

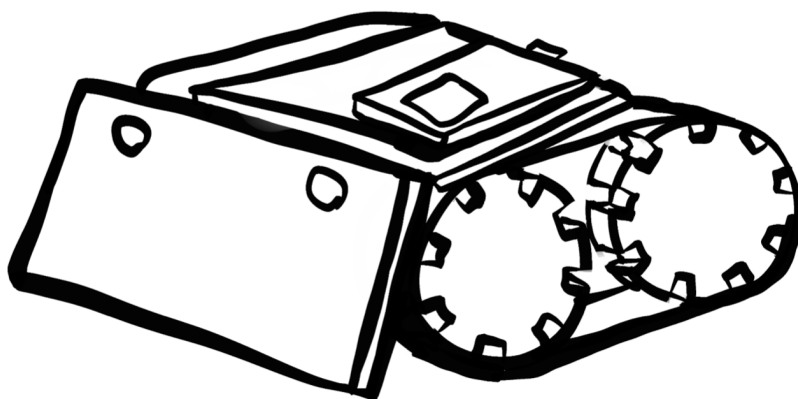
INSTITUTT FOR TEKNISK KYBERNETIKK SYSTEMER

AUTOMATISERING OG INTELLIGENTE SYSTEMER

---

## Oppstartslab

---



*Forfattere:*

Adrian Valaker Eikeland  
Ravn Erik Budde  
Jørgen Vesterheim Knutsen

August 11, 2025

---

# Git og oppstart

## Installasjon og oppsett av Python og Git

### Forhåndskrav - MATLAB

**OBS! VIKTIG:** Dette prosjektet krever MATLAB installert lokalt på datamaskinen din. Du må:

1. Ha betalt semesteravgift og fått NTNU-epost
2. Last ned MATLAB fra NTNU sin portal
3. **OBS:** Du må laste ned MATLAB lokalt på PC-en din - ikke bruk web-versjonen!
4. Du trenger også "Industrial Communication Toolbox" og "Instrument Control Toolbox" installert i MATLAB samt Simulink

Hvis du ikke har dette på plass, gjør det før du fortsetter med resten av guiden.

## Windows-versjon

### Steg 1: Installer Python

Først må vi installere Python på datamaskinen din. Python er programmeringsspråket vi skal bruke noen av gangene.

Åpne **Command Prompt** (CMD) ved å trykke **Windows + R**, skriv `cmd`, og trykk **Enter**.

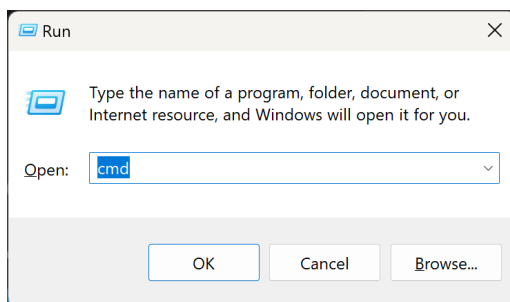


Figure 1: Windows + R

I terminalen, skriv følgende kommando for å installere Python:

```
winget install Python.Python.3.13 --accept-package-agreements
```

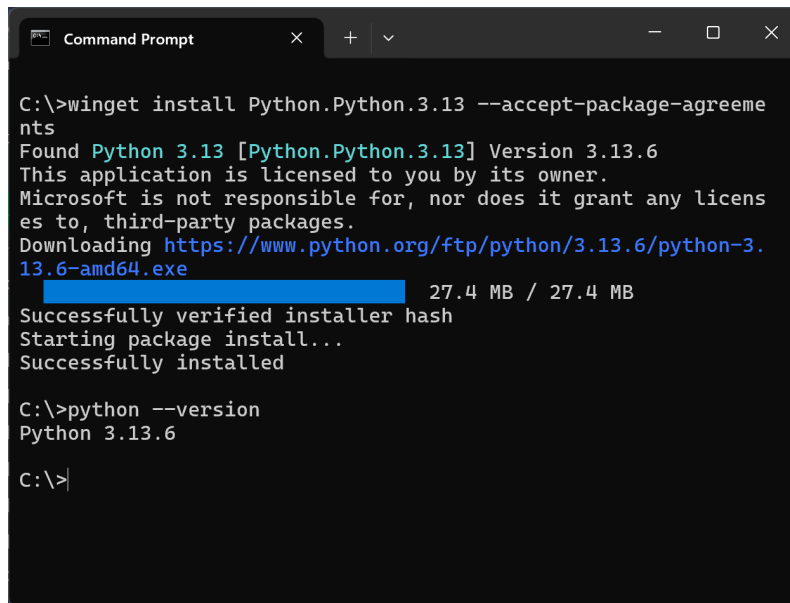
Vent til installasjonen er ferdig. For å sjekke at Python ble installert riktig, skriv:

```
python --version
```

Du skal nå se noe som ligner på `Python 3.x.x`. Hvis ikke, prøv å lukke terminalen og åpne en ny.

### Steg 2: Installer Git

Git er et verktøy for å laste ned og administrere kodeprosjekter. Vi installerer det med denne kommandoen:



```
C:\>winget install Python.Python.3.13 --accept-package-agreements
Found Python 3.13 [Python.Python.3.13] Version 3.13.6
This application is licensed to you by its owner.
Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.
Downloading https://www.python.org/ftp/python/3.13.6/python-3.13.6-amd64.exe
27.4 MB / 27.4 MB
Successfully verified installer hash
Starting package install...
Successfully installed

C:\>python --version
Python 3.13.6

C:\>
```

Figure 2: Python-installasjon og versjonsjekk

```
winget install --id Git.Git -e --source winget
```

