tallene, men i matematikken trenger vi flere tall. For eksempel trenger vi tallet a som er slik at $a^2 = 2$, altså kvadratroten $a = \sqrt{2}$. Som vi skal se senere er dette tallet ikke rasjonalt, det sies derfor å være irrasjonalt. Det finnes uendelig mange irrasjonale tall, for eksempel er alle kvadratrøtter av naturlige tall irrasjonale dersom de ikke er heltallige, og e og π er også irrasjonale. Faktisk er det slik at det i en presis forstand fins flere irrasjonale tall enn rasjonale tall.

Dersom vi legger de rasjonale tallene til de irrasjonale tallene får vi de reelle tallene $\mathbb{R}$. Denne mengden inneholder altså både de naturlige tallene, de hele tallene, de rasjonale tallene og de irrasjonale tallene og fyller tallinja fullstendig slik at det ikke blir hull. *Kalkulus* inneholder mye stoff om de reelle tallene, og vi skal komme tilbake til noe av dette senere.

Den siste utvidelsen av tallene som vi skal studere er de *komplekse tallene* som vi vanligvis betegner med $\mathbb{C}$. Fordelen med å innføre disse tallene er at da kan vi løse ligningen $x^2 = -1$ og andre ligninger som krever at vi tar kvadratroten og mer generelle røtter av negative tall. Et helt kapittel i læreboka er tilegnet de komplekse tallene, men vi skal ikke si så mye om disse i dette kompendiet.

[1] Legg merke til at vi ikke bruker desimalkomma, men desimalpunktum. Desimalpunktum er vanlig i de fleste andre land og skaper ikke så lett forvirring når vi skriver matematikk. Skal vi liste opp de tre tallene 3,2, 2,8, 4,1 blir det lett uoversiktlig med desimalkomma. Med desimalpunktum blir det mye klarere, 3.2, 2.8, 4.1.

2.2 Datamaskiner og heltall

Datamaskiner ble før ofte kalt for regnemaskiner fordi de var bygd spesielt for å automatisere regning med tall. I dag brukes datamaskiner til å manipulere både tekst, lyd og bilder (og mye annet) i tillegg til tall, men på laveste nivå representeres allikevel alt som tall. La oss se litt overfladisk på hvordan dette gjøres.

2.2.1 Representasjon av heltall

På en datamaskin skiller vi mellom *heltall* (*integers* på engelsk) og *flyttall* (*floating point numbers* eller *floats* på engelsk). Heltallene på en datamaskin er de samme som de matematiske heltallene $\mathbb{Z}$ som vi nevnte over. Forskjellen er bare at det er umulig å få en datamaskin til å håndtere alle mulige heltall. Siden det er uendelig mange av dem ville det kreve uendelig med ressurser (lagerplass og regnekapasitet) å regne med alle mulige heltall. Vi må derfor sette en grense for hvor store heltall maskinen skal arbeide med.

For å forstå begrensningene som maskinen setter er det nyttig å vite hvordan heltallene representeres. Tall kan representeres i mange tallsystemer. Vi er vant til å bruke titallsystemet og sifrene 0—9 slik at når vi skriver 3489 så angir det tallet

$$3489 = 3 \cdot 10^3 + 4 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0,$$

men det er ingenting i veien for å bruke andre tallsystemer med andre grunntall enn 10. Bruker vi for eksempel grunntall 7, så har vi sifrene 0, 1, 2, 3, 4, 5 og 6 til rådighet og tallet 235_7 (grunntallet angis med indeks hvis det ikke er 10) er det samme som

$$235_7 = 2 \cdot 7^2 + 3 \cdot 7^1 + 5 \cdot 7^0 = 2 \cdot 49 + 3 \cdot 7 + 5 = 124.$$

Datamaskiner bruker som regel to-tallsystemet. Dette har den fordelen at vi bare trenger de to sifrene 0 og 1, de to *binære* sifrene. Dette er en fordel fordi det gir slingringsmonn når maskinen skal tolke hva den har lagret. For å lese et lagret tall må maskinen lese hvert av sifrene og da er det bare to muligheter: enten er det 0 eller så er det 1. Hvis sifferet i maskinen for eksempel er representert ved en spenning så kan vi tillate at alt mellom 0 volt og 2 volt skal tolkes som 0 mens alt mellom 4 volt og 6 volt skal tolkes som 1 (andre spenninger vil rapporteres som feil). Dersom vi opererte i et tallsystem med flere siffer ville vi ikke kunne bruke et så stort spenningsområde for hvert siffer og dermed ville vi bli mer utsatt for feil.[2]

[2] Dette er forøvrig en av de store fordelene med dagens *digitale* teknologi som omgir oss på alle kanter. Når musikk lagres digitalt betyr det at lydsignalet leses av med jevne mellomrom (44 100 ganger pr. sekund på en CD-plate) og lagres som et tall på binær form. Når dette skal leses ved avspilling har vi samme fordel som over, det skal bare avgjøres om det er 0 eller 1. En del fett og støv kan derfor tolereres på en CD-plate siden 'vi har en del å gå på´. På en gammeldags kassett derimot lagres lydsignalet ved å magnetisere kassettbåndet og det er intensiteten i magnetiseringen som sier hvor sterkt lydsignalet er, på en *analog* (kontinuerlig) skala. Litt fett eller andre forstyrrelser på båndet vil svært lett endre magnetiseringen og dermed det avleste lydsignalet, og dette vil høres som støy eller sus. Det samme forholdet gjør seg gjeldende når vi sammenligner digital (GSM og ISDN) og analog telefoni eller analog eller digital overføring av TV-signaler.

Datamaskiner representerer altså tall i to-tallsystemet og bruker kun sifrene 0 og 1. I dette tallsystemet skrives for eksempel det desimale tallet 13 som

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13,$$

mens 121 binært blir 1111001_2. Dersom vi bruker 2 desimale siffer kan vi på en naturlig måte representere 100 heltall, nemlig tallene 0–99, og hvis vi bruker n desimale siffer kan vi representere 10^n tall (fra 0 opp til $10^n - 1$). På samme måte kan vi med n binære siffer representere 2^n heltall, nemlig tallene fra og med 0 opp til og med $2^n - 1$. Dagens datamaskiner har *maskinvare* (elektronikk) som håndterer 32 eller 64 binære siffer (ofte kalt *bits*) svært effektivt slik at programmer og programmeringsspråk som regel holder seg til dette. Dersom vi bruker 32 bits kan vi representere

$$2^{32} = 4294967296$$

forskjellige heltall. Siden vi skal ha både negative og positive tall fordeler vi disse tallene så jevnt vi kan på hver side av null og tillater heltallene mellom -2^{31} og $2^{31} - 1$, hvilket vil si tallene

$$-2147483648, \ldots, -3, -2, -1, 0, 1, 2, 3 \ldots, 2147483647.$$

Dette svarer til at vi bruker ett bit til å angi fortegnet til tallet og de resterende 31 bits til å angi tallverdien. Dessuten regner vi 0 til de positive tallene.

Hvis vi går opp til 64 binære siffer kan vi håndtere betydelig større heltall, nemlig tall fra -2^{63} opp til $2^{63} - 1$,

$$-9223372036854775808, \ldots, -2, -1, 0, 1, 2, \ldots, 9223372036854775807.$$

Det er støtte for disse to typene heltall i de fleste programmeringsspråk. I Java får vi tilgang på 32 bits heltall ved å benytte variabeltypen `int` mens 64 bits heltall er gitt ved variabeltypen `long`. I Java kan vi også benytte 8 bits heltall ved typen `byte` og 16 bits heltall ved typen `short`. Beregninger med `byte` og `short` tall foregår ved at de internt i maskinen blir betraktet som 32 bits heltall av typen `int`. Det er derfor ikke noe raskere å regne med disse mindre heltallstypene.

Det er verdt å merke seg at standardtypen for heltall i Java er `long`, altså 64 bits heltall. Når Java finner et heltallsuttrykk vil det derfor regnes ut som et 64 bits uttrykk.

I de fleste språk fins det predefinerte konstanter som inneholder største og minste heltall for en gitt heltallstype. I Java ligger disse konstantene i klassene `Long` (64 bits heltall) og `Integer` (32 bits heltall). Disse klassene inneholder også en del andre funksjoner som kan være nyttige ved håndtering av heltall.

2.2.2 Heltallsaritmetikk

I tillegg til å kunne lagre heltall i datamaskinen må vi også kunne regne med dem. Maskinen har egne elektroniske kretser som utfører aritmetikk med tall, og vi får tilgang til dette via et passende programmeringsspråk. I de fleste språk ligger *syntaksen* (notasjonen) for de vanlige aritmetiske operasjonene nær opptil det vi er vant til fra matematikkbøker, men i tillegg er det ofte støtte for andre operasjoner som det ikke fins

noen standard notasjon for i matematikk. I Java som i de fleste andre språk, betegner `+`, `-`, `*` og `/` henholdsvis addisjon, subtraksjon, multiplikasjon og divisjon, mens `a % b` gir resten når a divideres med b. En fullstendig liste over tillatte operasjoner for variable av typen `int` og `long` kan du finne i en lærebok for Java.

