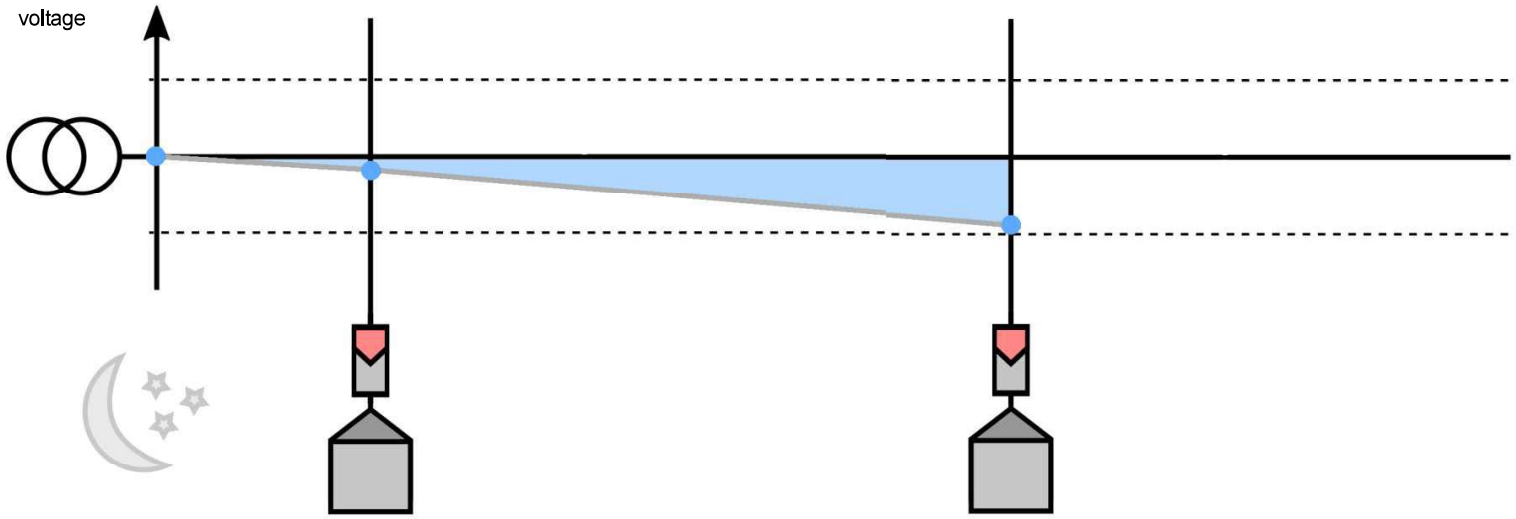
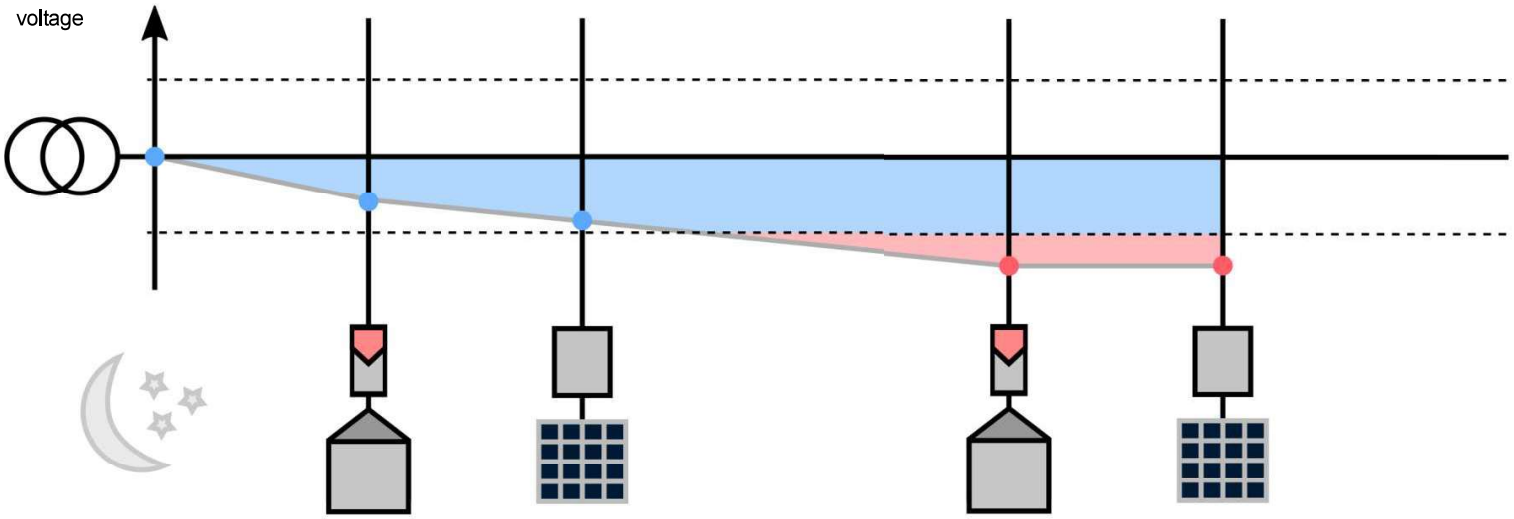
# From simulation to optimization models

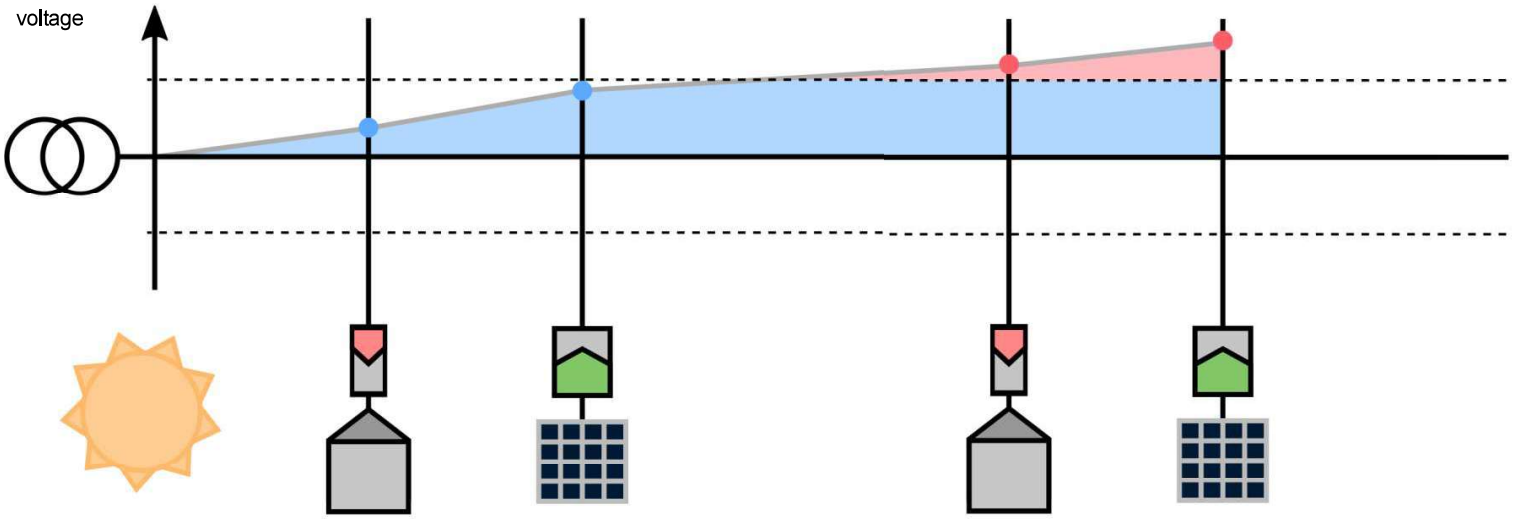
# distribution networks



# distribution networks

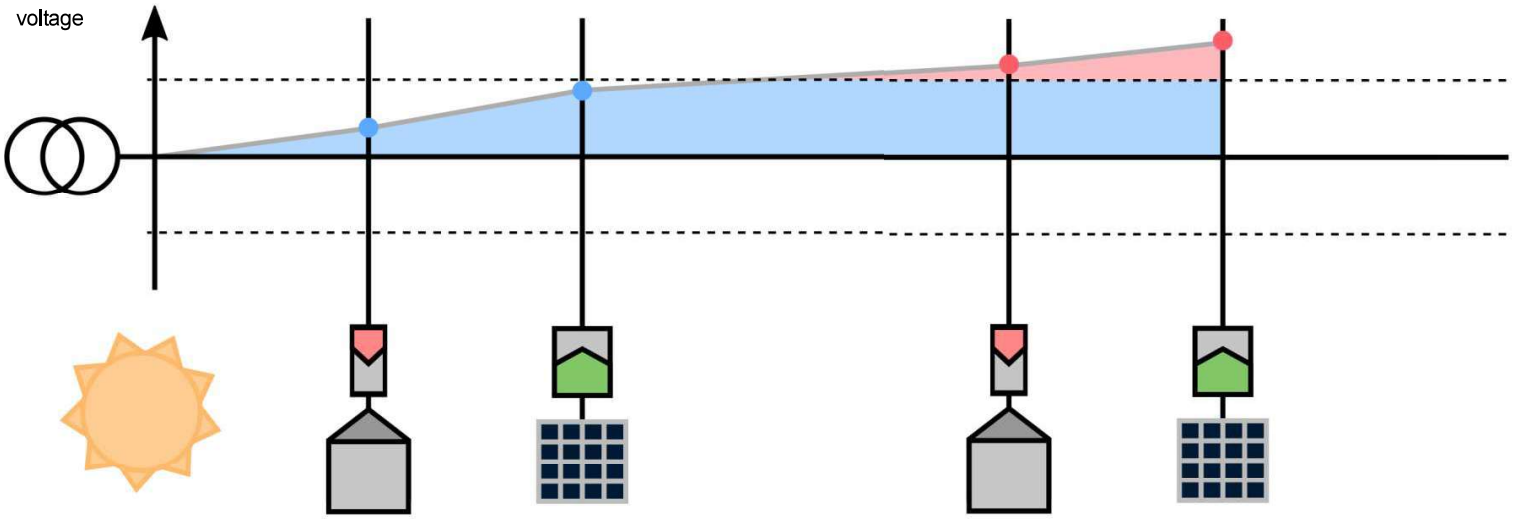


# distribution networks



# distribution networks

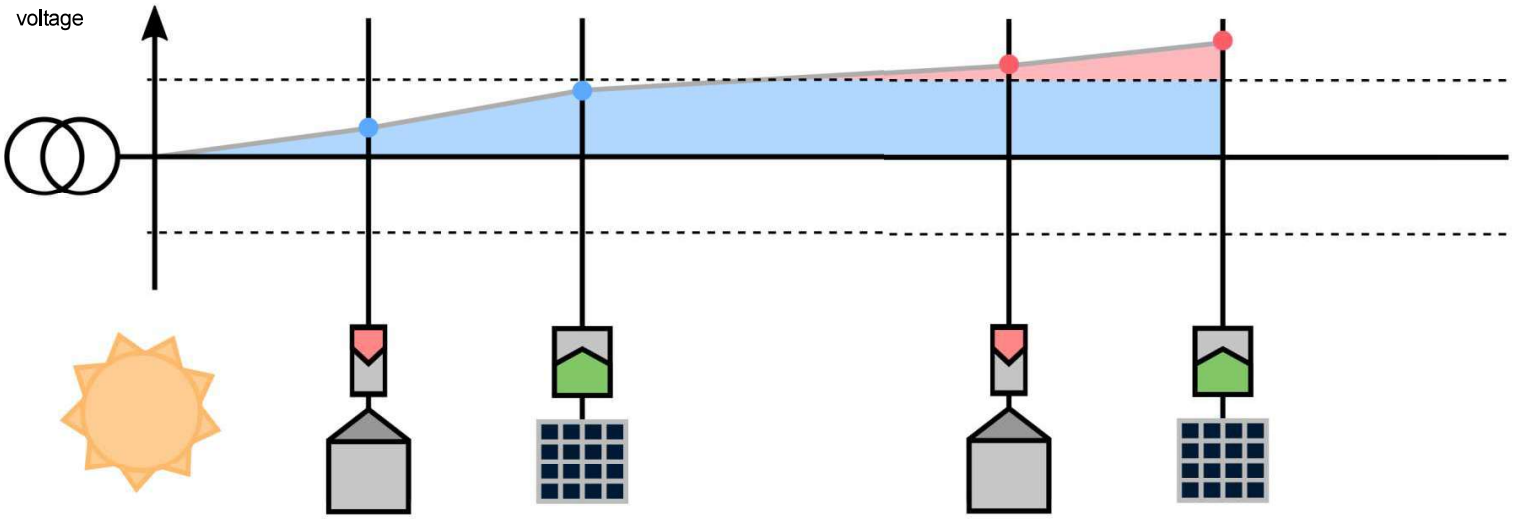
**PASSIVE SOLUTION**
**reinforce the network (more copper)**



