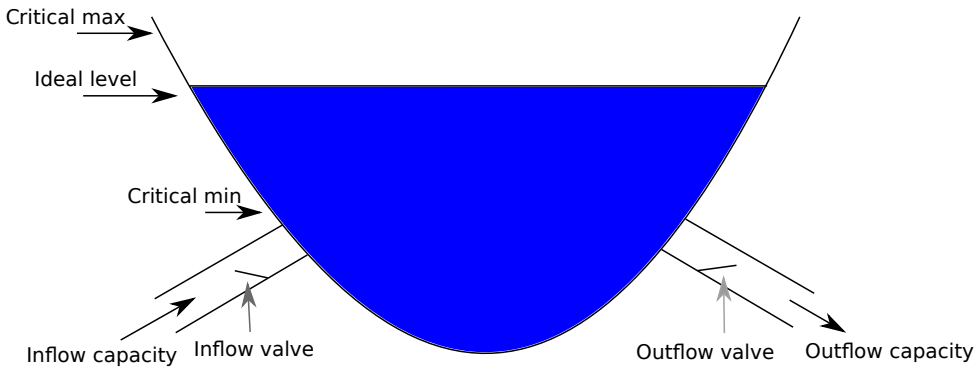
Source of Life

I denne oppgåva ser vi på ein simuleringsmodell av eit system for styring og overvaking av demningar som gir vatn til det kommunale vassanlegget.



Figur 1: Schematic of a dam with the model parameters included in our simulation model

Hensikten med systemet er å overvaka vasstanden til demningar og justera inn- og utstraumdragsventilar etter behov for å halda vatnet på eit ideelt nivå. Som ei forenkling, anta at inn- og utstraumdragsrøyra aldri går heilt tom og derfor kan levera eller forbruka så mykje vatn det er behov for. Dei har derimot ein avgrensa flytkapasitet. Dette betyr at berre ei avgrensa mengde vatn kan strøyma inn eller ut frå demninga på eit gitt tidspunkt, og det tek derfor tid å retta opp igjen vasstanden til normalen etter deteksjon av eit overskot eller mangel på vatn. Figur 1 viser skjematisk ein dam med nemnde modellparametrar.

I staden for målingar frå faktiske sensorar, er modellen her eit simuleringsscenario bestående av ein serie av diskrete tidspunkt. For kvart tidspunkt kan simuleringa motta sensoravlesninger som indikerer at vasstanden til ei demning har endra seg, og den kan då utføra ventiljusteringar for å bringa vassnivået i systemet nærare det ideelle nivået igjen. [1]

Model parameters

Dette avsnittet beskriv i meir detalj parametrane over, og presist kva kontekst dei blir brukte i (jf. Figur 1).

Inflow capacity How much water that can flow into a dam during a timestep

Ideal level The ideal level of the dam. When the simulation starts, the dams are at this level.

Critical high The critical high level of the dam. An alarm is raised if the dam fills above this level.

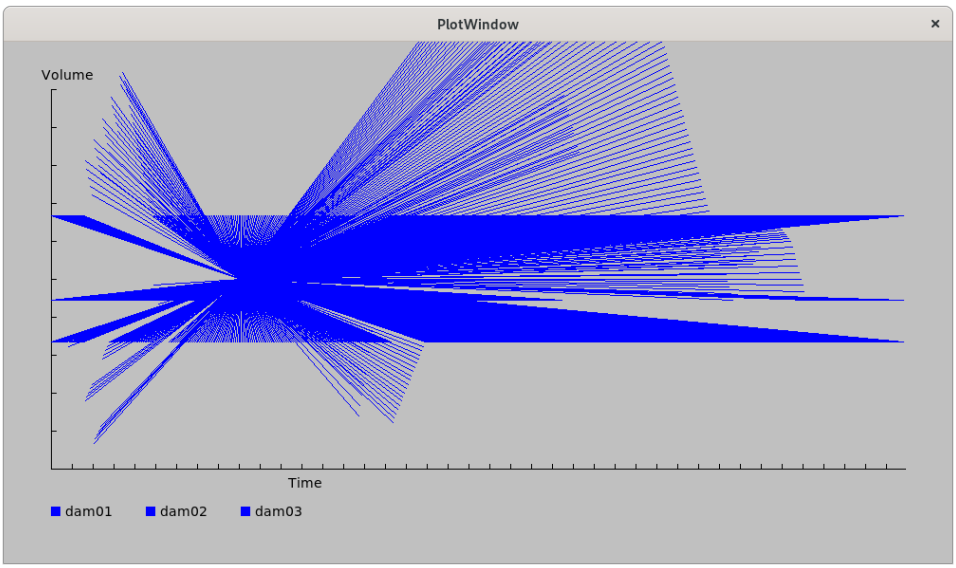
Critical low The critical low level of the dam. An alarm is raised if the dam empties below this level.