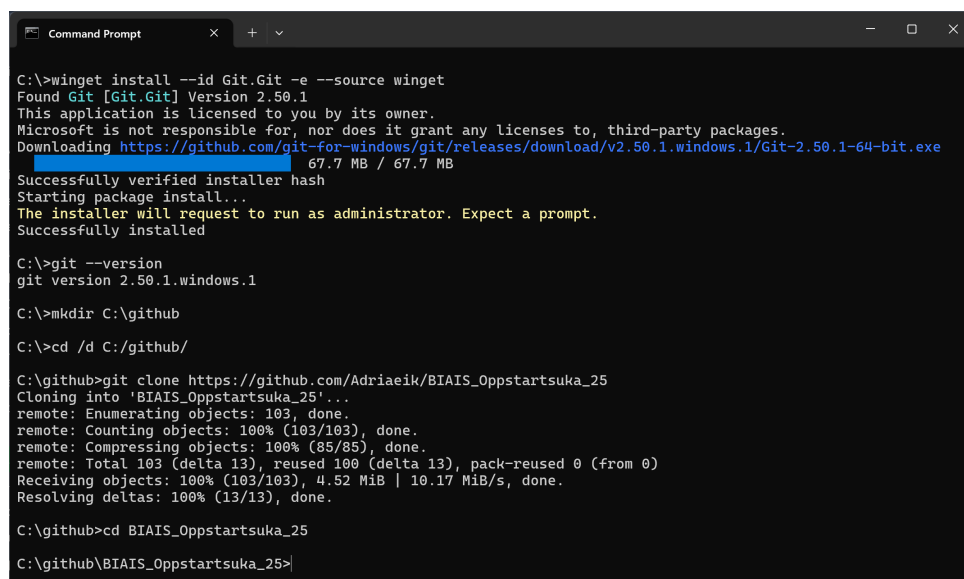
For å sjekke at Git ble installert riktig, skriv:

```
git --version
```

Du skal nå se noe som ligner på `git version 2.x.x`.



Hvis Git ikke blir gjenkjent, prøv å lukke terminalen og åpne en ny.



```
C:\>winget install --id Git.Git -e --source winget
Found Git [Git.Git] Version 2.50.1
This application is licensed to you by its owner.
Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.
Downloading https://github.com/git-for-windows/git/releases/download/v2.50.1.windows.1/Git-2.50.1-64-bit.exe
67.7 MB / 67.7 MB
Successfully verified installer hash
Starting package install...
The installer will request to run as administrator. Expect a prompt.
Successfully installed

C:\>git --version
git version 2.50.1.windows.1

C:\>mkdir C:\github

C:\>cd /d C:/github/

C:\github>git clone https://github.com/Adriaek/BIAIS_Oppstartsuka_25
Cloning into 'BIAIS_Oppstartsuka_25'...
remote: Enumerating objects: 103, done.
remote: Counting objects: 100% (103/103), done.
remote: Compressing objects: 100% (85/85), done.
remote: Total 103 (delta 13), reused 100 (delta 13), pack-reused 0 (from 0)
Receiving objects: 100% (103/103), 4.52 MiB | 10.17 MiB/s, done.
Resolving deltas: 100% (13/13), done.

C:\github>cd BIAIS_Oppstartsuka_25
C:\github\BIAIS_Oppstartsuka_25>
```

Figure 3: Git-installasjon og versjonsjekk

### Steg 3: Klon prosjektet

---

Nå skal vi laste ned prosjektfilene fra internett. Et *repository* (eller "repo") er ganske enkelt et prosjektarkiv som lagres på nettet.

Først lager vi en mappe kalt "github" der vi kan samle alle våre prosjekter:

```
mkdir C:\github\
```

```
cd /d C:\github\
```

Nå kan vi klonе (laste ned) prosjektet:

```
git clone https://github.com/Adriaek/BIAIS_Oppstartstuka_25
```

**OBS!** Får du en feilmelding om at mappen allerede finnes, kan du enten slette den eksisterende mappen eller klonе til et annet navn:

```
git clone https://github.com/Adriaek/BIAIS_Oppstartstuka_25 ny_mappe_navn
```

#### Steg 4: Gå til prosjektmappen

Når nedlastingen er ferdig, naviger inn i prosjektmappen:

```
cd BIAIS_Oppstartstuka_25
```



Hvis du endret mappenavnet over, bruk det nye navnet her.

#### Steg 5: Start MATLAB

Skriv inn matlab i terminalen og trykk enter:

```
matlab
```

Vent tålmodig mens MATLAB starter opp, og utnytt muligheten til å bli kjent med gruppa di.

#### Steg 6: Åpne første oppgave

I MATLAB: Naviger til `del1/oppgave1.slx` og åpne filen. Nå er dere klare til å sette i gang!

### Mac-versjon

**OBS!** Mac-versjonen er ikke testet og vil ikke bli testet. Mac-brukere som får problemer må søke hjelp på Stack Overflow eller andre ressurser. Studentassistenter bruker ikke tid på dette.

#### Forhåndskrav

På Mac trenger du først **Homebrew** (en pakkebehandler). Hvis du ikke har det, installer det først ved å følge instruksjonene på `brew.sh`.

#### Steg 1: Installer Python

Åpne **Terminal** (søk etter "Terminal" i Spotlight eller finn den i Applications/Utilities).

Installer Python med Homebrew:

---

```
brew install python
```

Sjekk at installasjonen fungerte:

```
python3 --version
```

## Steg 2: Installer Git

Git kommer ofte ferdig installert på Mac, men vi installerer den nyeste versjonen:

```
brew install git
```

Sjekk installasjonen:

```
git --version
```



Hvis Git ikke blir gjenkjent, prøv å lukke terminalen og åpne en ny.

## Steg 3: Klon prosjektet

Naviger til ønsket mappe og opprett en "github"-mappe for å samle prosjekter:

```
mkdir ~/github/
```

```
cd ~/github/
```

Klon prosjektet:

```
git clone https://github.com/Adriaek/BIAIS_Oppstartssuka_25
```

## Steg 4-6: Felles for begge plattformer

Følg steg 4-6 fra Windows-delen ovenfor. De er identiske for begge plattformer.



