Day 4 - Optimization in real life

Richard Oberdieck

Ørsted A/S

June 9, 2019

Outline

Outline

What does it take to build something useful?

There are mainly four main components to this:

Solution design: Where does the optimization problem fit?

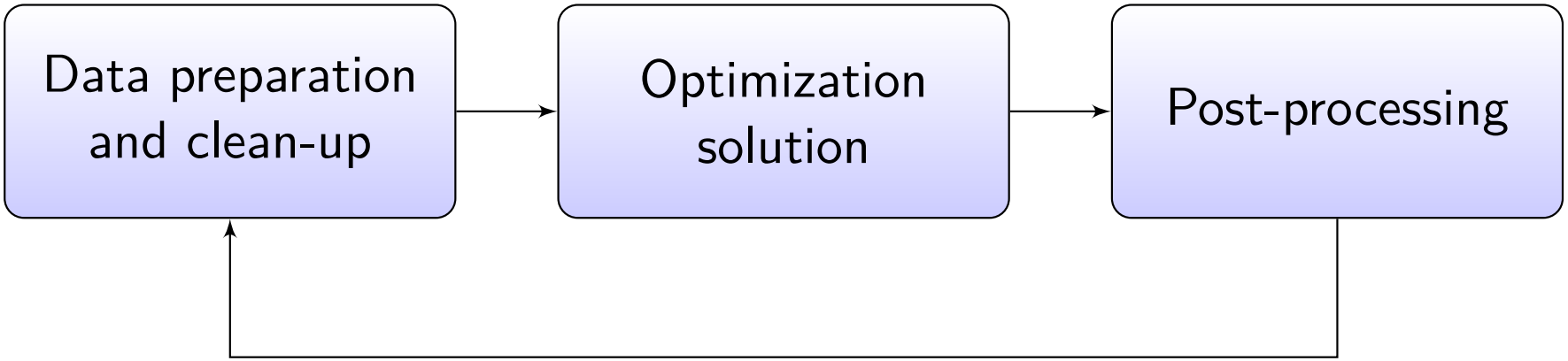
Process design: How do you arrive at the model/optimization strategy you are going to use?

Software development: How am I going to make the computer do what I want?

The human factor: How do people impact the tool?

Outline

Where does optimization fit?



- Typically, an optimization solution lives in a broader sea of applications.
- The arrows in the picture above may represent APIs (Application Programming Interfaces), i.e. ways to communicate with other programs and programming languages.
- Each box in the picture above typically has a front-end/user interface and a back-end, where the actual work happens.
- As a developer/data scientist, you will have to work with all three of these boxes regularly.

4 rules for data preparation and clean-up

Rule 1: Test

When importing data, it is insanely easy to make mistakes (e.g. units and rounding errors). Always **test** that your data import is correct.

Rule 2: Document data clean-up

Your actual data is most likely going to be messy, i.e. it needs clean-up. You **will** take some random decisions there (cut-off values, clustering etc.). Make sure to document that as well as you can:

- Collect all your "magic numbers" in one place.
- Comment your code
- Ideally, write a separate document to keep all of this clean.

4 rules for data preparation and clean-up

Rule 3: Know where your data lives

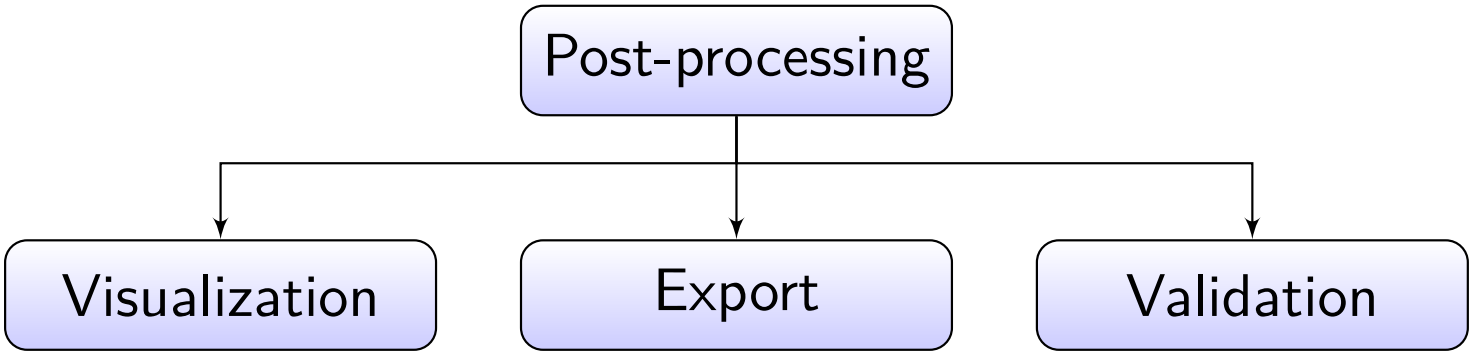
Are you taking some stock market prices from somewhere? Or the pressure measurement from somewhere else?

