

Online Training – Advanced session

February 2022

**Design of experiments, space
exploration, and numerical
optimization using DAKOTA and
OpenFOAM®**

Copyright and disclaimer

This offering is not approved or endorsed by OpenCFD Limited, the producer of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trademarks.

© 2014-2022 Wolf Dynamics.

All rights reserved. Unauthorized use, distribution or duplication is prohibited.

Contains proprietary and confidential information of Wolf Dynamics.

Wolf Dynamics makes no warranty, express or implied, about the completeness, accuracy, reliability, suitability, or usefulness of the information disclosed in this training material. This training material is intended to provide general information only. Any reliance the final user place on this training material is therefore strictly at his/her own risk. Under no circumstances and under no legal theory shall Wolf Dynamics be liable for any loss, damage or injury, arising directly or indirectly from the use or misuse of the information contained in this training material.

All trademarks are property of their owners.

Revision 1-2022

JG






Before we begin

On the training material

- **This training is based on OpenFOAM 9 and DAKOTA 6.14 (and newer).**
- In the USB key/downloaded files you will find all the training material (tutorials, slides, and lectures notes).
- You can extract the training material wherever you want. From now on, this directory will become:
 - **\$TM**
(abbreviation of **T**rainin**M**aterial)
- To uncompress the tutorials go to the directory where you copied the training material (**\$TM**) and then type in the terminal,
 - `$> tar -zxvf file_name.tar.gz`
- In the case directory of every single tutorial, you will find a few scripts with the extension `.sh`, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
 - `$> sh run_all.sh`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM and DAKOTA commands.
- If you are already comfortable with OpenFOAM and DAKOTA, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

Conventions used

The following typographical conventions are used in this training material

- Text in `Courier new` font indicates Linux commands that should be typed literally by the user in the terminal.
- Text in **`Courier new bold`** font indicates directories.
- Text in *`Courier new italic`* font indicates human readable files or ascii files.
- Text in **Arial bold font** indicates program elements such as variables, function names, classes, statements and so on. It also indicates environment variables, and keywords. They also highlight important information.
- Text in [Arial underline in blue](#) font indicates URLs and email addresses.
- This icon  indicates a warning or a caution.
- This icon  indicates a tip, suggestion, or a general note.
- This icon  indicates a folder or directory.
- This icon  indicates a human readable file (ascii file).
- This icon  indicates that the figure is an animation (animated gif).
- These characters `$>` indicate that a Linux command should be typed literally by the user in the terminal.

Conventions used

The following typographical conventions are used in this training material

- Large code listing, ascii files listing, and screen outputs can be written in a square box, as follows:

```
1  #include <iostream>
2  using namespace std;
3
4  // main() is where program execution begins. It is the main function.
5  // Every program in c++ must have this main function declared
6
7  int main ()
8  {
9      cout << "Hello world";           //prints Hello world
10     return 0;                        //returns nothing
11 }
```

- To improve readability, the text might be colored.
- The font can be `Courier new` or **Arial bold**.
- And when required, the line number will be shown.

Roadmap

- 1. Introduction to optimization methods**
- 2. Choosing an optimization method**
- 3. Optimization loop – The big picture**
- 4. DAKOTA overview**
- 5. Working with DAKOTA: Rosenbrock function**
- 6. Working with DAKOTA: Branin function**
- 7. Working with DAKOTA: Multi-objective optimization**
- 8. Coupling DAKOTA and OpenFOAM: driven cavity case**
- 9. Additional code coupling tutorials**
- 10. Some kind of conclusion**

Roadmap

- 1. Introduction to optimization methods**
2. Choosing an optimization method
3. Optimization loop – The big picture
4. DAKOTA overview
5. Working with DAKOTA: Rosenbrock function
6. Working with DAKOTA: Branin function
7. Working with DAKOTA: Multi-objective optimization
8. Coupling DAKOTA and OpenFOAM: driven cavity case
9. Additional code coupling tutorials
10. Some kind of conclusion

Introduction to optimization methods

In CFD,

- **Who owns the mesh, owns the solution**

In numerical optimization with CFD,

- **Who owns the parameterization, owns the optimal solution**

Therefore, meshing and scripting are of paramount importance.

No need to say that it is also important to understand the theory behind numerical optimization.

Introduction to optimization methods

What is optimization?

- In plain English, optimization is the act of obtaining the best result under given circumstances.
- The ultimate goal is to minimize, maximize, equalize or zeroed an outcome, a process or a function, which we are going to call a quantity of interest or QOI.
- In order to optimize a QOI, we should be able to measure it quantitatively or qualitatively.
- Optimization, in its broadest sense, can be used to solve any real-life or engineering problem, such as,
 - Finance, health, construction, operations, manufacturing, transportation, engineering design, sales, public services, mail, communication networks, energy distribution, delivery services, CFD and so on.

Introduction to optimization methods

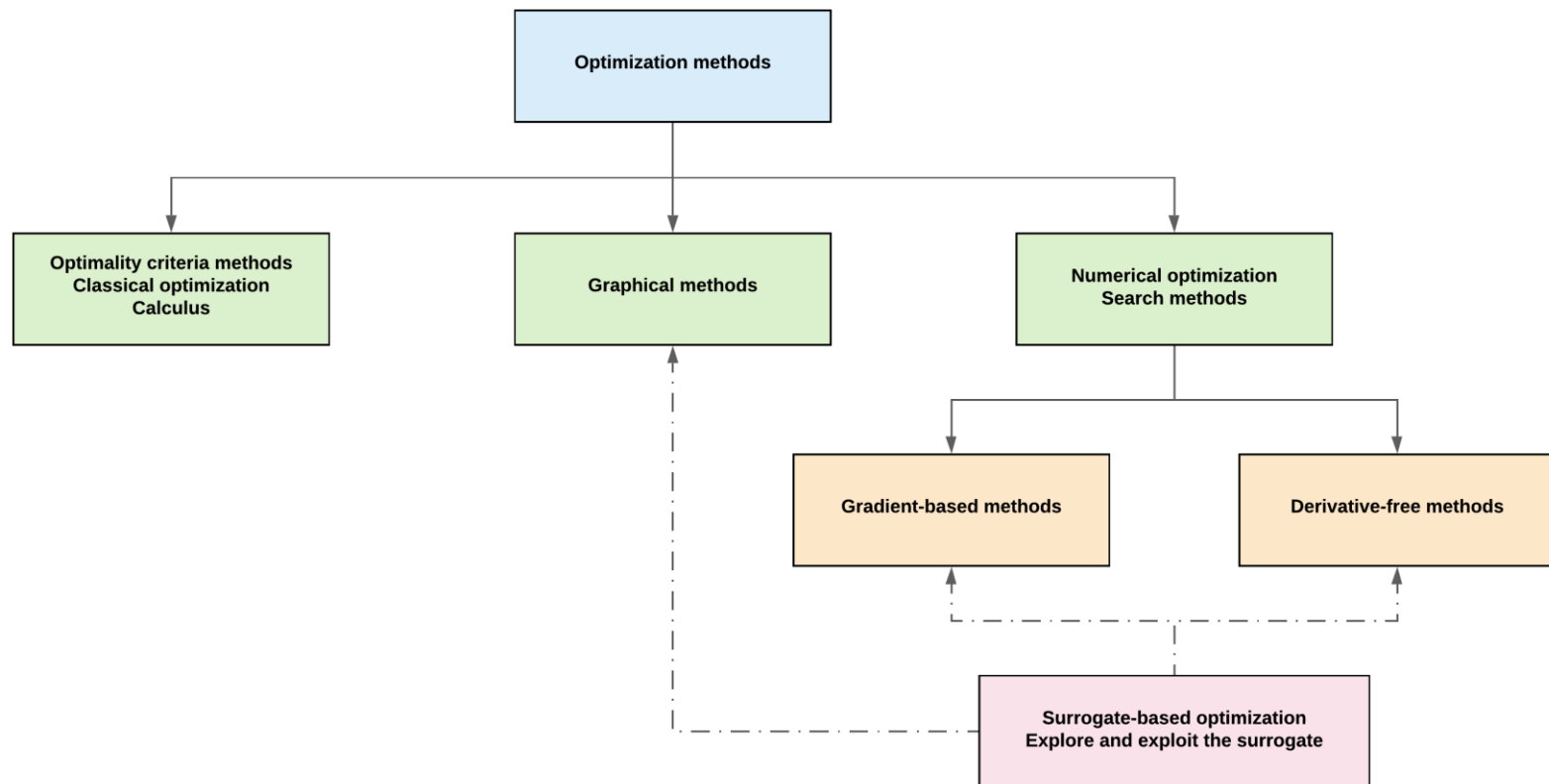
What is optimization?

- If you can measure the QOI, you can optimize it.
- And it does not matter how you measure the QOI.
- To find the optimal solution there are many methods available, in this training we re going to explore a few of them.
- And depending on your problem, finding that optimal solution might not be an easy task.
- There are also many tools to conduct optimization, we are going to use one them, namely, DAKOTA.

Introduction to optimization methods

Optimization and space exploration methods

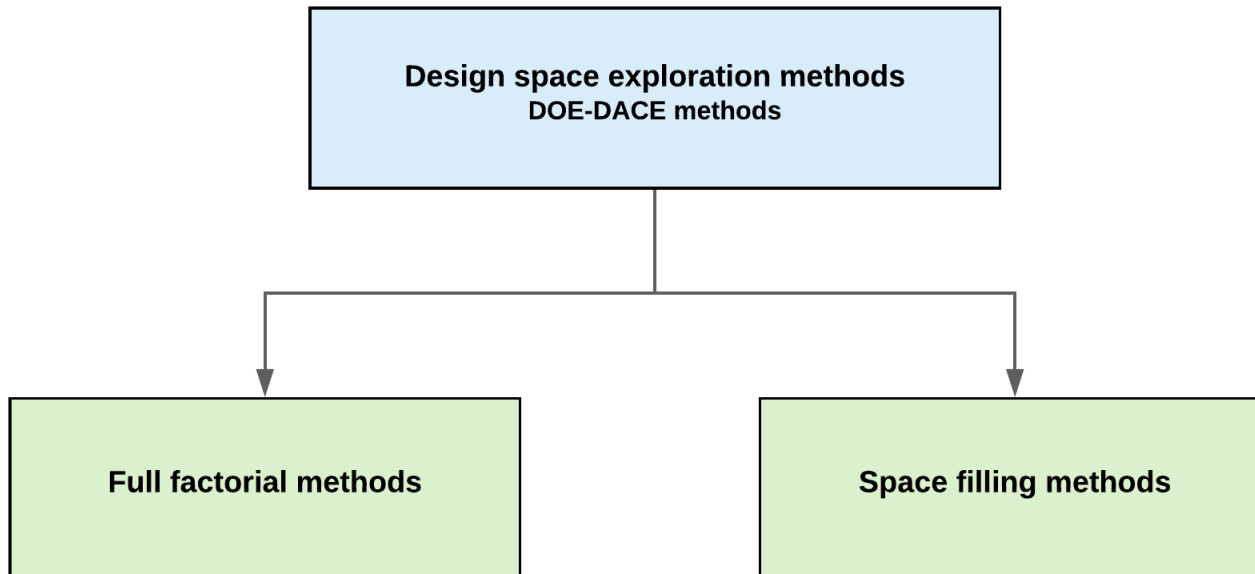
- In this training we are going to focus our attention of numerical optimization and search methods.
- In CFD, optimality criteria methods are not used because we do not know the behavior of QOI a-priori, that is, we do not have an analytical function of the QOI



Introduction to optimization methods

Optimization and space exploration methods

- Design space exploration methods are used to explore and characterize the design space.
- These methods, do not converge to the optimal solution, they are only used to gather information.



Introduction to optimization methods

Applications

- The following applications from different disciplines indicate the wide scope of the subject:
 - Design of aircraft and aerospace structures for minimum weight.
 - Reduce of fuel consumption in transportation.
 - Design of civil engineering structures for minimum cost.
 - Optimum design of mechanical components.
 - Optimum design of electrical networks.
 - Shortest route taken by a salesperson visiting various cities during one tour.
 - Optimal production planning, controlling, and scheduling.
 - Selection of a site for an industry.
 - Planning of maintenance and replacement of equipment to reduce operating costs.
 - Inventory control.
 - Allocation of resources or services among several activities to maximize the benefit.
 - Controlling the waiting and idle times and queuing in production lines to reduce the costs.
 - Planning the best strategy to obtain maximum profit in the presence of a competitor.
 - Analysis of statistical data and building empirical models from experimental results to obtain the most accurate representation of the social or physical phenomenon (big data).
- If you can measure it, you can optimize it.

Introduction to optimization methods

Mathematical definition of an optimization problem

- Mathematically speaking, an optimization problem can be formulated as follows,



The diagram illustrates the mathematical formulation of an optimization problem. It features a large curly brace containing the vector components x_1, x_2, \vdots, x_n . To the left of the brace, the text "Find $\mathbf{X} =$ " is present, with a red arrow pointing from a box labeled "Design vector" to the \mathbf{X} . To the right of the brace is the word "which". Further right, the words "minimizes", "maximizes", "equalizes", and "zeroes" are listed vertically in blue. To the right of these words is the expression $f_j(\mathbf{X}), \quad j = 1, 2, \dots, q$. A red arrow points from a box labeled "Quantity of interest" to the $f_j(\mathbf{X})$ term.

$$\text{Find } \mathbf{X} = \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} \text{ which } \begin{array}{l} \text{minimizes} \\ \text{maximizes} \\ \text{equalizes} \\ \text{zeroes} \end{array} f_j(\mathbf{X}), \quad j = 1, 2, \dots, q$$

Subject to the following constraints (linear and non-linear)

$$g_j(\mathbf{X}) \leq 0, \quad j = 1, 2, \dots, m$$

$$l_j(\mathbf{X}) = 0, \quad j = 1, 2, \dots, p$$

where \mathbf{X} is an n -dimensional vector called the design vector, $f_j(\mathbf{X})$ is the objective function or QOI, and $g_j(\mathbf{X})$ and $l_j(\mathbf{X})$ are known as inequality and equality constraints, respectively.

Introduction to optimization methods

The curse of dimensionality

- The biggest hurdle to overcome in numerical optimization is the design vector.

This guy is responsible for the
curse of dimensionality



$$\mathbf{X} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

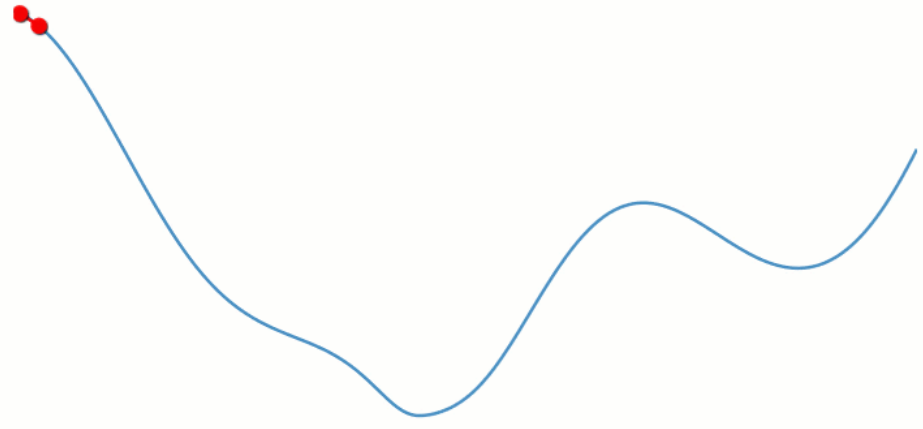
The curse of dimensionality...

“ The higher the number of design variables in a modelling problem, the more objective function measuring locations we need if we are to build a reasonably accurate predictor. ”

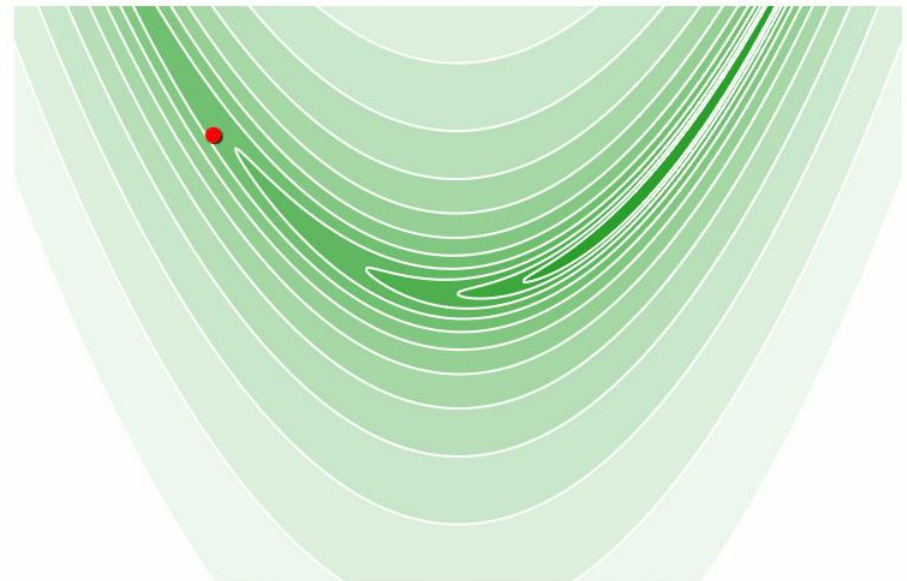
Introduction to optimization methods



- During this training session we will introduce a few basic concepts using univariate and bivariate functions.
- The choice of using univariate and bivariate functions is to help visualize the various concepts.
- However, have in mind that in optimization the design space can be multivariate or n dimensional.
- In the slides to follow, the goal is to optimize a toy function or quantity of interest (QOI).
- This toy function or QOI can easily represent the output or trend of an actual application.



<http://www.wolfdynamics.com/training/opt/ani1.gif>

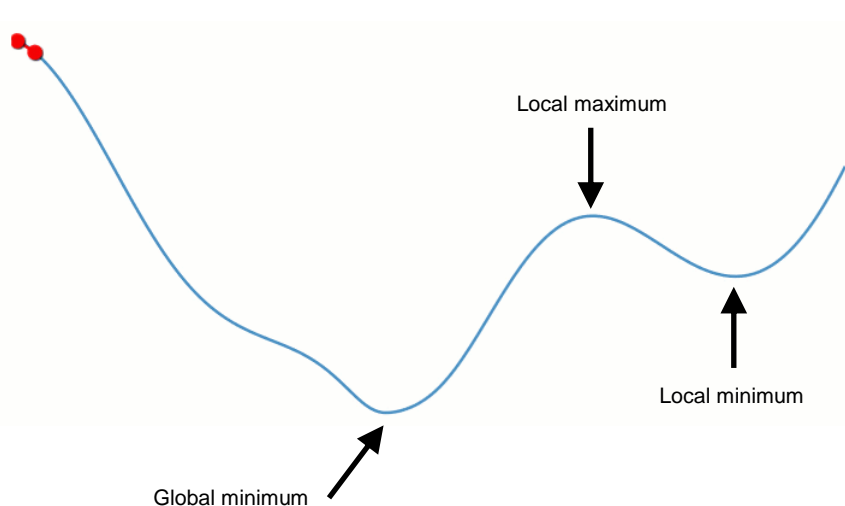


<http://www.wolfdynamics.com/training/opt/image6.gif>

Introduction to optimization methods

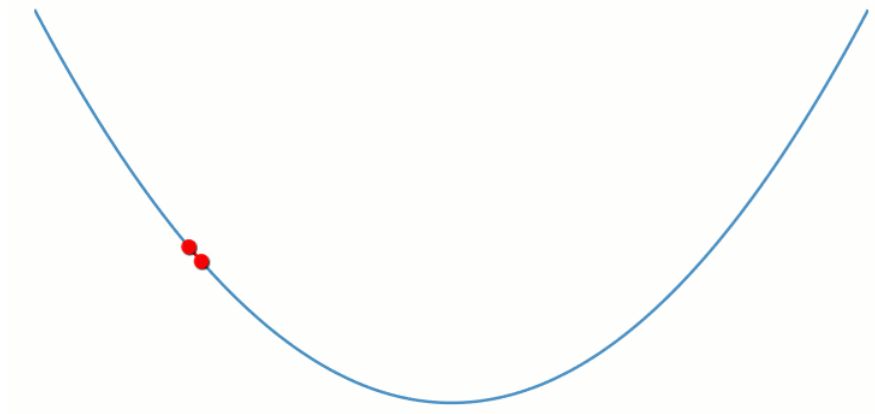
Unimodal vs. Multimodal function

- Let us use these figures to introduce a few concepts.
- A unimodal function contains one global minimum or maximum.
- A multimodal function contains many global and local minima/maxima.



Multimodal function – Many global and local minima/maxima

<http://www.wolfdynamics.com/training/opt/ani1.gif>



Unimodal function – One global minimum

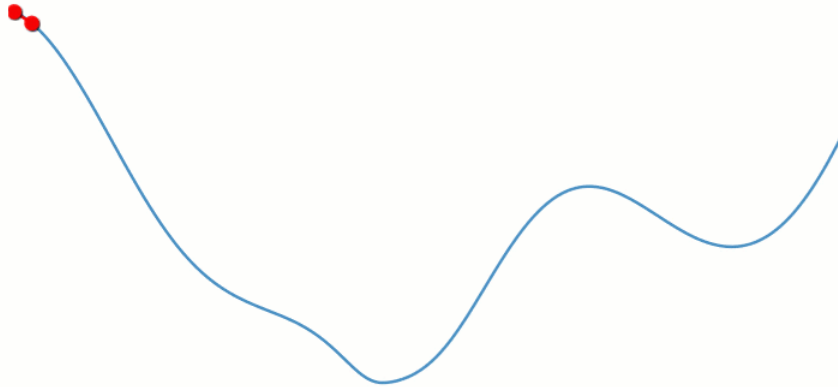
<http://www.wolfdynamics.com/training/opt/ani2.gif>



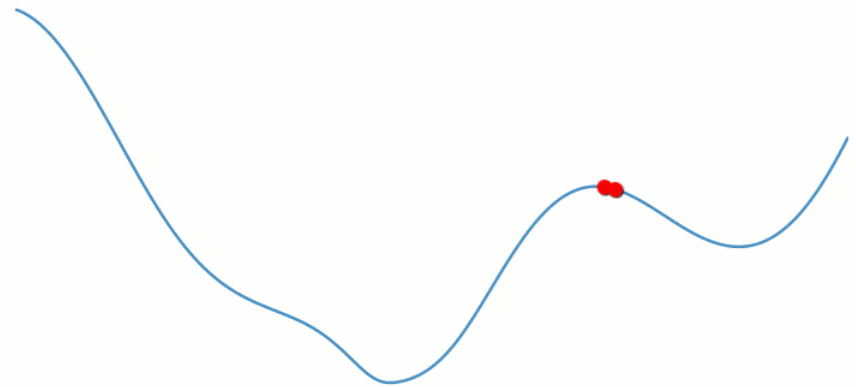
Introduction to optimization methods

Multimodal functions

- The goal is to optimize this function or QOI.
- Depending on our goal (local or global optimization), we should use a particular optimization method.
- Also, depending on the starting point we might arrive to a global or local minimum/maximum.



Multimodal function – Global minimum
<http://www.wolfdynamics.com/training/opt/ani1.gif>

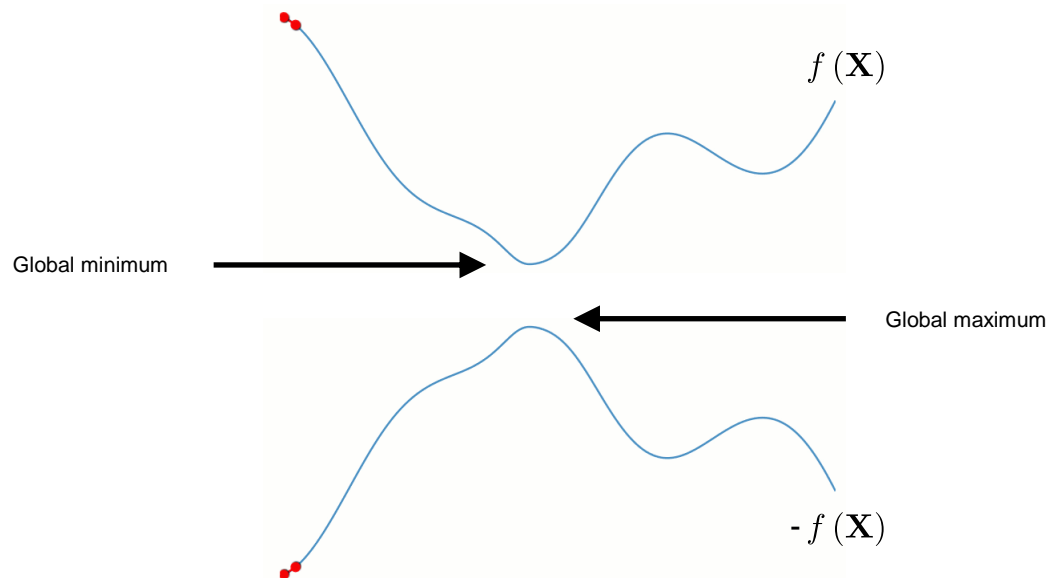


Multimodal function – Local minimum
<http://www.wolfdynamics.com/training/opt/ani3.gif>

Introduction to optimization methods

Symmetry of functions

- The goal is to optimize this function or quantity of interest.
- The function $f(\mathbf{X})$ is symmetrical. That is, we can minimize $f(\mathbf{X})$ or we can maximize $-f(\mathbf{X})$, the outcome will be the same.
- By the way, the function $f(\mathbf{X})$ represented on this figure, can easily represent the output or trend of an actual application.

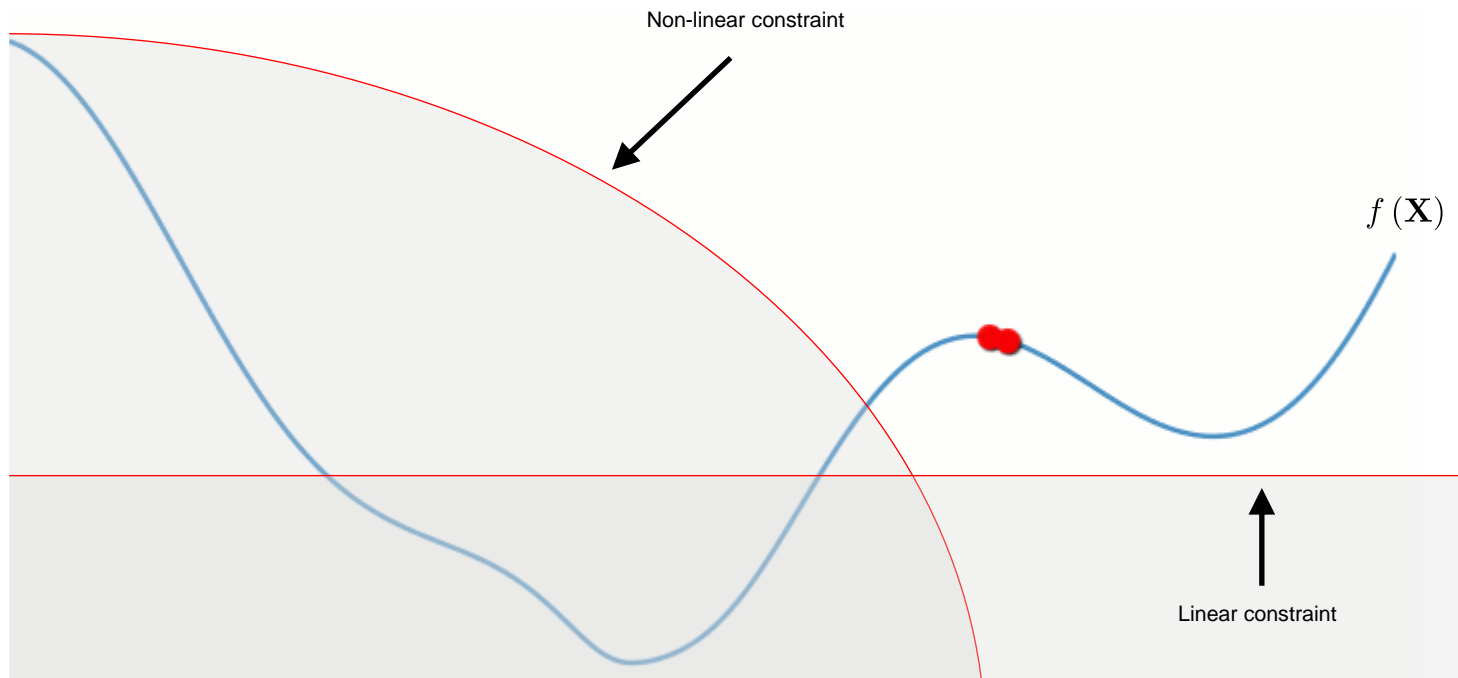


Minimum of $f(\mathbf{X})$ is the same as the maximum of $-f(\mathbf{X})$

Introduction to optimization methods

Constrained multimodal function

- The design space can be bounded or unbounded.
- We can optimize this function or QOI, subject to many linear and/or non-linear constraints (equalities and inequalities).

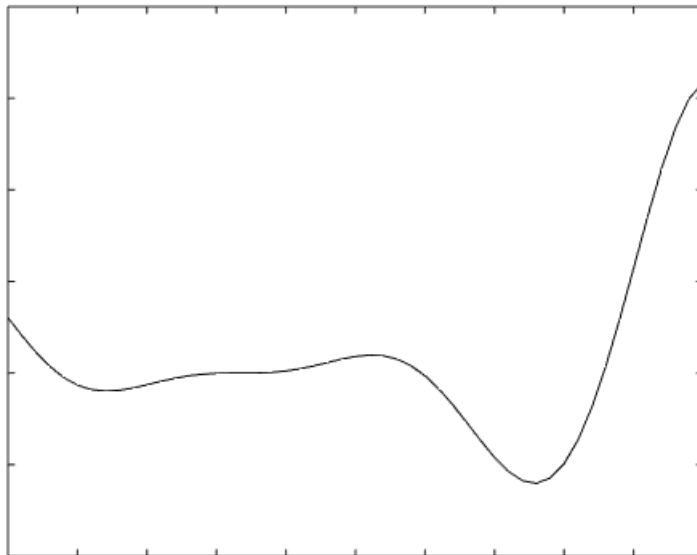


Constrained multimodal function – The shaded area represents the non-feasible region

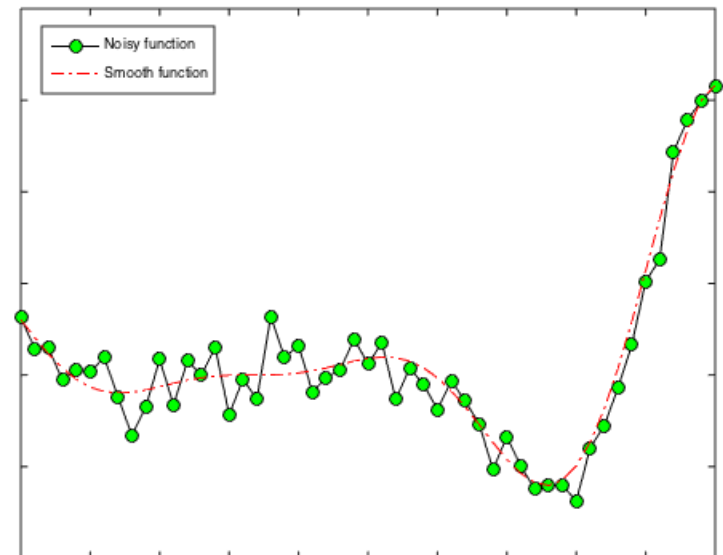
Introduction to optimization methods

Noisy functions

- In physical or computer experiments, the QOI often does not exhibit a smooth behavior.
- Very often the response is noisy, which makes optimization very challenging.
- Also, evaluating the objective function can be very expensive.
- And if you need the gradients and Hessians, things get even more expensive.



Smooth $f(\mathbf{X})$

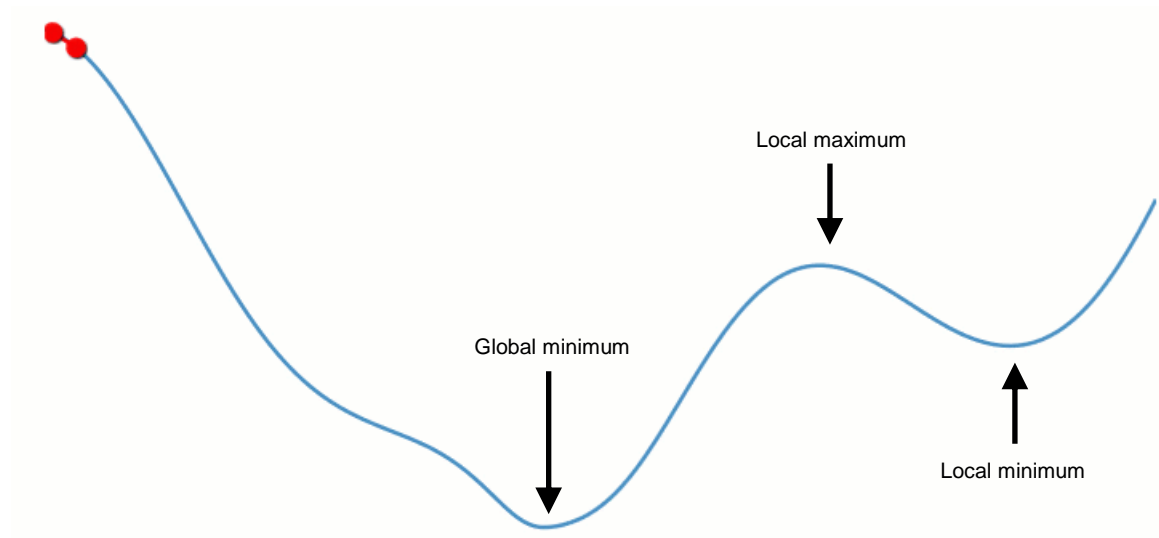


Noisy $f(\mathbf{X})$

Introduction to optimization methods

Single-objective optimization

- In single-objective optimization we are interested in optimizing one objective function.
- The optimization problem can be bounded and constrained.
- The optimization method can converge to a local or global optimal value.
- The optimal value depends on the initial value.



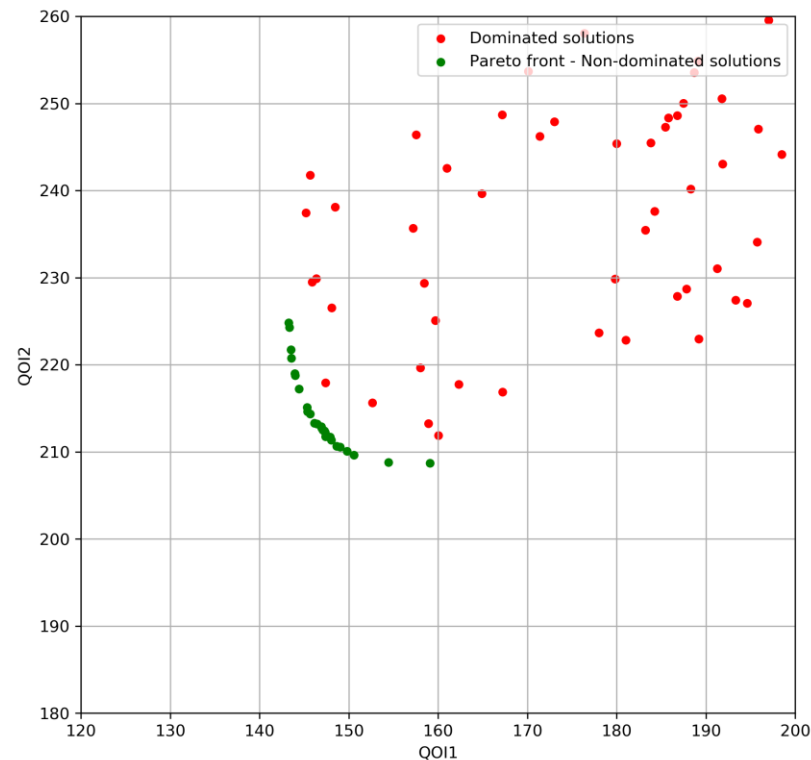
Multimodal function – Many global and local minima/maxima

<http://www.wolfdynamics.com/training/opt/ani1.gif>

Introduction to optimization methods

Multi-objective optimization

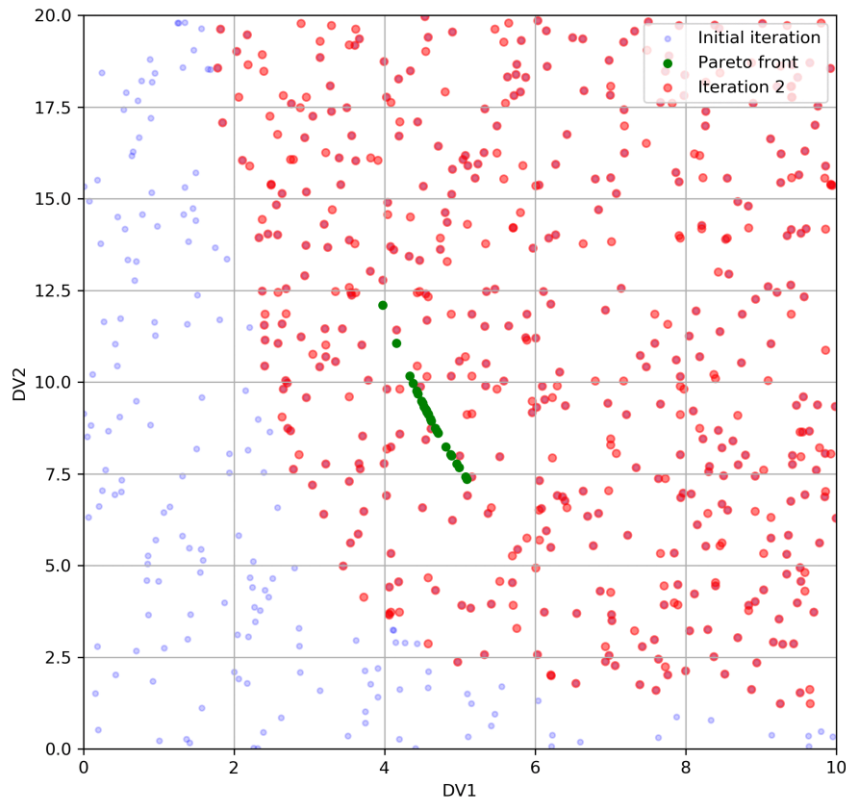
- In multi-objective optimization we are interested in optimizing more than one objective function or QOI simultaneously.
- The final goal is to find a representative set of optimal solutions (Pareto frontier or non-dominated solutions), quantify the trade-offs in satisfying the different objectives, and/or finding a single solution that satisfies the subjective preferences of a human decision maker.



Introduction to optimization methods

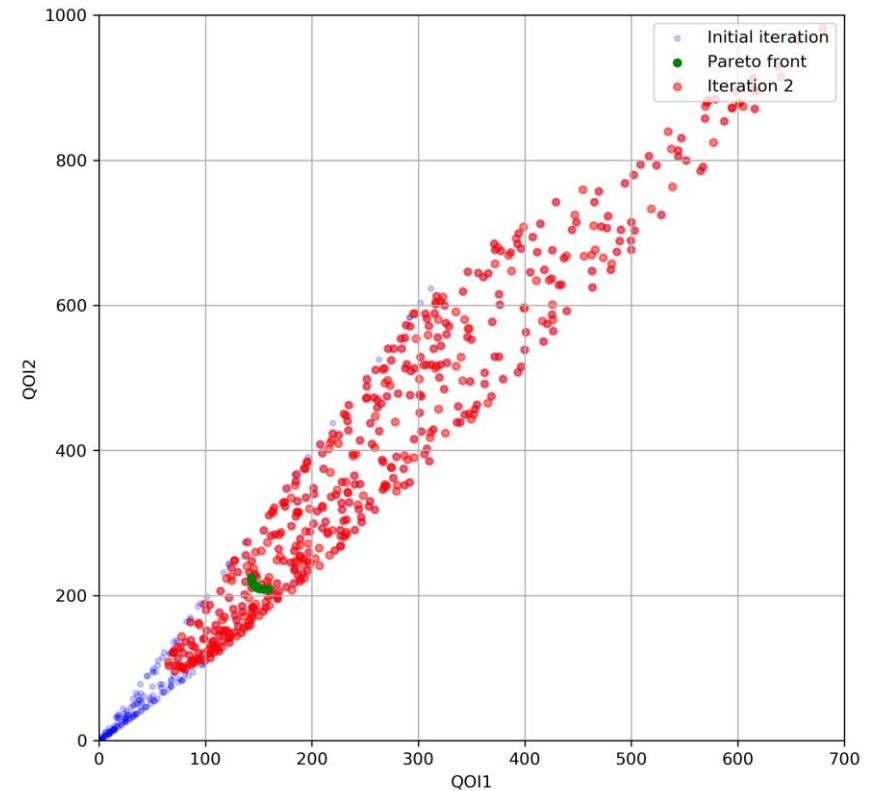
Multi-objective optimization

- Each optimal solution in the objective function space can be mapped to the design space.
- The functions to optimize can be constrained (linear and non-linear constraints).
- The design space can be bounded or unbounded.



Design space

<http://www.wolfdynamics.com/training/opt/ani4.gif>



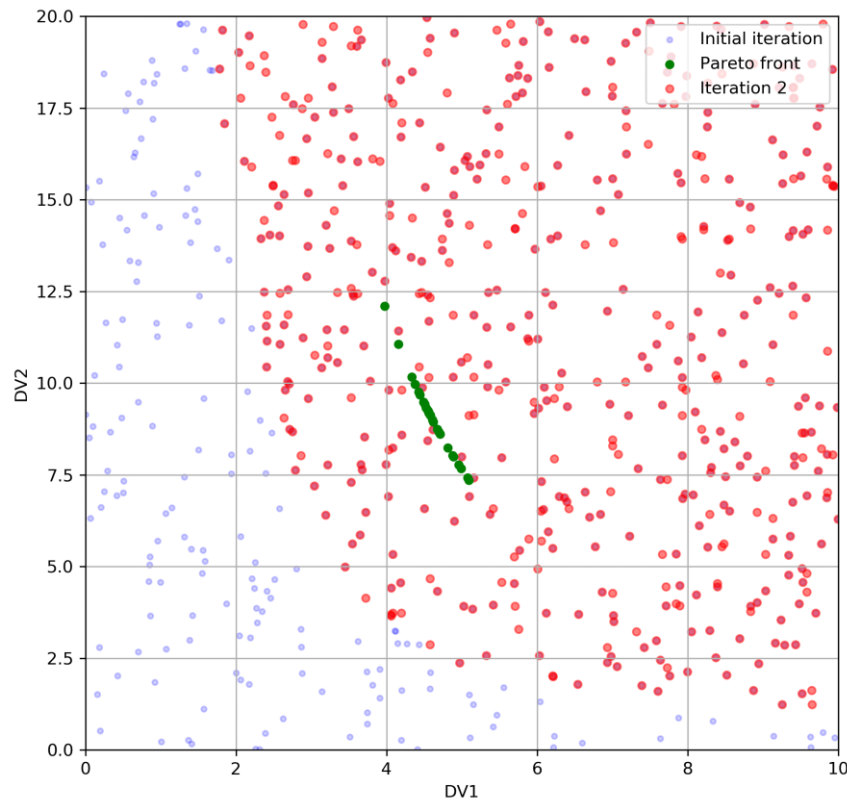
Objective function space

<http://www.wolfdynamics.com/training/opt/ani5.gif>

Introduction to optimization methods

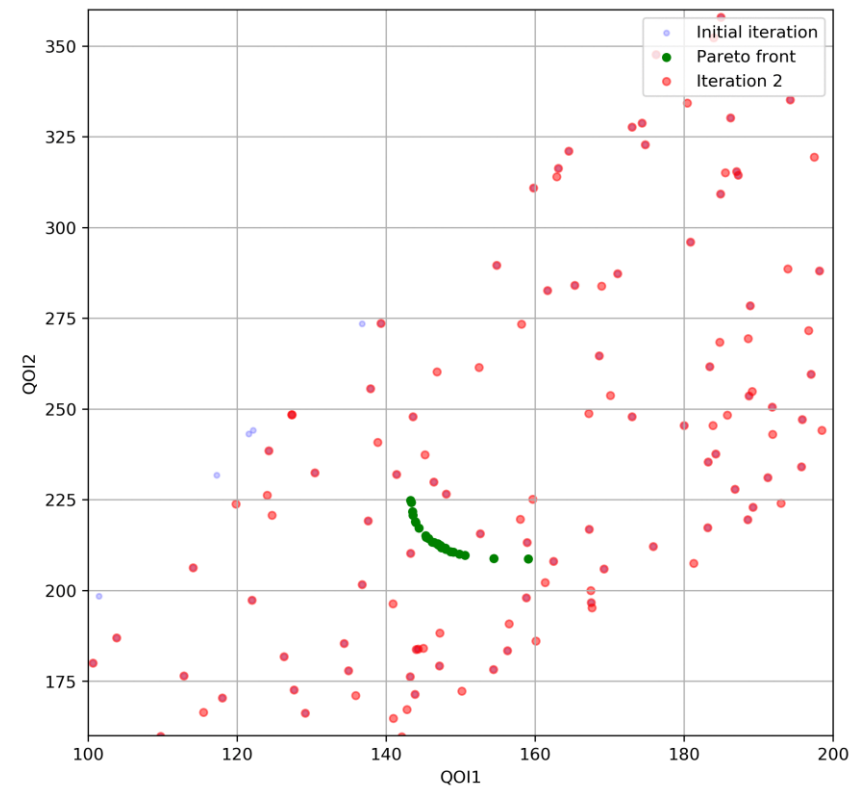
Multi-objective optimization

- Each optimal solution in the objective function space can be mapped to the design space.
- The functions to optimize can be constrained (linear and non-linear constraints).
- The design space can be bounded or unbounded.



Design space

<http://www.wolfdynamics.com/training/opt/ani4.gif>



Objective function space

<http://www.wolfdynamics.com/training/opt/ani5.gif>

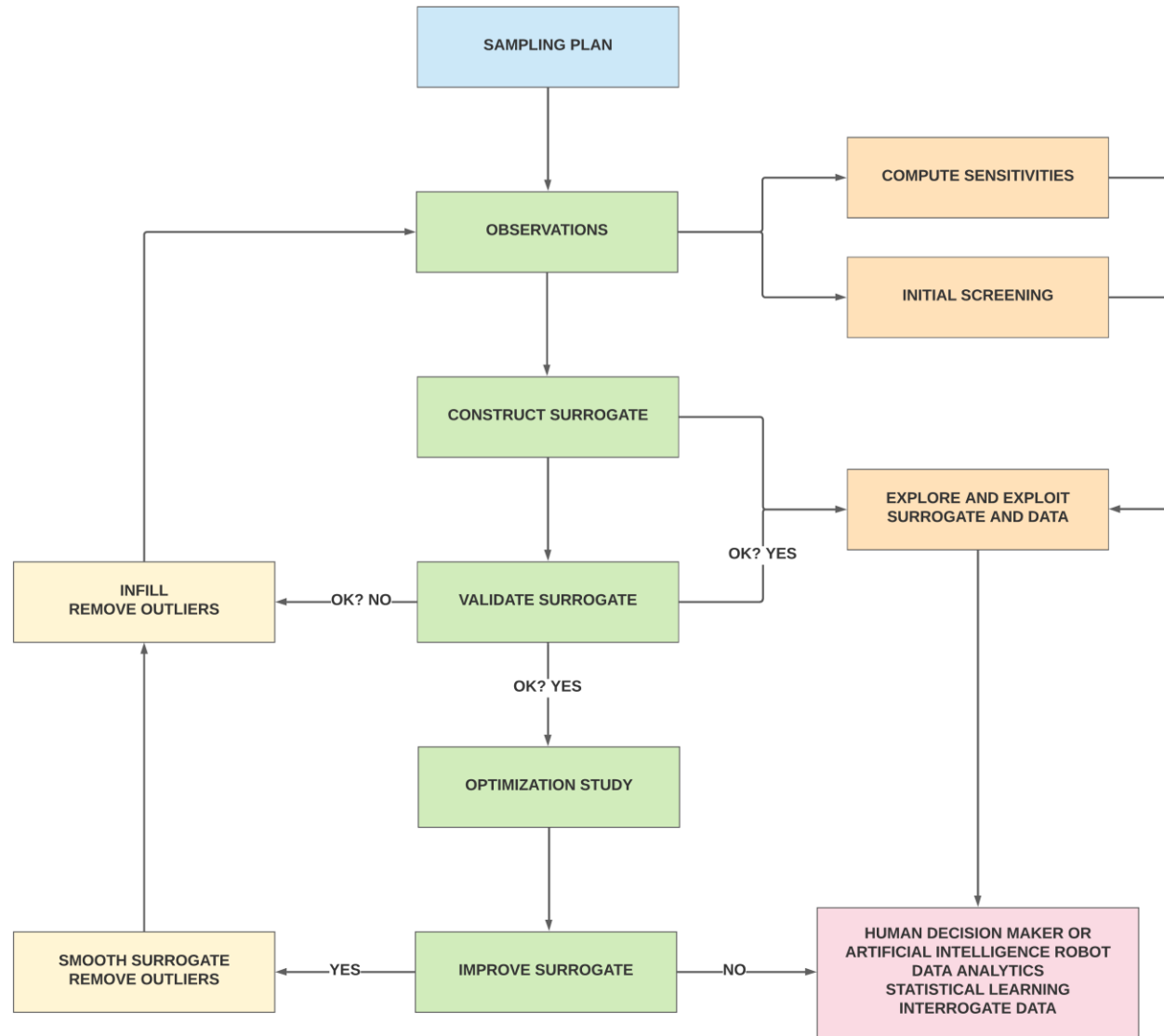
Introduction to optimization methods

Surrogate based optimization (SBO)

- When we do SBO, we use a surrogate model (also known as meta-model or response surface) to approximate an original high-fidelity model (e.g., expensive CFD simulations).
- The surrogate acts as a data fit or mathematical model to the observations so that new results can be predicted without recurring to expensive simulations.
- Once the surrogate is built, we can use any kind of optimization or calibration method.
- Evaluating the QOI at the surrogate level is inexpensive.
 - Working at the surrogate level is order of magnitudes faster than using high fidelity models.
- Surrogates can also be used with noisy and incomplete data.
- They can also be used for data mining and data analytics.
- In engineering design, surrogates can be used for initial screening and to provide information on the sensitivities of the data.

Introduction to optimization methods

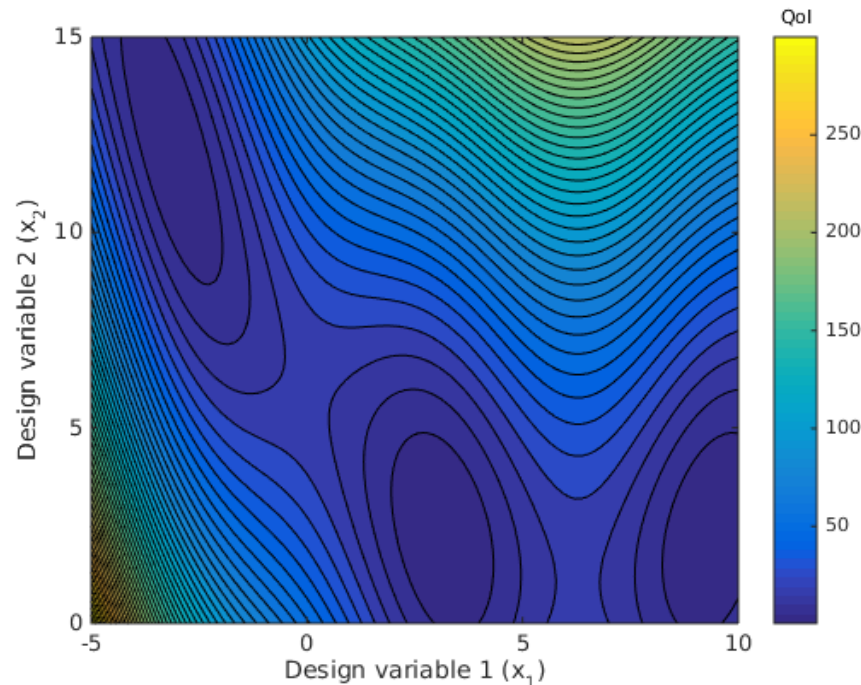
SBO workflow



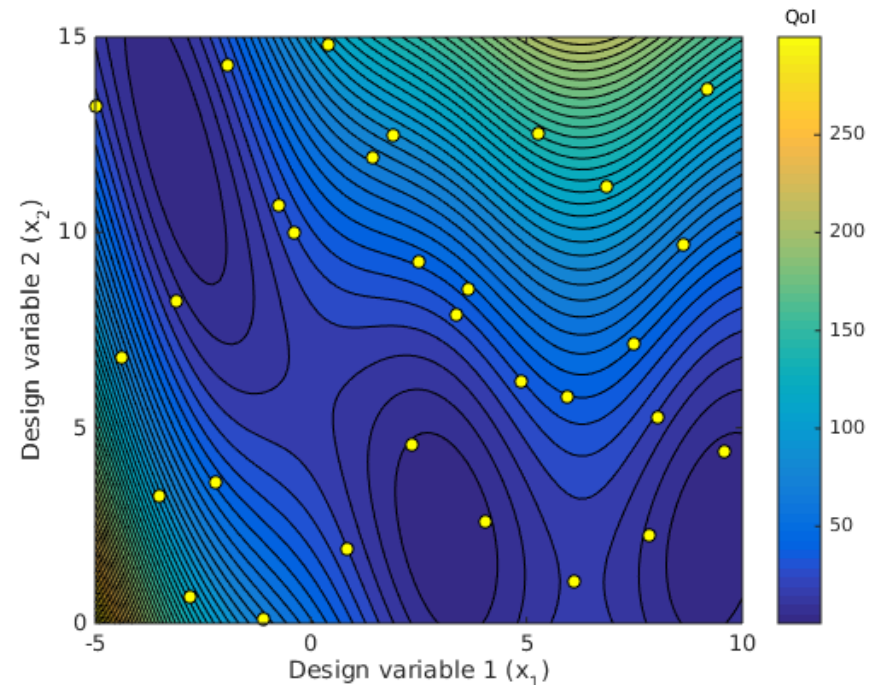
Introduction to optimization methods

Surrogate model

- Once the surrogate has been built, we can use any kind of optimization or calibration method.
- The surrogate is constructed using a limited number of observations in the design space.
- As we have a mathematical model of the observations, evaluating the QOI at the surrogate level is inexpensive.



Analytical function - Experiments



Surrogate, meta-model, response surface, data-fit, you name it.

Introduction to optimization methods

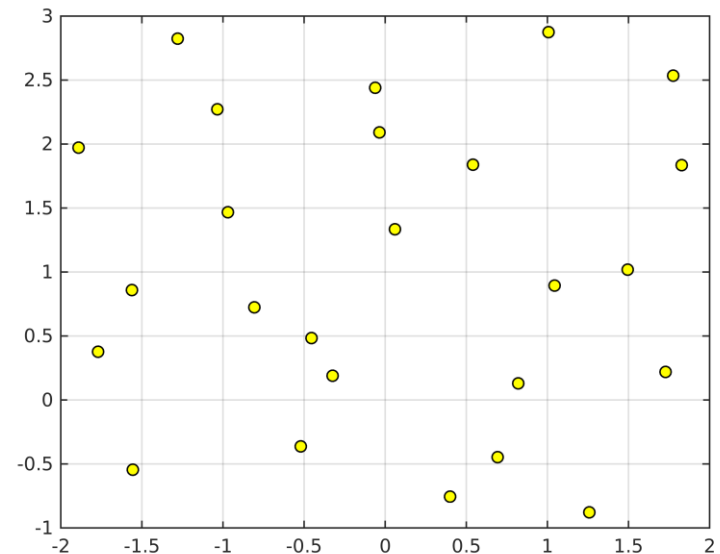
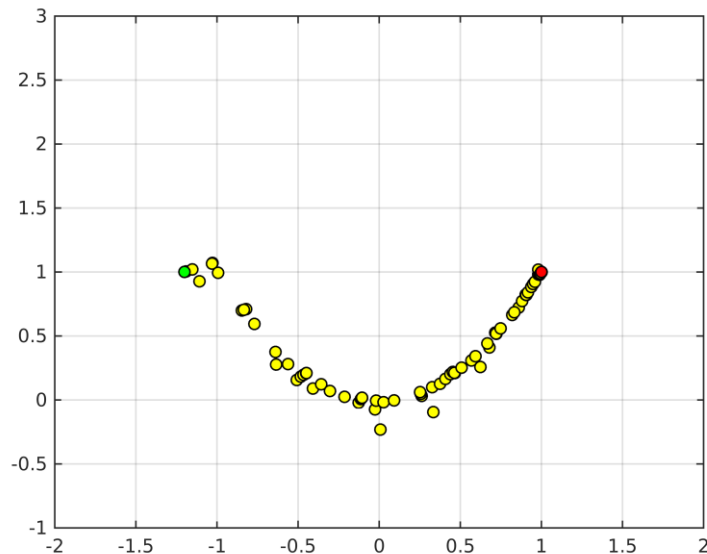
Design optimization vs. Design space exploration

Design optimization (DO)

- Converging-iterative process.
- DO aims at determining the optimum design.
- DO strategies have two distinct parts; formulate the problem and converge to the solution.
- DO depends on a well-posed optimization problem formulation (starting point, gradients, tolerance, etc).

Design space exploration (DSE)

- Diverging-iterative process.
- DSE aims at searching and characterizing the design space.
- Once we know the design space, a better solution can then be found through DO.
- Contrary to DO, in DSE we do not need a well formulated problem.

