Herman T. Aagaard

Rapport til arbeidskrav 1 i IDATT2101

1) (Se metode p() i vedlagt kildekode)

2) (Se metode p2() i vedlagt kildekode. Sammenligning av tid blir gjort i oppgave 3)

3) I vedlagt Java fil finner dere to metoder kalt p og p2 hvor p er den rekursive metoden fra oppgave 2.1-1, og p2 er den rekursive metoden fra oppgave 2.2-3 (Se bilder). For å sjekke om metodene mine funket så kjørte jeg to tester med potensene 2^10 og 3^14 og sammenlignet svarene fra mine rekursive metoder med Java sin egen Math.pow(). Alle metodene ga samme svar som er en god indikasjon på at metodene mine stemmer (Se bilde til høyre).

```java
private double p(double x, int n){
    if (n == 0){
        return 1;
    }
    return x * p(x, n-1);
}
private double p2(double x, int n){
    if (n == 0){
        return 1;
    }
    else if (n%2 == 0){
        return p2(x*x, n/2);
    }
    else {
        return x*p2(x*x, (n-1)/2);
    }
}
```

```
Metode som ikke tar hensyn til partall og oddetall:
2^10: 1024.0
3^14: 4782969.0

Metode som tar hensyn til partall og oddetall:
2^10: 1024.0
3^14: 4782969.0

Math.pow():
2^10: 1024.0
3^14: 4782969.0
```

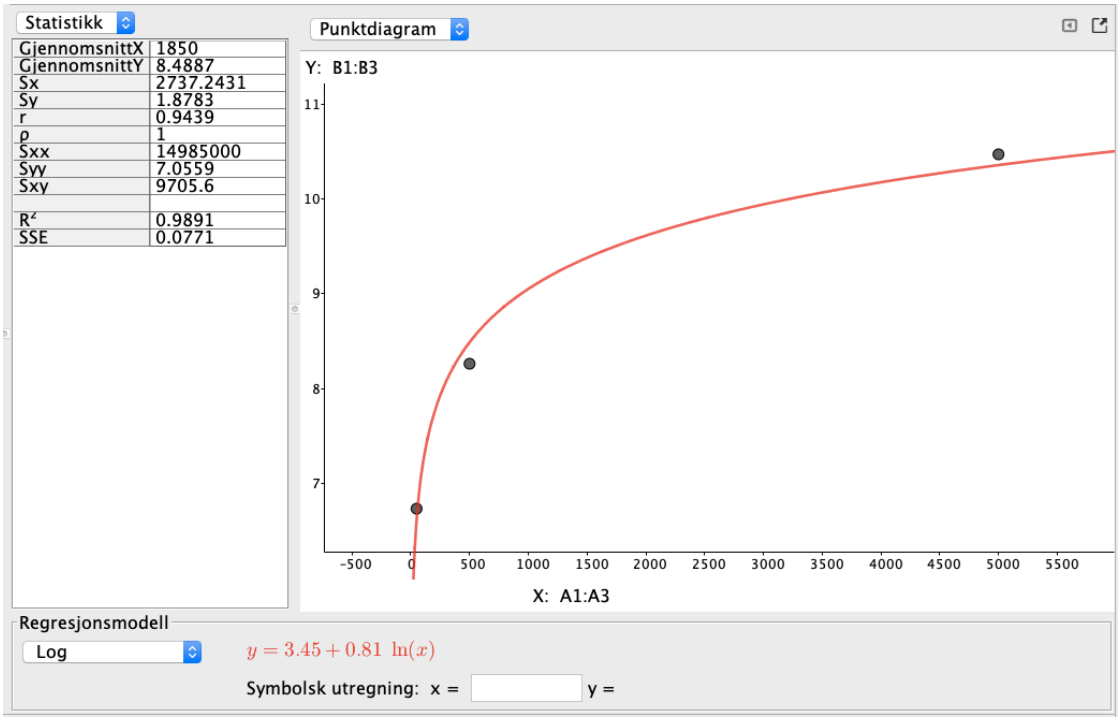
For å måle tiden brukte jeg koden som ble publisert i blackboard og la til en avrunding selv. For å få en mest mulig nøyaktig måling kjørte jeg tester fem ganger på hver metode med potensene 1.0001^50 1.0001^500 1.0001^5000. Hvis jeg fikk noen resultater som vek betraktelig fra de jeg vanligvis fikk så jeg vekk fra dem. Resultatene ble målt i millisekund og ga:

	p() (2.1-1)	p2() (2.2-3)	Math.pow()
1.0001^50	1,7486*10^-4	6,734*10^-5	4,384*10^-5
1.0001^500	1,478*10^-3	8,262*10^-5	3,982*10^-5
1.0001^5000	3,345*10^-2	1,047*10^-4	4,006*10^-5

