Development of a Gas-Electricity optimisation model

Modelling language migration and future extensions

Original work: Allahham & Hosseini
Re-coding: McKinnon, Dent & Garcia

National Centre for Energy System Integration (CESI)

April 2020

Outline

Goal

CESI is a multi-discipline knowledge and skill centre with topics and discussions crossing multiple fields, where easy-to-access tools are fundamental to enable these discussions.

While the existing optimisation models and implementations work well, we see advantages in coding the models using a modelling language.

CESI National Centre for Energy Systems Integration

Goal (cont.)

The benefits we see in using a modelling language are:

1. To provide a robust framework for easily extendable models using Allahham & Hosseini as a base.
2. To facilitate access to the tools by using open-source software.
3. To enable end-users to easily modify and/or adapt the models according to their needs.
4. To benefit from the use of mature optimisation solvers potentially speeding up the solution of models.
5. To be able to model and solve different forms of problems, e.g. models with integer decisions.
6. To allow for different datasets to be easily integrated into the models.

Introduction

Optimisation models

What is an optimisation model?

It's a mathematical description of a system that aims to minimise or maximise an objective with some constraints.

An optimization model has three main components:

1. An objective function. This is the function that needs to be optimised (maximised or minimised).
2. A collection of decision variables. The solution to the optimisation problem is the set of values of the decision variables for which the objective function reaches its optimal value.
3. A collection of constraints that restrict the values of the decision variables.

Introduction

Optimisation models (cont.)

What is an optimisation model?

Objective function

$$\min_{x} \quad f(x)$$

Contraints

subject to:

$$\begin{aligned} g(x) &= 0 \\ h(x) &\leq 0 \\ x &\geq 0 \end{aligned}$$

Variables and Data

- Variables. $x \in \mathbb{R}^n$
- Data. Encoded in f, g and h.

Introduction

Optimisation models (cont.)

The objective function togheter with the constraints and nature of the variables will determine the type of problem to be solved.

Some examples of types of problem can be:

- Linear programming: f,g and h are linear.
- Integer programming: integer solution required.
- Non-linear programming: f or g or h are non-linear.
- Dynamic programming: g or h describe system dynamics.
- Stochastic programming: f, g or h are functions of random variables.

Programming languages and software

In order to solve problems ranging from a couple to potentially hundreds of thousands variables with complex constraints we need to code the models in computer languages.

For optimisation problems, the software tools used can be divided arguably into 3 main categories:

1. General-purpose / Mathematical languages
2. Algebraic modelling languages
3. Solvers

Programming languages and software

General-purpose programming languages

General-purpose programming languages are designed to be used for writing software in the widest variety of application domains, therefore they lack specialized features for a particular domain such as optimisation. On the other hand, they are widely used and able to do a variety of tasks.

Examples: C++, Fortran, Visual Basic, Java, Python, Julia.

Programming languages and software

General-purpose programming languages

Mathematical programming languages are similar to general-purpose languages in the sense that they cover a broad range of applications, but they have been developed to be efficient with different mathematical aspects.

Examples:

- Numerical calculations: Matlab, R and Julia.
- Symbolic algebra: Mathematica and Maple.

Programming languages and software

Algebraic modelling languages

Algebraic modelling languages are computer programming languages for describing and solving high complexity problems for large scale mathematical computation. The main advantage is the similarity of their syntax to the mathematical notation of optimisation problems.

This allows for a very concise and readable definition of problems in the domain of optimization, which is supported by certain language elements like sets, indices, algebraic expressions, index and data handling variables, constraints with arbitrary names.

Examples: AMPL, AIMMS, GAMS, Lindo, JuMP[1].

[1] Technically not a modelling language, rather a quasi-core package for Julia.

Programming languages and software

Solvers

A **solver** is a piece of mathematical software, in the form of a stand-alone computer program or as a software library, that "solves" a mathematical problem by applying mathematical techniques. A solver takes problem descriptions in some sort of generic form and calculates their solution.

Commonly a solver specialises in a type of problem to solve depending on its characteristics.

Examples: CPLEX (LP, MILP, SDP, SOCP), Gurobi (LP, MILP SDP, SOCP), Knitro (MINLP), Ipopt (NLP).

IBM CPLEX | GUROBI OPTIMIZATION | IPOPT

Programming languages and software

Summary / Benefits

General-Purpose/Mathematical programming language. Widely used and can be used for different goals.

Algebraic modelling language. Allows for concise equation-like syntax, easy to understand and interpret. Can help modelling diverse optimisation problems.

Solver. Especialised solving packages, which exploit the structure of the problems and use built-in operations to solve them, e.g. Automatic forward/backward differentiation, using robust algorithms (IPM, Simplex, etcetera.)