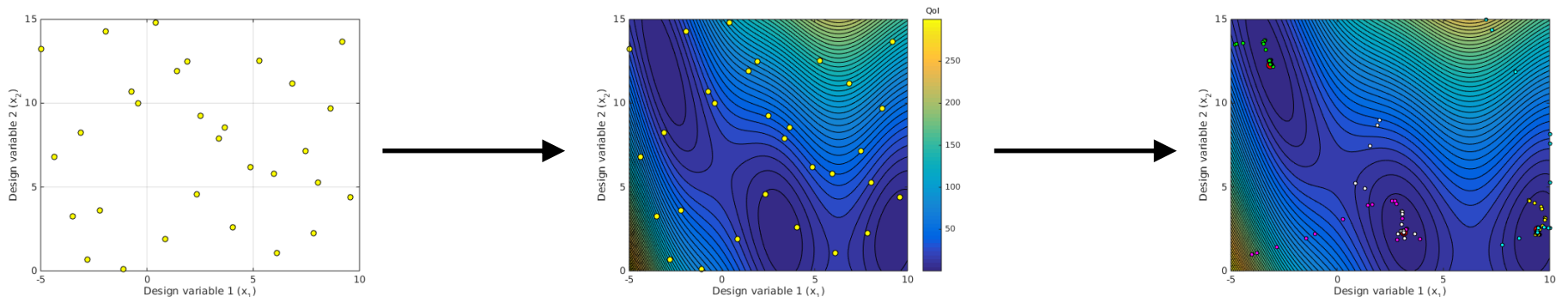


Introduction to optimization methods

Design optimization vs. Design space exploration

Surrogate-based optimization (SBO)

- It is a mix of DSE and DO (space exploration and converging-iterative process).
- SBO explores the design space from a limited number of observations (it can be used with high multi-dimensional design spaces).
- Then, SBO exploits and optimizes the design space by constructing a surrogate model (also known as meta-model, predictor model, or response surface).
- At the surrogate level, any optimization method can be used (gradient-based or derivative-free). Working at the surrogate level is orders of magnitude faster than working at the high-fidelity level.
- SBO is well fitted to engineering design. Especially during the conceptual and preliminary design phases.



Explore the design space

Construct the surrogate

Exploit and optimize at the surrogate level

Introduction to optimization methods

Design optimization vs. Design space exploration

- The essential difference between design optimization (DO) and design exploration (DSE) is the method used for characterizing the outcome.
 - In DO we use gradient-based or derivative-free methods to characterize the outcome.
 - In DSE we simply explore the design space using a method that covers the design space at a reasonable cost.

Introduction to optimization methods

Design optimization

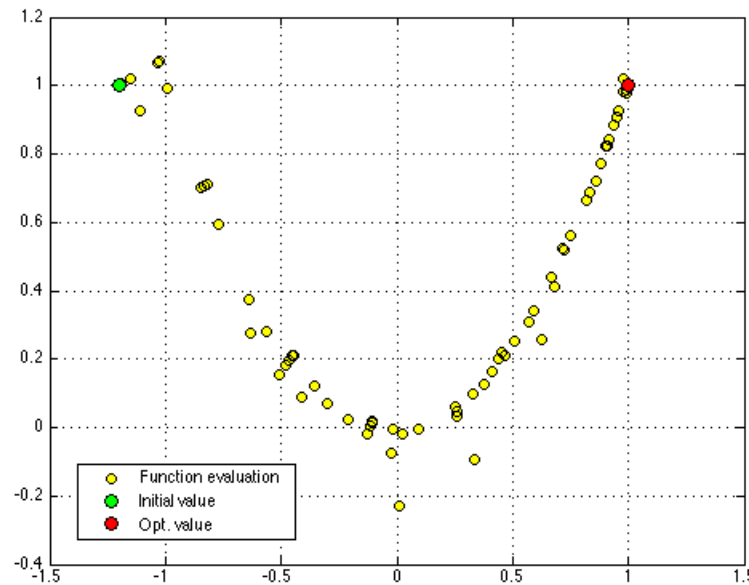
- **Design optimization (DO) is a converging-iterative process.**
- DO aims at determining the optimum design.
- DO strategies have two distinct parts;
 - Formulate the problem,
 - and converge to the solution.
- In DO it is assumed that the problem can be formulated before the search and convergence begins.
- DO depends on a well-posed optimization problem formulation.
- After defining the problem, we can proceed to find the optimal solution in an iterative way by using an appropriate search algorithm.
- In DO, many of the search algorithms used require the information related to the starting point, step size, stopping criteria, population size, and so on.

Introduction to optimization methods

Design optimization

Design optimization using a gradient-based method. Fletcher-Reeves method

- Gradient-based optimizers are best suited for efficient navigation to a local minimum in the vicinity of the initial point.
- They are not intended to find global optima in nonconvex design spaces.



Introduction to optimization methods

Design space exploration

- **Design space exploration (DSE) is diverging-converging-iterative process.**
- DSE is based on the belief that the problem formulation evolves during the process of searching the design space.
- The main idea of design space exploration is to search the design space in a very efficient way at a minimal cost.
- Once this is known, a better solution can then be found through DO.
- Contrary to DO, in DSE we do not need a well formulated problem.
- That is, we do not need to provide information related to the starting point, step size, stopping criteria, population size, and so on.

Introduction to optimization methods

Design space exploration

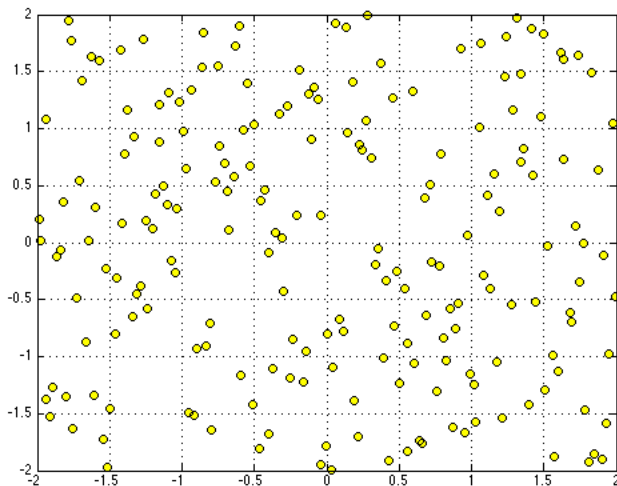
- The main idea of design space exploration is to search the design space in a very efficient way at a minimal cost.
- When we conduct design space exploration, we follow a systematic mathematical or statistical approach to acquire model behavior to the maximum extent.
- With design space exploration we can:
 - Gain a deep statistical understanding of the problem.
 - Explore a wide design space through intelligent sampling.
 - Identify the most important influencing design variables.
 - Create accurate mathematical models.
 - Provide a set of starting points for design optimization.

Introduction to optimization methods

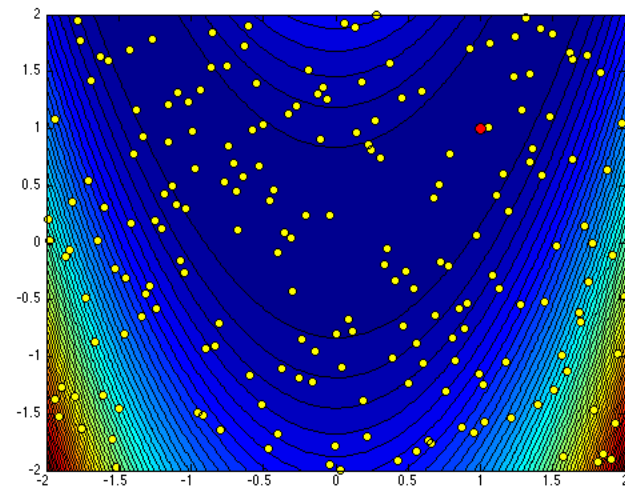
Design space exploration

Space filling - Latin Hypercube Sampling (LHS)

- LHS is a DACE type method (design and analysis of computer experiments).
- DACE methods are ideal for deterministic experiments (computer simulations) and design space exploration. In computer experiments we are interested in sampling the parameter space in a representative way with the minimum number of samples.
- **The output can be used in a sensitivity analysis, uncertainty quantification, or building a surrogate (mathematical model).**



LHS sampling



Response surface constructed using DACE sampling

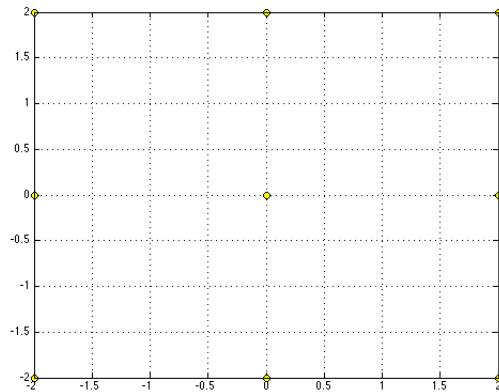
Introduction to optimization methods

Design space exploration

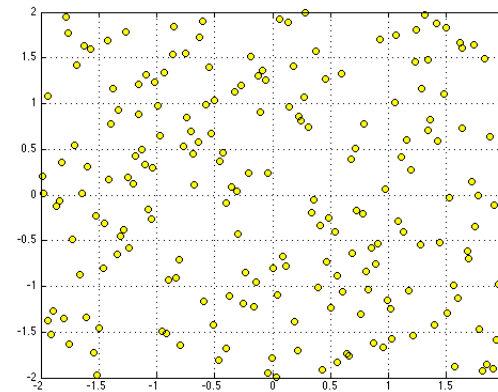
- Design space exploration studies can be conducted using multi-dimensional studies full factorial), and with computer experiments (e.g., DACE or design analysis of computer experiments).
- Multi-dimensional studies explore the effect of parametric changes within simulation models by computing response data sets at a selection of points in the parameter space, yielding one type of sensitivity analysis.
 - The selection of points is deterministic and structured, or user-specified.
- Classical **design of experiments** (DoE) methods and the more modern **design and analysis of computer experiments** (DACE) methods are both techniques which seek to extract as much trend data from a parameter space as possible using a limited number of sample points.
- In DACE, the sampling is stochastic and covers most of the design space (space filling experiments).
- A few DACE sampling techniques: orthogonal array sampling, Latin hypercube sampling, Quasi-Monte Carlo sampling.

Introduction to optimization methods

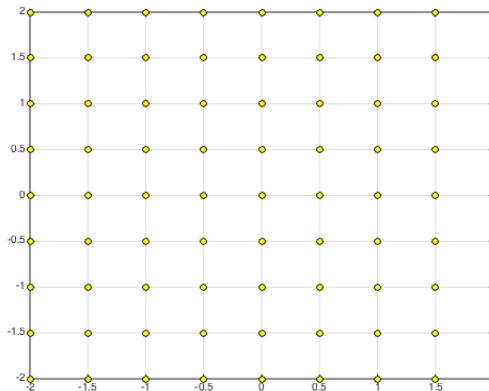
Design space exploration



DOE experiment



DACE experiment



Multidimensional study

Introduction to optimization methods

Design space exploration

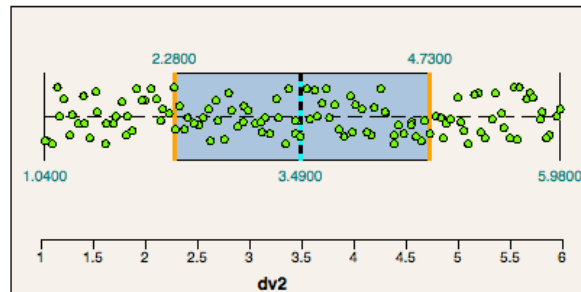
What else can we do with all the data collected?

- When conducting DSE studies, we collect a lot data.
- This data can be used to get a much better insight of the problem.
- We can analyze the data using machine learning, statistical learning, and exploratory data analysis techniques.
- We can also create bespoke visualizations and dashboards to communicate the data.

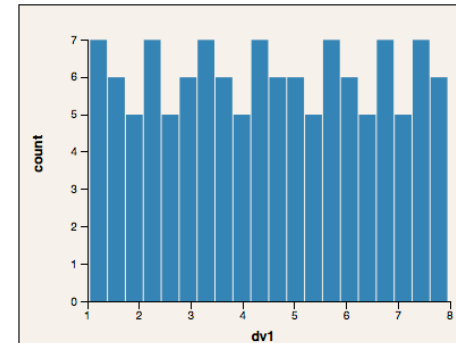
Introduction to optimization methods

Data visualization, machine learning and statistical analysis

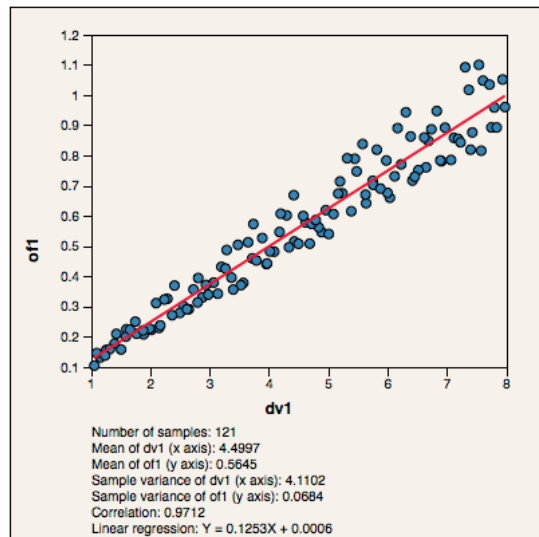
Boxplot



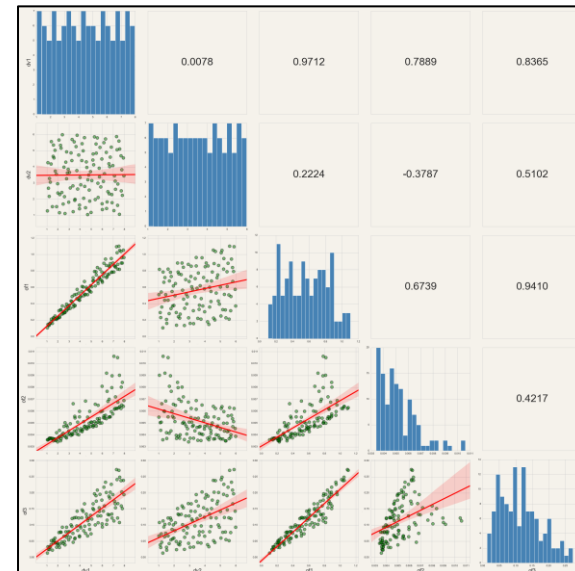
Histogram



Scatter plot + regression

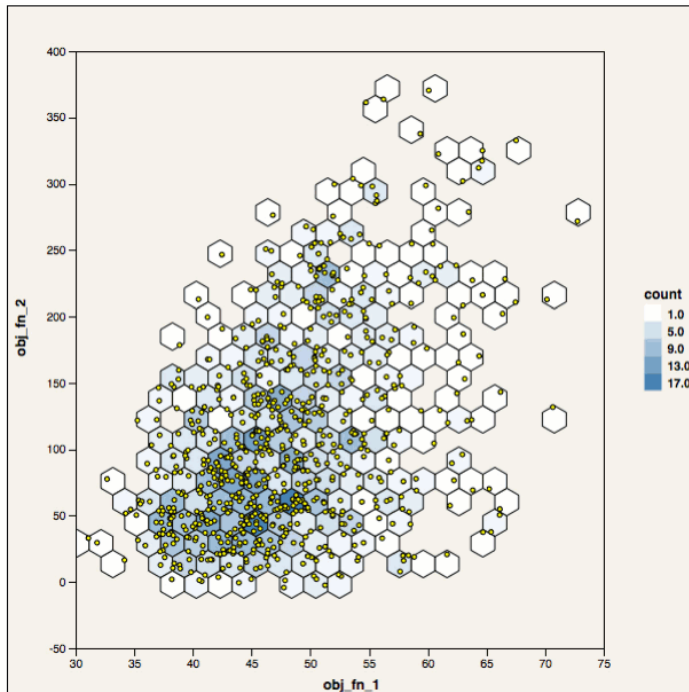


Scatter matrix plot



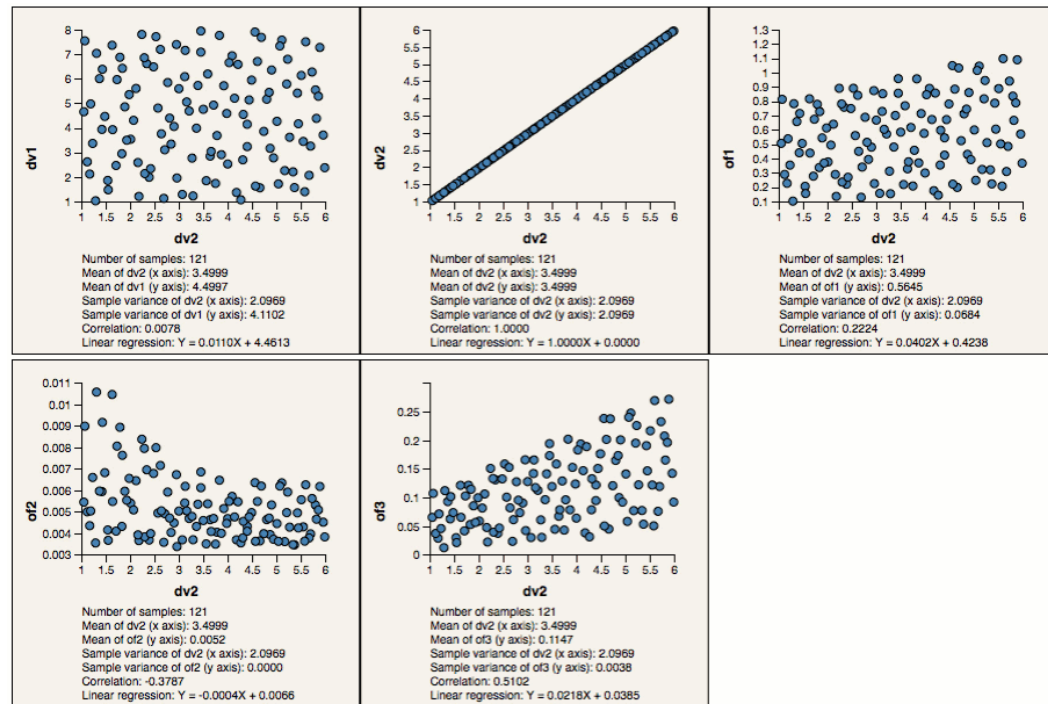
Introduction to optimization methods

Data visualization, machine learning and statistical analysis



Hexbin plot

<http://www.wolfdynamics.com/training/opt/image1.gif>

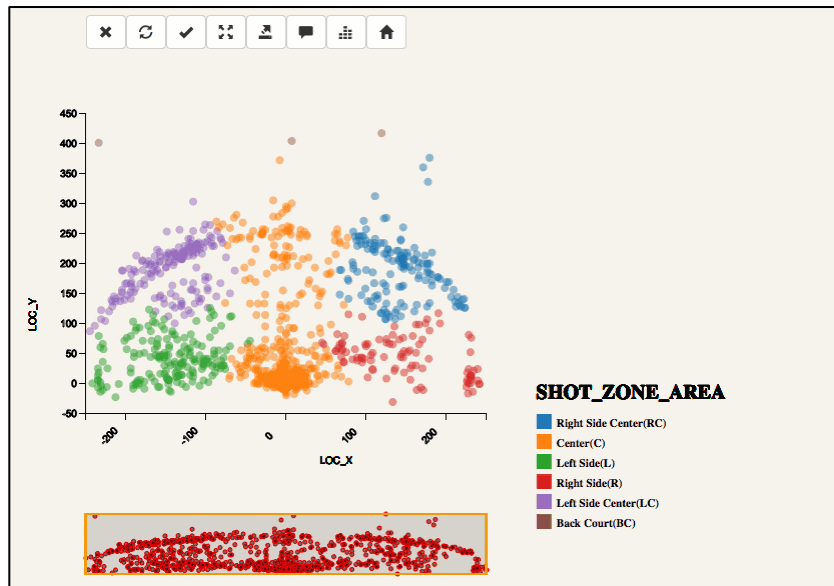


Regression plot – Paired plot

<http://www.wolfdynamics.com/training/opt/image2.gif>

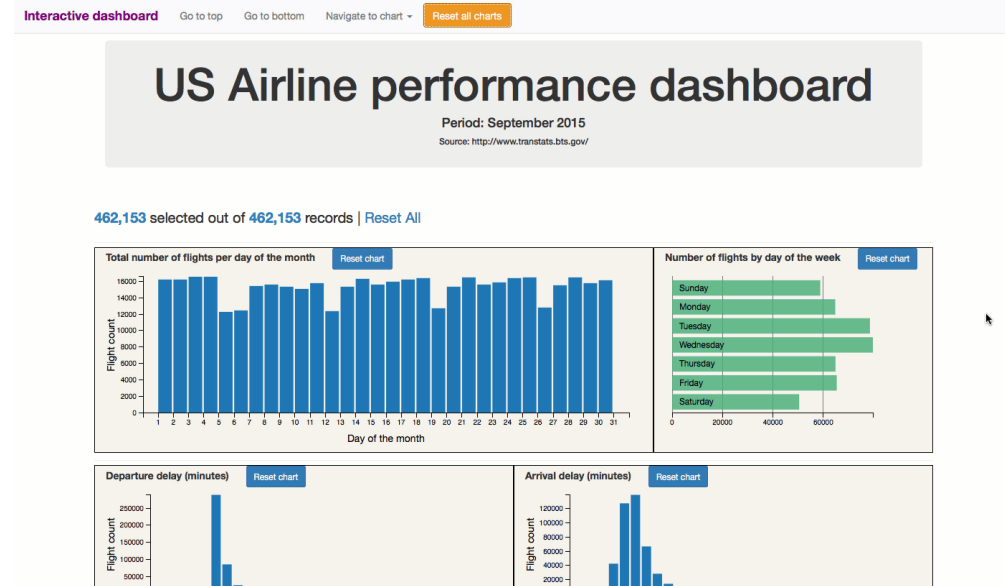
Introduction to optimization methods

Data visualization, machine learning and statistical analysis



Interactive scatter plot

<http://www.wolfdynamics.com/training/opt/image4.gif>



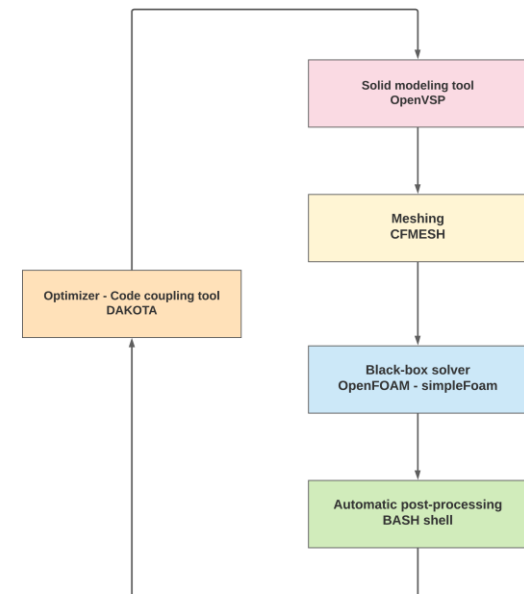
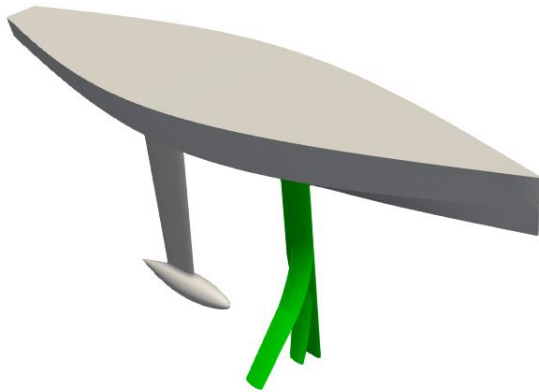
Interactive dashboard with cross filtering of data

<http://www.wolfdynamics.com/training/opt/image5.gif>

Introduction to optimization methods

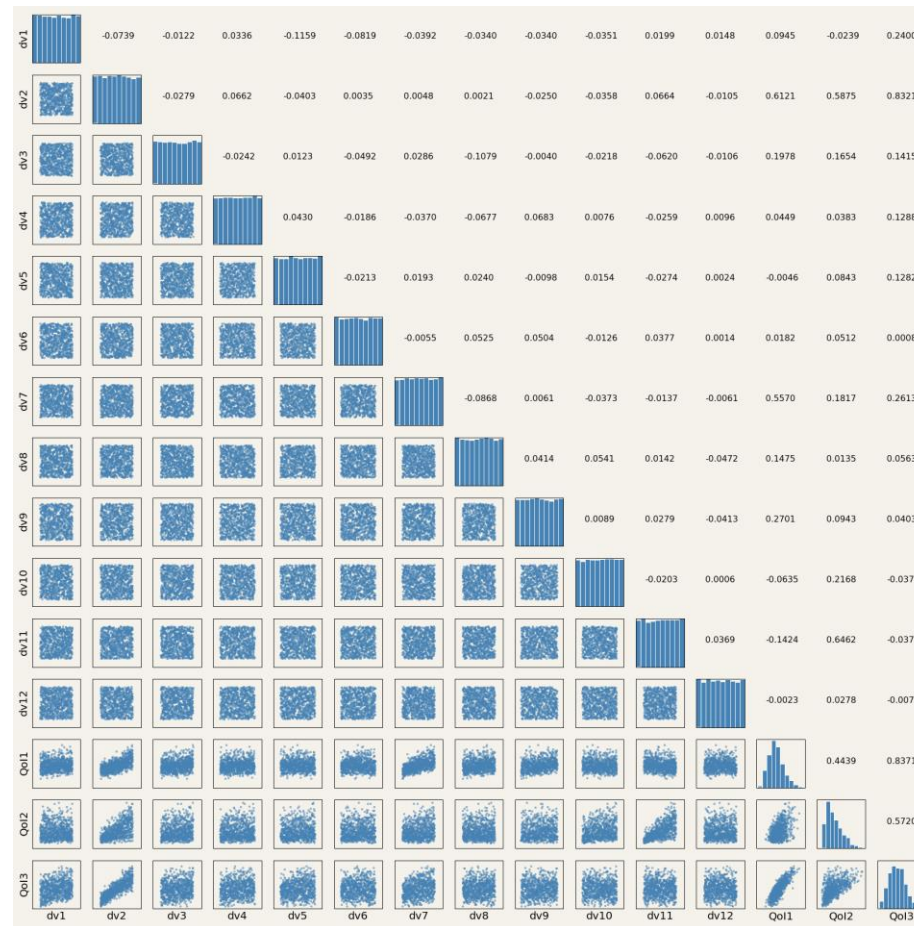
Data visualization, machine learning and statistical analysis

- To get a better idea how data visualization, machine learning and statistical analysis (which I prefer to call knowledge extraction) can help us to understand better a DO or DSE study, let us take a look at the following case: Sailing yacht daggerboard optimization case
- The goals were to maximize the vertical force and minimize the drag coefficient (2 objective functions).
- There are 12 design variables and 1 non-linear constraint (the lateral force on the daggerboard).
- All design variables are bounded and for the non-linear constraint we use an inequality.
- All the simulations were conducted in a workstation with 24 cores and in less than 32 hours.



Introduction to optimization methods

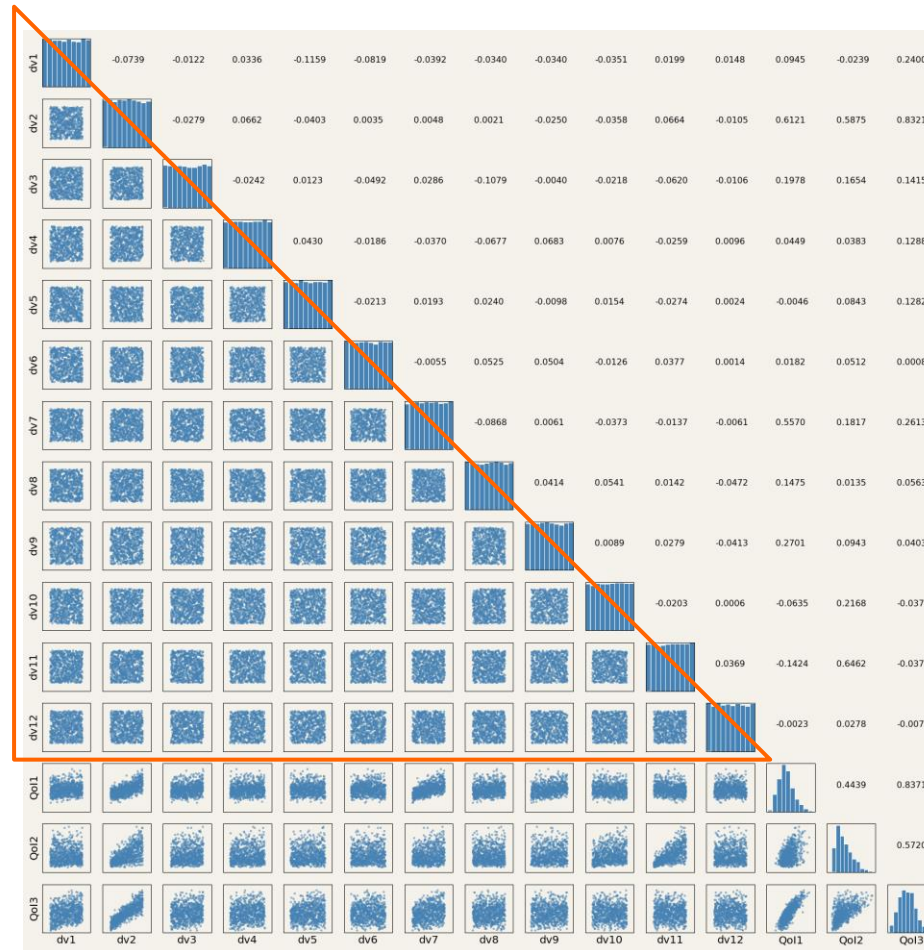
Data visualization, machine learning and statistical analysis



Scatter plot matrix of a DSE study – 700 experiments (high fidelity simulations)
This plot shows correlation, skewness, kurtosis, tendency and distribution of the data

Introduction to optimization methods

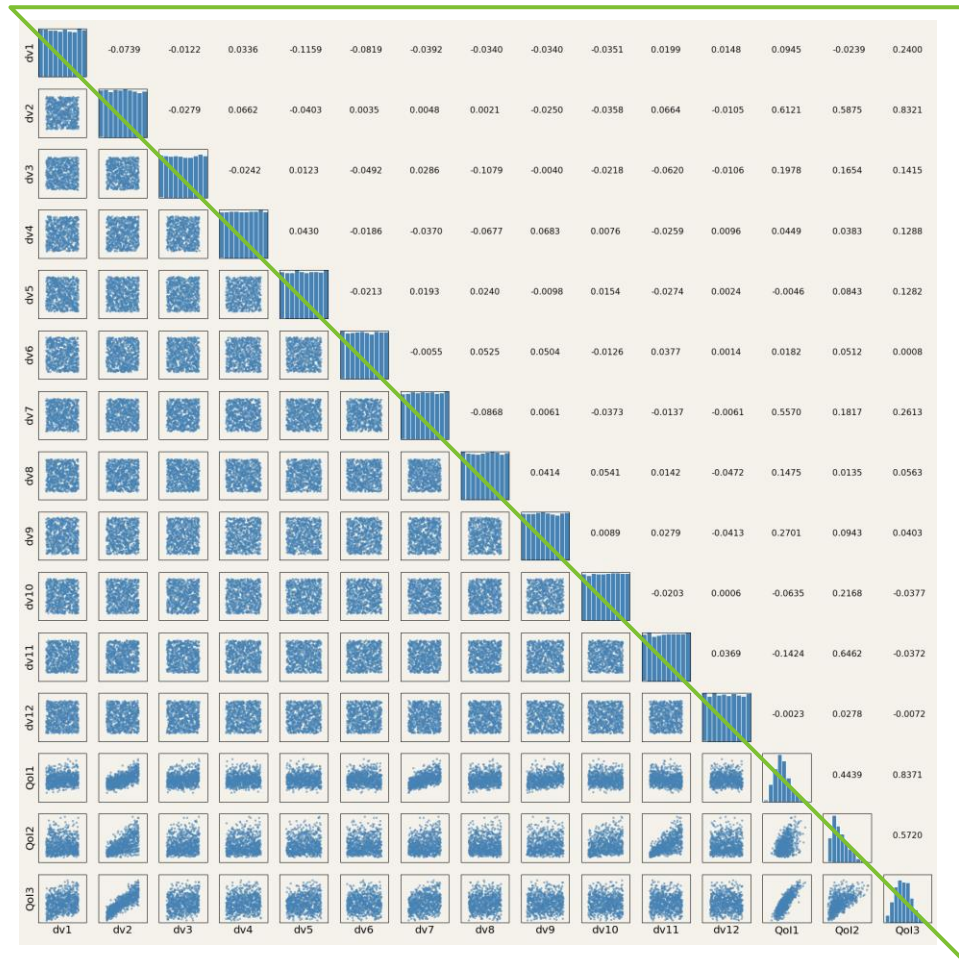
Data visualization, machine learning and statistical analysis



Scatter plot of design variables distribution (sampling distribution in design space)
Scatter plot matrix – 700 experiments (high fidelity simulations)

Introduction to optimization methods

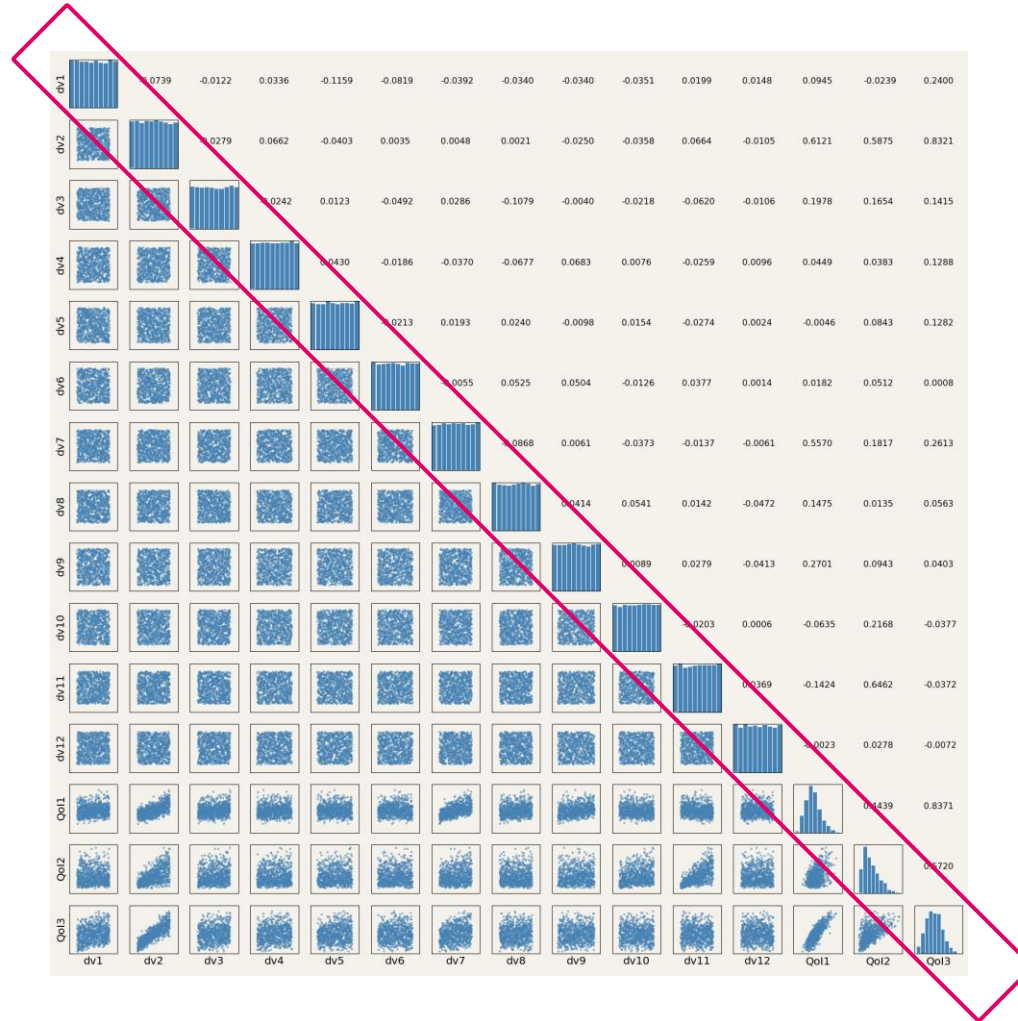
Data visualization, machine learning and statistical analysis



Correlation matrix of design space variables (design variables and objective functions)
Scatter plot matrix – 700 experiments (high fidelity simulations)

Introduction to optimization methods

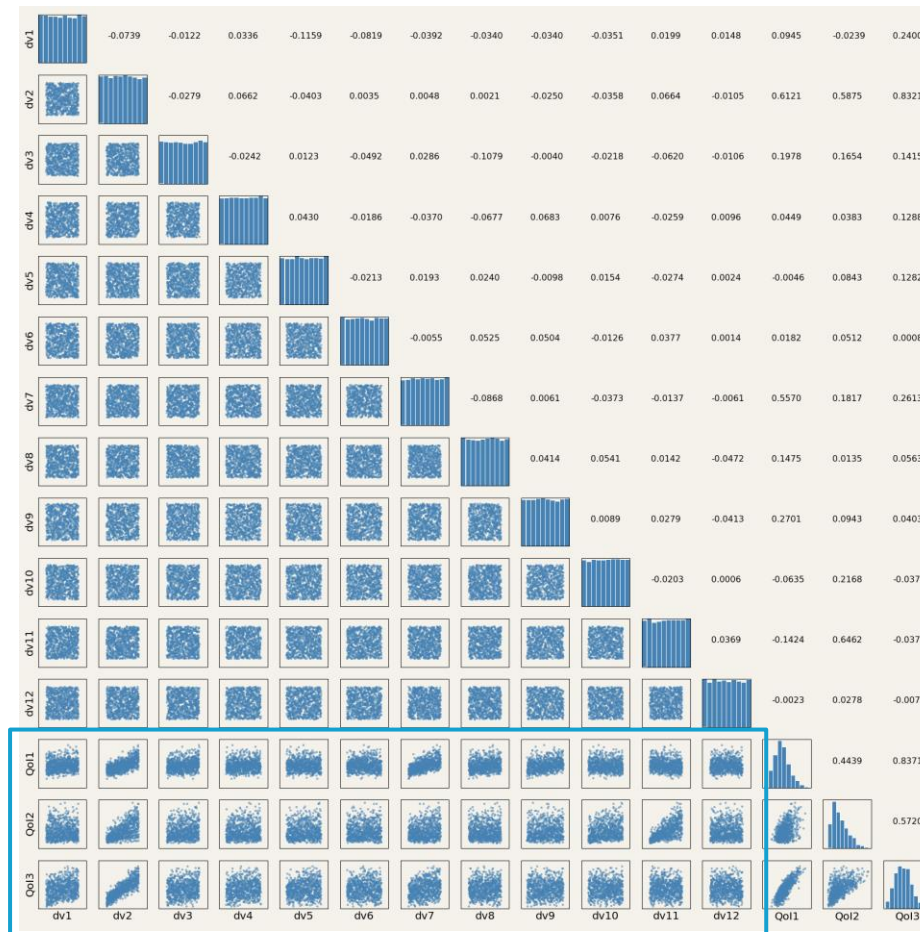
Data visualization, machine learning and statistical analysis



Histograms of design variables and objective functions
Scatter plot matrix – 700 experiments (high fidelity simulations)

Introduction to optimization methods

Data visualization, machine learning and statistical analysis

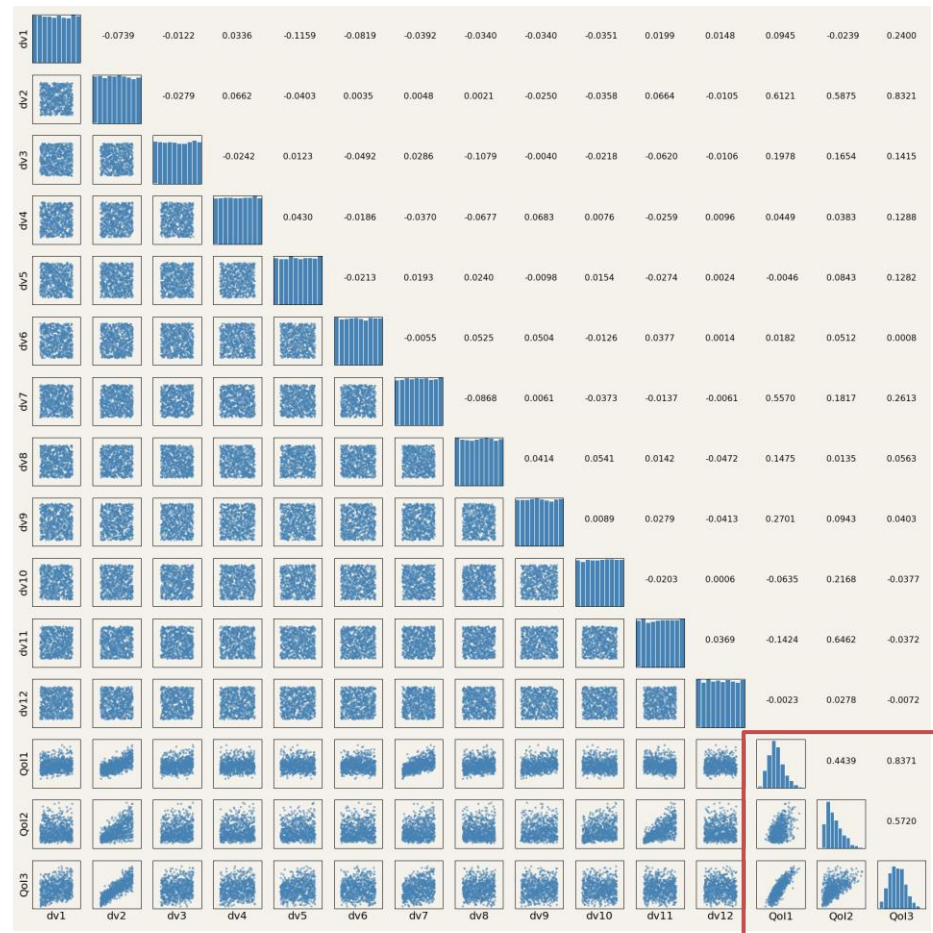


Response of design space (scatter plot of design variables vs. objective functions)

Scatter plot matrix – 700 experiments (high fidelity simulations)

Introduction to optimization methods

Data visualization, machine learning and statistical analysis

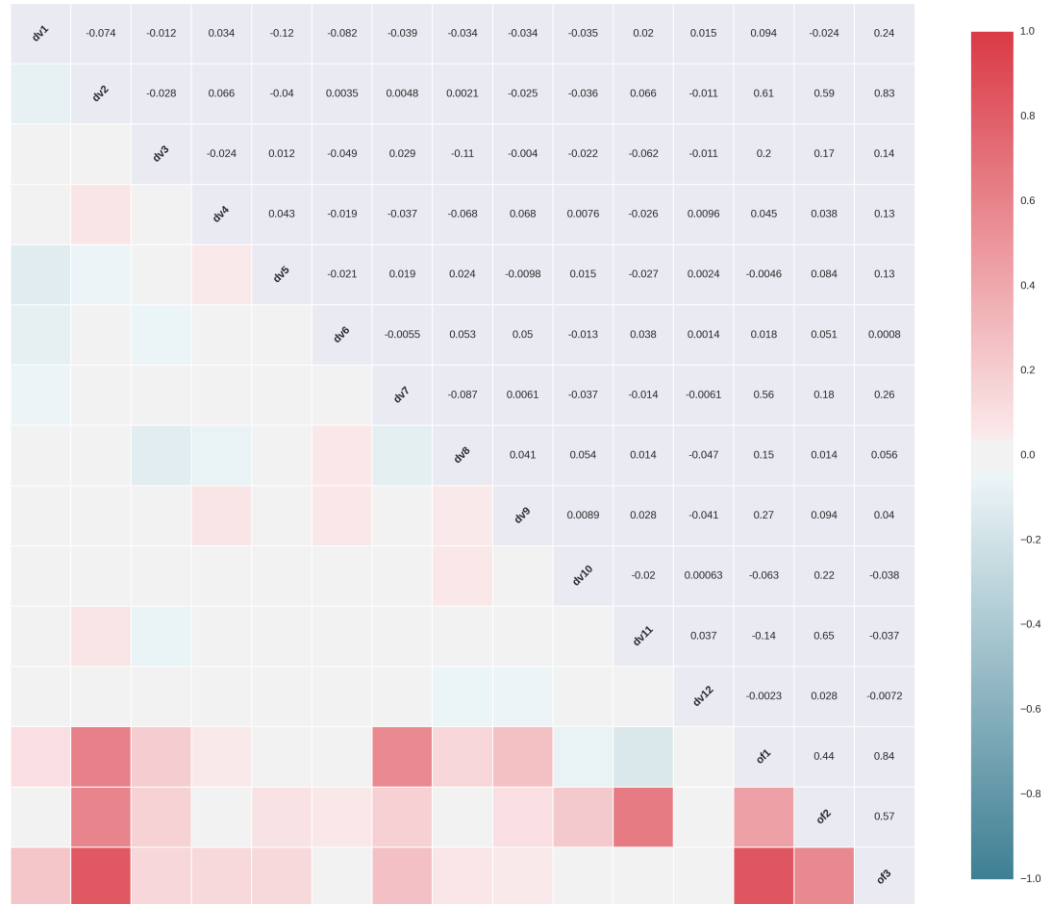


Response or trade-off of objective functions

Scatter plot matrix – 700 experiments (high fidelity simulations)

Introduction to optimization methods

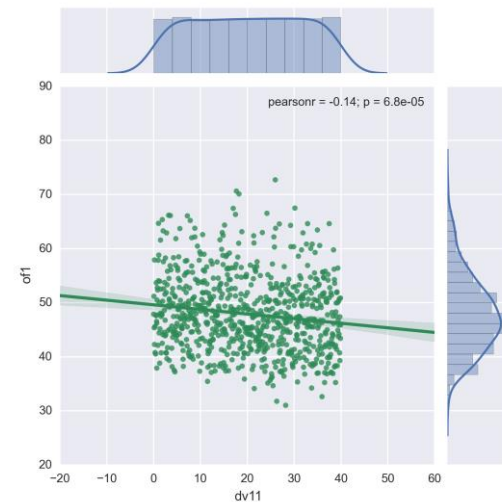
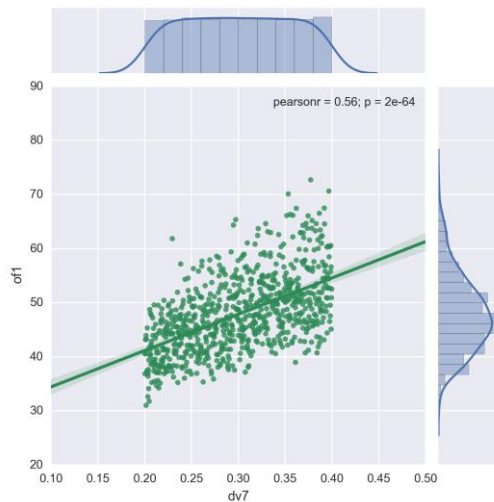
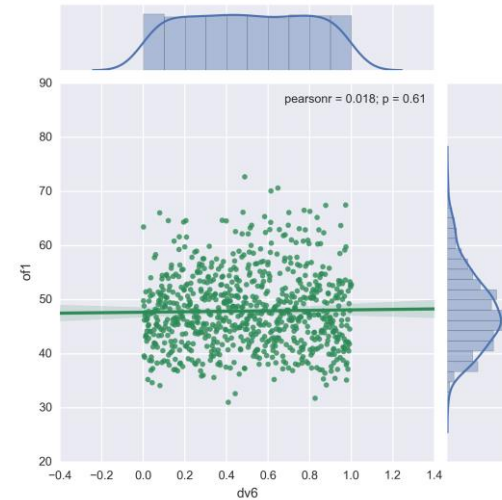
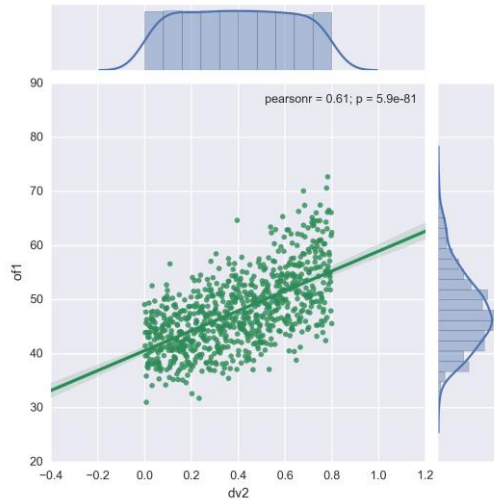
Data visualization, machine learning and statistical analysis



Correlation matrix (spearman coefficient) – 700 experiments (high fidelity simulations)

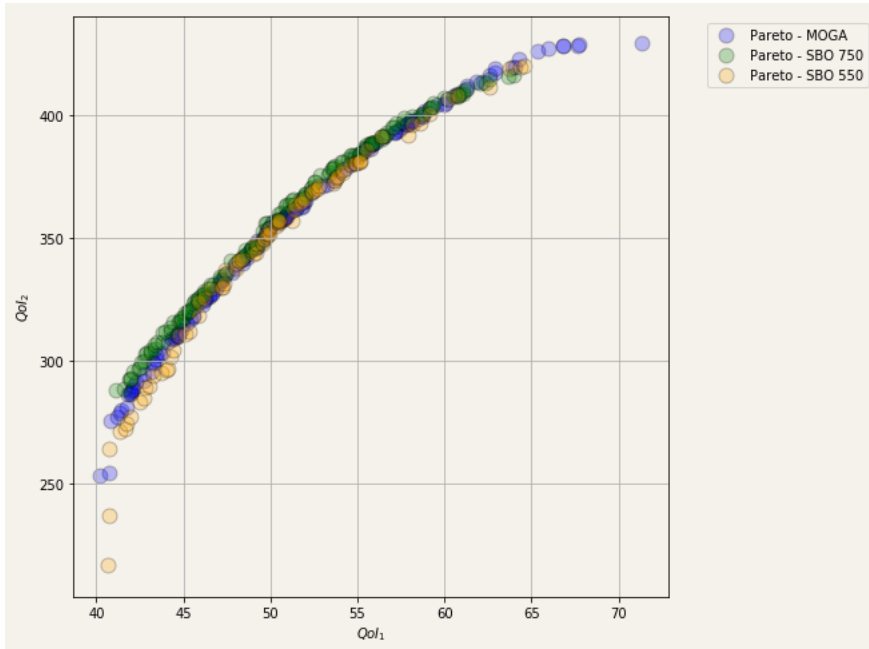
Introduction to optimization methods

Data visualization, machine learning and statistical analysis

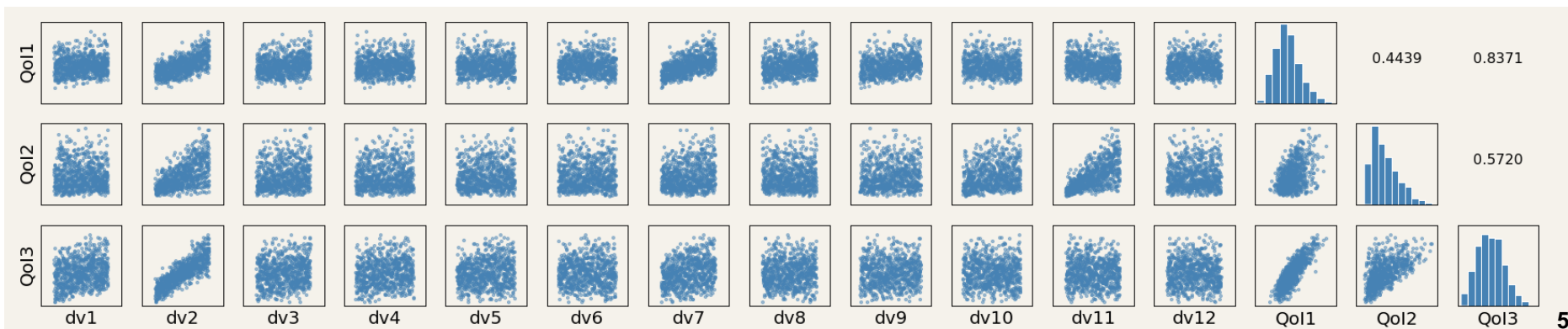


Introduction to optimization methods

Data visualization, machine learning and statistical analysis



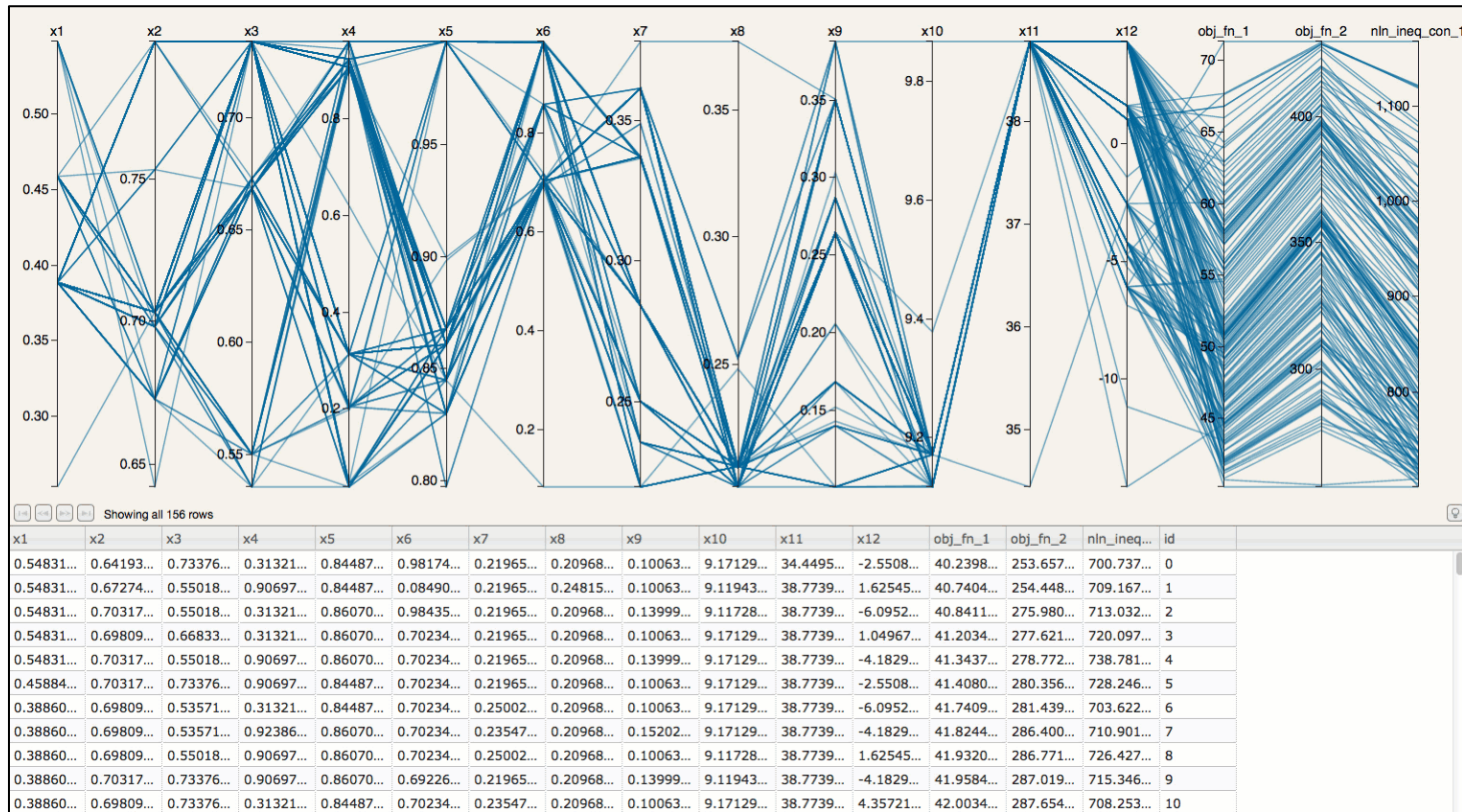
- Pareto front and overall response of the design space.
- The MOGA pareto was constructed using 1500 high-fidelity simulations and a genetic algorithm.
- The SBO 750 pareto was constructed using 750 high-fidelity simulations and kriging interpolation.
- The SBO 550 pareto was constructed using 550 high fidelity simulations and kriging interpolation.
- By using SBO we were able to obtain similar response in half the computational time.
- Plus the insight gained from the DSE study.



Introduction to optimization methods

Data visualization, machine learning and statistical analysis

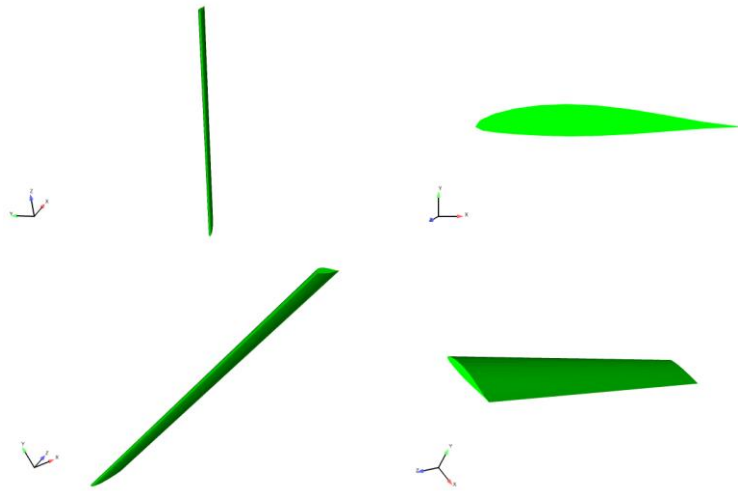
- When working with multi-dimensional data, the best way to explore the data is by using interactive parallel coordinates.
- These plots let us easily find correlations between design variables and system responses.