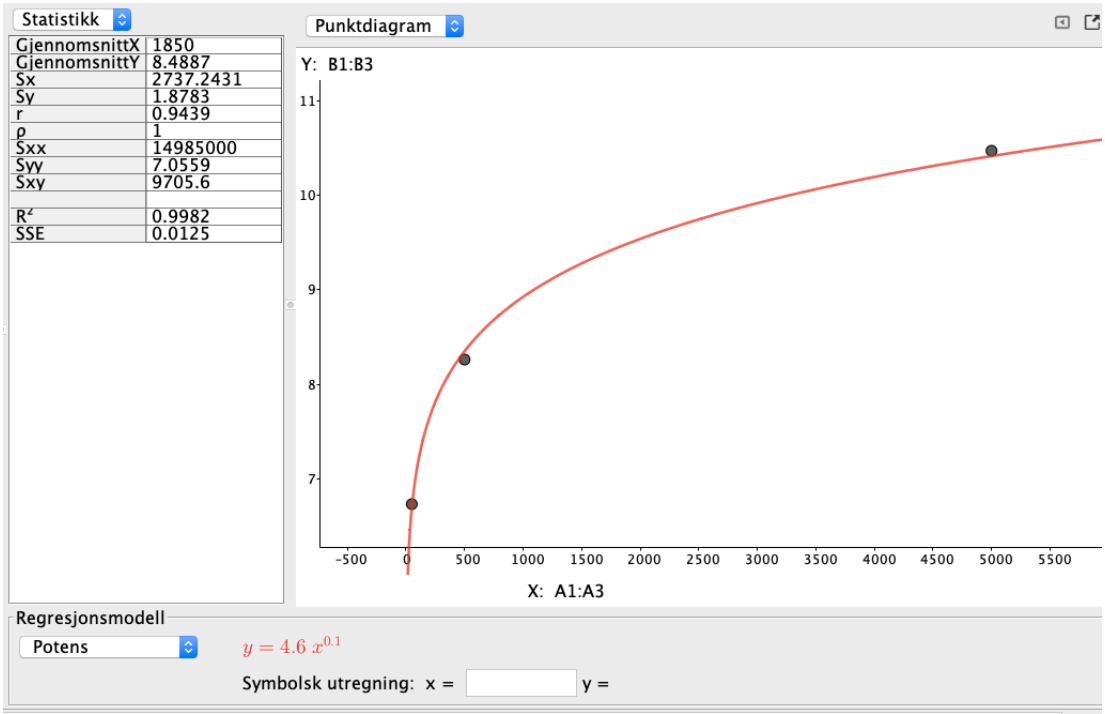
Her viser lite overraskende at Java sin egen metode Math.pow() er den raskeste når det kommer til potensregning. Endringen i eksponent viste seg å faktisk minske tiden brukt i mine tester. Etter dette kom metoden fra oppgave 2.2-3 som tar hensyn til oddetall og partall. Denne metoden var også relativt rask men viser tydelig at Math.pow() er et smartere alternativ. Til slutt kom metoden fra oppgave 2.2-1 som økte veldig raskt i hastighet og er en generelt dårlig løsning hvis man skal regne potenser med store eksponenter.

Fra denne tabellen er det enkelt å se at metoden fra oppgave 2.1-1 øker mer eller mindre lineært og kommer til å bli treig hvis man hadde brukt en stor eksponent. Hvis man plotter inn resultatene i geogebra og gjør en regresjonsanalyse ser vi at de to grafene som passer best til denne metoden er en logaritmisk eller en potens graf hvor kurven flater mer ut jo større eksponent man bruker.

Logaritmisk graf (x = eksponent):



Potens graf (x = eksponent):



Hva kommer forskjellen i tid fra?

Kort forklart kan man si at forskjellen i tid kommer fra antall operasjoner datamaskinen må gjøre. Jeg tar kun for meg de to metodene jeg selv har lagd og ikke Math.pow() siden jeg ikke har sett kildekoden på den metoden og fordi jeg ikke tror dette er hva foreleser ønsker at vi skal forklare. Derfor ser vi kun på metodene p() (oppgave 2.1-1) og p2() (oppgave 2.2-3).
Ved første øyekast ser det kanskje ut som at p() skal være en raskere metode enn p2() siden p() har færre linjer kode og kun en if-setning som skal sjekkes mens p2() har dobbelt så mange operasjoner (if-setninger, return statements og utreginger). Hvor p2() sparer tid i forhold til p() kommer når den rekursive metoden kalles. Her ser vi at p() kaller på seg selv n ganger, derimot i p2() ser vi at når den rekursive metoden kalles så halveres antall n. Altså hvis vi har en potens med eksponent 1000 så vil p() kjøres 1000 ganger før den kan returnere det fulle svaret. p2() derimot vil se at 1000 er et partall og så halvere eksponenten til 500 før metoden kalles på nytt. Allerede her har p2() hoppet over 500 kall på seg selv. Så selv om hver gjennomgang av p2() kanskje er tregere enn hver gjennomgang av p() så blir antall gjennomganger mye mindre i p2() enn det blir i p(). Med veldig lave eksponenter som 2 eller 3 viste en kjapp test at det varierte på hvilken metode som var raskest, men desto høyere eksponenten er desto flere steg vil p2() hoppe over og derfor også spare inn betraktelig mye tid som vist i tabellen på side 1.

```
private double p(double x, int n){
    if (n == 0){
        return 1;
    }
    return x * p(x,  n: n-1);
}
```

```
private double p2(double x, int n){
    if (n == 0){
        return 1;
    }
    else if (n%2 == 0){
        return p2( x: x*x,  n: n/2);
    }
    else {
        return x*p2( x: x*x, n: (n-1)/2);
    }
}
```