Beskrivelse af "Sisyfos lite"

Jeppe Schultz Nielsen

Maj 2023

Contents

1 Introduktion

Dette dokument beskriver kode, der kan findes på `https://github.com/JeppeSchultzNielsen/REO-Sisyfos-lite`, som er et forsøg på en rekonstruering af Energistyrelsens beregningsmodel Sisyfos. Beskrivelsen er ikke 100% udtømmende, så forventningen er at den fungerer som opslagsværk mens koden læses. Kun de komplicerede aspekter af koden bliver beskrevet – det vil sige at eksempelvis getter-metoder bliver ignoreret.

1.1 Sisyfos

Sisyfos er en beregningsmodel, der simulerer et netværk af "noder" der kan producere, forbruge og udveksle energi. Den er skabt til at undersøge effektsikkerheden i de allermest kritiske timer. Den gør derfor nogle antagelser, som f.eks. at ignorere fleksibelt forbrug, fordi fleksibelt forbrug ikke spiller en rolle i de allermest kritiske timer, fordi elprisen ville være høj nok til at fleksibelt forbrug er slukket. Sisyfos er i Energistyrelsens ord udelukkende end "fysisk-teknisk model" og kan derfor ikke modellere elpris eller ellagre (s. 4 i baggrundsrapporten).

2 Quickstart guide

Projektet kan downloades fra github på flere måder. Man kan (forudsat, at man har git installeret på ens computer) åbne en terminal, bevæge sig ind i mappen man ønsker at placere koden i ved brug af "cd" kommandoen, og så skrive

```
git clone https://github.com/JeppeSchultzNielsen/REO-Sisyfos-lite.git
```

Alternativt kan en editor som Visual Studie Code klone fra et git repository. Filen TVAR.csv er for stor til at uploade på github, og må i stedet downloades herfra: `https://drive.google.com/file/d/1yB4Ja9xugb-3ada8ennEZ5hWPANo3EFw/view?usp=sharing`. Den skal så placeres i mappen data.

De fleste pakker til Python, der er nødvendige for at køre koden, følger med, når man installerer python gennem Anaconda. Pulp, der bruges til at løse det lineære optimeringsproblem, gør muligvis ikke. Hvis koden ikke vil køre, fordi pakken mangler, kan den installeres ved at skrive

```
pip install pulp
```

fra terminalen. Det kan være nødvendigt at køre kommandoen

```
pulptest
```

for at garantere, at solveren virker (så vidt jeg husker var det ikke nødvendigt for mig). Konfigurationerne, som koden køres under, bestemmes i filen main.py, som altså skal redigeres vil man køre simuleringen under andre forhold. Denne kan eksempelvis se sådan ud:

```
from Simulation import Simulation
from DataHolder import DataHolder
from Options import Options

options = Options(2035,1985)
options.usePlannedDownTime = True
options.useUnplannedDownTime = True
options.energyIslandEast = True
options.energyIslandWest = True
options.tyndpYear = 2030
dh = DataHolder(options,"data/outage/Plan2035_1985_var.csv")
sim = Simulation(options, dh,"test.txt", True)
sim.RunSimulation(0,1000)
```

De første 3 linjer importerer kode fra andre steder i projektet og er altid nødvendige. Den fjerde linje laver et Options-objekt, hvor vi fortæller, at simuleringsåret er 2035 og klimaåret er 1985. De 5 efterfølgende linjer sætter nogle options (alle disse er faktisk sat sådan som default - her demonstreres bare, at det er muligt. Der er også flere muligheder tilgængeligt, se sektionen med Options-klassen). Efter dette laves et DataHolder-objekt, der læser og opbevarer de store datafiler, der skal bruges i simuleringen. Denne tager som parameter de options, der er blevet sat, og en adresse til den udetidsplan, der skal bruges (udetidsplaner tager lang tid at generere. Jeg har allerede lavet et par stykker med udgangspunkt i klimaåret 1985. Jeg tror ikke, at der er store konsekvenser ved at bruge disse i andre klimaår også). Så laves et Simulation-objekt, der tager options og dataholder som argumenter, og et navn på den fil, resultatet skal skrives til. Når kommandoen RunSimulation(0,1000) bliver kaldt på denne, køres simuleringen fra timen 0 til timen 999 – hvis man vil køre et helt år skal man altså skrive RunSimulation(0,8760). Så skal main.py køres. Dette kan man også ofte gøre gennem en editor, eller man kan navigere i terminalen til mappen hvori main.py ligger, og skrive

```
python main.py
```

Bemærk at mappen, som resultaterne skal gemmes i, skal eksistere (det vil sige, koden vil ikke kreere en mappe for dig, men smider i stedet en fejl).

Main.py kan eksempelvis konfigures til at køre det samme klimaår mange gange. Så er det ikke nødvendigt at lave et nyt DataHolder objekt hver gang, men man kan i stedet eksempelvis skrive:

```
from Simulation import Simulation
...
dh = DataHolder(options,"data/outage/Plan2035_1985_var.csv")
for i in range(40):
    sim = Simulation(options, dh,f"results/test/test{i}.txt", True, False)
    sim.RunSimulation(0,8760)
```

Det er altid bedst at re-initialisere Simulation-objektet, for ellers får man "tilstanden", den sidste simulering befandt sig i, som startstilstanden for den næste (det vil sige de værker, der er havareret i slutningen af tidligere simulering, er havareret i starten af den næste). Hvis man simulerer flere forskellige simuleringsår/klimaår er det selvfølgelig nødvendigt at lave nye Options og DataHolder objekter.

2.1 Et andet eksempel på en main fil

Her er en main-fil, der kører det samme simuleringsår flere gange på samme klimaår og postprocessor resultatet (det vil sige, sætter produktion, der ikke blev brugt, til 0)

```
1  from Simulation import Simulation
2  from DataHolder import DataHolder
3  from Options import Options
4  from PostProcessor import PostProcessor
5
6  options = Options(2035,1985)
7  options.usePlannedDownTime = True
8  options.useUnplannedDownTime = True
9  options.energyIslandEast = True
10 options.energyIslandWest = True
11 options.useVariations = True
12 options.tyndpYear = 2030
13 options.prioritizeVariableProduction = True
14 dh = DataHolder(options,"data/outage/Plan2035_1985_noVar.csv")
15
16 for i in range(2):
17     sim = Simulation(options, dh,f"results/2035_1985_{i}.txt", True, False)
18     sim.RunSimulation(0,8760)
19     pp = PostProcessor()
20     pp.process(f"results/2035_1985_{i}.txt",f"results/2035_1985_{i}_pp.txt")
```

På linje 11 er prioritizeVariableProduction sat til true, hvilket betyder, at modellen maksimerer brugen af uregulerbare kilder. På line 14 gives udetidsplanen "data/outage/Plan2035_1985_noVar.csv". Dette betyder at det er en udetidsplan lavet til 2035, baseret på klimaåret 1985, hvor der ikke er taget højde for variabel produktion, da udetidsplanen blev lavet (se eventuelt sektionen om CreateOutagePlan). Bemærk, at hvis jeg giver en adresse til en outagePlan, der ikke eksiterer, laver programmet selv en udetidsplan og gemmer den på denne adresse. Dette er en meget tidskrævende proces (ca. 2 timer). På linje 17 giver jeg først Simulation objektet parameteren True, fordi den skal gemme outputtet, og False fordi den ikke skal gemme en diagnostikfil, hvor der står hvad parametrene for simuleringen har været.