Det er essensielt bare en ting som kan gå galt ved behandling av heltall på en datamaskin og det er at vi støter på et tall som er for stort (i tallverdi) til å representeres i det formatet vi opererer i. Som vi har sett går grensen for 32-bits heltall ved tall som er mindre enn -2^{31} eller større enn $2^{31}-1$ mens den for 64-bits heltall går ved tall som er mindre enn -2^{63} eller større enn $2^{63}-1$. En typisk situasjon der vi vil kunne overskride formatet er ved addisjon av to store tall. Dersom vi bruker 64-bits heltall og forsøker å addere de to tallene $a = 2^{63}-2$ og $b = 2^{63}-7$ blir resultatet

$$a + b = 2^{63} - 2 + 2^{63} - 7 = 2 \cdot 2^{63} - 9 = 2^{64} - 9.$$

Tallet $2^{64}-9$ er større enn $2^{63}-1$ og kan derfor ikke representeres som et 64-bits heltall. I en slik situasjon vil maskinen bare overse noen siffer slik at resultatet kan virke nokså vilkårlig (denne oppførselen er definert av elektronikken i maskinen og ikke det aktuelle programmeringsspråket). En slik feilsituasjon kalles *overflow*, og vil alltid oppstå når resultatet av en operasjon på heltall resulterer i et tall med for stor tallverdi. I tillegg til ved addisjon kan dette også skje ved subtraksjon og multiplikasjon og ved tilordning av en for stor konstant til en variabel.

En annen feilsituasjon oppstår når vi forsøker å dividere noe med 0. Når dette skjer i beregninger som involverer heltallsvariable vil Java protestere og gi en såkalt `exception`.

2.2.3 Vilkårlig store heltall

Selv om vi med 64 bits heltallsvariable kan håndtere ganske store tall er det altså en øvre grense for størrelsen på tallene. Dersom maskinen ved en utregning ender opp med et tall som går utover de tillatte grensene får vi ingen feilmelding, men tallene som er for store blir gjort om til tall som ligger innenfor det lovlige området. Dette kan ofte være vanskelig å oppdage hvis vi ikke har klare tanker om hvordan resultatet av beregningene skal være. Det er derfor viktig å vite på forhånd at datatypen vi bruker (`int` eller `long`) er stor nok til å håndtere beregningene våre. Siden det i matematikkens verden finnes vilkårlig store heltall er det altså uendelig mange heltall vi ikke kan bruke på en fornuftig måte, selv med variable av typen `long` (64 bits). I de fleste sammenhenger er dette uten betydning siden det er svært sjelden vi har behov for å regne med heltall som i tallverdi er større enn 2^{63}. Men dersom behovet skulle melde seg så fins det en løsning.

Løsningen består i å skrive et program som håndterer store heltall. Det vanlige er å la størrelsen på tallene være helt fri, uten noen begrensninger. Når programmet møter et heltall, enten som inngangsverdi fra brukeren eller som resultatet av en beregning, vil det avsette plass nok til å representere hele tallet, uansett størrelse. For å utføre operasjoner på slike store tall må tilhørende aritmetiske algoritmer være implementert. På denne måten kan vi komme unna begrensningene som maskinvaren setter.

Ut fra dette kan man forledes til å tro at datatypene `long` og `int` som er definert ut fra maskinvaren er overflødige. Det er ikke tilfelle. Beregninger med store heltall som

krever egne programmer for aritmetiske operasjoner går svært, svært sakte i forhold til operasjoner på 'normale´ heltall som har aritmetiske algoritmer implementert i maskinvare. Dessuten skal man også være klar over at det ikke på noen datamaskin er mulig å representere alle mulige heltall. Selv om programmet som håndterer tallene ikke setter noen eksplisitt grense vil størrelsen på maskinens hukommelse og disk alltid sette begrensninger siden tallene må lagres internt i maskinen. Ved å øke disse ressursene kan vi arbeide med større heltall, men det vil aldri bli mulig å regne med uendelig mange siffer når vi har endelige ressurser. I tillegg vil det altså kunne ta svært lang tid å utføre beregninger med mange siffer.

Heldigvis har andre allerede gjort jobben med å implementere rutiner for store heltall. Programmer som Maple og Mathematica har innebygd støtte for slike tall og vil selv konvertere fra 64 bits til store heltall etter behov. I tradisjonelle programmeringsspråk er det vanligvis tilgang på biblioteker for å regne med store heltall. Java har klassen `java.math.BigInteger` som implementerer heltallsaritmetikk med vilkårlig mange siffer.

Når vi bruker en datatype som kan håndtere vilkårlig store heltall kan vi ikke få overflow siden det i prinsippet ikke er noen øvre grense for størrelsen på tallene vi kan håndtere. Men som nevnt over, setter de tilgjengelige maskinressursene (hukommelse og hastighet) begrensninger i praksis. Hastighetsgrensen skaper ikke større problemer enn at vi risikerer å måtte vente svært lenge på et resultat, men hukommelsegrensen er mer alvorlig. Dersom vi støter på et heltall som er så stort at programmet ikke har tilgang på nok hukommelse til å lagre tallet vil det måtte gi en feilmelding og beregningene må avbrytes.

2.3 Datamaskiner og reelle tall

For å kunne utnytte datamaskiner innen matematikk må maskinene kunne arbeide med reelle tall på en tilfredstillende måte, og dette gjøres vanligvis ved hjelp av *flyttall*. Flyttall og deres særegenheter er et omfattende tema som vi ikke skal gå inn på i detalj her, men det er nødvendig å kjenne de grunnleggende prinsippene for hvordan flyttall håndteres for å kunne forstå noen av problemene som kan oppstå i forbindelse med numeriske beregninger.

Mange har opplevd at lommeregneren ikke alltid regner helt riktig. For eksempel fins det en lommeregner (HP 48G) som regner ut $\sqrt{2}$ som

$$1.41421356237.$$

Kvadrerer vi dette tallet på den samme lommeregneren får vi ikke 2, men 1.99999999999 som svar. Feilen som lommeregneren gjør er svært liten, og hvis vi runder av til 11 siffer blir kvadratet 2, slik vi forventer. Problemet er at den aktuelle kalkulatoren bare regner med 12 siffer mens $\sqrt{2}$ er et irrasjonalt tall med uendelig mange desimaler. Når vi forsøker å representere $\sqrt{2}$ som et tall med 12 siffer (11 desimaler) gjør kalkulatoren derfor en liten feil, og når den tilnærmede kvadratroten multipliseres med seg selv trekker vi denne feilen med oss. I tillegg gjør også lommeregneren en liten feil i selve multiplikasjonen med den konsekvens at den forteller oss at kvadratroten av 2 opphøyd i annen ikke er nøyaktig 2. Dette fenomenet kalles *avrundingsfeil*.

Prosessen vi har skissert over gjelder også for moderne kalkulatorer og datamaskiner, selv om de fleste vil vise det riktige svaret 2 på regnestykket over. Dette kan skje fordi de regner med flere siffer enn de viser. Maskinen kan for eksempel regne med 16 siffer. Selv om det sekstende sifferet er feil vil svaret bli det riktige tallet 2 når det avrundes til 12 siffer. Men selv om svaret som maskinen viser er riktig i dette tilfellet er problemet med avrundingsfeil altså fremdeles til stede.

De fleste beregninger med desimaltall på kalkulatorer og datamaskiner er ikke helt nøyaktige og gir avrundingsfeil, og jo flere beregninger som er nødvendige før vi er framme ved det endelige resultatet dess større vil feilen i det endelige resultatet vanligvis være. Stort sett er feilene så små at det bare er de aller siste sifrene som er feil, men enkelte beregninger er så problematiske at en betydelig andel av de utregnede sifrene blir feil, ja vi skal senere se eksempler der svaret som datamaskinen produserer ikke inneholder noen riktige siffer i det hele tatt. Slike problematiske beregninger sies å være *dårlig kondisjonerte*, mens de som det kan påvises ikke gir særlig avrundingsfeil sies å være *godt kondisjonerte*.

Hvis det er mulig at alle sifrene i et beregnet tall kan bli feil er det åpenbart av stor betydning å vite når dette kan skje. Hvis en flyfabrikant for eksempel forsøker å estimere belastningene som vingene på et fly som er under planlegging kan bli utsatt for bør vi være sikre på at tallene som regnes ut er til å stole på, ellers kan følgene bli katastrofale. Slik analyse av beregninger med tanke på å påvise nøyaktigheten er en del av fagfeltet *numerisk analyse*.

Som nevnt over er studiet av avrundingsfeil for omfattende til at vi kan gå inn på alle detaljer her, men i resten av denne seksjonen skal vi se litt på hvordan reelle tall vanligvis representeres i en datamaskin og hvordan dette kan gi opphav til avrundingsfeil.