For Mac: `matlab`-kommandoen kan variere avhengig av hvordan MATLAB er installert på din Mac.

## Liten forklaring av hva som skjer

- **Python** er programmeringsspråket som ofte brukes for dataanalyse og analyse - vi bruker det for å visualisere data i dag
- **Git** er et versjonskontrollsystem som lar oss laste ned og administrere kodeprosjekter
- **Kloning** betyr å laste ned en kopi av et prosjekt fra internett til din lokale maskin
- **Repository** (repo) er en samling av filer og mapper som utgjør et prosjekt

---

## Poengberegning

Vi vil gi dere poeng både manuelt og automatisk gjennom hele uken. Dere kan følge med på live-scoreboardet for å se hvordan dere ligger an. Hver del teller 33 %, og dersom det er flere oppgaver i en del, fordeles prosenten på deloppgavene.

Poengene beregnes ut fra hvordan dere presterer relativt til de andre gruppene. Når dere har meldt en oppgave som ferdig, kan dere ikke gå tilbake til den før alle andre er ferdige. Når alle oppgaver er gjennomført, får dere lov til å gå tilbake og forbedre scoren.

### Slik fungerer det:

- Alle gruppenes resultater samles og danner en fordeling
- Deres plassering på denne fordelingen bestemmer poengsummen
- Jo bedre dere presterer relativt til andre, desto høyere poengsum

På oppgaver der poeng settes manuelt, skal dere vise resultatet når dere mener dere er klare. Dere skriver dere da opp på tavlen for vurdering. Etter hvert vurderingsforsøk pålegges det en tidsbestemt karantene før dere kan be om ny vurdering.

Scoreboardet ser ut som vist i figur 4. Seksjonene er som følger:

- Gruppe. Her vises gruppenavnet.
- Sum poeng. Her vises hvor mange poeng gruppen har til sammen. Poengene er dynamiske, og endrer seg basert på gruppens og andre gruppers resultater.
- Oppgave x. Her vises resultatet til hver oppgave før de er omgjort til score. Dermed er det lettere for gruppen å vite hvilke resultat man trenger for å forbedre scoren på oppgaven. Ved siden av står en %-verdi i parentes. Denne indikerer hvor på fordelinga av alle resultater gruppa ligger, og indikerer hvilke oppgaver en gruppe har gjort det bedre eller dårligere på i forhold til de andre gruppene.




Gruppe	Sum poeng	Oppgave 1.1	Oppgave 1.2	Oppgave 2	Oppgave 3
 3.0	216.53	1.24 (70%)	2.67 (30%)	33.01 (100%)	7.23 (66%)
 10.0	185.05	0.67 (92%)	0.89 (79%)	19.55 (42%)	8.33 (58%)
 2.0	173.78	1.93 (45%)	0.12 (100%)	9.87 (0%)	2.75 (100%)
4.0	163.82	0.98 (80%)	1.54 (61%)	18.79 (39%)	9.04 (52%)
7.0	159.01	0.58 (95%)	2.19 (43%)	21.67 (51%)	11.10 (37%)
1.0	156.18	0.45 (100%)	3.78 (0%)	28.34 (80%)	12.50 (26%)
8.0	144.67	3.12 (0%)	1.05 (75%)	17.99 (35%)	6.42 (72%)
6.0	134.51	1.11 (75%)	3.22 (15%)	12.55 (12%)	5.75 (77%)
9.0	127.22	2.46 (25%)	3.35 (12%)	24.28 (62%)	10.07 (45%)
5.0	122.34	2.85 (10%)	0.75 (83%)	27.43 (76%)	15.99 (0%)

Figure 4: Scoreboard

## Rask Simulink-intro

### Redigere blokker

De fleste blokker kan åpnes og endres ved å dobbeltklikke på dem. For eksempel: dobbeltklikk på de små trekantene i **PID-regulatoren** for å justere parametrene.

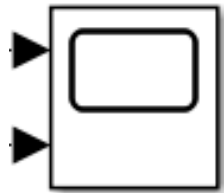
---

## Stop Time

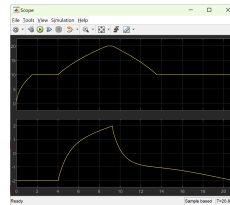
Øverst i verktøylinjen, under fanen **Simulation**, finner gruppen feltet **Stop time**. Verdien der bestemmer hvor lenge simuleringen skal kjøres.

## Scope

Blokken **Scope** viser signalverdier som funksjon av tid. Resultatet vises som en graf når gruppen dobbeltrykker på scope-ikonet i Simulink.



(a) Scope-blokk



(b) Scope-graf

## Integrator $\frac{1}{s}$

Blokken  $\frac{1}{s}$  er en integrator i  $s$ -domenet. Foreløpig trenger gruppen bare å vite at den utfører integrasjon av en verdi med hensyn på tid.

## Gain

Blokken **Gain** multipliserer inngangssignalet med en skalarverdi.

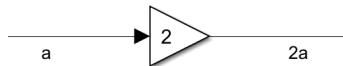


Figure 6: Gain-blokk

## Mux og Demux

- **Mux** (venstre) kombinerer flere separate signaler til ett vektorsignal.
- **Demux** (høyre) deler et vektorsignal opp i flere individuelle signaler.