3 Overordnet beskrivelse

Koden består af følgende filer, der med undtagelse af main.py indeholder klasser:

- Area.py
- AreaDK.py
- DataHolder.py
- Line.py
- main.py
- NonTDProduction.py
- Options.py
- Production.py
- Simulation.py
- PostProcessor.py

Filen main.py indeholder koden, der skal køres når en simulering skal startes. Den instantierer objekter af "Options", "DataHolder" og "Simulation" (og disse instantierer efterfølgende objekter af de andre klasser). Main.py er tiltænkt at være et sted hvor man hurtigt kan ændre på parametre for den simulering man skal køre, så for at vælge f.eks. simuleringsår redigerer man denne fil.

Options.py er en lille klasse, der holder information sat af brugeren, som definerer simuleringen, f.eks. om energiøerne medregnes, om der skal bruges stokastisk og planlagt nedetid for værkerne, og lignende. Den bliver givet som parameter til alle andre objekter i simuleringen, som så kan tilgå informationen.

DataHolder.py indeholder en klasse, der er ansvarlig for at åbne de større data-filer, der bruges i simuleringen (TVAR.csv og udetidsplanen) og holde på denne data, så filerne kun skal læses én gang og den relevante information kun lagres i ét objekt. DataHolder-objekter bliver instantieret i main.py, og så givet

som parameter til alle andre objekter i simuleringen, som tilgår informationen derved. Man kan også bruge det samme DataHolder objekt til flere simuleringer, hvis man ønsker at lave den samme simulering flere gange for at se på den stokastiske effekt.

Simulation.py indeholder en klasse, der er ansvarlig for at instantiere alle "Area"-objekter (der repræsenterer noder, eller el-områder) og "Line"-objekter (der repræsenterer transmissionslinjer mellem områderne), at køre simuleringen og at skrive resultaterne til en ekstern fil.

Line.py indeholder klassen der repræsenterer transmissionslinjer i simuleringen.

Area.py indeholder klassen der repræsenterer noder (områder med forbrug og produktion, eksempelvis Vestdanmark) i simuleringen. Den instantierer "Production"-objekter, der repræsenterer de forskellige typer af produktioner i området (f.eks. gas, vind, sol), men også repræsenterer forbrug, som er produktion med omvendt fortegn. AreaDK.py er en specialisering af Area klassen, der gælder specifikt for DK-Øst (DK2) og DK-Vest (DK1) som simuleres anderledes end andre områder i simuleringen.

Production.py indeholder klassen, der repræsenterer en specifik type af produktion, og holder informationen for denne type, så det kan tilgås af Area-objektet, denne produktion tilhører. NonTDProduction (non-time-dependent production; produktion, der ikke følger en variationskurve fra TVAR.csv) arver fra denne, men indeholder også funktioner, der gør det muligt for produktion at have nedetid (hvilket er antaget er være "indbygget" i variationskurverne for de produktioner, der følger TVAR.csv).

PostProcessor.py indeholder klasserne PostProcessor og Node. PostProcessor kan løbe en output-fil fra simuleringen igennem og sætte al unødig produktion lig 0, baseret på en prioriteret liste. Node er en hjælpeklasse til dette formål.

4 Input-data

En række data-filer er blevet lavet baseret på filen Sisyfos5data26.xlsx. Dataen er blevet opsplittet i flere forskellige filer for nemmere at overskue, hvilken data er hvor. Jupyter notebooken namefixer.ipynb er blevet anvendt på alle disse filer for at sikre konsistent navngivning af nodes på tværs af filer (Energistyrelsen har ikke været helt konsistente, men det er ikke et problem i excel, hvor opslag i tabeller ikke laves baseret på navne, men "koordinater" i arket).

4.1 TVAR.csv

Denne fil indeholder timeserier for forbrug, vindproduktion, solproduktion mm. for de klimaår, Energistyrelsen har brugt i deres model. Det er nødvendigt at "fixe" den med namefixer, for at den skal virke med Sisyfos-lite.

4.2 demands.txt

Denne fil indeholder det årlige forbrug i TWh for alle noder (bemærk at DK1 og DK2 også optræder i denne, men det er ikke dette data, der bruges til at beregne deres forbrug) for alle mulige simuleringsår. Forbrugsdataen i TVAR.csv er normeret til at summere til 1.000.000 MWh over et år, så det er altså disse tal man skal gange med variationskurven for at få forbruget i MWh for hver time. Dataet i denne fil fandtes oprindeligt i tabellen på A8 i arket "Demand" i Sisyfos5data26.xlsx.

4.3 relativeDemands2025.txt og relativeDemands2030.txt

TVAR.csv indeholder forbrugsvariationer fra to ENTSO-E estimater, 2025 og 2030. Jeg er ikke sikker på at jeg fuldt ud forstår denne del, men afhængigt af hvilket estimat der bruges, ganger de en faktor på, som varierer med område og år. Dette data findes på A42 og AC42 i arket Demand.

4.4 DK1Demand.txt og DK2Demand.txt

Disse filer indeholder forbruget fordelt på forskellige typer af forbrug i Danmark i alle simuleringsår. Opdelingen i forskellige typer af forbrug er nødvendigt, fordi Sisyfos kan modellere det danske forbrug med forskellige grader af fleksibilitet. Hvor stor en andel, der maksimalt kan gøres fleksibel, står i rækken "Potentielt_fleksibel_andel". Den sidste række, "Pot_fleks_TWh", bruges ikke. Disse tabeller findes på henholdsvis AP9 og BH9 i arket Demand.

4.5 capacities.txt

Denne fil indeholder alle produktioner for alle år, samt hvilken node de hører til, hvilken variation de følger i TVAR.csv ("-" betyder at det er "nonTDProduction") og hvilken "type" de er. Typen afgører deres nedetider og antallet af enheder for en given kapacitet, defineret ud fra en fil der vil blive beskrevet efterfølgende. For alle andre nodes end DK1 og DK2 er denne data hentet fra tabellerne der starter på A24 i arket MAFTYNDP. For DK1 og DK2 er de hentet fra tabellerne i arket Analyseforudsætninger. Udover det er der reserver, der hentes fra en tabel på A67 i Capacities.

4.6 outageParams.txt

Denne fil indeholder parametre, der bestemmer hvor ofte produktionsenheder er nede som følge af havari eller planlagt udetid for hver produktionstype (den type, der blev refereret til i capacities.txt). De definerer hvor stor kapacitet, der skal til én enhed af den type (et værk er delt op i flere enheder, der havererer uafhængigt af hinanden). For disse typer er der også givet varmebinding og hvor lang tid havari tager i gennemsnit.