2.3.1 Reelle tall på normalisert form, flyttall

Siden en datamaskin har endelige ressurser kan vi ikke ha noen forhåpning om å kunne representere alle mulige reelle tall, spørsmålet er bare hvilke som skal tas med og hvilke som skal utelates. Utgangspunktet er at vi betrakter et reelt tall som et desimaltall som kan ha vilkårlig mange siffer både til venstre og høyre for desimalpunktum (merk at vi bruker punktum for desimalkomma i dette kompendiet). For å kunne representere slike tall i datamaskinen må vi gjøre to begrensninger. Som for heltall må vi begrense det totale antall siffer vi tillater i et tall. Dette begrenser nøyaktigheten vi kan operere med. Hvis vi for eksempel begrenser antall siffer til fire vil den beste representasjonen av π, $1/7$ og $100003/17$ være

$$\begin{aligned} \pi &\approx 3.142, \\ \frac{1}{7} &\approx 0.1429, \\ \frac{100003}{17} &\approx 5883, \end{aligned}$$

i følge de vanlige avrundingsreglene. Disse tallene kan vi greit runde av til fire siffer, men hva med større tall, for eksempel 58830? Vi ser at når tallene blir større enn eller lik 10000 så trenger vi fem siffer for å skrive dem, selv om vi egentlig bare er interessert i de

fire første sifrene. Poenget er at vi må vite størrelsen på tallet, altså hvor desimalpunktet skal være i forhold til de fire sifrene. Og her kommer den andre begrensningen inn; vi må sette en grense for hvor langt desimalpunktet kan være fra de fire sifrene. En grei måte å ordne dette på er å skrive tallene på en standardisert form, og det viser seg at formatet

$$\begin{aligned} 3.142 &= 0.3142 \cdot 10^1, \\ 0.1429 &= 0.1429 \cdot 10^0, \\ 5883 &= 0.5883 \cdot 10^4, \\ 0.003581 &= 0.3581 \cdot 10^{-3} \end{aligned}$$

er hendig. Hvis det gitte tallet er a og er ulik 0, omskriver vi det altså som

$$a = b \cdot 10^n, \tag{2.3}$$

der $|b|$ ligger i intervallet

$$0.1 \leq |b| < 1, \tag{2.4}$$

og n er heltallet som gjør at denne likheten holder (legg merke til at det bare er en verdi av n som gjør likheten (2.3) med begrensningen (2.4) gyldig—vi sier at n er *entydig* bestemt). Med andre ord er 10^{-n} den potensen av ti som vi må multiplisere a med for at resultatet b skal bli et tall der første siffer står rett til høyre for desimalpunktum. Tallet b kalles ofte for *mantissen* mens n kalles *eksponenten* til a, og høyresiden av (2.3) kalles den *normaliserte formen* av a.

Når a er omskrevet som i (2.3) er det lett å gjøre de to begrensningene som må til for at a kan representeres med endelige ressurser: i tillegg til å begrense antall siffer i b må vi også begrense antall siffer i n, og dermed begrense hvor langt desimalpunktet er fra de sifrene vi bruker for å skrive a.

For å illustrere representasjon og aritmetikk med flyttall, skal vi i resten av denne seksjonen, som over, bruke en tallmodell der vi begrenser antall siffer i b til fire samt et fortegn, mens n begrenses til å ha ett siffer samt et fortegn. Da vil det største og det minste positive tallet vi kan representere være de to tallene

$$\begin{aligned} 0.9999 \cdot 10^9 &= 999900000, \\ 0.1000 \cdot 10^{-9} &= 0.0000000001. \end{aligned} \tag{2.5}$$

Det minste og det største negative tallet som kan representeres med firesifret mantisse og ensifret eksponent er tallene i (2.5) med motsatt fortegn,

$$\begin{aligned} -0.9999 \cdot 10^9 &= -999900000, \\ -0.1000 \cdot 10^{-9} &= -0.0000000001. \end{aligned}$$

Tilfellet $a = 0$ må behandles spesielt. I dette tilfellet må vi ha $b = 0$, mens n kan være hva som helst. Det er da vanlig å bruke $n = 0$, slik at vi omskriver 0 som

$$0 = 0.0000 \cdot 10^0.$$

Valget vi gjorde med å bruke fire siffer til mantissen og ett siffer til eksponten var tilfeldig; et mer realistisk valg ville for eksempel være å avsette ti siffer til mantissen og to siffer til eksponenten. I tillegg må vi også ta vare på fortegnene til mantissen og eksponenten.

Eksponenten forteller oss størrelsen på tallet (i tallverdi). Hvis eksponenten er stor er tallet enten et stort positivt tall eller et negativt tall med stor tallverdi. Hvis eksponenten er liten, nærmere bestemt et negativt tall med stor tallverdi (for eksempel -9 når ett siffer er avsatt til eksponenten) er tallet lite i den forstand at det ligger nær null (tallverdien er liten).

Mantissen inneholder sifrene lengst til venstre i tallet, uavhengig av tallets størrelse. For eksempel kan en mantisse på 0.14 angi tallet 14000, tallet 14, tallet 0.0014 og mange andre, alt avhengig av størrelsen på eksponenten.

2.3.2 Addisjon og subtraksjon av flyttall

Når vi har funnet fram til en standardisert måte for å representere reelle tall som flyttall er det ikke så vanskelig å tenke seg hvordan aritmetikk med flyttall foregår. La oss se på noen eksempler der vi, som over, bruker flyttall med fire desimale siffer (pluss fortegn) til mantissen og ett desimalt siffer (pluss fortegn) til eksponenten. For å addere de to tallene 2.35 og 4.64 skrives de først på normalisert form,

$$2.35 = 0.2350 \cdot 10^1,$$
$$4.64 = 0.4640 \cdot 10^1.$$

Deretter adderes de to mantissene, noe som gir svaret

$$0.699 \cdot 10^1,$$

og dette gjenkjenner vi som tallet 6.99.

Et litt mer komplisert eksempel er $4.78 + 6.01$. På normalisert form er disse tallene

$$4.78 = 0.4780 \cdot 10^1,$$
$$6.01 = 0.6010 \cdot 10^1.$$

Adderer vi nå de to mantissene får vi svaret

$$1.079 \cdot 10^1.$$

Selv om de to leddene i summen var på normalisert form er svaret altså ikke på normalisert form siden mantissen er større enn en. Dette kommer av at vi fikk 'en i mente´ da vi adderte de to sifrene 4 og 6 lengst til venstre i mantissen. Addisjonsalgoritmen vil kompensere for dette ved å flytte desimalpunktet en plass til venstre og dermed få svaret over på normalisert form,

$$1.079 \cdot 10^1 = 0.1079 \cdot 10^2.$$

I vanlig notasjon gjenkjenner vi tallet som 10.79

Legg merke til at vi i det siste eksemplet begynte med to tresifrede tall og endte opp med et firesifret tall. I det neste eksempelet øker antall siffer fra fire til fem. Vi adderer de to tallene 0.5645 og 0.7821 som på normalisert form er

$$0.5645 = 0.5645 \cdot 10^0,$$
$$0.7821 = 0.7821 \cdot 10^0.$$

Adderer vi mantissene og konverterer til normalisert form blir svaret

$$0.13466 \cdot 10^1.$$

Denne mantissen består av mer enn fire siffer og kan derfor ikke representeres med de flyttallene vi opererer med. I en slik situasjon vil maskinen runde av[3] svaret til

$$0.1347 \cdot 10^1.$$

Så langt har addisjonene vi har sett på enten bevart eller økt antall siffer i mantissen. Ved subtraksjon kan antall siffer reduseres, og dette kan gjøre det problematisk å vurdere nøyaktigheten i en sekvens av beregninger. La oss først se på et eksempel som ikke skaper alvorlige problemer. For å utføre subtraksjonen $1.438 - 1.113$ gjør vi som ved addisjon og omskriver først tallene til normalisert form

$$1.438 = 0.1438 \cdot 10^1,$$
$$1.113 = 0.1113 \cdot 10^1.$$

Subtraksjon av de to mantissene gir tallet

$$0.0325 \cdot 10^1.$$

Som vi ser er dette tallet ikke på normalisert form siden mantissen er mindre enn 0.1. Vi kan konvertere til normalisert form ved å flytte desimalpunktet en plass mot høyre,

$$0.0325 = 0.3250 \cdot 10^0.$$

Vi ser at dersom vi skal skrive svaret med fire siffer (som maskinen vil gjøre i sin interne representasjon) må vi ta med en null lengst til høyre. I dette tilfellet er det greit å legge til en null, men i andre sammenhenger kan dette skape problemer. Et eksempel vil illustrere hva som kan skje.

La oss regne ut uttrykket $x - \sin x$ når $x = 1/12$, målt i radianer. På normalisert form har vi

$$x = 1/12 \approx 0.8333 \cdot 10^{-1},$$
$$\sin x \approx 0.8324 \cdot 10^{-1}.$$

[3]Enkelte 'sære´ maskiner vil kunne trunkere svaret til $0.1346 \cdot 10^1$, altså bare stryke det femte sifferet. Dette er ikke å anbefale siden det gir en større feil enn avrunding.

Subtraherer vi på samme måte som over får vi derfor det normaliserte tallet

$$x - \sin x \approx 0.9000 \cdot 10^{-4}.$$

Men det riktige svaret, skrevet på normalisert form med fire siffer, er $0.9642 \cdot 10^{-4}$. Vi ser altså at bare det første av de fire sifrene er riktige. Problemet er at maskinen har lagt til ekstra nuller slik som over, men i motsetning til det tidligere eksempelet er det de tre sifrene 6, 4 og 2 vi skal legge til nå, og ikke tre nuller.

Så langt har tallene vi har regnet med vært omtrent like store. Dersom dette ikke er tilfelle må addisjonsalgoritmen justeres noe. Ved addisjon av tallene $10 = 0.1000 \cdot 10^2$ og $0.1 = 0.1000 \cdot 10^0$ kan vi ikke bare addere mantissene til de to tallene siden de har forskjellige eksponenter. I slike tilfeller omskrives det minste tallet slik at det får samme eksponent som det største før mantissene adderes,

$$10 = 0.1000 \cdot 10^2,$$
$$0.1 = 0.0010 \cdot 10^2.$$

Svaret blir $0.1010 \cdot 10^2$ som vi gjenkjenner som 10.1.