- Do you have access?
- What happens when the data is unavailable?
- Who is responsible?

Rule 4: Keep data preparation separate

Never mix data import and data preparation with other parts of your code.

Post-processing 101



Visualization: One figure says more than 1000 data points. Tip: draw something on paper. However, this is a bottomless pit, i.e. know when it is good enough.

Export: What format do you need? If possible, use a consistent (and automated) naming strategy! Also, databases can make your life a whole lot easier!

Validation: If the validation happens outside of your program (e.g. high-fidelity modelling system), make sure to respect the data preparation and clean-up rules.

3 tips for visualization

Tip 1: It's worth it

Even if it takes long, writing a good data visualization is **always** worth it. Everybody will ask for the visualization, nobody will ask for the data points.

Tip 2: Use other people

Try to use the community of your programming language as much as possible: data/information visualization is a very common thing to do.

Tip 3: Make it easy

Make visualization easy to use, e.g. something like:

```
ShowResults(optimizationResult)
```

and package all the "nasty" details away, since visualization code tends to be very long.

APIs - your best friends in programming

Application Programming Interface (API)

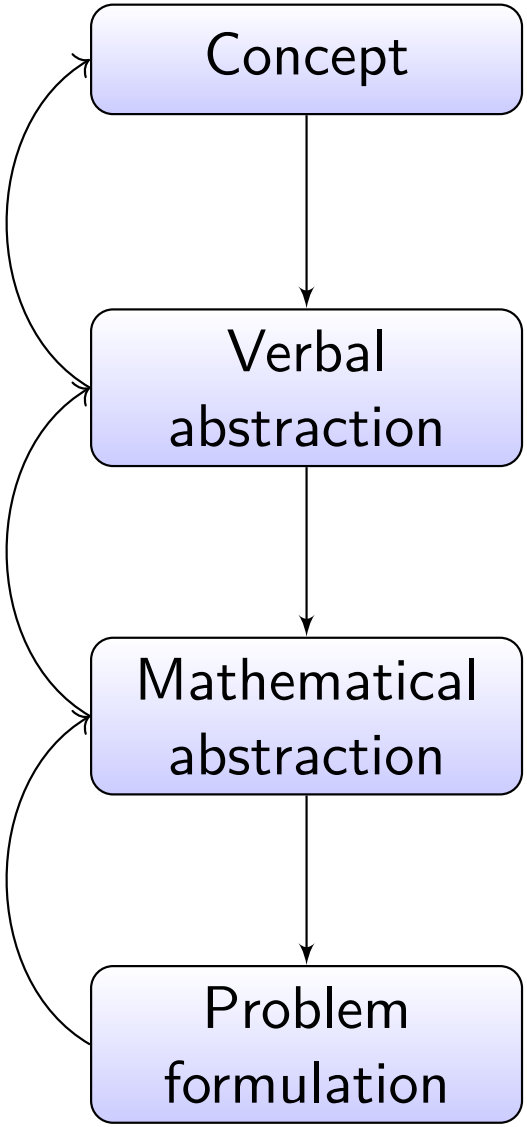
An API allows **application** A to use a line of **programming** to **interface** with application B.

- There APIs for all major programming languages and tools (even MATLAB).
- They allow you to use the strengths of language A and language B together!
- Also: if you want to migrate (say from MATLAB to Python), then you can use a "MATLAB for Python" API as an intermediary step.

```
import matlab.engine
eng = matlab.engine.start_matlab()
t = eng.gcd(100.0,80.0,nargout=3)
print(t)
(20.0, 1.0, -1.0)
```

Outline

The process of creating a model



"Let's optimize our batch schedule!"

minimize cost

subject to Physical constraints

...