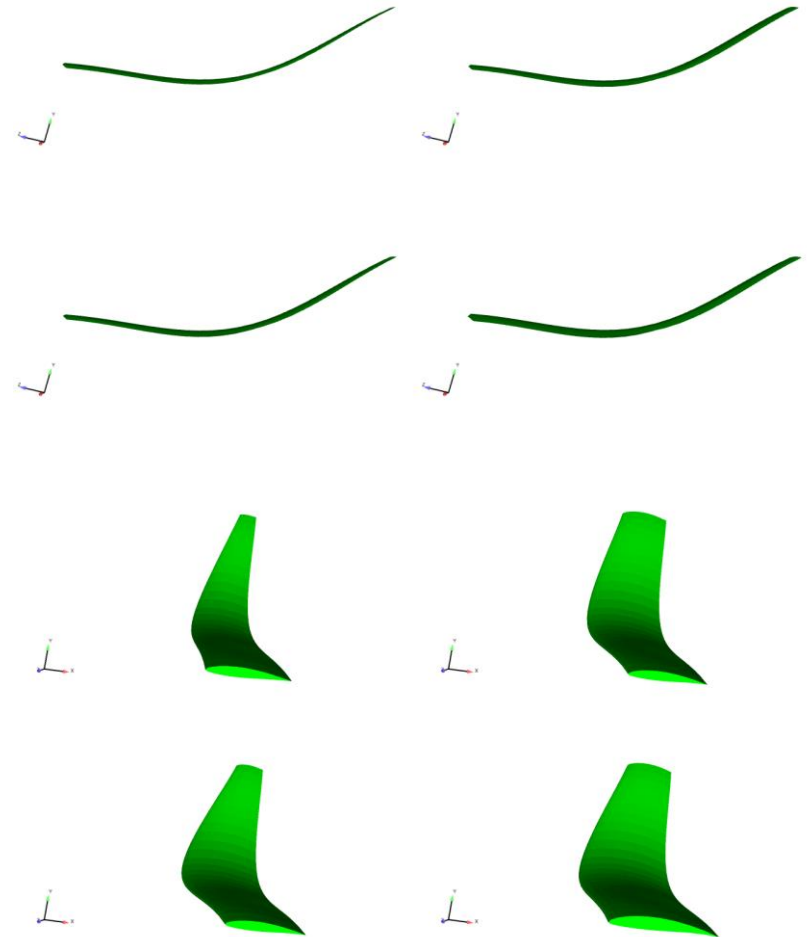
Interactive parallel coordinates

Introduction to optimization methods

Data visualization, machine learning and statistical analysis



Initial geometry



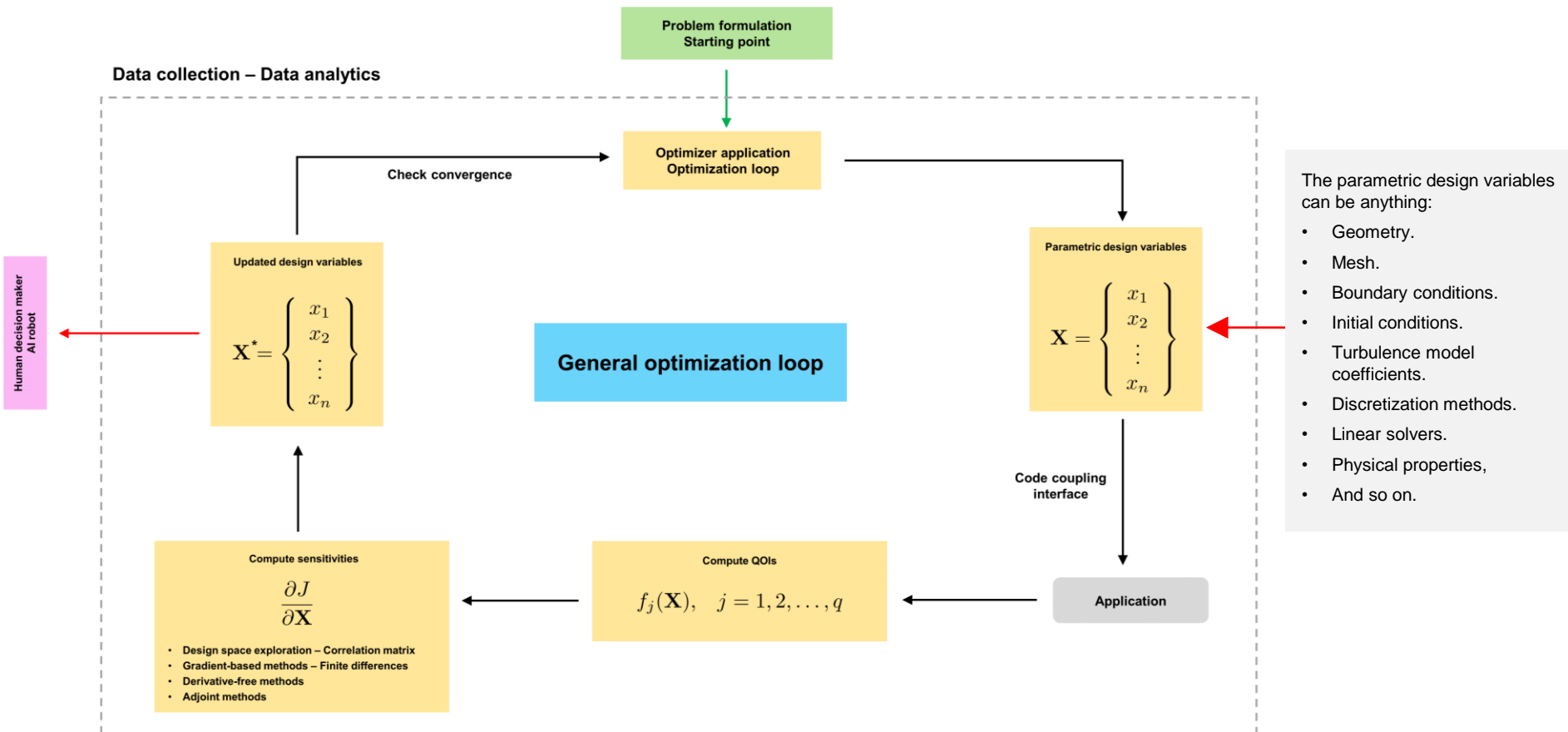
Optimal candidates

Four non-dominated solutions belonging to the pareto 54

General optimization loop in CFD

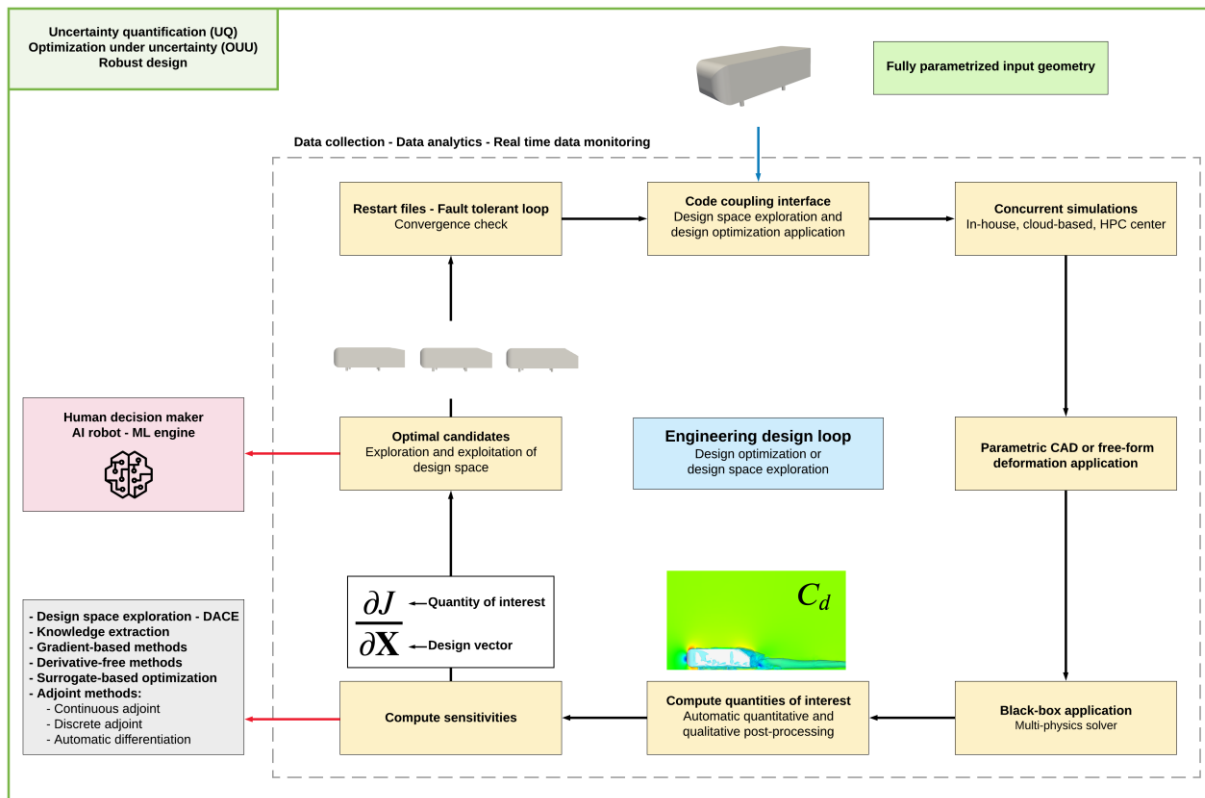
Introduction to optimization methods

- Graphical summary of a general optimization loop in CFD.
- This optimization loop can be based on a loosely coupled approach, where all the applications interact via shell scripting or the command line interface.
- Or in a strongly coupled approach (or monolithic framework), where all applications interact using a single interface, graphical or command line based (usually seen in commercial applications).



Introduction to optimization methods

- For example, a loosely coupled approach for shape optimization in CFD might look like the following one.
- To conduct different tasks in the optimization loop, we use different tools that can interact between each another using shell scripting or the command line interface.
- Also, all the proposed applications are open-source.
- This framework can be extended to any engineering application (aerospace, automotive, HVAC, AEC, medical devices thermal management, naval, and so on).



- **Code coupling/Optimizer:**
DAKOTA
- **Concurrent computations scheduler:**
DAKOTA
- **Parametric CAD:**
Onshape (API)
- **Black-box solver:**
OpenFOAM
- **Quantitative and qualitative post-processing:**
Python, paraview, JavaScript
- **Real time data monitoring:**
Python, R, BASH
- **Exploration and exploitation of design space:**
Python, R, BASH
- **Additional automation scripting:**
Python, BASH

Shape optimization methods in CFD

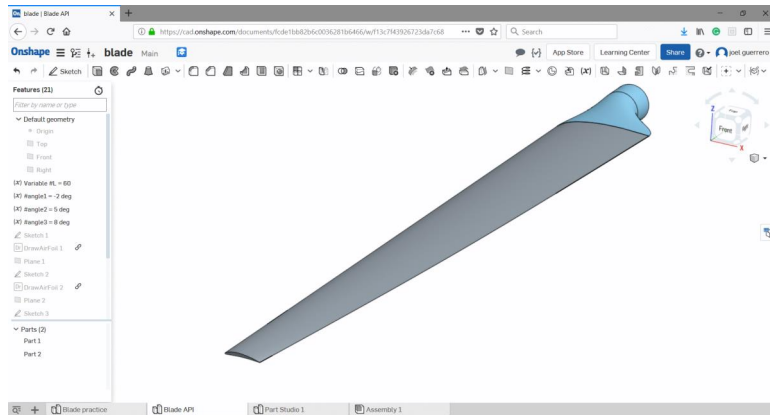
Introduction to optimization methods

Shape optimization methods in CFD

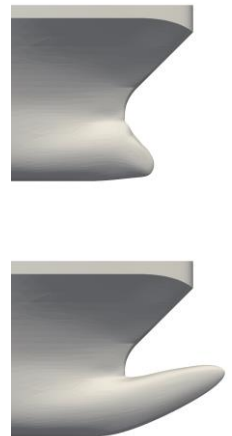
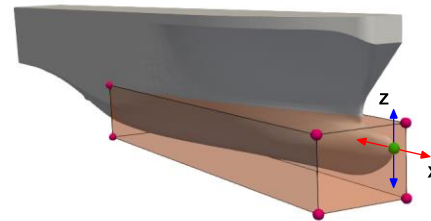
- We can also classify the optimization methods used in CFD according to the geometry parametrization level:
 - Parameter-**based** or CAD based optimization.
 - Parameter-**free** or free-form optimization.



<http://www.wolfdynamics.com/training/opt/ani7.gif>



Parameter-based or CAD based optimization



Parameter-free or free-form optimization

Introduction to optimization methods

Shape optimization methods in CFD – Parameter-based optimization

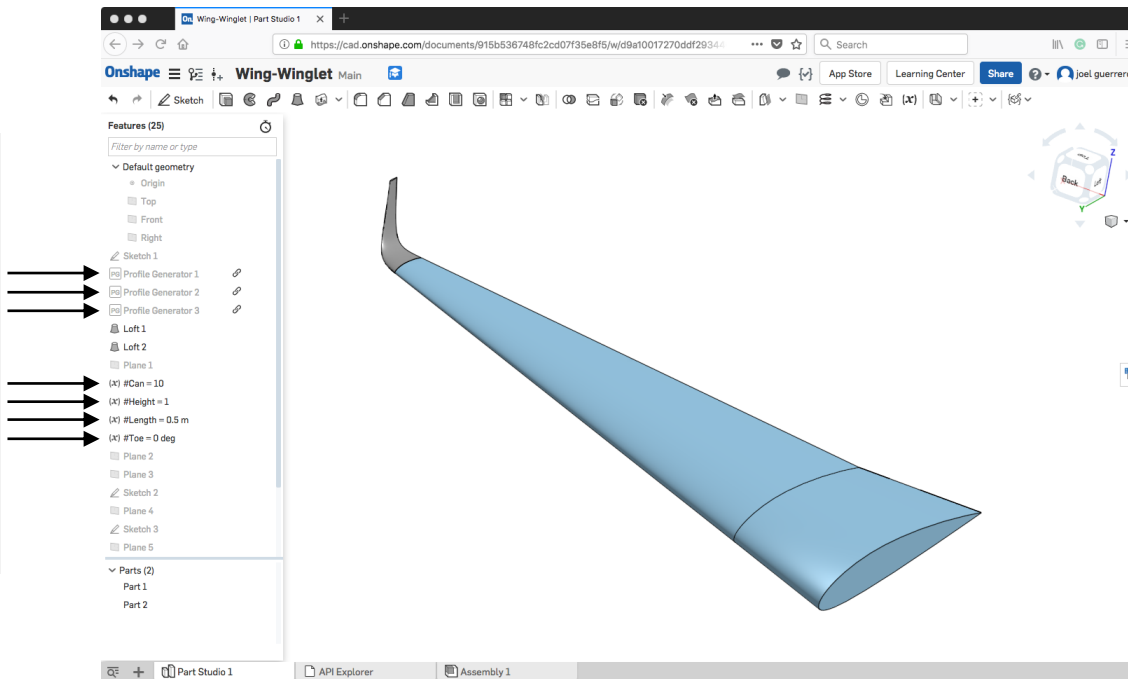
- Parameter-based optimization, works at the CAD level.
 - Gives the designer incredible level of control over the geometry.
 - A couple of parametrical variables are enough to make significant and well controlled changes in the final geometry.
 - Changes can be introduced easily.
 - The final geometry is ready to use for manufacturing or production.
 - The main difficulty is making the CAD application interact with the optimization loop.
 - It is usually used with gradient-based and derivative-free methods (local and global).
 - It is a very mature method and widely used in industry.

Introduction to optimization methods

Shape optimization methods in CFD – Parameter-based optimization

- Parameter-based optimization is usually used with gradient-based and derivative-free optimization methods.
- It can be used with multi-objective and multi-disciplinary optimization.

Parametric variables



Introduction to optimization methods

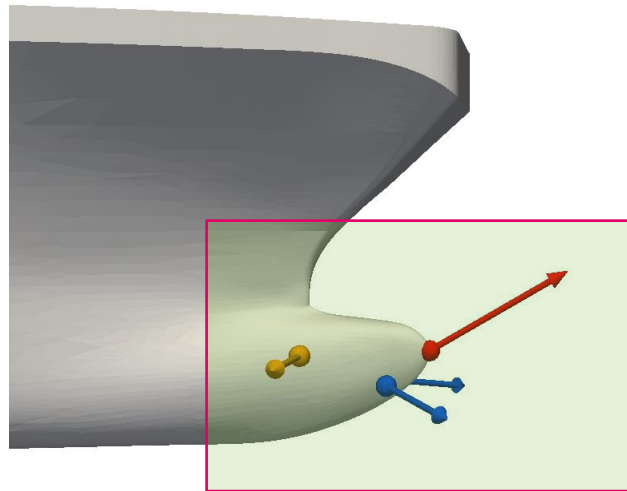
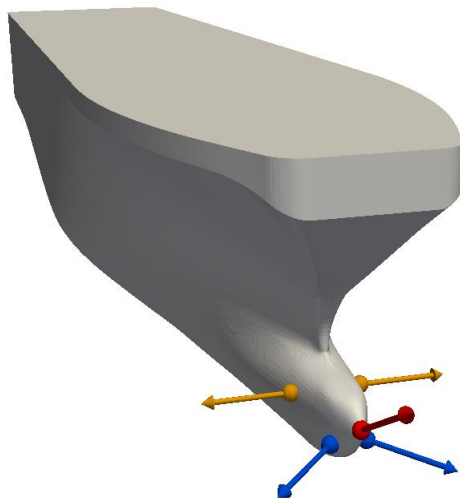
Shape optimization methods in CFD – Parameter-free optimization

- Parameter-free optimization, works at the surface mesh level or volume mesh level.
 - As is not based on parametrical variables gives a lot flexibility when deforming the geometry.
 - However, the flexibility gained does not necessarily means that the designer has extensive control on the mesh deformation.
 - It requires the selection of many control points or lattice boxes with many control points in order to define deformations.
 - As it can used at the mesh level, it does not require remeshing, reducing in this way the simulation time.
 - But for large mesh deformation, the quality of the mesh can be compromised.
 - It is usually used with adjoint methods.
 - The adjoint method has been proved a very efficient way to compute the sensitivities, but they are not easy to use.
 - A lot of R&D is being done and it has been used with success in very specific industries.

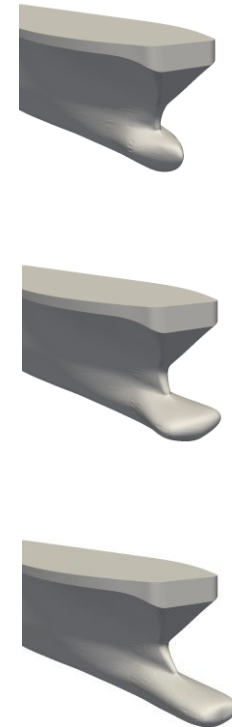
Introduction to optimization methods

Shape optimization methods in CFD – Parameter-free optimization

- Parameter-free optimization is usually used with adjoint optimization methods.
- Difficult to use with multi-objective and multi-disciplinary optimization.
- It is not easy to control, and it can generate unrealistic geometries.



Control points and control box selection



Introduction to optimization methods

Comparison matrix of shape optimization methods in CFD

Optimization approach		Pros	Cons	Design stage
• Parameter-free	<ul style="list-style-type: none"> • Topology optimization. • Surface optimization. • Often used with adjoint based methods. 	<ul style="list-style-type: none"> • Can generate innovative and unconventional designs. • Fast, it requires one solver run or a few design iterations. • Gives a lot insight on where to modify the geometry. 	<ul style="list-style-type: none"> • Topology optimization (volume based): <ul style="list-style-type: none"> • Requires reverse engineering the results into CAD. • Used only for internal flows. • Adjoint optimization (surface based): <ul style="list-style-type: none"> • Confined to small changes, unless used in an iterative way. • Can be difficult to interpret the results. • Limited to the implemented QOIs. • Difficult to use in multi-objective and multi-disciplinary optimization. 	<ul style="list-style-type: none"> • Topology optimization can be used from Initial design to fine tuning. • Adjoint optimization is preferably used for fine tuning the design.
	<ul style="list-style-type: none"> • Fully parametric CAD. 	<ul style="list-style-type: none"> • CAD geometry of high quality. • Can be used with gradient-based and derivative-free methods (local and global). • Applicable to multi-objective and multi-disciplinary optimization. • Gives a lot of insight, the designer can determine which parametric variables have higher correlations. 	<ul style="list-style-type: none"> • It requires many solver runs in order to compute the sensitivities. • Confined to the design space of the parametric model. • Requires a parametric CAD model. 	<ul style="list-style-type: none"> • Initial design to fine tuning.
• Parametric-based				

Introduction to optimization methods

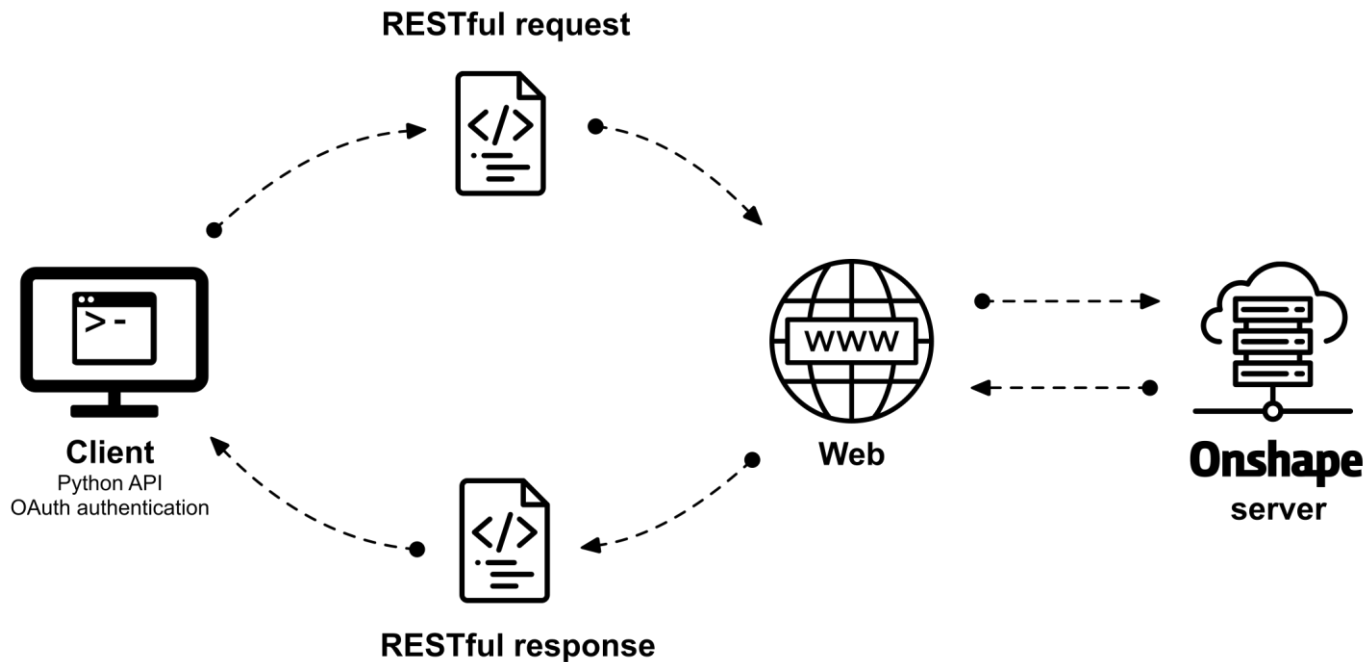
Parameter-based optimization on the cloud

- Using a feature-based, fully parametric CAD application gives the designer incredible level of control over the solid model.
- But the problem with most CAD apps, is that they do not work in Linux and they do not take input parameters using scripting language.
- In general, they are not easy to introduce in an optimization loop.
- To overcome this problem, we can use Onshape (www.onshape.com).
 - Full cloud based professional 3D CAD system.
 - Fully collaborative and simultaneous real time editing.
 - It runs on any device with a working web browser.
 - Academic and public versions → Free.
 - Professional version → Monthly/annual subscription.
 - All versions share same capabilities.
 - RESTful API, so it can be scripted using python or nodeJS.

Introduction to optimization methods

Parameter-based optimization on the cloud

- By using Onshape RESTful API, we are able to close our optimization loop using a fully parametric CAD system.



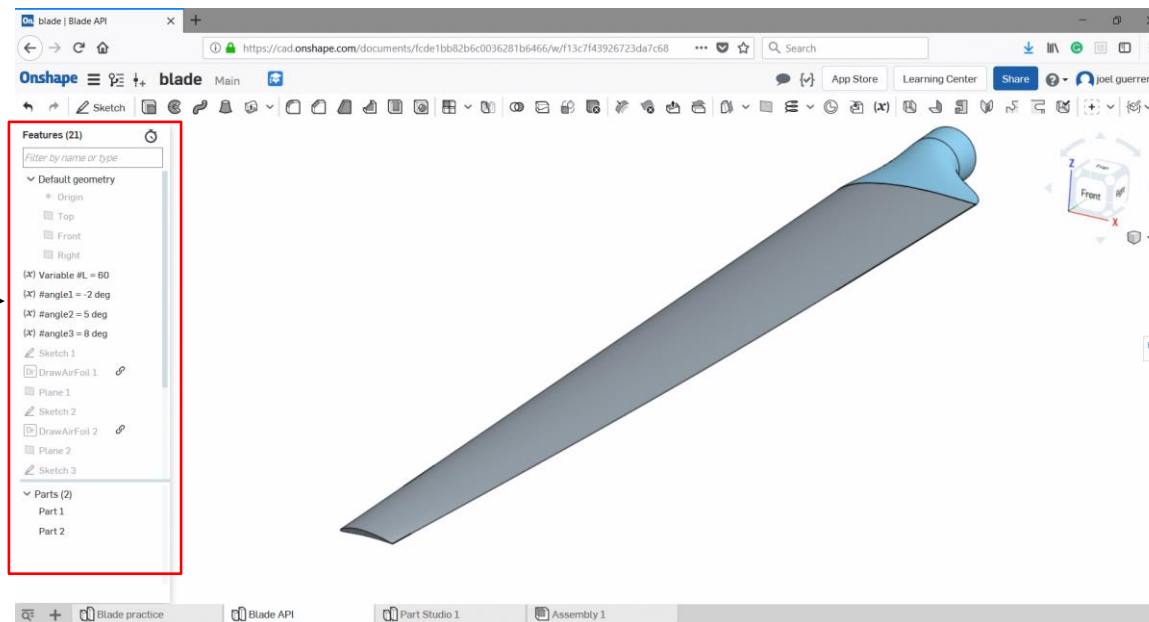
Introduction to optimization methods

Parameter-based optimization on the cloud

- RESTful requests → POST, GET, PUT, DELETE
- Request a feature/document update or change.
- Get the request response.
- Download the new solid model in STL format or any CAD exchange format.



Any feature in the tree can be modified using Onshape RESTful API



Oauth authentication

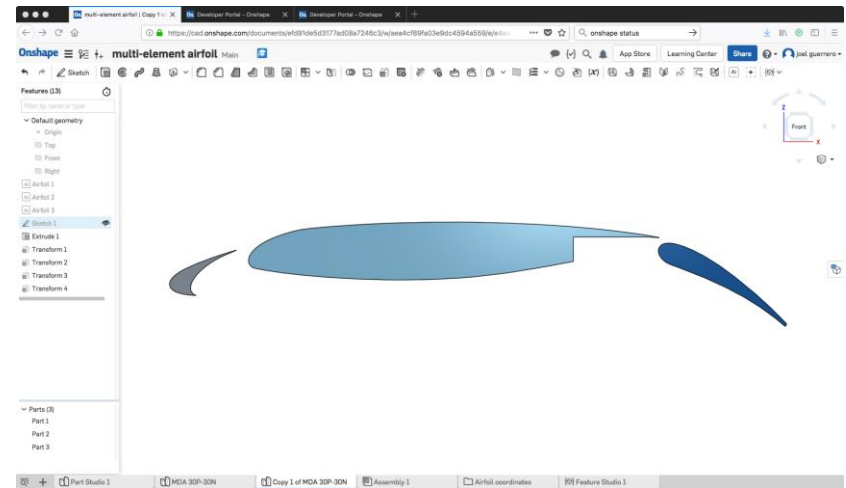
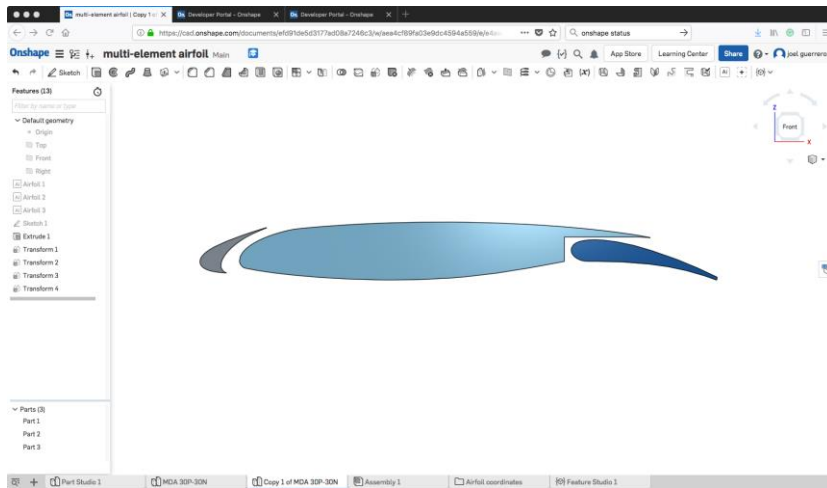
{ RESTful Request }

{ RESTful Response }

Introduction to optimization methods

Parameter-based optimization on the cloud

- By using a fully parametric CAD, things such as this high-lift wing can be easily parametrized.
- Doing such modifications using mesh morphing is not that easy and robust.
- But if you are still interested in working at the mesh level, a workaround can be the use of overset meshes.



More advanced optimization methods used in CFD
The adjoint method

Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- Gradient-based methods and derivative-free methods find the optimal value by computing the sensitivity of the QOI in response to changes in each design variable.
- By sensitivity of the solution, we mean the response of the QOI to changes of the inputs (design variables).
- For problems where computing the QOI is expensive or when we have a large design vector, this might not be the best way to compute the sensitivities.
- Remember the curse of dimensionality:

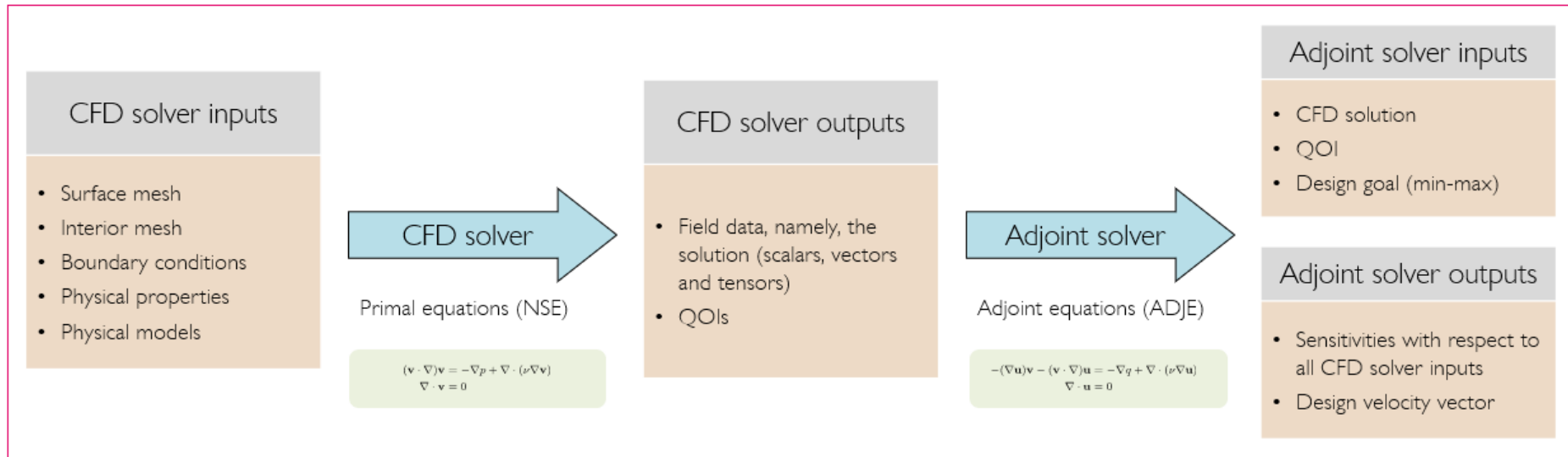
“ The higher the number of design variables in a modelling problem, the more objective function measuring locations we need if we are to build a reasonably accurate predictor. ”

- A way to circumvent the curse of dimensionality is by computing the gradients (or derivatives) using the adjoint method.

Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- In the framework of CFD, an adjoint solver takes a flow solution and calculate the sensitivity of the QOI in response to all the inputs of the system, simultaneously, in a single computation.
- The QOI can be a measure of the system performance such as the lift or drag of a body, heat flux from a surface, or the total pressure drop through a system.

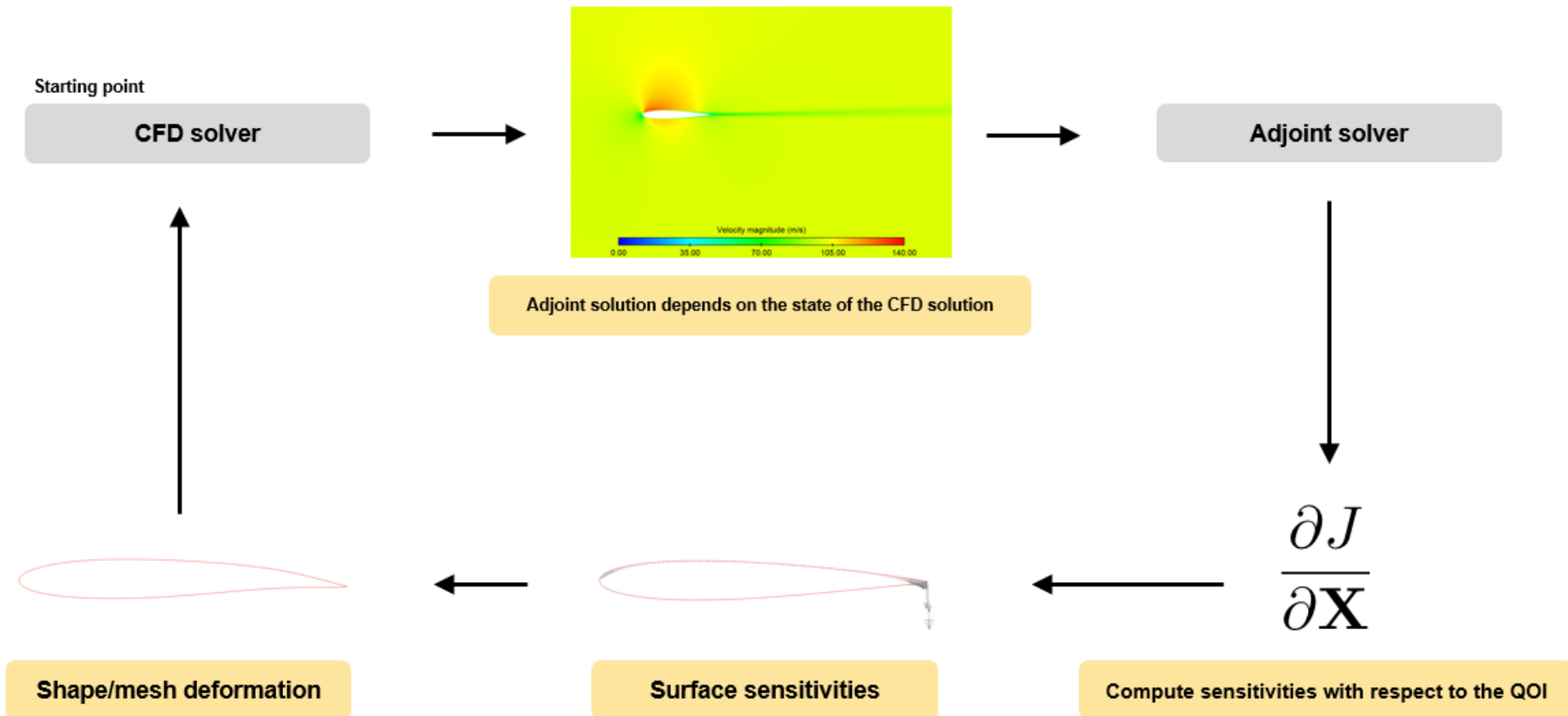


- We can iterate manually or using gradient-based methods.
- Many design iterations might be required to reach a good design or what we are aiming for.

Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

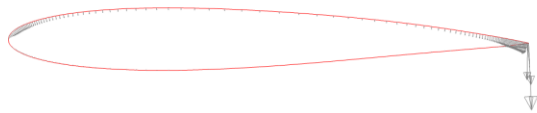
- A typical optimization loop using the adjoint method.
- Hereafter, an airfoil shape optimization design loop is illustrated.



Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- A typical optimization loop using the adjoint method.
- Hereafter, an airfoil shape optimization design loop is illustrated.
- We aimed at a percentage change of 5% between design iterations.



Iteration 10

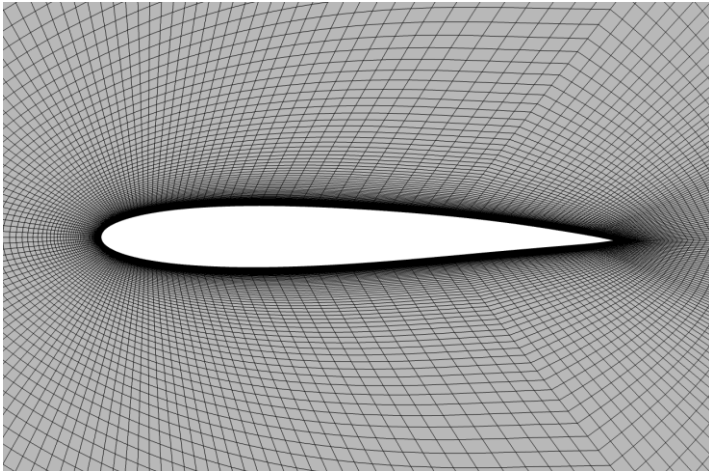


Iteration 20

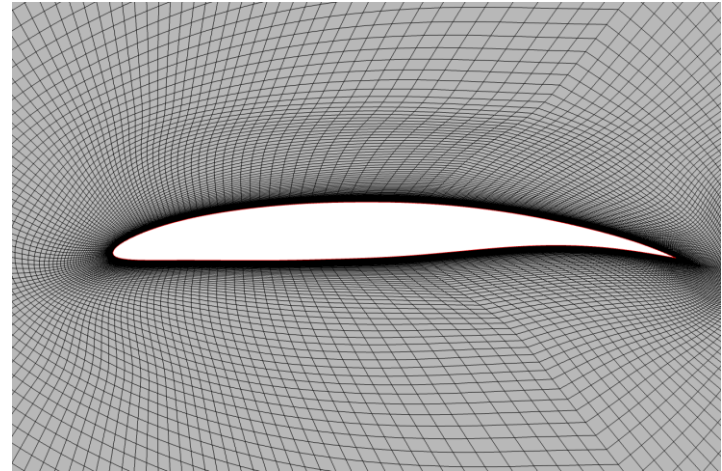
Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- A typical optimization loop using the adjoint method.
- Hereafter, an airfoil shape optimization design loop is illustrated.
- We aimed at a percentage change of 5% between design iterations.



Iteration 10

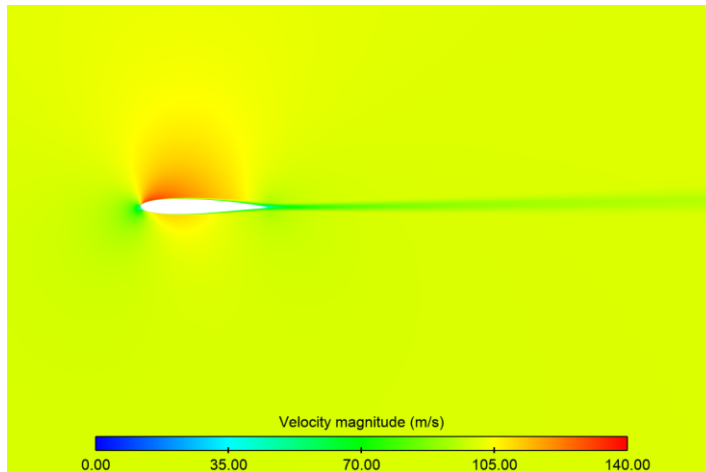


Iteration 20

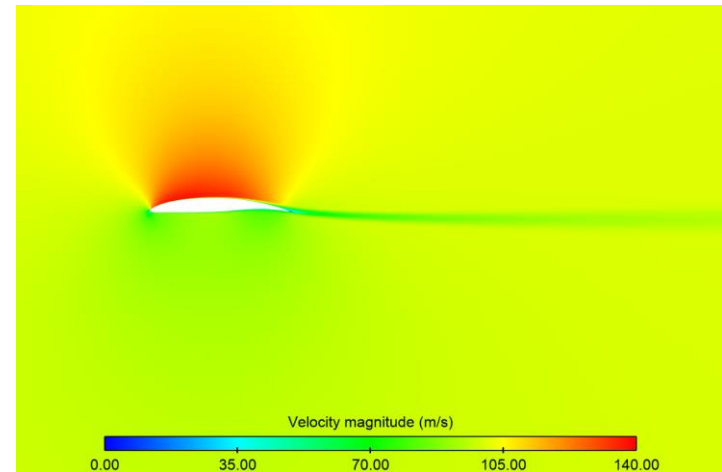
Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- A typical optimization loop using the adjoint method.
- Hereafter, an airfoil shape optimization design loop is illustrated.
- We aimed at a percentage change of 5% between design iterations.



Iteration 10

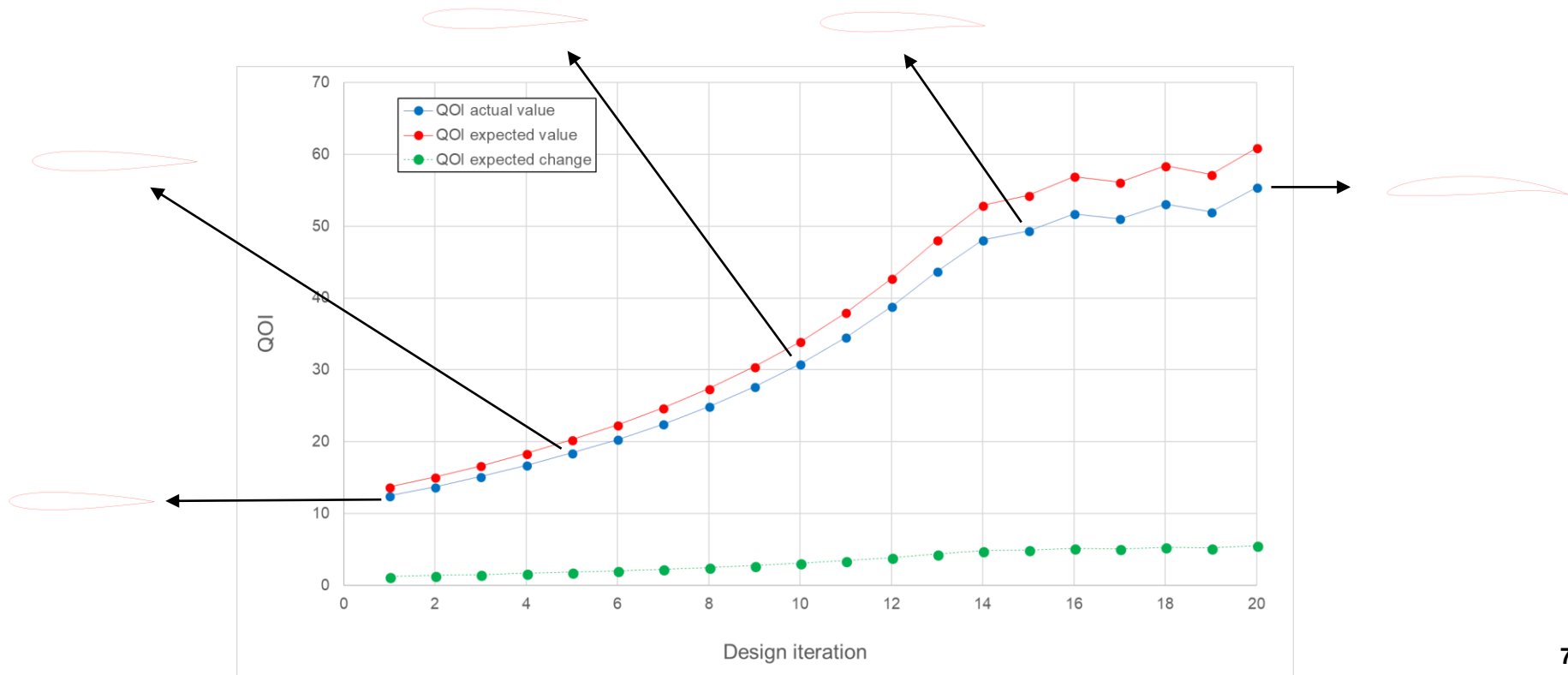


Iteration 20

Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- A typical optimization loop using the adjoint method.
- Hereafter, an airfoil shape optimization design loop is illustrated.
- We aimed at a percentage change of 5% between design iterations.



Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- Adjoint solvers can be:
 - Continuous solvers
 - Discrete solvers
 - Automatic differentiated solvers (algorithmic differentiation)
- Each solver type has different implementation details, limitations, and pros/cons, but at the end of the day, they all find derivatives with respect to the shape of the body or flow path, allowing the sensitivities to be evaluated.
- Obtaining an adjoint solution (sensitivities, derivatives, gradients, you name it) gives designers key information on how to modify the shape of the body.
- An adjoint solution can be used to estimate the effect of a change prior to actually making the change.
- To get an adjoint solution, about the same computational resources as for the flow solution are required.
- Some adjoint solvers can be memory eager.

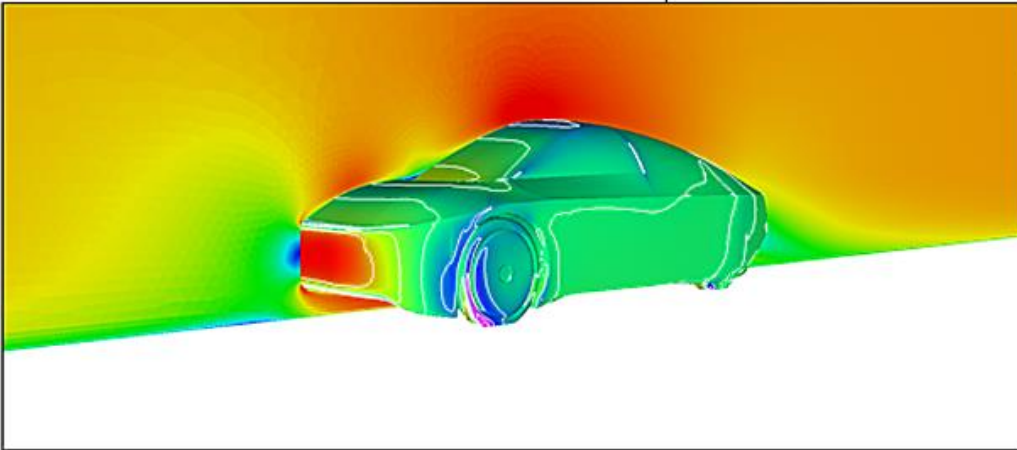
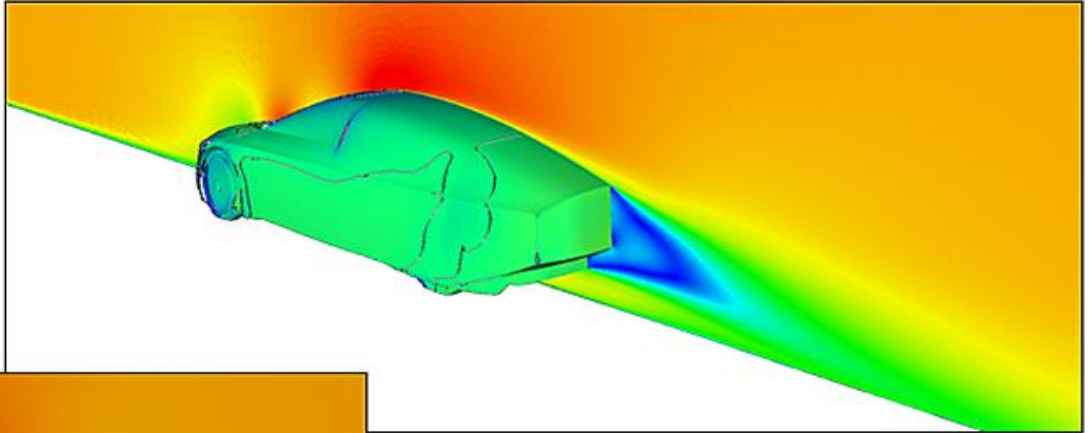
Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- Adjoint solvers are closely related to mesh morphing and free-form deformation techniques.
- Shape sensitivity data can be combined with mesh morphing to guide smooth mesh deformations.
- Mesh morphing and free-form deformation are powerful tools that allows designers to alter the geometry at the mesh level to evaluate effects of the design alterations.
- Designers can then iterate to achieve an optimum design without the need to return to the original CAD geometry.
- When using mesh deformation and free-form deformation, we lose the CAD parametrization, and we might get very unrealistic designs (that still are optimal) but are not feasible due to operational or manufacturing requirements.
- For complex geometries, the insight gained by adjoint solvers can take the design in unexpected directions.

Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

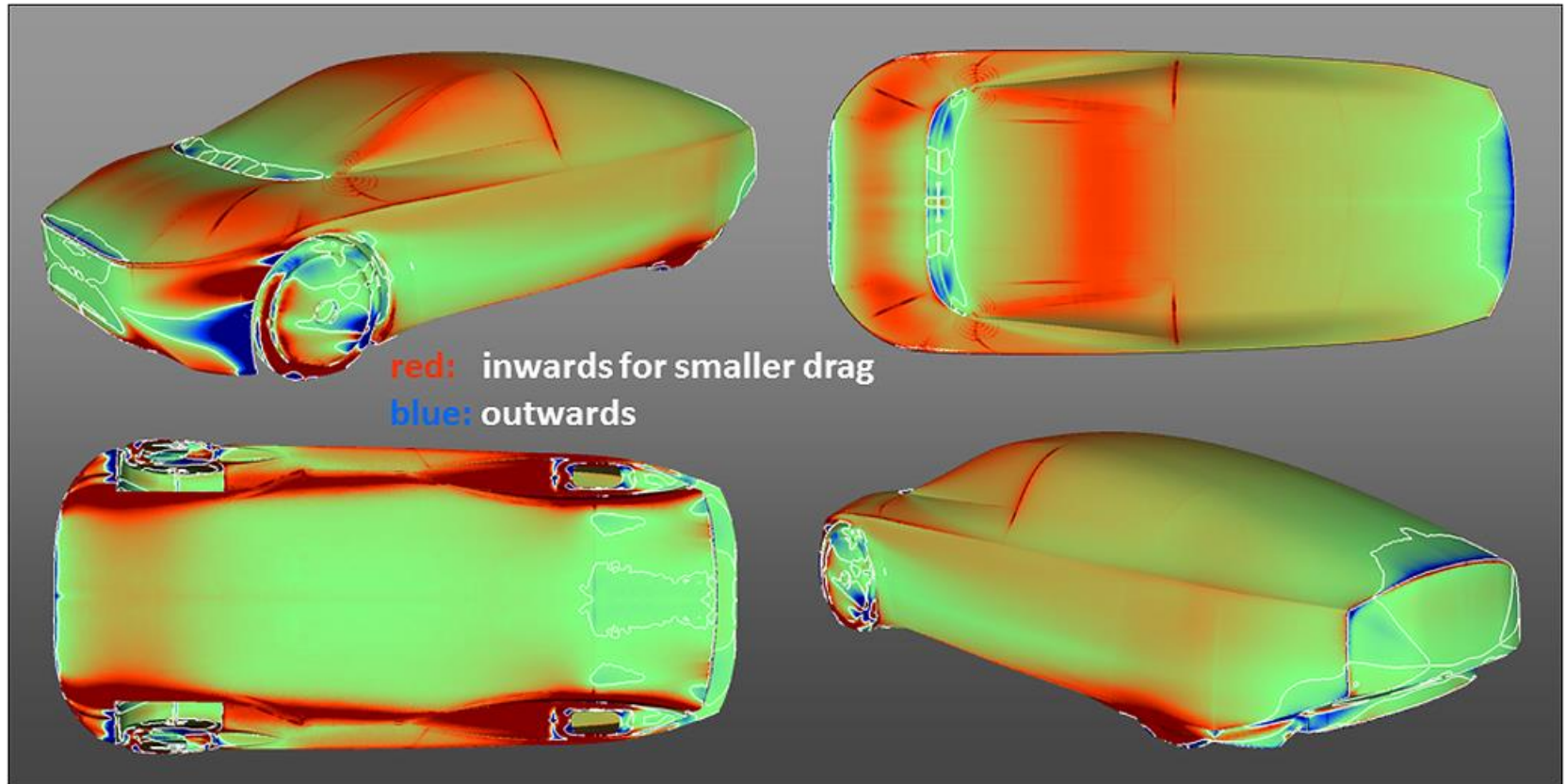


- $v=33\text{m/s}$
- stationary floor (slip up to 10m upstream)
- RANS with Spalart-Allmaras
- low-Reynolds mesh ($y^+ \sim 1$)
- half-model

The primal simulation for the Volkswagen XL1. The symmetry plane is colored according to the velocity magnitude, while the car surface shows the pressure distribution.

Introduction to optimization methods

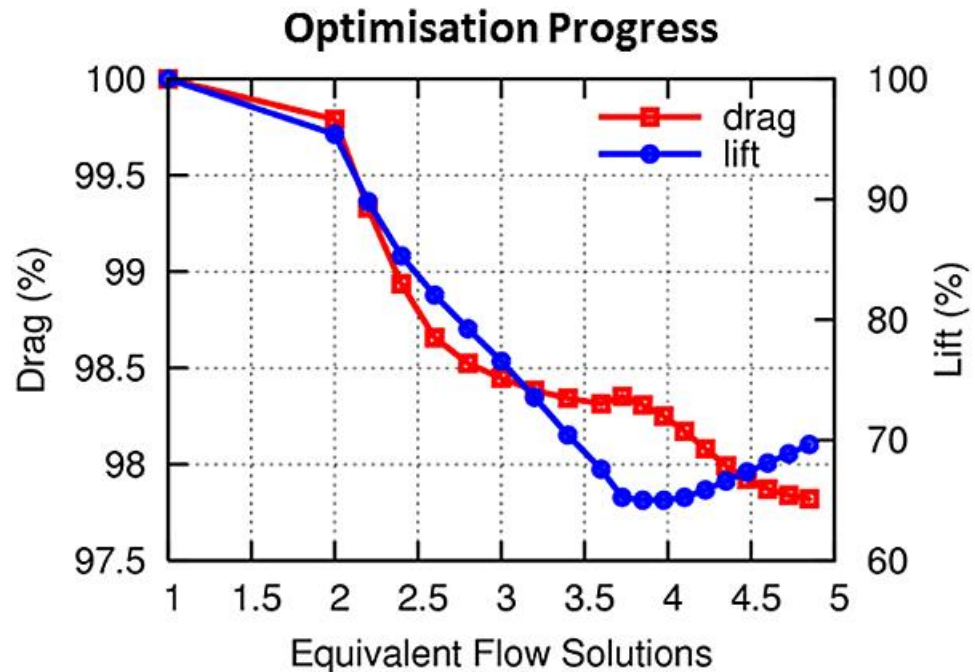
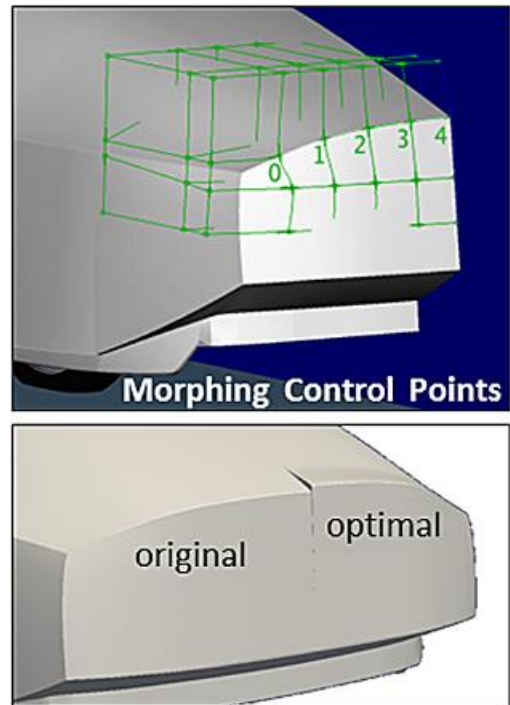
More advanced optimization methods used in CFD – The adjoint method



Drag sensitivity maps for the XL1. Isometric front and back view, bottom view (bottom left) and top view (top right).

Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method



One-shot optimization of the XL1 spoiler shape.

The rear edge of the XL1 half-model is parameterized with five morphing control points (top left). After twenty steepest descent driven shape updates in one-shot fashion, drag reduced by 2% and lift by 30% at a total cost of five equivalent flow solutions (right). The bottom left figure compares the original and the optimal shape of the rear car edge.