Når forskjellen mellom de to tallene ikke er så stor går dette bra, men det er ikke vanskelig å finne problematiske situasjoner. La oss se på addisjonen $1000 + 0.001$. På normalisert form er de to tallene $1000 = 0.1000 \cdot 10^4$ og $0.001 = 0.1000 \cdot 10^{-2}$. For å utføre addisjonen må det minste tallet skrives som et tall med 4 som eksponent,

$$0.001 = 0.1000 \cdot 10^{-2} = 0.0000001 \cdot 10^4.$$

Men husk at vi bare har fire siffer tilgjengelig for mantissen, så før addisjonen kan utføres må dette tallet avrundes til et tall med fire siffers mantisse og eksponent 4. Denne avrundingen gir $0.001 \approx 0.0000 \cdot 10^4$. Addisjonen vi faktisk ender opp med blir derfor

$$0.1000 \cdot 10^4 + 0.0000 \cdot 10^4 = 0.1000 \cdot 10^4 = 1000.$$

På en maskin med fire siffers mantisse og ett siffers eksponent vil altså addisjonen $1000 + 0.001$ gi svaret 1000 i steden for det riktige 1000.001. Dette er forsåvidt ikke dramatisk siden svaret er riktig avrundet til fire siffer, men som vi skal se under er dette fenomenet noe vi må ta hensyn til i visse situasjoner når vi programmerer.

Et problem som kan oppstå ved addisjon eller subtraksjon er at svaret blir for stort (i tallverdi) til å kunne representeres som et flyttall. Et eksempel innenfor vår flyttallsmodell er addisjonen $0.8 \cdot 10^9 + 0.4 \cdot 10^9$. På normalisert form blir svaret $0.12 \cdot 10^{10}$—et for stort tall siden vi trenger 2 siffer til eksponenten. Dersom det i en sekvens av beregninger på en datamaskin forekommer et tall med for stor tallverdi vil dette tallet på de fleste moderne datamaskiner få verdien `Positive_Infinity` eller `Negative_Infinity`.

2.3.3 Multiplikasjon og divisjon med flyttall

Multiplikasjon og divisjon av flyttall på normalisert form utføres som forventet. Ved multiplikasjon multipliseres mantissene sammen mens eksponentene adderes, og hvis mantissen

ble mindre enn 0.1 (i tallverdi) vil svaret justeres til normalisert form. Divisjon foregår på lignende måte. Mantissene divideres og eksponentene subtraheres, og om nødvendig justeres resultatet til normalisert form. Som vi så over kan addisjon av flyttall skape store problemer fordi vi kan miste siffer. Slikt kan ikke skje ved multiplikasjon og divisjon så vi går ikke inn på detaljer i algoritmene her.

Ved multiplikasjon og divisjon kan det lett skje at et tall blir for stort eller for lite, og som nevnt over fører dette til at variabelen som skal ha det endelige resultatet får verdien `Infinity` eller `-Infinity`. Men ved multiplikasjon og divisjon kan det også skje at et tall blir mindre (i tallverdi) enn det minste tallet (i tallverdi) som kan representeres på maskinen. I modellen som vi har brukt over skjer for eksempel dette dersom vi multipliserer $0.1 \cdot 10^{-6}$ med seg selv. Det riktige svaret er da $0.1 \cdot 10^{-13}$, og for å representere dette tallet trenger vi to siffer i eksponenten, samt fortegnet. En slik situasjon vil ikke gi noe feilmelding, men svaret vil bli avrundet til 0.0.

Ved divisjon kan en annen feilsituasjon oppstå, nemlig divisjon med 0. Dette er en udefinert situasjon rent matematisk, men dersom vi dividerer et tall som er forskjellig fra null med en følge av tall som nærmer seg null, vil svarene bli større og større i tallverdi. Siden verdiene `Infinity` og `-Infinity` er definert i Java er det derfor rimelig at en flyttallsoperasjon som 7.0/0.0 gir resultatet `Infinity` og at operasjonen $-5.0/0.0$ gir verdien `-Infinity`. Operasjonen 0.0/0.0 kan derimot ikke gis en rimelig verdi, det er en undefinert operasjon, og dette indikeres ved at resultatet blir `NaN` som er en forkortelse for Not a Number. Verdien `NaN` gir alltid `NaN` uansett hva den kombineres med og vil derfor lett infisere en beregningskjede fullstendig hvis den først dukker opp. På denne måten er det lett å oppdage at noe har gått galt. Andre udefinerte, matematiske operasjoner som for eksempel kvadratroten av et negativt tall, vil også gi `NaN` som resultat.

2.3.4 Binær representasjon av flyttall

Da vi diskuterte representasjon av heltall argumenterte vi for å bruke to-tallsystemet siden dette var mer robust i forhold til støy. Det samme gjelder selvsagt ved representasjon av reelle tall. Representasjonen som brukes i de fleste datamaskiner er derfor en totallsversjon av representasjonen over. Hvis a er tallet vi skal representere omskriver vi det som

$$a = c \cdot 2^m, \tag{2.6}$$

der c ligger i området $0.5 \leq c < 1$ hvis a er positiv og $-1 < c \leq -0.5$ hvis c er negativ, mens m er den heltallige eksponenten som gjør at de to sidene av likhetstegnet blir like. Hvis $a = 0$ er både c og m null. Vi setter så av det ønskede antallet binære siffer til representasjon av c og m. De aritmetiske operasjonene utføres så i to-tallssystemet, men de sammme problemene som vi så over (i ti-tallssystemet) vil kunne forekomme. Spesielt må det understrekes at vi ikke kommer unna problemet med avrundingsfeil.

Dagens datamaskiner følger stort sett en internasjonal standard for flyttallsaritmetikk, ofte omtalt som IEEE-standarden.[4] Maskinene har mulighet for å representere flyttall

[4]IEEE er forkortelse for *Institute of Electrical and Electronic Engineers* som er en stor forening for ingeniører i USA. Flyttallsstandarden er beskrevet i det som kalles *IEEE standard reference 754*.

med enten 32 eller 64 binære siffer (bits) (noen maskiner kan også arbeide med 80 eller 128 bits flyttall), og i Java kalles disse flyttallstypene `float` og `double`. For flyttall av typen `float` er det avsatt 24 binære siffer til mantissen og 8 binære siffer til eksponenten, begge inklusive fortegn. Dette betyr at eksponenten må være et heltall i området fra -127 til 128. Det største positive tallet som kan representeres med typen `float` er $3.4028235 \cdot 10^{38}$ mens det minste positive tallet av typen `float` er $1.4 \cdot 10^{-45}$. Disse tallene framkommer ikke direkte fra en binær versjon av den desimale modellen vi så på over, en del tekniske detaljer kommer i tillegg.

Variabeltypen `double` har 64 binære siffer, og av disse er 53 avsatt til mantissen, mens de resterende 11 er avsatt til eksponenten. Dette betyr at det største postive tallet som kan representeres med denne datatypen er $1.7976931348623157 \cdot 10^{308}$ mens det minste positive tallet er $4.9 \cdot 10^{-324}$. En mantisse på 53 binære siffer svarer til omtrent 15 desimale siffer. Med andre ord kan vi altså si at vi arbeider med 15 desimale siffer når vi bruker 64 bits flyttall.

Java har to klasser `Float` og `Double` som inneholder en del nyttige konstanter og funksjoner for de to flyttallstypene `float` og `double`. Blant annet inneholder disse klassene konstanter som gir største og minste positive tall for de respektive variabeltypene og konstantene `NaN`, `Negative_Infinity` og `Positive_Infinity`.

Siden moderne datamaskiner regner omtrent like raskt med 64 bits flyttall som med 32 bits, er det nå vanlig å bruke `double` variable i de fleste sammenhenger. Dette gir større nøyaktighet uten ekstra kostnad. Det største problemet med 64 bits flyttall er at de tar opp dobbelt så stor plass når de skal lagres.

2.3.5 Programmering med flyttall

For mange har den grunnleggende kunnskapen om flyttall kommet som resultat av dyrekjøpt erfaring etter flere dagers feilsøking i et dataprogram som oppfører seg helt uforståelig. Når vi programmerer gjør vi ofte, eksplisitt eller implisitt, bruk av tallenes matematiske egenskaper, men det er altså ikke alltid disse gjelder for flyttall. Informasjonen vi nå har bør gjøre det lettere å gjennomskue feil relatert til flyttall, og følgende enkle tips bør kunne redusere antall feil betraktelig.

Bruk av heltall. I forhold til flyttall har heltall den store fordelen at de er helt nøyaktige. Dersom det i en beregning er tall som bare antar heltallige verdier bør slike tall derfor ikke legges i flyttallsvariable. På samme måte kan beregninger ofte involvere enkle reelle tall som raskt kan regnes ut fra heltall. For eksempel er det ofte nødvendig å la en variabel x anta reelle verdier med fast avstand, la oss si

$$x = 0.0, \quad x = 0.1, \quad x = 0.2, \quad \ldots, \quad x = 0.9, \quad x = 1.0.$$

En måte å implementere dette på er å la x starte med verdien 0.0 og så øke x med 0.1 etterhvert som beregningene utvikler seg. Problemet med dette er at vi da gjør en liten avrundingsfeil hver gang x økes, og en test på om x er lik 1.0 ved slutten av beregningene vil kunne gi negativt svar. En bedre måte å gjøre disse beregningene på er å la en heltallsvariabel i anta verdiene 0, 1, 2, ..., 9, 10 og så regne ut x som $x = 0.1 * i$. Da får

vi fremdeles en liten avrundingsfeil i multiplikasjonen, men ingen akkumulert (oppsamlet) avrundingsfeil. Dessuten kan vi gjøre den problematiske sammenligningen av x med 0.1 ved i steden å sjekke om i har blitt lik 10.