# distribution networks

**ACTIVE**

**ACTIVE SOLUTION**
- PV curtailment
- EV charging coordination
- **batteries**
- ...

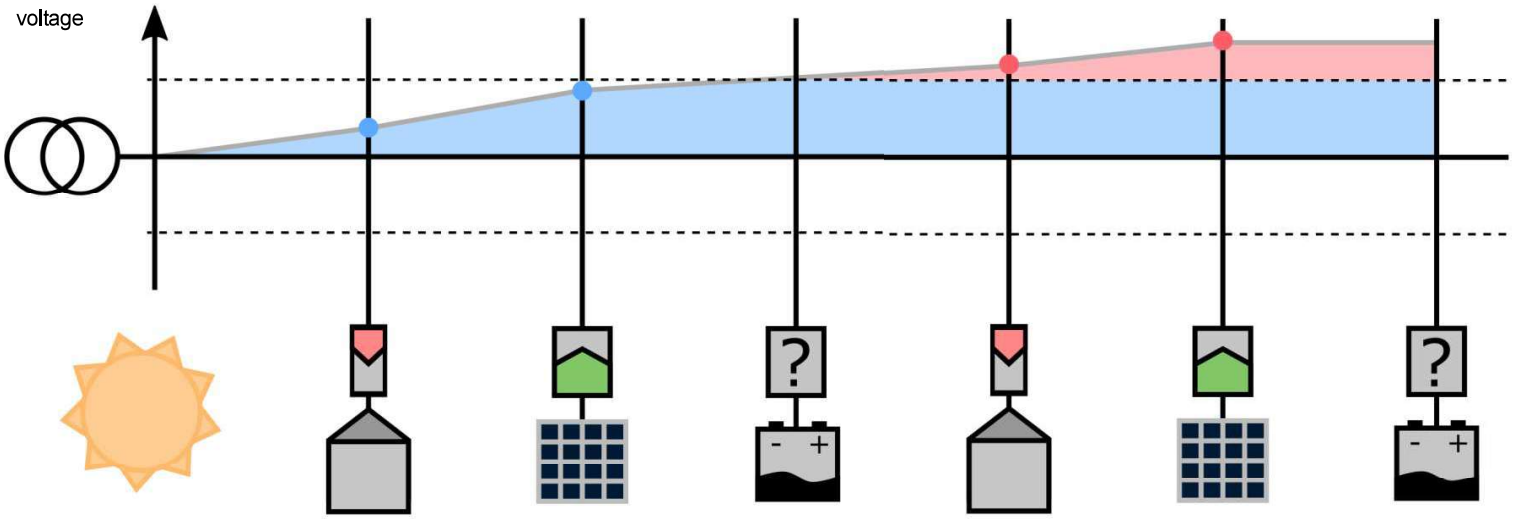



**in presence of distributed control, e.g. volt-var/watt PV inverters**

# distribution networks

## ACTIVE

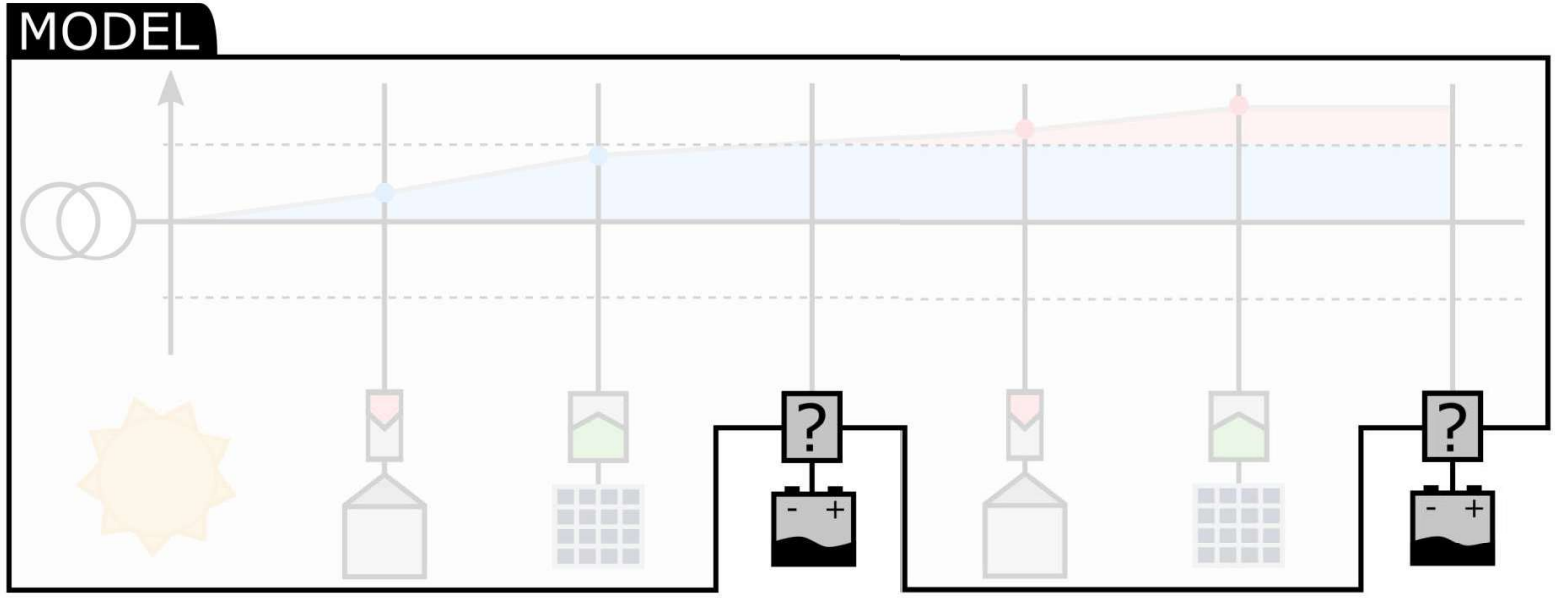
**ACTIVE SOLUTION**
- PV curtailment
- EV charging coordination
- **batteries**
- ...



# distribution networks

**ACTIVE**

**ACTIVE SOLUTION**
- PV curtailment
- EV charging coordination
- **batteries**
- ...

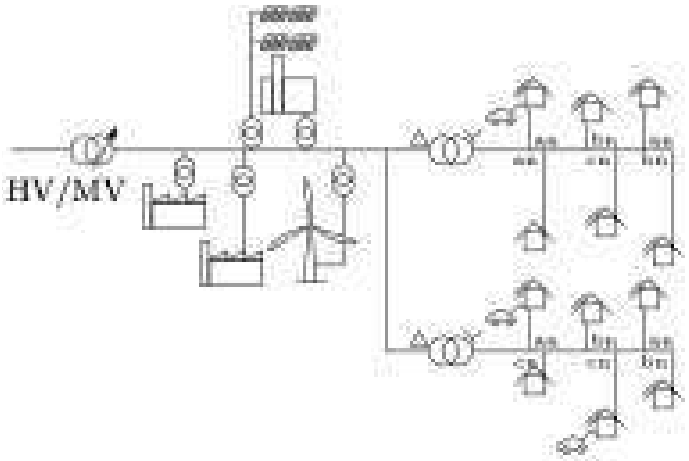


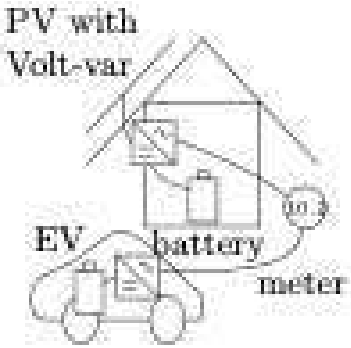
# distribution networks

**ACTIVE**

**ACTIVE SOLUTION**
- PV curtailment
- EV charging coordination
- **batteries**
- ...

# OPF?




OPF example

min. cost of dispatch

s.t. network physics
- operating limits: S, V, I
- devices: EV, PV, battery
- inverter control

outcome:
- feasible dispatch &
- network within
- operating limits

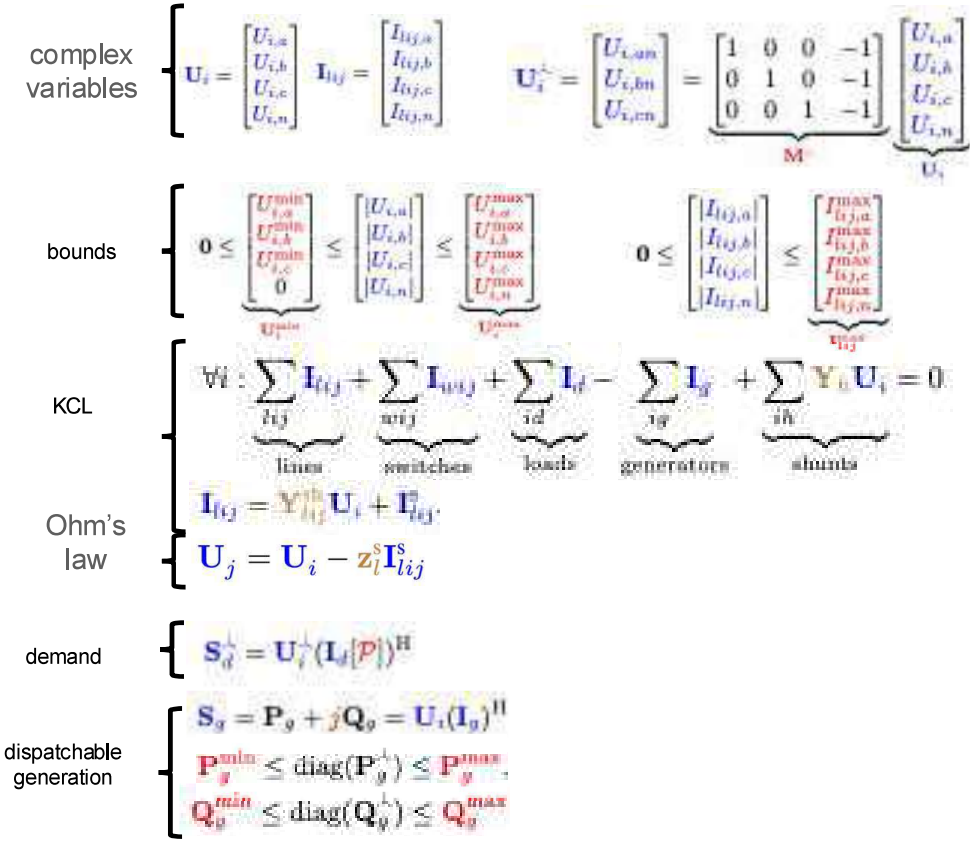
# Feasible set of canonical problem

System of complex-value nonlinear matrix equations

- In practice additional complexity to avoid padding for lines with fewer wires

Reformulations in lifted variables (I -> $UI^H=S$, V->$VV_H=W$) may enable better linearization or convexification

Transformers need detailed treatment, delta-connected loads etc



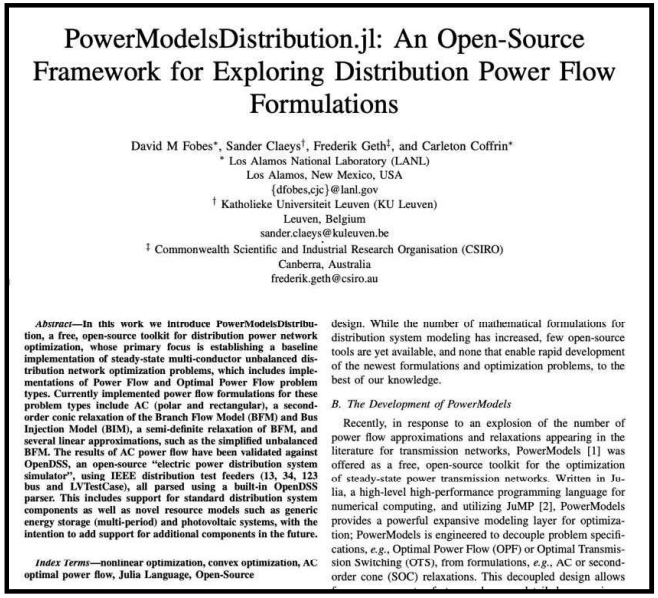
# PowerModelsDistribution.jl

Toolbox with 3-wire and 4-wire *optimization* models in various variable spaces

Supports majority of components from OpenDSS, OpenDSS parser for base case definitions

Supports multiperiod optimization too

Separates data, models and algorithms



PowerModelsDistribution.jl: An Open-Source Framework for Exploring Distribution Power Flow Formulations

David M Fobes*, Sander Claeys†, Frederik Geth‡, and Carleton Coffrin*
* Los Alamos National Laboratory (LANL)
Los Alamos, New Mexico, USA
{dfobes,cjc}@lanl.gov
† Katholieke Universiteit Leuven (KU Leuven)
Leuven, Belgium
sander.claeys@kuleuven.be
‡ Commonwealth Scientific and Industrial Research Organisation (CSIRO)
Canberra, Australia
frederik.geth@csiro.au

***Abstract*—In this work we introduce PowerModelsDistribution, a free, open-source toolkit for distribution power network optimization, whose primary focus is establishing a baseline implementation of steady-state multi-conductor unbalanced distribution network optimization problems, which includes implementations of Power Flow and Optimal Power Flow problem types. Currently implemented power flow formulations for these problem types include AC (polar and rectangular), a second-order conic relaxation of the Branch Flow Model (BFM) and Bus Injection Model (BIM), a semi-definite relaxation of BFM, and several linear approximations, such as the simplified unbalanced BFM. The results of AC power flow have been validated against OpenDSS, an open-source "electric power distribution system simulator", using IEEE distribution test feeders (13, 34, 123 bus and LVTestCase), all parsed using a built-in OpenDSS parser. This includes support for standard distribution system components as well as novel resource models such as generic energy storage (multi-period) and photovoltaic systems, with the intention to add support for additional components in the future.**

***Index Terms*—nonlinear optimization, convex optimization, AC optimal power flow, Julia Language, Open-Source**

design. While the number of mathematical formulations for distribution system modeling has increased, few open-source tools are yet available, and none that enable rapid development of the newest formulations and optimization problems, to the best of our knowledge.