$x_{i,j}$: flow variables; ...

$$\text{minimize}_{x} \quad \sum_{i,j} c_{i,j} x_{i,j}$$

$$\text{subject to} \quad \sum_{i,j} x_{i,j} - \sum_{j,k} x_{j,k} = 0$$

...

- This may take many iterations.
- Be ready to discard ideas if something better comes.
- Validate each step separately with the relevant people.
- Go to the whiteboard before you go to the computer.

Modelling FAQs

When is my model "good enough"?

A model is always an approximation, and you have to decide when you are happy with it. Oftentimes, performance is the key bottleneck in making a more accurate model.

Are modelling and optimization separate things?

Yes, to a good extent, since you need a model before you can optimize. However, you should know what you can optimize (LP/NLP/et.) and not be afraid to go back to the modelling if the optimization behaviour is not satisfactory.

How can I become a better modeller for optimization problems?

Learning by doing, essentially. But also reading other peoples work and approaches really helps to sharpen your skills.

Outline

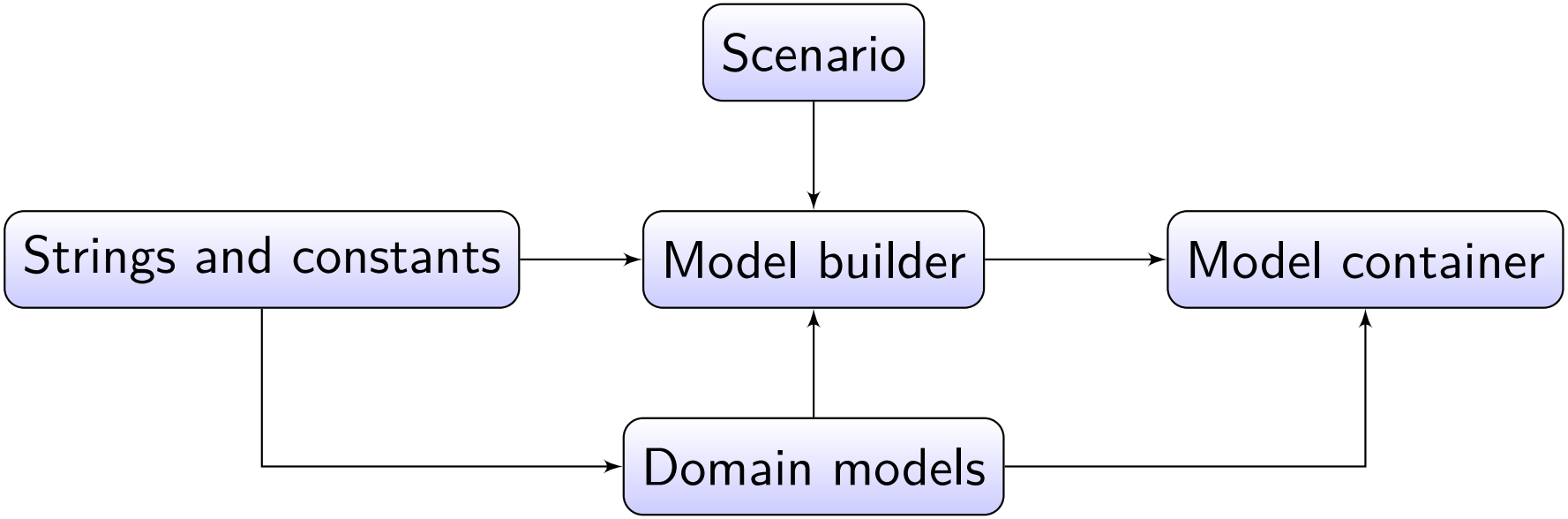
Why software development?

How much coding do I need?

"Unless you magically find yourself endowed with limitless programming support, sooner or later (most likely sooner) you are likely to need to do some coding." (Paul Rubin)

- Producing good code is hard!
- That said: by rigorously following some simple rules you can make sure that your code stays maintainable and usable.
- However for larger software projects it may be useful to work together with a software developer.