Antall siffer i konstanter. Selv om vi regner aldri så nøyaktig, vil vanligvis ikke resultatene vi kommer fram til være mer nøyaktig enn de mest unøyaktige tallene vi baserer beregningene på (som vi har sett vil sluttresultatet ofte være enda mindre nøyaktig). Det er derfor viktig å passe på at våre inngangsdata har tilstrekkelig nøyaktighet. Mange tall som for eksempel måledata har vi ingen kontroll over, men mange beregninger er også basert på matematiske konstanter og disse må vi passe på at blir angitt med tilstrekkelig nøyaktighet. Regner vi med 32 bits flyttall bør derfor konstanter angis med 8 siffer, mens de bør angis med 17-18 siffer hvis vi regner med 64 bits flyttall. Mange språk har de vanligste tallene som π og e tilgjengelige som predefinerte konstanter. Ellers er det en god regel å regne ut aktuelle konstanter i programmet i steden for å taste dem inn når det er mulig, siden de innebygde matematiske funksjonene (trigonometriske funksjoner, logaritmer og eksponensialfunksjoner, rotutdragning) vanligvis gir maksimal nøyaktighet for den aktuelle datatypen.

Håndtering av `NaN`. Som nevnt over gir maskinen beskjed om at noe tvilsomt har skjedd under flyttallsberegninger ved å gi svaret `NaN`. Den vanligste grunnen til dette er at det et eller annet sted har blitt en divisjon med 0. Noen ganger vil bare en eller noen få variable være infisert av `NaN`, og de andre verdiene som skrives ut være riktige, men uansett er dette et tegn på at noe er grunnleggende galt i programmet og det er på tide med feilsøking.

Sammenligning av flyttall. En viktig del av numeriske beregninger er tester på likhet mellom tall. Som et eksempel kan vi tenke oss at vi skal skrive et program for å finne nullpunkter for en funksjon f. Hvis vi har et tall x som vi tror er nullpunktet, må vi teste om $f(x)$ er null. Men ut fra hva vi har sett over vil avrundingsfeil gjøre at vi neppe kan håpe på å finne et nullpunkt helt nøyaktig og da vil heller ikke $f(x)$ bli null. I steden bør vi teste på om tallverdien til $f(x)$ er 'liten'. På grunn av problemene med avrundingsfeil gjelder dette generelt: test aldri om to flyttall er like, test i steden om differansen i tallverdi er nær null. En annen variant av samme fenomen er at når a og b er flyttall som begge er forskjellige fra null kan det allikevel godt hende at $a + b = a$, noe som er umulig for reelle tall, se eksempelet over der vi adderte 1000 og 0.001.

Sammenligning av flyttall kan bli svært mystisk dersom et av tallene som sammenlignes ved en feil har blitt til `NaN`. Det er nemlig slik at et logisk uttrykk som involverer `NaN` alltid gir verdien `false`, selv uttrykket `NaN == NaN` er `false`! Måten å teste om en variabel `a` av typen `double` i Java inneholder `NaN` er

```
if (Double.isNaN(a)) . . . // Hvis dette slår til har vi NaN
```

Avrundingsfeil. Som vi har sett kan antall riktige siffer i flyttall variere. Selv om utgangspunktet for beregningene var tall der alle sifrene var riktige kan vi underveis få

kansellering og avrundingsfeil som gjør at langt fra alle siffer i resultatet trenger være riktige. Det er ofte vanskelig å avgjøre om en metode kan gi store avrundingsfeil, men et godt tips er å teste koden med enkle data der resultatet er kjent. På den måten kan man få et godt inntrykk av følsomheten for avrundingsfeil. Det aller beste er selvsagt å gjennomføre en matematisk analyse av beregningene og gi nøyaktige estimater for avrundingsfeilen, men dette er ofte svært krevende eller umulig i praksis.

2.3.6 Flyttall med vilkårlig presisjon

Problemene med flyttall kommer av de to grunnleggende begrensningene at antall siffer i både mantisse og eksponent er begrenset. Ved å øke antall siffer blir problemene umiddelbart mindre framtredende. For eksempel er det mindre sannsynlig å få flyttallsrelaterte problemer med 64 bits flyttal en med 32 bits flyttall. På den annen side er det klart at de grunnleggende problemene ikke forsvinner selv om vi øker antall siffer; selv om antall siffer har økt er antallet fremdeles begrenset. På de aller fleste datamaskiner fins det bare støtte i elektronikken for å regne med 32 eller 64 bits flyttall, men man kan selvsagt selv skrive programmer som håndterer flyttall med flere siffer. Dette er gjort i de store matematikkprogrammene Maple og Mathematica, og i Java håndterer klassen `java.math.BigDecimal` flyttall med vilkårlig mange siffer. Selv om det kan være imponerende å kunne regne ut π med en million siffer er det viktig å være klar over at begrensningene med flyttall stadig er til stede. Dessuten setter selvsagt de tilgjengelige maskinressursene (hastighet og hukommelse) en øvre grense for det antall siffer som kan håndteres i praksis.

2.3.7 Regnehastighet

Det er vanskelig å si noe presist om regnehastigheten på en datamaskin siden den endrer seg raskt, i takt med teknologiutviklingen, og varierer mellom maskintypene. Tommelfingerregelen er at regnehastigheten dobles hver 18. måned.[5] Dette har grovt sett holdt stikk i mange år nå, men kan selvsagt ikke vare evig.

Den viktigste faktoren som påvirker regnehastigheten er *klokkefrekvensen* i maskinen. Litt forenklet angir denne hvor fort klokka i maskinen 'tikker', og poenget er at de enkleste regneenhetene i maskinen vanligvis klarer å levere et resultat hver gang klokka 'tikker'. Dette betyr at maskinen kan levere resultatet av en addisjon eller multiplikasjon ved hvert tikk, mens divisjon lett tar 3–4 tikk. Nå er dette bare riktig hvis maskinen får lov til å utføre en lang rekke med like operasjoner, for eksempel addisjoner eller multiplikasjoner. I praksis må den ofte veksle mye på hva som skal gjøres slik at den optimale hastigheten ikke oppnås.

Klokkefrekvensen (antall tikk i sekundet) på dagens (år 2000) PC'er ligger fra ca. 500 MHz til ca. 1 GHz, altså mellom 500 000 000 og 1 000 000 000 tikk i sekundet. Dette betyr at maskinene under optimale forhold kan levere et sted mellom en halv milliard og en milliard resultater pr. sekund (en milliard flyttallsoperasjoner pr. sekund refereres ofte til som en gigaflop). Det må understrekes at her er det variasjoner, og det fins 'småfinesser'

[5] Dette kalles Moores lov etter Gordon Moore. Han var en av grunnleggerne av selskapet Intel som produserer PC-brikker, og han kom med denne spådommen i 1965. Nå er det stadig flere forskere som uttaler at det kan bli vanskelig å leve opp til Moores lov selv om den har holdt ganske godt fram til nå.

i mange maskiner som kan gjøre at de under optimale forhold kan levere noe mer enn et resultat pr. tikk.

En måte å øke regnehastigheten er å øke klokkefrekvensen i maskinen, men dette er krevende rent elektronisk. Den andre muligheten er å hekte mange maskiner sammen og la dem arbeide sammen på samme problem, vi sier at de arbeider i *parallel*. Dersom vi setter 10 maskiner til å arbeide på et problem skulle en tro at det ville kunne løses på en tiendedel av tiden, men dette er bare unntaksvis tilfelle, mer realistisk kan vi kanskje få redusert tiden ned til 20 eller 30 % av den tiden en enslig slik maskin bruker. Dette kommer av at et problem sjelden består av 10 uavhengige småproblemer som kan løses hver for seg. Som regel er det derfor slett ikke opplagt hvordan problemet kan deles opp i mindre problemer som kan fordeles til de forskjellige maskinene, og vanligvis kommer vi ikke unna at noen maskiner må bruke en del av tiden til å vente på at andre maskiner har regnet ferdig. Dette gjør parallelisering utfordrende og til et svært aktivt forskningsområde i miljøer som har stort behov for regnekraft.

2.3.8 Valg av variabelnavn

Ved valg av variabelnavn fins det noen uskrevne regler i matematikk som delvis har blitt arvet i programmering, særlig i eldre språk. I matematikk brukes ofte bokstavene i, j, k, l, m og n om heltall, for eksempel som indekser i summasjoner som i $\sum_{i=1}^{n} i^2$. Bokstavene x, y og z brukes ofte om reelle variable som i $\sin x$, mens f, g og h ofte brukes som funksjonsnavn. I tillegg fins det mindre utbredte konvensjoner for bruk av store og små bokstaver og greske bokstaver.