4.7 linedata****.csv

Disse filer indeholder information om capaciteter, udetider, og antallet af enheder for transmissionslinjerne i simulationen for årstallet, som er givet på selve filen (det er kun muligt at køre simulationen for disse årstal - men når Energistyrelsen gør andet, laver de også bare en lineær interpolering mellem disse år, som i forvejen ikke er særlig interessant). Dataet er fundet i Lines-arket, hvor jeg har kopieret for forskellige år ved at ændre på simuleringsårstallet i Scenarios-arket.

4.8 outagePlan

Denne mappe indeholder planer for planlagt udetid for alle værker i alle nodes i alle timer i et år som er genereret gennem CreateOutagePlan. Det vil sige at for hvert værk er der en kolonne med 8760 indgange (antallet af timer i et år), som indeholder et heltal der svarer til hvor mange enheder i værket der er tændt i den pågældende time. Det betyder at man skal generere nye udetidsplaner for hvert simuleringsår, da antallet af enheder ændrer sig når produktionskapaciteten gør – disse udetidsplaner kan dog selvfølgelig genbruges når samme simuleringsår køres igen senere. Hvordan udetidsplanerne genereres vil blive forklaret i Area-klassen.

5 Options

På tabel **??** findes en beskrivelse af de options, der kan sættes i Options-klassen. Standardværdierne for flexibilitetsparametrene findes på tabel **??**. De er givet i Sisyfos5Data.xlsx Demand-arket ved celle A190.

6 DataHolder

DataHolder-klassen er ansvarlig for at læse de store datafiler (TVAR.csv, udetidsplanen og capacities.txt), så hvert enkelt Area-objekt ikke er nødt til at åbne disse filer og gennemløbe dem selv. Et DataHolder-objekt skabes i starten af simuleringen og gives som parameter til alle klasser, som så kan tilgå den læste data. I DataHolder.py findes også hjælpeklasserne OutageParams og ProductionParams, der bare er til for nemt at kunne opbevare dataet for henholdsvis et enkelt type af værk eller specifikt værk.

6.1 __init__

Parametre:

```
op: Options, outagePlanPath: str
```

Constructoren for DataHolder-klassen kalder metoderne LoadProductionDict(), LoadOutageDict(), InitializeEmptyTimeSeries(), InitializeTimeSeries() og LoadOutagePlan().

Option	Standardværdi	Tilladte værdier	Beskrivelse
simulationYear	-	2025, 2030, 2035, 2040	Året som simuleringen skal simulere. Denne gives som parameter til constructoren.
climateYear	-	1985, 1987, 1996, 2007-2015	Klimaåret, som timeserierne skal tages fra. Denne gives som parameter til constructoren.
usePlanned Downtime	True	True/False	Om værker skal kunne gå ned som følge af planlagt udetid.
useUnPlanned Downtime	True	True/False	Om værker skal kunne gå ned som følge af stokastisk udetid.
demFlex****	se tabel **??**	0-1	Hvor stor fleksibilitet der er i forbruget fra transport, PtX, klassisk forbrug og CVP. Se mere i beskrivelsen af AreaDK-klassen.
storebaelt2	False	True/False	Fordobler kapaciteten på Storebæltsforbindelsen hvis den er true.
oresundOpen	False	True/False	Laver nogle ændringer på forbindelser i Øresund. Definitionerne er taget fra Lines arket i Sisyfos5data.xlsx.
energyIslandEast	True	True/False	Om den østlige energiø skal medtages i simuleringen. Hvis den er False, sættes øens kapacitet til 0, og alle linjer der burde forbinde med den får 0 kapacitet.
energyIslandWest	True	True/False	Som energyIslandEast.
prioritizeVariable Production	False	True/False	Om simuleringen skal forsøge at maksimere forbruget af variabel produktion (det vil sige produktion, der følger timeserier). Dette gør omtrent simuleringstiden 3 gange længere.

Table 1: Tabel over variable i Options-klassen.

Variabel	2025	2030	2035	2040
demFlexTransport	0	0.1	0.2	0.3
demFlexCVP	0.7	0.9	0.95	1
demFlexPtX	1	1	1	1
demFlexKlassisk	0	0	0.25	0.30

Table 2: Tabel over default-værdier for flexibilitetparametre afhængigt af simuleringsår.

6.2 LoadProductionDict

Denne metode læser capacities.txt og fylder variablen productionDict, hvor keys er navnet på nodes og values er lister af de (i capacities.txt) læste produktionsparametre. Når Area-objekter senere initialiseres kan de slå op i denne dictionary for at finde deres produktioner. Metoden løber filen igennem og gemmer kapaciteten for produktionen i det valgte simuleringsår samt andre parametre (navn, produktionstype, tilhørende variationskurve) i et ProductionParams objekt, der lægges ind i listen for den pågældende node.

6.3 LoadOutageDict

Denne metode læser outageParams.txt og fylder variablen outageDict, hvor keys er produktionstyper (eksempelvis GasUdland, HydroRes, DKV_affald) og values er OutageParams objekter, der holder parametrene for denne type (planlagt udetid, udetid på grund af havari, varmebinding, osv.). Når et værk initialiseres senere kan disse parametre så hentes nemt i outageDict.

6.4 InitializeEmptyTimeSeriesArray

Denne metode laver et 3d-array med den rigtige størrelse som variationskurverne så kan læses ind i. I dette array kommer første index til at svare til index af noden i listen names, og andet index kommer til at svare til produktionen som variationskurven svarer til (0. indgang er demand, derefter følger de som givet i productionTypes-listen). Det tredje index svarer til timen.

6.5 CreateProdNamesList

Denne metode tager som parameter en nodes navn, og returnerer en liste af strings afhængigt af simuleringsår og klimaår, der svarer til navnene på de kolonner i TVAR.csv, der er relevante for noden. Rækkefølgen af disse navne svarer til rækkefølgen givet i InitializeEmptyTimeSeriesArray.

6.6 InitializeTimeSeries

Denne funktion læser TVAR.csv. Først findes indexerne på kolonner, der skal læses. Dette gøres ved at loope over alle node-navne og generere de tilsvarende navne på produktioner i TVAR.csv ved hjælp af CreateProdNamesList. Hvis et produktionsnavn ikke blev fundet i TVAR.csv (eksempelvis findes der ingen DK1_CSP, men CreateProdNamesList laver en sådan string alligevel) skrives en advarsel. Når indekserne er fundet loopes der over alle timerne i året (det vil sige rækkerne i TVAR.csv), og de Arrays der blev initialiseret med InitializeEmptyTimeSeriesArray fyldes på de indekser, der blev fundet.

6.7 LoadOutagePlan

Først tjekker LoadOutagePlan om den givne path til en outageplan eksisterer som fil. Hvis den ikke ikke gør skabes en ny fil på den path der blev givet, og den første kolonne i den laves, som indeholder timetallet. Når Area-objekter bliver initialiseret tilføjer de så selv deres udetidsplaner til denne fil. Hvis udetidsplanen findes, læses først den første række, der indeholder navne på nodes og værker (dvs. der står "DK1", så alle værkerne i DK1, så "DK2"), og indekserne på kollonerne, der tilhører hvert land kan så gemmes. Resten af filen kan så læses og en 2d-matrix kan laves for hvert land, hvor man kan finde antallet af enheder i et værk der er tilgængelige i en given time.