*B. The Development of PowerModels*

Recently, in response to an explosion of the number of power flow approximations and relaxations appearing in the literature for transmission networks, PowerModels [1] was offered as a free, open-source toolkit for the optimization of steady-state power transmission networks. Written in Julia, a high-level high-performance programming language for numerical computing, and utilizing JuMP [2], PowerModels provides a powerful expansive modeling layer for optimization; PowerModels is engineered to decouple problem specifications, *e.g.*, Optimal Power Flow (OPF) or Optimal Transmission Switching (OTS), from formulations, *e.g.*, AC or second-order cone (SOC) relaxations. This decoupled design allows

| | | | | LVTestCase | |
|---|---|---|---|---|---|
| | IEEE13 | IEEE34 | IEEE123 | t=500 | t=1000 |
| $\delta$ | 5.1E-8 | 1.4E-7 | 1.3E-8 | 3.2E-8 | 3.3-8 |
| $\min \lvert U \rvert_{ip}$ | 0.9750 | 0.9166 | 0.9858 | 1.0353 | 1.0226 |
| $\max \lvert U \rvert_{ip}$ | 1.0686 | 1.0500 | 1.0437 | 1.0499 | 1.0496 |

- <1E-7 relative error in all variables!

CRICOS code 00025B

# Nondimensionalisation

- Physical equations ($V_j = V_i - Z_l I_{lij}$) have units attached
  - $V_i, V_j - \mathrm{V}, \mathrm{Z_l} - \Omega, \mathrm{I_{lij}} - \mathrm{A}$
  - I.e. so do the circuit laws
- Most Linear algebra routines work with floating point numbers *without* units
- Libraries may have support for propagating units
  - E.g. https://github.com/JuliaPhysics/Unitful.jl works together with JuMP in https://github.com/trulsf/UnitJuMP.jl
  - This comes at some processing and code complexity cost
- When we do computations using standard linear algebra routines, we have to do the nondimensionalization ourselves.
  - $V_j/1V = V_i/1V - \left(\frac{Z_l}{1\Omega}\right)\left(\frac{I_{lij}}{1A}\right)$ this is now a purely mathematical equation
  - Pick a reference value for each SI base unit (that you need) and derive all other bases for all quantities you use $\frac{V_j}{V_{base}} = \frac{V_i}{V_{base}} - \left(\frac{Z_l}{Z_{base}}\right)\left(\frac{I_{lij}}{I_{base}}\right), V_{base} = x\,V, Z_{base} = y\Omega, \mathrm{I_{base}} = \frac{\mathrm{x}}{\mathrm{y}}\mathrm{A}$
  - Divide all quantities by their corresponding bases

# Nondimensionalisation vs per-unit

Bases can massively impact the numerical magnitudes

- E.g. Normalizing by 1 J in the context of power systems leads to high magnitudes, generally we think in terms of a MW x b h, i.e. a 3.6E9 J base
- 1 V vs 1 kV, 1s vs 1h, …

Per unit conversion is a specific choice of nondimensionalization based on common engineering practice / culture

- In power systems we 1) choose a power base 2) match the voltage base to the expected voltage / reference voltages, often line-to-line 3) pick time base of 1h 4) derive all other units from those.
- OpenDSS does not do any calculations in per unit, it calculates everything in V/A!
  - Ax=b type solvers are very numerically robust (e.g. KLU in Sundials.jl)
  - Some transformer parameters are specified in pu of their own power rating and primary voltage, this gets turned back into SI quantities.
- PMD does use nondimensionalisation, as we believe this may still be advantageous for numerical *optimization* solvers
- Note that admittance matrices across multiple voltage levels have very different magnitudes for the entries, and even more so if there are data artefacts (lines with superfluous buses)

# optimisation problems

- minimise cost (function)
- subject to
  - mathematical models for entities
    - laws of physics
    - technological envelopes
    - financials
      - CAPEX
      - OPEX

objective

constraint set:
linear and nonlinear
equalities and inequalities

# why *mathematical* optimisation?

- without ***structure***, the best approach to solve an optimisation problem is to enumerate all candidate solutions
  - evaluate, score, compare, pick best
- with - mathematical - ***structure***, things can ‘simplify’ a lot

# mathematical structure

- linear
- polynomial
- set inclusion
- integrality
- logical conditions

$$Ax = b$$

$$x^2 + y^2 + z^2 - 2xyx - 1 = 0$$

$$x \in \mathbb{S}^n_+$$

$$x \in \{0, 1\}$$

$$(x \geq 2 \wedge y = 1) \vee (x \leq y \leq 2)$$

# mathematical optimisation

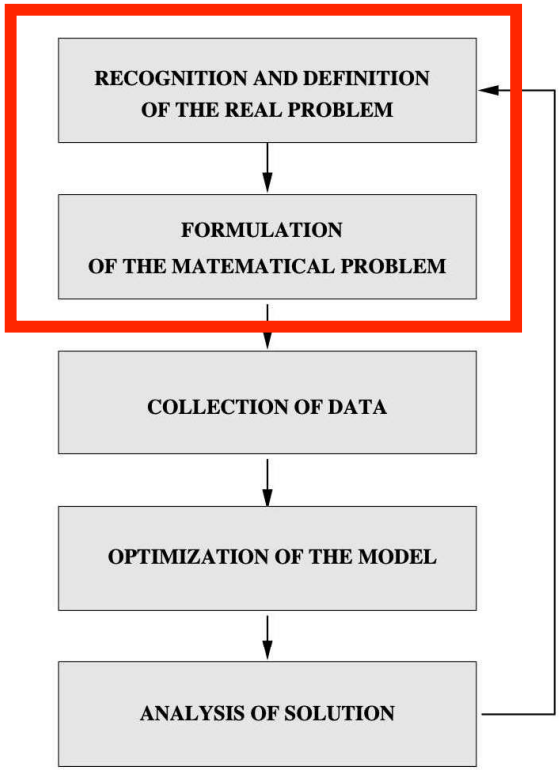
$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g_i(x) \leq 0 \\ & h_j(x) = 0 \end{aligned}$$

- decision variables (vector): $x \in \mathbb{R}^n$
- functions $f, g_i, h_j : \mathbb{R}^n \to \mathbb{R}$
  - what is their nature?
- # constraints and variables is typically finite $i \in \{1, \ldots, m\}, j \in \{1, \ldots, l\}$

# What is optimisation modelling?

# Optimisation problem modelling process

modeling task




good habit to separate

- model
- data
- solving

Fragni, E., & Gondzio, J. (1999). Optimization modeling languages.

# Mathematical model of a problem

- minimum cost flow problem
  - over a graph
    G = (V, E) = (vertices, edges)
  - V = {1,2, …n} (vertices)
  - source at vertex 1
  - sink at vertex n

$$\min_x \sum_{(i,j)\in E} c_{i,j}x_{i,j}$$

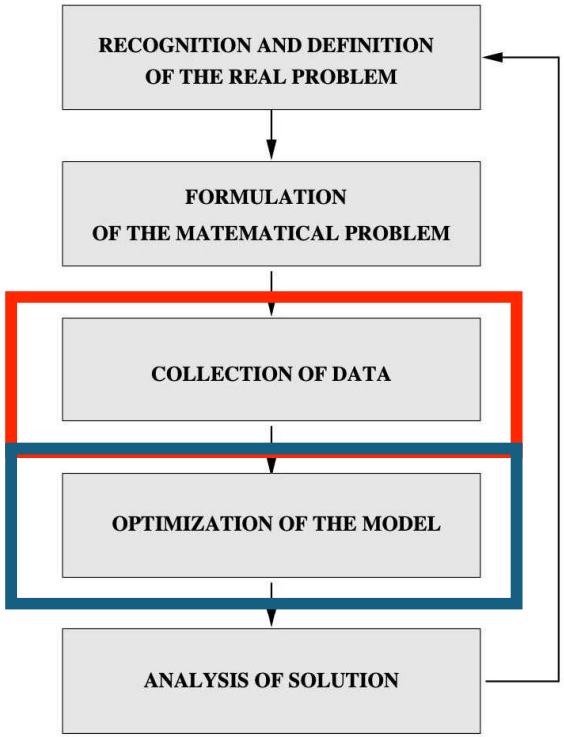
$$\text{s.t.} \sum_{(i,j)\in E} x_{i,j} = \sum_{(j,k)\in E} x_{j,k} \quad \forall j \in V \setminus \{1,n\}$$

$$\sum_{(i,n)\in E} x_{i,n} = 1$$

$$0 \le x_{i,j} \le C_{i,j} \quad \forall (i,j) \in E$$

Dunning, I., Huchette, J., & Lubin, M. (2015). JuMP: a modeling language for mathematical optimization, 1–21. Retrieved from http://arxiv.org/abs/1508.01982

# Optimisation problem modelling process




data interfacing

ask solver kindly

good habit to separate

- model
- data
- solving

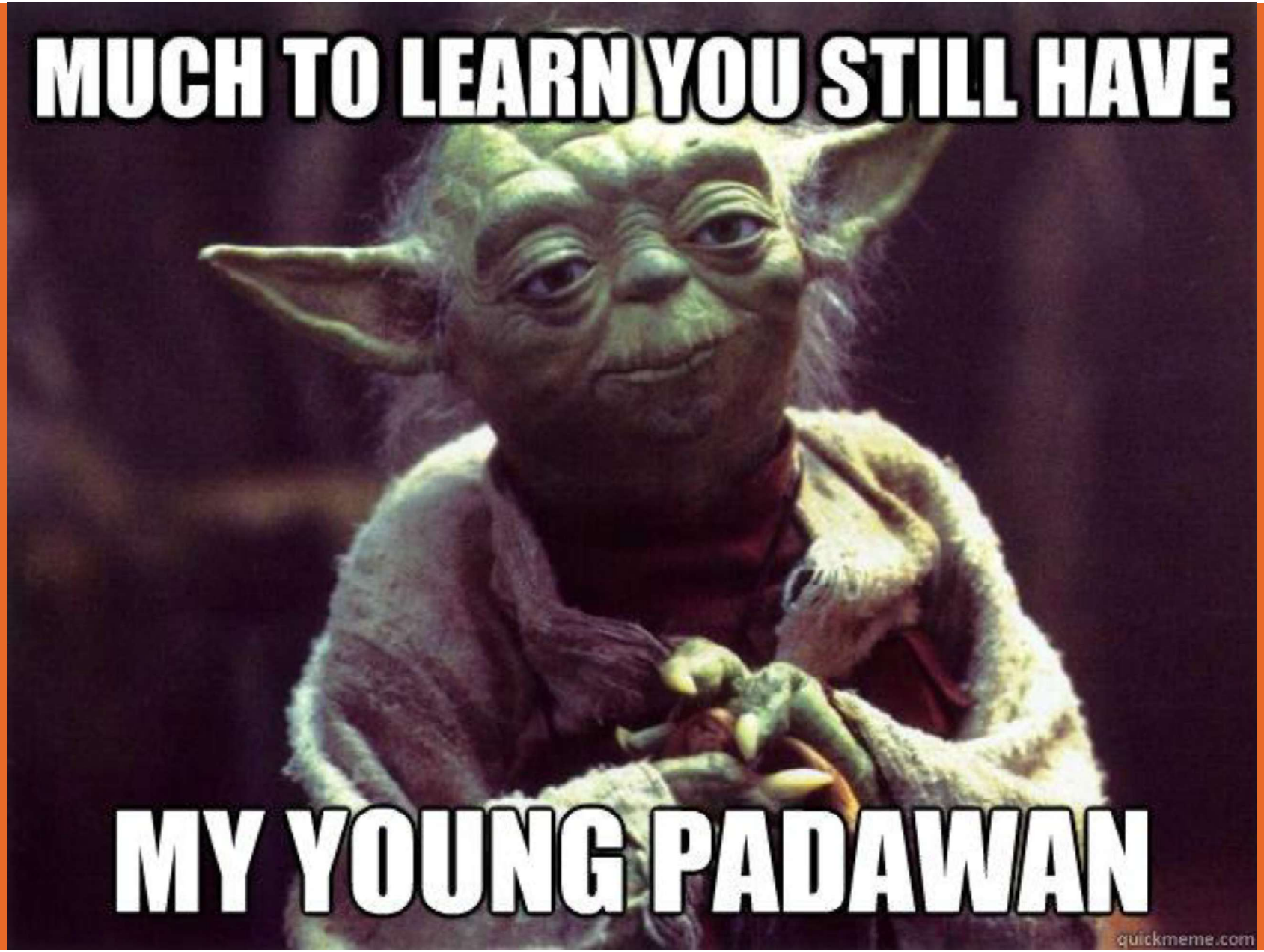
Fragni, E., & Gondzio, J. (1999). Optimization modeling languages.