I matematikk er det uvanlig å bruke variable som består av mer enn en bokstav. I programmering bruker vi som regel lengre variabelnavn, men særlig i matematisk programmering har det vært vanlig å bruke konvensjonene fra matematikk og la heltallsvariable begynne med en av bokstavene i, j, k, l, m eller n. Dette er grunnen til at det ikke er uvanlig å finne et variabelnavn som `iant` i et program skrevet av en nordmann. Dette vil typisk betegne en variabel som inneholder et antall, og i´en er der fordi dette er et heltall. I dag er det vanlig (og god) praksis i programmering å la variabelnavn reflektere hva variabelen betegner, og dette gjelder selvsagt også i matematisk programmering. På grunn av konvensjonene i matematikk betyr dette at `i` og `k1` ofte betegner heltall, mens `x` og `z3` ofte betegner reelle tall i et matematisk inspirert program.

2.4 Absolutt og relativ feil

Som vi har sett er det ikke til å unngå at vi får avrundingsfeil når vi arbeider med flyttall, og det er derfor viktig å ha en viss kontroll på denne feilen. Men for å kunne ha kontroll på feilen må vi aller først vite hvordan vi skal måle feilen, og det er tema for denne seksjonen.

For å måle feil trenger vi ikke skille mellom flyttall og reelle tall, så vi tenker oss at vi har et reelt tall a og en tilnærming $\tilde{a}$ til a. Den opplagte måten å definere feilen i denne tilnærmingen er ved den *absolutte feilen*, som er gitt ved

$$|a - \tilde{a}|. \tag{2.7}$$

Her har vi satt på tallverditegn siden vi ikke bryr oss med om feilen er positiv eller negativ. Hvis for eksempel $a = 1$ og $\tilde{a} = 1.001$ ser vi at den absolutte feilen er $|-0.001| = 0.001$.

I mange tilfeller er den absolutte feilen en god måte å måle feil på, men enkelte ganger gir den oss ikke den informasjonen vi er ute etter. Hvis vi har tallet $b = 1000$ og en tilnærming $\tilde{b} = 1000.001$ så er igjen den absolutte feilen 0.001. Med denne tolkningen av feil er altså $\tilde{b}$ en like god tilnærming til b som det $\tilde{a}$ er til a. Men hvis vi ser på antall siffer så inneholder $\tilde{a}$ bare tre riktige siffer mens $\tilde{b}$ inneholder 6 riktige siffer. Utfra en slik betraktning er det derfor ikke urimelig å si at $\tilde{b}$ er en mye bedre tilnærming til b enn det $\tilde{a}$ er til a. For å få fram denne forskjellen definerer vi den *relative feilen* ved

$$\frac{|a - \tilde{a}|}{|a|}. \tag{2.8}$$

når a er forskjellig fra 0 (når $a = 0$ er ikke den relative feilen definert). Den relative feilen framkommer altså ved å skalere den absolutte feilen med det tallet vi betrakter som det 'riktige'. Med andre ord angir den relative feilen hvor stor andel den absolutte feilen utgjør av det 'riktige' tallet.

Vi kan nå regne ut den relative feilen i $\tilde{a}$ og $\tilde{b}$ og finner

$$\frac{|a - \tilde{a}|}{|a|} = 1.0 \cdot 10^{-3}, \qquad \frac{|b - \tilde{b}|}{|b|} = 1.0 \cdot 10^{-6}.$$

I vårt tilfelle er derfor den relative feilen i $\tilde{b}$ mye mindre enn den relative feilen i $\tilde{a}$ og vi skal se senere at størrelsen på den relative feilen er nært knyttet til antall riktige siffer i tilnærmingen vår.

En god analogi til absolutt og relativ feil er hvordan vi måler avkastning av penger. Anta at vi har kr. 2430 i banken ved inngangen til ett år. Vi rører ikke pengene, og når året er omme har pengene vokst til kr. 2551.50. Vi ser at vi har hatt en avkastning på kr. 121.50 siden $2551.50 - 2430 = 121.50$. Hvis vi tenker på $c = 2430$ som det opprinnelige beløpet og $\tilde{c} = 2551.50$ som en tilnærming til dette så svarer den absolutte feilen $|c - \tilde{c}| = 121.50$ nettopp til avkastningen i kroner. Hvis vi regner ut den relative feilen i dette tilfellet så får vi

$$\frac{|c - \tilde{c}|}{|c|} = \frac{121.50}{2430} = 0.05.$$

Vi tar altså avkastningen i kroner og dividerer med det beløpet vi begynte med. Vi finner dermed hvor stor andel avkastningen er av beløpet vi startet med og svaret er 0.05 eller 5%. Dette svarer selvsagt til at vi har hatt en rente på 5% det året vi har hatt pengene i banken. Dette enkle eksemplet illustrerer at absolutt feil kan sammenlignes med avkastning i kroner mens relativ feil svarer til avkastning i %, altså rentefoten.

2.4.1 Relativ feil angir antall riktige siffer

Hvis vi går tilbake til de to eksemplene over med a og b og deres tilnærminger, så ser vi at $\tilde{a}$, som inneholder 3 riktige siffer, har en relativ feil på 10^{-3}, mens $\tilde{b}$, som har 6 riktige siffer, har en relativ feil på 10^{-6}. Generelt er det slik at hvis den relative feilen i $\tilde{c}$ er

$$r = \frac{|c - \tilde{c}|}{|c|} = x \cdot 10^{-m} \qquad \text{med } 0.5 \leq |x| < 5, \tag{2.9}$$

så vil de m første sifrene i c og $\tilde{c}$ stemme overens. Dette er en tommelfingerregel som må tolkes litt romslig, men la oss forsøke oss på et litt upresist argument for at dette er omtrent riktig.

Hvis c er 1 er ikke dette så overraskende. I dette tilfellet er den relative og den absolutte feilen like, slik at hvis de m første sifrene i c og $\tilde{c}$ er like (ett siffer til venstre for desimalkomma og $m-1$ siffer til høyre), så er m'te desimal det første sifferet der de to tallene skiller seg. Men dette er åpenbart det samme som at $r = |c - \tilde{c}| = x \cdot 10^{-m}$ for en passende x som er omtrent 1, slik som i (2.9). Et eksempel er gitt ved a og $\tilde{a}$ over der $|a - \tilde{a}| = 0.001 = 1.0 \cdot 10^{-3}$.

Hvis c er nær 1 i den forstand at $0.5 \leq |c| < 5$, gjør vi ikke så stor feil om vi tilnærmer den relative feilen med den absolutte feilen og bruker argumentet over. Et slikt eksempel er gitt ved $c = 2.135$ og $\tilde{c} = 2.133$. Den absolutte feilen er $2 \cdot 10^{-3}$ og vi ser at de 3 første sifrene i c og $\tilde{c}$ stemmer overens.

I det generelle tilfellet der $c \neq 1$ skriver vi c på formen

$$c = d \cdot 10^{-n}, \qquad \text{med } 0.5 \leq |d| < 5.$$

Tallet d inneholder nøyaktig de samme sifrene som c, men skalert slik at $|d|$ er så nær 1 som mulig. Vi skalerer tilnærmingen $\tilde{c}$ på tilsvarende vis,

$$\tilde{c} = \tilde{d} \cdot 10^{-n},$$

med samme eksponent som c. Vi ser da at $|\tilde{c} - c| = |\tilde{d} - d| \cdot 10^{-n}$, slik at vi i uttrykket for den relative feilen r kan forkorte bort 10^{-n}. Dette gir

$$r = \frac{|c - \tilde{c}|}{|c|} = \frac{|d - \tilde{d}|}{|d|}.$$

Den relative feilen i $\tilde{c}$ er altså lik den relative feilen i $\tilde{d}$. Men siden $|d|$ er ganske nær 1, er vi tilbake i en situasjon der den absolutte feilen $|d - \tilde{d}|$ gir en god tilnærming til den relative feilen. Altså har vi

$$r \approx |d - \tilde{d}| = x \cdot 10^{-m}, \qquad \text{der } 0.5 \leq |x| < 5,$$

hvilket betyr at d og $\tilde{d}$ har omtrent m siffer felles. Siden c og $\tilde{c}$ er tall med nøyaktig samme siffer som d og $\tilde{d}$ må disse også ha omtrent m siffer felles.

Siden den relative feilen måler hvor mange siffer som er riktige egner den seg svært godt til å måle feilen i flyttallsberegninger som jo foregår med et fast antall siffer.

2.5 Noen ord om objektorientert programmering og matematikk

På laveste nivå opererer en datamaskin på binære siffer ved hjelp av et knippe tillatte operasjoner. For noen svært spesielle anvendelser gir dette tilstrekkelig funksjonalitet, men som regel er det nødvendig med mer komplekse datatyper. I vår sammenheng trenger vi særlig å kunne arbeide med tall, og elektronikken i maskinen gir god tilgang til å kunne arbeide både med heltall og reelle tall (i form av flyttall). I Java (som i de fleste

programmerimngsspråk) fins det datatyper for tall som er direkte tilpasset elektronikken i maskinen, for eksempel `int`, `long`, `float` og `double`, med sine tilhørende aritmetiske operasjoner og andre metoder knyttet til datatypen. Men matematikk handler ikke bare om heltall og reelle tall, det fins mange andre matematiske objekter som kan dra nytte av en datamaskins regneferdigheter.