7 Simulation

Her følger en beskrivelse af metoderne i Simulation-klassen, der kører selve simuleringen.

7.1 __init__

Parametre:

```
  options: Options, dh: DataHolder, asaveFilePath: str, asaving: bool
= True, saveDiagnosticsFile: bool = True
```

Constructor for Simulation-klassen. Denne initialiserer mange arrays, der gives i tabellen nedenunder (tabel **??**). Da der skrives til og laves opslag ofte i disse tabeller når simuleringen køres, er de lavet som numpy arrays i håb om en lille gevinst i effektivitet. Når disse arrays er blevet lavet laves en header til filen som output skal gemmes til, og, såfremt brugeren ønsker det (givet ved saveDiagnosticsFile-parameteren) gemmes der en diagnostics fil, hvor der står hvilke forudsætninger (options, kapacitet og demands faktorer for nodes og transmissionslinjer) simuleringen er lavet under.

7.2 InitializeLines

Denne metode læser linedata****.csv, hvor stjernerne står for simuleringsåret. Den giver hvert linje et unikt navn (hvis der eksempelvis er flere transmissionslinjer mellem DELU og DK1, vil de få navnene DELU_to_DK1, DELU_to_DK1_1, DELU_to_DK1_2 osv.). Og initialiserer et Line-objekt i linesList-arrayet med de parametre, der blev givet i linedata filen. Denne metode indeholder mange if-statements, der håndterer specielle scenarier. Disse scenarier kommer fra Sisyfos5data26.xlsx Lines arket, hvor de fleste værdier er hardcoded, men nogen altså afhænger af specifikke scenarier.

7.3 InitializeToFrom

Gennemløber listen af transmissionslinjer (linesList) og fylder arraysene fromIndeces, toIndeces, fromLength og toLength på de rigtige positioner, der er

Navn	Beskrivelse
areaList	Indeholder alle Area-objekter. Den initialiseres ved at loope over listen af navne som givet i DataHolder, og giver også områderne et indeks svarende til deres position i denne liste. DK1 og DK2 laves som specielle områder, AreaDK-objekter, der er en underklasse af Area.
demandList	Indeholder demand for alle nodes. Den bliver fyldt hver time, og er altså en af de arrays, der skal bruges når maxFlow problemet skal løses.
productionList	Indeholder produktion for hver node i den time, der skal løses.
linesList	Indeholder alle lines-objekter der skal bruges i simuleringen. Den initialiseres af InitializeLines-metoden.
transferList	Indeholder værdien af den energi, der er blevet ført på hver linje i den time, der sidst blev løst.
productionTypeNames	Er en 2d-list som indeholder navnet på alle værker, hvor det første index svarer til en node, og det andet index svarer til værket med det pågældende index i denne node.
productionTypeMatrix	Er et 2d numpy array som indeholder produktionen på alle værker i den sidst beregnede time, hvor det første index svarer til en node, og det andet index svarer til værket med det pågældende index i denne node.
fromIndeces	Et 2d-array hvor første index svarer til en node, og andet index svarer til indekset af alle "opsplittede" transmissionslinjer (se beskrivelsen af SolveMaxFlowProblem). Det er nødvendigt, når maxflow-problemet skal løses, at opdele hver transmissionslinje i to ensrettede linjer. Disse opslittede linjer opbevares i arrays, så lige indekser svarer til linjer der går "den rigtige vej" i forhold til hvordan de er defineret i linedata, og ulige indekser svarer til linjer, der går den modsatte vej. fromIndeces-arrayet indeholder altså indekser på de linjer, der går fra et land. Den initialiseres af metoden InitializeToFrom.
toIndeces	ligesom fromIndeces, men indekser på opslittede linjer, der går ind i landet.
fromLength	Et 1d-array, der indeholder antallet af linjer der for hver node er blevet sat ind i fromIndeces matricen.
toLength	Se fromLength.
alreadyUsed	Når simuleringen kører i flere skridt fordi option prioritizeVariableProduction er valgt skal alreadyUsed holde produktionen i hvert land, der allerede er eksporteret i de tidligere simuleringsskridt.
variableProductionList	1d-array der for hver node indeholder kapaciteten af variabel produktion i den pågældende time.

Table 3: Tabel over variable i Simulation klassen.

beskrevet i tabel **??**.

7.4 RunSimulation

Denne metode er ansvarlig for at loope over de timer, der skal simuleres. Den kaldes fra main.py efter Simulation-objektet er initialiseret. Der er to grene af funktionen, afhængigt om man vil maksimere variabel produktion. Hvis man ikke maksimerer variabel produktion (dette er altså den gren, som Energistyrelsen bruger) gør koden følgende for hver simuleringstime: kalder PrepareHour, kalder SolveMaxFlowProblem (parameter step 1), og kalder SaveData. Generelt tager denne sekvens, som altså simulerer en time og skriver den til en datafil, omtrent 44 ms på min computer, eller 385 sekunder for et år (8760 timer). Heraf bliver ca. 9 ms brugt på at kalde PrepareHour, 30 ms på SolveMaxFlowProblem (af hvilke 22 ms er kaldet til pulps solver - resten er at opdatere modellen, som pulp løser, med værdierne for den pågældende time), og ca. 5 ms på at skrive dataet til en fil.

Den anden gren tages hvis man gennem options har valgt at prioritere variabel produktion. Så gøres følgende trin:

- Kald PrepareHour som normalt.
- Kald SolveMaxFlowProblem med parameter step = 0. Her løses maxflowproblemet udelukkende med overskuddet fra variabel produktion. På den måde garanteres det, at underskud i udlandet præferentielt bliver dækket med uregulerbar produktion.
- Kald readjustParams, der forbereder på at bruge den gamle maxflow løsning i et nyt maxflowproblem.
- Kald SolveMaxProblem med parameter step = 1. Denne gør det samme som i den normale gren, men noget af underskuddet er altså allerede dækket af variabel produktion.
- Kald readjustParams
- Kald SolveMaxFlowProblem med step = 2. Her løses maxflowproblemet igen, men denne gang ignoreres ikke-variabel produktion. Det betyder at der udveksles energi mellem lande der teknisk set ikke har underskud, men som har underskud hvis de kun skulle bruge variable kilder. På den måde maksimeres forbruget af variable energikilder.

Denne metode tager ca. 3 gange længere end den originale.

7.5 PrepareHour

Denne metode kalder prepareHour-metoderne i alle Area- og Line-objekter, som opdaterer deres kapaciteter i forhold til stokastisk udetid. Så fylder den listerne productionList, variableProducionList og demandList og arrayet productionTypeMatrix, så disse værdier kan bruges af SolveMaxFlowProblem og SaveData

7.6 SolveMaxFlowProbem

SolveMaxFlowProblem er metoden, der beregner hvordan strømmen skal fordele sig på transmissionslinjerne mellem noderne, så det totale energiunderskud minimeres. Som beskrevet i Sisyfos5_Modelguide_slp.docx s. 18 kan dette formuleres som et lineært optimeringsproblem, hvor flowet ind i nodes med underskud maksimeres under følgende betingelser:

- Netto-flowet ind i en node med underskud af energi skal være mindre end eller lig underskuddet.
- Flow på transmissionslinjerne er indenfor kapaciteten af transmissionslinjerne.
- Netto-flowet ud af en node med overskud af energi er mere end eller lig overskuddet.