Introduction to optimization methods

More advanced optimization methods used in CFD – The adjoint method

- The adjoint method is a very powerful technique for shape optimization in CFD.
- However, is not an entry level method. It requires a well prepare user.
- It is recommended to use this method for fine tuning and not during the initial stage of product development.
- Have in mind that it can give you very unrealistic shapes that still are optimal.
- We will not address the adjoint method during this training.
- But if you are interested in using it in OpenFOAM, there are a few open-source implementations available. Just no name a couple:
 - Discrete adjoint:
 - <https://github.com/mdolab/dafoam>
 - Algorithmic differentiation:
 - <https://www.stce.rwth-aachen.de/research/software/discreteadjointopenfoam>
 - Continuous adjoint
 - <https://www.openfoam.com/releases/openfoam-v2006/numerics.php>

**Numerical simulations, product development,
and the need of optimization**

Introduction to optimization methods

Numerical simulations, product development, and the need of optimization

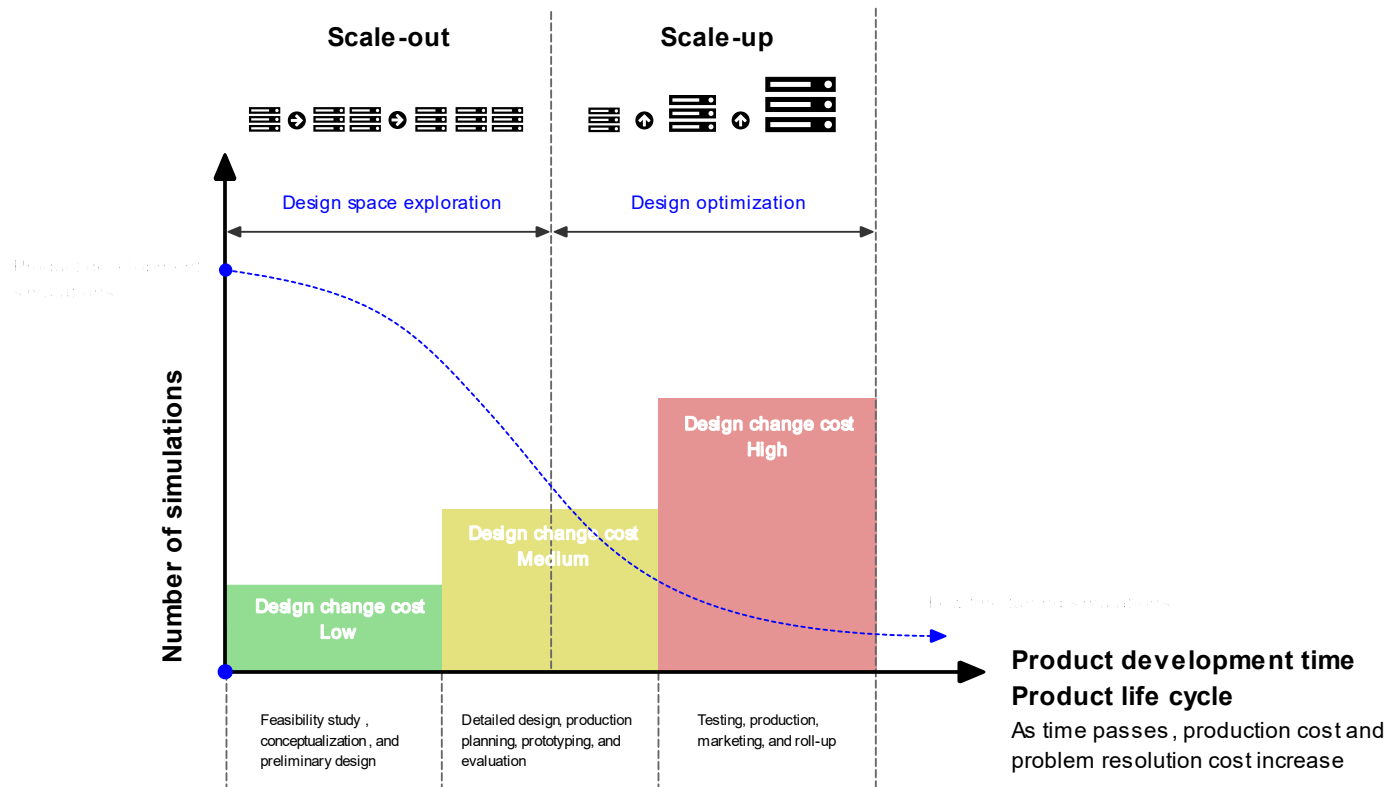
“ During the life cycle of a product, and to improve its performance and reduce production and operational costs, industry is relying more and more in numerical simulations, numerical optimization, exploratory data analysis* and business intelligence**. ”

* Exploratory data analysis (EDA) is an approach to analyzing data sets to summarize their main characteristics, often with visual methods.

** Business intelligence (BI) comprises the strategies and technologies used by enterprises for the data analysis of business information. BI technologies provide historical, current and predictive views of business operations.

Introduction to optimization methods

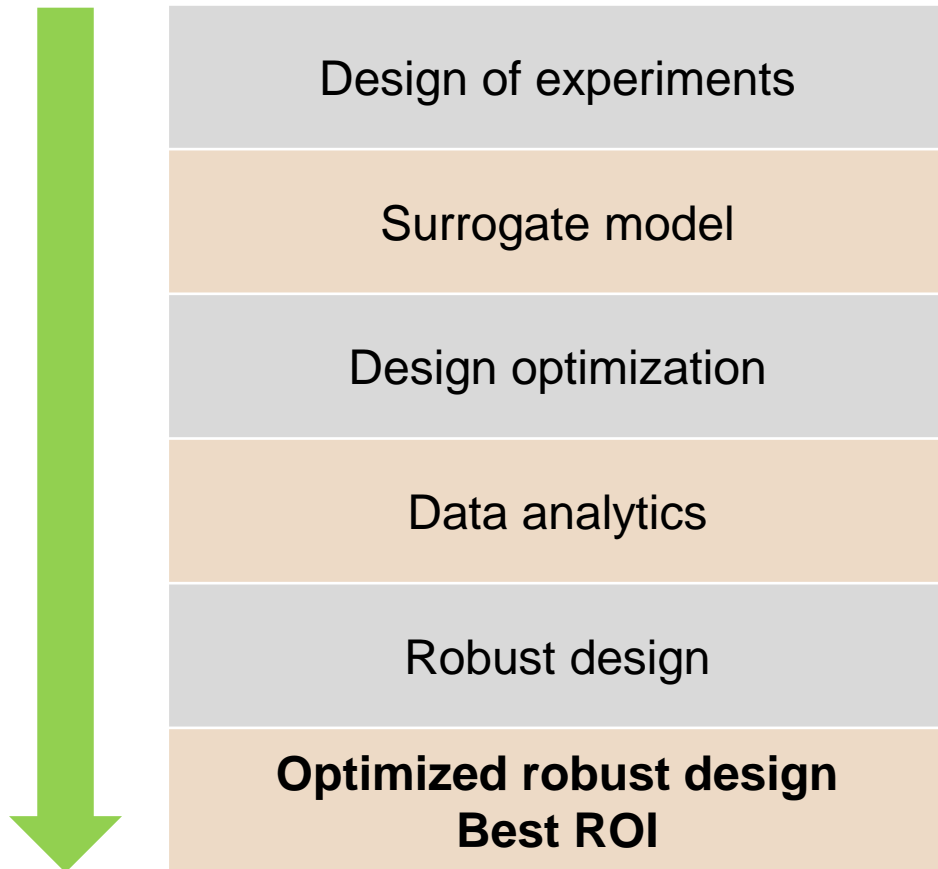
Numerical simulations, product development, and the need of optimization



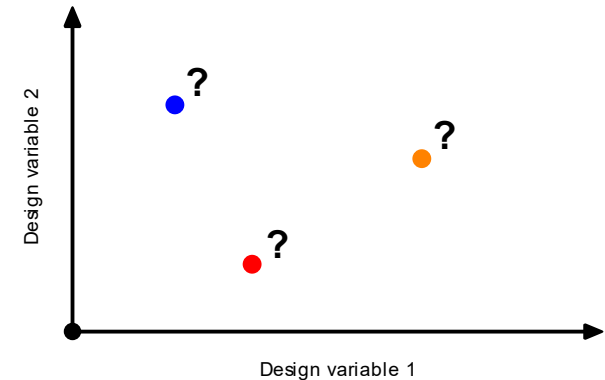
- To design innovative products in a cost-effective way, the industry is relying more and more on numerical simulations.
- The benefits of simulating far outweighs the costs related to physical experiments and constructing prototypes.
- Simulate early, simulate often. Get it right the first time.
- Plan your optimization study and exploit your computational resources in an efficient way.

Introduction to optimization methods

Path to optimized design, robust design, and best ROI in product development



Right approach in product development



Single/random simulations



What if study
We do not want to do this

Trial and error. This approach is not recommended in product development

Introduction to optimization methods

And in case you did not know it

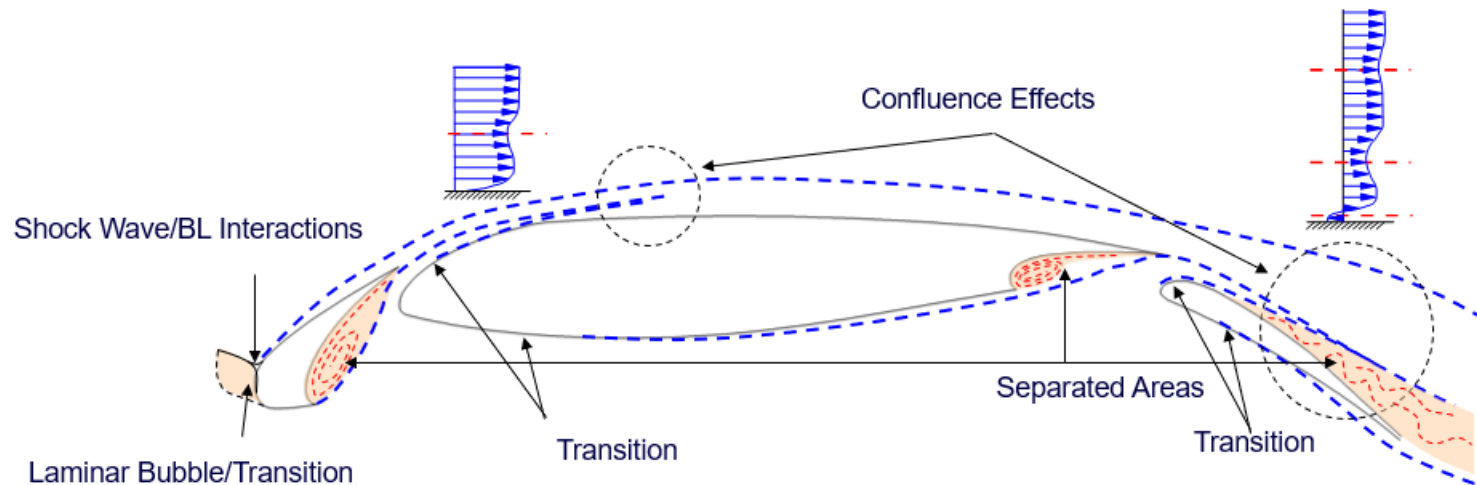
- Robust design optimizes design variables to achieve a particular probabilistic level, such as Six Sigma, which translates into 3.4 defective features in one million opportunities.
- Robust design takes into account the variation of input parameters and seeks a design with a probabilistic goal.
- In order to arrive to a robust design, we need to specify probabilistic parameters and use probabilistic optimizations algorithms.
- Guess what?, the probabilistic input is obtained from a design exploration study.

Planning your optimization study

Introduction to optimization methods

Planning your optimization study

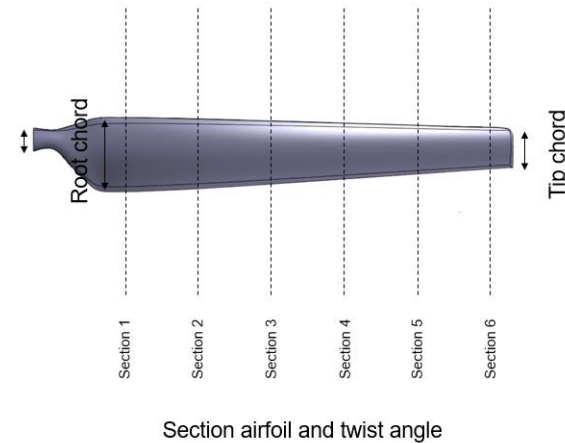
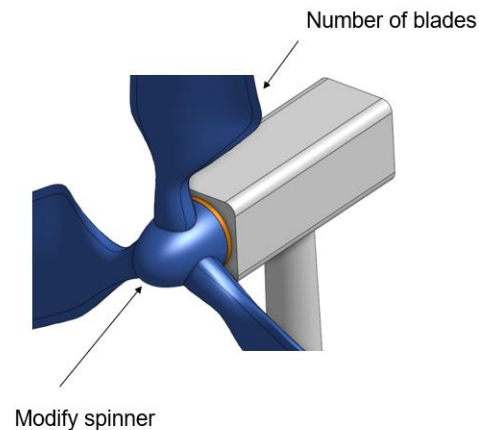
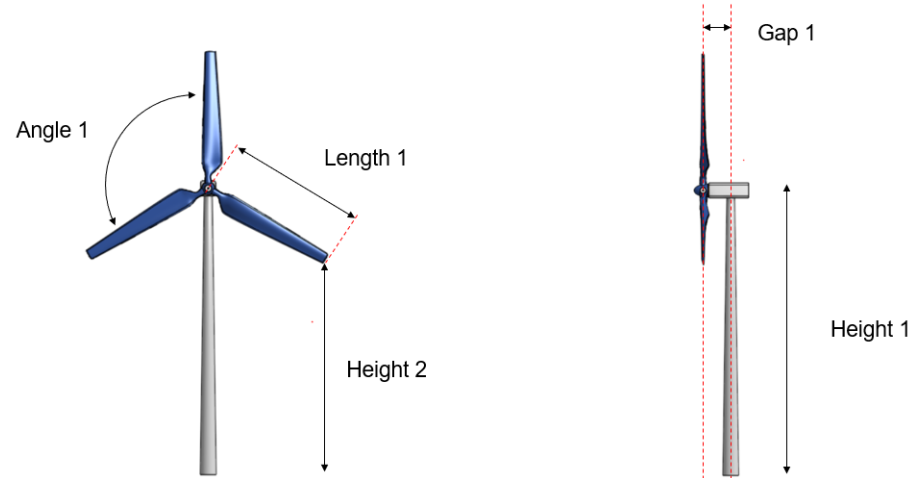
- Conventional high-lift configuration of a civil aircraft.
- What do we want to do?
 - Are we interested in studying the influence of slats/flaps in the configuration? If this is the case, we should conduct a design space exploration study with a full parametric CAD.
 - Are we interested in controlling the flow to avoid flow separation/shock waves? If this is the case free-form mesh deformation is the way to go.



Introduction to optimization methods

Planning your optimization study

- Wind turbine configuration.
- What do we want to do?



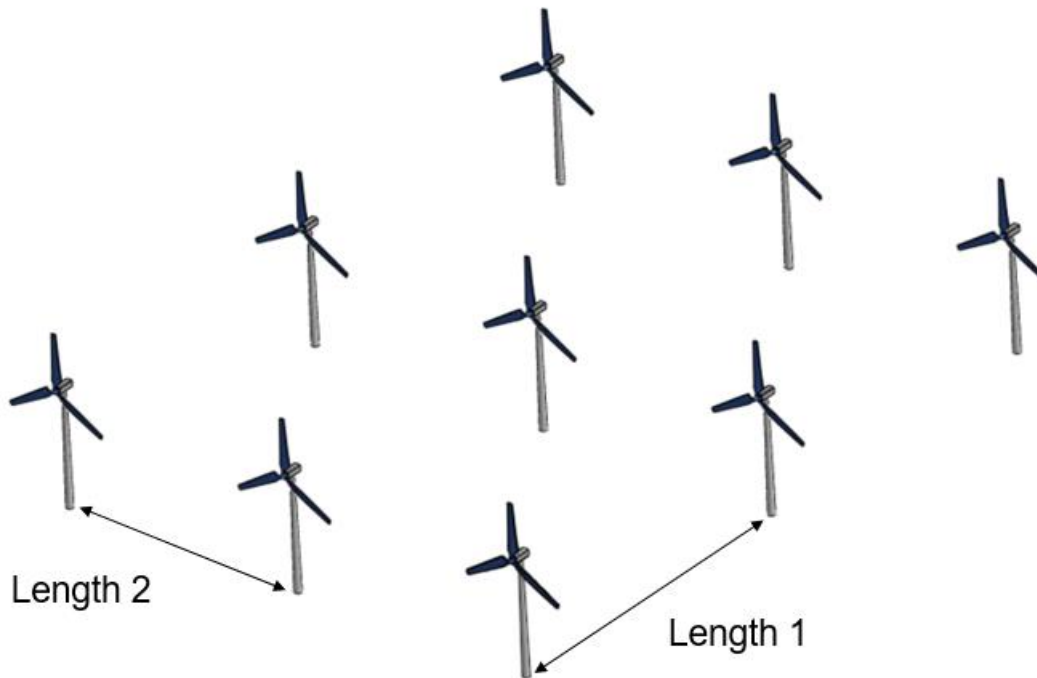
Introduction to optimization methods

Planning your optimization study

- Wind turbine configuration.
- What do we want to do?

BCs:

- RPM
- Wind velocity
- Cross flow



Final remarks

Introduction to optimization methods

Final remarks

- Can optimization methods guarantee the existence and uniqueness of a global (or a local) optimum solution?
 - **The short answer is no.**
- Optimization is very subjective.
- So, for a designer or engineer an optimal value found by the optimizer might not be a practical solution.
- It is also not uncommon to have multiple solutions, as in the case of multi-objective optimization.
- And many times, we need to optimize concepts a little bit more abstract, such as, customer satisfaction, manufacturing process, packing factor, form factor, idle time, cost reduction. And formulating these kind of problems is not very easy.
- Also, ill-conditioned problems can result in poor convergence.

Introduction to optimization methods

Final remarks

- And to make matters worst, in engineering design we often deal with multi-disciplinary optimization (MDO).
 - For example, improving the aerodynamic performance of an airplane (aerodynamics group), can result in larger weight of the airplane due to larger aerodynamic loads (structural group), and this might reduce the payload and the handling qualities of the airplane (flight performance group), and therefore the revenues (sales department).
- At the end of the day, the current design iteration should be an improvement of the previous iterations.
- And we find a better solution by using any of the methods we just described.
- We aim at using a sound approach instead of the what-if approach (guessing).

Introduction to optimization methods

Final remarks

- In engineering design, usually the optimization cycle is conducted until,
 - We run out of money.
 - We run out of time.
 - We say it is enough, as the current solution is better than the starting point and we stop the optimization cycle.
- Finally, have in mind that,
 - Optimization is a slow converging, meticulous and thoughtful process that requires careful planning, fault tolerant loops, and real-time data monitoring and analysis.
 - Do not expect fast outcomes leading to miraculous solutions.

Introduction to optimization methods

Final remarks

- Nowadays as the competition is getting stronger and stronger the need for robust designs is bigger than ever.
- The cost of failure has never been so high, even for successful companies.
- Take the automotive industry for example, where manufacturers have been forced to recall thousands of cars to fix some problem.
- Design exploration and robust design is a great aid to increase product performance and integrity in less time.
- We can easily consider more design variants, find the optimal design and ensure performance across a wide range of conditions.
- Design exploration and robust design can be used to identify problems as the product evolve.

Introduction to optimization methods

Final remarks

Tue, May 19, 2015, 2:41pm EDT - US Markets close in 1 hr and 19 mins

AP Source: Takata air bag recalls to double in US

AP Source: Takata to declare 33.8M air bags defective; largest auto recall in US history

AP By Tom Krisher, AP Auto Writer
43 minutes ago



A billboard advertisement of Takata Corp is pictured in Tokyo September 17, 2014. REUTERS/Toru Hanai

Introduction to optimization methods

New trends in design optimization

- Digital twins.
- Internet of things (IoT).
- Machine learning, deep learning, and artificial intelligence.
- Image manipulation, image recognition, image segmentation, image similarity, object detection, and pattern recognition.
- Reduce order models (ROM).
- Collaborative frameworks. Everyone working together on the same document, real-time, on any-device, anywhere.
- Cloud computing.
- In summary, the new trends are towards,

Agile product development, real-time collaboration, automation, concurrent tasks, rapid iterations, data driven insight, interactive data analysis.

Roadmap

- ~~1. Introduction to optimization methods~~
- 2. Choosing an optimization method**
- ~~3. Optimization loop – The big picture~~
- ~~4. DAKOTA overview~~
- ~~5. Working with DAKOTA: Rosenbrock function~~
- ~~6. Working with DAKOTA: Branin function~~
- ~~7. Working with DAKOTA: Multi-objective optimization~~
- ~~8. Coupling DAKOTA and OpenFOAM: driven cavity case~~
- ~~9. Additional code coupling tutorials~~
- ~~10. Some kind of conclusion~~

Choosing an optimization method

General guidelines

- The choice of the optimization method highly depends on the problem you are trying to solve.
- Before formulating the optimization problem, ask yourself the following questions:
 - Smooth or noisy behavior?
 - Unimodal or multimodal function?
 - Single or multi objective?
 - Constrained or unconstrained?
 - Any special structure, e.g., quadratic objective, highly linearly constrained?
 - Computational expensive?
 - Resources available?
 - Local optimization or global optimization?
 - Well defined problem?
 - Variable types present (real, integer, categorical)?

Choosing an optimization method

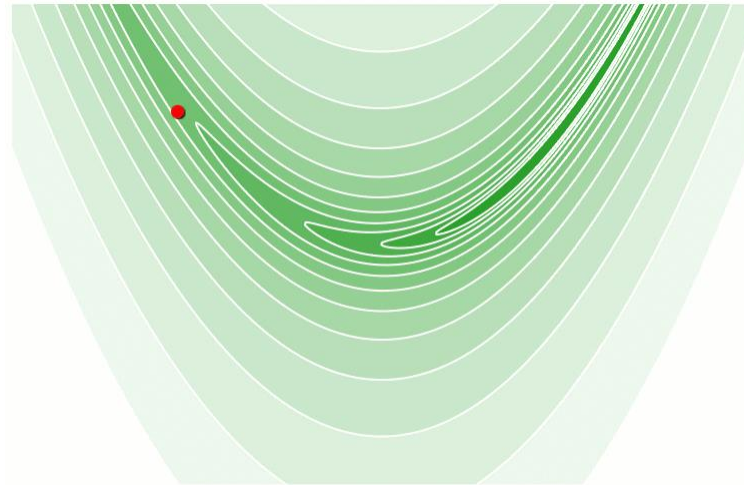
General guidelines – Decision matrix

	Unconstrained or bound-constrained problems	Nonlinearly-constrained problems
Smooth and inexpensive	Any method. Gradient-based will be the fastest	Gradient-based methods
Smooth and expensive	Gradient-based methods	Gradient-based methods
Non-smooth and inexpensive	Derivative-free methods Surrogate based optimization	Derivative-free methods Surrogate based optimization
Non-smooth and expensive	Surrogate based optimization	Surrogate based optimization
Multi-objective	Derivative-free methods Surrogate based optimization	Derivative-free methods Surrogate based optimization

Choosing an optimization method

General guidelines – Gradient based methods

- Gradient-based optimization methods are highly efficient, with the best convergence rates of all the optimization methods.
- Gradient-based optimization methods look for improvement based on derivative information.
- Gradient-based optimization methods are the clear choice when the problem is smooth, unimodal, and well-behaved.
- However, when the problem exhibits non-smooth, discontinuous, or multimodal behavior, these methods can also be the least robust since inaccurate gradients will lead to bad search directions.



Choosing an optimization method

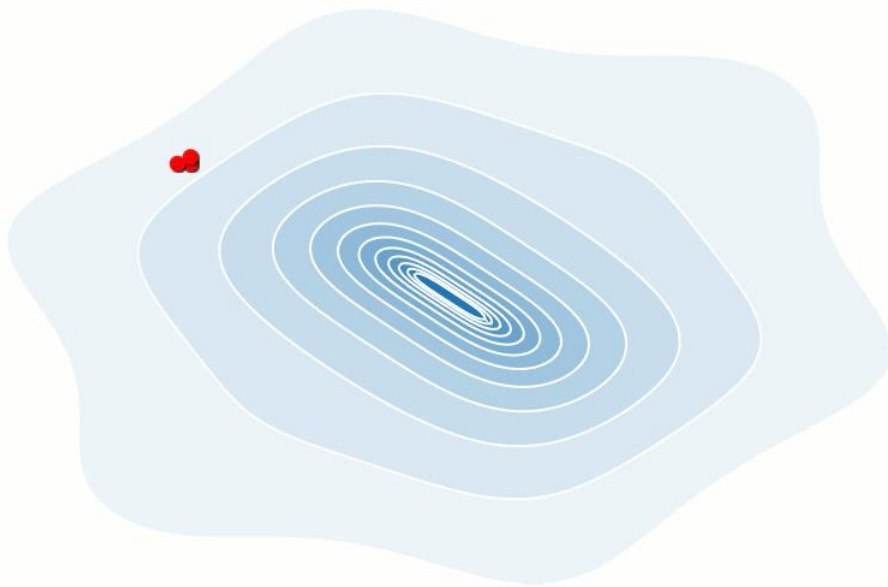
General guidelines – Derivative free methods

- Derivative-free optimization methods can be applied in situations where gradient calculations are too expensive or unreliable.
- Derivative-free based methods requires only function values (no need to compute gradients or high order derivatives).
- Derivative-free based methods can be used for local and global optimization.
- Derivative-free local methods sample the design space with bias/rules toward improvement.
- Derivative-free global methods do broad exploration of the design space with selective exploitation.
- These methods deserve consideration when the problem may be non-smooth, multimodal, poorly behaved, or the derivative are expensive to compute.
- If you are dealing with multi-objective optimization, you should use derivative-free methods.
- Derivative-free methods exhibit much slower convergence rates for finding an optimum, and as a result, tend to be much more computationally demanding than gradient-based methods.

Choosing an optimization method

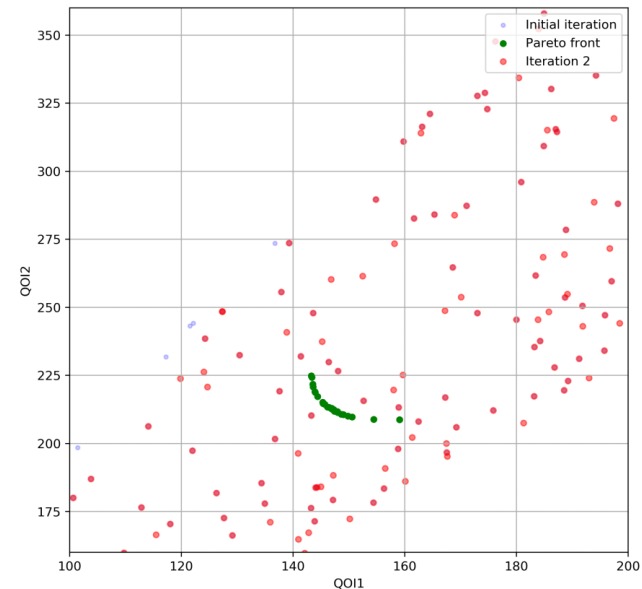
General guidelines – Derivative free methods

- Clearly, for non-gradient optimization studies, the computational cost of the function evaluation must be relatively small in order to obtain an optimal solution in a reasonable amount of time.
- Genetic algorithms, division of rectangles, pattern search, simplex method, Nelder-Mead method, particle swarm, greedy search, are examples of derivative free methods.



Nelder-Mead method

<http://www.wolfdynamics.com/training/opt/image7.gif>



MOGA – Genetic algorithm

<http://www.wolfdynamics.com/training/opt/ani5.gif>



Choosing an optimization method

General guidelines – DOE/DACE and SBO

- Parameter studies, classical design of experiments (DOE), design/analysis of computer experiments (DACE), and sampling methods share the purpose of exploring the design space.
- The distinction between DOE and DACE methods is that the former are intended for physical experiments containing an element of non-repeatability (and therefore tend to place samples at the extreme parameter vertices), whereas the latter are intended for repeatable computer experiments and are more space-filling in nature.
 - DOE for stochastic behavior (physical experiments).
 - DACE for deterministic behavior (computer simulations).
- When a global space-filling set of samples is desired, then DACE and sampling methods are recommended. These techniques are useful for scatter plot and variance analysis as well as surrogate model construction.
- DACE experiments can be used to construct a surrogate (also known as response surface or meta-model).
- Optimizing at the surrogate level is inexpensive.
- SBO is recommended for expensive experiments and in the presence of noisy data.
- SBO is a good choice for conducting multi-objective optimization.

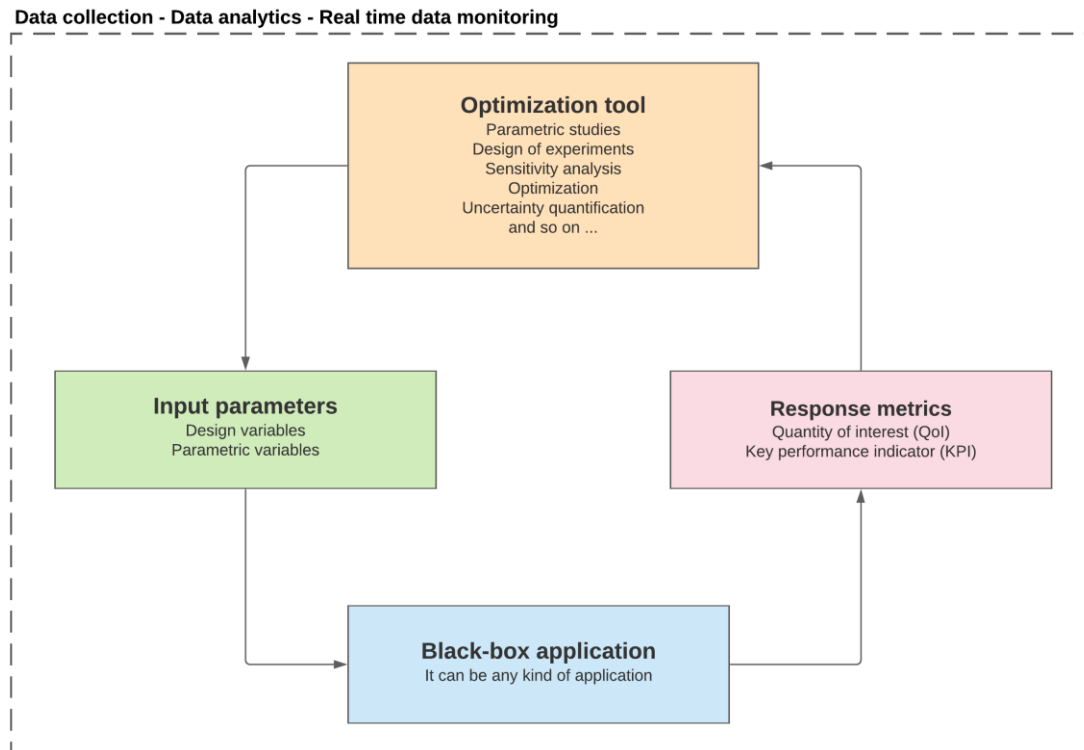
Roadmap

- ~~1. Introduction to optimization methods~~
- ~~2. Choosing an optimization method~~
- 3. Optimization loop – The big picture**
- ~~4. DAKOTA overview~~
- ~~5. Working with DAKOTA: Rosenbrock function~~
- ~~6. Working with DAKOTA: Branin function~~
- ~~7. Working with DAKOTA: Multi-objective optimization~~
- ~~8. Coupling DAKOTA and OpenFOAM: driven cavity case~~
- ~~9. Additional code coupling tutorials~~
- ~~10. Some kind of conclusion~~

Optimization loop – The big picture

General optimization loop – The big picture

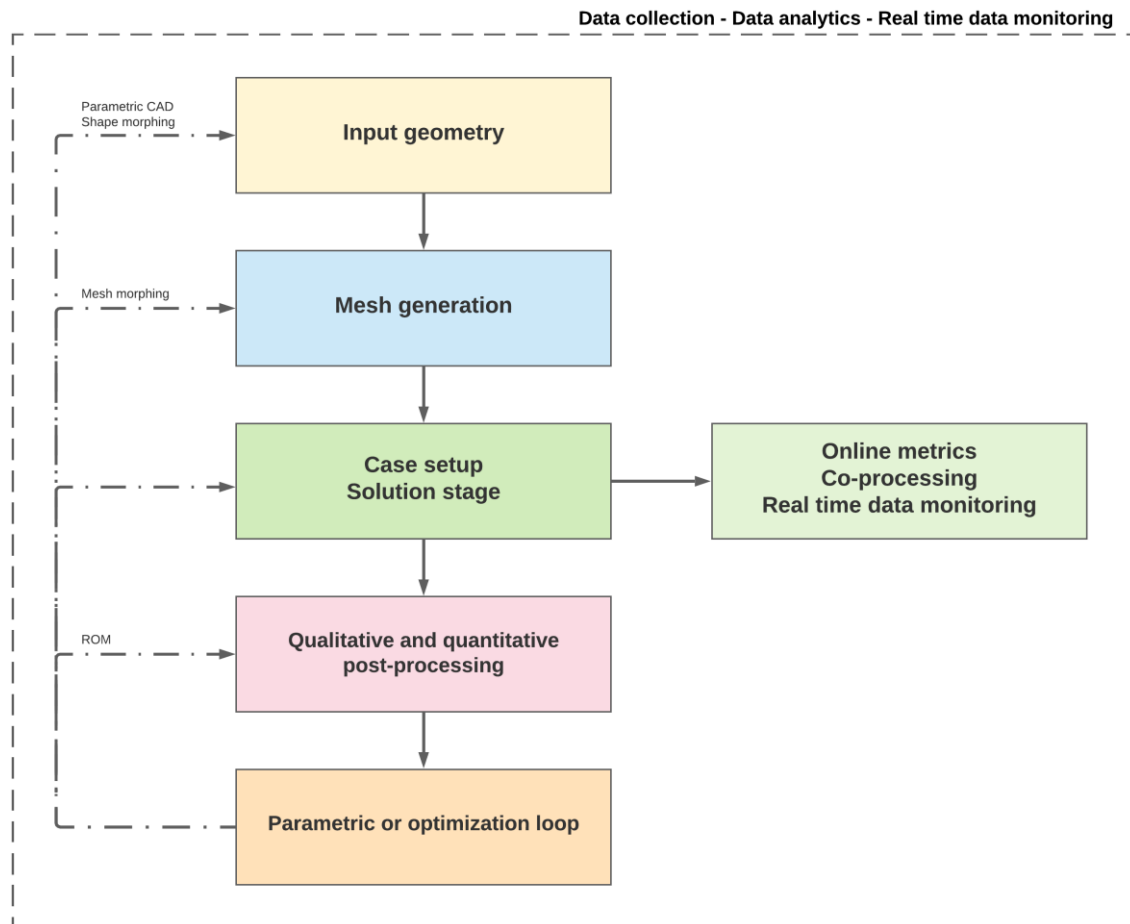
- A general optimization loop can be used in any field.
- The input parameters can be any kind of design variable (numerical, categorical, text, an image, and so on).
- The quantity of interest (QOI) can be any kind quantitative or qualitative data (numerical, categorical, text, an image, and so on).
- The black-box application can be any kind of application.
 - In a loosely coupled optimization loop, as the one that we are going to study, the black box application must be able to interact via the command line interface or using parametric input files.
 - Preferable, it also should be able to work with no graphical user interface



Optimization loop – The big picture

General optimization loop in CFD – Vertical approach

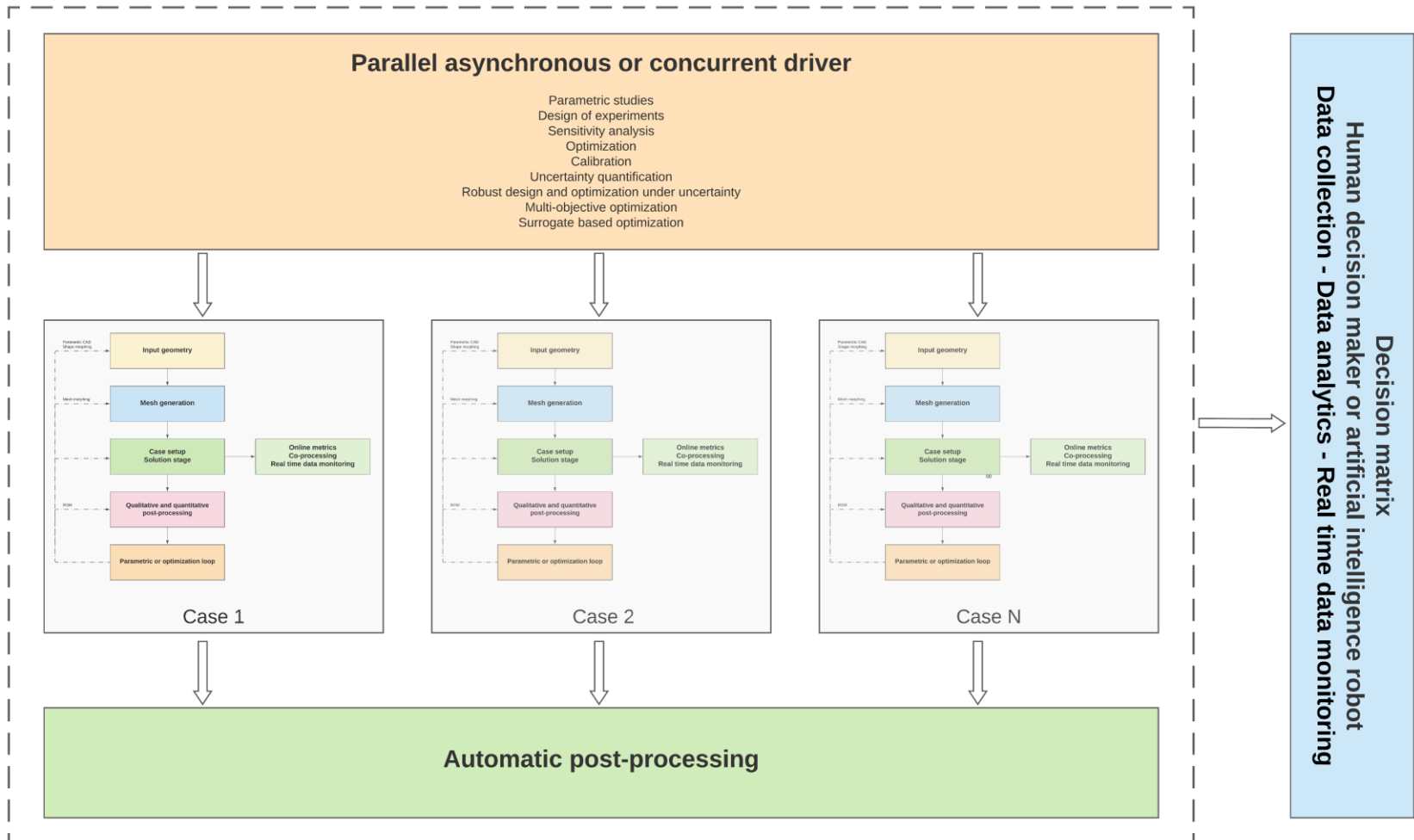
- Typical vertical approach when conducting CFD studies and launching one simulation at a time.
- In this vertical approach the design optimization loop can iterate back to any of the previous steps.
- This single case can be run in parallel.



Optimization loop – The big picture

General optimization loop in CFD – Vertical approach

- Concurrent design optimization loop for CFD studies.
- Here we use many processors to solve many problems at the same time.
- At the same time, we solve each problem using many processors.



Optimization loop – The big picture

General optimization loop in CFD – Tools in use

- From the point of view of code coupling there are two types of optimization loops:
 - **Loosely coupled approach**
 - Where all the applications interact via shell scripting or the command line interface.
 - Usually, the applications belong to different software developers and do not share common data exchange formats.
 - Not very user friendly.
 - **Strongly coupled approach**
 - where all applications interact using a single interface, graphical or command line based.
 - Often, the applications belong to the same software developer, and they share a common data exchange format.
 - User friendly (to some extension).

Optimization loop – The big picture

General optimization loop in CFD – Tools in use

- The **strongly coupled approach** is commonly found when using commercial applications, such as, Ansys Fluids, StarCCM, VisualDOC, CAESES.
- The **loosely coupled approach**, is found when using open-source applications or applications that do not belong to the same software developer or are not meant to interact with other applications.
- From the performance point of view, the **strongly coupled approach** performs better due to the tight integration.
- But this does not mean that the **loosely coupled approach** is much worse.
 - By using good practices, the user can get an acceptable performance when integrating different applications.
 - It is also more flexible, because it allows user to use applications that do not belong to the same software developer, which may be a limitation when using a GUI based **strongly coupled approach**.
- Hereafter, we will work with a **loosely coupled approach**.
 - We will propose an optimization loop using DAKOTA as optimizer and code coupling application.
 - Then, by using the code line interface and scripting, we will integrate different applications.

Optimization loop – The big picture

General optimization loop in CFD – A few open-source tools

- **Code coupling/Optimizer:**
 - DAKOTA – <https://dakota.sandia.gov/>
 - RAVEN – <https://raven.inl.gov/>
 - OpenMDAO – <https://openmdao.org/>
- **Concurrent computations scheduler:**
 - DAKOTA – <https://dakota.sandia.gov/>
 - Python
 - Bash shell and GNU parallel
- **Uncertainty quantification**
 - DAKOTA – <https://dakota.sandia.gov/>
 - OpenTURNS – <https://openturns.github.io/www/index.html>
 - UQLab – <https://www.uqlab.com/>
 - UQ Toolkit – <https://www.sandia.gov/uqtoolkit/>
 - UQ Tools – <https://uqtools.larc.nasa.gov/>
 - Uncertainty toolbox – <https://github.com/uncertainty-toolbox/uncertainty-toolbox>

Optimization loop – The big picture

General optimization loop in CFD – A few open-source tools

- **Parametric CAD:**
 - Onshape (API) – <https://onshape-public.github.io/>
 - FreeCAD – <https://www.freecadweb.org/>
 - SALOME – <https://www.salome-platform.org/>
 - OpenSCAD – <https://www.openscad.org/>
- **Meshing tools:**
 - GMSH – <https://gmsh.info/>
 - TETGEN – <http://wias-berlin.de/software/tetgen/>
 - SALOME – <https://www.salome-platform.org/>
 - pyhyp – <https://github.com/mdolab/pyhyp>
 - MeshKit – <https://sigma.mcs.anl.gov/meshkit-library/>
- **Mesh morphing tools**
 - MESQUITE – <https://trilinos.github.io/mesquite.html>
 - PyGeM – <https://mathlab.sissa.it/pygem>
 - idwarp – <https://github.com/mdolab/idwarp>
 - pygeo – <https://github.com/mdolab/pygeo>

Optimization loop – The big picture

General optimization loop in CFD – A few open-source tools

- **Black-box solver:**
 - OpenFOAM – <https://openfoam.org/>
 - SU2 – <https://su2code.github.io/>
 - CFL3D – <https://github.com/nasa/CFL3D>
 - FLUBIO – <https://flubiopetsc.github.io/flubiopetsc/>
- **Quantitative and qualitative post-processing:**
 - Python, paraview, JavaScript, VTK
- **Real time data monitoring:**
 - Python, R, JavaScript, BASH
- **Exploration and exploitation of design space:**
 - Python, R, BASH
- **Additional automation scripting:**
 - Python, BASH

Optimization loop – The big picture

General optimization loop in CFD – Automatic loop

- The automatic loop should cover the whole workflow of a CFD simulation:

Solid modeling → Meshing → Case setup → Simulations and monitoring → Post-processing

- We will illustrate an automatic loop for simple simulations with many applications interacting.
- But have in mind that the proposed framework can be easily extended to deal with complex scenarios
- And as we are dealing with CFD, the proposed framework can cover different fields in science and engineering (aerospace, automotive, HVAC, AEC, medical devices, thermal management, naval, and so on).
- For the cases that we will be presenting, the simulations are run using a pre-specified level of accuracy and iterative marching (which is not bad).
- However, by using data and metadata (data-of-data) to compute basic descriptive statistics and by leveraging a few concepts of SL/ML, the design loop can freely iterate until it reaches an acceptable level of convergence.

Optimization loop – The big picture

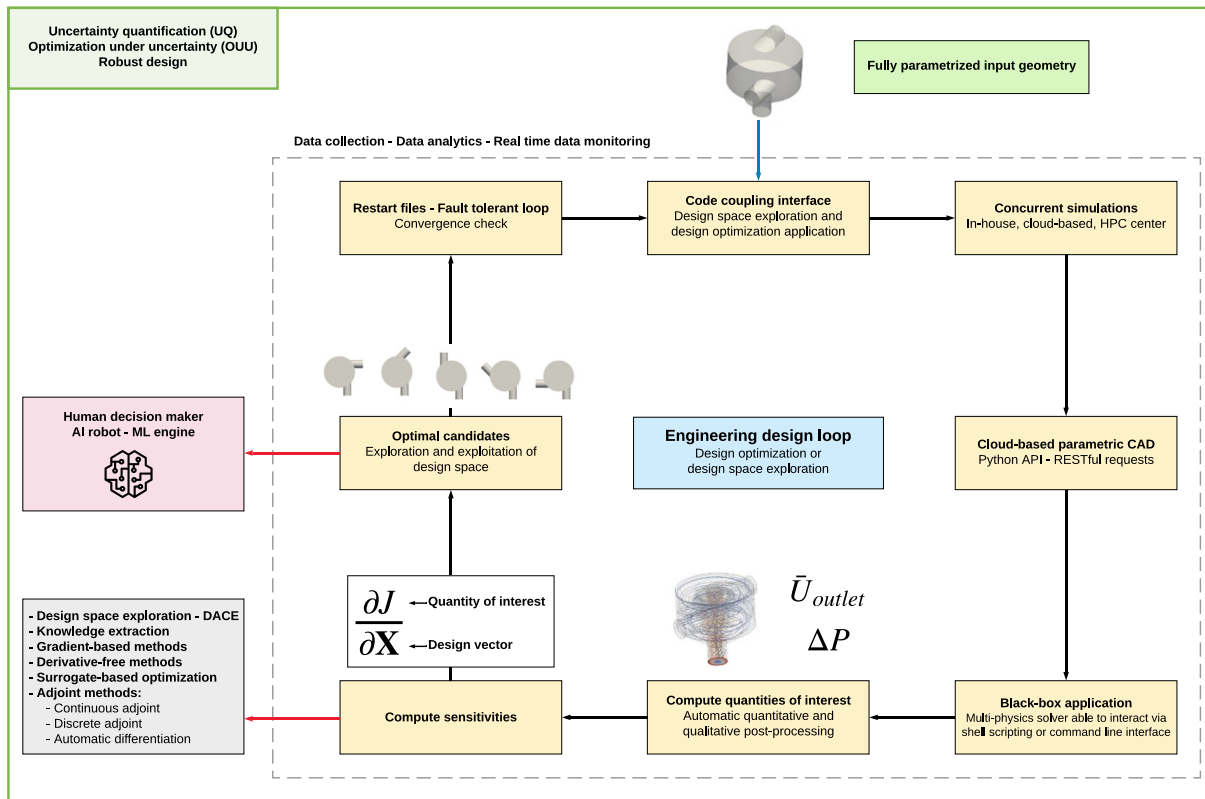
General optimization loop in CFD – Automatic loop

- A few comments on the automatic loop or optimization framework to be proposed:
 - The framework is automatic and to some extent fault tolerant.
 - But in the case of fatal failure, the user can restart from the latest stable solution.
 - In the case of anomalies while the loop is running, the input parameters can be changed on-the-fly to stabilize the solution, this can be done automatically (a lot SL/ML involved) or manually.
 - To achieve this, a lot of things need to be monitored.
 - Therefore, it is important to monitor all the QOIs and KPIs real-time.
 - Every single modification is recorded and reported to the user.
 - The bottleneck is the meshing stage.
 - In case of meshing failure or bad quality meshes, the domain is remeshed using more robust parameters (which will increase the meshing time and mesh size).
 - If the mesh issues cannot be repaired in an automatic way, the user must fix the problems manually, which is not desirable.

Optimization loop – The big picture

General optimization loop in CFD – Automatic loop

- Graphical summary of an engineering design loop using a feature-based CAD.



- Code coupling/Optimizer:**
DAKOTA
- Concurrent computations scheduler:**
DAKOTA
- Parametric CAD:**
Onshape (API)
- Black-box solver:**
OpenFOAM
- Quantitative and qualitative post-processing:**
Python, paraview, JavaScript
- Real time data monitoring:**
Python, R, BASH
- Exploration and exploitation of design space:**
Python, R, BASH
- Additional automation scripting:**
Python, BASH

Optimization loop – The big picture

Optimization loop – Automation, parametrization, and concurrency

- If you are planning to conduct optimization studies, things need to be automatic and parametric.
- It is also important to take advantage of concurrency, that is, running many simulations at the same time.
- While iterating to the optimal solution, do not forget to use real-time data analytics and exploratory data analysis to study the data gathered.
 - Sometimes there might be a hidden story in this data.
- Avoid to iterate in your optimization loop manually, it is too slow and prone to errors.
- As stated in the NASA contractor report “CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences”,

“ A single engineer/scientist must be able to conceive, create, analyze, and interpret a large ensemble of related simulations in a time-critical period (e.g., 24 hours), without individually managing each simulation, to a pre-specified level of accuracy. ”

Roadmap

- ~~1. Introduction to optimization methods~~
- ~~2. Choosing an optimization method~~
- ~~3. Optimization loop – The big picture~~
- 4. DAKOTA overview**
- ~~5. Working with DAKOTA: Rosenbrock function~~
- ~~6. Working with DAKOTA: Branin function~~
- ~~7. Working with DAKOTA: Multi-objective optimization~~
- ~~8. Coupling DAKOTA and OpenFOAM: driven cavity case~~
- ~~9. Additional code coupling tutorials~~
- ~~10. Some kind of conclusion~~

DAKOTA overview

DAKOTA in a nutshell

- DAKOTA stands for **D**esign and **A**nalysis tool**K**it for **O**ptimization and **T**erascale **A**pplications.
- DAKOTA is a general-purpose software toolkit for performing optimization, uncertainty quantification, parameter estimation, design of experiments, and sensitivity analysis on high performance computers.
 - DAKOTA is developed and supported by U.S. Sandia National Labs.
 - DAKOTA is well documented and comes with many tutorials.
 - Support via a dedicated mailing list.
- You can download DAKOTA toolkit at the following link: <http://dakota.sandia.gov/>
- Official releases and nightly stable releases are freely available worldwide via GNU GPL.
- Current version: 6.15 (released in November 2021)



DAKOTA overview

DAKOTA capabilities

- Parameter Studies (PS).
- Design of Experiments (DOE) – Design and Analysis of Computer Experiments (DACE) .
- Sensitivity Analysis (SA).
- Uncertainty Quantification (UQ).
- Optimization (OPT) via Gradient-based methods, and derivative-free local and global methods.
- Surrogate based optimization (SBO).
- Calibration (CAL) or data fitting – Parameter estimation or Nonlinear Least Squares Capabilities.
- Generic interface to black box solvers.
- Scalable parallel computations from desktop to clusters.
- Asynchronous evaluations.
- Simulation failure capturing.
- Restart capabilities.
- Matlab, scilab, AMPL, Python interface.
- Time-tested and advanced research algorithms to address challenging science and engineering simulations.

DAKOTA overview

Why DAKOTA? Why not matlab, scilab, octave, Java, Python, R, or any other optimization framework?

- DAKOTA does pretty much what any other optimizer does.
- A few DAKOTA's features:
 - Generic interface to black box solvers.
 - Scalable parallel computations, from desktop to clusters to the cloud.
 - Extensively validated.
 - Fully scriptable.
 - Simulation failure capturing.
 - Restart capabilities.
 - Parallel asynchronous or concurrent evaluations.
 - Can be linked to third-party optimization libraries.
 - No license fee.
 - Open-source.

← These are the main reasons
why I use DAKOTA

DAKOTA overview

Why DAKOTA? Why not matlab, scilab, octave, Java, Python, R, or any other optimization framework?

- In DAKOTA, two interfaces are available to introduce the simulation code into the optimization loop, namely, **direct** interface and **fork** interface.
 - The **direct** interface is the fastest, but it is intrusive. You will need to modify your simulation code to interact with DAKOTA.
 - The **fork** interface is slower but very flexible. It interacts with the simulation code via scripting files and input/output files.
- All the DAKOTA-OpenFOAM coupling tutorials that we are going present are based on the **fork** interface.
- However, we are going to see the **direct** interface in action using a toy application.

DAKOTA overview

DAKOTA Input file – The file *.in



- DAKOTA uses a single input file to orchestrate the optimization loop.
- The input file is human readable (ASCII format) and has the extension ***.in** (the extension is superfluous in UNIX/Linux like OS).
- In this input file the user formulates the problem, that is:
 - Method to use, variables, and responses.
 - Additionally, the user can define the interface to the black box solver, the level of parallelism, and general settings (such as the format of the output).
- If you misspell something or use a keyword that does not exist in the input file, DAKOTA will list the available options.
- Also, if you forget a compulsory entry in the input file, DAKOTA will complain and will ask you for that value.
- Optional entries will use the default values.
- Refer to the documentation for more information about the compulsory and optional entries of each method.

DAKOTA overview

Sample DAKOTA input file – The file *.in



Define algorithm

```
environment
  tabular_data
  tabular_data_file = 'output.dat'
```

environment (required): specify general settings such as tabular output. It also identifies the top-level method.

```
method
  max_iterations = 100
  convergence_tolerance = 1e-4
  conmin_frcg
```

method (required): specifies the method (DO or DSE) and specific settings

```
model
  single
```

model (optional): a model provides a logical unit for determining how a set of variables is mapped into a set of responses. The model allows one to specify a single interface or to manage more sophisticated mappings involving surrogates or nested iterations. Default value is single.

```
variables
  continuous_design =      2
  initial_point          -1.2      1.0
  lower_bounds           -2.0     -2.0
  upper_bounds            2.0      2.0
  descriptors             'x1'     'x2'
```

variables (required): parameters input to the simulation; these are the design variables.

```
Interface
  fork
  analysis_driver = 'simulator_script'
  asynchronous
```

interface (required): map from variables to responses; control parallelism

```
responses
  objective_functions = 1
  numerical_gradients
  no_hessians
```

responses (required): model output(s) to be studied, these are the response metrics.

Define problem (inputs and outputs)
Set interface

DAKOTA overview

DAKOTA execution

- DAKOTA can be run from a UNIX or Windows command line interface.
- DAKOTA also comes with a GUI (version 6.12 and newer), but we will not address how to use this graphical user interface.
- To run DAKOTA just type in the terminal window,

```
$> dakota -i input_file_name.in
```
- To run DAKOTA and save the standard output stream (stdout), input variables and response functions information for each function evaluation, method-specific info and so on,

```
$> dakota -i input_file_name.in -o output_file.out
```
- To save the standard error stream (stderr) as well,

```
$> dakota -i input_file_name.in -o output_file.out -e error.dat
```

DAKOTA overview

DAKOTA execution

- DAKOTA can be run from a UNIX or Windows command line interface.
- You can also use these options,

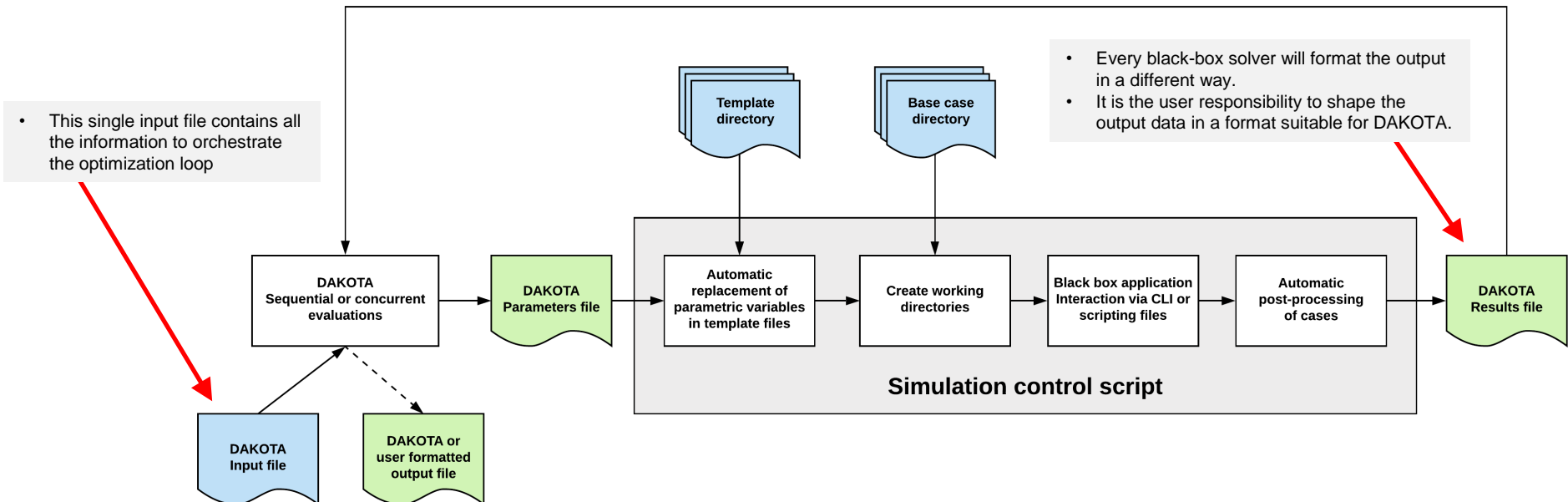
```
$> dakota -i input_file_name.in -o output_file.out > stdout.dat  
$> dakota -i input_file_name.in > stdout.dat  
$> dakota -i input_file_name.in | tee stdout.dat
```
- To get additional information and command line options,

```
$> dakota -help
```

DAKOTA overview

Workflow for data exchange between DAKOTA and a black-box application

- Workflow for data exchange between Dakota and a black-box application.
 - The white rectangles denote process blocks.
 - The light-shaded blue document symbols denote unchanging sets of files.
 - The light-shaded green document symbols indicate files that change with each set of design parameters generated by Dakota or after the end of the evaluation of the QOI.
 - The light-shaded grey area denotes the domain of the control script that automatically prepares the case; including, automatic formatting of input and output files, organization of the data generated, and executing the application in serial or parallel.



DAKOTA overview

DAKOTA documentation

- DAKOTA comes with extensive documentation and tutorials.
- You can access or download DAKOTA's documentation from the following link:

<https://dakota.sandia.gov/content/manuals>

Documentation tab

The screenshot shows the DAKOTA documentation website. The top navigation bar includes links for Home, Download, Documentation, Community, About, and Login (SNL Only). The 'Documentation' tab is highlighted, and a red arrow points to it. Below the navigation bar, the 'Manuals' section is displayed, listing the following manuals:

- **User's Manual** — The most comprehensive manual, containing overall information on Dakota, including its purpose and capabilities, a tutorial, interfacing guide, examples, and more.
- **Reference Manual** — A detailed input specification guide organized in the same hierarchical format as the Dakota input file
- **Developer's Manual** — Dakota design principles, services, best practices for team members, and development resources.
- **Theory Manual** — This manual accompanies the Users' Manual and provides more mathematical detail and deeper discussion of Dakota methods.
- **GUI User Manual** — A manual for the Dakota GUI, which doubles as a step-by-step tutorial to help users get studies up and running in the GUI.