5 rules of optimization code



1. The code is structured as depicted above.
2. Every index set is its own object in `Domain models`.
3. Variables are written as dictionaries with indices as keys and variable objects as values, defined in `Model container`.
4. Each constraint and callback is a separate functions in `Model container`.
5. All the individual components as well as the entire optimization problem have to be unit tested.

Tools and techniques for optimization code

Model document: A document that details how the model looks like. I like to use LaTeX, since it is the cleanest and most flexible.

Naming: Name your variables/functions such that you don't need a comment to understand what they mean (e.g. `FlowBalanceForTeams`).

Version control: A code repository that you `commit` your incremental changes to the code to. I like to use git and github.com

Code review: If you know that somebody will look at your code, you will write better code. Also, people's comments are most times gold.

Unit tests: Every single aspect of your code should have unit tests. The most important one is "input $\rightarrow$ output", i.e. that your optimization problem returns what you expect.

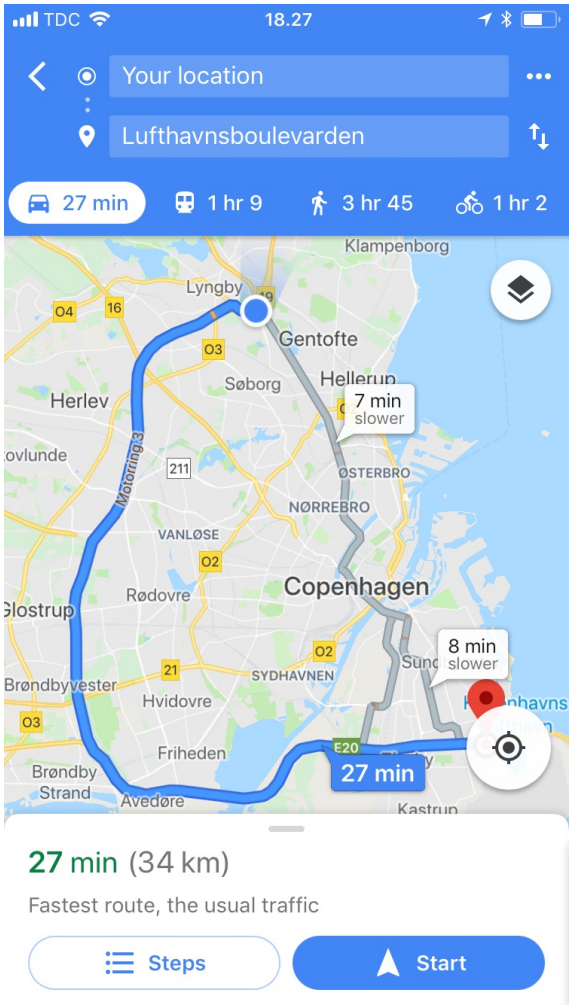
Continuous integration: A setup such that every time you commit your code, all the unit tests are run. It shows you as soon as things are broken.

From scripting to a software tool

- Oftentimes, we start by scripting ("let's just check that"), which then grows more and more if it is a good idea.
- Very rarely do we go back and look at the original code. We simply build around it.
- That can lead to grave problems, so try to do the following:
  1. Do not rush: take your time to write a clean script. The 5 minutes you save by rushing will cost you 5 hours later.
  2. Already when scripting, devise good naming rules. Names such as `test` and `temp` are **never** good.
  3. Whenever you see the opportunity, "package" something away, even if it is just one line. For example, if you often calculate the Euclidean distance, write `Euclidean(a,b)` as a function.
  4. When you decide that this will be something to work on more, start a new project and include the leanings you got from scripting.

Outline

What is the human factor?



- Some applications are truly autonomous (e.g. landing of SpaceX rockets).
- However, in many cases a tool does not make a decision without a human checking (Google Maps).
- People want **decision support tools**, rather than autonomous robots (in most cases). It is key to keep this in mind when designing a tool.
- A tool which provides suggestions rather than makes decisions is typically much more accepted and used.

3 rules for considering humans

Rule 1: Keep them up-to-date

Show them very early drafts, discuss the visualization with them and **listen** to what they have to say!