"Step" parameteren i SolveMaxFlowProblem giver mulighed for at løse MaxFlow på 3 forskellige måder (ved at give step = 0, 1, 2). Se mere i RunSimulation metoden.

Pakken "pulp" til Python tillader at man kan definere denne type problemer algebraisk, så den gør koden forholdsvis nem at skrive. En demonstration af denne kode kan ses i Jupyter notebooken Maxflow.ipynb.

Generelt er ét tal nok til at beskrive en transmissionslinje, hvor fortegnet så bestemmer retningen (eksempelvis svarer -500 på "DK1_to_DELU" at DELU sender DK1 500 MWh, mens 500 svarer til at DK1 sender DELU 500 MWh), men pulp ser ud til at minimere alle variable til at starte med, og så løse problemet derfra. Det betyder at som udgangspunkt ville der altid blive sendt noget på linjerne, om det var nødvendigt eller ej – derfor er det nødvendigt at "splitte transmissionslinjerne op" i deres retninger, så den mindste værdi de kan tage er 0 – altså ved at lave dobbelt så mange ensrettede linjer. Pulp-variablene for disse linjer opbevares i arrayet F_vec, hvor lige indekser svarer til linjer, der går i den "retning" som deres navn definerer i linedata, mens ulige indekser svarer ti linjer, der går den modsatte retning.

Solveren der bruges hedder GLPK_CMD. Den sorterer altid variablene alfabetisk, før den løser problemet, og optimerer derfor favorabelt variable, der har navne, der ligger tidligt alfabetisk. For at løse dette problem gives transmissionslinjerne navne der starter med "a###", hvor "###" er et tal mellem 0 og antallet af opsplittede linjer. Det er nødvendigt at starte navnet med "a" fordi solveren giver en fejl, hvis et variabelnavn starter med et tal. Tallet tildeles ved at lave et array kaldt shuffledNames ved at kalde np.random.shuffle på et array kaldt indexArray, der er et 1d-array med længde på antallet af opsplittede linjer, og som indeholder tallene 0, 1, 2... på den måde får alle transmissionslinjer et unikt præfix. Det gør det også nemt efterfølgende at finde ud af, hvor transmissionslinjen ligger i den løsning, solveren spytter ud (som er alfabetisk sorteret) - hvis eksempelvis det 0te indeks i shuffledNames er 51, betyder det at værdien for linje 0 ender med at være i den 51. indgang i løsnings-arrayet. At blandin-

gen af linjer er baseret på alfabetisk sortering, betyder at den løsning, der på tidspunktet er implementeret, udelukkende gælder for GLPK_CMD solveren.

Før problemet løses fyldes F_vec med LpVariable-objekter, der svarer til transmissionslinjer. LpVariable er de variable, pulp kan optimere på når modellen kører. Variablene skabes med øvre grænser, der svarer til den nuværende maximale kapacitet i den retning, linjen kører – dette tal kan ændre sig, hvis en enhed på linjen havarerer.

Dernæst skabes et model-objekt, der er den model, der skal løses. Nu skal variablene fyldes ind i modellen. Der loopes over alle nodes. For hver node finder vi ud af hvad forskellen mellem demand og produktion er, og om noden har overskud. Så loopes der over linjerne, der går ind i noden - hvor mange der er, er givet i fromLength-arrayet (og også toLength, men der står samme tal – der går lige så mange linjer "ind" i en node som der går "ud"). Vi bygger en build variabel, der bliver en sum af variable, der giver hvor meget energi der totalt flyder ind eller ud af noden - de linjer der går ud fra noden skal selvfølgelig indgå i denne med negativt fortegn, og omvendt for linjer der går ind i noden. "build" bliver så et udtryk for det totale netto flow ind i noden. Hvis der er overskud i noden tilføjes der de begrænsninger til modellen, at build-variablen (dvs nettoflowet ind i noden) er mindre end 0 og flowet ud af noden (det vil sige build-variablen med negativt fortegn) højst er lig overskuddet. Hvis noden har underskud skal flowet ud af noden være mindre end 0, og flowet ind skal højst være underskuddet, og netto-flowet skal maximeres - det lægges derfor til variablen obj_func, altså optimerings objective for modellen. Når dette er gjort for alle noder, lægges obj_func til modellen, og den løses.

Når modellen har løst problemet ligger outputtet i en liste, der hedder model.variables(). Herfra kan man få navne og værdier på de løste variable. Nogle gange (slag på tasken, 1 ud af 1000 simuleringstimer) bliver der, af grunde jeg ikke forstår, lagt ekstra variable ind i starten af listen med navnet "__dummy". Så hvis listen er længere end jeg forventer, springer jeg de første indgange over. Vi kan så fylde transferList (der skal holde værdien for overførslen på alle transmissionslinjer, når modellen er løst), ved at udnytte, at vi kan slå op i shuffledNames for at finde ud af, hvor i variabellisten, den pågældende linje er.

7.7 readjustParams

Mellem simuleringsskridt skal overskuddet i hver node og kapaciteten på transmissionslinjerne justeres for at tage højde for den kapacitet, der allerede er brugt. Dette gøres ved at loope over den tilstand transferList har fået efter sidste simuleringsskridt, og trække nettoeksporten af energi fra en node fra dens produktion. Transmissionslinjernes kapacitet opdateres, så de er lavere med den kapacitet der allerede er brugt.

7.8 SaveData

Når denne metode kaldes er demandList, productionsList, transferList og productionTypeMatrix opdateret med værdierne for den foregående time. I SaveData funktionen bliver dette data skrevet i samme rækkefølge som headeren, der blev defineret i constructoren for Simulation-klassen.

8 Line

Line-klassen gør ikke andet end at holde på værdier for et linje-objekt og at tillade en transmissionslinje at havarere. Der vil derfor ikke gives en beskrivelse af andet end mekanismen for den stokastiske udetid.

8.1 PrepareHour

I data-filerne gives udetiden som hvor meget af året, en enhed forventeligt er ude d, samt hvor lang den gennemsnitlige udetidsperiode t_{avg} er. Følgelig må (som beskrevet på s. 6 af baggrundsrapporten) sandsynligheden for at en enhed går i stykker i en specifik time P_{hav} være

$$P_{hav} = \frac{d}{t_{avg}(1-d)}, \tag{1}$$

og sandsynligheden for at en havareret enhed bliver fixet P_{fix} i den pågældende time er:

$$P_{fix} = \frac{1}{t_{avg}}, \tag{2}$$

I et kald til PrepareHour genereres der derfor et tilfældigt tal mellem 0 og 1 for hver enhed på transmissionslinjen. Hvis tallet er mindre end P_{hav} havarerer enheden. Kapaciteten på transmissionslinjen opdateres så. Derefter tillades enhederne at fixes: for hver havareret enhed, hvis et tilfældig tal er mindre end P_{fix}, så stopper enheden med at være havareret (men kapaciteten opdateres først i næste time – at det skal være sådan er et valg jeg har truffet arbitrært, men det betyder ikke meget. Der står ikke noget om hvordan de har gjort i Sisyfos).