Below the manuals list, there is a 'Notes' section and a 'Version 6.14' section. The 'Version 6.14' section lists the following links:

- User's Manual — pdf
- Reference Manual — html, tar.gz, (pdf only if needed)
- Developer's Manual — html, tar.gz, (pdf only if needed)
- Theory Manual — pdf
- GUI Manual — html

A black arrow points from the text 'Reference documentation of all available methods' to the 'Reference Manual' link in the Version 6.14 section.

DAKOTA overview

Summary of DAKOTA optimization methods

Gradient-based Optimization:

- **CONMIN:** frcg, mfd
- **OPT++:** cg, Newton, quasi-Newton
- **DOT:** frcg, bfgs, mmfd, slp, sqp (external package – commercial)
- **NPSOL:** sqp (external package – commercial)
- **NLPQLP:** sqp (external package – commercial)

Derivative-free Optimization:

- **COLINY:** PS, EA, Solis-Wets, COBYLA, DIRECT
- **JEGA:** MOGA, SOGA
- **EGO:** efficient global optimization via Gaussian Process models
- **NCSU:** DIRECT
- **OPT++:** PDS (Parallel Direct Search, simplex based method)
- **NOMAD:** mesh_adaptive_search

Parameter studies: vector, list, centered, grid, multidimensional, user defined

Design of experiments:

- **DDACE:** LHS, MC, grid, OA, OA_LHS, CCD, BB
- **FSUDace:** CVT, Halton, Hammersley
- **PSUADE:** MOAT
- **Sampling:** LHS, MC, Incr. LHS, IS/AIS/MMAIS

Multi-objective optimization, pareto, hybrid, multi-start, surrogate-based optimization (local and global), uncertainty quantification.

DAKOTA overview

Summary of DAKOTA optimization methods

- adaptive_sampling
- asynch_pattern_search
- bayes_calibration
- branch_and_bound
- centered_parameter_study
- coliny_beta
- coliny_cobyla
- coliny_direct
- coliny_ea
- coliny_pattern_search
- coliny_solis_wets
- conmin
- conmin_frcg
- conmin_mfd
- dace
- dl_solver
- dot
- dot_bfgs
- dot_frcg
- dot_mmfd
- dot_slp
- dot_sqp
- efficient_global
- efficient_subspace
- fsu_cvt
- fsu_quasi_mc
- genie_direct
- genie_opt_darts
- global_evidence
- global_interval_est
- global_reliability
- gpais
- hybrid
- importance_sampling
- list_parameter_study
- local_evidence
- local_interval_est
- local_reliability
- mesh_adaptive_search
- moga
- multi_start
- multidim_parameter_study
- ncsu_direct
- NI2sol
- nlpql_sqp
- nlssol_sqp
- nonlinear_cg
- npsol_sqp
- optpp_cg
- optpp_fd_newton
- optpp_g_newton
- optpp_newton
- optpp_pds
- optpp_q_newton
- pareto_set
- pof_darts
- polynomial_chaos
- psuade_moat
- richardson_extrap
- Rkd_darts
- sampling
- sogas
- stanford
- stoch_collocation
- surrogate_based_global
- surrogate_based_local
- vector_parameter_study

Note: The list is not complete

DAKOTA overview

Cleaning the case directory

A valuable advice before starting to work with DAKOTA:

Remember to always clean the case directory

- When running DAKOTA (using a direct or fork interface), it is highly advisable to start from a clean directory structure.
- In the case directory of every single tutorial distributed with this training material, you will find the script *dakota_cleanup*. This script will clean the case directory.
- To use this script, type in the terminal window:
 - `$> ./dakota_cleanup`
- By the way, you can customize this script.

Roadmap

- ~~1. Introduction to optimization methods~~
- ~~2. Choosing an optimization method~~
- ~~3. Optimization loop – The big picture~~
- ~~4. DAKOTA overview~~
- 5. Working with DAKOTA: Rosenbrock function**
- ~~6. Working with DAKOTA: Branin function~~
- ~~7. Working with DAKOTA: Multi-objective optimization~~
- ~~8. Coupling DAKOTA and OpenFOAM: driven cavity case~~
- ~~9. Additional code coupling tutorials~~
- ~~10. Some kind of conclusion~~

Working with DAKOTA: Rosenbrock function

- Optimization with DAKOTA.
- The Rosenbrock function.
- You will find this tutorial in the following directory:

```
$TM/dakota_sample_cases/model_problems/rosenbrock_direct_interface
```

Working with DAKOTA: Rosenbrock function

- In this tutorial, we will use the Rosenbrock function to illustrate many of the features included in DAKOTA.
- We are going to work with gradient based and derivative-free algorithms.
- We are going to do parametrical studies and design of experiments as well.
- Feel free to explore DAKOTA's input file.

Working with DAKOTA: Rosenbrock function

Rosenbrock function

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

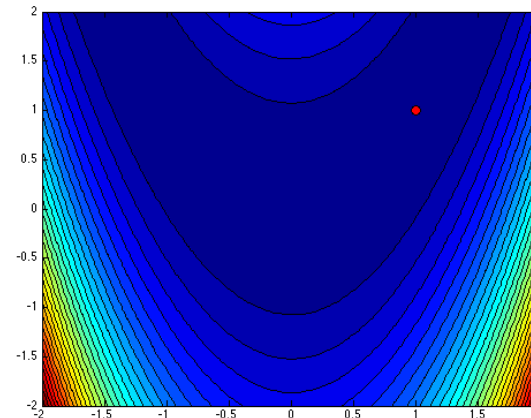
Search domain

$$-2 \leq y \leq 2 \quad -2 \leq x \leq 2$$

Global minimum

$$(x, y) = (1, 1)$$

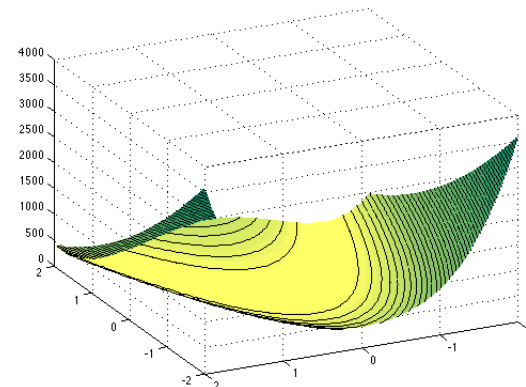
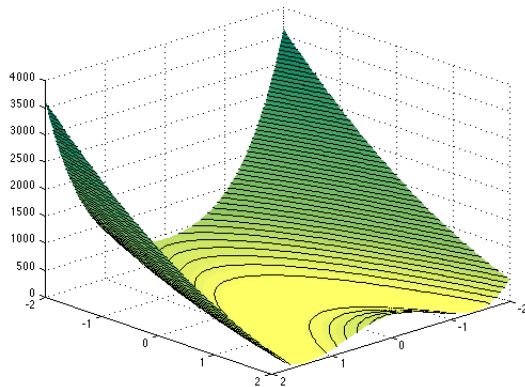
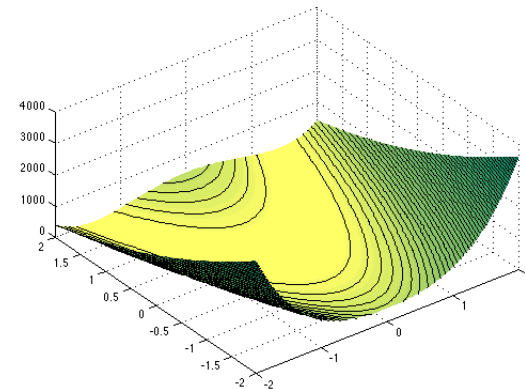
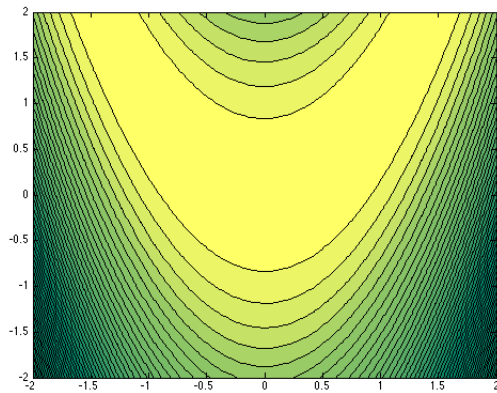
$$f(x, y) = 0$$



Working with DAKOTA: Rosenbrock function

Rosenbrock function

- The Rosenbrock function, is also known as the banana function because it actually looks like a banana (well, if you use the right color map and your imagination).



Working with DAKOTA: Rosenbrock function

Rosenbrock function

- In two dimensions (or two design variables), this problem seems to be an easy one. We can pinpoint the minimum visually.
- This problem can be extended to multiple dimensions (or design variables), where visual methods are of no use.
- Here is where optimization methods are valuable.

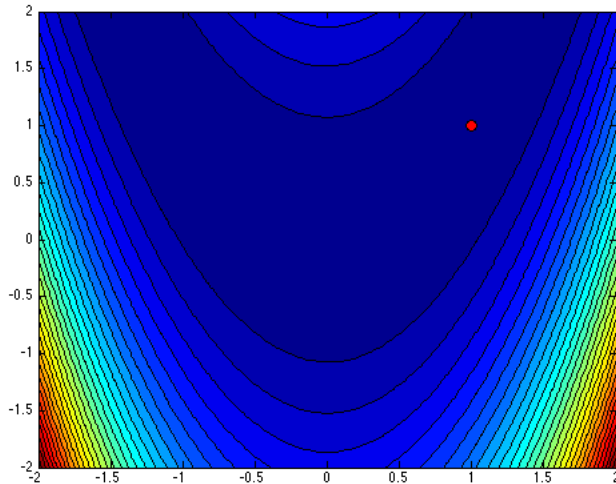
$$f(\mathbf{x}) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right]$$

Global minimum

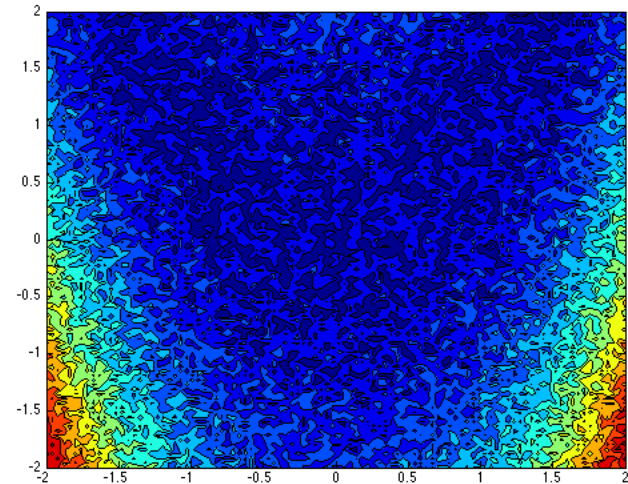
$$f(\mathbf{x}) = 0, \quad \text{at} \quad \mathbf{x} = (1, \dots, 1)$$

Working with DAKOTA: Rosenbrock function

Rosenbrock function with white noise



Smooth function



Noisy data

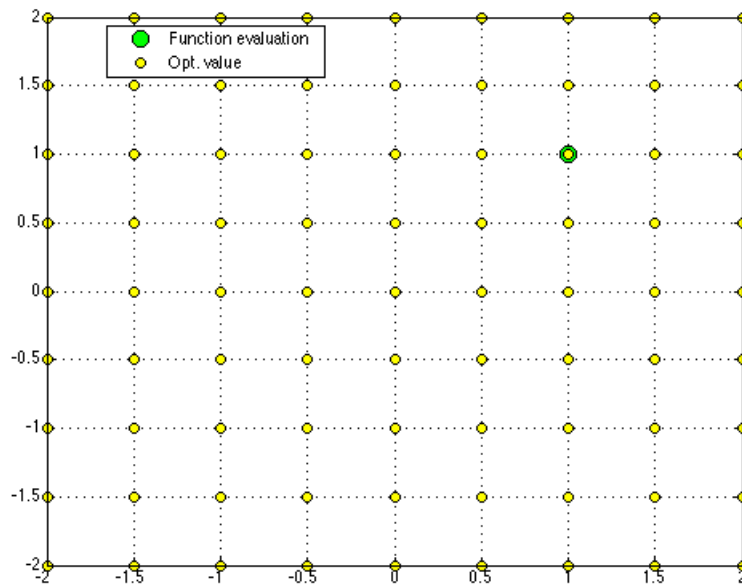
- In the ideal world, the optimization is done in a smooth function.
- But in reality (numerical or physical experiments) we have noisy data.
- Doing optimization in noisy data is tricky.
- Here, optimization using gradient based methods does not perform very well.

So, what can we do with DAKOTA?

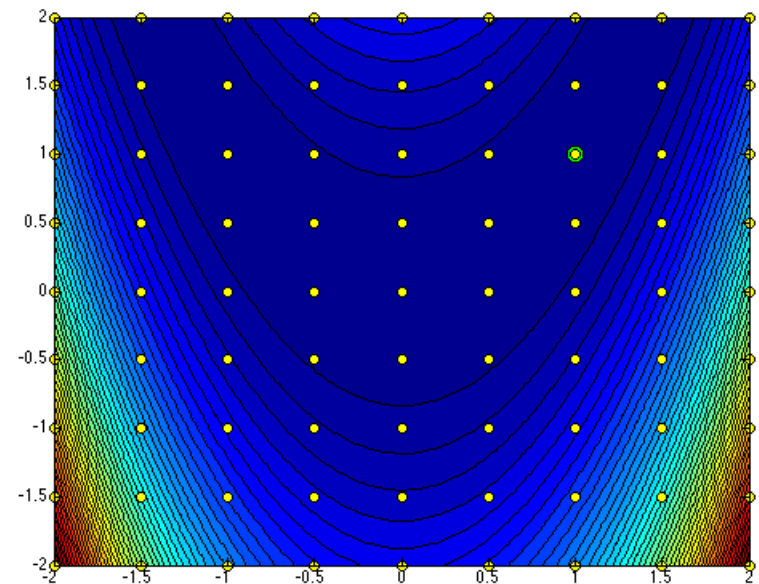
Working with DAKOTA: Rosenbrock function

Multidimensional study

- Use multidimensional experiments for parametrical studies and design space exploration.
- The output can be used in a sensitivity analysis, uncertainty quantification, initial screening or building a surrogate.



Sampling



SBO constructed from sampling

- In the directory `$TM/dakota_sample_cases/model_problems/rosenbrock_direct_interface/multi1` you will find the input files to run this case.

Let us use this simple case to illustrate how to use DAKOTA. But first we need to take a look at the input file.

Working with DAKOTA: Rosenbrock function

Running a simple DAKOTA case – *dakota_case.in* file



environment

graphics ←

Display a 2D graphics window of variables and responses (not used any more in DAKOTA 6.10 and newer releases)

tabular_data ←

Write an ascii file with the results (variables and responses).

tabular_data_file = 'rosen_multidim.dat' ←

Name of the tabular_data file.

method

multidim_parameter_study ←

Name of the optimization/parametrization method.

partitions = 8 8 ←

Parameters related to the optimization/parametrization method.

model

single ←

Method used to map variables into responses.
If no method is specified, the single method is used by default.

...

- If you do not want to use the graphics option in the environment block, you can erase it or commented.
- To comment a line in Dakota's input file, just add at the beginning the **#** symbol, e.g.:
#graphics.

Working with DAKOTA: Rosenbrock function

Running a simple DAKOTA case – *dakota_case.in* file



...

variables

continuous_design = 2 ← Number of design variables in a real interval.
lower_bounds -2 -2
upper bounds 2 2 ← Upper and lower bounds of the design variables.
descriptors 'x1' 'x2' ← Labels for the variables.

interface

direct ← Defines how Dakota should evaluate the function.
In this case we use the direct method, which is a driver
Compiled with Dakota.
analysis_driver = 'rosenbrock' ← Name of the driver that Dakota will use.

responses

response_functions = 1 ← Number of outputs or objective functions.
no_gradients
no_hessians ← Do not compute gradients and hessians.

Working with DAKOTA: Rosenbrock function

Running a simple DAKOTA case – *dakota_case.in* file



- If you want to get more information about all the options available for each of the blocks in Dakota's input file, refer to the reference manual and look for the section **keywords Area**.
 - <https://dakota.sandia.gov/content/latest-reference-manual>

The screenshot displays the DAKOTA Reference Manual website. At the top, there is a navigation bar with links for Home, Download, Documentation, Community, About, and Login (SNL Only). The main content area features the DAKOTA logo and the title '6.14 Reference Manual'. Below this, the 'Dakota Reference Manual' header is visible, along with a search bar. The left sidebar contains a list of navigation links, with 'Keywords Area' highlighted by a red box and an arrow pointing to it. The main content area shows the 'Keywords Area' section, which includes a summary of the input file structure and a list of keywords: environment, method, model, variables, interface, and responses. Each keyword is followed by its required status and frequency. The 'Introduction to Dakota Keywords' section follows, explaining the role of the environment block in managing execution modes and I/O streams.

Keywords Area

This page summarizes the overall input file structure, syntax, and the six types of blocks that may appear in Dakota input. Some are optional and some may appear multiple times:

- **environment** (optional; 0 or 1 may appear)
- **method** (required; 1 or more may appear)
- **model** (optional; 0 or more may appear)
- **variables** (required; 1 or more may appear)
- **interface** (required; 1 or more may appear)
- **responses** (required; 1 or more may appear)

Introduction to Dakota Keywords

In Dakota, the *environment* manages execution modes and I/O streams and defines the top-level iterator. Generally speaking, an iterator contains a model and a model contains a set of *variables*, an *interface*, and a set of *responses*. An iterator repeatedly operates on the model to map the variables into responses using the interface. Each of these six components (environment, method, model, variables, interface, and responses) are separate specifications in the user's input file, and as a whole, determine the study to be performed during an execution of the Dakota software.

A Dakota execution is limited to a single environment, but may involve multiple methods and multiple models. In particular, advanced iterators (i.e., meta- and component-based iterators) and advanced models (i.e., nested and surrogate models) may specialize to include recursions with additional sub-iterators and

Working with DAKOTA: Rosenbrock function

How to run this tutorial

- This case is ready to run, go to the following directory,

```
$> cd $TM/dakota_sample_cases/model_problems/rosenbrock_direct_interface/multil
```
- To run it, type in the terminal:
 1.

```
$> ./dakota_cleanup
```
 2.

```
$> dakota -i dakota_case.in
```
- You can run all the tutorials in a similar way.
- Remember to use the right name of the input file.
- **Also, we are going to run first using the direct interface.**

Working with DAKOTA: Rosenbrock function

How to run this tutorial

- After running the case, you will find two new files in the case directory, namely:
 - `dakota.rst`
 - `rosen_multidim.dat`
- The file `dakota.rst` is created automatically by Dakota, and it can be used to restart the simulation.
- The file `rosen_multidim.dat` contains the information about the variables and function evaluations. In the **environment** block of the input file, we enable this option and gave a name to the output file.
- Depending on the method you are using, you might find more files.
- But generally speaking, these two files are the most important ones.

Working with DAKOTA: Rosenbrock function

Screen output

- Also, depending on the method you are using you will get a different screen output with the information of the simulation.
- The screen output contains information about the problem, each function evaluation and summary statistics.

...

Using Dakota input file 'rosen_multidim.in'
Writing new restart file dakota.rst

>>>> Executing environment.

>>>> Running multidim_parameter_study iterator.

Multidimensional parameter study variable partitions of
8
8

...

Working with DAKOTA: Rosenbrock function

Screen output

- Also, depending on the method you are using you will get a different screen output with the information of the simulation.
- The screen output contains information about the problem, each function evaluation and summary statistics.

...

Begin Evaluation 1

Parameters for evaluation 1:

-2.0000000000e+00 x1

-2.0000000000e+00 x2

Direct interface: invoking rosenbrock

Active response data for evaluation 1:

Active set vector = { 1 }

3.6090000000e+03 response_fn_1

...

Working with DAKOTA: Rosenbrock function

Screen output

- Also, depending on the method you are using you will get a different screen output with the information of the simulation.
- The screen output contains information about the problem, each function evaluation and summary statistics.

...

<<<<< Function evaluation summary: 81 total (81 new, 0 duplicate)

Simple Correlation Matrix among all inputs and outputs:

	x1	x2	response_fn_1
x1	1.00000e+00		
x2	1.73472e-17	1.00000e+00	
response_fn_1	-3.00705e-03	-5.01176e-01	1.00000e+00

Partial Correlation Matrix between input and output:

	response_fn_1
x1	-3.47498e-03
x2	-5.01178e-01

...

Working with DAKOTA: Rosenbrock function

Screen output

- Also, depending on the method you are using you will get a different screen output with the information of the simulation.
- The screen output contains information about the problem, each function evaluation and summary statistics.

...

Simple Rank Correlation Matrix among all inputs and outputs:

	x1	x2	response_fn_1
x1	1.00000e+00		
x2	-3.87308e-02	1.00000e+00	
response_fn_1	-4.11247e-02	-5.03071e-01	1.00000e+00

Partial Rank Correlation Matrix between input and output:

	response_fn_1
x1	-7.01821e-02
x2	-5.05471e-01

<<<<< Iterator multidim_parameter_study completed.

...

Working with DAKOTA: Rosenbrock function

Screen output

- Also, depending on the method you are using you will get a different screen output with the information of the simulation.
- The screen output contains information about the problem, each function evaluation and summary statistics.

...

<<<<< Environment execution completed.

DAKOTA execution time in seconds:

Total CPU = 0.074166 [parent = 0.076, child = -0.001834]

Total wall clock = 0.181436

Exit graphics window to terminate DAKOTA.

- If you want to save the screen output to a file, you can proceed as follows:
 - `$> dakota -i dakota_case.in -o log.dat`

Working with DAKOTA: Rosenbrock function

Tabular output file – *rosen_multidim.dat* file



- Let us take a look at the *rosen_multidim.dat* file.

%eval_id	interface	x1	x2	response_fn_1
1	NO_ID	-2	-2	3609
2	NO_ID	-1.5	-2	1812.5
3	NO_ID	-1	-2	904
...				
...				
...				
79	NO_ID	1	2	100
80	NO_ID	1.5	2	6.5
81	NO_ID	2	2	401

- This file summarizes the variables and responses of the problem. Useful for Excel, Matlab, scilab, gnuplot, Python, or any other data analysis and plotting package.

Working with DAKOTA: Rosenbrock function

Plotting the tabular output

- Let us use gnuplot to plot the results of the tabular output file.
- Type in the terminal:
 - `$> gnuplot`
- You should get a screen output similar to this one:

GNUPLOT

Version 4.6 patchlevel 5 last modified February 2014 ← [gnuplot version](#)
Build System: Linux x86_64

Copyright (C) 1986-1993, 1998, 2004, 2007-2014
Thomas Williams, Colin Kelley and many others

gnuplot home: <http://www.gnuplot.info>
faq, bugs, etc: type "help FAQ"
immediate help: type "help" (plot window: hit 'h')

Terminal type set to 'qt'

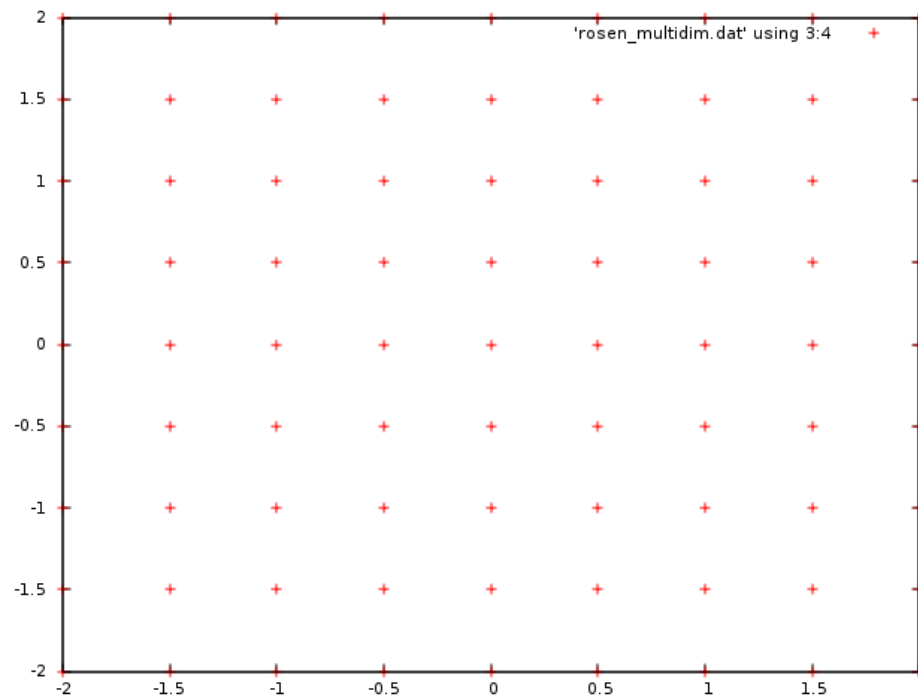
gnuplot>

[gnuplot](#)
prompt →

Working with DAKOTA: Rosenbrock function

Plotting the tabular output

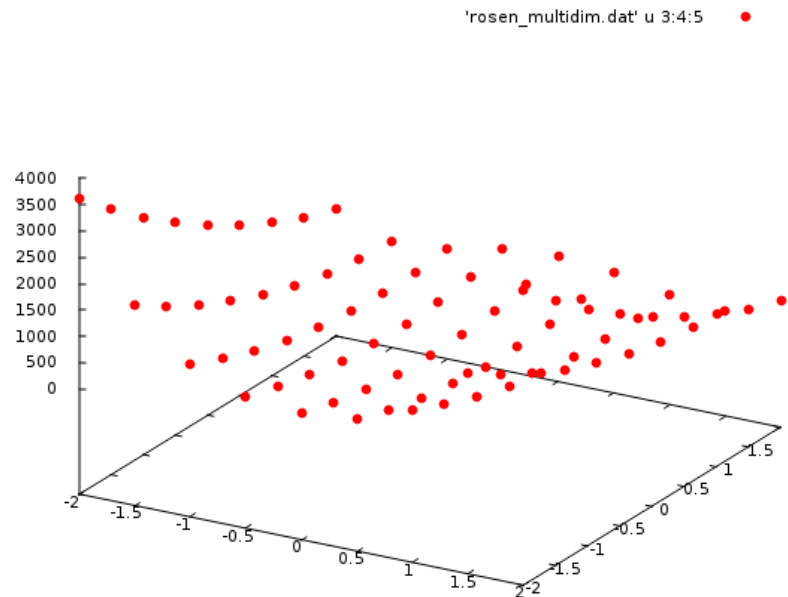
- To plot the variables (columns 3 and 4 in the file *rosen_multidim.dat*), type in the gnuplot prompt:
 - `gnuplot> plot 'rosen_multidim.dat' using 3:4 with points`



Working with DAKOTA: Rosenbrock function

Plotting the tabular output

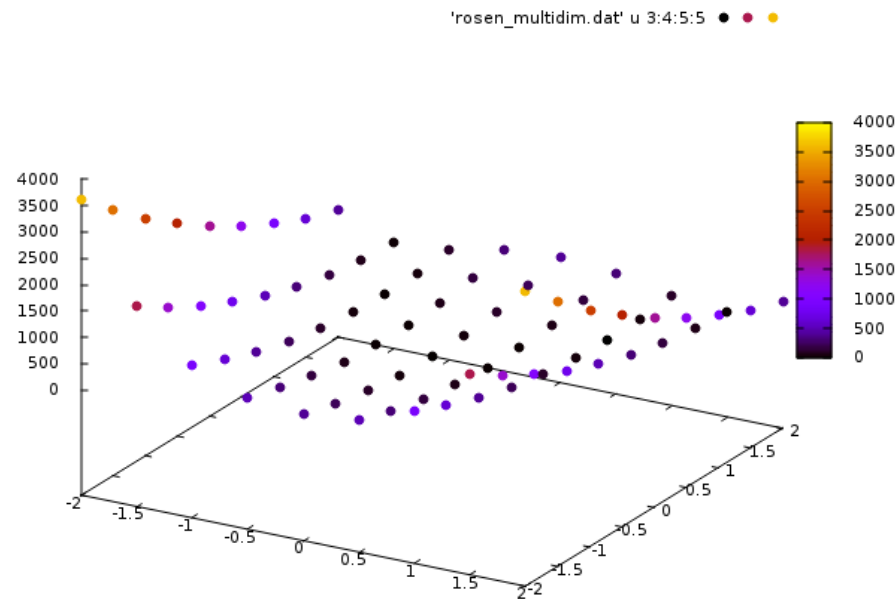
- To do a 3D plot of the variables and the response (columns 3, 4 and 5 in the file *rosen_multidim.dat*), type in the gnuplot prompt:
 - `gnuplot> splot 'rosen_multidim.dat' using 3:4:5 with points pointtype 7`



Working with DAKOTA: Rosenbrock function

Plotting the tabular output

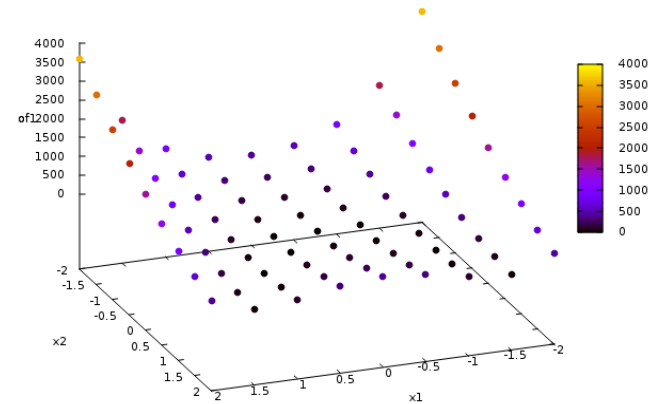
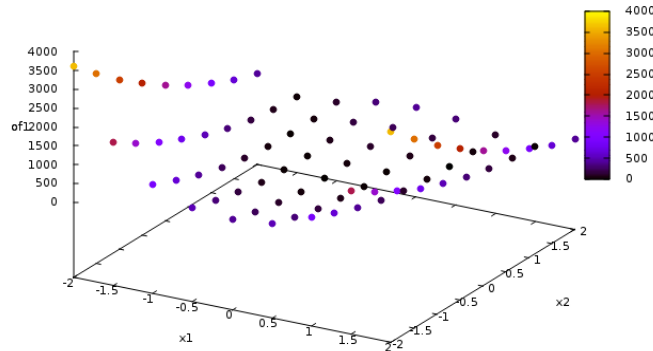
- To do a 3D plot of the variables and the response (columns 3, 4 and 5 in the file *rosen_multidim.dat*), color the points using the value of the response and add a colorbar, type in the gnuplot prompt:
 - `gnuplot> splot 'rosen_multidim.dat' using 3:4:5:5 with points pointtype 7 palette`



Working with DAKOTA: Rosenbrock function

Plotting the tabular output

- Finally, let us add axis labels and turn off the legend,
 - `gnuplot> set xlabel "x1"`
 - `gnuplot> set ylabel "x2"`
 - `gnuplot> set zlabel "of1"`
 - `gnuplot> set key off`
 - `gnuplot> plot 'rosen_multidim.dat' u 3:4:5:5 pt 7 ps 1 palette`



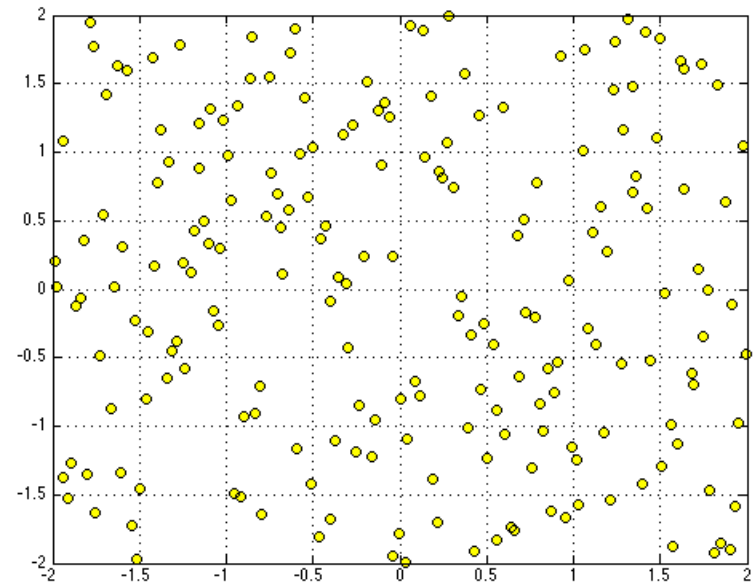
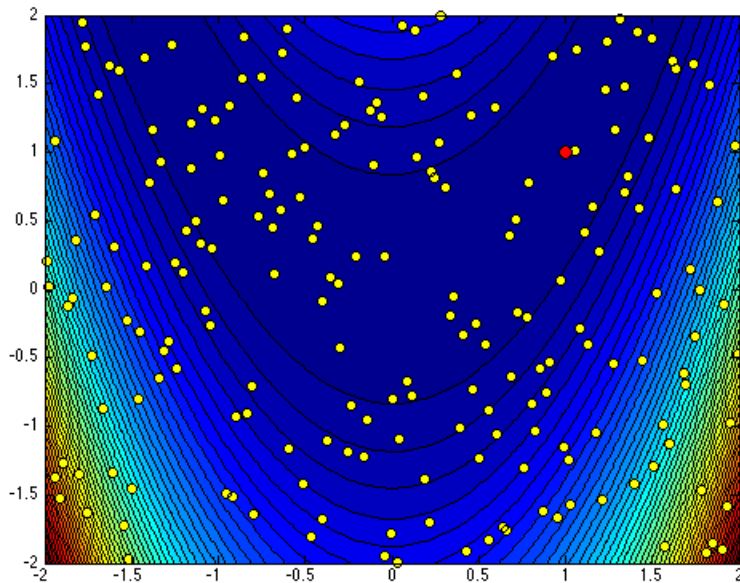
Working with DAKOTA: Rosenbrock function

- **You can run all the tutorials in a similar way.**
- **Depending on the method you are using, you might find more files in the case directory.**
- **Also, depending on the method you are using, you might need to use different options.**

Working with DAKOTA: Rosenbrock function

DACE – Latin Hypercube Sampling (LHS)

- DACE stands for design and analysis of computer experiments.
- Use DACE methods for deterministic experiments (computer simulations) and design space exploration. In computer experiments we are interested in sampling the parameter space in a representative way with the minimum number of samples.
- The output can be used in a sensitivity analysis, uncertainty quantification, or building a surrogate.

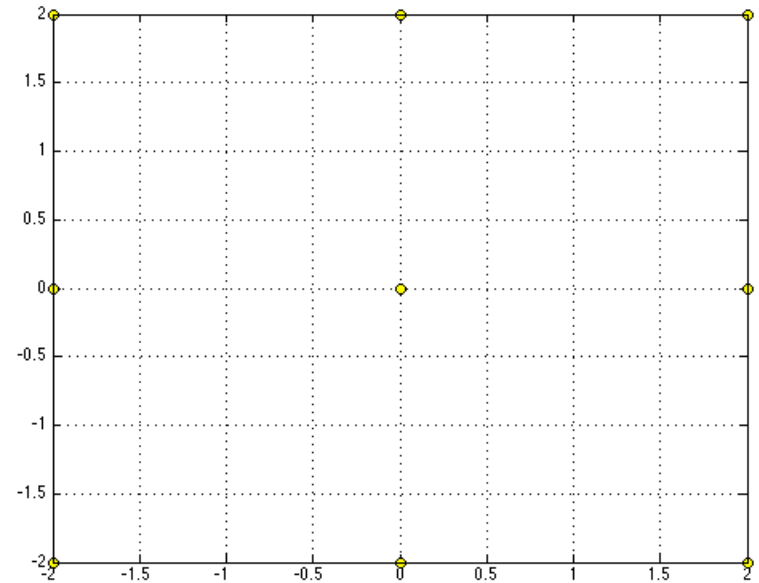
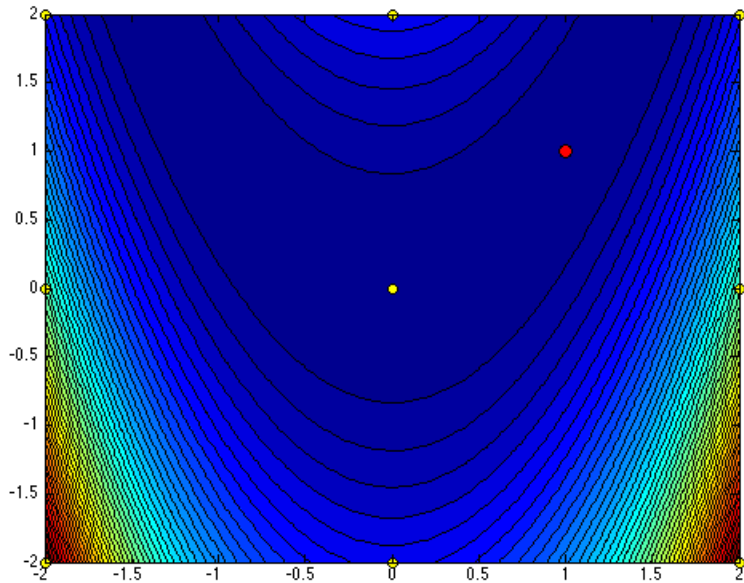


- In the directory `$TM/dakota_sample_cases/model_problems/rosenbrock_direct_inteface/dace1` you will find the input files to run this case.

Working with DAKOTA: Rosenbrock function

DOE – Central Composite Design (CCD)

- DOE stands for design of experiments.
- Use DOE methods for stochastic experiments (physical experiments) and design space exploration.

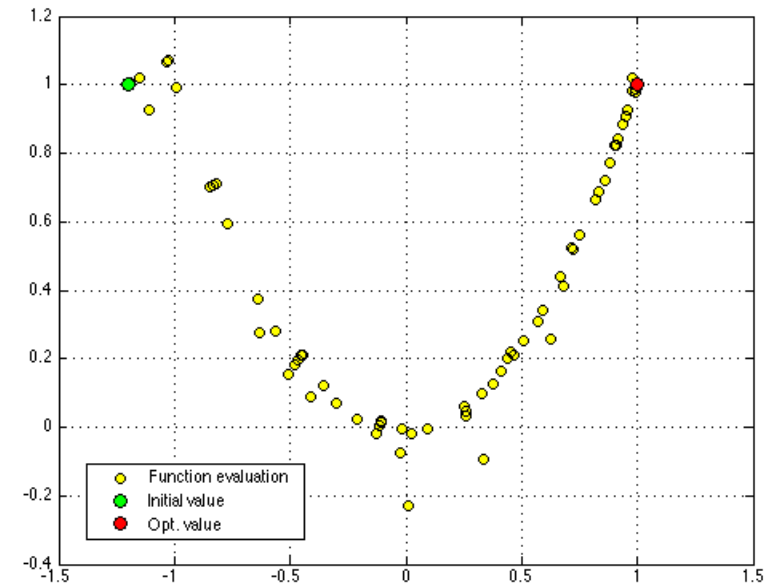
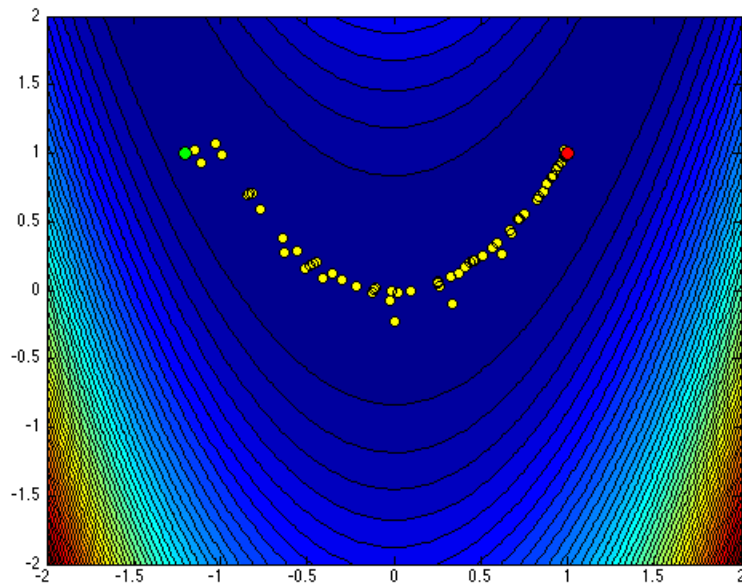


- In the directory `$TM/dakota_sample_cases/model_problems/rosenbrock_direct_inteface/dae1` you will find the input files to run this case.

Working with DAKOTA: Rosenbrock function

Gradient-based optimization – Fletcher-Reeves

- Gradient-based optimizers are best suited for efficient navigation to a local minimum in the vicinity of the initial point. Use this method for design optimization.
- They are not intended to find global optima in nonconvex design spaces.
- There are many gradient-based optimizer implemented in DAKOTA. The Fletcher-Reeves method requires first derivative information and can only be used in unconstrained problems.



- In the directory `$TM/dakota_sample_cases/model_problems/rosenbrock_direct_interface/grad1` you will find the input files to run this case.

- **Let us study this example to introduce a few new options in DAKOTA's input.**
- **We are also going to study how to formulate an optimization problem.**

Working with DAKOTA: Rosenbrock function

Gradient-based optimization case – *dakota_case.in* file



environment

#graphics

← Display a 2D graphics window of variables and responses (not used any more in DAKOTA 6.10 and newer releases)

tabular_data

tabular_data_file = 'rosen_grad_opt.dat'

method

conmin_mfd

← Optimization method (method of feasible directions).

max_iterations = 100

convergence_tolerance = 1e-4

← Parameters related to the optimization method. Remember, each optimization method has its own options. Refer to the reference manual for more information on the options available.

#linear_inequality_constraint_matrix 1 0

#

0 1

← Define coefficients of the linear inequality constraints.

#linear_inequality_lower_bounds -2 2

#linear_inequality_upper_bounds -2 2

← Define lower and upper bounds for the linear inequality constraint.

...

Working with DAKOTA: Rosenbrock function

Gradient-based optimization case – *dakota_case.in* file



...

model

single

variables

continuous_design = 2 ← Number of design variables in a real interval.

lower_bounds -2 -2

upper bounds 2 2

descriptors 'x1' 'x2'

← Upper and lower bounds of the design variables.

← Labels for the variables.

interface

direct

analysis_driver = 'rosenbrock'

...

Working with DAKOTA: Rosenbrock function

Gradient-based optimization case – *dakota_case.in file*



...

responses

response_functions = 1 ← Number of outputs or objective functions.

#analytic_gradients ← Compute analytical gradients,
must be provided by the user.

numerical_gradients ← Compute numerical gradients.

method_source
dakota ← Compute numerical gradients using dakota.

interval_type
forward ← Type of gradient (forward, central).

fd_gradient_step_size = 1.e-5 ← Gradient step size.

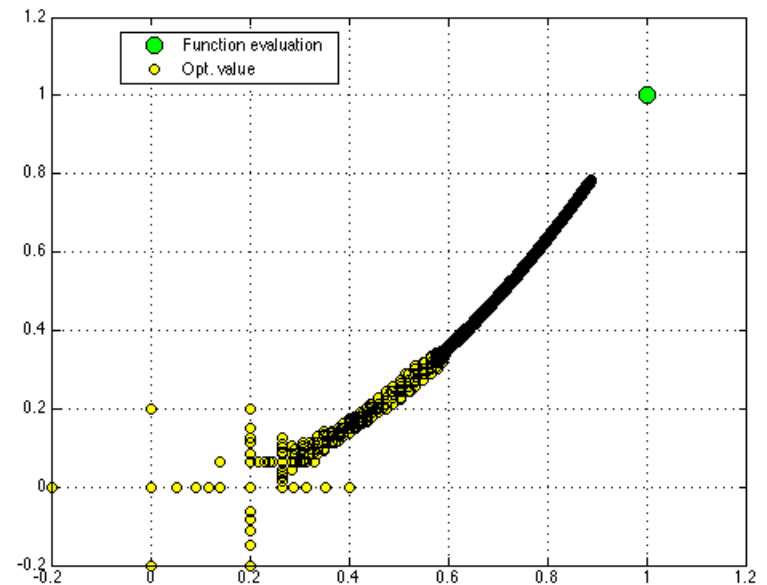
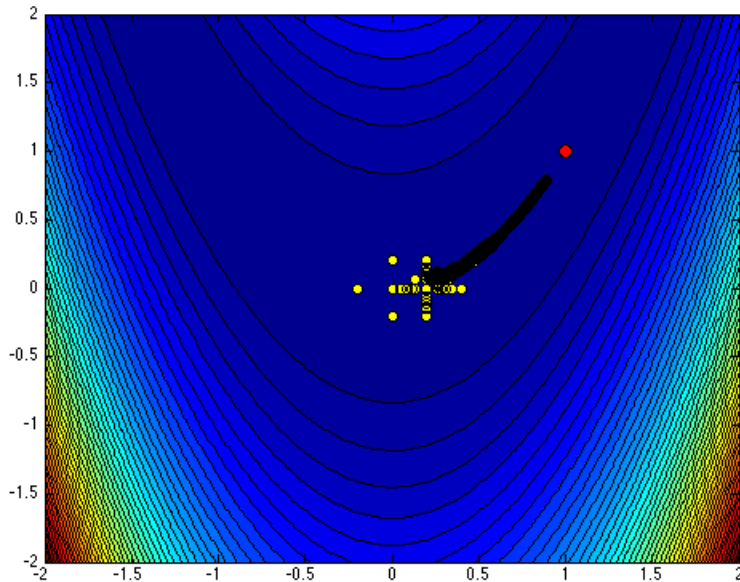
no_hessians ← Do not compute hessians.

sense 'min' ← Goal of the optimization, min or max.
If it is not specified it will minimize by the default (min).

Working with DAKOTA: Rosenbrock function

Gradient-free optimization – Pattern search

- Derivative/gradient-free methods can be more robust than gradient-based approaches.
- They can be applied in situations where gradient calculations are too expensive or unreliable.
- Pattern Search methods can be applied to nonlinear optimization problems. Use this method for design optimization.
- They generally walk through the domain according to a defined stencil of search directions.

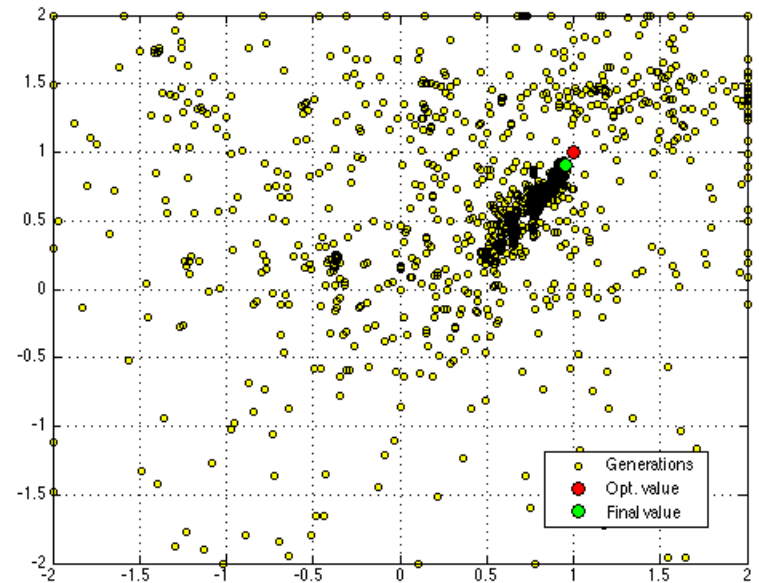
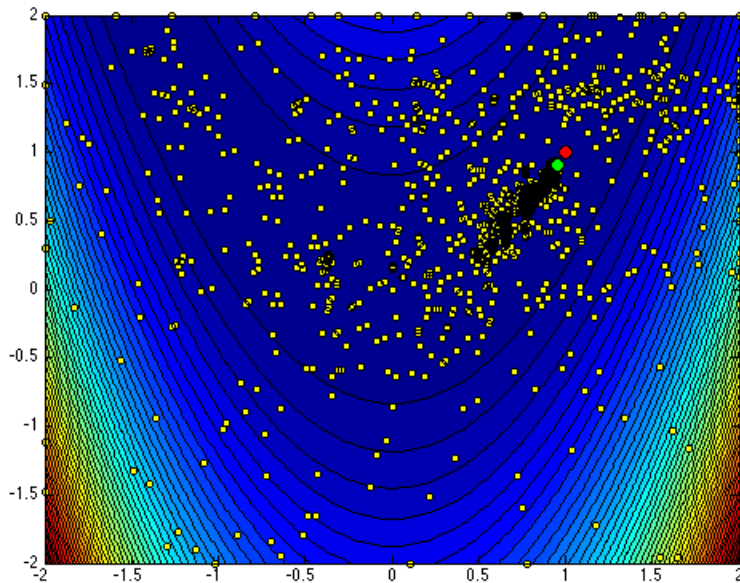


- In the directory `$TM/dakota_sample_cases/model_problems/rosenbrock_direct_interface/pattern_search` YOU will find the input files to run this case.

Working with DAKOTA: Rosenbrock function

Evolutionary algorithm – COLINY_EA

- Evolutionary algorithm are used for global optimization or multi-objective optimization.
- Evolutionary Algorithms (EA) are based on Darwin's theory of survival of the fittest.
- The EA simulates the evolutionary process by employing the mathematical analogs of processes such as natural selection, breeding, offspring and mutation.
- Use this method for design optimization. The data can be also used for design exploration.

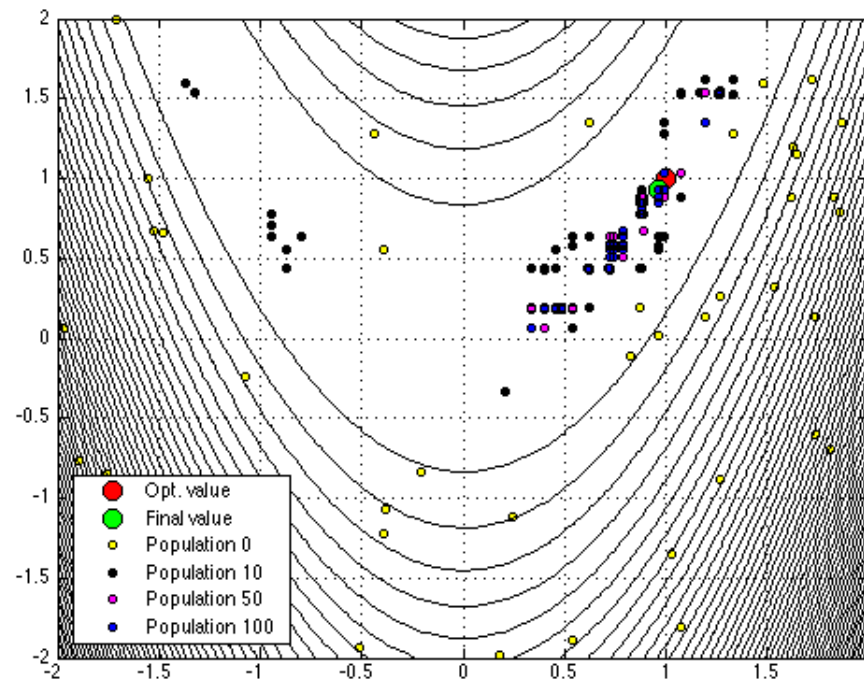


- In the directory `$TM/dakota_sample_cases/model_problems/rosenbrock_direct_interface/ea1` you will find the input files to run this case.

Working with DAKOTA: Rosenbrock function

Evolutionary algorithm – SOGA

- Ultimately, the EA identifies a design point (or a family of design points) that minimizes the objective function.
- EA methods are often used when the problem is non-smooth, multimodal, or poorly behaved.
- Use this method for design optimization. The data can be also used for design exploration.



- In the directory `$TM/dakota_sample_cases/model_problems/rosenbrock_direct_interface/ea2` you will find the input files to run this case.

Working with DAKOTA: Rosenbrock function

- **Let us couple DAKOTA with a generic application using the fork interface.**
- **This applies to any external application that can be run from the terminal window or command line interface.**
- **These cases are located in the directory**
`$TM/dakota_sample_cases/model_problems/python3/rosenbrock_fork_interface`

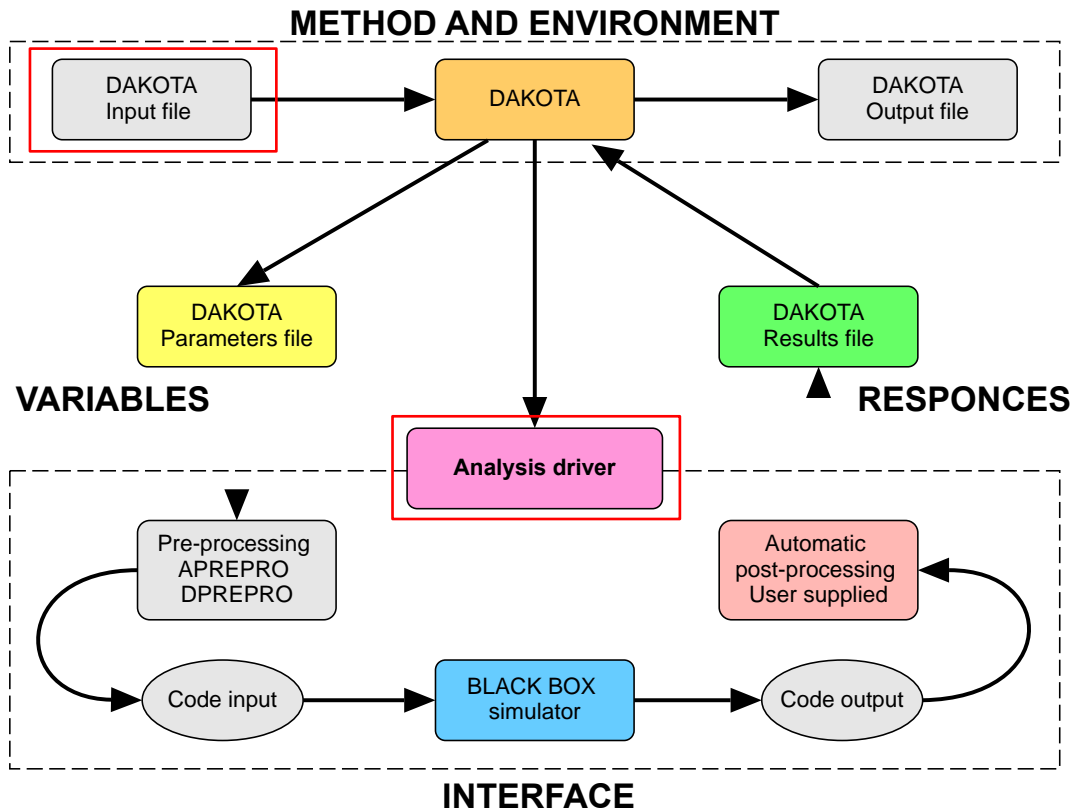
Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface

- So far, we solved the Rosenbrock function using a program built-in in DAKOTA via **direct** simulation interface.
- It is also possible to call an external program by using DAKOTA black-box interface.
- Two interfaces are available to link a simulation code with DAKOTA, namely, **system** interface and **fork** interface.
- Disregarding of the interface used (**system** or **fork**), pre-processing and post-processing functionality typically needs to be supplied (or developed) in order to transfer the parameters from DAKOTA to the black-box application and to extract the response values from the black-box application output file for return to DAKOTA.
- The big question is,
 - **fork** or system **interface**, which one do we use?.
- Following recommendations from the developers, we are going to use **fork** interface.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface

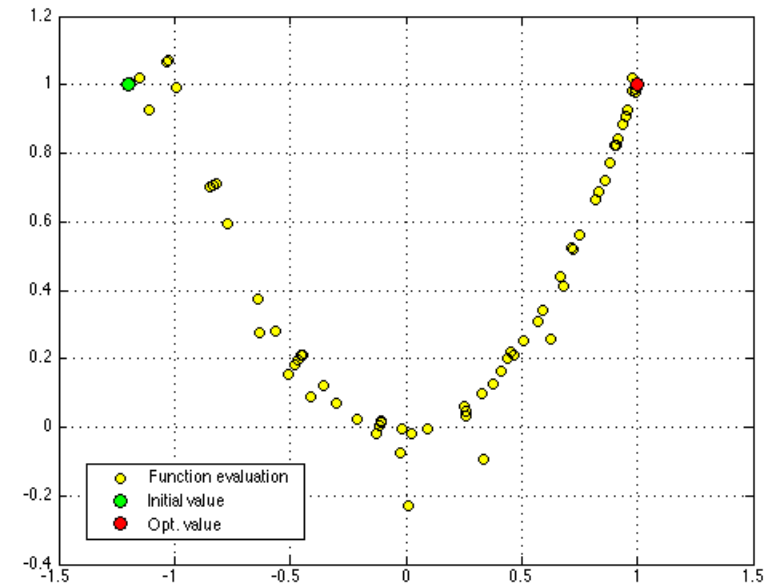
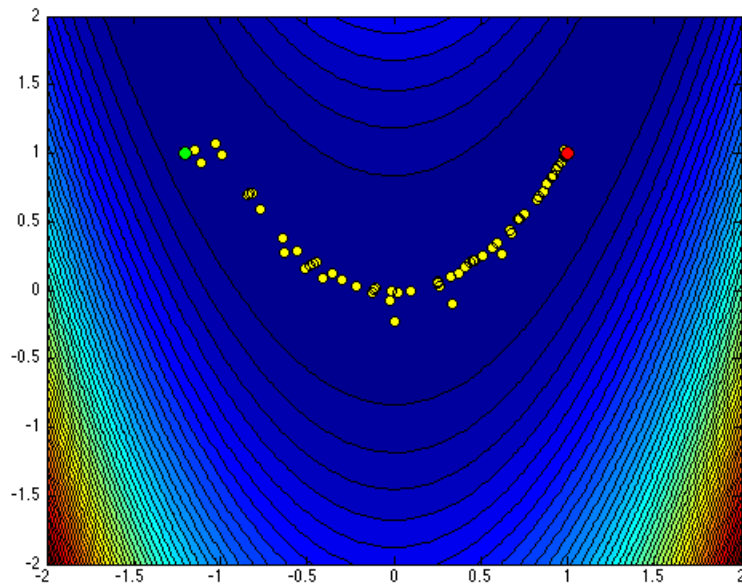


- To run a simulation loop using the **fork** interface, we only need to modify the block **interface** in dakota's input file.
- In the **interface** block we define the level of parallelism, analysis driver, and the name of the files generated and read by DAKOTA.
- Let us study how to interface a black-box program with DAKOTA.
- The program to be used is written in Python.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface

- The results of this simulation are the same as the one using the direct interface.
- The only difference is that it takes more time as it needs to parse all the information using the black-box interface.
- Remember, all the information of the design variables and quantities of interest is saved in the file *dakota_output.dat*. This was defined in the block environment in the .in file.