The three different dams to simulate in this exam have parameters as shown in Tabell 1.

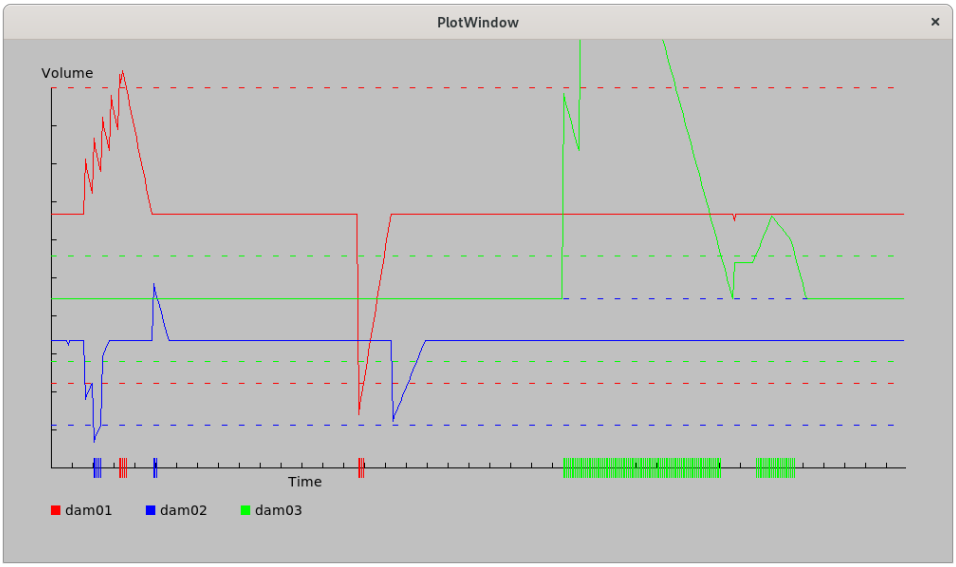
Name	Ideal level	Critical high	Critical low	Outflow capacity	Inflow capacity
dam01	1200	1800	400	40	50
dam02	600	800	200	30	20
dam03	800	1000	500	30	40

Tabell 1: Parameters for the simulated dams.

[1] Dette liknar eit verkeleg distribusjonsystem der det kjem ekte sensoravlesningar med ein bestemd frekvens.



Figur 2: The initial plot produced by the handed-out code. This is not very useful.



Figur 3: An example of a plot produced by the code after the plotting (P) functions have been implemented. The colored lines intersecting the x-axis indicates alarms for levels exceeding the critical maximums and minimum limits. The dashed horizontal lines indicate the limits.

Implementation overview

Implementasjonen av denne oppgåva er delt inn i fleire ulike filer, men du treng berre å endra nokon av dei for å løysa oppgåvene. Ein betydeleg del av koden er berre eit stillas som trengs for å få programmet til å køyra. Hugs dette, då det ikkje er lurt å bruka tid på å forstå heile kodebasen, sidan det ikkje er nødvendig for å kunna løysa oppgåva. For å gjera det lettare å navigera i koden vil kvar enkelt deloppgåve fortelja deg kva filer du treng å endra og kva deklarasjonar du treng å forstå for å løysa den deloppgåva.