# declarative programming

- a style of building the structure and elements of computer programs
  - expresses the *logic of a computation* **without describing its control flow** (think: if then else)
  - describing what the program must accomplish in terms of the problem domain
  - **not**: describe how to accomplish it as a sequence of programming language primitives

# Learning JuMP by example


- James D. Foster
- Github: @jd-foster


MUCH TO LEARN YOU STILL HAVE
MY YOUNG PADAWAN
quickmeme.com

# A Modelling and Simulation Computation Process

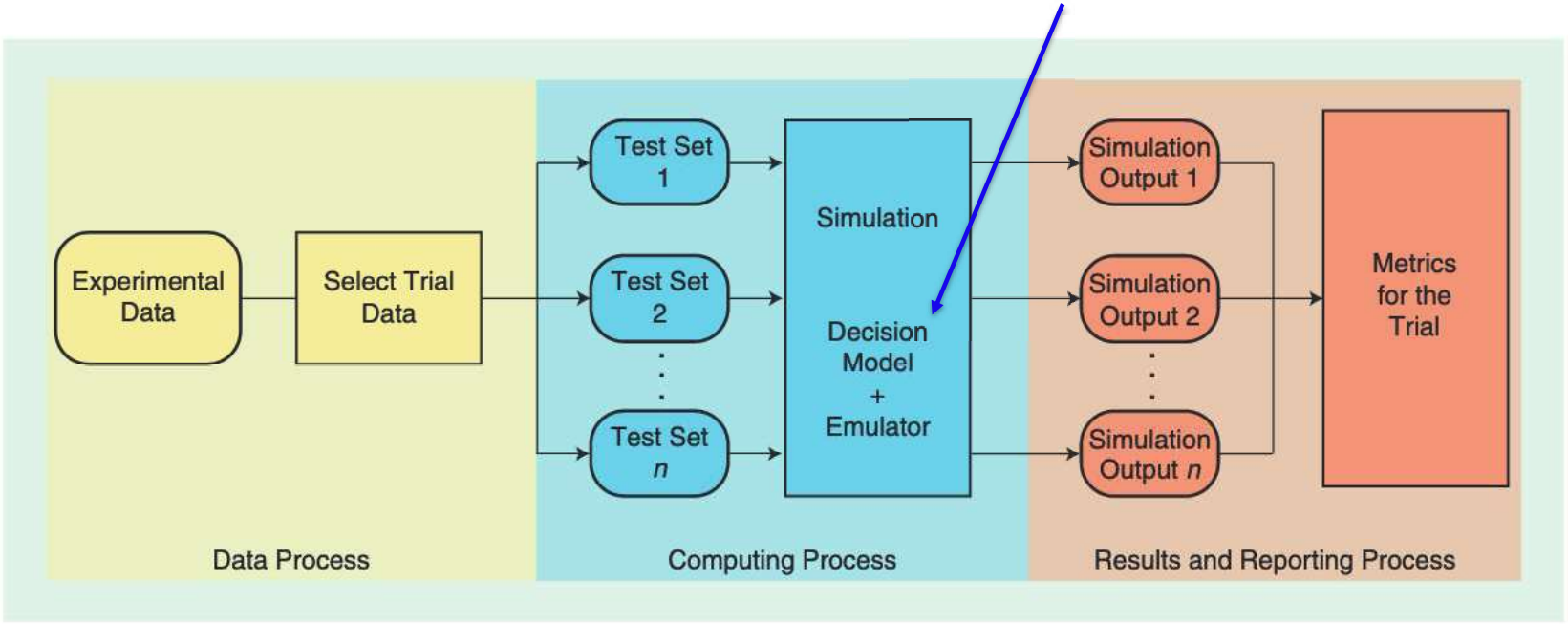
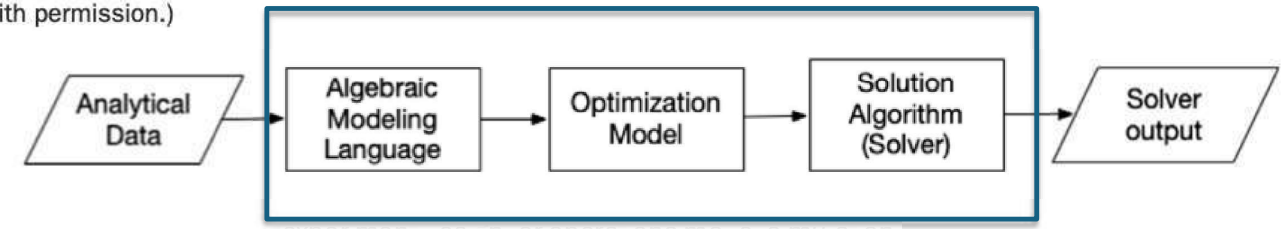



**Figure 1.** The computing steps required for conducting a single trial in a simulation experiment. These steps follow the best practices in scientific computing to achieve reproducibility and validation: data, computational modeling, and results subprocesses. (Adapted from Lara et al. 2020; used with permission.)



# What are Algebraic Modelling Languages

… such as JuMP?

- They are high-level **domain-specific languages** for succinctly describing decision models.
- Use algebra and mathematical functions to create *software* models that correspond well to your *mental* model.
- Their strength is that they separate the **descriptive** formulation of a model from the **process** of determining a solution or optimal decision variables.
- The **model formulation** stage involves defining decision variables and algebraic constraints on those variables.
- Good for decision-making with **simultaneous** (and conflicting) requirements.
- They mostly **automate** any data transformations that are necessary.
- They can be used to create **readable, reproducible and easily maintainable components** in a larger model or simulation.

# Algebraic Modelling Components

- Objective function: a “benefit” to maximise, or a “cost” to minimise
  - Maximise throughput, maximise expected value, minimise waste, minimise deviation

- Sets (for indexing or organising model components)
  - e.g. lists: {1, 2, 3, …}
  - key strings: {“Factory1”, “Retailer5”,…}

- Parameters (fixed data)
  - e.g. transfer matrix: A[i,j]
  - key-value stores (dictionaries): Capacity[“Factory1”,”Product3”].

- Decision variables (the “unknowns” to solve for)
  - e.g. Production levels: Produced[i]

- Constraints (equations and inequalities)
  - Implicitly describe the allowable “space” of decisions.

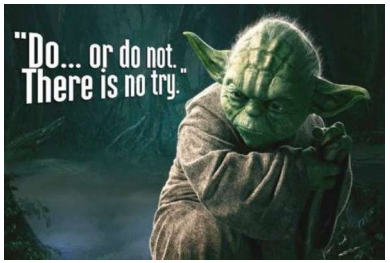
# The classic knapsack problem

$$\max \sum_{i=1}^{n} c_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq b$$

$$x_i \in \{0, 1\} \quad \forall i = 1, \ldots, n.$$

maximise a linear combination of benefits (“profit”)
*provided*
the weighted sum is within budget
*and*
an item can be either included (1) or excluded (0).



# The classic knapsack problem

$$x_i \in \{0, 1\} \quad \forall i = 1, \ldots, n,$$

$$\max \sum_{i=1}^{n} c_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq b$$

```
using JuMP   # modelling language package
import HiGHS  # solver interface

function example_knapsack(; verbose = true)
    profit = [5, 3, 2, 7, 4]
    weight = [2, 8, 4, 2, 5]
    capacity = 10

    model = Model(HiGHS.Optimizer)

    @variable(model, x[1:5], binary = true)

    # Objective: maximise profit
    @objective(model, Max, profit' * x)

    # Constraint: can carry all
    @constraint(model, weight' * x <= capacity)

    # Solve problem using MIP solver
    optimize!(model)
end
example_knapsack()
```

# Finding solutions (I)

- The software implementation of the solution algorithm is called a **solver**.
  - Delegation of responsibility for numerical implementation, precision, tolerances and so on.
  - Can be “tuned” to a particular use case by adjust many algorithmic parameters.
- The AML “compiles” a model down to the exact data inputs that a solver needs in order to run.

# Finding solutions (II)

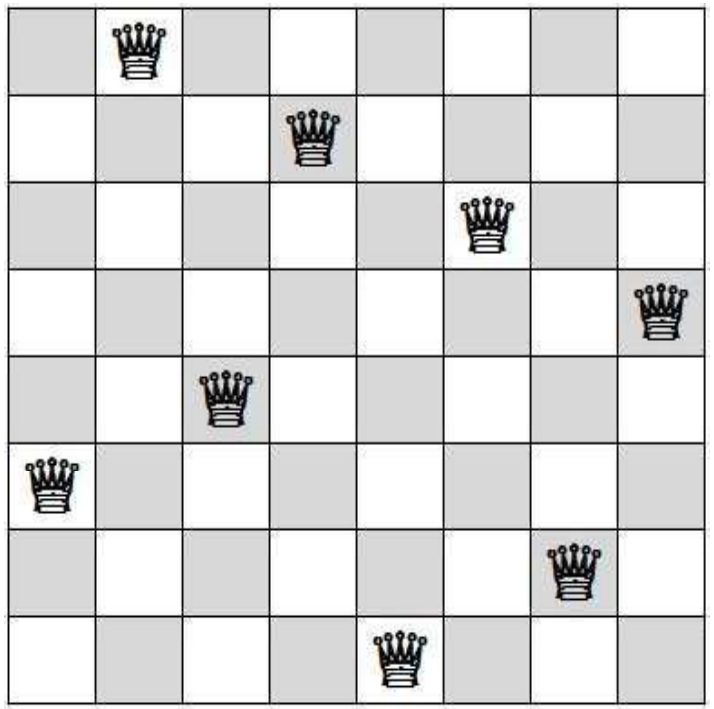
- Access powerful solution algorithms with **interfaces**
  - The user describes the data to the algorithm, but leaves the solution process to the algorithm.
  - A key reason of the abstraction for algebraic modelling languages is to be solver-independent: you can change to solver to a **different, more effective** solver if need be.
  - The translation interface between AML and solver must be specified.
  - There is a lot of “intelligence” applied to pre-processing the algebraic description:
  - Worry less about coming up with the “perfect” way to express the model
  - Work with what feels more natural.
- Writing modelling extensions
  - Two main approaches are new algebraic forms, or the user-defined functions derived within a general purpose programming language.
  - But, by *giving up* some of the full generality of a programming language function calls, you get highly targeted solution algorithms for your problem class.