Programming languages and software

Example syntax in JuMP (Algebraic modelling)

- Constraint for Power Generation Limits:
  - Expression:

$$P_g^{LB} \leq p_{g,t} \leq P_g^{UB} \quad \forall g \in \mathcal{G}, \, \forall t \in \mathcal{T}$$

  - Code:

```
@constraint(m, PLim[g ∈ G, t ∈ T], P_LB[g] ≤ p[g,t] ≤ P_UB[g])
```

- Objective function:
  - Expression:

$$\min \quad \sum_{g \in \mathcal{G}} C_g \sum_{t \in \mathcal{T}} p_{g,t}$$

  - Code:

```
@objective(m, Min, ∑(C[g]*∑(p[g,t] for t ∈ T) for g ∈ G))
```

Programming languages and software

Proposal

To use and integrated programming approach by incorporating into our modelling and solving approaches the 3 main softare elements used for optimisation. This allows for flexibility, robustness and speed. In concrete to:

1. Use **Julia** as a general/mathematical progamming software.
2. Use **JuMP** as a algebraic modelling language.
3. Use a solver (**Ipopt** or others).

Programming languages and software

Proposal (cont.)

Why?

Julia is free, fast (Figure 1) and has a friendly syntax (similar to Matlab). Julia has had an important increase of optimisation users that benefit from the tight integration with JuMP. Julia is compatible with other languages and have packages to integrate it with other languages such as Python.

JuMP is also free, fast and friendly. It benefits from being solver-agnostic and benchmarking has shown that can create problems at similar speeds than AMPL.

Programming languages and software

Proposal (cont.)

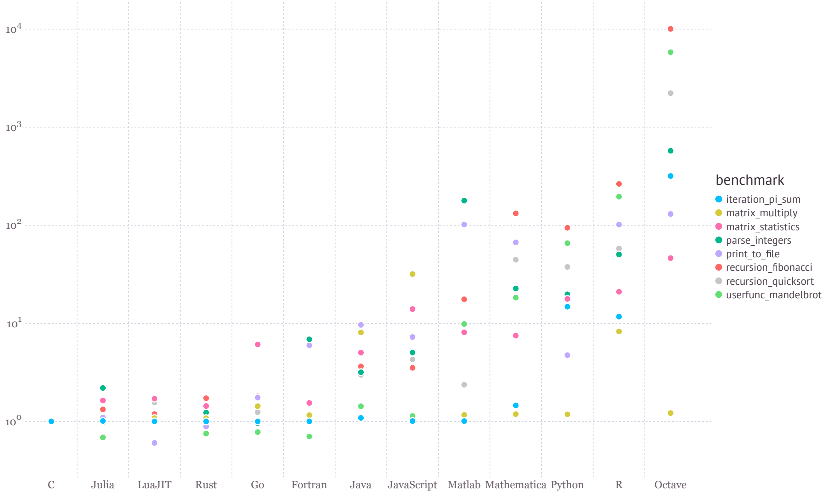

Figure 1: Benchmark of some common programming languages

Modelling components

Model

The optimisation **model** considers the whole set of equations that define the behaviour and response of a system. The model once written in an algebraic way can be then "coded" in the algebraic software (JuMP).

If new components of the system are considered or the dynamics change, then the model needs to be modified to reflect this situation. Examples of modifications of the model in this context are: extensions to multi-period operational problems, capacity planning problems, heat-network integrations, etcetera.

Modelling components

Data

The same dynamics in a system (model) can behave differently depending on the **Data**. The data needs to be read and integrated as parameters in the model. The data can come from different datasets and have information of different places. Such as information about Findhorn or Newcastle.

There are many different ways of reading data into Julia, directly or with the use of diverse packages and formats. It is important to mention that standard matpower formats are limited in their reach, as they cannot easily handle multi-period settings or other extensions.

Re-coding

Activities

The aim of this work is to use Allahham & Hosseini gas-electricity optimsiation model and recoding it to benefit from the use of Julia, JuMP and solvers (for the reasons stated before).

Also, McKinnon has done a recoding in AMPL that shares most of the benefits of the Julia/JuMP recode.

Re-coding
Progress

Progress to date:

- Read the full code, analysed how the data and model are linked and what were the implications of extending this code to more general cases.
- Developed an integrated data management module for reading and writing friendly datafiles, without the need of external packages. We've chosen to use a clean and robust "format" based on the AMPL data formats for the electrical networks, gas networks and the coupling nodes.
- Coded routines for matpower data file reading, so standard matpower cases can be easily used, without the need of Matlab or manual conversions of the data files.
- Verified cases between re-codings Matlab-AMPL-Julia