Første gong du kompilerer og køyrer programmet skal du sjå vindauget vist i Figur 2. Det er klart at plottet ikkje er veldig nyttig, men ved å løysa oppgåvene, vil du gradvis implementera dei manglande delane av programmet og jobba mot eit ønskt resultat som vist i Figur 3. Det er ikkje eit krav for å bestå eksamen at all innlevert kode faktisk køyrar. Vi anbefaler likevel at du testar svara dine for å stadfesta at dei er korrekte.

Dei individuelle deloppgåvene som utgjer heile denne tredje oppgåva viser alle kva kildefiler som må

endrast for å løysa den deloppgåva. I kildefilene vil du finna deloppgåvene merka med ein kommentar:

`// Assignment XX`

følgd av skildringa av oppgåva (på engelsk). Desse skildringane er identiske med tilsvarande omsette skildringar du finn i dette dokumentet. Du vil òg sjå kommentarar som seier

`// Used for assignment XX.`

Desse kommentarane blir brukte til å trekkja merksemda di mot implementasjonsdetaljer, t.d. deklarasjonar, som er nødvendige for å løysa ei bestemd oppgåve. Til slutt, i kvar oppgåve er det eit matchande par kommentarar

`// BEGIN: XX` og `//END: XX`.

Det er veldig viktig at alle svara dine er skrivne mellom slike kommentar-par, for å støtta sensurmekanikken vår. Viss det allereie er skrive nokon kode mellom BEGIN- og END-kommentarane i filene du har fått utdelt, så kan, og ofte bør, du erstatta den koden med din eigen implementasjon.

Til dømes, for oppgåve P1 ser du følgjande kode i utdelte `plotter.cpp`

```
Color::Color_type GraphColor::next_color()
{
  // BEGIN: P1
  (void)cur_color;
  return Color::black;
  // END: P1
}
```

Etter at du har implementert løysinga di, bør du enda opp med dette i staden for

```
Color::Color_type GraphColor::next_color()
{
  // BEGIN: P1

  // Your code here

  // END: P1
}
```

Merk at BEGIN- og END-kommentarane **IKKJE skal fjernast**.

Til slutt, viss du synest nokon av oppgåvene er uklare, si korleis du tolkar dei og skriv korleis du antek at oppgåva skal løysast som kommentarar i den koden du sender inn.

Theme 1: Plotting

Alle oppgåvene i denne Plotting-delen gjeld plottfunksjonaliteten implementert i filene `plotter.h` og `plotter.cpp`

3.1 P1: next_color

Implementer funksjonen `next_color` slik at kvart kall til funksjonen returnerer neste farge frå vektoren `colors` definert i klassen `Color` (`plotter.h`). Når det ikkje er fleire fargar i lista, skal funksjonen kaste eit unntak med ei beskrivande melding. Bruk den private klassemedlemsvariabelen `cur_color` for å halde oversikt over gjeldande posisjon i `colors` vektoren.

Eksempel: Det første kallet til `next_color` returnerer `Color::red` som er den første i vektoren `colors`. Det andre kallet returnerer `Color::blue` som er nummer to i lista, og så vidare.

3.2 P2: Random color

Klassen `GraphColorRandom`, definert i `plotter.h`, arvar frå klassen `GraphColor` og endrar funksjonaliteten slik at dens `next_color`-medlemsfunksjon no returnerer fargane i ei tilfeldig rekkjefølgje. Merk at kvar farge framleis berre skal returnerast ein gong. Implementer konstruktøren til `GraphColorRandom`-klassen slik at den tilfeldig endrar rekkjefølgja i `colors`-vektoren når klassen blir instansiert.

For å teste implementasjonen din, endre instansieringa av `GraphColor colors` i klassedefinisjonen av `Plotter` i `plotter.h` som angitt i den fila.

Implementer `GraphColorRandom` klassen.