# A tour of examples in the JuMP docs

- Problem types
  - Combinatorial
  - Continuous
- Applications
- Recent additions

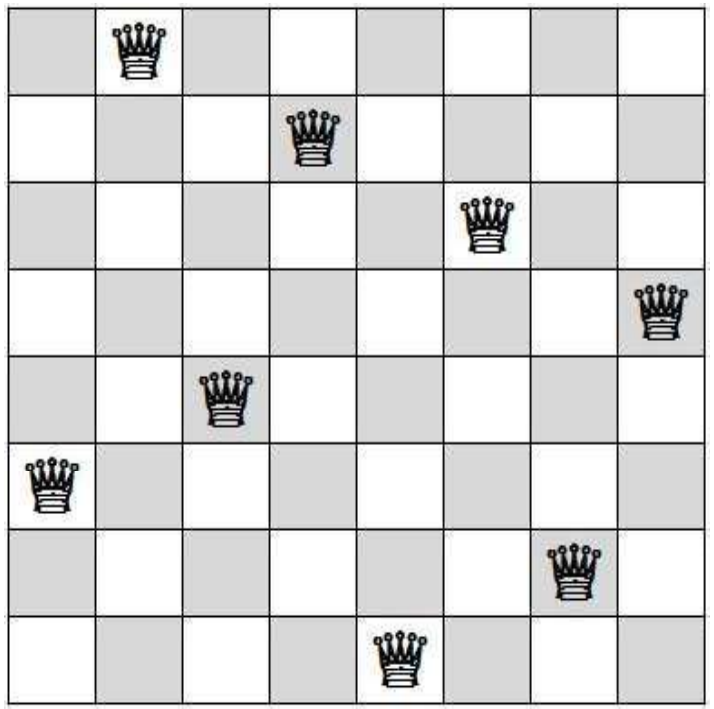
# N-Queens

```
using JuMP
import HiGHS
import LinearAlgebra

N = 8
model = Model(HiGHS.Optimizer)

@variable(model, x[1:N, 1:N], binary = true);
# There must be exactly one queen in a given row/column:
for i in 1:N
    @constraint(model, sum(x[i, :]) == 1)
    @constraint(model, sum(x[:, i]) == 1)
end
# There can only be one queen on any given diagonal:
for i in -(N - 1):(N-1)
    @constraint(model, sum(diag(x, i)) <= 1)
    @constraint(model, sum(diag(reverse(x; dims = 1), i)) <= 1)
end
optimize!(model)
```



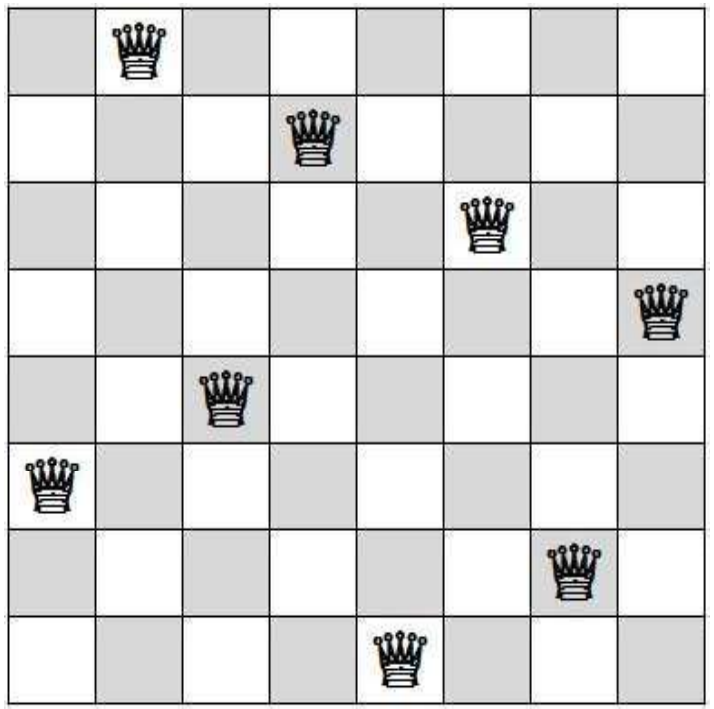
# N-Queens

```
julia> solution = round.(Int, value.(x))

8×8 Matrix{Int64}:
 0  0  0  0  1  0  0  0
 0  0  1  0  0  0  0  0
 0  0  0  0  0  0  0  1
 0  0  0  1  0  0  0  0
 0  0  0  0  0  0  1  0
 1  0  0  0  0  0  0  0
 0  0  0  0  0  1  0  0
 0  1  0  0  0  0  0  0
```



# N-Queens

```
julia> solution = round.(Int, value.(x))

8×8 Matrix{Int64}:
 0  0  0  0  1  0  0  0
 0  0  1  0  0  0  0  0
 0  0  0  0  0  0  0  1
 0  0  0  1  0  0  0  0
 0  0  0  0  0  0  1  0
 1  0  0  0  0  0  0  0
 0  0  0  0  0  1  0  0
 0  1  0  0  0  0  0  0
```



# Sudoku

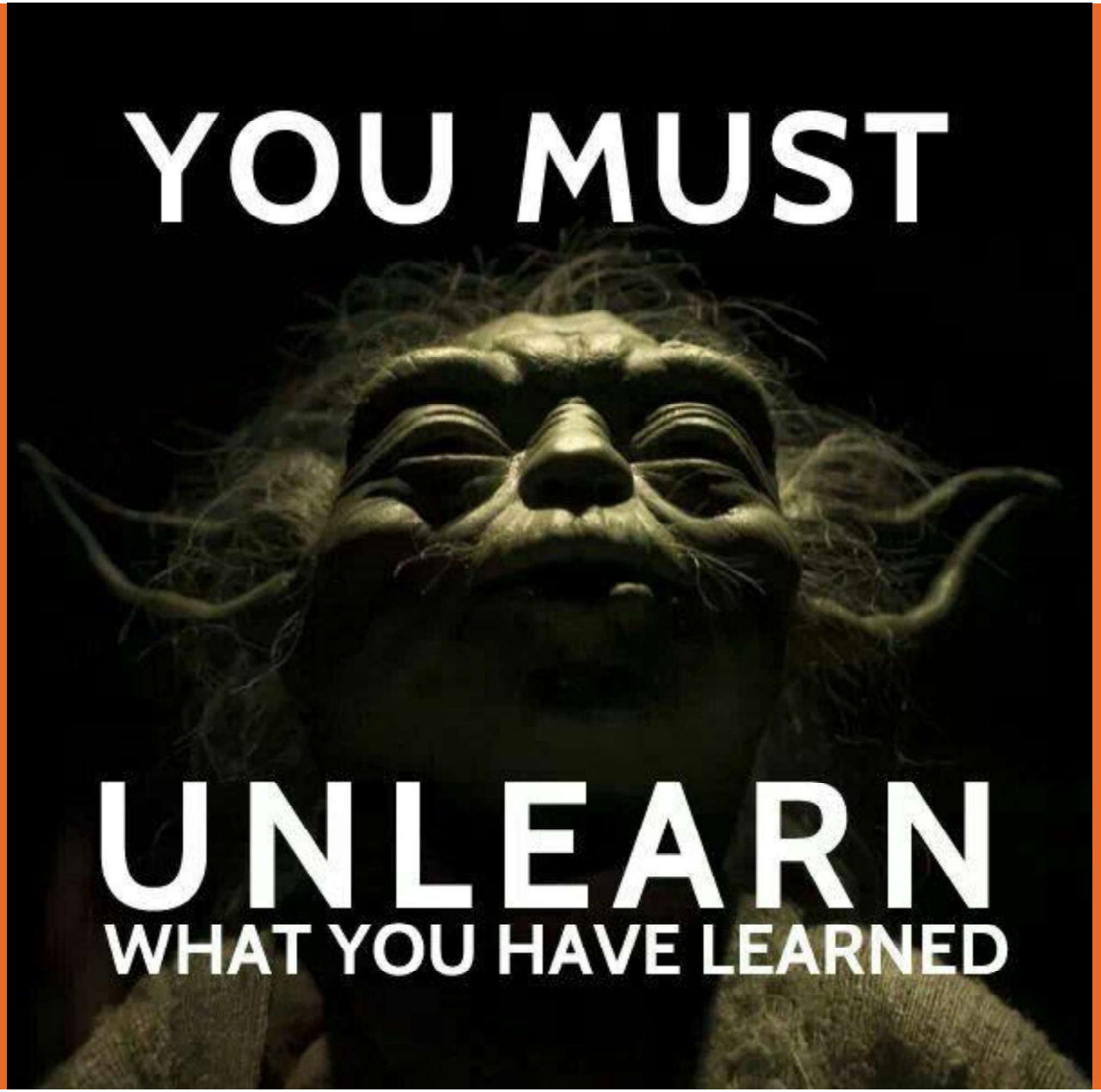
```
using JuMP
using HiGHS

sudoku = Model(HiGHS.Optimizer)
@variable(sudoku, x[i = 1:9, j = 1:9, k = 1:9], binary = true)

for ind in 1:9  # Each row, OR each column
    for k in 1:9  # Each digit
      # Sum across columns (j) - row constraint
      @constraint(sudoku, sum(x[ind, j, k] for j in 1:9) == 1)
      # Sum across rows (i) - column constraint
      @constraint(sudoku, sum(x[i, ind, k] for i in 1:9) == 1)
    end
end

# Enforce the constraint that each digit appears
# once in each of the nine 3x3 sub-grids

# i is the top left row, j is the top left column.
for i in 1:3:7
    for j in 1:3:7
        for k in 1:9
     # We'll sum from i to i+2, for example, i=4, r=4, 5, 6.
     @constraint(sudoku,
       sum(x[r, c, k] for r in i:(i+2), c in j:(j+2)) == 1)
        end
    end
end
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

# Sudoku

```
init_sol = [
     5 3 0 0 7 0 0 0 0
     6 0 0 1 9 5 0 0 0
     0 9 8 0 0 0 0 6 0
     8 0 0 0 6 0 0 0 3
     4 0 0 8 0 3 0 0 1
     7 0 0 0 2 0 0 0 6
     0 6 0 0 0 0 2 8 0
     0 0 0 4 1 9 0 0 5
     0 0 0 0 8 0 0 7 9
]

for i in 1:9
    for j in 1:9
        # If the space isn't empty
        if init_sol[i, j] != 0
# Then the corresponding variable for that digit and location
must be 1.
          fix(x[i, j, init_sol[i, j]], 1; force = true)
        end
    end
end

optimize!(sudoku)
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |


YOU MUST
UNLEARN
WHAT YOU HAVE LEARNED

# Constraint programming toolkit

- AllDifferent
  - every element in a list takes a different integer value.
- BinPacking
  - divide up a set of items into different groups, such that the sum of their weights does not exceed the capacity of a bin.
- Circuit
  - construct a *tour* of a list of N variables
- CountAtLeast
  - at least n elements in a set of variables belong to a set of values.
- CountBelongs
  - count how many elements in a set of variables belong to a set of values.
- CountDistinct
  - count the number of *distinct* elements in a set of variables.
- CountGreaterThan
  - strictly upper-bound the number of distinct elements in a set of variables that have a value equal to another variable.
- Table
  - constrain a vector to exactly match one of the rows in a table.

# Sudoku:
# constraint programming and the *all-different* constraint

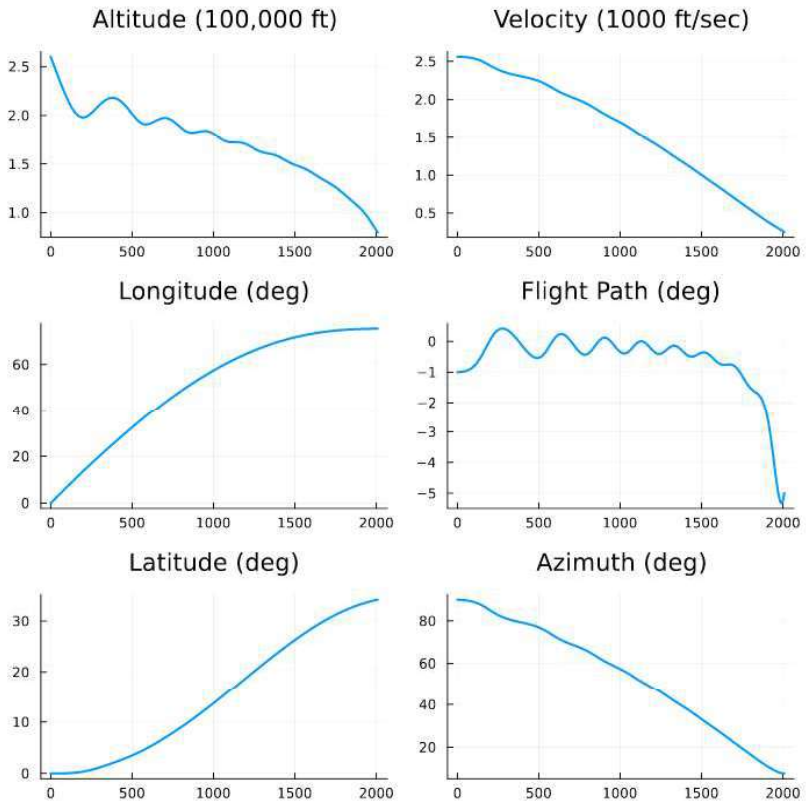
```
model = Model(HiGHS.Optimizer)

@variable(model, 1 <= x[1:9, 1:9] <= 9, Int);

@constraint(model, [i = 1:9],
                x[i, :] in MOI.AllDifferent(9));

@constraint(model, [j = 1:9],
                x[:, j] in MOI.AllDifferent(9));

for i in (0, 3, 6), j in (0, 3, 6)
    @constraint(model,
  vec(x[i.+(1:3), j.+(1:3)]) in MOI.AllDifferent(9))
end
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

Including automatic reformulation to the linear programming solver.

# Optimal control for a Space Shuttle reentry trajectory

```
using JuMP
import Interpolations
import Ipopt
…
# Aerodynamic and atmospheric forces on the vehicle
const ρ₀ = 0.002378
const hᵣ = 23800.0
const Rₑ = 20902900.0
const μ = 0.14076539e17
const S = 2690.0
const a₀ = -0.20704
…

model = Model(optimizer_with_attributes(Ipopt.Optimizer, user_options...))