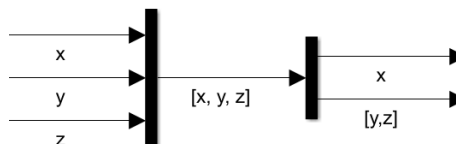
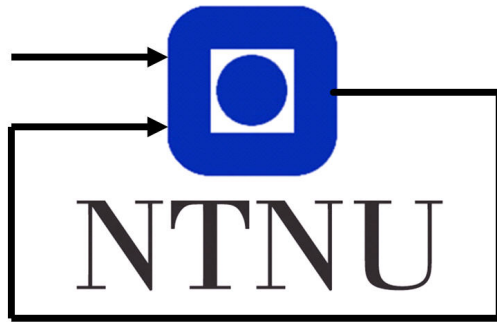


Figure 7: Mux-blokk

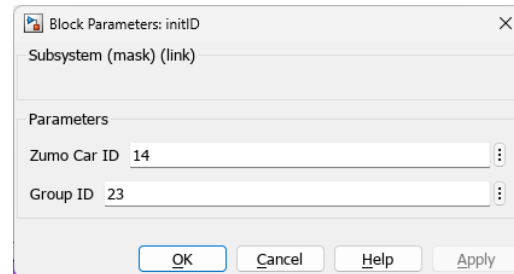
---

## En viktig blokk i Simulink

I alle oppgaver vil dere se en blokk som vist i figur 8. Her må dere åpne blokken og angi både gruppe-ID og bilens ID. Dette er to separate verdier, og det er viktig at dere ikke blander dem. Husk å trykke *Apply* når dere har fylt inn informasjonen. Dette må gjøres i hver eneste oppgave.



(a) Symbolet for blokken der gruppe- og bil-ID skrives inn.



(b) GUI for innskriving av ID.

Figure 8: Blokken for registrering av gruppe-ID og bil-ID.

## 1 Lokalt nettverk – dere mister nå tilgang til internett på PC-en

For å kommunisere med bilene og scoreboardet skal vi koble oss til et lokalt nettverk. Nettverksinformasjonen er vist i tabell 1.

Nettverksnavn (SSID)	BIAIS_OppstartsVeka
Passord	shinyteapot294

Table 1: Informasjon for tilkobling til lokalt nettverk.

Logg på dette nettverket før dere starter oppgavene.

### 1 PID regulator

## Del 1 - Reguleringsteknikk

### Innledning til regulering

I denne delen skal dere lære grunnleggende reguleringsteknikk gjennom praktiske oppgaver. Dere kommer til å jobbe med å kontrollere en lastebil i Simulink.

Å regulere et system betyr å styre det mot en ønsket tilstand. Den enkleste formen er **åpen sløyfe**: vi bestemmer et pådrag uten å se hvordan systemet faktisk oppfører seg, som sjeldent fungerer godt. Et bedre alternativ som nesten alltid er det som brukes i praksis er **lukket sløyfe**, der vi måler forskjellen mellom:

- referansen  $r$  – det vi *ønsker*
- målt verdi  $y$  – det vi *har*



---

Avviket (*error*) defineres som:

$$e(t) = r(t) - y(t)$$

Målet er å få  $e(t) \rightarrow 0$ .

### Tidskonstanten $\tau$ (tau)

I reguleringsteknikk bruker vi tidskonstanten  $\tau$  for å måle hvor raskt et system responderer. Dette er tiden det tar for systemet å nå 63,2% av sin endelige verdi etter en trinnforandring.

**Hvorfor akkurat 63,2%?** Dette kommer fra formelen  $1 - e^{-1} \approx 0,632$ , som beskriver oppførselen til førstegrads systemer.

Tidskonstanten gir oss et standardisert mål på systemets hurtighet, uavhengig av størrelsen på endringen. Jo mindre  $\tau$ , desto raskere er systemet.

**Valgfri teori:** Teori for den interesserte

I det aller meste av industrien gjøres dette med en **PID-regulator** (*Proportional-Integral-Derivative*). Hele ideen kan samles i én formel:

$$u(t) = K_p e(t) + K_i \int_{t_0}^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

hvor:

- $u(t)$  = pådraget vi sender til systemet
- $K_p, K_i, K_d$  = regulatorparametere som må justeres

**P-leddet** (*ser på "nå"*):  $u_P = K_p e(t)$ . Jo større  $|e|$ , desto hardere gasser (eller bremses) vi. Nær referansen minker  $u_P$ , og vi kan få **stasjonært avvik** – systemet legger seg under fartsgrensen og kommer aldri helt opp.

**I-leddet** (*ser på "fortiden"*):  $u_I = K_i \int e dt$ . Små feil over lang tid summeres opp til noe stort, slik at regulatoren trykker litt ekstra på gassen og fjerner stasjonært avvik. Velges  $K_i$  for høyt, får dere treg og ustabil regulering.

**D-leddet** (*ser inn i "fremtiden"*):  $u_D = K_d \frac{de}{dt}$ . Reagerer på hvor raskt feilen endrer seg. Demper oversving og gjør systemet roligere, men for høy  $K_d$  kan forsterke støy.

Alle tre leddene summeres til ett felles pådrag  $u(t)$ .

---

## Del 1.1 - Regulering av fart

Nå skal dere få en lastebil til å holde riktig fart! Bevegelsen til lastebilen kan beskrives med Newtons andre lov:

$$F = m a$$

Hvis vi i tillegg modellerer luftmotstand, kan vi gjenskape dynamikken til en ekte lastebil med høy nøyaktighet.

I deres Simulink-modell vil:

- Motoren skaper drivkraft når dere gir gass
- Luftmotstand jobber mot lastebilen
- Dere kan **kun** kontrollere gasspådraget – ingen bremses!

### Oppgave 1.1 - Cruise-kontroll

Dere skal implementere en cruise-kontroll som får lastebilen til å kjøre i 20 m/s så raskt som mulig – men uten å bryte fartsgrensen!

