Faraday Grid Wholesale Market Model v0.1

Installation instructions

## 1. Install Julia

First, download Julia (command line version) for your platform from the [downloads page](http://julialang.org/downloads/). Then:

* On Windows: Run the downloaded installer, using the default settings.
* On OS X: Double-click the downloaded disk image to load it. Drag the Julia-0.X.Y app onto the Applications folder shortcut.
* On Linux: Unzip the downloaded folder and place it in your home directory. Make sure to have dependencies curl and cmake installed.

NB: The model has been tested in Julia 0.6.3. Julia 1.0 has since come out. Initial testing suggests that several key packages do not yet work with Julia 1.0, so an older version should be installed.

## 2. Install Atom

Download, install and open [Atom](https://atom.io/). If you have it already, make sure it's up to date.

## 3. Install Juno

* In Atom, go to Settings (Ctrl+,, or Cmd+, on OS X) and go to the "Install" panel.
* Type uber-juno into the search box and hit enter. Click the install button on the package of the same name.
* Atom will then set up Juno for you, installing the required Atom and Julia packages.

## 4. Use Juno

Juno should now work – try opening the REPL with Packages > Julia > Open Console or Ctrl-J Ctrl-O and press Enter to start a Julia session.

## 5. Install the necessary Julia packages

Execute the Install.jl file to install all the necessary packages. If you are already using Julia, save time by only installing packages that are not yet installed. **NB:** One of the packages installed is Taro.jl, which manages the link between Julia and Excel. Taro uses a Java VM. If Java is not installed on the system, this will prompt error messages. In some cases, the default memory size for the VM will not be sufficient to load the dataset. If this is the case, find the Taro.jl file on the system (default location: ~/.julia/vx.x/Taro/src, where x.x is the Julia version installed, e.g., 0.6), open it in a text editor and find the line that specifies the maximum memory use of a VM, which will look like *JavaCall.addOpts("-XmxXXXXM")*, where XXXX is a value. Using 6144 instead of the default has been verified to work with the UK dataset. Save the file and restart Julia.

NB: packages are updated regularly. Newer versions may not be compatible. The models have been tested with JuMP 0.18.2, Gurobi 0.4.1, Complementarity 0.3.0, Taro 0.6.0, Plots 0.17.3, and DataFrames 0.11.6. If newer versions generate errors, revert to these versions.

## 6. Install the necessary solver(s)/licenses

* Gurobi for linear and quadratic optimisation. If Gurobi is not available, an open-source alternative such as Ipopt can be used. This requires
  + All *solver=GurobiSolver()* statements in ModelFunctions.jl need to be changed to the new solver (e.g., *solver=IpoptSolver()*)
  + The Julia package for the solver must be installed. The easiest way to do this is to add it to the Install.jl file before it is run (e.g., add a line to *Pkg.get(Ipopt)*
  + The Julia package for the solver must be loaded before any model is run. The easiest way to do this is to add a new line at the top of ModelFunctions.jl (e.g., *using Ipopt*)
* PATH for imperfectly competitive market simulation. This solver is free and will be installed automatically if Install.jl is run; however, a free license bust be installed to solve larger problems. Details are available at <http://pages.cs.wisc.edu/~ferris/path/LICENSE>

Faraday Grid Wholesale Market Model v0.1

Usage instructions

## Workflow

## The model consists of a number of files

## ModelFunctions.jl: This is where all the models are defined. Developers can add new connstraints or change model definitions here.

## ExecuteModel.jl: This is where models are executed. Within this file, there are a number of usage cases. To execute one, select the relevant lines, uncomment them, and execute the file; alternatively, select the lines to be executed and select “execute block”.

## DemandFunctions.xlsx: This is the default location where demand functions are saved and loaded from for models with elastic demand

## InputDataUK2.xlsx: Input data for the UK

## Output files:

## MarketOutcome.xlsx: This is the default location where market results are saved

## RedispatchOutcome.xlsx: This is the default location where redispatch results are saved.

## Procedures

## ExecuteModel.jl can run the following procedures:

## ProcessData(*input\_data*)

## This procedure loads the GB dataset and transform it to a form that can be used in the optimisation models. This includes calculating some additional parameters.

## *input\_data* is a string, containing a file name and location. For ease of use, these are

## defined at the top of ExecuteModel.jl

## NB: The JVM used by this procedure may complain about segmentation faults and/or illegal reflective access operations. These have no consequence and can be ignored.