@variables(model, begin
    0 ≤ scaled_h[1:n]                    # altitude (ft) / 1e5
    ϕ[1:n]                               # longitude (rad)
    deg2rad(-89) ≤ θ[1:n] ≤ deg2rad(89)  # latitude (rad)
    1e-4 ≤ scaled_v[1:n]                 # velocity (ft/sec) / 1e4
    deg2rad(-89) ≤ γ[1:n] ≤ deg2rad(89)  # flight path angle (rad)
    ψ[1:n]                 # azimuth (rad)
    deg2rad(-90) ≤ α[1:n] ≤ deg2rad(90)  # angle of attack (rad)
    deg2rad(-89) ≤ β[1:n] ≤ deg2rad(1)   # bank angle (rad)
    #        3.5 ≤       Δt[1:n] ≤ 4.5   # time step (sec)
    Δt[1:n] == 4.0                       # time step (sec)
end);
…
```

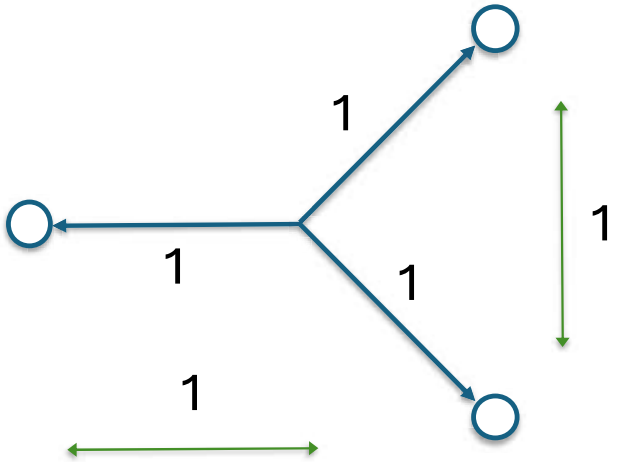


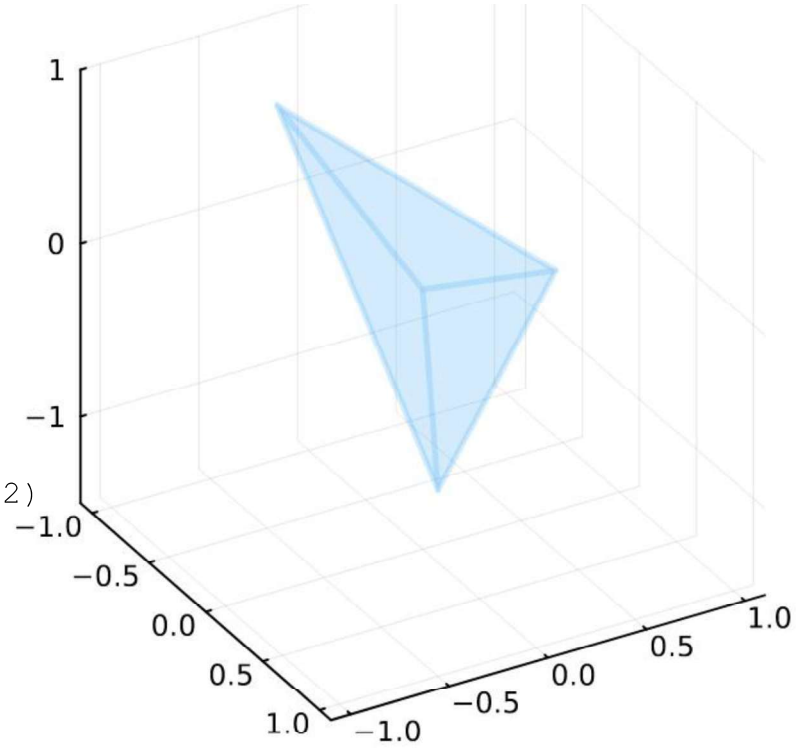
# Simple semidefinite programming (SDP) examples

- Maximum cut via SDP
- K-means clustering via SDP
- The correlation problem
- The minimum distortion problem
- Lovász numbers
- Robust uncertainty sets

# Simple semidefinite programming examples

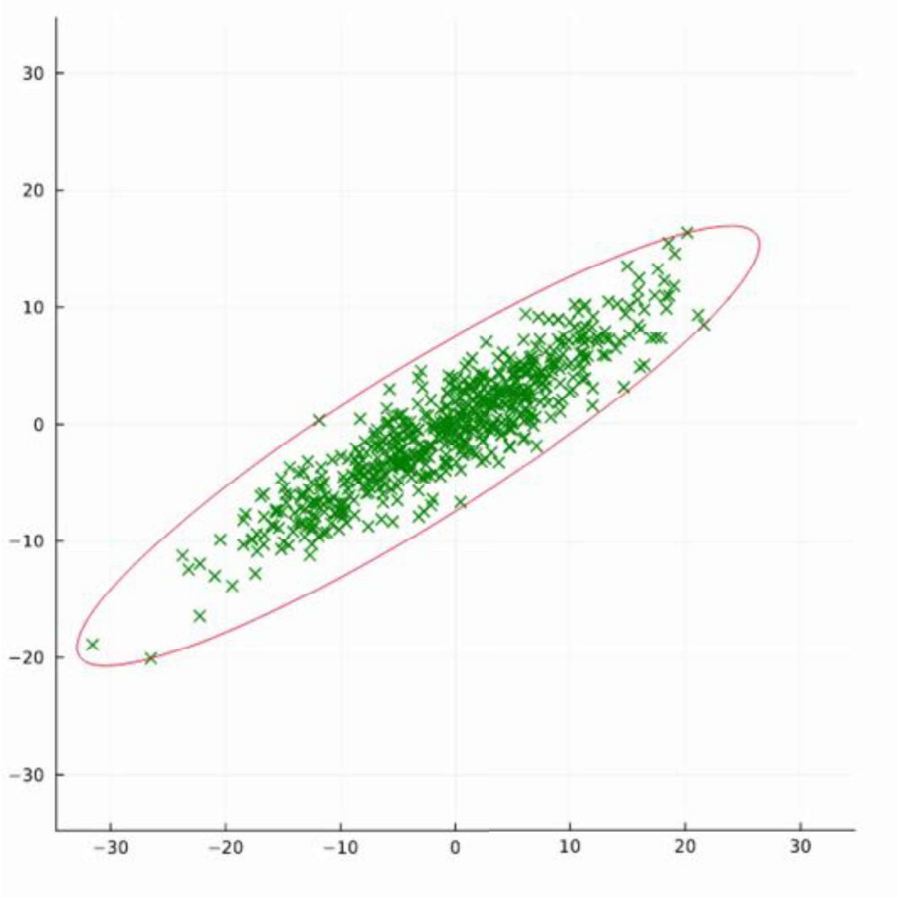
```
model = Model(SCS.Optimizer)

D = [
    0.0 1.0 1.0 1.0
    1.0 0.0 2.0 2.0
    1.0 2.0 0.0 2.0
    1.0 2.0 2.0 0.0
]

@variable(model, c² >= 1.0)
@variable(model, Q[1:4, 1:4], PSD)

for i in 1:4, j in (i+1):4
    @constraint(model, D[i, j]^2 <= Q[i, i] + Q[j, j] - 2 * Q[i, j])
    @constraint(model, Q[i, i] + Q[j, j] - 2 * Q[i, j] <= c² * D[i, j]^2)
end

fix(Q[1, 1], 0)
@objective(model, Min, c²)
optimize!(model)

Test.@test objective_value(model) ≈ 4 / 3 atol = 1e-4

# Recover the minimal distorted embedding:
X = [zeros(3) sqrt(value.(Q)[2:end, 2:end])]
```



# Ellipsoid approximation

```
model = Model(SCS.Optimizer)

S = generate_point_cloud(600);

m, n = size(S)

@variable(model, z[1:n])
@variable(model, Z[1:n, 1:n], PSD)
@variable(model, s)
@variable(model, t)

@constraint(model, [s z'; z Z] >= 0, PSDCone())
@constraint(
    model,
    [i in 1:m],
    S[i, :]' * Z * S[i, :] - 2 * S[i, :]' * z + s <= 1,
)
@constraint(model,
   [t; vec(Z)] in MOI.RootDetConeSquare(n))

@objective(model, Max, t)

optimize!(model)
```



# Solving the dual language problem

## Dualization

The purpose of this tutorial is to explain how to use Dualization.jl to improve the performance of some conic optimization models. There are two important takeaways:

1. JuMP reformulates problems to meet the input requirements of the solver, potentially increasing the problem size by adding slack variables and constraints.
2. Solving the dual of a conic model can be more efficient than solving the primal.

Dualization.jl is a package which fixes these problems, allowing you to solve the dual instead of the primal with a one-line change to your code.

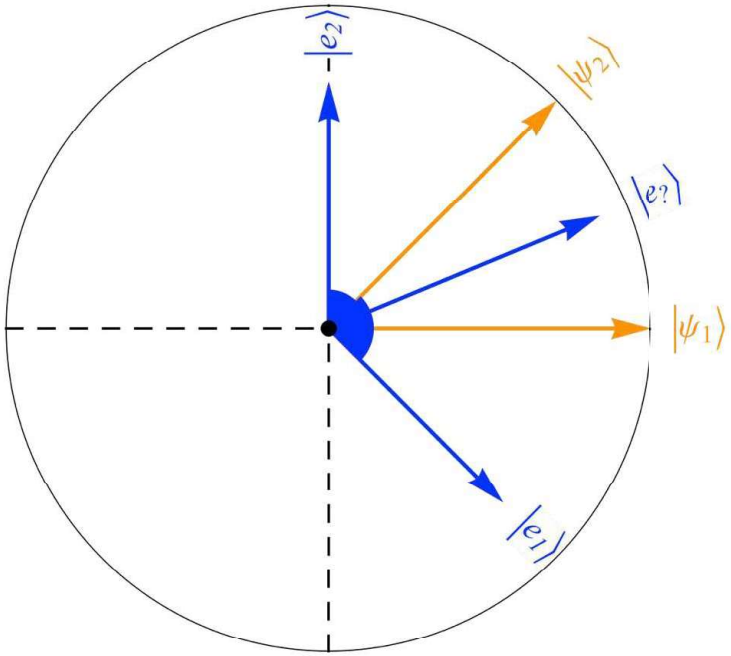
This tutorial uses the following packages

```
using JuMP
import Dualization
import SCS
```

| Model | #PrimalVars | #Cons | #DualVars | #DualCons |
|---|---|---|---|---|
| **Maximum cut** | 10 | 5 | 4 | 1 |
| **K-means** | 21 | 29 | 28 | 22 |
| **Correlation** | 6 | 8 | 7 | 5 |
| **Minimum distortion** | 11 | 15 | 14 | 15 |
| **Theta** | 15 | 7 | 6 | 1 |
| **Robust** | 12 | 4 | 24 | 10 |

# Quantum state discrimination

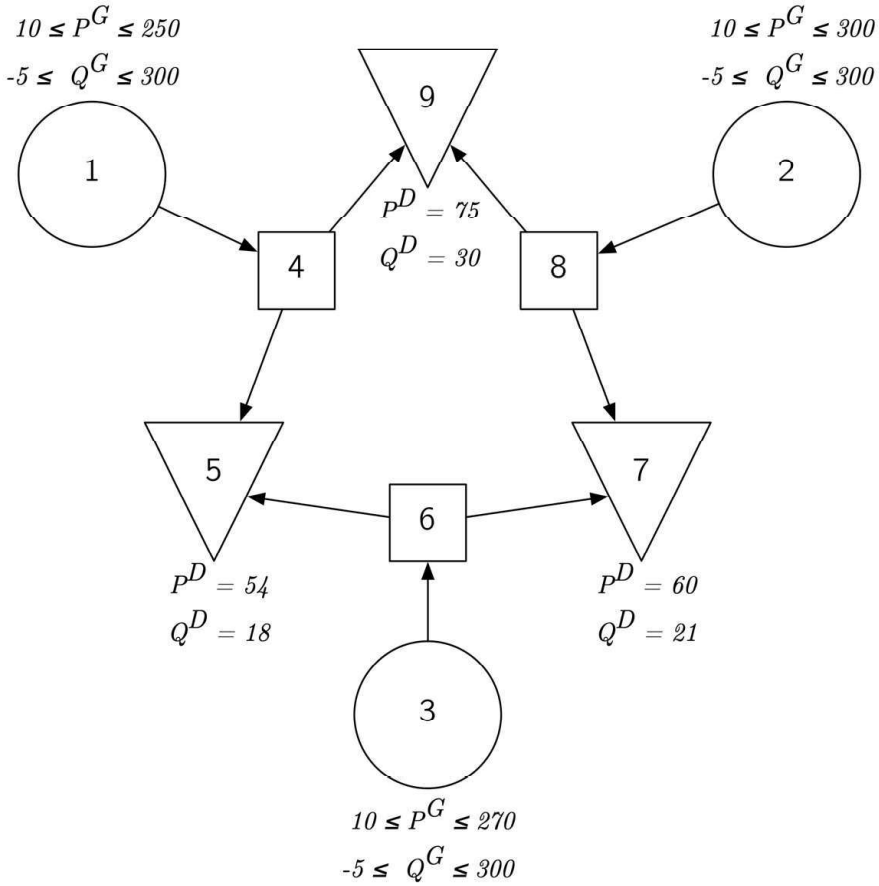
```
using JuMP
import LinearAlgebra
import SCS

function random_state(d)
    x = randn(ComplexF64, (d, d))
    y = x * x'
    return LinearAlgebra.Hermitian(y / LinearAlgebra.tr(y))
end