1. Åpne `oppgave1.slx` i MATLAB
2. Juster PID-parameterne ( $K_p$ ,  $K_i$ ,  $K_d$ ) for å få best mulig regulering
3. Eksperimenter og undersøk:
  - Hva skjer når  $K_p$  er for stor? For liten?
  - Får dere stasjonært avvik (lastebilen når ikke helt 20 m/s)? Prøv å auke  $K_i$
  - Oscillerer systemet frem og tilbake? Øk  $K_d$  for demping
4. mål tiden  $\tau$  ved å opne scopet og trykke på linjalikonet til høyre i toolbaren.

**Mål:** Raskest mulig til fartsgrensen UTEN oversving!

- Hurtighet måles med tidskonstanten  $\tau$  – tiden til 63% av referansen
- For deres tilfelle: når lastebilen når 12,6 m/s
- Oversving = å kjøre over 20 m/s (diskvalifiserer resultatet)



#### Læringsmål for oppgaven:

- Få intuitiv følelse for hvordan tilbakekopling påvirker et system
- Forstå rollen til P-, I- og D-leddet i praksis
- Finne balansen mellom ”for rask” og ”for treg” regulering

Bruk ikke uhorvelig tid, men se øvingen som en mulighet til å teste og lære.

---

## Del 1.2 - Forbikjøring

Nå som dere mestrer fartsregulering, er det tid for en større utfordring! Tenk dere at dere kjører bak en skikkelig *sinke* – for å redde både pulsen og køen bak må dere ta en forbikjøring. Da må dere regulere både **fart** og **sideposisjon** samtidig.

### PID for styring

Dere har allerede en **hvit PID-blokk** i modellen som regulerer farten. Bruk parameterne dere fant i forrige oppgave her!

Nå må dere selv sette inn en PID-blokk for å kontrollere rattet:

1. Dobbelklikk på et tomt område i Simulink-modellen
2. Skriv PID og trykk Enter
3. Koble signal til den nye PID-blokken slik:
  - **Input:** `e_pos` (avstand fra midtlinjen i meter)
  - **Output:** `u_ratt` (rattvinkel i radianer)

**OBS!** Rattet er VELDIG følsomt!

### Automatisk forbikjøringssekvens

Simulink-modellen inneholder en automatisk sekvens som styrer forbikjøringen:

Fase	Handling	Fartsreferanse	Posisjonsreferanse
1	Klargjøring	Fartsgrense	Høyre felt
2	Forbikjøring	Maks tillatt	Venstre felt
3	Tilbake	Fartsgrense	Høyre felt

Simuleringen avsluttes automatisk når dere er trygt tilbake i høyre felt.

### Oppgave 1.2 - Sikker forbikjøring

1. Sett inn og koble opp PID-blokken for styring som beskrevet
2. Juster PID-parameterne for rattet slik at:
  - Lastebilen holder seg stabilt i filen
  - Filskiftene skjer raskt men kontrollert
  - Ingen oscillerende eller "slingring"
3. Optimaliser både fart- og styringskontroll for raskest mulig forbikjøring

**Mål:** Raskest mulig gjennom hele forbikjøringssekvensen!

- Tiden måles automatisk fra start til dere er tilbake i høyre felt
- Krav: Ingen oversving, oscillasjoner eller ustabil kjøring - minnimer rundetid og noise
- Tips: Balanse mellom aggressiv og forsiktig innstilling er nøkkelen



**Læringsmål for oppgaven:**

- Oppleve utfordringen med å regulere flere variabler samtidig (fart og posisjon)
- Lære å finne kompromisser mellom motstridende krav (hurtighet vs. stabilitet)
- Se hvordan regulatorinnstillinger påvirker hverandre
- Få praktisk erfaring med typiske reguleringsproblemer i kjøretøy

---

## 2 Sensorer

I denne oppgaven skal vi se på en sentral del av alle reguleringssystemer- nemlig sensorer. Sensorer er det vi bruker for å la systemet vårt ”se” verden rundt seg. Dette er viktig fordi hvis vi ikke kan se verden, så kan vi ikke reagere på den!

Denne oppgaven er designet for å gi et lite innblikk i hvordan type data vi kan forvente å få inn fra en sensor, og hvordan vi veldig ofte må behandle datene før de er nyttige for oss!

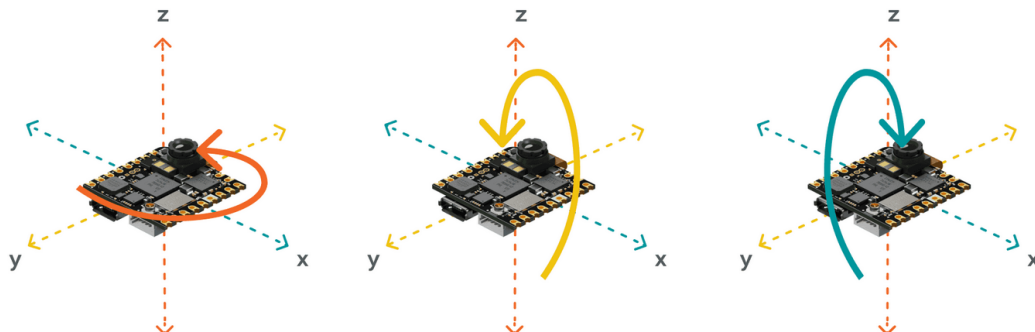
### Bakgrunn

#### Inertial Measurement Unit

En Inertial Measurement Unit, eller en IMU, er en sensorpakke som måler bevegelse og orientering. Den består vanligvis av tre typer sensorer: akselerometre, gyroskoper og noen ganger magnetometre. Systemet vårt har et magnetometer. Akselerometrene måler lineær akselerasjon (hvor fort noe endrer hastighet), gyroskopene måler rotasjonshastighet, og magnetometrene gir et mål på retning i forhold til jordens magnetfelt (kompassfunksjon).

Ved å kombinere målingene fra disse sensorene kan vi estimere hvordan et objekt beveger seg i rommet — både hvilken vei det peker, og hvordan det akselererer. Dette er spesielt nyttig i alt fra droner og roboter til mobiltelefoner og VR-briller. Kanskje har du lurt på hvordan mobilen din vet at den skal snu skjermen når du legger den på siden? Dette er hvordan!