- In the directory `$TM/dakota_sample_cases/model_problems/python3/rosenbrock_fork_interface/grad1` you will find the input files to run this case.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



interface

→ fork

```
#asynchronous
#evaluation_concurrency = 2
analysis_driver = 'simulator_script'
parameters_file = 'params.in'
results_file = 'results.out'
work_directory directory_tag
copy_files = 'templatedir/*'
named 'workdir' file_save directory_save
#aprepro
```

fork interface


- The fork interface is defined in the block **interface** of DAKOTA's input file (.in)
- In this block we define the level of parallelism, analysis driver, and the name of the files generated and read by DAKOTA.
- We also define the name of the template directory.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



interface
fork



```
#asynchronous
#evaluation_concurrency = 2
analysis_driver = 'simulator_script'
parameters_file = 'params.in'
results_file = 'results.out'
work_directory directory_tag
copy_files = 'templatedir/*'
named 'workdir' file_save directory_save
#aprepro
```

asynchronous

- First at all, this entry is commented. The **#** symbol is used to comment lines.
- This entry defines if we want to run an asynchronous simulation, i.e., several simulations at the same time.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

evaluation_concurrency

- This keyword is related to the entry asynchronous and is also commented.
- This entry defines the number of concurrent simulations we want to conduct.
- If we do not define a value, DAKOTA will use the maximum number of cores available.
- The concurrent simulations can be parallel simulations as well.
- Concurrent simulations is an efficient way to exploit computational resources.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

analysis_driver


- In this entry we call the simulation script. In the simulation script we define what we want to do (we call the programs, we manipulate files, do the post-processing, and so on).
- In this script we give the instructions of how to copy the input generated by DAKOTA (*params.in*) to the input needed by the simulator program.
- We also create the simulation output file (*results.out*) that DAKOTA reads.
- We are going to study the *simulator_script* later on.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```



parameters_file

- This is the name of the file generated by DAKOTA.
- This file will be used later as an input for the simulation program.
- The default name is *params.in*, but you can change it.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

results_file

- This is the name of the file that DAKOTA reads.
- This file is generated by the simulation program.
- The default name is *results.out*, but you can change it.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

These are two
different entries

work_directory

- When the **work_directory** feature is enabled, DAKOTA will create a directory for each evaluation, with optional tagging (**directory_tag**) and saving (**directory_save**).
- Everything will be done in the working directory and all evaluations are relative to the current working directory.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

These are two
different entries

directory_tag

- If this keyword is used, DAKOTA will append a period and the function evaluation number to the work directory names.
- If this keyword is omitted, the default is no tagging, and the same work directory will be used for all function evaluations.
- Tagging is most useful when multiple function evaluations are running simultaneously.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

copy_files

- In this entry we define the directory where all files needed to run the simulation are located.
- Every file needed to run the simulation must be located here.
- The location of this directory is in reference to the case directory.
- That is, **templatedir** is located in the working directory.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

These are three
different entries

named 'workdir'

- In this entry we give the base name of the simulation directories (**workdir.N**).
- Every file needed to run the simulation will be copied in the directory **workdir.N**.
- The location of this directory is in reference to the case directory, which is enable by using the entry **work_directory**.
- That is, **workdir.N** is located in the case directory.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

These are three
different entries

directory_save

- Preserve the work directory after function evaluation completion.
- By default, when a working directory is created by DAKOTA using the **work_directory** keyword, it is deleted after the evaluation is completed.
- The **directory_save** keyword will cause DAKOTA to leave (not delete) the directory.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file



```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

These are three
different entries

file_save

- Keep the parameters and results files after the analysis driver completes.
- If **file_save** is used, Dakota will not delete the parameters and results files after the function evaluation is completed.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *dakota_case.in* file




```
interface
  fork
    #asynchronous
    #evaluation_concurrency = 2
    analysis_driver = 'simulator_script'
    parameters_file = 'params.in'
    results_file = 'results.out'
    work_directory directory_tag
    copy_files = 'templatedir/*'
    named 'workdir' file_save directory_save
    #aprepro
```

aprepro

- The format of data in the parameter files can be modified for direct usage with the APREPRO pre-processing tool using the aprepro [1] specification
- Without this keyword, the parameters file are written in DPrePro format.
- DPrePro is a utility included with Dakota, described in the Users Manual.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *simulator_script* file



```
dprepro $1 ros.template ros.in
```

```
python3 p1.py
```

```
mv results.txt $2
```

dprepro \$1 ros.template ros.in


- In the *simulator_script* file, **\$1** refers to the file *params.in* generated by DAKOTA and **\$2** refers to the results file returned to DAKOTA.
- Here, dprepro will take the values generated by DAKOTA (the file *params.in*), and will copy them into *ros.template*.
- The final file will be named *ros.in*.
- The script dprepro will copy the values generated by DAKOTA in the places where it finds the entries **{x1}** and **{x2}** in the file *ros.template*.
- The values **x1** and **x2** correspond to the design variables defined in the .in file.
- In this way, we automatically copy the values generated by DAKOTA to the file *ros.in*, that will be read by the program.
- The file *ros.template* is located in the directory **templatedir**, as defined in the *dakota_rosenbrock.in* file.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *simulator_script* file



```
dprepro $1 ros.template ros.in
```



```
python3 p1.py
```

```
mv results.txt $2
```

python3 p1.py

- After creating the file *ros.in*, we can proceed to run the program.
- In this case, we are using a small Python script (Python 3) to compute the Rosenbrock function and its derivatives.
- Have in mind that at this point, you can call any program. The only requirement is that it must be able to run from the terminal, and preferably with no graphical interface.
- The Python script *p1.py* will read the file *ros.in*, will do some computations and will create the file *results.txt*
- Remember, we are now working in a subdirectory located in the case directory, namely **workdir.1**, **workdir.2**, ... **workdir.N**.
- All the files needed, and the simulation information are saved in this directory.
- The python script *p1.py* is located in the directory **templatedir**, as defined in the *dakota_rosenbrock.in* file.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *simulator_script* file



```
dprepro $1 ros.template ros.in
```

```
python3 p1.py
```

```
→ mv results.txt $2
```

mv results.txt \$2

- Finally, we copy the file *results.txt* into the file *results.out*.
- DAKOTA will read the file *results.out* and it will keep iterating until reaching convergence or the maximum number of function evaluations.
- Creating all the files, doing the post-processing and setting the rules, is the user responsibility.



Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *simulator_script* file



- The file *ros.in* (which is created from the file *ros.template*), needed by your program and generated by DAKOTA, is structured as follows:

{x1} **{x2}**

- The script *dprepro* will copy the values generated by DAKOTA in the places where it finds the entries **{x1}** and **{x2}** in the file *ros.template*.
- The values **x1** and **x2** correspond to the design variables defined in the **variables** block in the *dakota_rosenbrock.in* file.
- The values generated by DAKOTA will be automatically copied to the file *ros.in*, that will be read by the program.
- Remember, the file *ros.template* is located in the directory **templatedir**, as defined in the interface block of the *dakota_rosenbrock.in* file.
- You can have as many templates files as you like.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *simulator_script* file



- The file *results.out* needed by DAKOTA, must be structured as follows:

Objective functions

Non-linear inequalities

Non-linear equalities

[Analytical gradients]

[[Analytical hessians]]

- The entries are used only if they are required by DAKOTA.
- If a required entry is not specified, DAKOTA will complain. In the same way, if an additional entry is provided and it is not required, DAKOTA will complain.
- In a few words, if the amount of data in this file does not match the function request vector, DAKOTA will abort execution with an error message.

Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *simulator_script* file



- For example, for a problem with two objective functions, one non-linear inequality, three non-linear equalities, no analytical gradients, and no analytical hessians, the *results.out* file should look like this one:

1.510	#Objective function 1
3.743	#Objective function 2
0.35	#Non-linear inequality 1
2.1	#Non-linear equality 1
1.0	#Non-linear equality 2
0.514	#Non-linear equality 3

- It is the user responsibility to create and format this file.



Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – *simulator_script* file

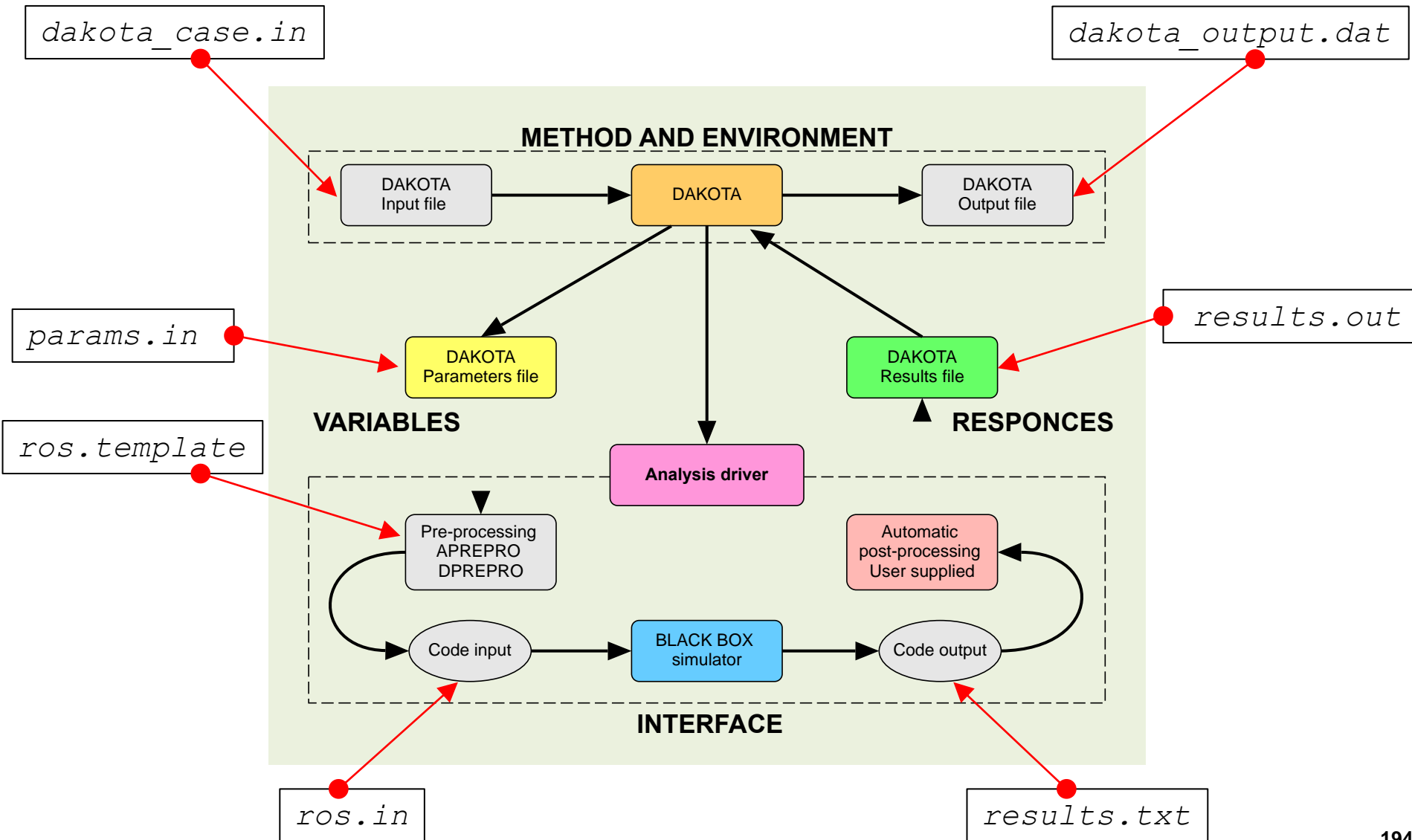


- In this case, DAKOTA will only ask for one input (objective function value).
- Therefore, the file *results.out* needed by DAKOTA, is structured as follows:

Objective function numerical value

Working with DAKOTA: Rosenbrock function

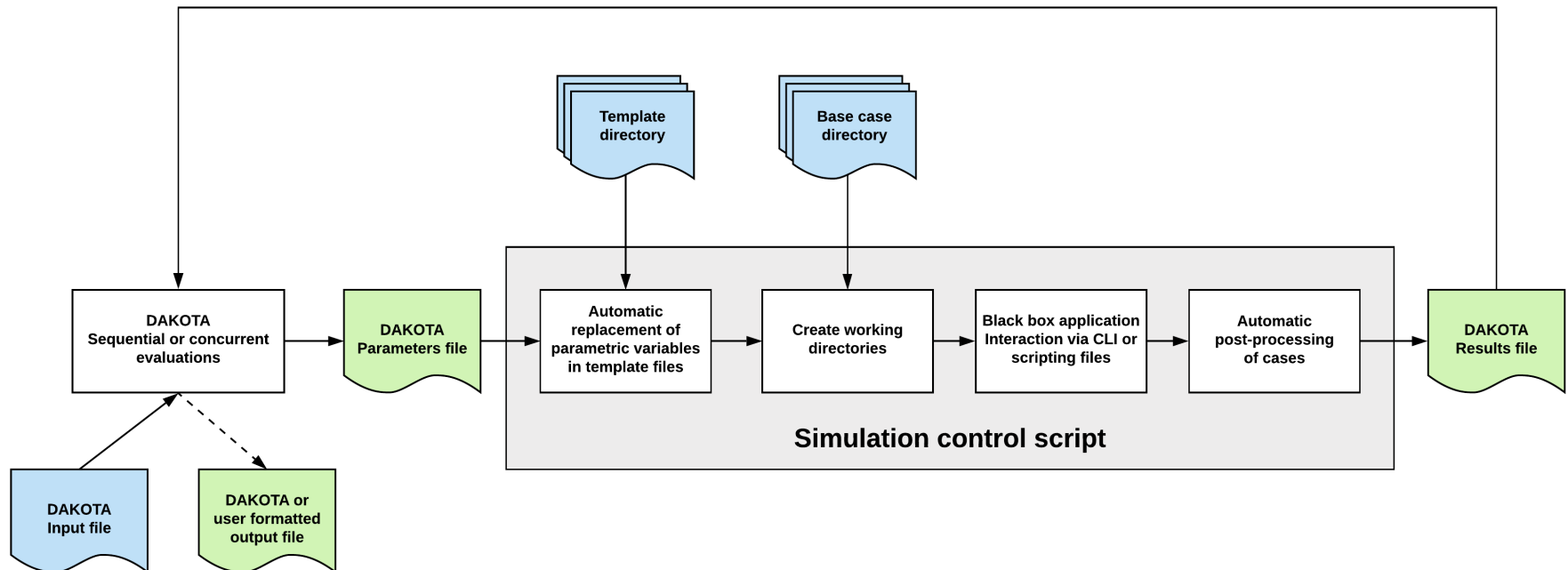
So, what just happened here?



Working with DAKOTA: Rosenbrock function

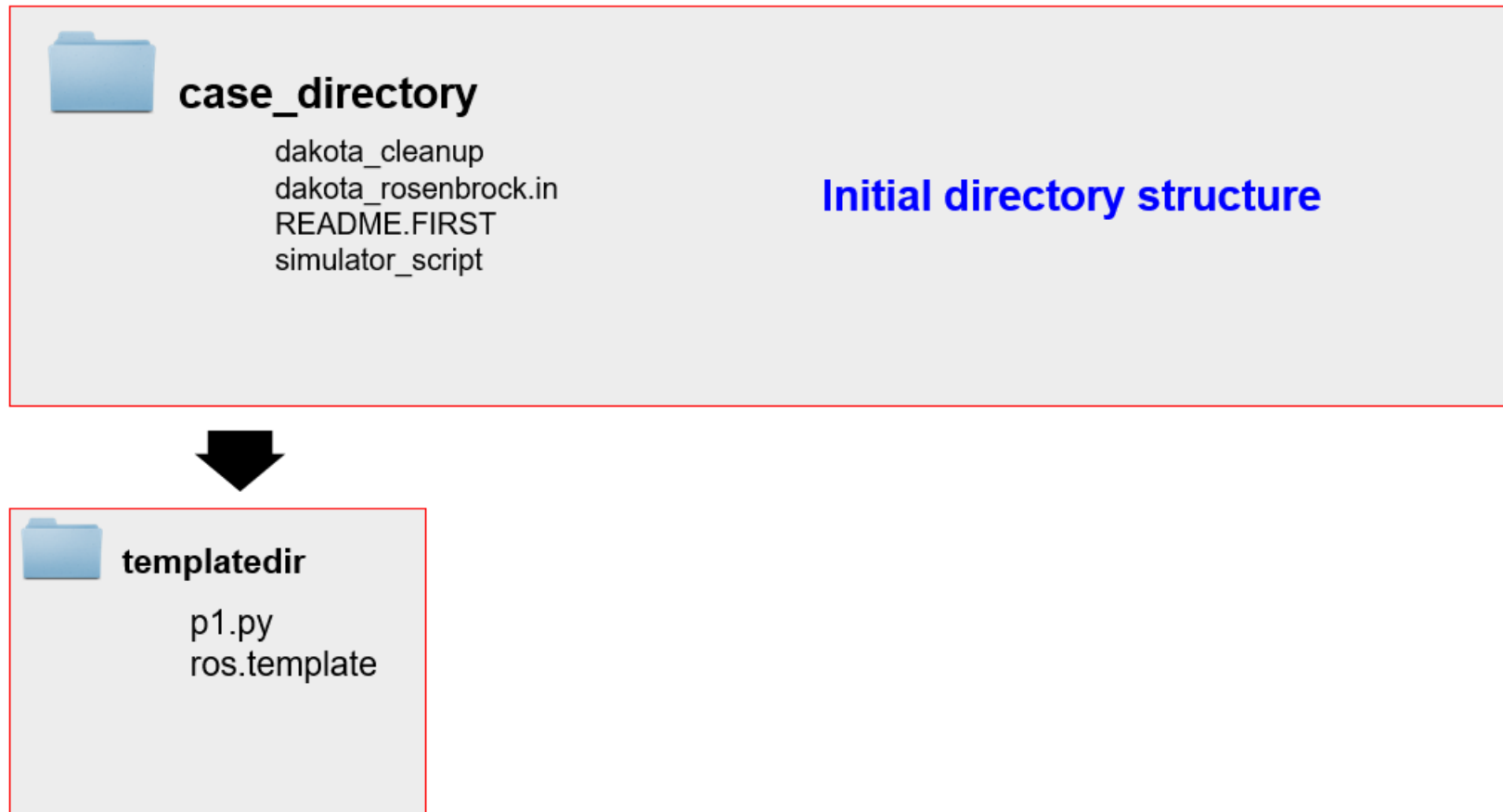
So, what just happened here?

- At this point we hope that the workflow for data exchange between Dakota and the black-box application is crystal clear.



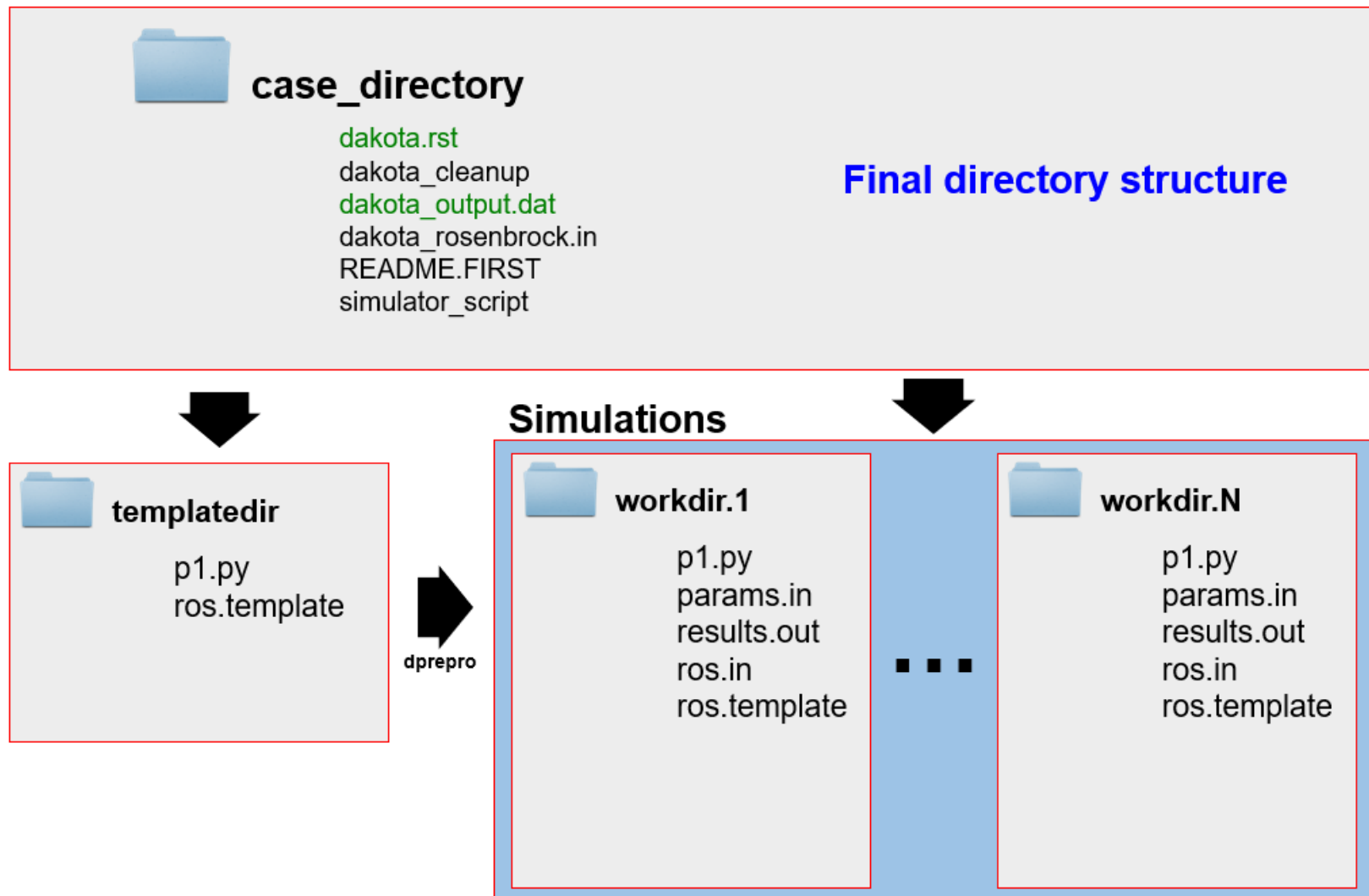
Working with DAKOTA: Rosenbrock function

Fork interface directory structure



Working with DAKOTA: Rosenbrock function

Fork interface directory structure



Working with DAKOTA: Rosenbrock function

Running DAKOTA using a fork interface – Files and directories that will be used

- Directories:
 - Case directory.
 - **templatedir**
- Input files:
 - *dakota_case.in*
 - *simulator_script*
 - *templatedir/ros.template*
 - *templatedir/p1.py*
- Output files:
 - *dakota_output.dat*
 - A lot of files inside the generated working directories (**workdir.N**).

Working with DAKOTA: Rosenbrock function

How to run this tutorial

- This case is ready to run, go to the directory
- `$> cd $TM/dakota_sample_cases/model_problems/python3/rosenbrock_fork_interface/grad1`
- To run it, type in the terminal:
 1. `$> ./dakota_cleanup`
 2. `$> dakota -i dakota_case.in`
- At this point, let us explore the rest of the sub-directories.

Working with DAKOTA: Rosenbrock function

Restarting a DAKOTA simulation

- It is possible to restart a DAKOTA simulation.
- Restarting can be necessary after interruptions imposed by computer usage policies, power failures, system failures, and so on.
- Also, it may happen that you would like to conduct the same optimization, but to a tighter final convergence tolerance.
- Dakota automatically records the variable and response data from all function evaluations so that new executions of DAKOTA can pick up where previous executions left off.
- Unless otherwise specified, the restart information is written to the default restart file, i.e., *dakota.rst*, which is written in binary format.
- Restarting can be used with **direct** and **fork** interfaces.

Working with DAKOTA: Rosenbrock function

Restarting a DAKOTA simulation

- To write a restart file using a particular name, the `–write_restart` command line input (may be abbreviated as `–w`) is used:
 - `$> dakota -i input.in -write_restart my_restart_file`
- To restart DAKOTA from a restart file, the `–read_restart` command line input (may be abbreviated as `–r`) is used:
 - `$> dakota -i input.in -read_restart my_restart_file`
- To read in only a portion of a restart file, the `–stop_restart` control (may be abbreviated as `–s`) is used to specify the number of entries to be read from the database. For example, to read in the first 50 evaluations from `dakota.rst`:
 - `$> dakota -i input.in -r dakota.rst -s 50 -w dakota_new.rst`

Working with DAKOTA: Rosenbrock function

The DAKOTA restart utility

- The Dakota restart utility program provides a variety of facilities for managing restart files from Dakota executions. The executable program name is `dakota_restart_util` and it has many options, as shown by the usage message returned when executing the utility without any options.
- To print the contents of a particular restart file in human-readable format, you can proceed as follows:
 - `$> dakota_restart_util print dakota.rst`
- To save the contents of a particular restart file in tabular format, you can proceed as follows:
 - `$> dakota_restart_util to_tabular dakota.rst output.txt`
- You can also remove corrupted data from the restart file, but we will not address this. For more information refer to DAKOTA's user guide.

Working with DAKOTA: Rosenbrock function

How to restart a case

- Let us run again the case

```
$> cd $TM/dakota_sample_cases/model_problems/python3/rosenbrock_fork_interface/grad1
```
- To run it, type in the terminal:
 1.

```
$> ./dakota_cleanup
```
 2.

```
$> dakota -i dakota_case.in
```
- At any point in the simulation press **ctrl-c**, in my case I will stop the simulation about the 45th function evaluation.

Working with DAKOTA: Rosenbrock function

How to restart a case

- Now let us print the information contained in the file *dakota.rst*, type in the terminal:
 - ```
$> dakota_restart_util print dakota.rst
```
- It will print all the simulations that were correctly evaluated and saved.
- This means, that we can restart the simulation from the last record saved in the file *dakota.rst*.
- However, it is a good practice to restart one or two records before the last one reported in the file *dakota.rst*.



# Working with DAKOTA: Rosenbrock function

## How to restart a case

- In my case I get the following information:

...  
...  
...

-----  
**Restart record 43 (evaluation id 43):**  
-----

**Parameters:**

-4.8642720829849601e-01 x1  
1.8615938867625639e-01 x2

**Active response data:**

**Active set vector = { 1 }**  
2.4640066825800000e+00 obj\_fn

**Restart file processing completed: 43 evaluations retrieved.**

- To play it safe, I will restart from the record 40.

# Working with DAKOTA: Rosenbrock function

## How to restart a case

- To restart the simulation, type in the terminal:
  - `$> dakota -i dakota_case.in -r dakota.rst -s 40 -w dakota1.rst`
- We are restarting the simulation using the file *dakota.rst*, we are restarting from the record 40 (check that the folder exist in the case directory), and we are writing a new restart file named *dakota1.rst*.
- Before restarting, feel free to change any parameter in the *dakota\_case.in* file.
- For instance, you can change the **interval\_type** and **fd\_gradient\_step\_size** keywords.

# Roadmap

- ~~1. Introduction to optimization methods~~
- ~~2. Choosing an optimization method~~
- ~~3. Optimization loop – The big picture~~
- ~~4. DAKOTA overview~~
- ~~5. Working with DAKOTA: Rosenbrock function~~
- 6. Working with DAKOTA: Branin function**
- ~~7. Working with DAKOTA: Multi-objective optimization~~
- ~~8. Coupling DAKOTA and OpenFOAM: driven cavity case~~
- ~~9. Additional code coupling tutorials~~
- ~~10. Some kind of conclusion~~

# Working with DAKOTA: Branin function

- Optimization with DAKOTA.
- The Branin function.
- You will find this tutorial in the following directory:

```
$TM/dakota_sample_cases/model_problems/python3/branin
```

# Working with DAKOTA: Branin function

- In this tutorial, we will use the Branin function to illustrate the idea behind surrogate-based optimization (SBO)
- First, we are going to work with design and analysis of computer experiments (DACE).
- After building the surrogate, we are going to work with gradient based algorithms.
- Feel free to explore DAKOTA's input file.

# Working with DAKOTA: Branin function

## The Branin function

$$f(x, y) = \left( y - \frac{5.1}{4\pi^2}x^2 + \frac{5}{\pi}x - 6 \right)^2 + 10 \left( 1 - \frac{1}{8\pi} \right) \cos(x) + 10$$

Subject to

$$s.t. \ 0 \leq y \leq 15 \quad s.t. \ -5 \leq x \leq 10$$

Global minimum

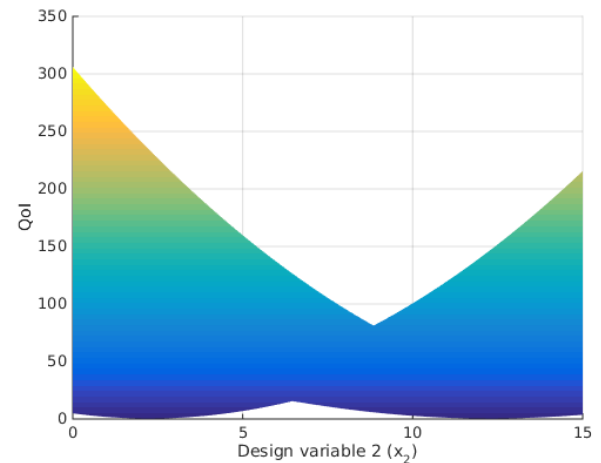
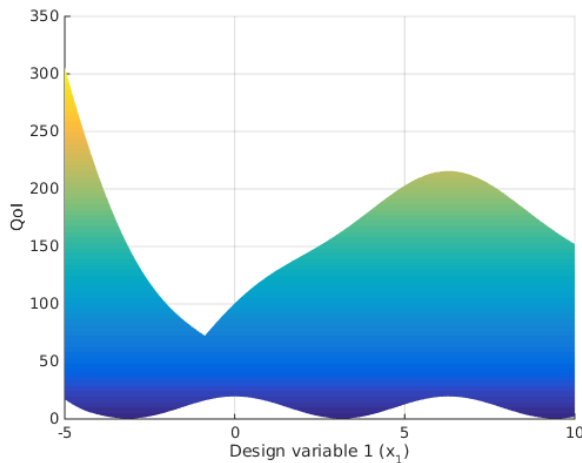
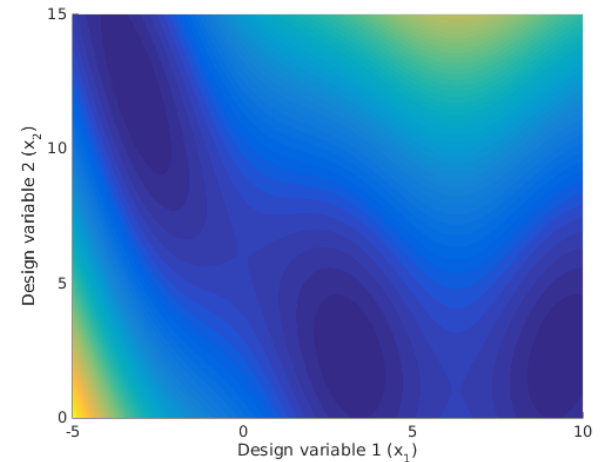
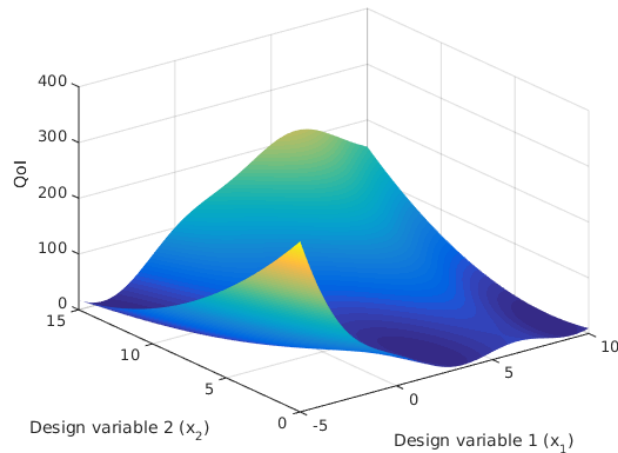
$$f(x, y) = 0.397887$$

$$(x, y) = (-\pi, 12.275), (\pi, 2.275), (9.42478, 2.475)$$

# Working with DAKOTA: Branin function

## The Branin function

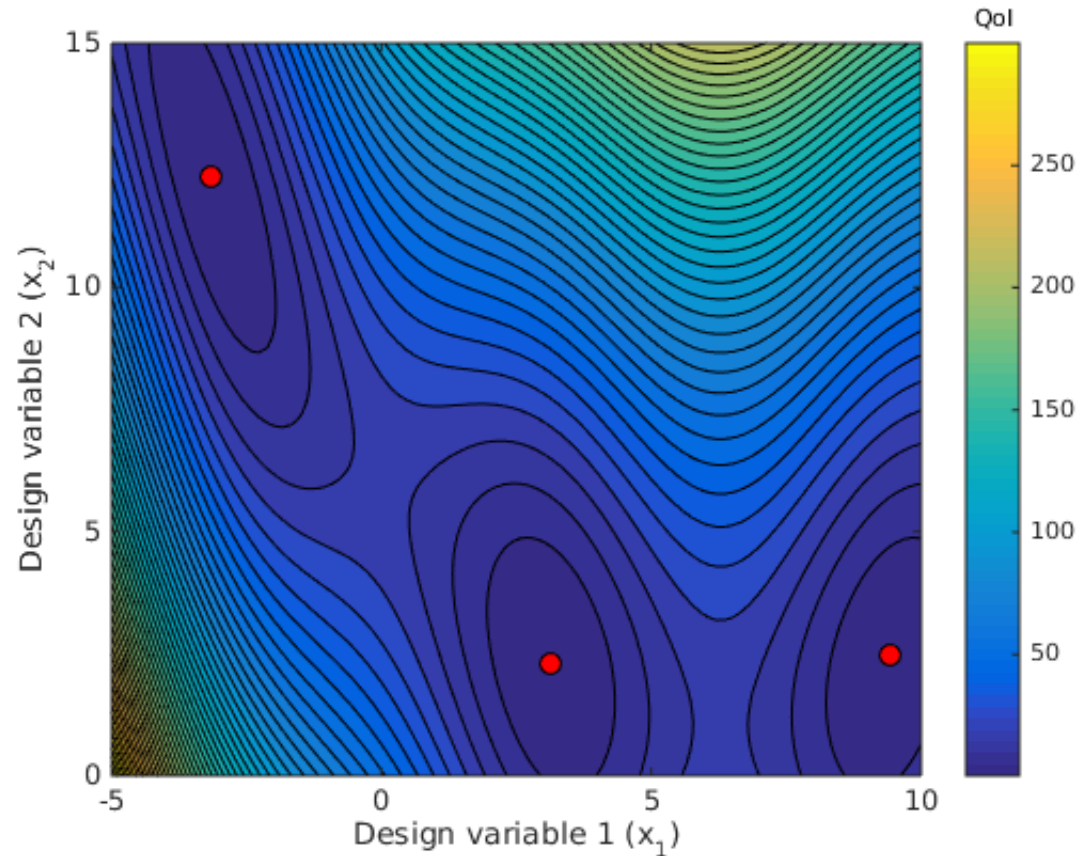
Analytical function – Surface representation



# Working with DAKOTA: Branin function

## The Branin function

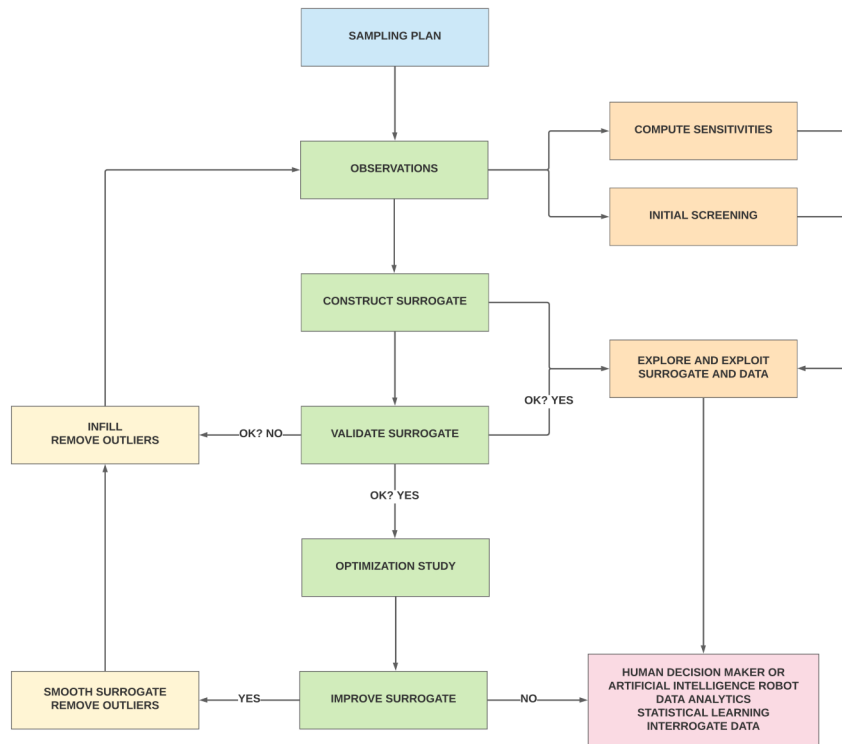
Analytical function – Contour plot and minimum values





# Working with DAKOTA: Branin function

## The Branin function - SBO workflow



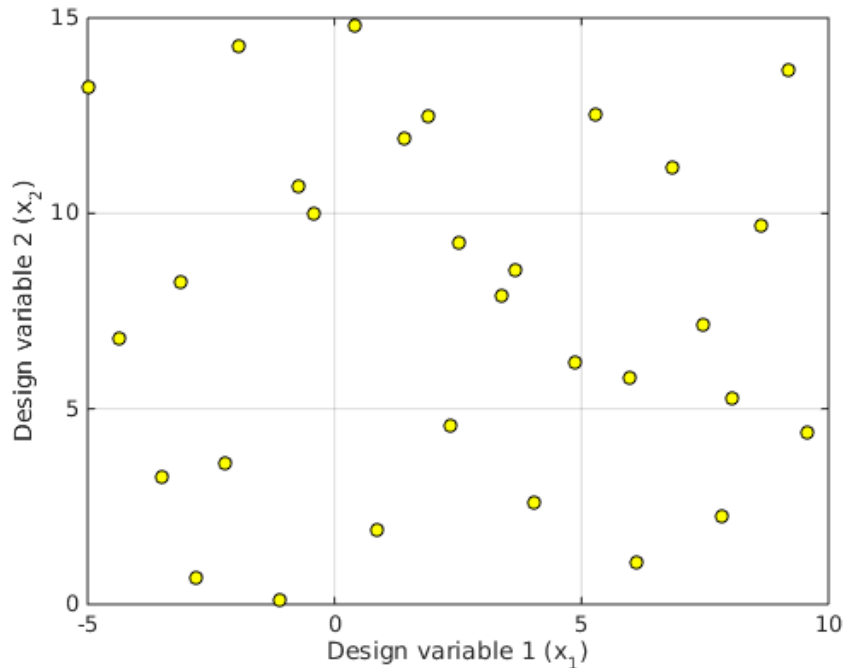
To perform the SBO, we proceed as follows:

- Design an experiment.
- Run high fidelity simulations.
- Construct the surrogate.
  - There are many methods, just to name a few: kriging interpolation (Gaussian process), neural networks, radial basis functions, polynomial functions, least squares and so on.
- Compute initial sensitivities and do initial screening.
- Explore the design space.
- Validate the surrogate.
- Improve the surrogate.
  - This includes training the surrogate, removing outliers and smoothing the surrogate.
- Do the optimization at the surrogate level.
- Visualize to design scenario.

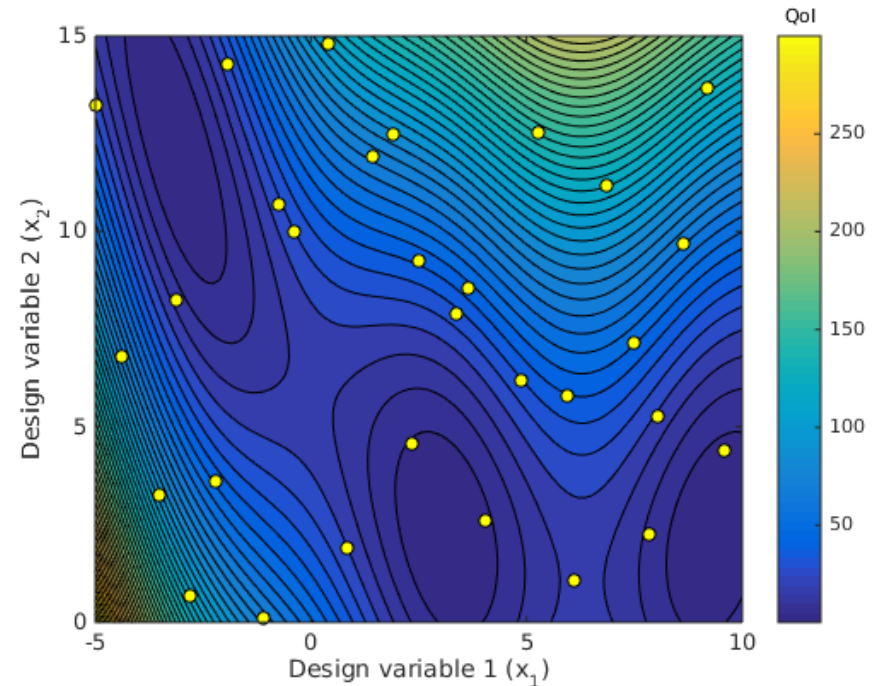
# Working with DAKOTA: Branin function

## The Branin function

DACE experiment



LHS sampling in design space  
(30 experiments)

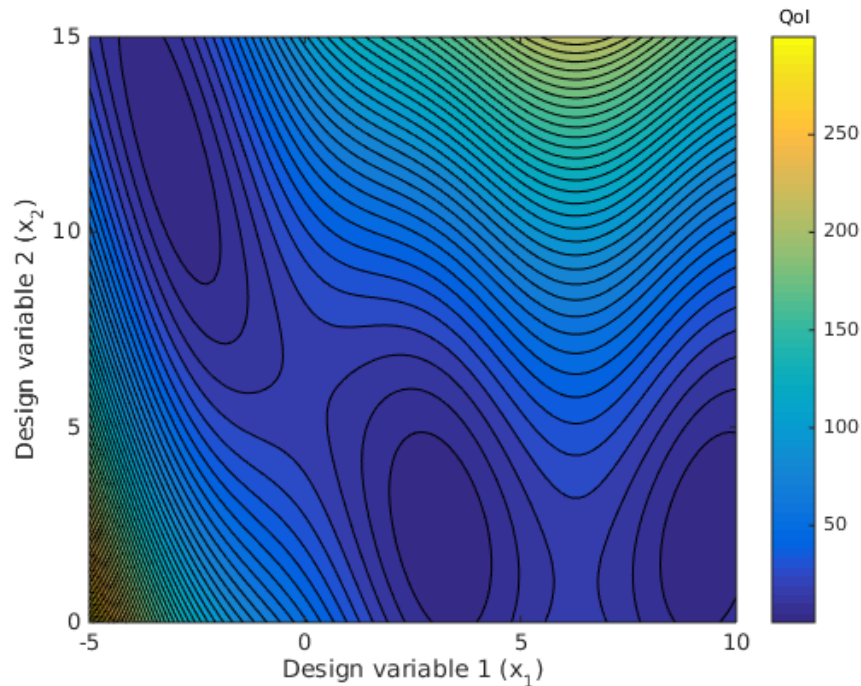


Branin function - Analytical

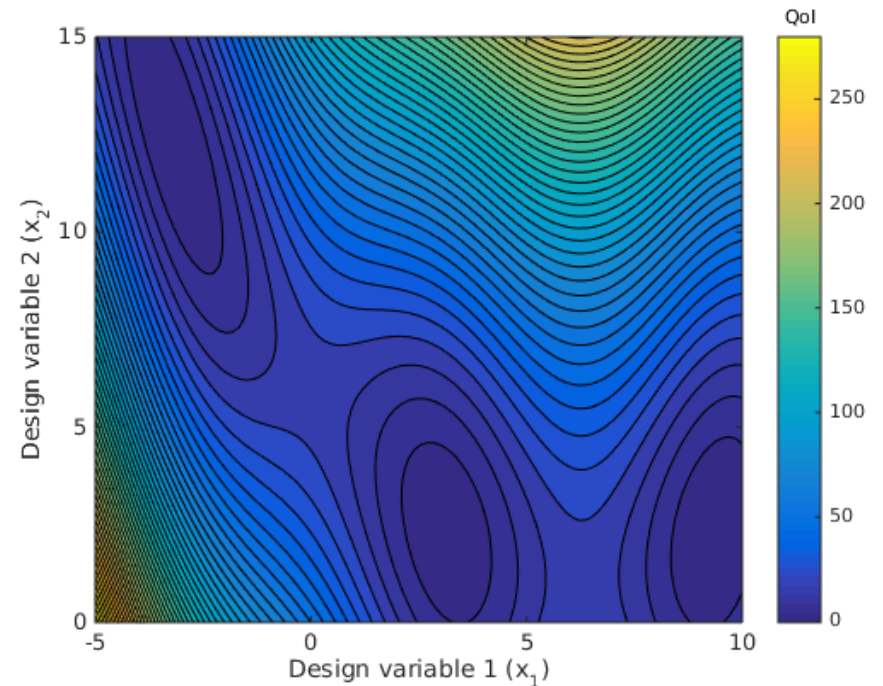
# Working with DAKOTA: Branin function

## The Branin function

Surrogate – Kriging interpolation



Branin function - Analytical

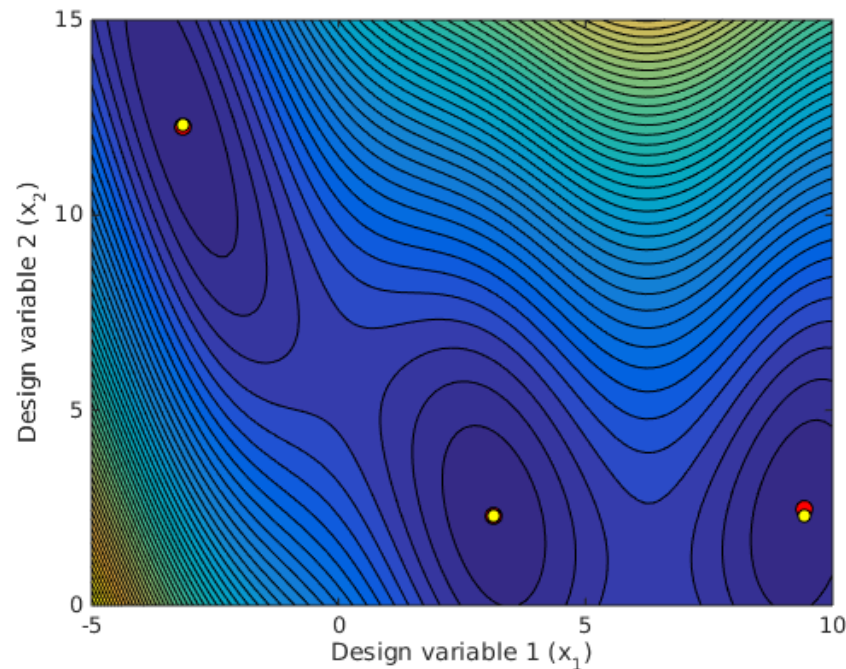


Branin function – Surrogate, meta-model, response surface, you name it.

# Working with DAKOTA: Branin function

## The Branin function

Surrogate based optimization at the surrogate level

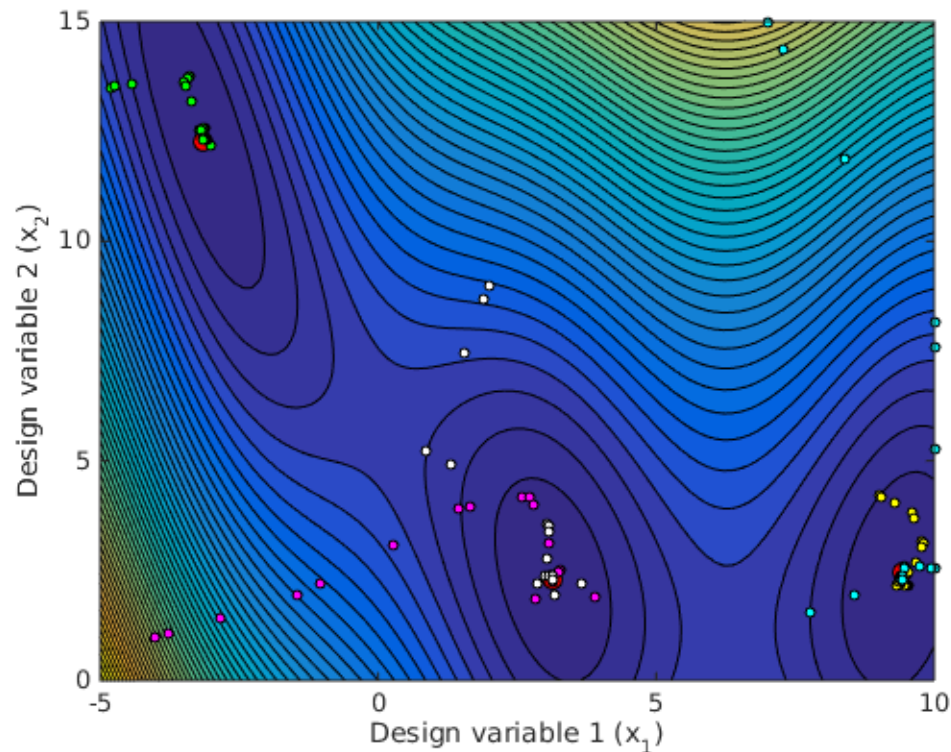


- The red points are global minimum of the analytical Branin function, and the yellow points are the global minimum in the surrogate.
- We conduct constrained gradient-based optimization on the surrogate, for this we use the method of feasible directions (MFD), with multiple starting points (multi-start).
- We choose different initial points because we want to increase the possibilities of finding all the minimum.

# Working with DAKOTA: Branin function

## The Branin function

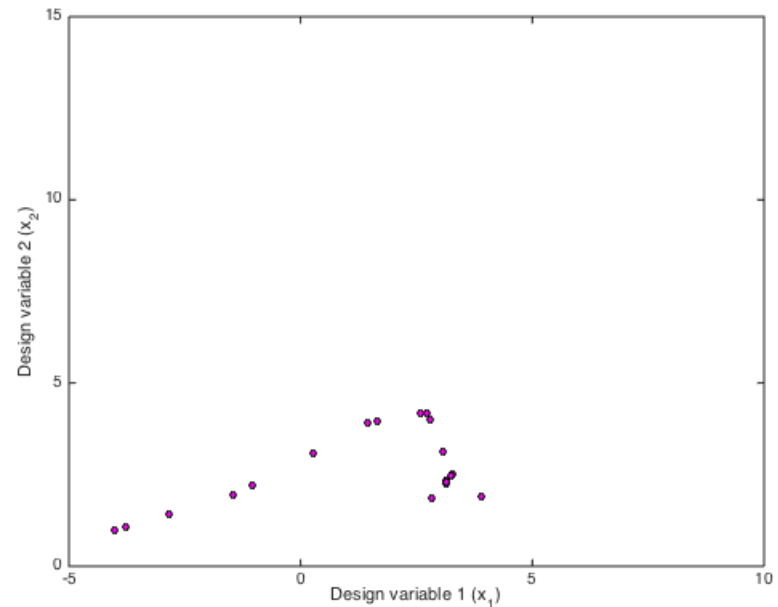
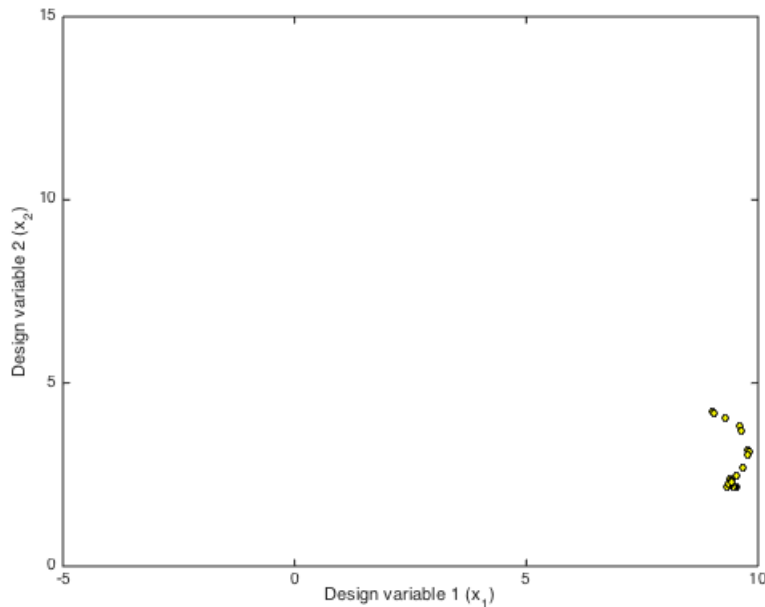
- Surrogate based optimization using the MFD gradient based method.
- Surrogate generated using kriging interpolation.



# Working with DAKOTA: Branin function

## The Branin function

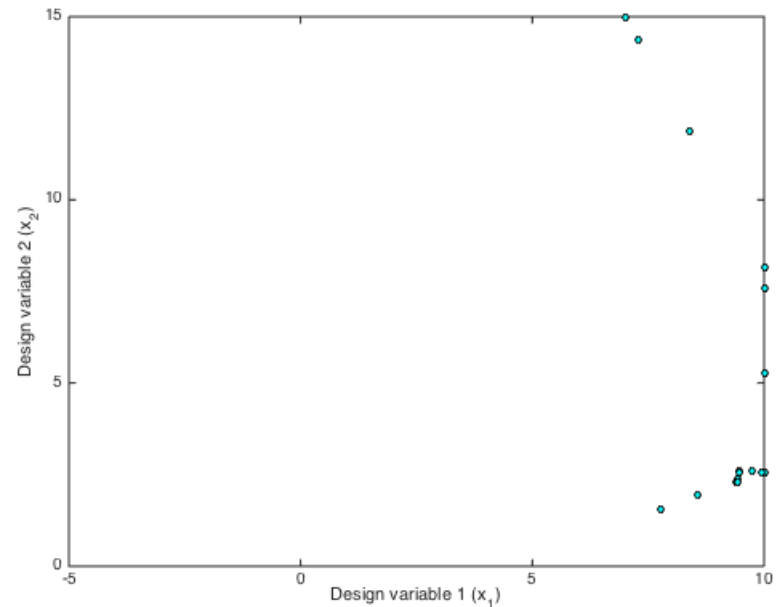
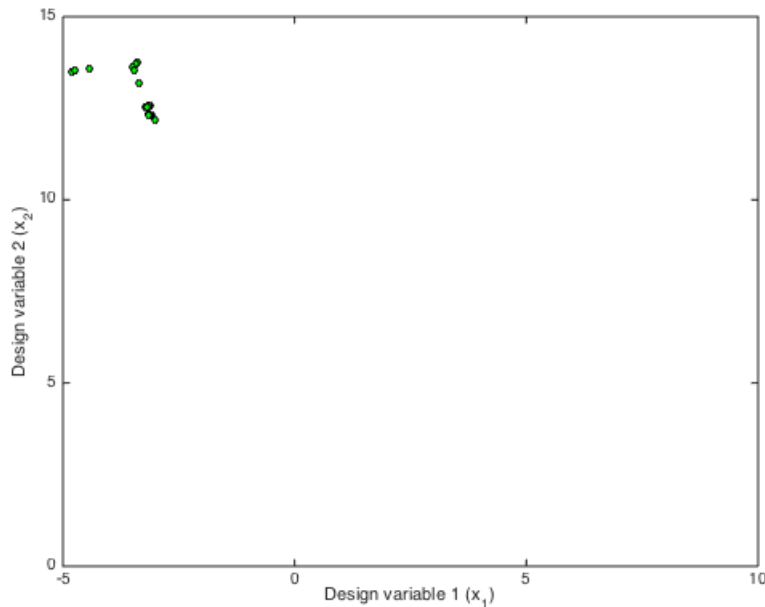
- Optimization using the MFD gradient based method and high-fidelity simulations.
- Maybe we have a very well posed problem, but we do not know the design space.
- We also get different optimal values depending on the starting point.



# Working with DAKOTA: Branin function

## The Branin function

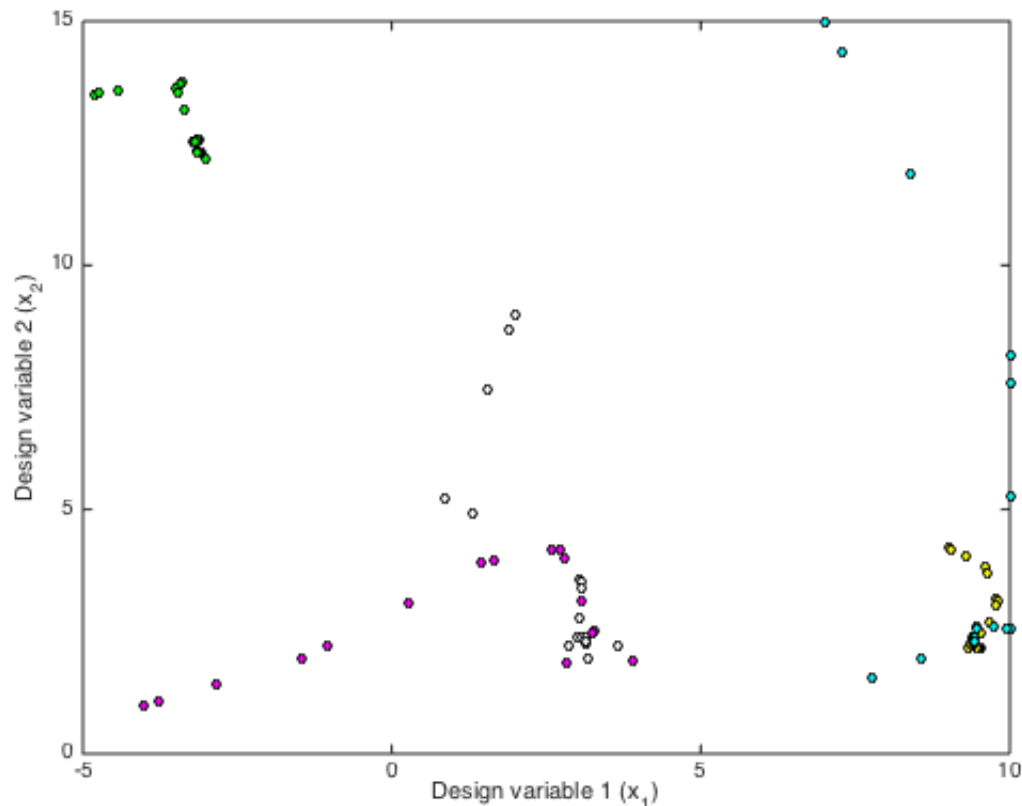
- Optimization using the MFD gradient based method and high-fidelity simulations.
- Maybe we have a very well posed problem, but we do not know the design space.
- We also get different optimal values depending on the starting point.



# Working with DAKOTA: Branin function

## The Branin function

- Optimization using the MFD gradient based method and high-fidelity simulations.
- Maybe we have a very well posed problem, but we do not know the design space.
- We also get different optimal values depending on the starting point.

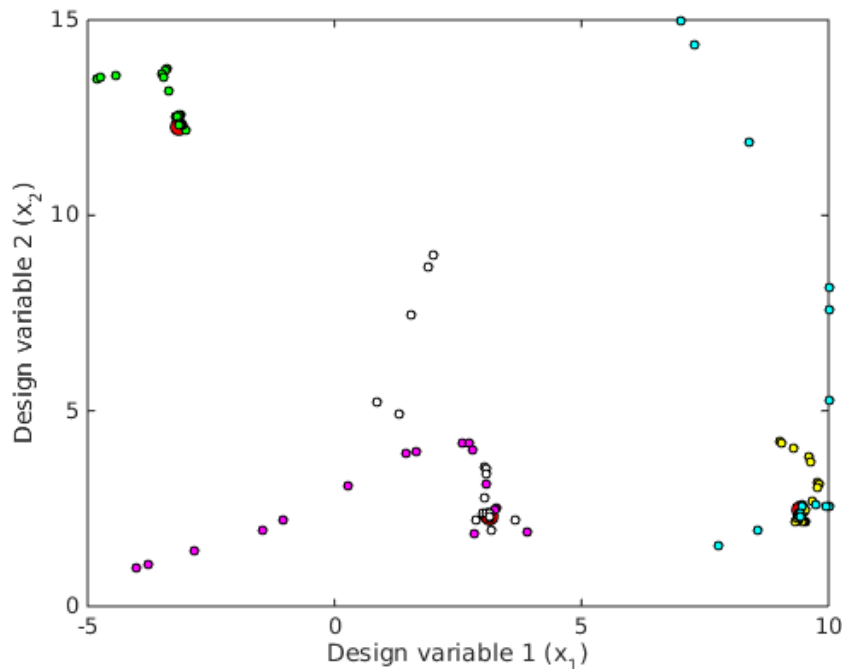




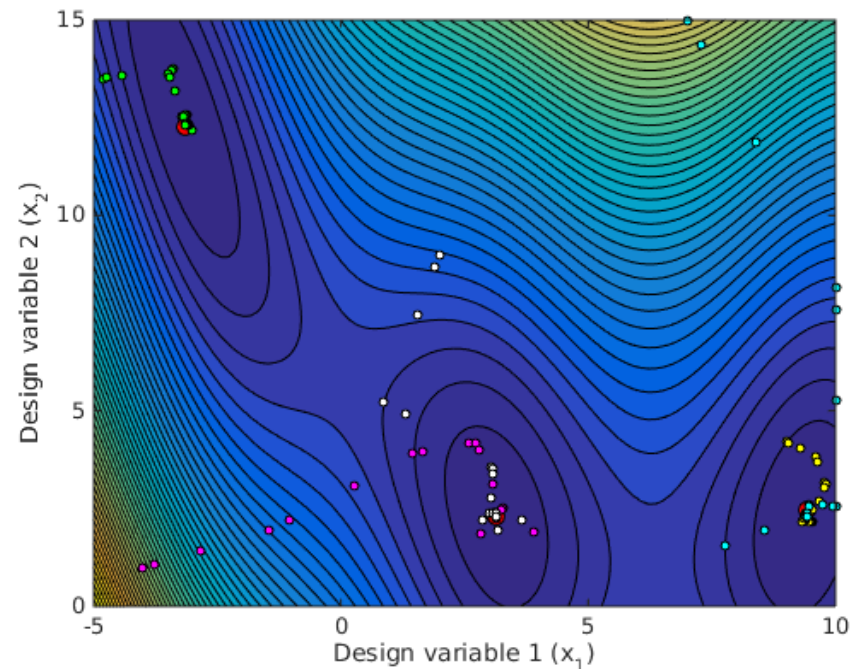
# Working with DAKOTA: Branin function

## The Branin function

- Comparison of optimization using high fidelity simulations and SBO.
- The red points are the global minimum of the multimodal function.
- At the surrogate level we can find optimal and sub-optimal values.



Optimization using high fidelity simulations  
(more than 60 experiments)

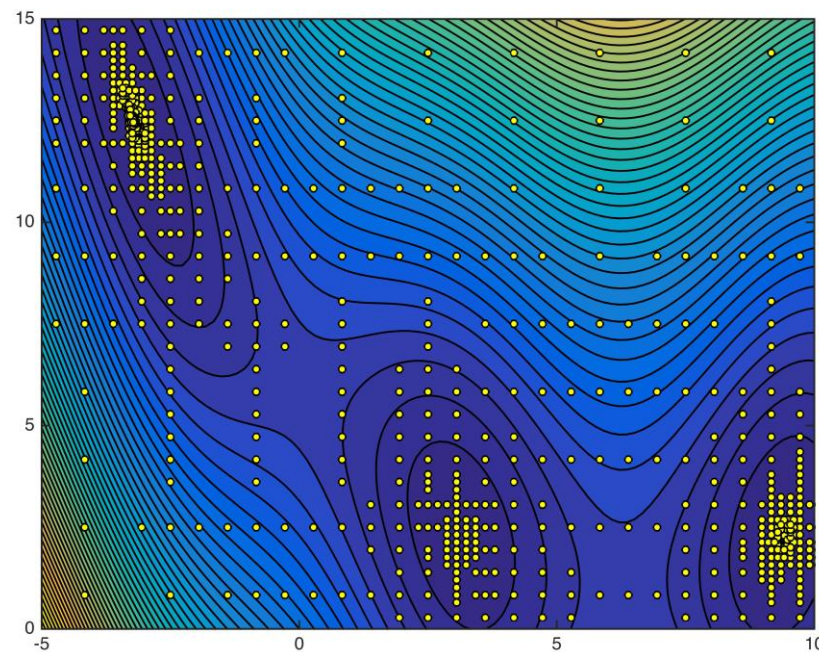


SBO (30 experiments)

# Working with DAKOTA: Branin function

## The Branin function

- This function is highly non-linear and multimodal.
- Local methods will have problems in finding all the optimal points.
- Global methods can find all the optimal points.



Derivative free global method – DIRECT (division of rectangles)  
1000 function evaluations at surrogate level

# Working with DAKOTA: Branin function

## Surrogate based optimization

- So, how many experiments do we need to run to get a good surrogate?
  - One way to determine the number of experiments is to use 10 experiments for each design variable.
  - For example, if you have 5 design variables, you can use 50 experiments.
  - Have in mind that this approach is very conservative.
  - For non-linear problems and in order to better explore the design space, you will need to use much more experiments.

# Working with DAKOTA: Branin function

## Surrogate based optimization

- So, how many experiments do we need to run to get a good surrogate?
  - The following equation will give you a fairly good initial estimate of the number of experiments,

$$(2 \times n + 1)^{(2+\Delta)}$$

where

$$0 \leq \Delta \leq 1$$

$n$  is the number of design variables

$\Delta$  is a correcting factor

- This equation will give you a good initial estimate only for linear and quadratic problems with no non-linear constraint, two QOI's and a maximum of 10 design variables.



# Working with DAKOTA: Branin function

- To illustrate how to setup a SBO case, let us go to the directory:

```
$TM/dakota_sample_cases/model_problems/python3/branin/surrogate/c1
```

# Working with DAKOTA: Branin function

**SBO** – *dakota\_case.in* file



## environment

tabular\_data

tabular\_data\_file = 'table\_out.dat'

method\_pointer = 'MS' ← Identify which method leads the Dakota study

## method

id\_method = 'MS' ← Identifier of the method block.

method\_pointer = 'METHOD\_ON\_SURR' ← Pointer to sub-method to run from each starting point.

multi\_start ← Method We will use a multi-start method.

starting\_points =

9 4.2

-4 1

7 15 ← Multiple starting points for the method

2 9

-4.8 13.5

...

# Working with DAKOTA: Branin function

**SBO** – *dakota\_case.in* file



...

## method

**id\_method = 'METHOD\_ON\_SURR'** ← Identifier of the method block.

**method\_pointer = 'SURR\_MODEL'** ← Identifier for model block to be used by a method.

**output verbose** ← Control how much method information is written to the screen and output file.

**conmin\_mfd** ← Optimization method (method of feasible directions).

...

# Working with DAKOTA: Branin function

**SBO** – *dakota\_case.in* file



## model

...  
**id\_model = 'SURR\_MODEL'** ← Give the model block an identifying name, in case of multiple model blocks.  
**dace\_method\_pointer = 'DACE'** ← Pointer to method to be used to gather training data.  
**surrogate global** ← An empirical model that is created from data or the results of a submodel. Select a surrogate model with global support.  
**samples\_file = 'data1.txt'** ← File containing points to be used to build the surrogate.  
**custom\_annotated header eval\_id** ← Format of the input file  
**gaussian\_process**  
**surpack** ← Use the Surpack version of gaussian process surrogates.  
**export\_model**  
**filename\_prefix = 'my\_surrogate.txt'** ← Exports surrogate model in user-selected format  
...



# Working with DAKOTA: Branin function

**SBO** – *dakota\_case.in* file



...

## method

`id_method = 'DACE'` ← Identifier of the method block.  
`model_pointer = 'DACE_M'` ← Identifier for model block to be used by a method  
`sampling` ← Randomly samples variables according to their distributions.  
`samples 0` As we are reading in a file, we use 0 samples (no need to generate random experiments).

## model

`id_model = 'DACE_M'` ← Identifier of the model block.  
`single`  
`variables_pointer = 'V1'` ← Specify which variables block will be included with this model block  
`interface_pointer = 'I1'` ← Interface block pointer for the single model type  
`responses_pointer = 'R1'` ← Specify which responses block will be used by this model block

...

# Working with DAKOTA: Branin function

**SBO** – *dakota\_case.in* file



```
...

variables
 id_variables = 'V1' ← Identifier of the variables block.
 continuous_design = 2
 upper_bounds -10.0 -15.0
 lower_bounds -5.0 0.0
 cdv_descriptors 'x1' 'x2'

interface
 id_interface = 'I1' ← Identifier of the interface block.
 fork
 analysis_driver 'simulator_script'
```

```
...
```

# Working with DAKOTA: Branin function

**SBO** – *dakota\_case.in* file



...

## responses

**id\_responses = 'R1'** ← Identifier of the responses block.

**num\_objectives\_functions = 1**

**numerical\_gradients**

**method\_source**

**dakota**

**interval\_type**

**forward**

**no\_hessians**

**sense 'min'**

# Working with DAKOTA: Branin function

## SBO fork interface – *simulator\_script* file



```
echo "1" > tmp.txt
```

```
mv tmp.txt $2
```

- We save the numerical value 1 in the file *tmp.txt*., and then we pass this value to the file *results.out*.
- As we are creating a surrogate model using external data, we do not need to interface with an external application, we will use dakota's surrogate library (**surfpack**) to construct the surrogate and evaluate the function internally.
- However, DAKOTA will always ask for the *results.out* file.
- Therefore, we create a dummy file.
- Also, we do not need to use templates files.

# Working with DAKOTA: Branin function

## SBO fork interface – Files and directories that will be used

- Directories:
  - Only the case directory.
- Input files:
  - *dakota\_case.in*
  - *simulator\_script*
  - *data1.txt*  
(this file contains the training points to be used to construct the surrogate, it must be generated a priory)
- Output files:
  - *table\_out.dat*
  - *my\_surrogate.txt.obj\_fn.alg*
  - *my\_surrogate.txt.obj\_fn.sps*



# Working with DAKOTA: Branin function

## How to run this tutorial

- This case is ready to run, go to the directory

```
$> cd $TM/dakota_sample_cases/model_problems/python3/branin/surrogate/c1
```

- To run it, type in the terminal:

```
1. | $> ./dakota_cleanup
2. | $> dakota -i dakota_case.in
```

- At this point, explore the rest of the sub-directories.

# Roadmap

- ~~1. Introduction to optimization methods~~
- ~~2. Choosing an optimization method~~
- ~~3. Optimization loop – The big picture~~
- ~~4. DAKOTA overview~~
- ~~5. Working with DAKOTA: Rosenbrock function~~
- ~~6. Working with DAKOTA: Branin function~~
- 7. Working with DAKOTA: Multi-objective optimization**
- ~~8. Coupling DAKOTA and OpenFOAM: driven cavity case~~
- ~~9. Additional code coupling tutorials~~
- ~~10. Some kind of conclusion~~

# Working with DAKOTA: Multi-objective optimization

- Multi-objective optimization with DAKOTA.
- Optimization case 1. Parabolic function.
- You will find this case in the following directory:

```
$TM/dakota_sample_cases/model_problems/python3/parabolic_function
```



# Working with DAKOTA: Multi-objective optimization

- In this tutorial, we will use two parabolic functions to illustrate the idea behind multi-objective optimization (MOO).
- First, we are going to do gradient based optimization in each individual function.
- Then, we are going to solve the MOO problem using a derivative free method.

# Working with DAKOTA: Multi-objective optimization

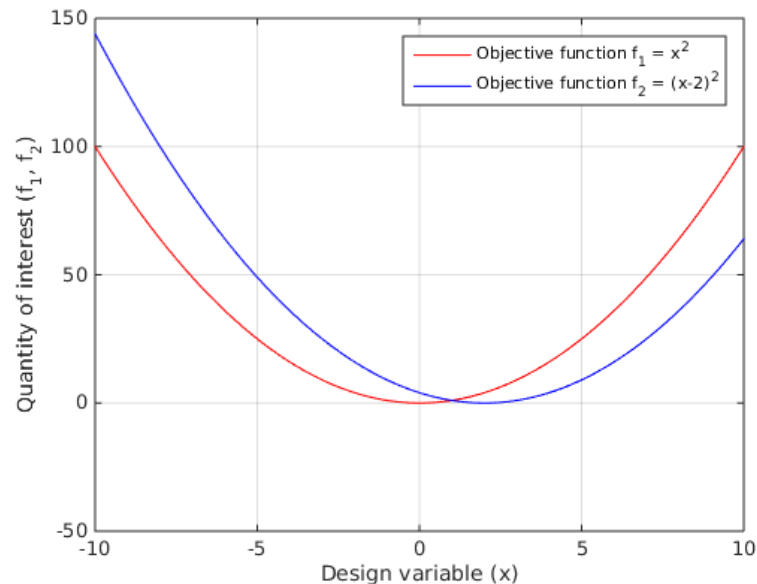
## Case 1. Parabolic function

- Problem definition:

$$\min f_1 = x^2 \quad \text{and} \quad \min f_2 = (x - 2)^2$$

- In the following search domain,

$$-10 < x < 10$$



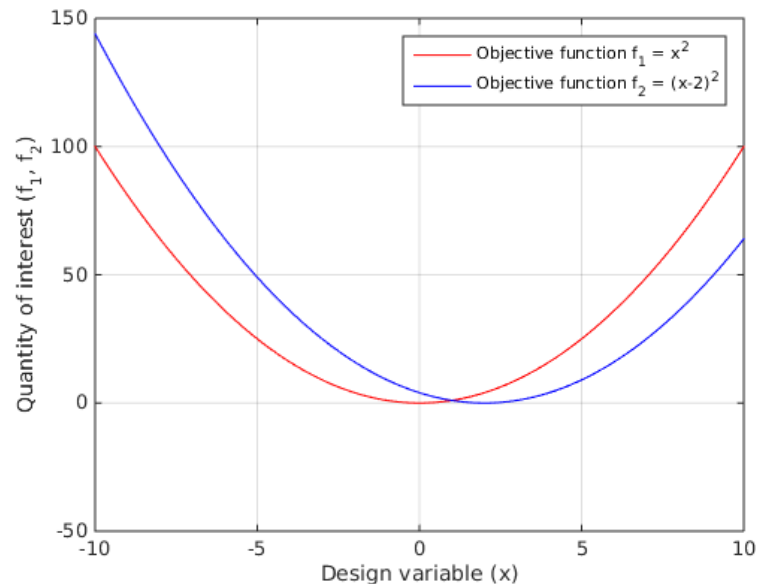
# Working with DAKOTA: Multi-objective optimization

## Case 1. Parabolic function

- If we solve for each function independently (single-objective optimization), we get,

$$\min f_1 = 0 \quad \text{at} \quad x = 0$$

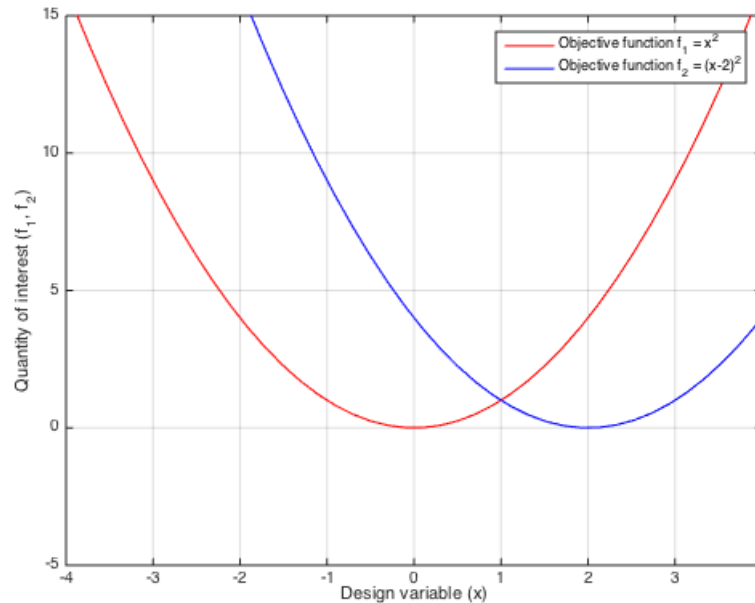
$$\min f_2 = 0 \quad \text{at} \quad x = 2$$



# Working with DAKOTA: Multi-objective optimization

## Case 1. Parabolic function

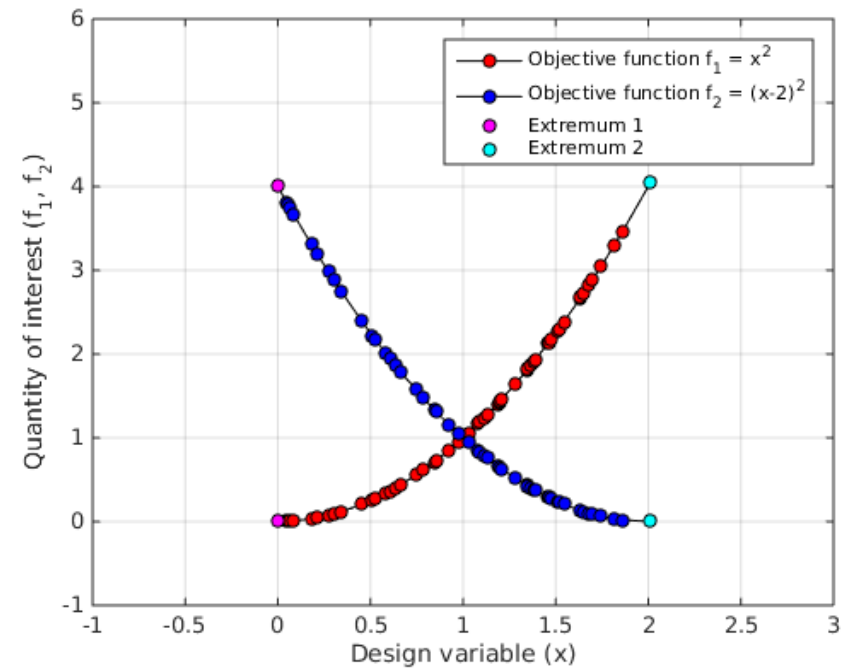
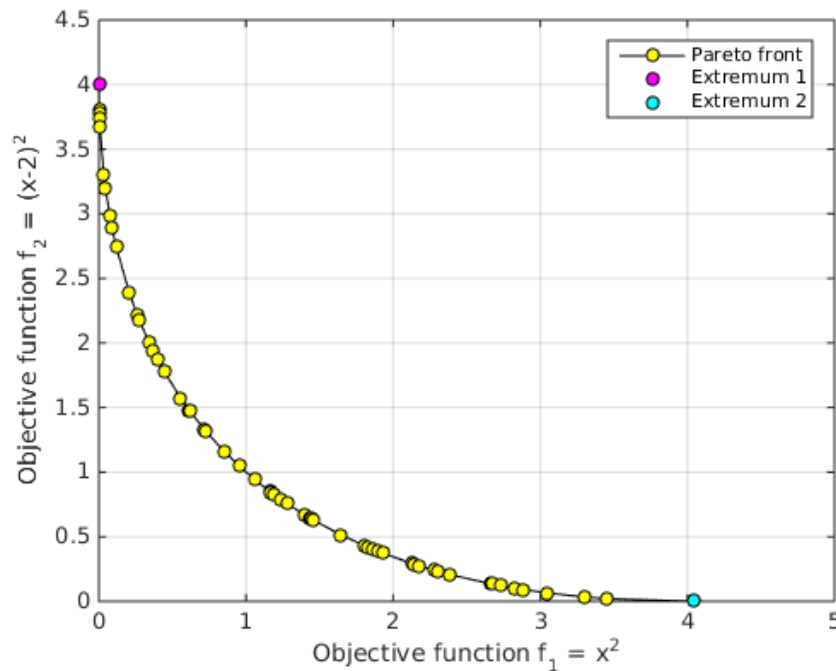
- Single-objective optimization is easy.
- Even easier when we have a well posed problem (bounded, constrained, and smooth problem).
- Let us work with the same problem but this time we will solve both functions at the same time.
- That is, we want to minimize QOI 1 and at the same time we want to minimize QOI 2.
- It is clear that we do not have a single solution.
- Instead, we have a set a solutions, that represents the optimal solutions for a combination of design variables.



# Working with DAKOTA: Multi-objective optimization

## Case 1. Parabolic function

- Multi-objective optimization.
- Pareto front and feasible solutions in the design space.



# Working with DAKOTA: Multi-objective optimization

## Case 1. Parabolic function

- From this point on, please follow me. We are all going to work at the same pace.
- In the directory `$TM/dakota_sample_cases/model_problems/python3/parabolic_function` you will find four directories containing different case setup.
- Let us go to the directory `fork1` which corresponds to the MOO case, and type in the terminal,
  1. `$> ./dakota_cleanup`
  2. `$> dakota -i dakota_case.in`
- At this point, explore the rest of the sub-directories.

# Working with DAKOTA: Multi-objective optimization

## Case 1. Parabolic function

- These are the directories and files that you will find in each directory:
  - Directories:
    - `templatedir`
  - Files:
    - `dakota_case.in`
    - `simulator_script`
    - `templatedir/input.template`
    - `templatedir/p1.py`

# Working with DAKOTA: Multi-objective optimization

- Multi-objective optimization with DAKOTA.
- Optimization case 2. Cone problem.
- You will find this case in the following directory:

```
$TM/dakota_sample_cases/model_problems/python3/cones
```



# Working with DAKOTA: Multi-objective optimization

## Case 2. Cone problem

• Problem definition:

$$\min S \quad \text{and} \quad \min T$$

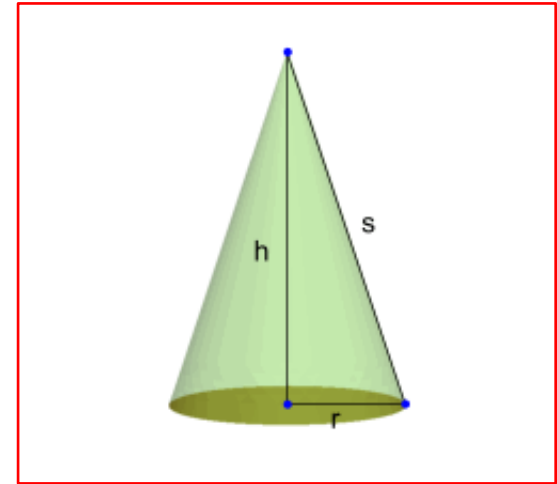
where

$$T = B + S = \pi r(r + s)$$

$$S = \pi r s$$

$$V > 200$$

$$0 < h < 20 \quad 0 < r < 10$$



# Working with DAKOTA: Multi-objective optimization

## Case 2. Cone problem

- And where,

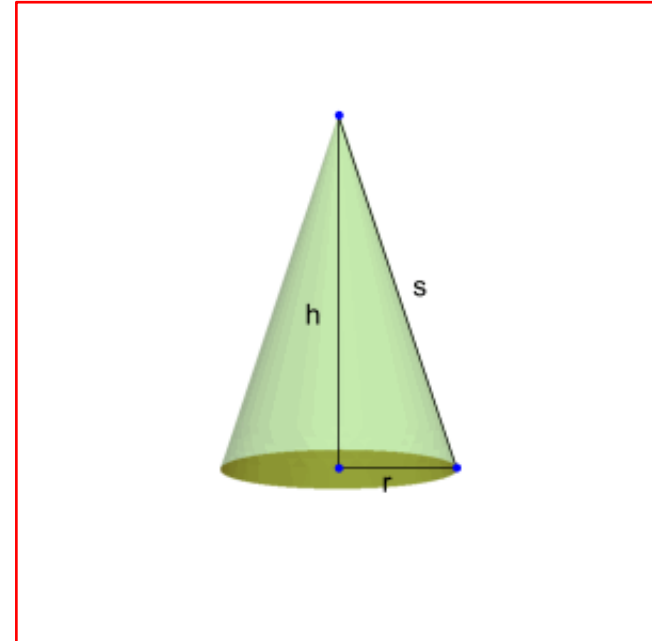
$$V = \frac{\pi}{3} r^2 h \quad (\text{Volume})$$

$$B = \pi r^2 \quad (\text{Base area})$$

$$S = \pi r s \quad (\text{Lateral surface area})$$

$$T = B + S = \pi r(r + s) \quad (\text{Total area})$$

$$s = \sqrt{r^2 + h^2} \quad (\text{Slant height})$$

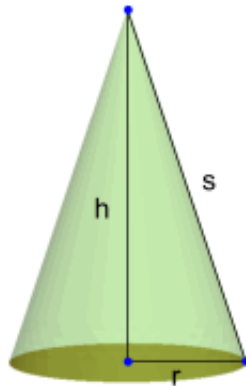


- And  $h$  is the height and  $r$  is the base radius.
- If you want, consider all the linear dimensions in meters (pretty much it does not matter what units do you use).

# Working with DAKOTA: Multi-objective optimization

## Case 2. Cone problem

- Let us study this problem from two different points of view:
  - Single-objective optimization problem. We are interested in finding one minima.
  - Multi-objective optimization problem. There is no single solution, the solution is a set of optimal solutions (pareto front or non-dominated solutions).



Goals:

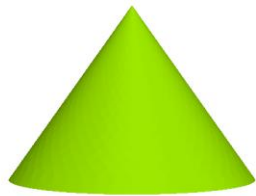
$$\min S$$

$$\min T$$

# Working with DAKOTA: Multi-objective optimization

## Case 2. Cone problem

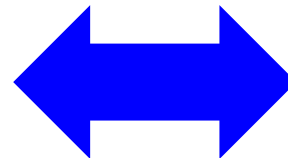
- Single-objective optimization problem.
- Each design represents the optimum solution for its corresponding single-objective optimization problem.
- But, what about the designs between both scenarios?



$\min S$

- $r = 5.131$
- $h = 7.254$
- $V = 200$
- $S = 143.23$
- $T = 225.935$

?



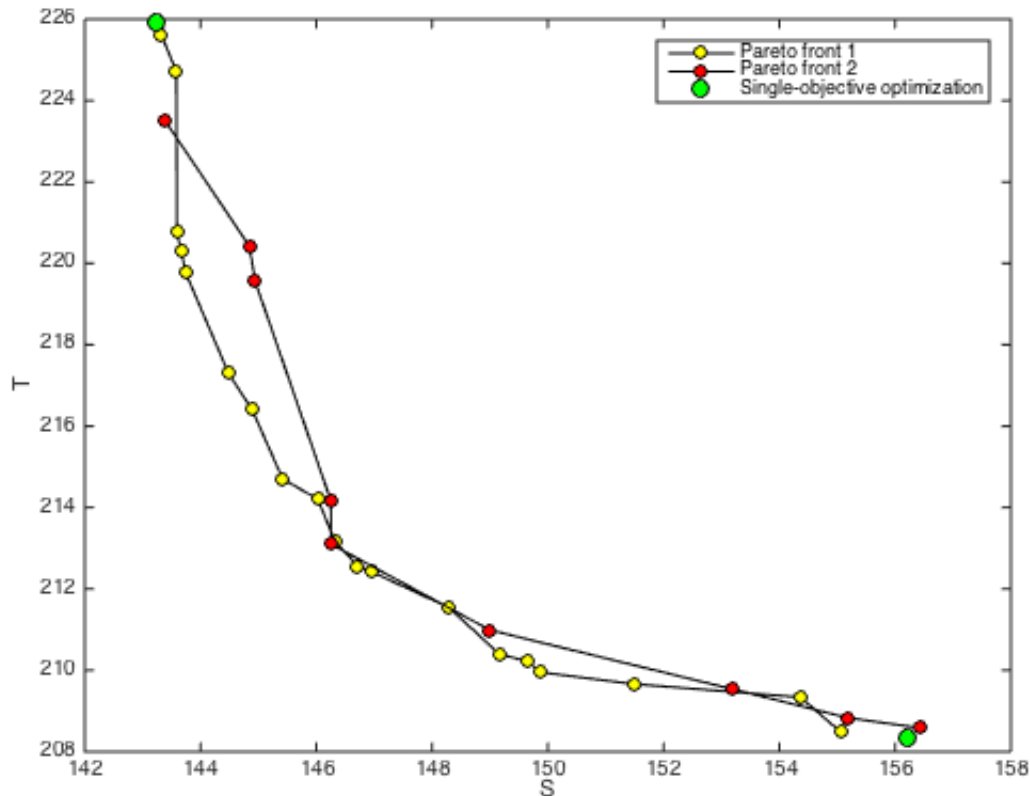
$\min T$

- $r = 4.073$
- $h = 11.51$
- $V = 200$
- $S = 156.227$
- $T = 208.345$

# Working with DAKOTA: Multi-objective optimization

## Case 2. Cone problem

- Multi-objective optimization problem.
- All the non-dominated solutions are represented by the Pareto frontier.



# Working with DAKOTA: Multi-objective optimization

## Case 2. Cone problem

- From this point on, please follow me. We are all going to work at the same pace.
- We are going to conduct two cases in the following order:
  - Multi-objective optimization using the MOGA method (directory `fork1`).
  - Single-objective optimization using a gradient based method (directory `fork2`).
- These are the directories and files that will be used
  - Directories:
    - `templatedir`
  - Files:
    - `dakota_case.in`
    - `simulator_script`
    - `templatedir/input.template`
    - `templatedir/p1.py`

# Working with DAKOTA: Multi-objective optimization

## Case 2. Cone problem

- To run this case, go to the directory:

```
$> cd $TM/dakota_sample_cases/model_problems/python3/cones
```

- Go to the desired sub-directory and type in the terminal

```
1. | $> ./dakota_cleanup
2. | $> dakota -i dakota_case.in
```

- At this point, explore the rest of the sub-directories.

# Roadmap

- ~~1. Introduction to optimization methods~~
- ~~2. Choosing an optimization method~~
- ~~3. Optimization loop – The big picture~~
- ~~4. DAKOTA overview~~
- ~~5. Working with DAKOTA: Rosenbrock function~~
- ~~6. Working with DAKOTA: Branin function~~
- ~~7. Working with DAKOTA: Multi-objective optimization~~
- 8. Coupling DAKOTA and OpenFOAM: driven cavity case**
- ~~9. Additional code coupling tutorials~~
- ~~10. Some kind of conclusion~~



# Coupling DAKOTA and OpenFOAM: driven cavity case

- Coupling DAKOTA and OpenFOAM.
- The driven cavity case.
- You will find this tutorial in the following directory:

```
$TM/dakota_openfoam_coupling/cavity
```

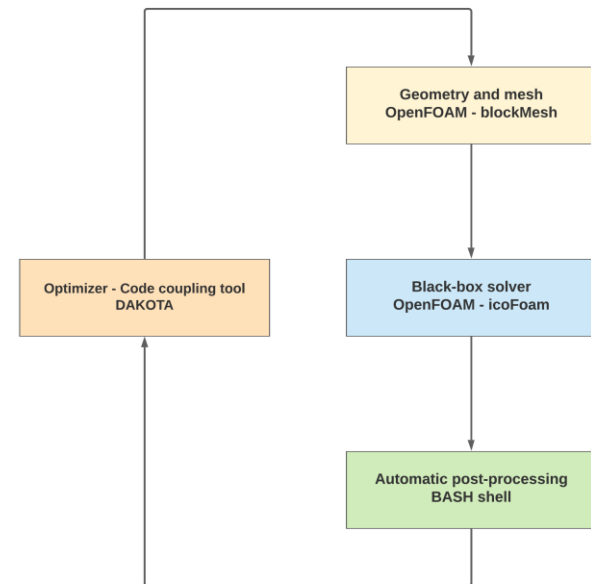
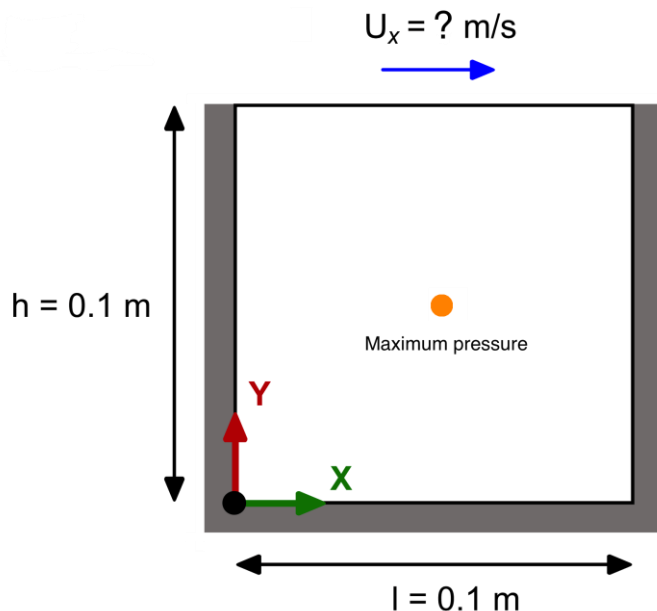
# Coupling DAKOTA and OpenFOAM: driven cavity case

- In this tutorial, we will couple DAKOTA with OpenFOAM.
- Hereafter, we demonstrate DAKOTA code coupling, parallel asynchronous execution, and optimization capabilities.
- We are assuming that we all know how to use OpenFOAM.
- Therefore, unless strictly necessary we will not go into details on how to run OpenFOAM or how setup a case.

# Coupling DAKOTA and OpenFOAM: driven cavity case

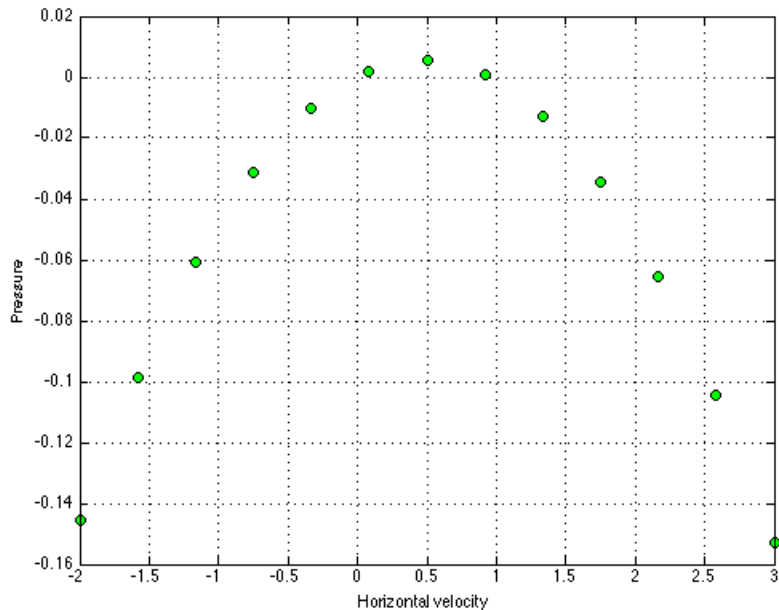
## Driven cavity optimization

- In this tutorial we conduct a parametric study, a bounded-unconstrained gradient optimization, nonlinear least squares study (calibration), a genetic algorithm optimization and SBO.
- The design variable is velocity, and the objective function is the pressure measured at a point located in the middle of the cavity.
- We aim at finding the optimal velocity (going to the left or to the right), to obtain the maximum pressure at the center of the cavity.

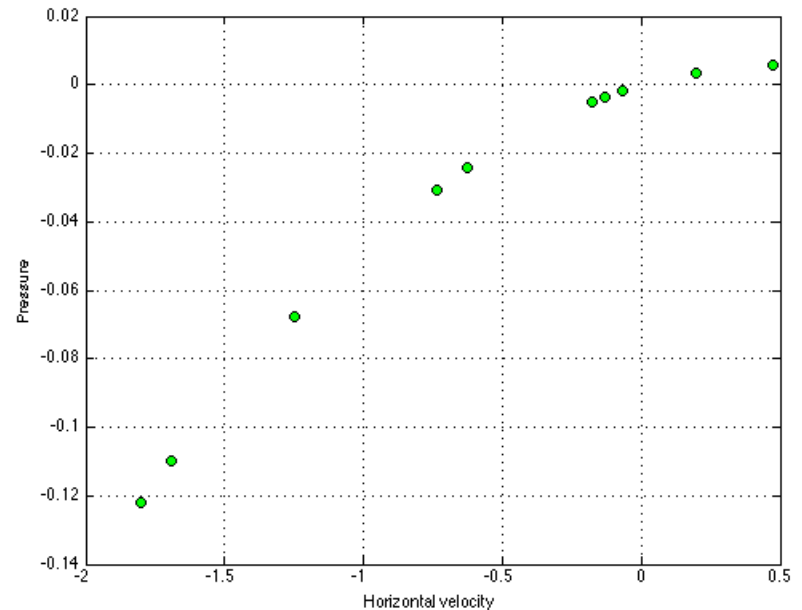


# Coupling DAKOTA and OpenFOAM: driven cavity case

## Driven cavity optimization



Parametric study



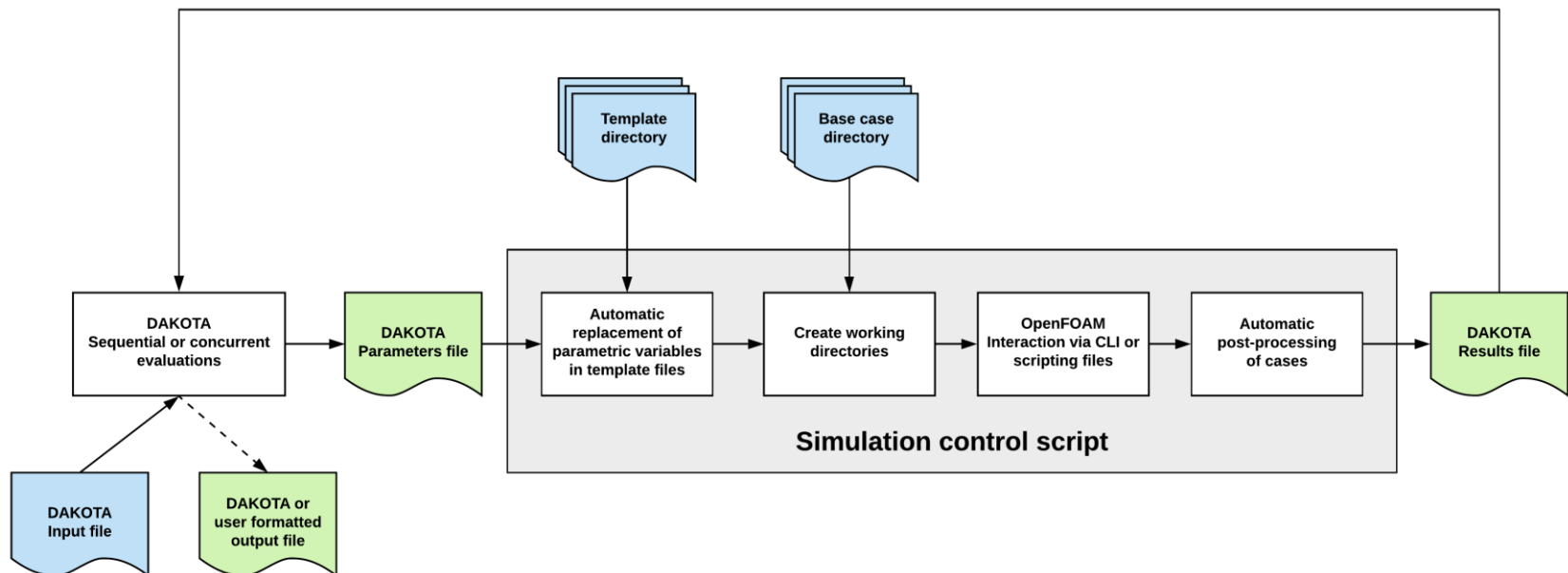
High fidelity gradient-based optimization  
(CONMIN – FRCG)

# Coupling DAKOTA and OpenFOAM: driven cavity case

- We are going to conduct five studies in the following order:
  - Parametric study.
  - Bounded-unconstrained gradient optimization.
  - Nonlinear least squares study (calibration).
  - Genetic algorithm optimization.
  - SBO.
- From this point on, please follow me. We are all going to work at the same pace.
- But first let us recall the fork interface directory structure.

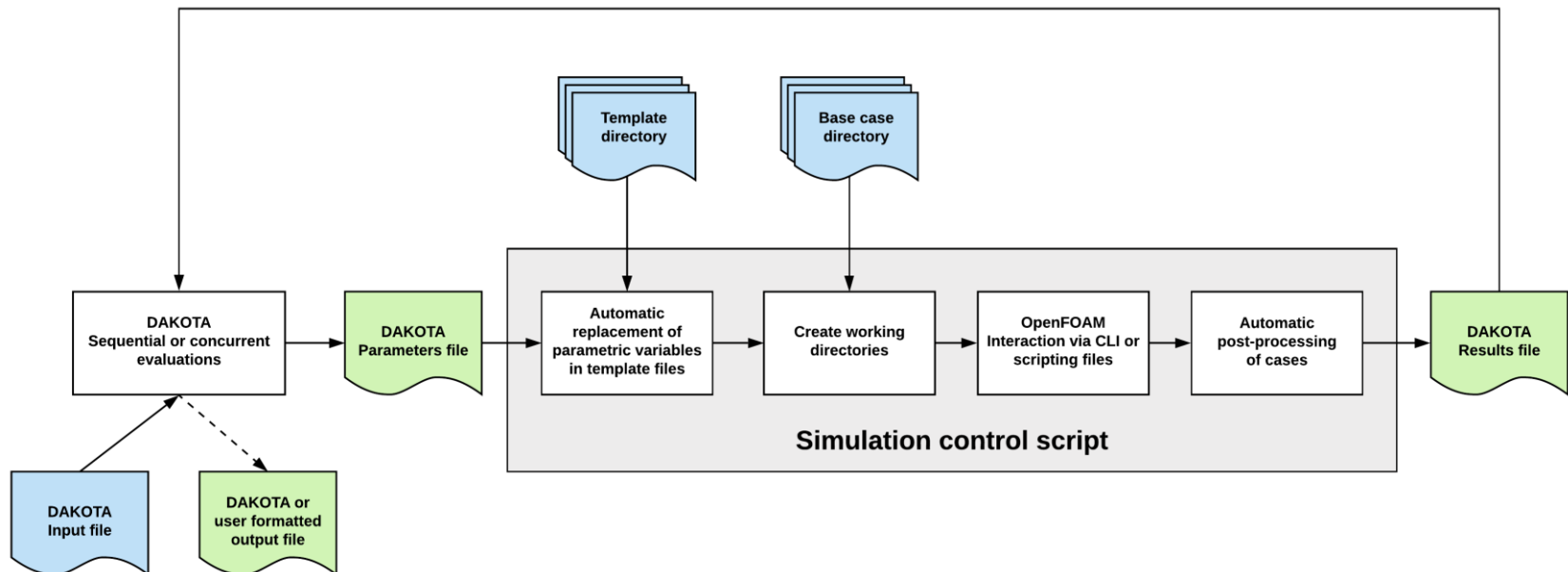
# Coupling DAKOTA and OpenFOAM: driven cavity case

- Workflow for data exchange between DAKOTA and OpenFOAM.
  - The white rectangles denote process blocks.
  - The light-shaded blue document symbols denote unchanging sets of files,
  - The light-shaded green document symbols indicate files that change with each set of design parameters generated by DAKOTA or after the end of the evaluation of the QOI.
  - The light-shaded grey area denotes the domain of the control script that automatically prepares the case.
    - This includes, CAD geometry, mesh generation, launching the solver, quantitative and qualitative post-processing, and automatic formatting of input and output files.



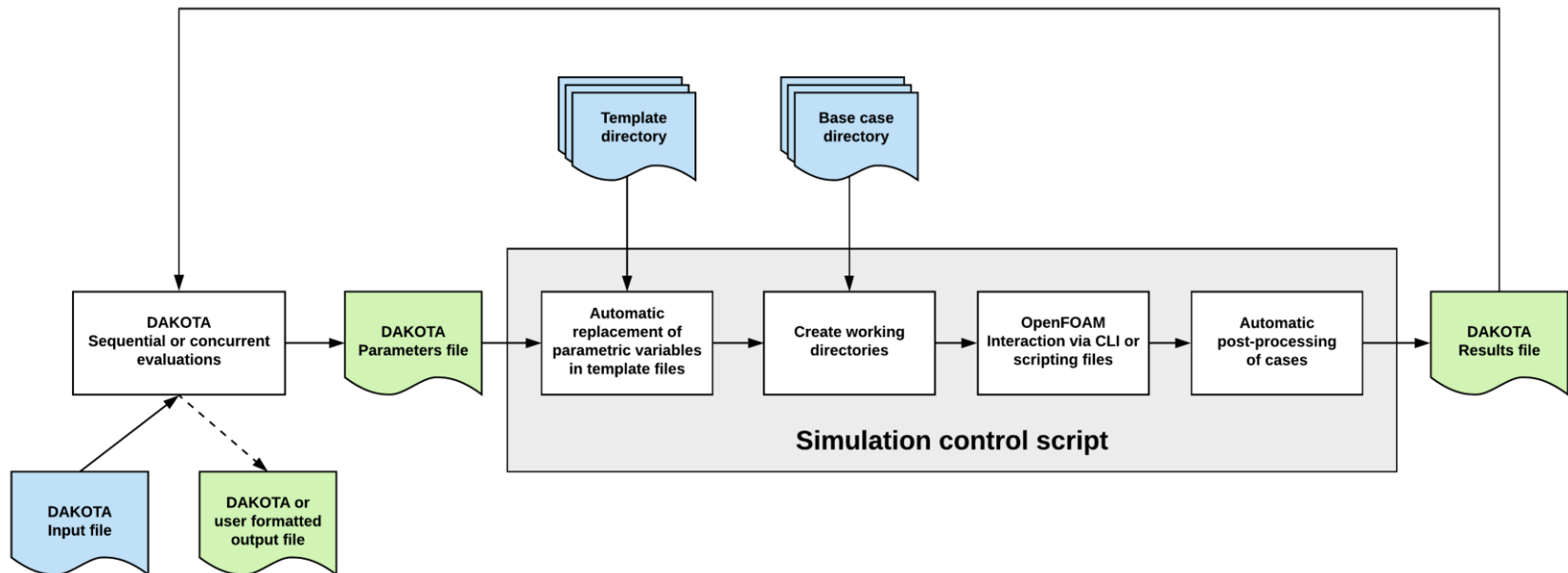
# Coupling DAKOTA and OpenFOAM: driven cavity case

- Workflow for data exchange between DAKOTA and OpenFOAM.
  - The first input is **DAKOTA input file**, where the problem is defined.
  - A **Template directory** is created to store the parametrical input files, *i.e.*, subject to change as a result of the optimization process (*e.g.*, files containing the definition of the geometry, boundary conditions, physical properties, and so on).
  - The automatic update of the parametrical files located in the **Template directory** is done automatically by using a DAKOTA supplied utility or user-defined scripts.
    - These utilities skim all files located in the **Template directory** and automatically insert the values generated by DAKOTA, into the predefined locations in the template files.



# Coupling DAKOTA and OpenFOAM: driven cavity case

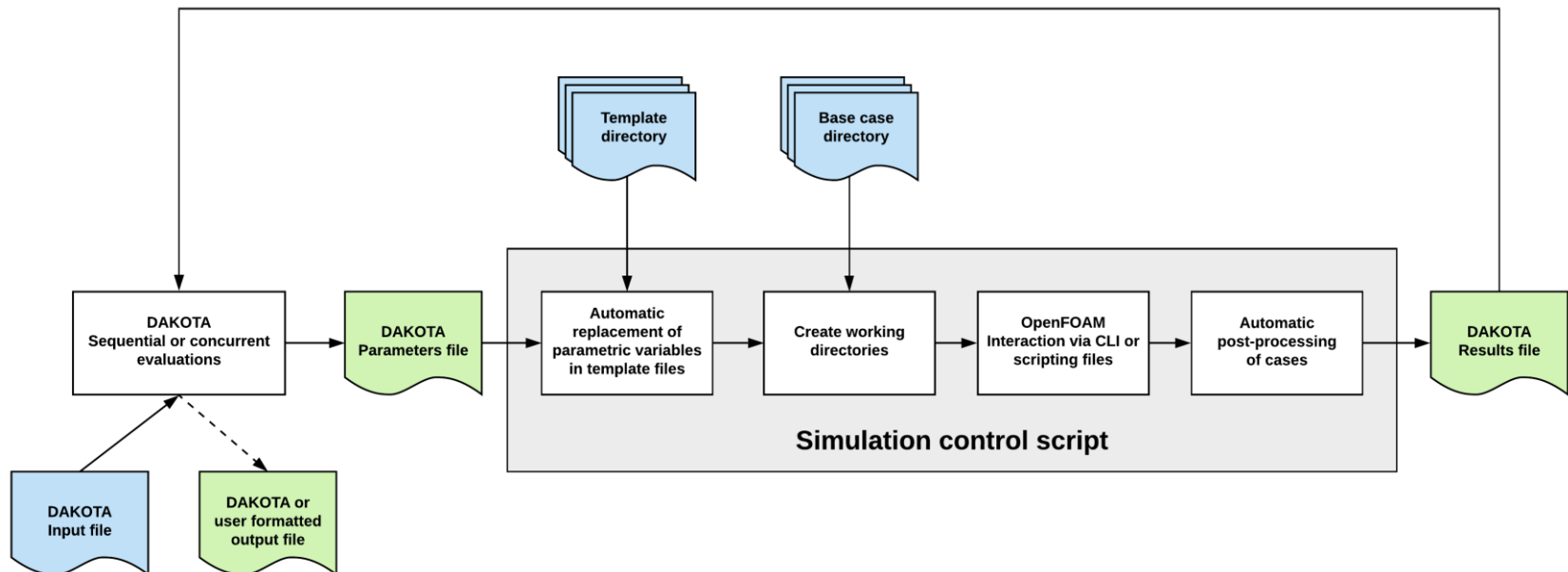
- Workflow for data exchange between DAKOTA and OpenFOAM.
  - A **Base case directory** is also created, where all the files needed to run the OpenFOAM simulations are stored.
  - The **Simulation control script** file (or simulation driver), merges the automatically edited files in the **Template directory** with the **Base case directory**, creating in this way a **working directory** for a specific set of design parameters.
  - At this point, the **Simulation control script** executes all the steps related to the simulation, *i.e.*, geometry update, meshing, and launching the solver (in serial or parallel).





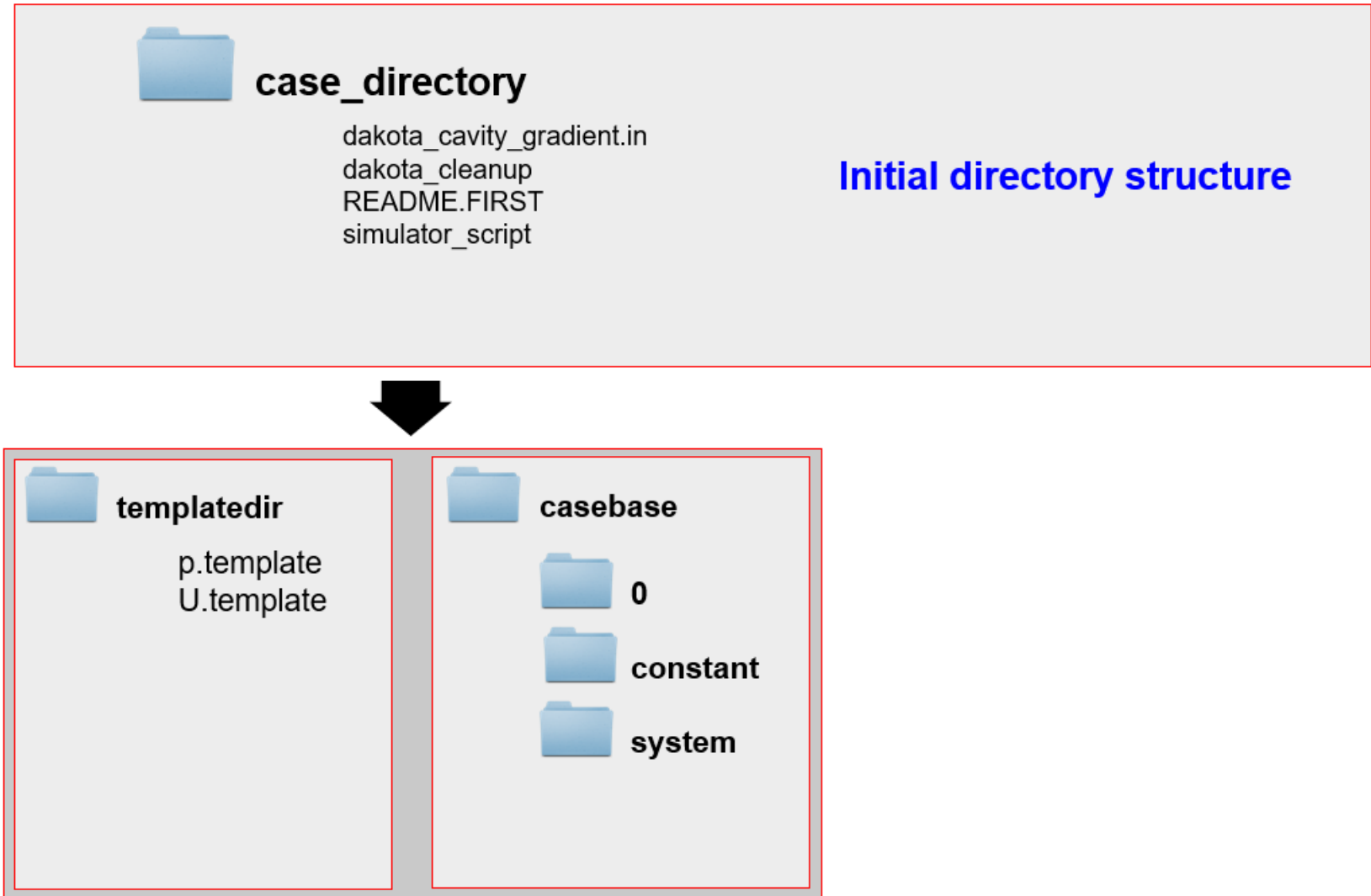
# Coupling DAKOTA and OpenFOAM: driven cavity case

- Workflow for data exchange between DAKOTA and OpenFOAM.
  - All the data generated is automatically post-processed following the instructions defined in the **Simulation control script**.
  - It is important to mention that the output of OpenFOAM is converted into **DAKOTA Results file** following the instructions defined in the **Simulation control script**.
    - This instructions are encoded by the user.
  - It should be emphasized that the **Template directory** and **Base case directory** are created by the user. And he **Base case directory** contains a working OpenFOAM case.



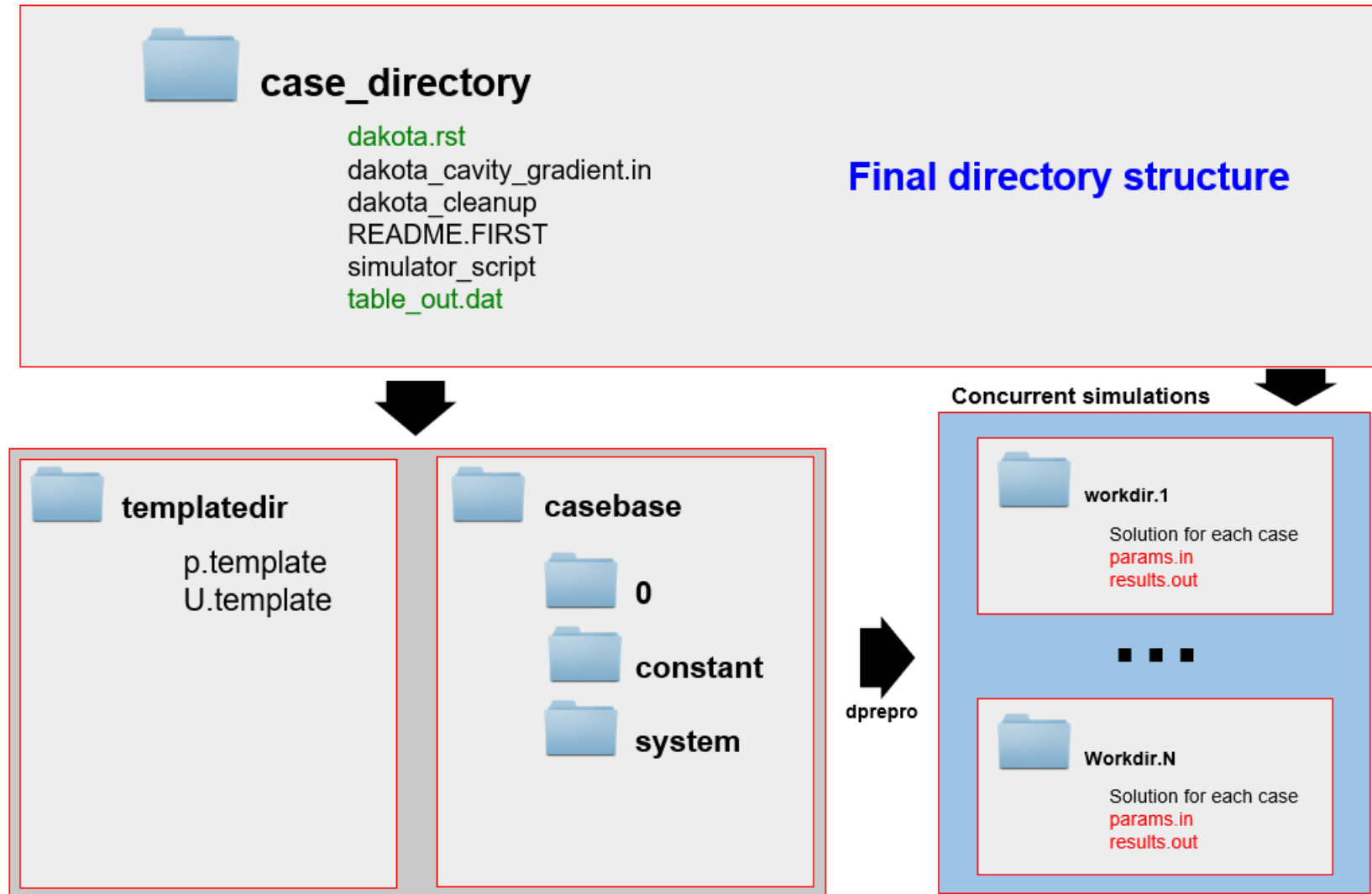
# Coupling DAKOTA and OpenFOAM: driven cavity case

## Fork interface directory structure



# Coupling DAKOTA and OpenFOAM: driven cavity case

## Fork interface directory structure



# Coupling DAKOTA and OpenFOAM: driven cavity case

## Applications needed to run this case

- To run this case, you need the following applications,
  - DAKOTA.
  - OpenFOAM.
  - Bash utilities.

# Coupling DAKOTA and OpenFOAM: driven cavity case

**These are the directories and files that will be used**

- Directories:
  - **casebase**
  - **templatedir**
- Files:
  - *dakota\_case.in* (DAKOTA's input file)
  - *simulator\_script*
  - *templatedir/p.template*
  - *templatedir/U.template*

# Coupling DAKOTA and OpenFOAM: driven cavity case

## How to run this tutorial

- To run the case, go to the case directory

```
$> cd $TM/dakota_openfoam_coupling/cavity
```

- Go to the desired sub-directory (**cavity\_gradient** for instance) and type in the terminal

```
1. $> ./dakota_cleanup
```

```
2. $> dakota -i dakota_case.in
```

- At this point, explore the rest of the sub-directories.

# Roadmap

- ~~1. Introduction to optimization methods~~
- ~~2. Choosing an optimization method~~
- ~~3. Optimization loop – The big picture~~
- ~~4. DAKOTA overview~~
- ~~5. Working with DAKOTA: Rosenbrock function~~
- ~~6. Working with DAKOTA: Branin function~~
- ~~7. Working with DAKOTA: Multi-objective optimization~~
- ~~8. Coupling DAKOTA and OpenFOAM: driven cavity case~~
- 9. Additional code coupling tutorials**
10. Some kind of conclusion

# Additional code coupling tutorials

- Geometry parameterization.
- You will find this case in the following directory:

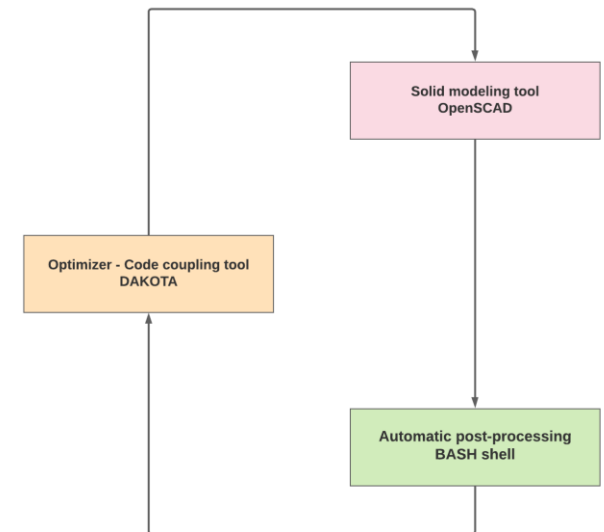
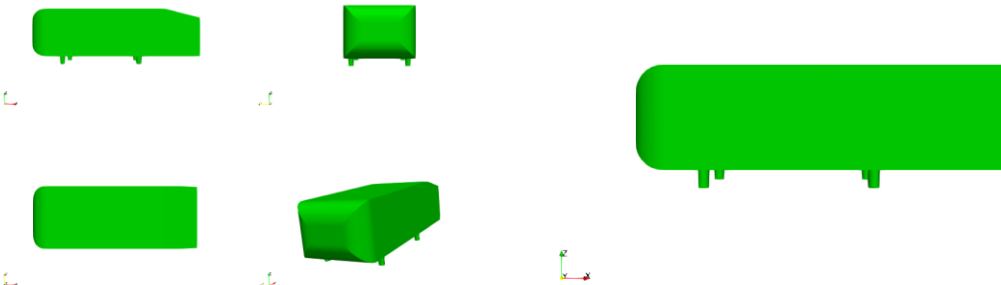
```
$TM/dakota_openfoam_coupling/ahmed_OPENSCAD/geo
```



# Additional code coupling tutorials

## Ahmed body

- In this case we parameterize the geometry using openscad.
- The design variable is the slant angle.
- This geometry can be used as input file for snappyHexMesh or cfMesh.



<http://www.wolfodynamics.com/training/opt/image7.gif>

# Additional code coupling tutorials

## Applications needed to run this case

- To run this case, you need the following applications,
  - DAKOTA.
  - OpenFOAM.
  - OpenSCAD.
  - Bash utilities.

# Additional code coupling tutorials

**These are the directories and files that will be used**

- From this point on, please follow me.
- We are all going to work at the same pace.
- Directories:
  - **casebase**
  - **templatedir**
- Files:
  - *dakota\_case.in*
  - *simulator\_script*
  - *templatedir/geo.template*
  - *templatedir/input.template*

# Additional code coupling tutorials

## How to run this tutorial

- This case is ready to run.
- To run it, go to the case directory:

```
$> cd $TM/dakota_openfoam_coupling/ahmed_OPENSCAD/geo
```

- Inside the directory **geo** type in the terminal

1. 