N, d = 2, 2

ρ = [random_state(d) for i in 1:N]

model = Model(SCS.Optimizer)
E = [@variable(model, [1:d, 1:d] in HermitianPSDCone()) for i in 1:N]
@constraint(model, sum(E) == LinearAlgebra.I)
@objective(
    model,
    Max,
    sum(real(LinearAlgebra.tr(ρ[i] * E[i])) for i in 1:N) / N,
)
optimize!(model)
```




https://commons.wikimedia.org/wiki/File:Unambiguous_quantum_state_discrimination_-_example.svg

# Optimal power flow

```
model = Model(Ipopt.Optimizer)

@variable(
    model,
    S_G[i in 1:N] in ComplexPlane(),
    lower_bound = P_Gen_lb[i] + Q_Gen_lb[i] * im,
    upper_bound = P_Gen_ub[i] + Q_Gen_ub[i] * im,
)

@variable(model,
  V[1:N] in ComplexPlane(), start = 1.0 + 0.0im)

@constraint(model, [i in 1:N], 0.9^2 <= real(V[i])^2 + imag(V[i])^2 <= 1.1^2)

@constraint(model, S_G - S_Demand .== V .* conj(Y * V))

P_G = real(S_G)
@objective(
    model,
    Min,
    (0.11 * P_G[1]^2 + 5 * P_G[1] + 150) +
    (0.085 * P_G[2]^2 + 1.2 * P_G[2] + 600) +
    (0.1225 * P_G[3]^2 + P_G[3] + 335),
);
optimize!(model)
```



# Nested optimization problems

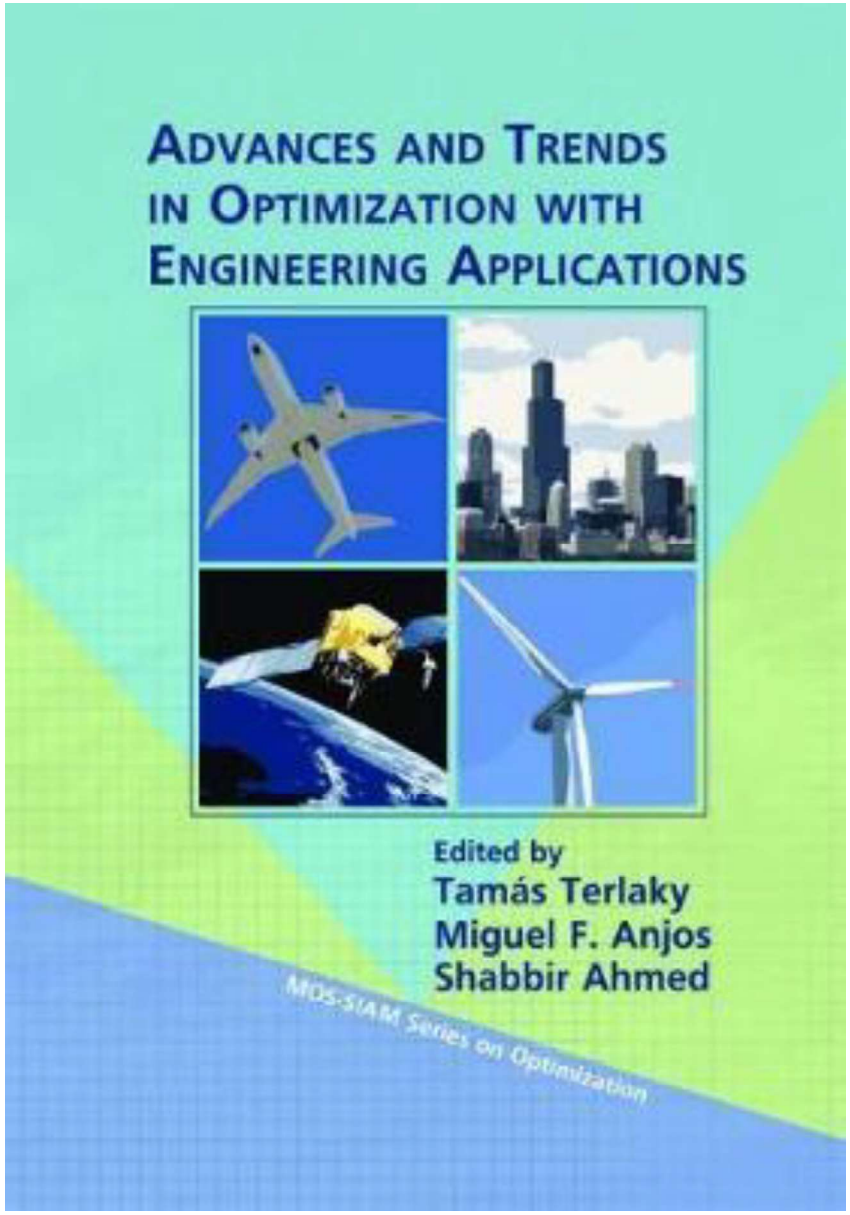
```
using JuMP
import Ipopt

function solve_lower_level(x...)
…
end

function V(x...)
    f, _ = solve_lower_level(x...)
    return f
end

function ∇V(g::AbstractVector, x...)
    _, y = solve_lower_level(x...)
    g[1] = 2 * x[1] * y[1] - y[1]^4
    g[2] = 2 * x[2] * y[2] - 2 * y[2]^4
    return
end


model = Model(Ipopt.Optimizer)

@variable(model, x[1:2] >= 0)

register(model, :V, 2, V, ∇V)