En utfordring med IMU-data er at de ofte inneholder støy og små feil som, hvis de integreres over tid, kan føre til store avvik i estimatene. Derfor er det vanlig å filtrere eller kombinere IMU-data med andre sensorer (f.eks. GPS eller kamera) for å få mer nøyaktige og stabile posisjons- og orienteringsmålinger. Under ser du en illustrasjon av en hva en IMU kan måle.

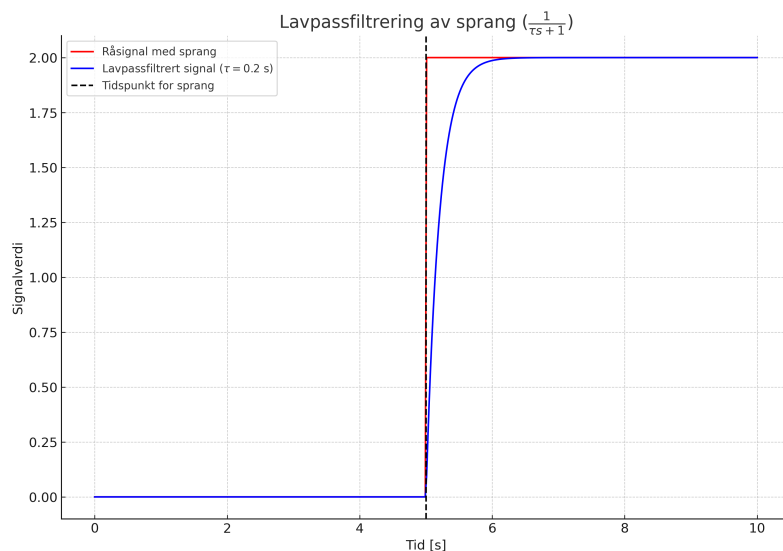


### Filtrering

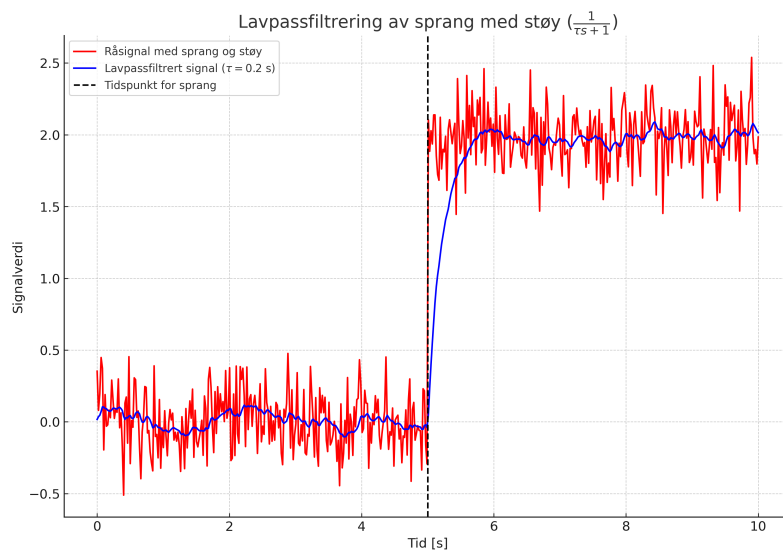
Som nevnt over kan støyete data være et stort problem. Det finnes mange spennende måter å håndtere dette på, og dere skal bli gode på dette i løpet av studiene. I første omgang skal vi se på en enkel, men effektiv måte å filtrere data på: lavpassfilter.

Et lavpassfilter gjør akkurat det navnet tilsier — lave frekvenser får passere, mens høye frekvenser blir dempet eller fjernet. Når vi snakker om ”lave frekvenser” mener vi signalendringer som skjer sakte over tid, for eksempel en langsom bevegelse eller en jevn temperaturendring. ”Høye frekvenser” er raske endringer, som plutselige hopp (sprang) eller vibrasjoner.

Figuren under viser et eksempel: det røde signalet inneholder et tydelig sprang, mens det blå signalet viser samme data etter at et lavpassfilter er brukt. Legg merke til hvordan det blå signalet blir glattere og mindre påvirket av de brå endringene. Dette gjør det enklere for reguleringssystemet å tolke signalet uten å reagere unødvendig kraftig på kortvarige forstyrrelser.



Her så vi et eksempel på en stor forandring. Når vi snakker om støyete målinger, handler det ofte om mange små variasjoner i dataen, gjerne rundt den reelle verdien. Under ser vi enda et eksempel på et lavpassfilter, men denne gangen er målingen forstyrret av støy.



Som man ser her, kan det være svært gunstig å filtrere, eller “glatte ut” dataen! Heldigvis er det veldig enkelt å implementere et lavpassfilter i Simulink. Dette kan gjøres ved å legge til en **Transfer Fcn**-blokk. Dette står for *transfer function* (overføringsfunksjon), som beskriver sammenhengen mellom inngang og utgang.

Lavpassfiltrene vi skal implementere her er av første orden, og ser slik ut:

$$H(s) = \frac{1}{\tau s + 1}$$

Som nevnt i oppgave 1 forteller tidskonstanten  $\tau$  hvor hurtig systemet reagerer. En stor  $\tau$  gir et tregt system, mens en liten  $\tau$  gir rask respons. Her er det **viktig med en balansegang**— en for liten tidskonstant gir lite filtrering, mens en for stor kan ødelegge signalet.

---

## TiltMaze

Nå skal vi teste ut disse konseptene i praksis. For å demonstrere hvordan vi kan bruke dem, har vi laget et lite spill til dere: **TiltMaze**. Dette er en digital versjon av det klassiske, fysiske spillet med samme navn. Skjermbildet ser dere under:



Målet med spillet er å rulle den blå ballen hele veien til det blå feltet. Underveis må dere unngå hindringer og kan samle stjerner for å øke poengsummen.