9 Area

Area-klassen simulerer en node. Den holder på Production-objekter som svarer til værker, og er ansvarlig for at lave planlagte udetidsplaner.

9.1 __init__

Parametre:

Navn	Beskrivelse
demand	Et production-object (se Production-sektionen), der skal holde det demand, der følger en timeserie. Demand kan behandles som produktion med modsat fortegn.
nonTDdemand	Et production-object, der indeholder det demand, der ikke følger en timeserie (reserve og andre nogle ting, der kun er relevante for DK1 og DK2)
productionList	En list af production-objects, der har en timeserie.
productionNames	Navnene på de produktionstyper, der ligger i productionList.
nonTDProductionList	Ligesom productionList men for produktioner, der ikke har timeserie.
nonTDProductionNames	Ligesom productionNames.
variableProd	Identisk med productionList, med den undtagelse at den ikke indeholder produktioner, der har en "HYLimit" timeserie. Grunden til dette er, at dette ikke er uregulerbar produktion - HYLimit betyder bare at det har en maksimal produktion, der er givet ved en timeserie, men produktionen kan altså skrues op og ned, modsat sol og vind (HYLimit kommer så vidt jeg forstår af hydroværker, hvor der er en sæsonbetinget begrænsning på, hvor meget vand der må trækkes ud på et givet tidspunkt).
timeSeriesProductionList	Array til at indeholder alle timeserierne for denne node.

Table 4: Tabel over variable i Area klassen.

`options: Options, dh: DataHolder, areaName: str, nodeIndex: int`

nodeIndex er nodens indgang i listen af nodes, som er givet i DataHolder klassen. Constructoren til area-klassen henter de relevante timeserier fra DataHolder, og kalder nok vigtigst af alt de tre metoder InitializeDemand, InitializeFactors, og LoadOrCreateOutagePlan. Den initialiserer variable som kan ses i tabel **??**.

9.2 InitializeDemand

Bemærk: denne metode er anderledes for DK1 og DK2, og der er altså en metode i AreaDK-klassen, der overrider den.
InitializeDemand læser filen "demands.txt", og finder nodens årlige energiforbrug for simuleringsåret. Tallet er opgivet i TWh, og timeserierne er normeret til at summere til 1 TWh – så ved at gange dette tal med timeserien får man

altså det rigtige energiforbrug i den pågældende time. TVAR.csv indeholder to timeserier: ENTSO-E timeserier for 2025 og 2030. I Sisyfos bliver disse skaleret med en faktor. Begrundelsen for dette har jeg ikke fuldt ud forstået, men jeg citerer fra side 26 af baggrundsrapporten:
"Når der regnes på et konkret klima-år, skaleres forbruget op eller ned, så det svarer til de samlede forbrug i ENTSO-E's timeserier. Årsforbruget kan variere op til +/- 5% for enkelte lande i enkelte år i forhold til det gennemsnitlige årsforbrug."
Jeg forstår ikke dette, da de allerede har normeret ENTSO-E's timeserier til 100 TWh – men det er altså implementeret i koden alligevel, så der returneres tal, der tilsvarer dem, man kan se på "Nodes" fanen i Sisyfos5data.xlsx. Men der slåes så op i relativeDemands filerne, og faktoren for noden for klimaåret ganges på faktoren. Til sidst tilføjes en "producer" (se sektionen om Production-klassen) til demand-produktionen (demand kan modelleres som en produktion med modsat fortegn).

9.3 InitializeFactors

InitializeFactors looper over de værker, der er listet for den pågældende node i capacities.txt. Denne fil er allerede læst af DataHolder og dataet kan hentes fra variablen productionDict. Der loopes så over alle værkerne fra productionDict.

Fra productionDict fås også produktionstypen (det vil sige vind/gas/nuclear osv). Fra typen kan man få udetidsparametre og antallet af enheder (undtagen for mange danske værker, hvor antallet af enheder er hardcoded – for udenlandske værker er der givet et tal der svarer til kapaciteten af én enhed). Disse udetidsparametre ligger i outageParams.txt og er allerede blev læst af DataHolder.

Når parametrene for denne produktion er fundet, skal den lægges i det rigtige Production-object. Hvis der endnu ikke findes et Production-object til typen, det pågældende værk har, laves et nyt production-object og lægges i den rigtige liste. Ellers tilføjes værket til den eksisterende produktion.

Til sidst anvendes CreateArrays() metoden på alle Production-objekter, som skaber numpy-arrays af dataet i Production-objekterne, som forhåbentligt kan øge effektiviteten lidt.

For nonTDdemand sættes en tom udetidsplan og den første time forberedes, så den er klar til at spytte værdier ud.

9.4 LoadOrCreateOutagePlan

Som navnet foreslår hentes enten udetidsplanen fra DataHolder, eller også skabes den, hvis DataHolder ikke har loaded en outageplan. Den skabes gennem metoden CreateOutagePlan.

9.5 CreateOutagePlan

Når denne metode kaldes er alle produktioner og forbrug allerede indlæst. Udetidsplanen laves kun for produktioner der ikke har en timeserie. Først laves en total liste af kapaciteter, antal enheder, navne og længden af planlagte udetider. Så laves "onMatrix", der for hvert værk er 8760 indgange (antallet af timer i et år), hvor der på hver indgang står antallet af enheder, der er tændte i den pågældende time. Fra start lader vi alle enheder være tændte. Variablen "demandCopy" bliver forbruget i alle timerne i et år. Hvis brugeren har valgt, at der skal tages højde for variationer (produktioner med timeserier) når udetidsplanen laves, kan produktionen fra variable trækkes fra forbruget.

Materialet om Sisyfos er tvetydigt om, hvordan Sisyfos gør. I sektion 2.2 af baggrundsrapporten står der "Vind og sol deltager ikke i revisionsplanen. Modellen ved altså ikke, hvad den fremtidige vind- og solproduktion er på det tidspunkt, hvor revisionsperioden fastlægges". I Sisyfos5Modelguide står der i Appendix 5: "First total hourly demand is computed. Then the total capacity taking variations into account. That is, each plant's capacity is multiplied with the variation profile that describes the share of reduction in the capacity. The area between these two curves describes the surplus capacity". Der er derfor lavet udetidsplaner baseret på begge metoder. Ellers genereres udetidsplanerne som beskrevet i Appendix 5:

Der loopes over alle værker, og værket med den største enhedskapacitet findes; det er altså dette værk, der har størst konsekvens ved at have udetid. Så løbes der igennem alle timer i året, og det tidspunkt, hvor underskudet uden dette værk, er lavest, findes. Så sættes værket til at være slukket i denne periode, forbruget sættes til at være tilsvarende højere i perioden, og vi gentager for næsthøjeste værk. Efter det skrives udetidsplanen til en fil, der genereres løbende for hver node.