Hint: Ein vanleg algoritme for å stokke ei liste tilfeldig er Fisher-Yates[2] attgitt nedanfor:

```
for i from 0 to n-2 do
    j ← random integer such that i <= j < n
    exchange a[i] and a[j]
```

3.3 P3: get_color

Implementer funksjonen `get_color` som tek namnet til ein dam som parameter og returnerer plottfargen som er tilordna den dammen. Funksjonen skal slå opp fargen i klassemedlems-map `dam_colors` definert i `plotter.h`. Viss ei demning ikkje finst i `dam_colors`, skal funksjonen opprette ein plass for demninga og tilordne ein farge ved å bruke `colors.next_color()`-funksjonen. `colors` er `GraphColor`-instansen som er definert i klassen `Plotter` i `plotter.h`

3.4 P4: Track last point

Plottene av vasstanden til dammane er samansett av ein serie med linjesegment. For kvart tidssteg får vi berre den noverande verdien av ein gitt dam. Derfor må vi følgje med på kor linjesegmentet for førre tidssteg stoppa for å vite kor vi skal starte linjesegmentet for gjeldande tidsperiode. Dette er formålet med `get_last_point`-funksjonen.

Implementer funksjonen `get_last_point` som returnerer det punktet der førre linje for demninga stoppa. Funksjonen tek to parametrar; namnet på dammen og det neste punktet som skal plottast. Alle slike "siste-punkt" skal lagrast i klassemedlems-map `last_points` definert i `plotter.h`.

`last_points` knyter namnet til ei demning til det siste punktet som førebels er plotta for den demninga. Viss det ikkje finst noka oppføring i `last_points` for demninga, skal funksjonen opprette eit nytt innslag for dammen og returne punktet `(0,0)`. Viss dammen finst i konteinaren frå før, skal funksjonen heller returnere det førre punktet og oppdatere `last_points` med det neste punktet som no blir teikna.

3.5 P5: Dam alarms

Viss ei demning hamnar i ein kritisk tilstand, dvs. at vasstanden går utover dei kritiske maksimums- og minimumsgrensene, ønskjer vi å visualisere dette spesielt i grafen.

[2] `https://en.wikipedia.org/wiki/fisher%E2%80%93Yates_shuffle`

Implementer funksjonen `plot_alarm` som plottar ei vertikal linje som kryssar x-aksen til grafen (som vist i Figur 3) for å indikerer at det oppstod ein vasstands-alarm. Bruk funksjonen `get_color` frå oppgåve P3 for å fargeleggje linjene med riktig farge.

Funksjonen `plot_alarm` tek tre parametrar: Namnet på dammen alarmen vart utløyst for, tidspunktet som alarmen oppstod på, og typen alarm. Verdien `timestep` blir brukt til å berekne x-koordinaten for plassering av linja. Legg merke til at funksjonen `calc_x` finn riktig verdi på x-aksen til koordinatsystem gitt eit `timestep`. Alarmen blir gitt opp som ein parameter av typen `Dam::Wateralarm`, ein `enum` definert i `dam.h`. Den private medlemsvariabelen `y_len` inneheld høgda på y-aksen. Du kan bruke dette til å bestemme høgda på linja.

Funksjonen `plot_alarm` blir kalla på kvart `timestep`, og verdien av parameteren `alarm` er returverdien for funksjonen `Dam::get_alarm` (frå oppgåve D2) i `dam.cpp`. Du skal berre teikne "alarm-linja" viss verdien til alarmparameteren indikerer at det oppstod ein alarm på oppgitt `timestep`. Merk at vår default implemenasjon av `get_alarm`-funksjonen produserer nokre heilt tilfeldige alarmar. Derfor kan du bruke `get_alarm` til testing no, sjølv om du ikkje har løyst oppgåve D2 enno.

Hint: For å få ei linjeplassering som kryssar x-aksen (som vist i Figur 3) kan du teikne linjer med start- og sluttpunkt

`(calc_x(timestep), y_len + 40)` og `(calc_x(timestep), y_len + 60)`

høvesvis.

Denne oppgåva skal implementerast i fila `plotter.cpp`.

3.6 P6: Dam limits

For å gjere det enklare å sjå når vasstanden til ein dam overskrid grensene for maksimal eller minimal vasstand, ønskjer vi òg å visualisere desse grensene i sjølve grafen.