## LoadDemandFunctions(*input\_demandfunctions*)

## This procedure loads previously saved demand functions. It only needs to be called if the elastic demand model is used and if there are no demand functions in memory (i.e., WriteDemandFunctions has not been called during the current session).

## *input\_demandfunctions* is a string, containing a file name and location. For ease of use, these are defined at the top of ExecuteModel.jl

## NB: The JVM used by this procedure may complain about segmentation faults and/or illegal reflective access operations. These have no consequence and can be ignored.

## LoadMarketOutcomes(*input\_marketoutcomes*)

## This prodecure loads previously saved market outcomes. It only needs to be called if the redispatch model is used and a market model has not been called during the current session.

## *input\_marketoutcomes* is a string, containing a file name and location. For ease of use, these are defined at the top of ExecuteModel.jl

## NB: The JVM used by this procedure may complain about segmentation faults and/or illegal reflective access operations. These have no consequence and can be ignored.

## WriteMarketData(*output\_marketoutcomes*)

## This procedure writes market outcomes to a file. It should only be called if a market model has been executed before. *output\_marketoutcomes* is a string containing a file name and location. For ease of use, these are defined at the top of ExecuteModel.jl

## WriteRedispatchData(*output\_redispatch*) This procedure writes market outcomes to a file. It should only be called if a redispatch model has been executed before. *output\_redispatch* is a string containing a file name and location. For ease of use, these are defined at the top of ExecuteModel.jl

## WriteDemandFunctions(*output\_demandfunctions*,*epsilon*)

## This procedure generates linear demand function around reference prices and quantities, using a pre-defined demand elasticity. A market model must be called first, or market outcomes must be loaded from file.

## *output\_demandfunctions* is a string containing a file name and location. For ease of use, these are defined at the top of ExecuteModel.jl

## *epsilon* is a numeric value which must be strictly negative, representing the demand elasticity. For ease of use, it is defined at the top of ExecuteModel.jl; the default value is -0.2.

## RollingHorizonModel(*hh*)

## This procedure runs a rolling horizon market model with fixed demand. *hh* is an integer between 1 and 8761, representing the number of hours market players look ahead. The default is 24 but this can be adjusted. For ease of use, it is defined at the top of ExecuteModel.jl. Larger values will increase the solution time.

## RollingHorizonModelElasticDemand(*hh*) This procedure runs a rolling horizon market model with elastic demand. It requires demand functions to be in memory, so WriteDemandFunctions or LoadDemandFunctions must have been called in the current session. Note that if the same input data set that has been used to calibrate demand functions is used by this model, this is an unnecessary computation as demand turnouts will be equal to the reference demand input. This model should only be used to consider the effect of changes in the initial dataset after demand functions have been calculated. *hh* is an integer between 1 and 8761, representing the number of hours market players look ahead. The default is 24 but this can be adjusted. For ease of use, it is defined at the top of ExecuteModel.jl. Larger values will increase the solution time.

## ImperfectCompetitionModel(*hh*) This procedure runs a rolling horizon Nash-Cournot imperfectly competitive market model, which can be used instead of RollingHorizonModelElasticDemand. *hh* is an integer between 1 and 8761, representing the number of hours market players look ahead. The default is 24 but this can be adjusted. For ease of use, it is defined at the top of ExecuteModel.jl. Larger values will increase the solution time.

## NB: This procedure is much more computationally expensive than the others.

## RedispatchModel(*rhh,elastic*) This procedure runs a least cost redispatch model. *rhh* is an integer between 1 and 8761, representing the number of hours the system operator looks ahead. The default is 1 but this can be adjusted. For ease of use, it is defined at the top of ExecuteModel.jl. Larger values will increase the solution time.

## *elastic* is a Boolean variable taking the values *“true”* or *“false”*. If *“true”* is used, the redispatch model will use demand output from a previously called RollingHorizonElasticDemand or ImperfectCompetition model (hence, one of these models needs to have been called in the current session). If *“false”,* it will use fixed demand from the database.

## GeneratePricePlot() This generates a plot of prices. Requires a market model to have been called before, or market outcomes loaded from file. A copy of the plot will automatically be saved in the project folder.

## GenerateRedispatchCostPlot() This generatates a plot of redispatch costs. Requires a redispatch model to have been called before, or redispatch outcomes loaded from tile. A copy of the plot will automatically be saved in the project folder.