Forskellen på, om man tager højde for variation eller ej ser drastisk ud hvis man plotter det, men begge metoder vil placere udetiderne om sommeren, hvor der sjældent er energimangel alligevel, og det har derfor ikke stor indflydelse på EENS, hvilken metode man bruger.

10 AreaDK

AreaDK-klassen arver fra Area-klassen og er AreaDK objekter er derfor identiske til Area-objekter med undtagelsen, at InitializeDemand metoden er anderledes. Sisyfos tager højde for ændringer i forbrugsvaner og fleksibilitet i DK1 og DK2 (alle andre noder antages at forbruge som i fortiden). Det fleksible forbrug holdes fuldstændigt udenfor modellen – man må altså regne med, at det forbrug først realiseres når der er overskud i Danmark, som ikke går til at opveje et underskud andetsteds. Udover det er der et fladt energiforbrug fra datacentre, som altid kører. Det variable energiforbrug fra andre forbrug beregnes således:

$$\text{forbrug} = \text{årligt forbrug}(1 - \text{flexibel andel} \cdot \text{flexibilitetsparameter}). \quad (3)$$

Årligt forbrug og flexibel andel er givet i DK*demand.txt, mens flexibilitetsparameteren er en option sat af brugeren. Hvor præcist de tal kommer fra (er de bare estimater?) ved jeg ikke.

11 Production

Production-klassen er hjælp til bogholderi. Klassen repræsenterer en type af produktion i en node - ikke nødvendigvis et værk. Eksempelvis er Amagerværket Blok 1 og Asnæsværket blok 2.1 begge type CKV_BP (jeg har endnu ikke forstået hvad disse typer specifikt betyder), og ligger derfor i samme Production-object. Der er altså et hierarki: Et production-object kan indeholde mange værker, der hver kan indeholde mange enheder.

Bemærk at det generelt er produktioner med timeserier, der laves som Production-objekter. Produktioner der ikke følger timeserier, laves som NonTDProduction-objekter.

11.1 AddProducer

AddProducer metoden tilføjer et værk til listen af værker ved at forlænge listerne i Production-klassen. Disse indeholder så information om det værk der er blevet indsat som navn, kapacitet, uplanlagt udetid, planlagt udetid, gennemsnitlig udetidsvarighed, afhængighed af temperatur, antal af enheder, type, variationstype, og selve timeserien. Alle disse parametre er ikke nødvendigvis relevante. Eksempelvis følger ikke alle værker en timeserie - her kan så gives en timeserie, der er 1 konstant i stedet (dataHolder har en variabel der hedder constantTimeSeries).

11.2 GetCurrentValue

Denne metode looper over alle værkerne i Production-objektet, og udregner den totale produktion i dette production-object som

$$P = \sum_{i}^{\text{alle værker}} \text{kapacitet}_i \frac{\text{antal fungerende enheder}_i}{\text{totalt antal enheder}_i} \text{timeserien i pågældende time}_i \tag{4}$$

Denne metode arver NonTDProduction, og derfor er det nødvendigt at tage højde for havarerede enheder.

12 NonTDProduction

NonTDProduction er ligesom Production-klassen som den nedarver fra, men en smule mere kompliceret for at tage højde for udetid som kan ske i alle værker, der ikke følger en timeserie, og varmebinding, som er implementeret for danske værker.

12.1 SetOutagePlan

Denne metode tager som input den udetidsplan, der enten er blevet læst eller skabt i CreateOutagePlan samt headeren (det vil sige navnene på de værker, hver søjle i udetidsplanen svarer til), og matcher værkerne med søjlerne i udetidsplanen, og gemmer det i en liste.

12.2 InitializeFailedUnits

Man må forvente, at der i simuleringens første time er nogle værker, der allerede er havererede "fra det forrige år". Denne metode løber igennem alle værker og sørger for, at de med en chance der er lig chancen for at de har uplanlagt nedetid, er nede, når simuleringen starter. Jeg ved ikke om Sisyfos rent faktisk også gør dette (det er ikke beskrevet noget sted), men det burde den.

12.3 PrepareHour

Prepare hour metoden skal forberede Production-outputtet til den kommende time. For det første skal der initialiseres nogle havererede enheder, hvis dette er simuleringens første time. Dernæst loopes der over alle værker. Hvis værket ikke kan haverere (det vil sige det har en uplanlagt udetid på 0) lægges denne værdi til den totale produktion:

$$P_c = C\frac{N_p}{N_t}(1-H) + THC\frac{N_p}{N_t} \tag{5}$$

hvor C er kapaciteten af værket, N_p er antallet af enheder, der er funktionelle i følge udetidsplanen, N_t er det totale antal enheder, H er værkets varmebinding og T er temperaturen. Det vil altså sige H bestemmer, hvor meget værket er afhængigt af temperaturen. Temperaturen kommer fra en kolonne i TVAR.csv, og værdierne for H er givet i OutageParams (det er 0.7 for nogle udvalgte værker i Danmark og 0 for alle andre).

Hvis værket kan haverere loopes der først over alle units, der allerede er havererede, for at se om de regenerer. Metoden til dette er den samme som er beskrevet i sektion **??**. Jeg har her foretaget et lidt arbitrært valg om, at rækkefølgen er, at værker først regenerer, så havererer, og så tælles produktionen. Når værkerne har haft mulighed for at haverere beregnes produktionen som i ligning (**??**), med den undtagelse at N_p nu er antallet af enheder ifølge udetidsplanen, fratrukket antallet af enheder der er havererede.

13 PostProcessor

PostProcessor-klassen er til for at læse en fil efter simuleringen er kørt (resultatsfilen) og skrive en ny fil, hvor produktionerne er justerede, så der kun produceres det, der er nødvendigt (det vil sige nodens eget forbrug plus netto eksporten). Man er derfor nødt til at vælge en prioriteringsliste, så programmet ved hvilken

produktion der skal slukkes først – jeg har lavet en default, hvor variabel produktion sættes højest, men man kan variere den selv i PostProcessor.Py. Det er bare vigtigt, at listen indeholder alle de samme typer produktion, som er givet i OutageParams.txt.

I samme fil er der lavet en hjælpeklasse der hedder Node. Det er denne, der justerer værdierne i hver time.

13.1 process

Denne metode tager som parametre stien til en fil der skal postprocesses, og stien hvortil den prostprocessede fil skal gemmes. Disse kan ikke være det samme, for så bliver filen overskrevet mens den læses.

Først læses headeren i filen, der skal postprocesses, og denne header gives til alle Node-objekter, så de kan finde ud af, hvilke indekser der betyder noget for dem.

Derefter køres filen igennem, hvor hver Node læser linjen, justerer produktionen, og sætter den justerede produktion i et "shouldBe" array, der læses ud, og skrives til den nye fil.

14 Node

Denne klasse repræsenterer en Node i en resultatsfil fra simuleringen, og holder information om hvilke indekser i resultatsfilen, der indeholder relevant information, og er i stand til at beregne hvad der skal stå på dem.