Rule 2: Make them work

Ask them to provide you with what they want to see. They may just say "yes" to something because they can't be bothered to think about it.

Rule 3: Not all users are alike

Every user (group) is different, so it is not enough to talk to one person and tick the box. Really go out there and get the diverse and real feedback from everybody.

Looking at a real application

- We will now organize one of our notebooks together into usable code.
- Open up `BatchScheduling.ipynb`, and let's take it from there.

Outline

The golden rule of performance tuning

Always set yourself a **quantifiable** performance goal.

- Optimization is a computationally very heavy operation.
- The most frequently asked question for optimization problems, "how fast can you solve it?", is the wrong question. The right question is: "how fast do you need it to be solved?".
- We have different leavers we can pull, depending on the specific problem at hand (ordered from easy to difficult):
  1. Get better hardware
  2. Parameter tuning
  3. Modify the model
  4. Develop a good heuristic
  5. Change the optimizer

Get better hardware

- This is the easiest thing to do: just go to the cloud (Azure, AWS etc.).
- **However**: Some solvers have this in-built (e.g. Gurobi) while others are more difficult to get working (e.g. Xpress).
- Alternatively, you can add more (1) RAM or (2) CPU power. RAM is a better first option because it is cheap and often the bottleneck for the size of the tree in branch-and-bound.
- If you hit up your computer science colleagues/IT department, you should be able to get this up and running quickly.

Parameter tuning

- Solvers have up to 200 parameters, that define everything from random seed, tolerances and branching strategies.
- It can have a **huge** impact on the performance to find the right parameters.
- Most solvers have automated routines that try out different settings and find a good choice.
- Some solvers (e.g. Xpress) even allow you to tune on a *family* of problems, i.e. not just one instance. This will make your choices more robust.

Modify the model

- Modifying your model to improve speed mostly comes in three flavors:
  - Making your model simpler (remove complexity)
  - Create a "better" model, i.e. fewer equations/variables for the same model
  - Create a model representation that the solver is better equipped to handle
- **However**, as we will see, you will now know whether these things will in face improve your performance. They might even decrease it.

Develop a good heuristic

Definition

"A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution" (Wikipedia)

Examples on how to tackle the development of a heuristic:

Imitate the human: Encode the decision processes of a human, e.g. "I would always pick the closest node".

Check similar problems: Check whether your problem structure is similar to a "famous" problem for which good approximate algorithms exist.

Break down into subproblems: Construct a feasible solution by decomposing the main problem and solving the subproblems.

Change the optimizer

- If you are an academic and have used a general modelling framework (pyomo, PuLP etc.), then this is actually easy (and you should do it almost immediately).
- Otherwise, this will take some time, because you have to re-write your entire `Model container`.
- I would always export the problem as a `.mps` file (standard file format for optimization problems) to check whether the new solver will be better, before you move over.

Let's do some performance tuning

- Let's open `OptiSeating.ipynb` and have a look
- Now try your hands on one of our problems and see how much speedup you get from tuning!

Outline

Future trends in core optimization solvers

- Algorithms will continue to improve, primarily based on new, clever heuristics for specific problems.
- More modelling capabilities will be exposed to the user (e.g. multi-objective optimization).
- Strong focus on cloud computing and facilitating this using e.g. docker containers.

Future trends in modelling environments

- Domain-specific langauges (GAMS, AMPL etc.) will loose ground, just like MATLAB.
- More and more universities and practitioners will switch to Python and Julia, followed by C# and C++.
- Due to the increase in modelling activities (big data, machine learning etc.), the need for "talking" the same language and having suitable APIs will increase.

Future trends in optimization techniques

- In industry, most classical OR problems are now solved using optimization (scheduling, vehicle routing etc.). Some industries (e.g. aviation) are better however than others (e.g. pharmaceuticals). This will become fully handled by optimization in the future.
- Optimization is also being used as "smart" decision support tools (e.g. "where should I make a cut in the wood to minimize waste?"), however this is less widespread. Optimization is competing e.g. with machine learning to who can solve the problem in the best way.
- The biggest trend is though incorporating big data and machine learning into optimization problems. Currently, in most cases machine learning models are used to provide input to optimization problems, however other interfaces are investigated as well (e.g. ML as heuristics within the solver).