@NLobjective(model, Min, x[1]^2 + x[2]^2 + V(x[1], x[2]))
optimize!(model)
```

$$
\begin{aligned}
\min_{x,z} \quad & x_1^2 + x_2^2 + z \\
s.t. \quad & z = \max_{y} \quad x_1^2 y_1 + x_2^2 y_2 - x_1 y_1^4 - 2 x_2 y_2^4 \\
& \qquad s.t. \quad (y_1 - 10)^2 + (y_2 - 10)^2 \le 25 \\
& x \ge 0.
\end{aligned}
$$

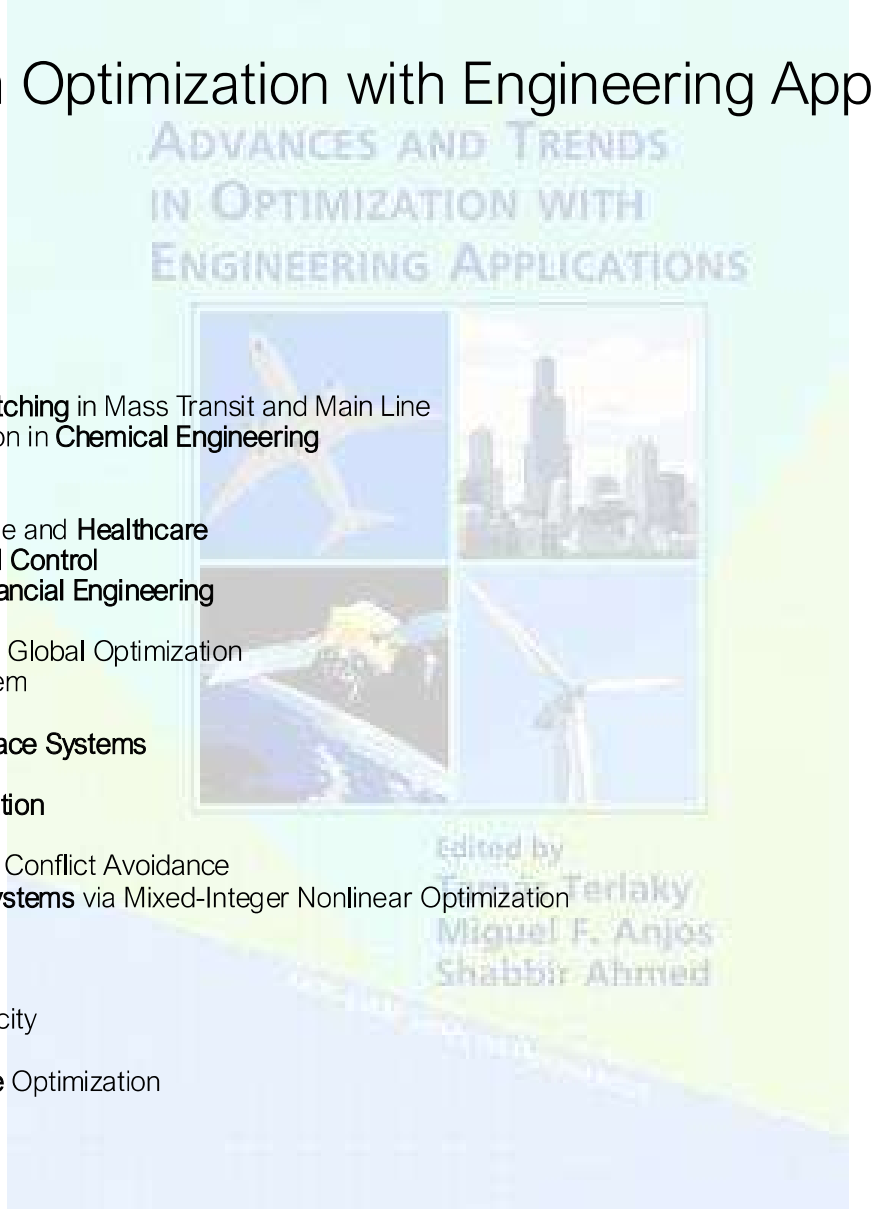
# Extensions

- MultiObjectiveAlgorithms.jl (MOA)
  - Algorithms for multi-objective optimisation.
- BilevelJuMP.jl
  - Model and solve bilevel optimisation problems.
- DiffOpt.jl
  - Differentiate through and then optimising over a set of structurally parameterised problems.
- ParametricOptInterface.jl
  - Efficiently modify key data and constants in an optimisation problem.
- PolyJuMP.jl
  - Formulate and solve problems involving polynomials as decision variables.
- SDDP.jl
  - A solver for multistage stochastic optimization problem.
- PiecewiseLinearOpt.jl
  - Model optimisation problems containing piecewise linear functions.

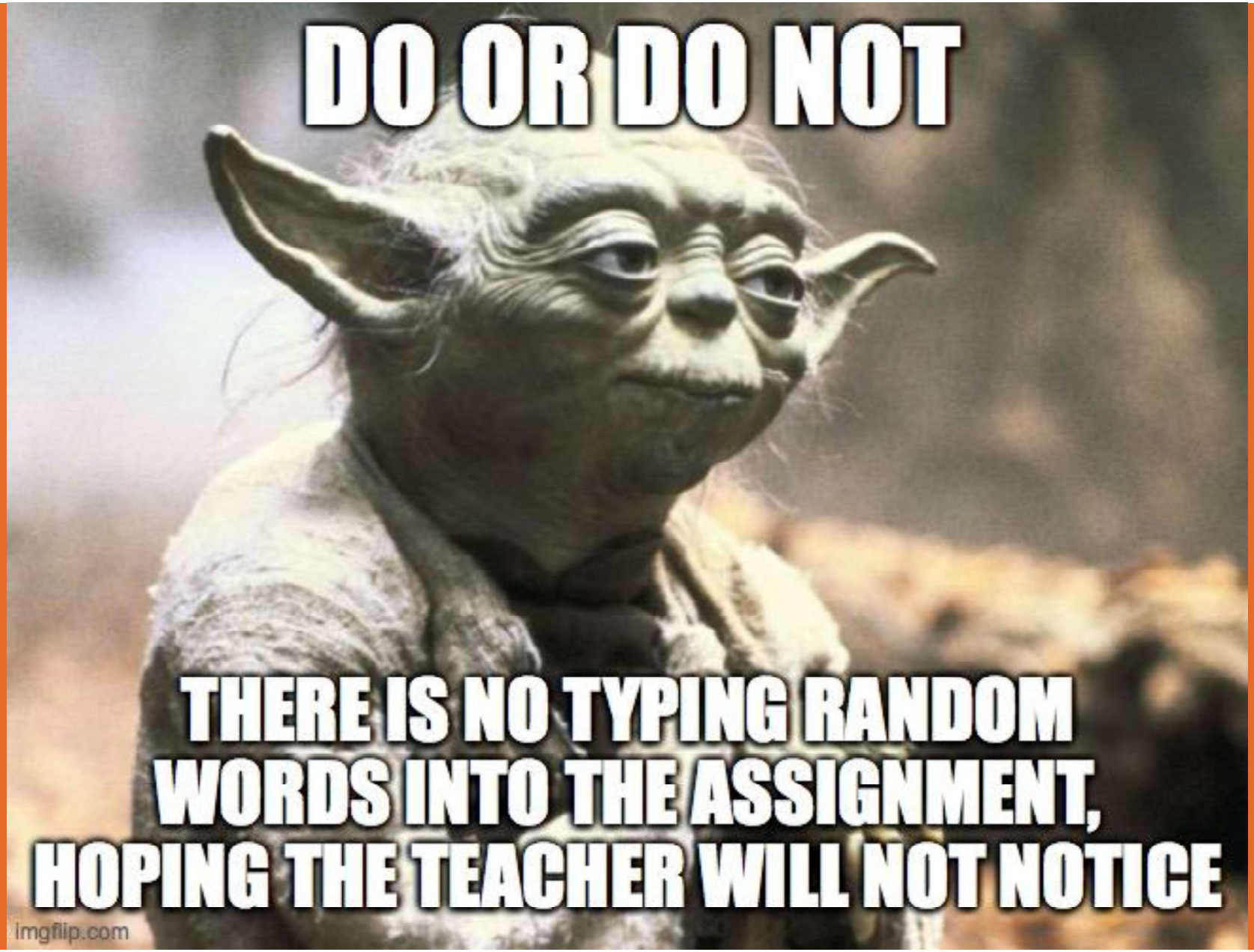
2017


ADVANCES AND TRENDS
IN OPTIMIZATION WITH
ENGINEERING APPLICATIONS
Edited by
Tamás Terlaky
Miguel F. Anjos
Shabbir Ahmed
MOS-SIAM Series on Optimization

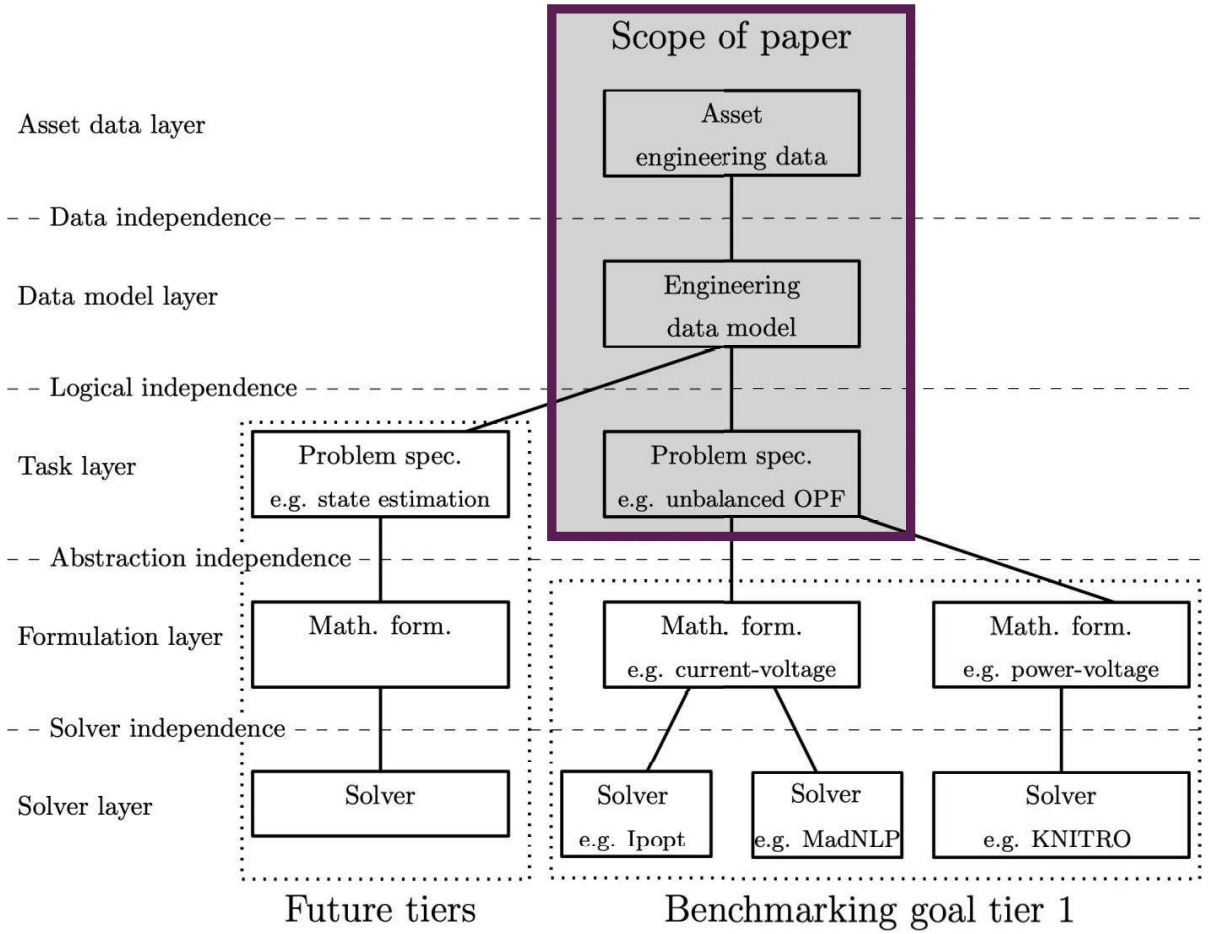
# Advances and Trends in Optimization with Engineering Applications




- **Truss Topology** Design by Linear Optimization
- Optimization in **Electrical Engineering**
  - Filter design
  - Pattern Classification
  - Information and Coding Theory
- Optimization in **Chemical Engineering**
  - Production Planning
  - Model Predictive Control
- Integer Optimization Techniques for **Train Dispatching** in Mass Transit and Main Line
- Applications of Mixed-Integer Linear Optimization in **Chemical Engineering**
  - LNG **Inventory Routing**
  - Chemical **Supply Network** Optimization
- Applications of Discrete Optimization in Medicine and **Healthcare**
- Conic Linear Optimization for Nonlinear **Optimal Control**
- Applications of Conic Linear Optimization in **Financial Engineering**
  - **Portfolio Optimization** Problems
- De Novo Design of **Protein-DNA Systems** Using Global Optimization
- Global Optimization: **Optimal Power Flow** Problem
- Optimization of **Distillation Systems**
- Multidisciplinary Design Optimization of **Aerospace Systems**
  - **Aerodynamic Shape** Optimization
  - Optimization of a **Satellite's Design and Operation**
- Nonlinear Optimization for **Building Automation**
- Optimization in **Air Traffic Management**: Aircraft Conflict Avoidance
- Short-Term Planning of **Cogeneration Energy Systems** via Mixed-Integer Nonlinear Optimization
- Robust Optimization in **Radiation Therapy**
  - Treatment Planning and Optimization
- Robust **Wind Farm Layout** Optimization
- On the **Marginal Value of Water** for Hydroelectricity
- Inventory and **Supply Chain** Optimization
- Methodologies and Software for **Derivative-Free** Optimization


DO OR DO NOT
THERE IS NO TYPING RANDOM WORDS INTO THE ASSIGNMENT, HOPING THE TEACHER WILL NOT NOTICE
imgflip.com

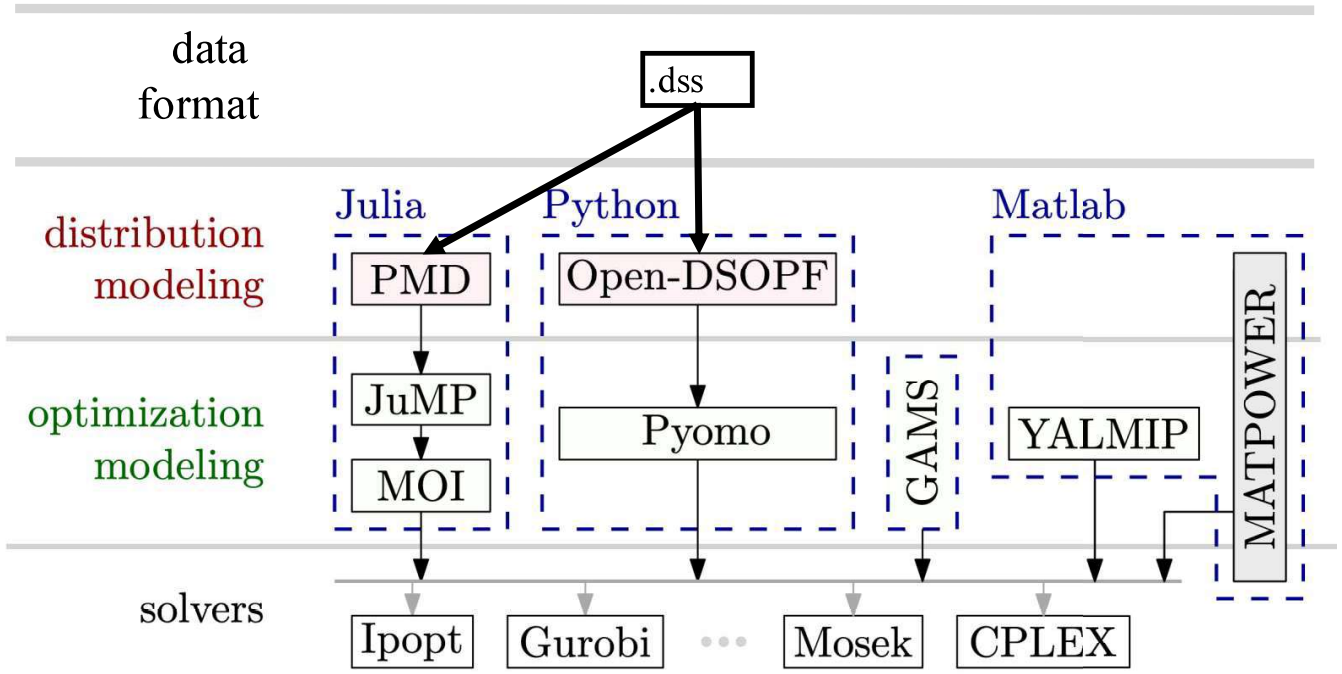
# Benchmarking unbalanced OPF




- Data *models* to represent electrical components in an unbalanced way
  - Modelling principles that avoid mistakes of the past
  - Easier to discuss properties of physical objects than math parameters
- Data *formats* that are easy to parse as a researcher

# Optimization-focused network data format

**.dss was a workaround**



# PowerModels OPF Problem Definition

- formulation of problem from academic paper

**variables:**

$$S_i^g \ \ \forall i \in N$$
$$V_i \ \ \forall i \in N$$

**minimize:**

$$\sum_{i \in N} f(S_i^g)$$

**subject to:**

$$(\boldsymbol{v_i^l})^2 \leq V_i V_i^* \leq (\boldsymbol{v_i^u})^2 \ \ \forall i \in N$$
$$\boldsymbol{S_i^{gl}} \leq S_i^g \leq \boldsymbol{S_i^{gu}} \ \ \forall i \in N$$
$$S_i^g - \boldsymbol{S_i^d} = \sum_{(i,j) \in E \cup E^R} S_{ij} \ \ \forall i \in N$$
$$S_{ij} = \boldsymbol{Y_{ij}^*} V_i V_i^* - \boldsymbol{Y_{ij}^*} V_i V_j^* \ \ (i,j) \in E \cup E^R$$
$$|S_{ij}|^2 \leq (\boldsymbol{s_{ij}^u})^2 \ \ \forall (i,j) \in E \cup E^R$$
$$-\boldsymbol{\theta_{ij}^\Delta} \leq \angle(V_i V_j^*) \leq \boldsymbol{\theta_{ij}^\Delta} \ \ \forall (i,j) \in E$$

# PowerModels OPF Problem Definition

```
function post_opf(pm::GenericPowerModel)
    variable_voltage(pm)
    variable_generation(pm)
    variable_line_flow(pm)

    objective_min_fuel_cost(pm)

    constraint_theta_ref(pm)
    constraint_voltage(pm)

    for (i,bus) in pm.ref[:bus]
        constraint_kcl_shunt(pm, bus)
    end

    for (i,branch) in pm.ref[:branch]
        constraint_ohms_yt_from(pm, branch)
        constraint_ohms_yt_to(pm, branch)

        constraint_phase_angle_difference(pm, branch)

        constraint_thermal_limit_from(pm, branch)
        constraint_thermal_limit_to(pm, branch)
    end
end
```

Implicit

d

**variables:**

$S_i^g \;\; \forall i \in N$

$V_i \;\; \forall i \in N$

**minimize:**

$$\sum_{i \in N} f(S_i^g)$$

**subject to:**

$$(\boldsymbol{v_i^l})^2 \leq V_i V_i^* \leq (\boldsymbol{v_i^u})^2 \;\; \forall i \in N$$

$$\boldsymbol{S_i^{gl}} \leq S_i^g \leq \boldsymbol{S_i^{gu}} \;\; \forall i \in N$$

$$S_i^g - \boldsymbol{S_i^d} = \sum_{(i,j) \in E \cup E^R} S_{ij} \;\; \forall i \in N$$

$$S_{ij} = \boldsymbol{Y_{ij}^*} V_i V_i^* - \boldsymbol{Y_{ij}^*} V_i V_j^* \;\; (i,j) \in E \cup E^R$$

$$|S_{ij}|^2 \leq (\boldsymbol{s_{ij}^u})^2 \;\; \forall (i,j) \in E \cup E^R$$

$$-\boldsymbol{\theta_{ij}^\Delta} \leq \angle(V_i V_j^*) \leq \boldsymbol{\theta_{ij}^\Delta} \;\; \forall (i,j) \in E$$