14.1 ReadHeader

ReadHeader metoden får som input den første linje i resultatsfilen, og skal læse den og finde de indekser, der er relevante for den specifikke node. Det gør den ved at splitte hvert element i headeren op efter hvor mange underscores der er. Hvis der er 2, må det være en transmissionslinje, siden disse er navngivet "A_to_B". Hvis en af A og B er navnet på noden, gemmes indexet. Hvis der er ét underscore, har vi fat i en produktion (eller demand), og hvis navnet er nodens navn, må det også gemmes. Indekset gemmes så i en liste, på det indeks der svarer til produktionstypens placering i prioritetslisten.

14.2 ReadHour

ReadHour kaldes for hver time efter ReadHeader er blevet kaldt. Den får som input hele linjen fra resultatsfilen, og den ved fra indekserne fundet i ReadHeader, hvor den skal lede efter relevant information. Den henter så demand ind, netto import, og sætter produktionen ind i en liste, der følger den prioriterede rækkefølge.

14.3 AdjustProduction

I den metode opdateres nodens "ShouldBe" array, der ender med at indeholde de værdier, der skal skrives til den postprocessede fil. Der laves et target for den pågældende time, der er lig demand minus nettoimporten – det må være det, der skal produceres i Noden. Så summes der over den produktion, der har været i Noden i den prioriterede rækkefølge indtil target er nået, mens shouldBe opdateres med de værdier der lægges til. Når man er ved at overstige target, lægges der kun det til, der skal til for at overstige target. Alt produktion der endnu ikke er nået, sættes til 0.

15 Kommentarer om Sisyfos

15.1 Flexibelt forbrug

For de to danske områder er der indført fleksibelt forbrug. Sisyfos blev lavet til kun at undersøge de kritiske timer i forhold til effektoverskud. Derfor holdes fleksibelt forbrug, som PtX, fuldstændigt udenfor modellen, da antagelsen er, at dette forbrug ville blive slukket under spidsbelastning. Hvis man vil se på energiforbrug i timer udenfor spidsbelastning kan disse ikke ignoreres, og man har brug for en mere kompliceret model for forbruget.

15.2 Værker udenfor modellen

I Sisyfos (men ikke Sisyfos-lite) er der lavet nogle grænselande, der er implementeret meget groft. Disse lande er et kraftværk med en transmissionslinje forbundet til et land, der er i modellen. I Sisyfos5Data25.xlsx er det de værker, der ligger alle nederst i Plants arket: Slovenien, Montenegro, Grækenland, Ukraine, Kroatien, Rumænien, Serbien, Nordirland, Irland, Morocco og Rusland.

Disse lande er essentielt kraftværker uden forbrug. Energistyrelsen vurderer selv, at de ligger for langt fra Danmark til at have reel indflydelse på kapaciteten i Danmark (med undtagelse af de Russiske transmissionslinjer, men disse er lukkede grundet konflikten i Ukraine alligevel). Eksempelvis Ungarn er dog meget afhængige af lande udenfor modellen, og får meget EENS som følge af deres udelukkelse.

At behandle dem som en transmissionslinje forbundet til et værk kan nogenlunde retfærdiggøres under Sisyfos' problemstilling, hvor man kun kigger på de allermest kritiske timer – i disse timer ville man ikke eksportere energi alligevel. Implementeringen er dog så grov, at man nok lige så godt kunne have undladt, og det er derfor ikke med i Sisyfos-Lite. Ønskede man at tilføje dem, ville det ikke være svært; det ville kræve en opdatering af capacities.txt med de nye værker i nye noder, en opdatering af lines****.txt med de nye transmissionslinjer, og en opdatering af DataHolder og PostProcessor med de nye navne.

15.3 Ændrede forbrugsvaner

For de danske områder er der indbygget ændringer i forbrugsvaner i fremtiden i form af fleksibelt forbrug. For alle andre områder er der ikke gjort nogen antagelser, og man tager altså bare det forventede årlige energiforbrug for disse nodes og fordeler det som en timeserie fra fortiden. Men dette er igen en lidt grov approksimation, for man må forvente at der i dette energiforbrug også er lagt PtX ind, som i hvert fald bør være mere fleksibelt end fortidens energiforbrug. Og ligeledes er der en forventning om flere datacentre, der er mindre flexible end fortidens elforbrug.

Det er et problem der ville kræve en løsning, der går langt ud over kompleksiteten af Sisyfos, for forbruget i fremtiden forventes at være koblet stærkere til de uregulerbare kilder end de er nu.

Sisyfos har forsøgt at imødekomme problemet gennem DSR (demand-side response), der er indkodet som et kraftværk (fordi et fald i forbrug er ækvivalent med tilvækst i produktion) i alle andre lande end Danmark. Det vil sige at man har en forventning til, at der i de kritiske timer vil blive slukket for X MW fordi strømprisen stiger. Disse tal kommer ligesom kapacitet fra MAFTYNDP.

15.4 Batterier

I Sisyfos er energilagring (eksempelvis batterier og pumped hydro) modelleret som kraftværker med høj uplanlagt udetid, som ifølge baggrundsrapporten skal simulere, at batterierne tømmes. Men et batteri er ifølge udetidsparametrene delt op i enheder af 10 MW, og hydro 50 MW. Det betyder at eksempelvis Holland med 3475 MW batterikapacitet i 2040 har 348 enheder. Grundet det store antal enheder betyder den uplanlagte udetid bare, at der effektivt set er 20% mindre kapacitet, men der er altså energi fra batterier tilgængeligt på ethvert tidspunkt (grundet det store antal enheder er der altså ikke effektivt set stokastisk udetid).

Det betyder dog ikke, at batterierne konstant giver energi – realistisk set ville de kun tages i brug, efter andre kilder har fejlet i at give strøm. Men Sisyfos kan altså ikke modellere, at batterier løber tør. Om dette har reelle konsekvenser må en analyse af resultaterne vise – det vil sige, undersøge om det det på noget tidspunkt har været nødvendigt at trække strøm ud af batterierne så mange timer i streg, at de burde være gået tørre.

Sisyfos har heller ikke nogen mekanisme til at oplade batterier på en realistisk måde. Lige nu må man forvente, at batterierne oplades kontinuært gennem forbruget, fordi forbruget er udregnet ud fra det årlige totale forbrug, som må inkludere batteriernes forbrug. Paradoksalt må det betyde, at der nogle gange bruges batterier til at dække et forbrug, der kommer fra at oplade batterier.

Fundamentalt ligger problemet i at der skal to tal til at karakterisere et batteri: hvor meget energi der er i det, og hvor meget effekt der kan trækkes ud – og energien, der er i batteriet, skal bestemmes dynamisk, så der er energibevarelse. Sisyfos bruger kun ét tal: hvor meget effekt, der kan trækkes ud.

En eventuel løsning på problemet kunne være en ny underklasse af Production, der tager højde for dette. Men at bestemme hvornår batterier skal oplades og tages i brug er et ikke-trivielt problem, for det afhænger af el-pris som er udenfor Sisyfos' scope.