Implementer funksjonen `plot_limit` som teiknar ein stipla horisontal linje på grafen (nokre døme vart viste i Figur 3). Funksjonen tek inn namnet til demninga og ei grense som parametrar, og den kallast når dammane blir initialisert. Hugs at du kan rekne om ein verdi til ein y-koordinat ved hjelp av `calc_y`-funksjonen, og at `calc_x`-funksjonen kalla med verdiar mellom 0 og `timesteps` vil generere alle dei moglege x-koordinatene. Teikn linjene i den fargen som blir returnert av `get_color`-funksjonen. Til slutt, hugs å leggje til dei `Shapes` du teiknar til `shapes`-vektoren slik at dei blir deallokert når klassen `Plotter` går ut av skopeet.

Theme 2: Dam models

Oppgavene i Dam-klassen dreier seg om dam-modellen implementert i fila `dam.cpp`. Sjå òg definisjonen av klassen `Dam` i `dam.h`

3.7 D1: Water level delta

Implementer funksjonen `get_water_level_delta` som returnerer forskjellen mellom noverande og ideell vasstand. Funksjonen skal returnere den absolutte verdien av delta (avstanden), så du kan bruke funksjon `abs` til det. Hugs at dagens vasstand og den ideelle vasstanden blir lagra i medlemsvariablane `water_level` og `ideal_level` høvesvis, i klassen `Dam`. Klassen `Dam` er definert i `dam.h`.

3.8 D2: Water level alarms

Implementer funksjonen `get_alarm` som indikerer til kalleren om vasstanden til demninga er over det kritisk høge nivået eller under det kritisk låge nivået. Funksjonen skal returnere ein av verdiane som er definerte i `Wateralarm` `enum` i `dam.h`. Viss vasstanden er innanfor dei kritiske grensene, `dam.h`. Hvis vannstanden er i mellom de kritiske grensene, skal funksjonen også da returner en passende verdi fra samme `enum`.

Husk at reell vasstand, kritisk høg-, og kritisk låg-nivåa er lagra i dei private medlemsvariablane `water_level`, `water_level_max` og `water_level_min`. Desse variablane er alle definerte i `dam.h`.

3.9 D3: Water level restoration

Viss vasstanden går utanfor dei kritiske grensene for demninga må systemet setje i verk tiltak for å rette opp igjen vasstanden til det ideelle.

De fysisk dam begrensningene er beskrive i innleiinga. Mengda vatn som kan strøyme inn eller ut av dammen per tidseining finst i klassemedlemsvariablane `outflow_capacity` og `inflow_capacity`. Dette betyr at det ofte tek tid før vasstanden kan komme heilt tilbake til ideelt nivå igjen. Merk at viss forskjellen mellom faktisk og ideelt vannivå er mindre enn innstraumdraget eller utstraumdragskapasiteten kan du stille vasstanden til ideell direkte. Du kan bruke `water_level_delta`-funksjon definert i oppgåve D1 for å sjekke dette.

Implementer funksjonen `restore_water_level` som rettar opp igjen vasstanden i demninga viss den er over eller under det ideelle nivået.

Theme 3: File input/output

Oppgavene i denne fila handlar om å lese og skrive frå og til tekstfiler i komma-separerte verdiar (CSV) filformat. For å løyse oppgåvene må du endre filene `damsim.cpp` og `sensors.cpp`.

3.10 F1: CSV file output

Løsningen for denne oppgåva skal implementerast i fila `damsim.cpp`.

Skriv ein funksjon som skriv simuleringsdata i komma-separerte verdiar (CSV) filformat til filstrømmen `output_file`. Skriv linjene til filstrømmen på følgjande format:

```
<dam name>,<current timestep>,<current water level>
```

Du kan få namnet på demninga og noverande vasstand frå funksjonane `get_name` og `get_water_level` frå Dam-objektet-referanse parameter v. Sjå deklarasjonen av `Dam`-klassen i `dam.h` for meir detaljar. Gjeldande `timestep` blir sendt inn i parameteren `cur_timestep`.

3.11 F2: CSV file reading

Foreløpig er sensoravlesningene som blir brukt i simuleringa hardkoda i konstruktøren til klassen `SensorsFile` i `sensors.cpp`.

```
timestep,dam,inflow,outflow
10,dam02,0,40
10,dam01,40,0
20,dam01,300,0
20,dam02,0,300
25,dam01,300,0
25,dam02,0,300
...
```

Figur 4: Excerpt from the sensor data CSV file `sensor_data.csv`.