### Fargekoder:

- **Grønn:** Startfelt
- **Rød:** Hull
- **Gul:** Stjerne
- **Rosa:** Checkpoint
- **Blå:** Mål

Oppgaven er enkel å forklare, men ikke alltid like enkel å mestre: Kom deg til mål med flest mulig poeng! Poeng trekkes eksponentielt avtakende ettersom tiden går, men dere kan hente dem tilbake ved å samle stjerner. Faller dere ned i et hull, mister dere både poeng og blir sendt tilbake til forrige checkpoint.

Dessverre glemte vi å kjøpe inn kontrollere til dere, men frykt ikke! Vi er ingeniører, og vi har en sensor tilgjengelig i den radiostyrte bilen. La oss bruke den til å lage en joystick så vi kan spille spillet og kjempe om høyest poengsum og evig ære.

Åpne opp simulink filen fra oppgave 2, og prøv dere på oppgavene.

### Lavpassfilteret

1. Sett inn et scope og inspiser og bli kjent med dataen som kommer fra gyroskop blokken
2. Bruk **Transfer Fcn**-blokken til å implementere et lavpass-filter. Prøv dere frem og se hva slags tidskonstant som er lur! **Hint:** Mux og demux blokkene vil nok være nyttige her
3. Send dataen videre inn i **estimate\_orientation**-blokken.



#### Læringsmål for oppgaven:

- Observere problemstillinger med støyete sensordata
- Implementere og teste ut lavpassfilter

Ser dataen dere sender inn i **estimate\_orientation**-blokken fin ut, så kan vi bevege oss videre. Nå som vi har fin data, så må vi faktisk se om vi klarer å tyde den og skape litt mening av den.

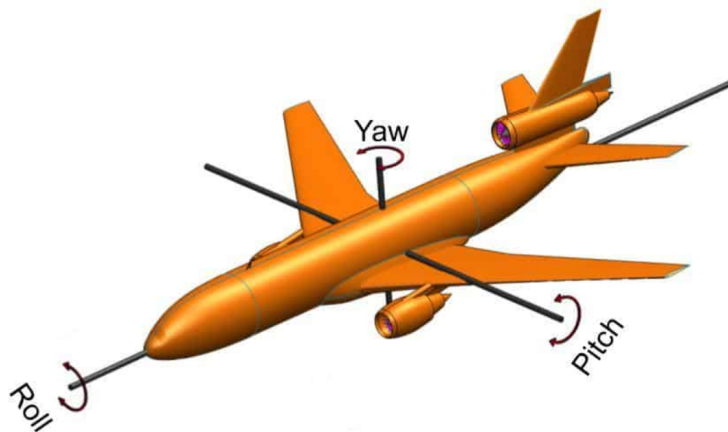
### Fra akselerometer til roll og pitch

Neste steg å finne ut hvilken vinkel ballen “heller” i— altså **roll** og **pitch**. For å gjøre dette må vi først vite hva disse begrepene betyr.

Når vi snakker om orientering i 3D-rom, bruker vi ofte tre vinkler:

- **Roll** — rotasjon rundt x-aksen (tipping til venstre eller høyre)
- **Pitch** — rotasjon rundt y-aksen (tipping fremover eller bakover)
- **Yaw** — rotasjon rundt z-aksen (svinge til siden)

Figuren under viser hvordan dette ser ut



I denne oppgaven bryr vi oss mest om roll og pitch, fordi spillet handler om hvordan brettet heller fremover/bakover og sideveis. Yaw er mer relevant for navigasjon og retning.

Fra et akselerometer får vi målinger av akselerasjon i tre retninger:  $a_x$ ,  $a_y$  og  $a_z$ . Når systemet står stille, er dette i praksis tyngdekraftens retning målt i IMU-ens koordinatsystem. Med litt trigonometri kan vi finne vinklene.



---

Denne oppgaven kan være krevende, løsningsforslag vil bli skrevet på tavle etterhvert. Det er forresten lov å samarbeid med andre grupper her! Løser dere oppgaven riktig får dere bonuspoeng.

### Tyde data og hente ut nyttig informasjon

1. Bruk trigonometri til å finne ut vinkelen Roll. **hint:** Lurt å tegne!
2. Bruk enda mer trigonometri til å finne vinkelen til pitch! Ta høyde for allerede eksisterende roll. **Hint:** Lurt å tegne og pythagoras dukker opp i løsningen!
3. Inne i `estimate_orientation` skal dere fylle inn kode der det er markert.



#### Læringsmål for oppgaven:

- Få erfaring med å bruke trigonometri i praksis
- Lære å behandle data fra en IMU
- Øve på problemløsning i grupper

Denne oppgavene kan kreve litt grubling, så diskuter det mellom dere.

### Lavpassfilteret del 2

1. Inspiser dataen på roll-pitch-yaw. Er denne fin nok til å bruke?
2. Vurder om dere trenger et filter her og!
3. Implementer eventuelt filter, eller fjern det hvis dere vurderer at det ikke er nødvendig. Send signalet inn i siste blokken for roll, pitch, og yaw.



#### Læringsmål for oppgaven:

- Vurdere nødvendighet for filter
- Prøve seg på designvalg
- Øve på problemløsning i grupper

Den siste blokken finner ut hvilken retning z-aksen peker akkurat nå. Med andre ord vet vi nå hva som er “oppover” for IMU-en. Denne informasjonen brukes i den siste funksjonsblokken til å dekomponere retningene, slik at bilen nå kan brukes som en joystick!