Re-coding

Data

At the moment, the data can be read easily in matpower format or in the AMPL-like format. Here are snapshots of both formats.

```
Branches//
:Name :bsend :brec :blength :bdiameter :befficiency :btemp  :bm1       :Fp  :flow_capacity
    1      1     3     80.5      19.56          0.9     520   2     7.93e6              8.5e6
    2      2     4     80.3      19.56          0.9     520   2     6.08e6               12e6
    3      3     4     55.9      19.56          0.9     520   2    -6.12e6               14e6
    4      3     5     81.1      19.62          0.9     520   2     4.24e6               12e6
    5      4     6     87.9      19.62          0.9     520   2      4.2e6               12e6
    6      5     7     25.0      19.62          0.9     520   2      4.5e6               10e6
    7      6     9     25.0      19.62          0.9     520   2     4.23e6                7e6
    8      8    11     93.5      19.62          0.9     520   2      4.5e6               11e6
    9     10    13     99.7      16.69          0.9     520   2     4.23e6              7.5e6
   10     12    15     93.5      16.69          0.9     520   2      4.5e6                6e6
   11     14    16     97.9      16.69          0.85    520   2     4.23e6                6e6
   12     15    16     86.6      16.69          0.9     520   2 -5.6276e6                6e6
   13     15    17     79.7      16.69          0.8     520   2    0.525e6                6e6
   14     16    17    107        10.00          0.9     520   2    0.525e6                6e6
//Branches
```

Figure 2: AMPL-like data format

Re-coding

Data (cont.)

```
%% bus data
%	bus_i	type	Pd	Qd	Gs	Bs	area	Vm	Va	baseKV	zone	Vmax	Vmin
mpc.bus = [
	1	3	0	0	0	0	1	1	0	345	1	1.1	0.9;
	2	2	0	0	0	0	1	1	0	345	1	1.1	0.9;
	3	2	0	0	0	0	1	1	0	345	1	1.1	0.9;
	4	2	0	0	0	0	1	1	0	345	1	1.1	0.9;
	5	2	90	30	0	0	1	1	0	345	1	1.1	0.9;
	6	2	0	0	0	0	1	1	0	345	1	1.1	0.9;
	7	2	100	35	0	0	1	1	0	345	1	1.1	0.9;
	8	2	0	0	0	0	1	1	0	345	1	1.1	0.9;
	9	2	125	50	0	0	1	1	0	345	1	1.1	0.9;
];
```

Figure 3: Matpower-like data format

Re-coding

Models

The models are written in JuMP and consist in roughly 3 groups of equations. The first group consist in capturing the physics and limitations of the electrical network (in A&H integrated model, they rely on the model specified in matpower), the physics of the gas networks (specified by A&H in their own equations) and finally, the links between networks.

The problem is then solved using interior point methods by the solver (Ipopt) without the need of manually specifying and creating vectors with derivatives (Jacobians) or any other symbolic or numerical routine. All is done efficiently by the solver.

Re-coding
Models (cont.)

An extract of the electrical model can be seen here:

```
# CONSTRAINTS
@constraints(m, begin
    VoltageLimit[b in sB, t in sT], Data.Buses[b].vB_lb <= v[b,t] <= Data.Buses[b].vB_ub;
    DeltaRef[t in sT],delta[BusType[3][1],t]==0.0;
end);
@NLconstraints(m,begin
    PowerBalance1[b in sB, t in sT], sum(pG[g,t] for g in sGcon if Data.Generators[g].AtBus==b) - sum(pL[l,t] for l in sLco
    PowerBalance2[b in sB, t in sT], sum(qG[g,t] for g in sGcon if Data.Generators[g].AtBus==b) - sum(qL[l,t] for l in sLco
end);

if !isempty(sLcon)
    @NLconstraints(m, begin
    BFlow1[l in sLcon,t in sT],pL[l,t]== (1/Data.Lines[l].Ratio)*-v[Data.Lines[l].From,t]*v[Data.Lines[l].To,t]*(lineBG(Dat
    BFlow2[l in sLcon,t in sT],pLt[l,t]== (1/Data.Lines[l].Ratio)*-v[Data.Lines[l].To,t]*v[Data.Lines[l].From,t]*(lineBG(Da
    BFlow3[l in sLcon,t in sT],qL[l,t]== (1/Data.Lines[l].Ratio)*v[Data.Lines[l].From,t]*v[Data.Lines[l].To,t]*(-lineBG(Dat
    BFlow4[l in sLcon,t in sT],qLt[l,t]== (1/Data.Lines[l].Ratio)*v[Data.Lines[l].To,t]*v[Data.Lines[l].From,t]*(-lineBG(Da
    end);
    if !isempty(sLlim)
        @NLconstraints(m, begin
            LineLimit1[l in intersect(sLlim,sLcon),t in sT],pL[l,t]^2+qL[l,t]^2<= Data.Lines[l].Sl_ubA^2;
            LineLimit2[l in intersect(sLlim,sLcon),t in sT],pLt[l,t]^2+qLt[l,t]^2<= Data.Lines[l].Sl_ubA^2;
        end);
    end
end

end
```

Figure 4: Electrical network contraints (extract)

Re-coding
Demonstration

Live walktrough the code and its execution

(Show Julia/JuMP code vs. original code to show the dramatic difference in complexity)

Re-coding

Discussion

How can be used and what is needed to take it forward?

(Discussion by Chris, ask him for some points here.)