## Example: increasing grid capacity

## Create a copy of the original input data file. In the copy, increase grid capacity by 10% by setting the derating value from 0.65 to 0.715. We will refer to the original file as *input\_data* and the new file as *input\_data2*. We will use *output\_redispatch* to refer to the redispatch outcomes in the base case, and *output\_redispatch2* to refer to redispatch outcomes in the new situation, with an increase in transmission capacity.

## An increase in transmission capacity will not change market outcomes, but will reduce redispatch costs. It is therefore not necessary to run a model with elastic demand (as we know what demand is – it will not change).

## Generate baseline data:

## Run ProcessData(input\_data) to load base case data

## Run RollingHorizonModel(24) to simulate market operation

## Optional: run WriteMarketData(output\_marketoutcomes) in case we need this later

## Run RedispatchModel(1, “false”) to calculate redispatch costs

## Run WriteRedispatchData(output\_redispatch) to save data.

## Optional: Run GenerateRedispatchCostPlot() to plot redispatch costs.

## Generate data for new scenario:

## Run ProcessData(input\_data2) to load new data

## (assuming market outcomes are still in memory; if not – load them)

## Run RedispatchModel(1, “false”) to calculate redispatch costs

## Run WriteRedispatchData(output\_redispatch2) to save data.

## Optional: Run GenerateRedispatchCostPlot() to plot redispatch costs.

## Use your favourite tool (e.g., Excel) to analyse the difference in redispatch costs and quantities.

## Example: increasing embedded wind and solar generation

## Create a copy of the input data file. In the copy, triple embedded generation through setting the multipliers to 300%.. We will refer to the original file as *input\_data* and the new file as *input\_data2*.

1. Generate baseline data:  
   Run ProcessData(input\_data) to load base case data  
   Run RollingHorizonModel(24) to get baseline prices and quantities.

Run WriteMarketData(output\_marketoutcomes) to save market data  
Run WriteDemandFunctions(output\_demandfunctions,-0.2) to calculate demand functions

Optional: Run RedispatchModel(1, “false”) to calculate redispatch costs

Optional: Run GeneratePricePlot() to save price a price plot

## Optional: Run GenerateRedispatchCostPlot() to plot redispatch costs.

1. Generate data for new scenario:

Run ProcessData(input\_data2) to load new scenario  
Run RollingHorizonElasticDemand(24)  
(assuming demand functions are still in memory, if not, load them)  
Run WriteMarketData(output\_marketoutcomes2) to save market dataOptional: Run RedispatchModel(1, “false”) to calculate redispatch costs

Optional: Run GeneratePricePlot() to save price a price plot

## Optional: Run GenerateRedispatchCostPlot() to plot redispatch costs.

1. Compare the two datasets with your favourite tool.

Faraday Grid Wholesale Market Model v0.1

Model assumptions

## Model structure

## The model consists of a series of procedures which can be run individually or in sequence:

## Wholesale market model with fixed demand

## This is a simple optimal dispatch model which simulates the outcome of a perfectly competitive market, assuming rational agents with perfect information within the pre-defined horizon and no information for hours beyond the horizon. Demand is assumed to be fixed, and, as in the UK market, transmission constraints are not considered – there is only one price for the whole market in each period. It minimises the total cost of generation over a pre-defined horizon, subject to capacity constraints on the generators and a market clearing constraint on the total level of generation:

## s.t.

## where is the generation level of plant at time period , the variable cost of plant , the subset of time periods considered in iteration , the demand at bus in time period , the capacity of generator and the capacity factor of generator in time period .

## The model starts the first period, , such that in the first iteration, , where is the pre-defined model horizon. Solutions are saved for , is set to , the model is solved again, and solutions for are saved. This is continued until the final time period. If there are fewer than time periods left until the final time period, will have a smaller dimension; e.g., in a model with 8761 time periods, in the penultimate iteration is final iteration is and in the final iteration simply . The dual of the market clearing constraint gives prices.

## Wholesale market model with elastic demand

## This is similar to the model above, but now with price-sensitive demand. A perfectly competitive market outcome can be obtained by finding a solution to a social welfare maximisation:

## s.t.

## 

## Demand functions are assumed to be linear. They can be defined directly or the can be calculated by the model. In the latter case, a fixed-demand model should be executed first to obtain reference prices and demand levels. The model will then calculate a linear demand function with a pre-defined elasticity around each reference price-demand pair. The shadow prices on the second constraint represent market prices.