**OBS!** Koordinatrammen til IMU-en er i det som kalles en NED-frame (North–East–Down). Det betyr at “oppover” i denne rammen egentlig peker ned i vår intuitive forståelse. For at styringen skal oppføre seg som forventet, kan det derfor være lurt å snu bilen på hodet før man tester.

**Kortversjon:** Ferdig! Snu bilen på hodet, og bruk den som joystick.

---

## Hvordan bruke joystick

I mappen **Oppgave 2** finner dere enda en mappe som heter **TiltMaze**. Gå inn i mappen. Der finner dere to programmer som kan kjøres. Første gangen dere skal kjøre, må dere kjøre filen som heter **setup**. Dobbeltrykk på denne, og la den gjøre seg ferdig. Deretter kan dere trykke på **RunTiltMaze**, og sett i gang. Poengene kommer telles automatisk, og kun den beste scoren vil være tellende. Lykke til!

---

### 3 Linjefølging

I denne oppgaven skal dere få bilen til å kjøre av seg selv, ved å følge en bane laget av sort teip. Det vil være en 'offisiell' bane foran i rommet som vil bli brukt til konkurransen. For å unngå at den ikke er full konstant, vil hver gruppe få en karantene etter hvert forsøk. Dere vil få utdelt teip så dere kan lage egne baner under testing.

#### Bakgrunn

##### IR sensorer

Zumoen er utstyrt med 6 sensorer som oppfatter infrarødt lys (IR). IR sensorer kan brukes som nærhetsdetektorer, eller så kan de brukes til å "se" linjen på bakken ved å sjekke hvor mye infrarødt lys som reflekteres fra en IR LED-lampe. Denne skal vi bruke til "se" mørk teip, eller "linjer" basert på hvor mye IR-lys som reflekteres fra underlaget (Figur 9).

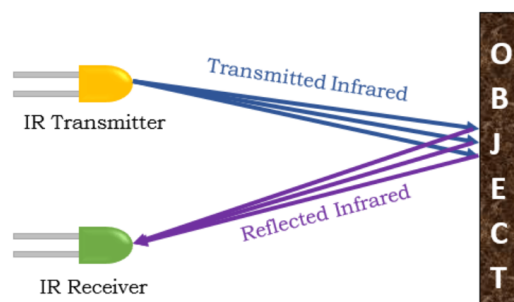


Figure 9: IR transmittor og mottaker

##### Linjesensor

IR-sensorene foran på bilen kan oppfatte 'fargen' på gulvet. Disse kan vi bruke til å få bilen til å 'se' hvor den mørke teipen er i forhold til bilen, ved å se hvilken av sensorene som mottar minst IR-lys.

**OBS!** Første gang vi bruker funksjonen må bilen være plassert på en linje for å lære seg forskjellen på linja og gulvet, altså må sensorene kalibreres. Dette kan gjøres ved å sende en kommando til bilen.

##### Bilens oppbygning

For å gjøre oppgaven litt enklere og mindre teknisk, er bilen allerede utstyrt med all nødvendig kode for å følge linjene. Det vil si at kode for å bruke og kalibrere de 6 IR-sensorene, og en PID-regulator allerede finnes i bilen. For å bruke disse funksjonene, er det bare å sende kommandoer fra kommando-blokka i simulink dere har brukt tidligere.

Bilene lagrer ikke nye parametre dere sender til den, og verdiene vil derfor nullstilles ved strømtap. Det er derfor viktig at dere skriver ned parametre dere finner som fungerer.

##### Simulink-blokker

Dere vil trenge noen nye blokker som ikke er brukt i de tidligere oppgavene.

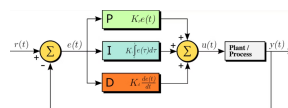
- **Linjedetektor.** Denne blokken kan brukes for å se hva bilen leser fra linjesensoren. Figur 10a.
- **PID parametre.** Denne blokken brukes til å sende P, I og D parametre til bilens innebygde PID-kontroller. Figur 10b.
- **Send-speed.** Denne brukes for å justere farten til bilen. Under PID-regulering vil standardfarten være lik farten satt til høyre belte. Figur 10c.



Command blokken sender kun kommando ved endring av kommando i menyen. Om det virker som bilen ikke mottar en kommando, prøv å bytte sendt kommando, og bytt tilbake.



(a) Linjesensor



(b) PID parametre



(c) Fartjustering

Figure 10: Simulink blokker til linjefølging

---

## Oppgaven

For å se linjen, brukes linjesensoren utstyrt på bilene. Det kan være lurt å ha en idé om hvordan disse målingene ser ut. Bruk et scope i simulink for å visualisere dataen mens dere beveger sensoren over linjen. Dette kan gi en idé til vilken orden dere bør ha regulerings-parametrene i, og kan verifisere at sensorene er kalibrert.

Til slutt skal bilen reguleres. Bruk simulinkblokkene som beskrevet i bakgrunnsdelen. Det kan være en prosess å finne parametre som får bilen til å kjøre bra. Husk at farten dere velger å kjøre i også påvirker reguleringen.

### Oppgave 3 - Linjefølging

1. Lag en bane som kan brukes under testing.
2. Les linje-data fra bilen
3. Sett inn og koble opp Simulink-blokkene som beskrevet
4. Juster PID-parameterne for bilen slik at bilen holder seg stabilt på linjen
5. Optimaliser både fart- og styringskontroll for raskest mulig banetid.

**Mål:** Raskest mulig gjennom hele banen!

- Tiden måles manuelt
- Krav: Bilen skal følge linja fra start til slutt.
- Tips: Juster hastigheten bilen kjører i i PID-modus.
- Tips: Husk at line-sensoren må kalibreres over linjen for å lese riktig



#### Læringsmål for oppgaven:

- Utforske utfordringer og oppleve mestring ved styring av et fysisk system
- Erfare hvordan regulering fungerer i en reell, praktisk situasjon
- Anvende kunnskap fra tidligere oppgaver i en praktisk sammenheng