Et opplagt eksempel er komplekse tall. I Java fins det ingen datatype for komplekse tall så en slik datatype må vi eventuelt implementere selv (eller finne fram til biblioteker som andre har skrevet). Java er et språk som er godt tilpasset *objektorientering*, og denne programmeringsmetodologien er godt egnet til å implementere matematiske objekter med sine metoder. For å kunne regne med komplekse tall på en ryddig måte er det derfor rimelig å definere en klasse `complex(x,y)` med passende metoder. Et objekt i denne klassen har da to parametre x og y som angir henholdsvis realdelen og imaginærdelen til tallet. For å kunne gjøre beregninger trenger vi metoder som implementerer aritmetiske operasjoner med komplekse tall, representasjon med polarkoordinater, evaluering av elementære funksjoner med komplekse argumenter og lignende. I tillegg trenger vi metoder for å kunne skrive ut og lese inn komplekse tall.

Matematikken er full av andre objekter med tilhørende metoder som kan implementeres naturlig i en objektorientert omgivelse. Nært beslektet med tallene har vi vektorer både i planet, rommet og flere dimensjoner. De enkleste metodene er da addisjon av vektorer og multiplikasjon av en vektor med en skalar (et tall). Vektorregning er en del av lineær algebra[6] der vektorer er et spesielt eksempel på et mer generelt objekt, nemlig matriser. Disse har sine tilhørende metoder som addisjon, multiplikasjon med skalar, multiplikasjon av matriser, invertering, og dermed divisjon av matriser også videre.

Et eksempel som ikke er en direkte generalisering av tallene er funksjoner. Som objekt er en funksjon f bestemt ved sin definisjonsmengde og en presis beskrivelse av hva den gjør med sitt argument x, altså hvordan den beregner resultatet $f(x)$. Metodene i en funksjonsklasse kan være så mangt siden det er svært mye vi kan gjøre med funksjoner. For eksempel kan vi addere, multiplisere og dividere funksjoner, og hvis vi er interessert i derivasjon er det naturlig å implementere derivasjonsreglene som metoder.

2.6 Konvertering fra desimaltall til binærtall

I dette seksjonen skal vi ta for oss hvordan tall kan konverteres fra desimal til binær form. Vi ser først på heltall og deretter på tall i intervallet $[0, 1]$.

2.6.1 Konvertering av heltall

Det fins en enkel metode for å konvertere fra desimal til binær representasjon som er basert på gjentatt divisjon med 2. Metoden finner de binære sifrene i rekkefølge fra høyre mot venstre, vi sier at vi finner de minst signifikante sifrene først. I presentasjonen under bruker vi operatoren // som angir heltallsdivisjon uten rest. Med andre ord er $5//2 = 2$ og $11//3 = 3$.

[6] Lineær algebra er en viktig del av grunnutdanningen i matematikk og står sentralt i de to neste grunnkursene (MAT 1110 og MAT 1120).

Hvis tallet vi skal konvertere er 23, begynner vi med å regne ut $23//2 = 11$ som gir rest 1. Vi gjentar dette med 11 og får $11//2 = 5$ og rest 1. Vi gjentar igjen og får $5//2 = 2$ og rest 1. Neste gang får vi $2//2 = 1$ og rest 0. Endelig får vi $1//2 = 0$ og rest 1. Litt mer systematisk kan vi sette opp dette som

23	1
11	1
5	1
2	0
1	1

Et tall i høyre kolonne er resten når vi dividerer tallet rett til venstre med 2, mens et tall i venstre kolonne er resultatet når tallet rett over divideres med 2, uten å ta vare på resten (det første tallet er selvsagt tallet vi skal konvertere). De binære sifrene til 23 kan nå avleses i høyre kolonne, med det minst signifikante sifferet øverst. Vi har altså

$$23 = 10111_2$$

som det er lett å sjekke at stemmer.

La oss ta et eksempel til og konvertere $a = 349$ til binær form. Vi får nå

349	1
174	0
87	1
43	1
21	1
10	0
5	1
2	0
1	1

og har derfor at $349 = 101011101_2$.

2.6.2 Hvorfor er denne metoden riktig?

Dette ser jo ut til å være en enkel og grei metode for å konvertere et tall til binær form, men hvorfor gir den riktig resultat? For å se det må vi ta med litt mer detaljer. La oss ta for oss det første eksempelet. Vi ønsker å finne tall $b_0, \ldots, b_4$ som alle skal være 0 eller 1 slik at

$$23 = b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0. \tag{2.10}$$

Grunnen til at vi ikke tar med flere ledd på høyre side er at det neste leddet ville være $b_5 2^5$ (de andre leddene vil inneholde enda høyere potenser av 2), og siden $2^5 = 32 > 23$ vil vi ikke trenge disse høyereordens leddene i dette tilfellet. I (2.10) legger vi merke til at de fire første leddene på høyre side alle er partall, mens det siste leddet $b_0 2^0 = b_0$ enten

er 0 eller 1. Nå er 23 et oddetall så hvis vi ikke skal komme skjevt ut allerede i starten må vi sette $b_0 = 1$ slik at også høyresiden blir et oddetall.

Vi fortsetter med å dividere begge sider av (2.10) med 2, uten å ta vare på resten. Dette gir

$$23//2 = 11 = b_4 2^3 + b_3 2^2 + b_2 2^1 + b_1 2^0. \tag{2.11}$$

Vi har altså fått et nytt problem av samme type som vi startet med, men tallet vårt er redusert i størrelse fra 23 til 11 og antall binære sifre er redusert fra 5 til 4. Vi kan da bruke samme resonnement som over og konkludere med at vi må ha $b_1 = 1$ siden 11 er et oddetall. Vi dividerer med 2 igjen og får

$$11//2 = 5 = b_4 2^2 + b_3 2^1 + b_2 2^0. \tag{2.12}$$

Nok en gang er konkulsjonen at $b_2 = 1$ siden 5 er et oddetall. En ny divisjon gir

$$5//2 = 2 = b_4 2^1 + b_3 2^0 \tag{2.13}$$

som gir $b_3 = 0$ siden venstresiden nå er et partall. En siste divisjon gir

$$2//2 = 1 = b_4 2^0 \tag{2.14}$$

så $b_4 = 1$. Vi har altså $23 = 10111_2$ slik vi fant over.

Vår opprinnelige konvertering av 23 til binær form er ikke noe mer enn en litt mer kompakt måte å skrive opp dette på, og metoden er såpass enkel at et slikt omstendelig eksempel bør være nok til å overbevise oss om at metoden er riktig. I oppgavene under skal du finne fram til en tilsvarende metode for å konvertere heltall til tretallsystemet.

2.6.3 Konvertering av tall i intervallet $[0, 1]$

Et positivt, reelt tall x kan alltid skrives på formen $x = n + y$ der $n \geq 0$ er et heltall og y er et reelt tall i intervallet $[0, 1)$. Vi vet allerede hvordan n kan konverteres til binær form — nå skal vi se hvordan y kan konverteres til binær form. Legg merke til at det er enkelt å konvertere negative tall når vi vet hvordan positive tall skal konverteres: Vi konverterer det positive tallet med samme tallverdi og setter $-$ foran både n og y.

Anta at vi skal konvertere desimaltallet 0.65625 til binær form. Vi ønsker altså å skrive dette tallet som

$$0.6875 = b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + b_4 2^{-4} + \cdots$$

der hver b_i enten er 0 eller 1. Merk at vi generelt må ha uendelig mange siffer for å få likhet, akkurat som i 10-tallssystemet. Hvis vi multipliserer begge sider av likheten med 2 får vi

$$1.375 = b_1 + b_2 2^{-1} + b_3 2^{-2} + b_4 2^{-3} + \cdots .$$

Siden summen av leddene med b_2, b_3 og b_4 aldri kan bli større enn 1 ser vi at vi må ha $b_1 = 1$. Vi trekker så fra 1 på begge sider og fortsetter med

$$0.375 = b_2 2^{-1} + b_3 2^{-2} + b_4 2^{-3} + \cdots .$$

Multiplikasjon med 2 gir

$$0.75 = b_2 + b_3 2^{-1} + b_4 2^{-2} \ + \cdots .$$

Dermed ser vi at $b_2 = 0$ for ellers vil høyresiden bli større enn 1. Siden $b_2 = 0$ trekker vi ikke fra noe i neste steg, men multipliserer bare med 2 en gang til,

$$1.5 = b_3 + b_4 2^{-1} + \cdots .$$

Dermed ser vi at $b_3 = 1$. Vi trekker fra 1 og får

$$0.5 = b_4 2^{-1} \ + \cdots .$$

Multiplikasjon med 2 gir

$$1 = b_4 + \cdots .$$

Altså er $b_4 = 1$. Med dette har vi fått med oss hele det opprinnelige tallet siden vi nå får null når vi trekker fra 1. Altså er

$$0.6875 = 0.1011_2.$$

Vi kan føre dette på tilsvarende måte som når vi konverterer heltall,

0.6875	1
0.375	0
0.75	1
0.5	1

Et tall i høyre kolonne er 1 hvis tallet til venstre er større enn eller lik 0.5 og 0 hvis tallet er mindre enn 0.5. Hvis et tall x i venstre kolonne er mindre enn 0.5 vil tallet rett under være $2x$, hvis $x \geq 0.5$ vil tallet rett under være $2x - 1$. Det øverste tallet til venstre er tallet vi skal konvertere.