I denne oppgåva gjer vi dam-simulatoren meir fleksibel ved å utvide ho med moglegheita til å lese sensordata frå enkle CSV-filer[3]. Fleire slike enkle CSV-filer følgjer med denne oppgåva i `data/`-mappa. Sensordatafilene heiter `sensor_dataX.csv` der X er eit heiltal.

Figur 4 viser dei første linjene i fila `sensor_data1.csv`. Som du kan sjå, er formatet som følgjer: Den første linja i fila inneheld namna på dei komma-separerte kolonnane. I dette tilfellet er første kolonne `timestep`, den andre er `dam`-id, osb. De følgjande linjene inneheld faktiske avlesningar. I dette dømet finst den første avlesninga på `timestep` 10 for dam02, med en `inflow` på 0 og `outflow` på 40.

Implementer konstruktøren til klassen `SensorsFile` i fila `sensors.cpp`. Funksjonen skal lese inn frå filnamnet som blir sendt som parameter `fname` til konstruktøren. For kvar linje du les inn, skal du kalle funksjonen `insert_reading` for å registrere avlesninga. Vi forventar ikkje at du skal fokusere på unntaks-handtering som til dømes feil i filformatet. Du kan derfor anta at funksjonen din berre vil bli brukt til å analysere filer som er heilt i samsvar med formatet beskrive ovanfor.

[3]Merk at CSV-filene du møte i verkelegheita kan vere mykje meir komplisert enn desse.

Figur 5: An example of what the simulation plot with randomized sensor data might look like.

Theme 4: Class inheritance

Oppgavene i denne delen blir løyst ved å endre filene `sensors.h` og `sensors.cpp`. I tillegg, for å teste implementasjonen din så må du endre template-parameteren i instansieringen av `DamSim` i `main.cpp`. Den relevante plasseringa i `main.cpp` er markert. Du må òg hugse å fjerne kommentar-teikna på linja nedst i `damsim.h` som anvist.

Default-implementasjonen som er delt ut bruker ein førekomst av klassen `SensorFile` for å få sensoravlesninger. Denne fila implementerer den abstrakte klassen `Sensors` og les sensordata frå ei fil. I denne delen av eksamen ber vi deg om å implementere ein sensorlesargenerator som genererer tilfeldige sensordata i staden for å lese dei frå ei fil.

Ein `Sensor` har berre ein funksjon: `get_reading`. Funksjonen `get_reading` returnerer ein `Reading struct` (definert i `sensors.h`) som inneheld målingane for oppgitt `Dam`/`timestep`. Viss det ikkje er nokon sensorverdiar å lese for ein gitt `timestep` og `Dam`-kombinasjon, så skal funksjonen berre returnere ein `Reading struct` med begge verdiane sett til 0.

Når du har implementert alle delspørsmåla i denne delen kan du ende opp med eit plot som liknar på Figur 5. Resultatet ditt vil sjølvsagt avvike litt frå dette, avhengig av kva algoritme du velger for å generere tilfeldige avlesninger i `get_reading`-implementasjonen.

3.12 S1: Virtual class declaration

I `sensors.h`, skriv ein klassedeklarasjon for `SensorsRandom`. Klassen skal innehalde deklarasjonar for funksjonane frå den abstrakte base-klassen `Sensors`. Implementer ein delegerande konstruktør som kallar konstruktøren til den virtuelle klassen `Sensors`.

3.13 S2: Generating reading

Implementer funksjonen `get_reading` definert i `SensorsRandom`-klassen. Funksjonen får det gjeldande tidspunktet og namnet på dammen som parametrar og returnerer ein struct `Readings` (definert i `sensors.h`) som inneheld sensorverdiane.

Du står fritt til å bestemme logikken for denne funksjonen, den må likevel returnere fornuftige verdiar for innstrømming og utstrømmming generert ved hjelp av `rand ()`-funksjonen. Til dømes, ei enkel tilnærming vil vere å returnere tilfeldige verdiar mellom 0 og 100 for innstrømming og utstrømmming kvart 10. timestep.

Slutten på denne eksamen.