## Wholesale market model with imperfect competition

## This model finds a Nash-Cournot equilibrium between a number of competing generation owners. Each firm maximises its own profits, taking the others’ generation decisions as given. To simplify this problem, realise that the demand functions at the individual buses all have the same intercept, as they only depend on prices. The aggregate demand curve is therefore linear and given by

## where and . Since , each firm’s maximisation problem is then:

## s.t.

## The KKT conditions for each generation owner are therefore

## A separate market clearing condition is not necessary because this has already been substituted in the profit functions. The model uses the PATH solver to solve the above equations for all generation owners. Prices and locational demand time series can be calculated ex post, using the locational demand functions.

## Redispatch model

## The model assumes cost-based redispatch if market outcomes are not feasible. This means that generators are reimbursed for increases or decreases in power output according to their variable costs – they do not make or lose money in the redispatch stage. The model uses a DC approximation of power flows. The system operator minimises costs:

## s.t.

## Where is the redispatch quantity, is the flow on line at time , the voltage angle at bus , time the set of generators at bus , the set of lines ending at bus and the set of lines originating at bus . and are the voltage angles at the origin and destination, of line , at time . is the thermal limit of line , and its reactance.

## GB test system

## The GB test system data that comes with the model is based on industry-standard methods. There is scope for further development. Data for other systems can be loaded. If these contain a different number of time periods or buses, input and output data ranges will need to be adjusted in ModelFunctions.jl

## Grid representation and data: This is based on the representative GB network model (RGBN) developed at the University of Strathclyde[[1]](#footnote-1) in the context of the Supergen Networks project, which is widely used. Buses correspond to zones used by National Grid in its own studies, and data on line reactances and flow limits are provided. The real network is operated in a security-constrained manner, using N-1 criteria. As is usual in these types of models, this is not explicitly taken into account, but instead, flow limits are reduced by a pre-defined factor. This can be defined in the input data sheet; by default, flow limits are reduced to 65% of their original levels.

## Generators: The base list of generators is taken from the RGBN model, but this has been adjusted to match a more recent list of generators that bid into the GB market. It is consistent with unit lists on the Elexon Portal, but wind generators have been aggregated at each bus to reduce the computational expense. A disaggregated generation data set is provided for future development. At each bus, a “load shedding” option is added, the cost of which corresponds to the current UK Value of Lost Load (VoLL). This keeps the redispatch problem feasible under all circumstances, but may overstate redispatch costs.

## Generation costs: Based on a linear approximation of power curves for fossil fuel plants, combined with current fuel prices. Marginal renewable generation costs are assumed to be zero. Ramp rates are provided by not currently used in the model – these are a very conservative estimate of ramp rates, based on ramping capability of power plants already spinning; these may need to be reduced. Ramp rates are given in MW/minute; all other variables are hourly.

## Resource availability: Wind capacity factor time series at the exact location of each wind farm are obtained by hindcasting a large mesoscale atmospheric flow model.[[2]](#footnote-2) When aggregating wind farms, a weighted average of capacity factors is taken. Solar availability is not currently modelled beyond embedded solar, as there is little large-scale solar on the system at present.

## Load: 2015 load time series are taken from the National Grid Data Portal.[[3]](#footnote-3) This was a relatively average year, but load data can be scaled up or down to consider different future assumptions.

## Embedded wind and solar: as above, taken from the National Grid Data Portal. This can be scaled up or down in the input data sheet. This data is not available in disaggregated form, so the assumption is that embedded wind and solar is distributed among the buses proportional to the load.

## Storage: not currently modelled explicitly, as large-scale storage facilities currently do not play a big role in electricity markets, instead bidding for longer-term contracts in ancillary services markets. This is an area for development if future power systems are to be modelled.

1. See e.g., K. Bell and A. Tleis, “Test system requirements for modelling future power systems,” in *IEEE Power and Energy Society General Meeting, 2010*, july 2010, pp. 1 –8; “Representative model of the GB transmission system” [online’. Available: <http://www.supergen-networks.org.uk/filebyid/748/RepresentativeGBNetwork.pdf>; L. P. Kunjumuhammed, B. C. Pal and N. F. Thornhill, "A test system model for stability studies of UK power grid," *2013 IEEE Grenoble Conference*, Grenoble, 2013, pp. 1-6. doi: 10.1109/PTC.2013.6652283. [↑](#footnote-ref-1)
2. S. Hawkins, High resolution reanalysis of wind speeds over the British Isles for wind energy integration. PhD thesis, University of Edinburgh, 2013. [↑](#footnote-ref-2)
3. https://www.nationalgrid.com/uk/electricity/market-operations-and-data/data-explorer [↑](#footnote-ref-3)