La oss prøve dette på tallet 0.45.
Her ser vi at algoritmen aldri stopper: tallene 0.8, 0.6, 0.2 og 0.4 vil gjenta seg i venstre kolonne og sifrene 1, 1, 0, 0 i høyre kolonne. Dette betyr at

$$0.45 = 0.011100110011001100\cdots_2$$

Det at algoritmen ikke stopper bør ikke overaske oss, vi kjenner jo det samme fenomenet fra 10-tallssystemet. Vi har for eksempel

$$\frac{1}{3} = 0.3333333333\cdots$$

$$\frac{3}{7} = 0.4285714285714\cdots$$

Dette kan få noen uventede konsekvenser når vi arbeider med flyttall, se oppgavene 3 og 14 (d).

0.45	0
0.9	1
0.8	1
0.6	1
0.2	0
0.4	0
0.8	1
0.6	1
0.2	0
0.4	0
0.8	1
⋮	⋮

Oppgaver

2.1 Programmer summene under og skriv ut resultatet. Husk at Java har en operator `+=` som kan være nyttig her.

a) $\sum_{i=1}^{5} i^2$.
b) $\sum_{i=1}^{500} 1/i$.
c) $\sum_{i=-15}^{15} i^4$.
d) $\sum_{i=6}^{20} 5$.
e) $\sum_{i=1}^{1000} \sin(i/50)$.

2.2 I matematikk brukes ofte en notasjon for produkter tilsvarende den for summer. Uttrykket $\prod_{i=1}^{6} i$ er definert ved

$$\prod_{i=2}^{6} i = 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6.$$

Symbolet $\prod$ er altså helt analogt med $\sum$ bortsett fra at plusstegnene blir endret til multiplikasjonstegn, slik at de aktuelle tallene multipliseres i steden for å adderes. Programmer og skriv ut resultatet av følgende produkter.

a) $\prod_{i=1}^{30} i$.
b) $\prod_{i=3}^{20} 2$.
c) $\prod_{i=1}^{10} i^3$.
d) $\prod_{i=5}^{100} 1/i$.
e) $\prod_{i=-4}^{4} i/(i+1)$.

2.3 Programmer for-løkka

```
double x;
for (x=0.0; x<= 2.0; x+=0.1)
  println("x er " + x);
```

Hvorfor blir aldri `x` lik 2.0?

2.4 Ut fra teksten er det klart at på en datamaskin vil addisjonen $1 + \epsilon$ bli regnet ut til 1 hvis bare ϵ er liten nok. Skriv et program som kan hjelpe deg til å finne det minste heltallet n slik at $1 + 2^{-n}$ blir 1. Gjør dette både for 32- og 64-bits flyttall (`float`- og `double`-variable i Java).

2.5 Binomialkoeffisientene $\binom{n}{i}$ er definert ved

$$\binom{n}{i} = \frac{n!}{i!\,(n-i)!} \tag{2.15}$$

der $n \geq 0$ er et heltall og i er heltall i intervallet $0 \leq i \leq n$. Binomialkoeffisientene forekommer i mange formler og må derfor derfor ofte beregnes på datamaskin, og siden alle binomialkoeffisienter er heltallige (divisjonen i (2.15) må altså alltid gå opp) er det rimelig å bruke heltallsvariable til slike beregninger. For små verdier av n og i går dette greit, men for litt større verdier får vi fort problemer fordi teller og nevner i (2.15) da kan bli større enn de største heltall som kan representeres på maskinen selv om binomialkoeffisienten i seg selv ikke er så stor. Ved å bruke flyttallsvariable i steden kan vi håndtere litt større tall, men igjen vil vi kunne få for store tall underveis selv om sluttresultatet ikke er for stort. I tillegg vil få avrundingsfeil når vi bruker flyttall.

Det som er uheldig med formelen (2.15) er at selv om binomialkoeffisienten vi er ute etter ikke nødvendigvis er så stor vil teller og nevner allikevel kunne være svært store. Slikt er generelt uheldig for numeriske beregninger og bør unngås så sant det er mulig og ikke koster for mye i form av ekstra regnetid. Hvis vi ser litt nøyere på formelen (2.15) så ser vi at vi kan forkorte slik at

$$\binom{n}{i} = \frac{1 \cdot 2 \cdots i \cdot (i+1) \cdots n}{1 \cdot 2 \cdots i \cdot 1 \cdot 2 \cdots (n-i)} = \frac{i+1}{1} \cdot \frac{i+2}{2} \cdots \frac{n}{n-i}.$$

Med andre ord kan vi hjelp av produktnotasjon skrive $\binom{n}{i}$ som

$$\binom{n}{i} = \prod_{j=1}^{n-i} \frac{i+j}{j}.$$

a) Programmer en metode for å beregne binomialkoeffisienter basert på denne formelen. Hvorfor må du bruke flyttall?

b) Test metoden din på binomialkoeffisientene

$$\binom{10000}{3} = 166616670000,$$
$$\binom{100000}{80} = 1.353770149276343 \cdot 10^{281},$$
$$\binom{1000}{500} = 2.702882409454366 \cdot 10^{299}.$$

c) Er det nå mulig å få for store tall underveis i beregningene dersom binomialkoeffisienten vi skal beregne ikke er større enn det største flyttallet vi kan representere.

d) Når vi utledet metoden vår forkortet vi $i!$ mot $n!$ i (2.15) og forenklet på den måten uttrykket for $\binom{n}{i}$. Vi kan utlede en annen metode ved i steden å forkorte $(n-i)!$ mot $n!$. Utled denne metoden på samme måte som over, og diskuter når de to metodene bør brukes.

2.6 Skriv et program for å regne ut den eksakte verdien av 500! ved hjelp av Java-klassen for å håndtere vilkårlig store heltall (som fasit kan du bruke at $500! \approx 1.220136825991110 \cdot 10^{1134}$).

2.7 Skriv et program for å beregne e^{π} med 100 siffer (her er e grunntallet for naturlige logaritmer) ved hjelp av Java-klassen for å håndtere flyttall med vilkårlig mange siffer.

2.8 Reelle tall kan betraktes som desimaltall, og flyttall framkommer når desimaltallene gjøres 'endelige'. Men reelle tall kan også ses på som grenser for rasjonale tall slik at vi kan alltid finne en vilkårlig god tilnærming til et reelt tall ved hjelp av et rasjonalt tall. Et alternativ til maskinrepresentasjon av reelle tall med flyttall kan derfor være representasjon ved hjelp av rasjonale tall. Diskuter fordeler og ulemper med en slik representasjon (hvordan kommer begrensingene med endelige ressurser til uttrykk, vil vi få avrundingsfeil etc.).

2.9 I denne oppgaven skal du utvide Java med en klasse for å håndtere komplekse tall.

a) Skriv en klasse for å representere komplekse tall i Java. Klassen skal inneholde metoder som implementerer addisjon, multiplikasjon, divisjon og kompleks konjugering, og lag også en metode for å skrive ut komplekse tall på en naturlig måte.

b) Skriv en metode som implementerer den komplekse eksponensialfunksjonen definert ved

$$e^z = e^{x+iy} = e^x(\cos y + i \sin y).$$

Test metoden på de tre funksjonsverdiene

$$e^{i\pi} = -1, \qquad e^{i\frac{\pi}{2}} = i, \qquad e^{\frac{\ln 2}{2} + i\frac{\pi}{4}} = 1 + i.$$

2.10 I denne oppgaven skal vi se hvordan overflow og verdien `NaN` oppfører seg i Java.

a) Skriv et program som forsøker å utføre de to udefinerte divisjonene 1.0/0.0 og 1/0, lagrer resultatet i henholdsvis en `double`-variabel og en `long`-variabel og til slutt skriver ut resulatet.

b) Bruk variable av typen `long` og multipliser sammen de to tallene 2^{50} og 2^{30}. Sjekk om svaret stemmer med den riktige verdien som er

$$2^{80} = 1208925819614629174706176.$$

c) Bruk `double`-variable, multipliser sammen de to tallene 10^{200} og 10^{300} og skriv ut svaret. Blir det 10^{500}?

d) Adder 1 og ta kvadratroten til svaret i foregående oppgave. Hva blir resultatet?

2.11 Skriv et program der du deklarer en variabel `n` av typen `long` og en variabel `x` av typen `float`. Gi først `n` verdien 10^{18}, sett `x` lik `n` og skriv ut det som nå er innholdet i `x`. Forklar resultatet.

2.12 Anta at $\tilde{a}$ er en tilnærming til a og regn ut den relative feilen i tilfellene under. Sammenlign også den relative feilen med antall riktige siffer.

a) $a = 1$ og $\tilde{a} = 0.9994$.

b) $a = 24$ og $\tilde{a} = 23.56$.

c) $a = -1267$ og $\tilde{a} = -1267.345$.

d) $a = 124$ og $\tilde{a} = 7$.

2.13 Skriv følgende tall i totallsystemet.

a) 35

b) 100

c) 256

d) -67

2.14 Skriv tallene under i totallsystemet.

a) 0.75

b) 0.125

c) 0.4

d) 0.1

2.15 Konvertering til tretallsystemet.

a) Bruk metoden over for å konvertere helttall fra desimal til binær form som utgangspunkt, og utled en metode for å konvertere heltall fra titallsystemet til tretallsystemet. Test metoden på eksemplene $67 = 2111_3$ og $100 = 10201_3$.

b) Utled en metode for å konvertere desimaltall i intervallet $(0, 1)$ til tretallsystemet, og test metoden på eksemplene

$$
\begin{aligned}
1/3 &= 0.1_3, \\
0.5 &= 0.111111111\ldots_3\,, \\
0.1 &= 0.0022002200220022\ldots_3\,.
\end{aligned}
$$