```
$> ./dakota_cleanup
```
2. 

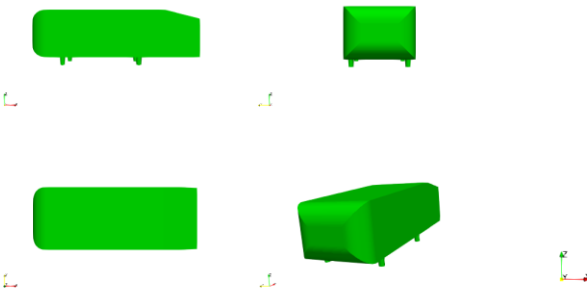
```
$> dakota -i dakota_case.in
```

- **If you want to take the extra step, you can do the actual optimization.**
- **The case is ready to run.**
- **Have in mind that running the case (meshing and simulation) is computationally intensive.**

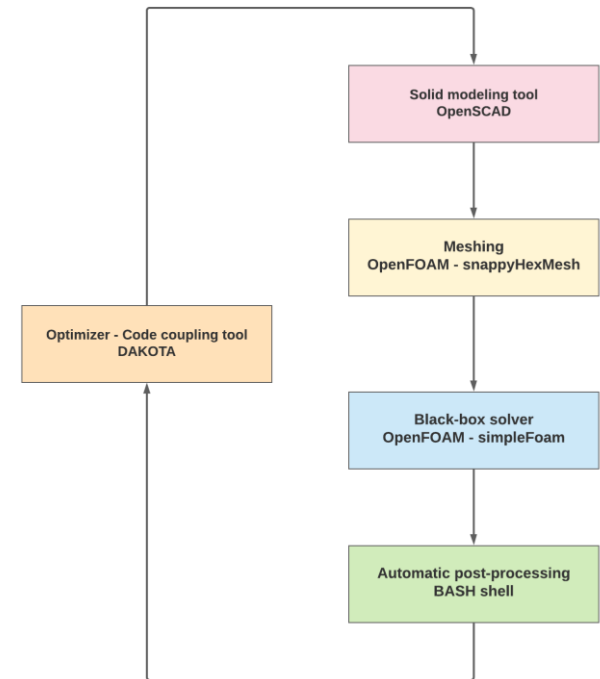
# Additional code coupling tutorials

## Ahmed body

- In this case we aim at optimizing the ahmed body.
- The design variable is the slant angle, and the objective function is the drag coefficient.
- You can conduct a parametric study, a constrained gradient optimization and/or a surrogate based optimization (SBO).



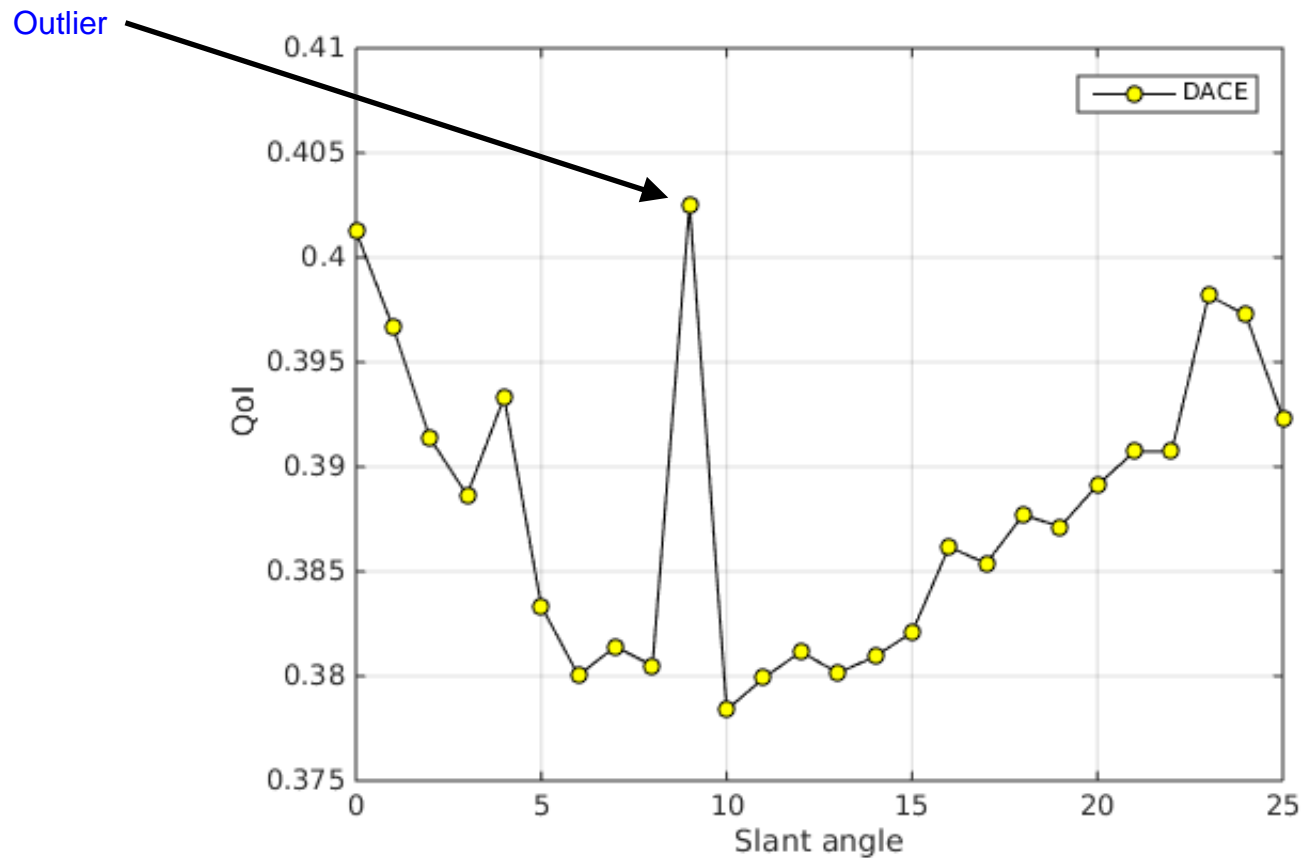
<http://www.wolfodynamics.com/training/opt/image7.gif>



# Additional code coupling tutorials

## Ahmed body

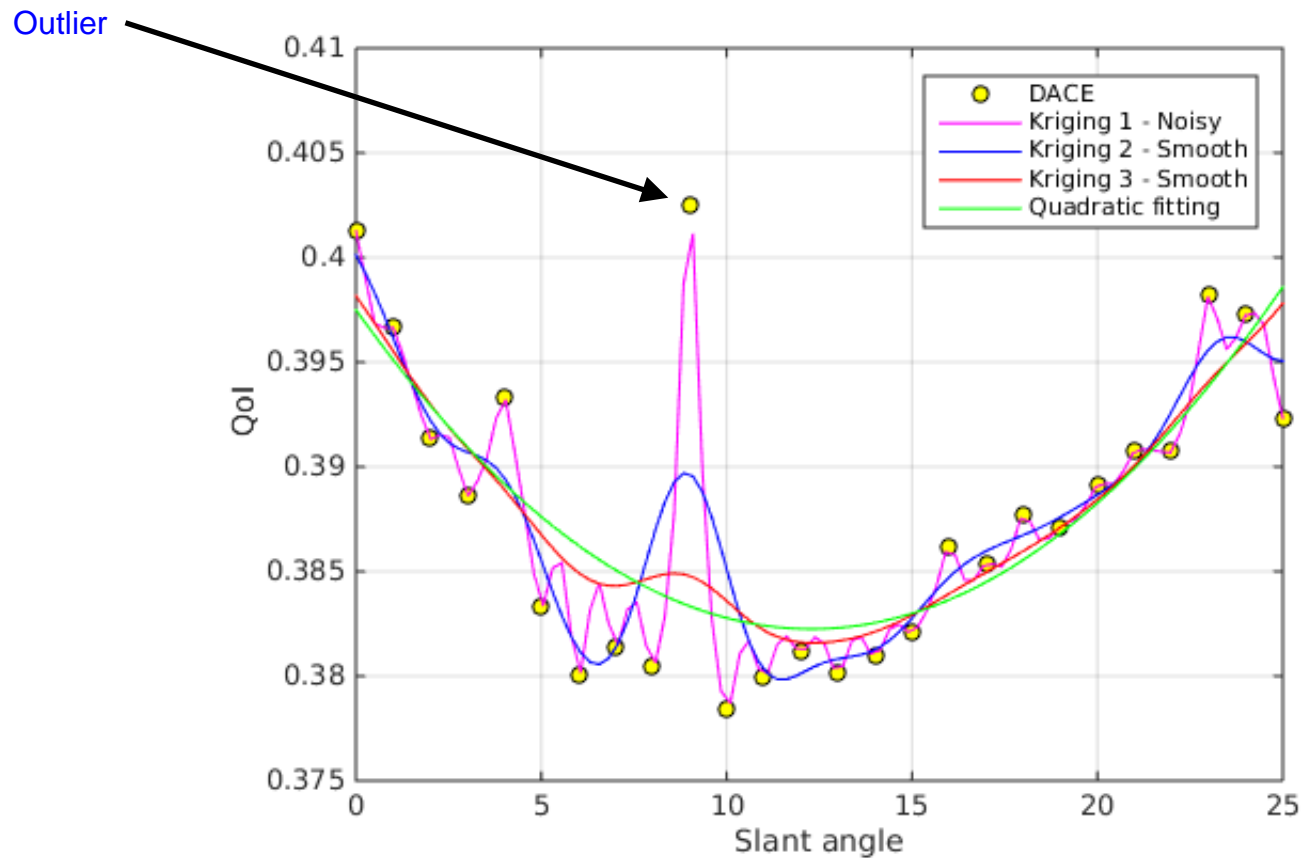
Parametric study



# Additional code coupling tutorials

## Ahmed body

Surrogate, meta-model or response surface.

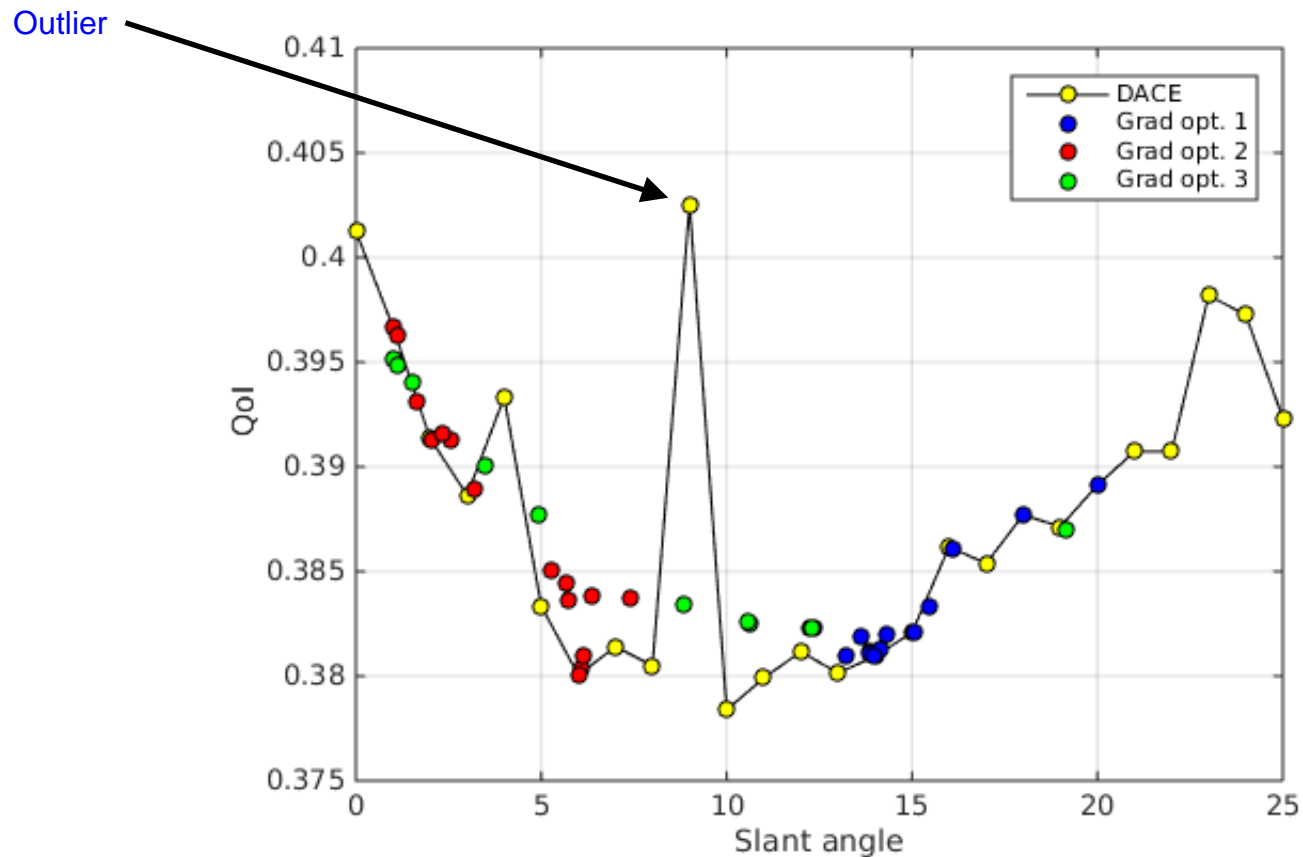




# Additional code coupling tutorials

## Ahmed body

SBO on the surrogate model built using Kriging interpolation.  
Optimization method: MFD.



# Additional code coupling tutorials

**These are the directories and files that will be used**

- Directories:
  - **casebase**
  - **templatedir**
- Files:
  - *dakota\_case.in*
  - *simulator\_script*
  - *templatedir/input.template*
  - *templatedir/run\_simulation.template*

# Additional code coupling tutorials

## How to run this tutorial

- This case is ready to run.
- To run it, go to the case directory:

```
$> cd $TM/dakota_openfoam_coupling/ahmed_OPENSCAD/multi_OF
```

- Go to the desired sub-directory and type in the terminal

1. 

```
$> ./dakota_cleanup
```
2. 

```
$> dakota -i dakota_case.in
```

# Additional code coupling tutorials

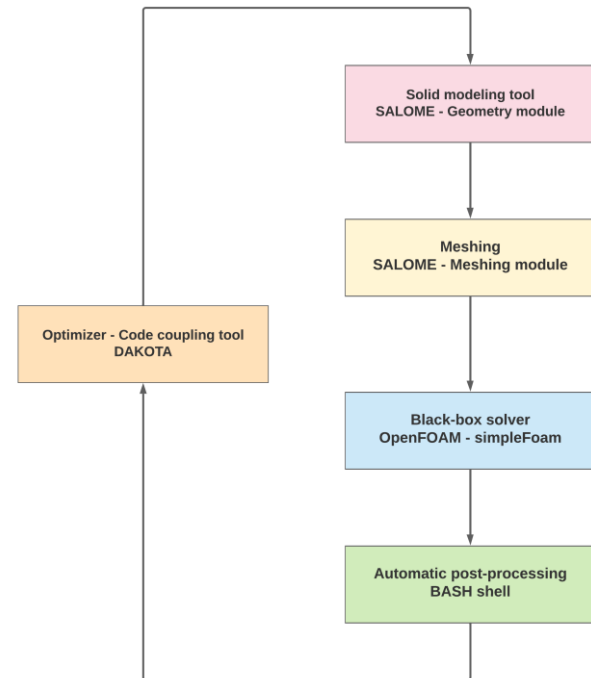
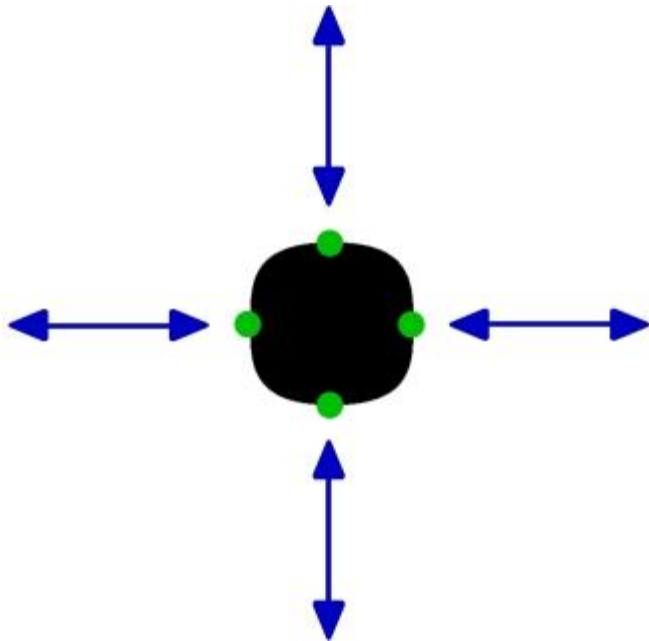
- Blunt body shape optimization.
- You will find this case in the following directory:

```
$TM/dakota_openfoam_coupling/blunt_body_SALOME
```

# Additional code coupling tutorials

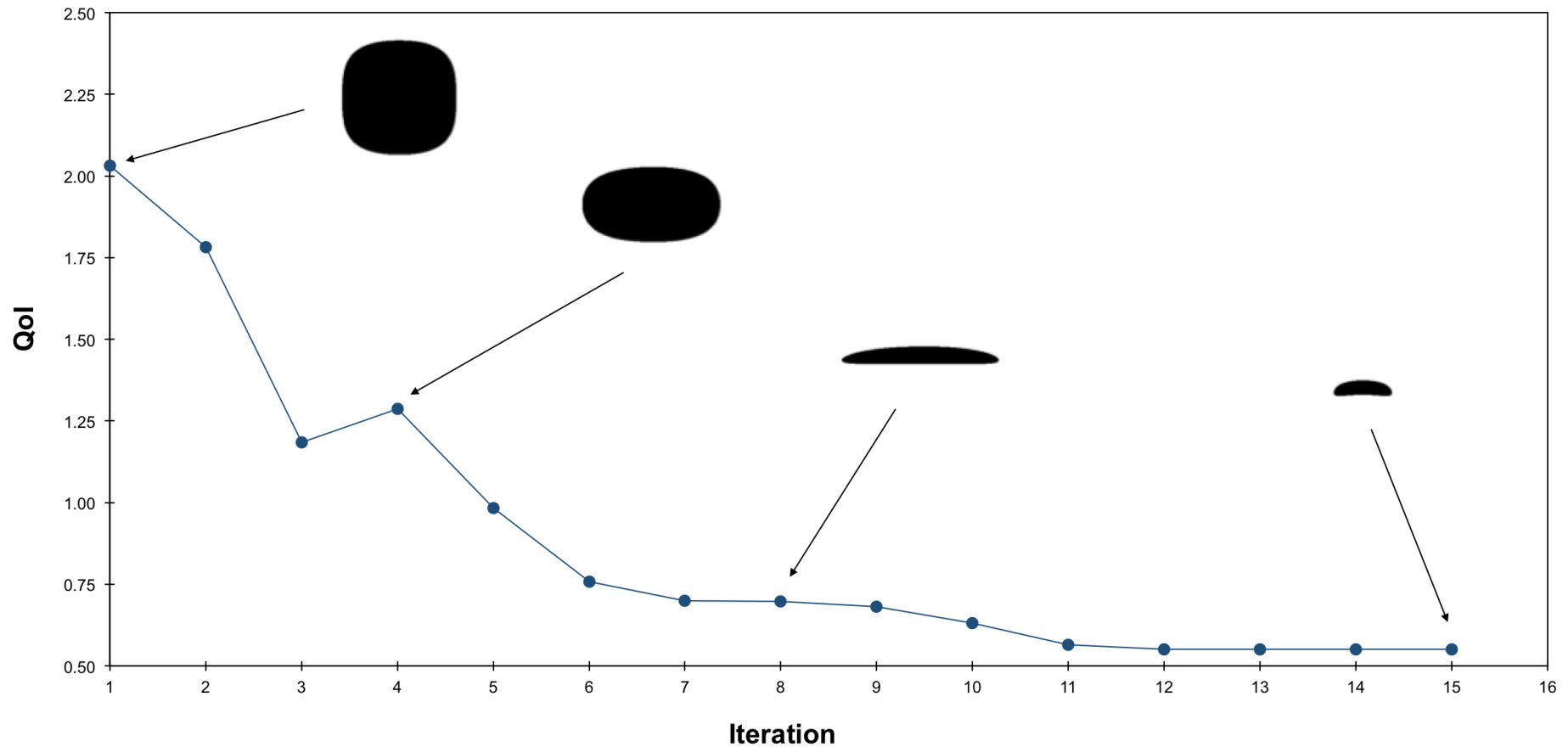
## Blunt body shape optimization

- In this case we aim at optimizing the shape of a blunt body.
- The goal is to minimize the drag coefficient.
- The body is parametrized using Bezier curves with four control points.
- In this case we use gradient-based optimization and four linear constraints.



# Additional code coupling tutorials

## Blunt body shape optimization



# Additional code coupling tutorials

## Blunt body shape optimization

- We are going to conduct two cases in the following order:
  - A DACE experiment where we only generate the geometry and the mesh.
  - Bounded-constrained gradient optimization.
- To run this case, you need the following applications,
  - DAKOTA.
  - OpenFOAM.
  - SALOME.
  - Bash utilities.

# Additional code coupling tutorials

**These are the directories and files that will be used**

- Directories:
  - **casebase**
  - **templatedir**
- Files:
  - *dakota\_case.in*
  - *simulator\_script*
  - *templatedir/input.template*
  - *templatedir/profile4points.py.template*



# Additional code coupling tutorials

## How to run this tutorial

- This case is ready to run.
- To run it, go to the case directory:

```
$> cd $TM/dakota_openfoam_coupling/blunt_body_SALOME
```

- Go to the desired sub-directory and type in the terminal

1. 

```
$> ./dakota_cleanup
```
2. 

```
$> dakota -i dakota_case.in
```

# Additional code coupling tutorials

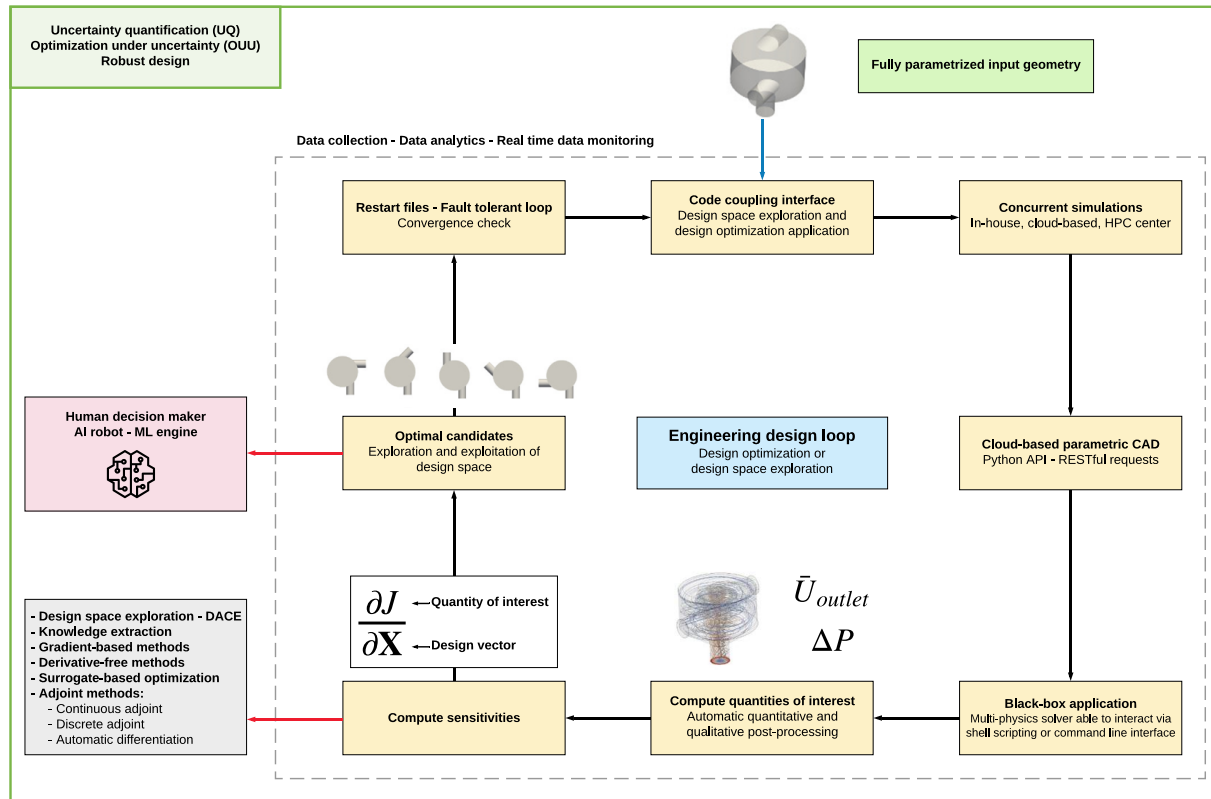
- Static mixer optimization.
- You will find this case in the following directory:

```
$TM/dakota_onshape/API_python2/test_cases/DAKOTA_static_mixer
```

# Additional code coupling tutorials

## Static mixer optimization

- In this example we will use a workflow a little bit more complicate.
- The workflow involves optimization using a cloud-based parametric CAD application.
- We will use image similarity to drive the optimization study.

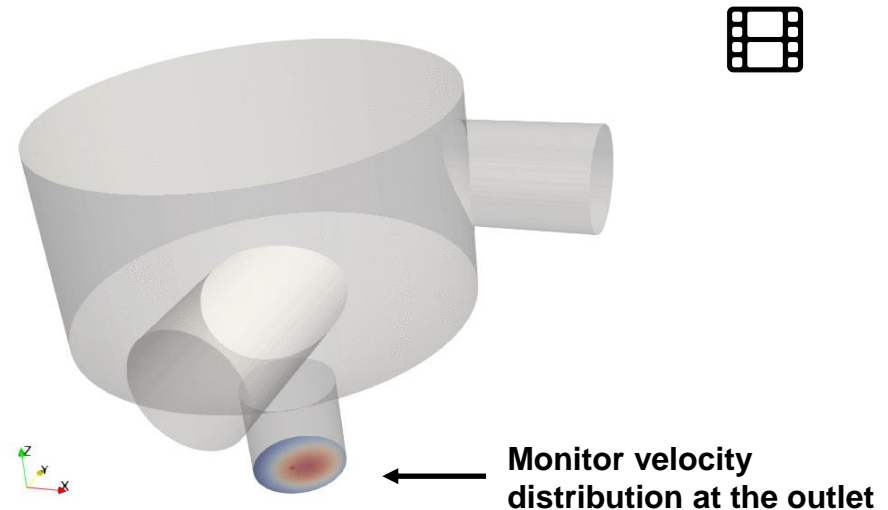
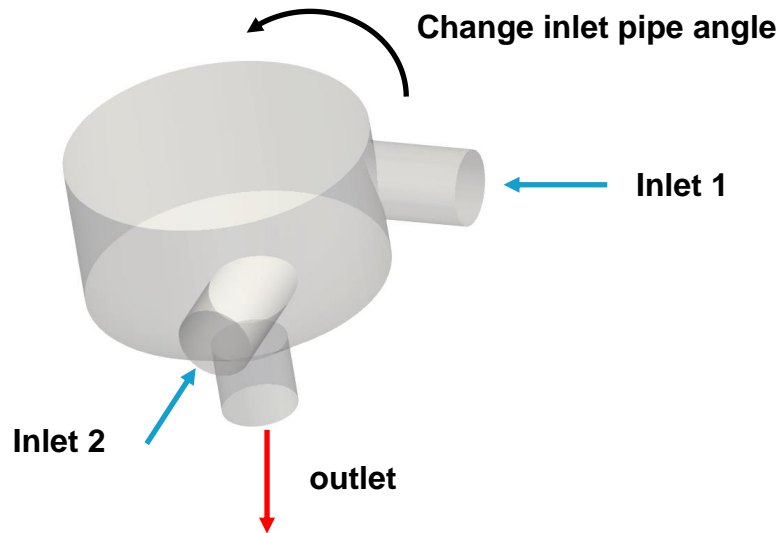


- **Code coupling/Optimizer:**  
DAKOTA
- **Concurrent computations scheduler:**  
DAKOTA
- **Parametric CAD:**  
Onshape (API)
- **Black-box solver:**  
OpenFOAM
- **Quantitative and qualitative post-processing:** Python, paraview, JavaScript
- **Real time data monitoring:**  
Python, R, BASH
- **Exploration and exploitation of design space:** Python, R, BASH
- **Additional automation scripting:**  
Python, BASH

# Additional code coupling tutorials

## Static mixer optimization

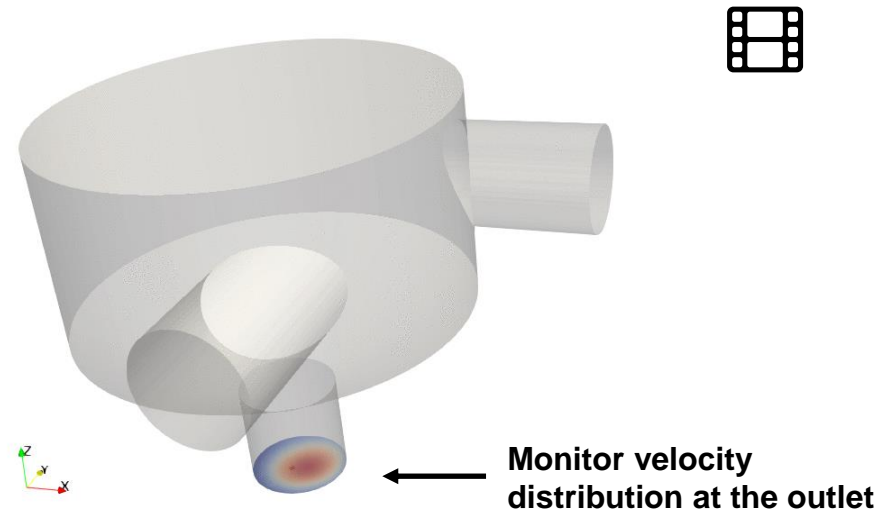
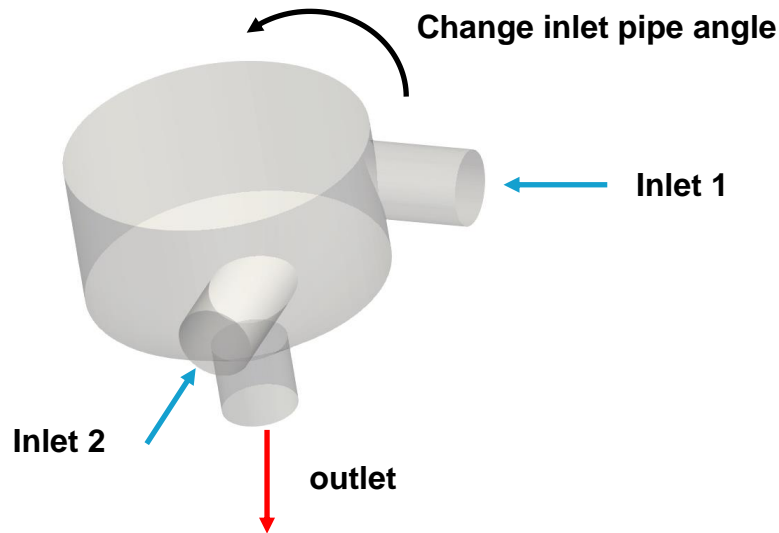
- Let us see this engineering design framework in action.
- The main goal in this case is to obtain a given velocity distribution at the outlet by changing the angle of the inlet pipe 1 (refer to the figure below).
- The velocity distribution field at the outlet was designed in such a way that the velocity normal to the outlet surface has a paraboloid distribution.
- Then, by using the SSIM index method we can compare the target image with current image.



# Additional code coupling tutorials

## Static mixer optimization

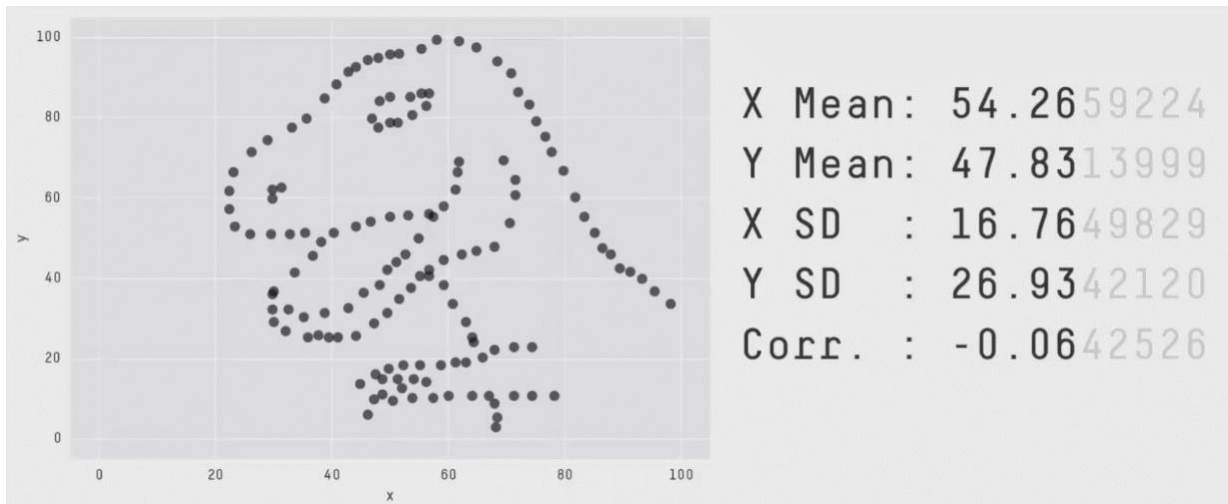
- The advantage of using image similarity is that we can now fit or optimize the problem according to a given visual field (which can come from an experiment).
- This kind of problems are often optimized using integral quantities such a uniformity index, distortion coefficient, or swirl index.
- These key performance indicators (KPI) not necessarily indicate that we are satisfying a given distribution of a field variable in a given surface or section of interest.



# Additional code coupling tutorials

## Static mixer optimization

- Let us digress from the main topic to stress the importance of visualization.
- In reference [1] you can find an enlightening discussion about the importance of visualizing the data.
- In the datasaurus dataset used in reference [1], we can see how the data points morph from one shape to another, all the while maintaining the same summary statistical values to two decimal places throughout the entire process.



“...make both calculations and graphs. Both sorts of output should be studied; each will contribute to understanding.”

F. J. Anscombe [2].

<http://www.wolfdynamics.com/training/opt/image12.gif>

[1] J. Matejka, G. Fitzmaurice. Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing. Autodesk Research. ACM SIGCHI Conference on Human Factors in Computing Systems, 2017.

[2] F. Anscombe. Graphs in Statistical Analysis. The American Statistician 27, 1, 17–21, 1973.

# Additional code coupling tutorials

## Static mixer optimization

- The datasaurus dataset it is a variant of the Anscombe's quartet on steroids.
- The Anscombe' quartet [1] is a set of four datasets with similar statistic.

| I    |       | II   |      | III  |       | IV   |       |
|------|-------|------|------|------|-------|------|-------|
| X    | Y     | X    | Y    | X    | Y     | X    | Y     |
| 10.0 | 8.04  | 10.0 | 9.14 | 10.0 | 7.46  | 8.0  | 6.58  |
| 8.0  | 6.95  | 8.0  | 8.14 | 8.0  | 6.77  | 8.0  | 5.76  |
| 13.0 | 7.58  | 13.0 | 8.74 | 13.0 | 12.74 | 8.0  | 7.71  |
| 9.0  | 8.81  | 9.0  | 8.77 | 9.0  | 7.11  | 8.0  | 8.84  |
| 11.0 | 8.33  | 11.0 | 9.26 | 11.0 | 7.81  | 8.0  | 8.47  |
| 14.0 | 9.96  | 14.0 | 8.10 | 14.0 | 8.84  | 8.0  | 7.04  |
| 6.0  | 7.24  | 6.0  | 6.13 | 6.0  | 6.08  | 8.0  | 5.25  |
| 4.0  | 4.26  | 4.0  | 3.10 | 4.0  | 5.39  | 19.0 | 12.50 |
| 12.0 | 10.84 | 12.0 | 9.13 | 12.0 | 8.15  | 8.0  | 5.56  |
| 7.0  | 4.82  | 7.0  | 7.26 | 7.0  | 6.42  | 8.0  | 7.91  |
| 5.0  | 5.68  | 5.0  | 4.74 | 5.0  | 5.73  | 8.0  | 6.89  |

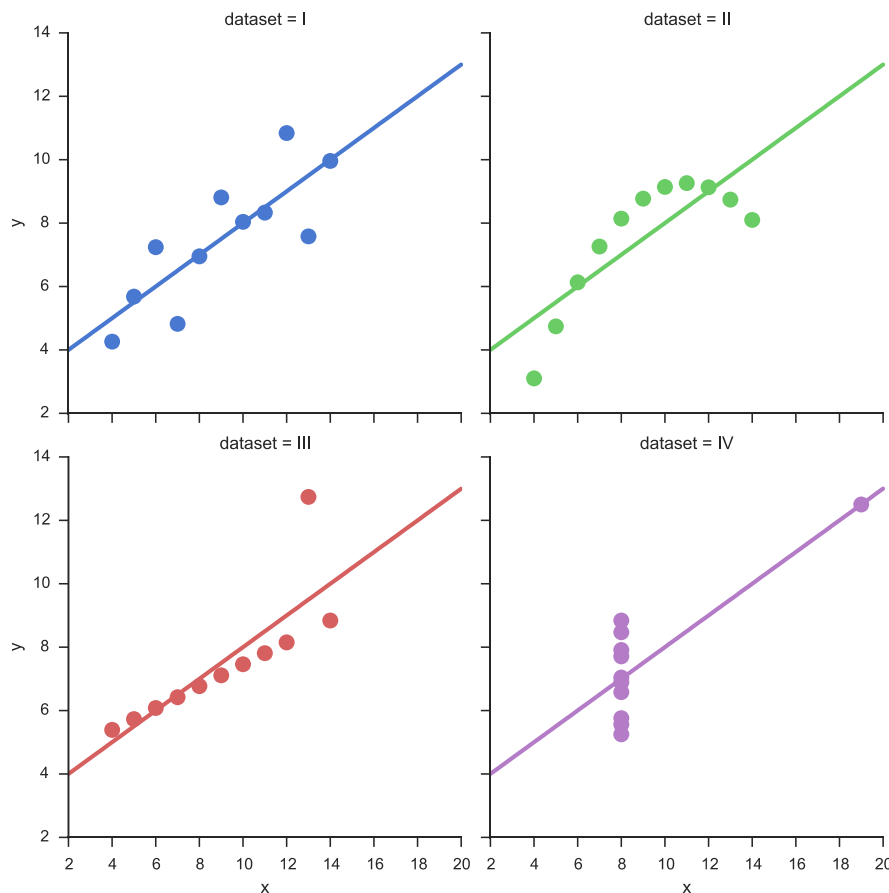
For all datasets:

| Statistical property | Value                |
|----------------------|----------------------|
| Sample size          | 11                   |
| Mean (x)             | 9                    |
| Variance (x)         | 11                   |
| Mean (y)             | 7.50                 |
| Variance (y)         | 4.122                |
| Correlation          | 0.816                |
| Linear regression    | $Y = 3.00 + 0.5000X$ |

# Additional code coupling tutorials

## Static mixer optimization

- Anscombe's quartet comprises four datasets that have nearly identical simple statistical properties, yet appear very different when graphed



For all datasets:

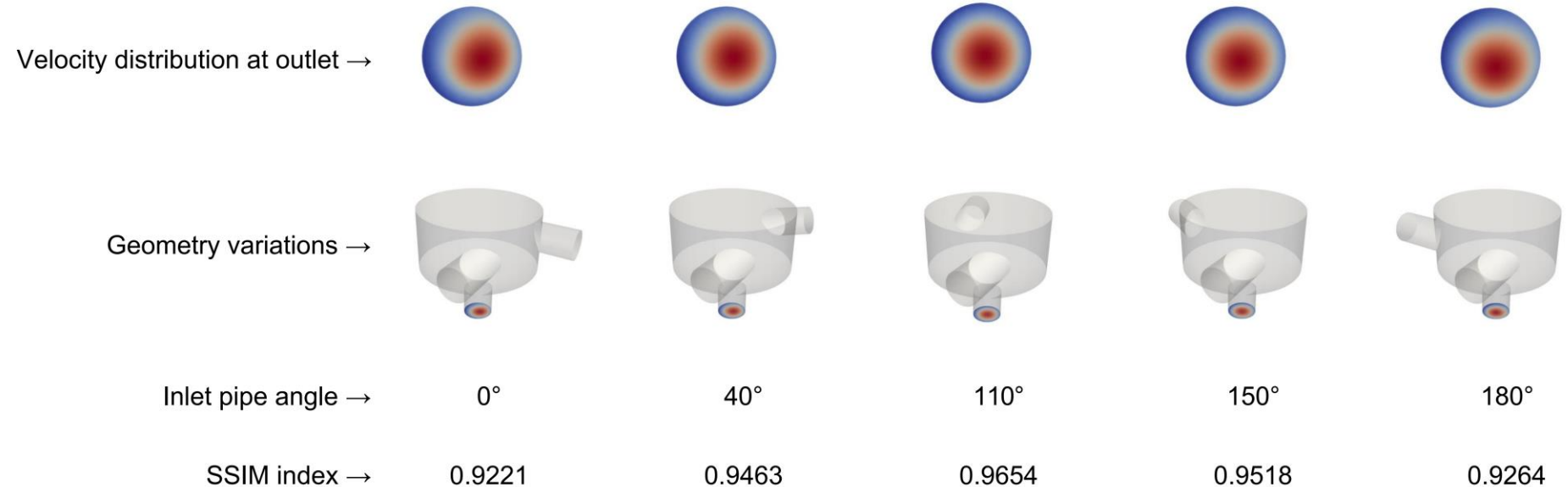
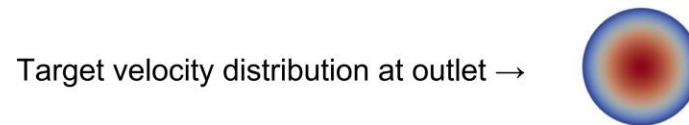
| Statistical property | Value                |
|----------------------|----------------------|
| Sample size          | 11                   |
| Mean (x)             | 9                    |
| Variance (x)         | 11                   |
| Mean (y)             | 7.50                 |
| Variance (y)         | 4.122                |
| Correlation          | 0.816                |
| Linear regression    | $Y = 3.00 + 0.5000X$ |



# Additional code coupling tutorials

## Static mixer optimization

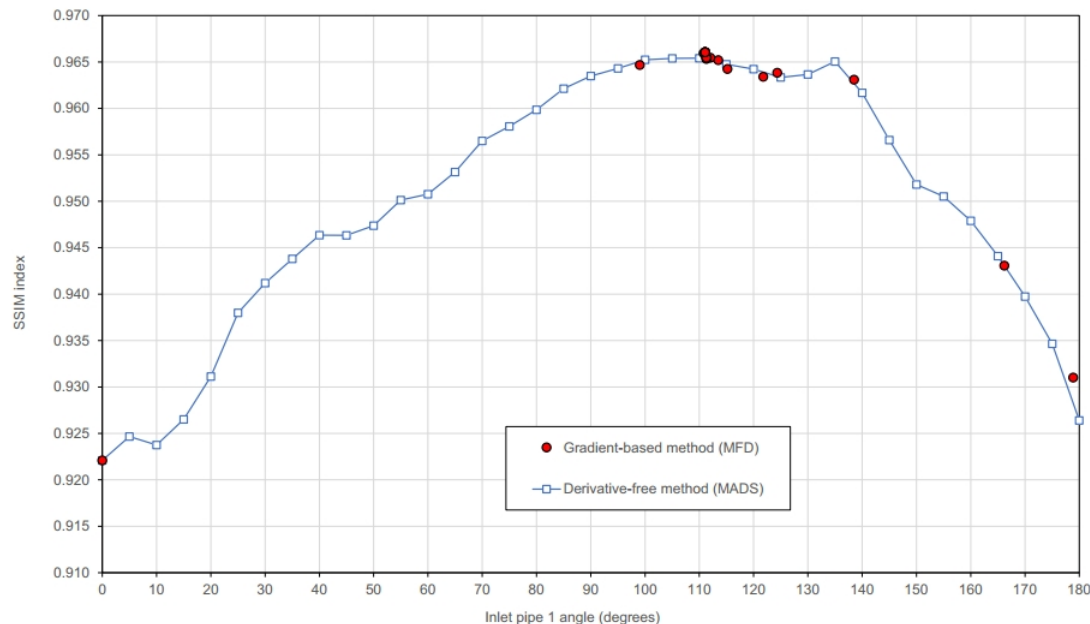
- Qualitative comparison of velocity distribution at the outlet.
- To compare the images, we will use the SSIM method.
- In the SSIM, a value of 1 means that the images are identical.



# Additional code coupling tutorials

## Static mixer optimization

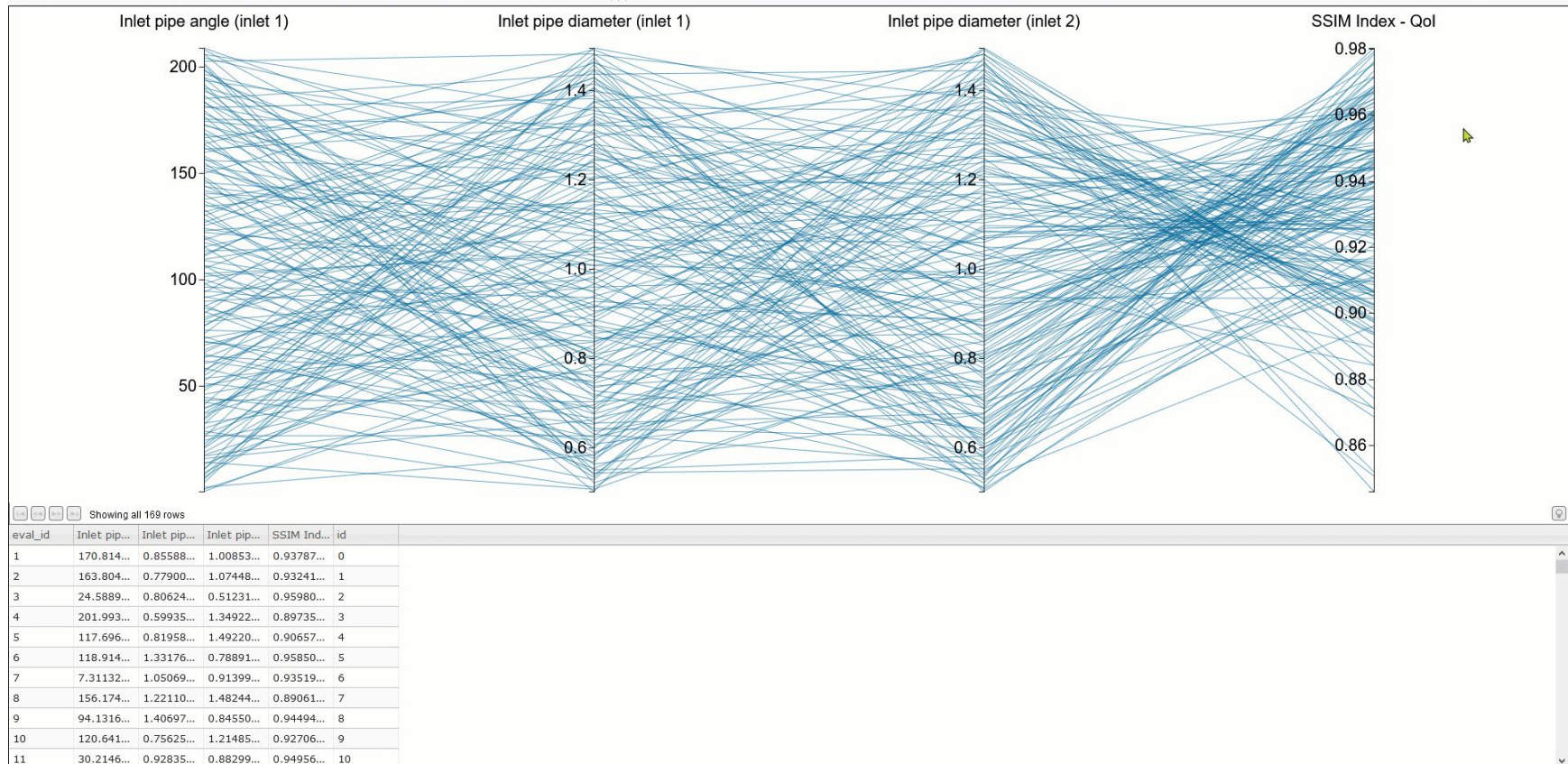
- Comparison of the outcome of a DO study and a DSE study.
- The DO study was conducted using the method of feasible directions (gradient-based method) with numerical gradients computed using forward differences.
- For the DO case, the starting point was 0 degrees, and the case converged to the optimal value in 31 function evaluations.
- Optimal value: pipe angle equal to 111.0549 degrees and SSIM index equal to 0.9660
- In the DSE case, we explored the design space from 0 to 180 degrees, in steps of 5 degrees (36 function evaluations).
- So roughly speaking, we used the same number of function evaluations as for the DO case.
- The DSE study, while not formerly converging to the optimal solution, gives more information about the design space than the DO method.



# Additional code coupling tutorials

## Static mixer optimization

- This case can be easily extended to more design variables.
- The use of exploratory data analysis techniques is of extremely importance when studying high dimensional design spaces.
- In the figure below, the outcome of a case with three design variables is visualized using parallel coordinates (interactive).
  - [https://joelguerrero.github.io/parallel\\_coordinates\\_dse\\_case/](https://joelguerrero.github.io/parallel_coordinates_dse_case/)



# Additional code coupling tutorials

## Applications needed to run this case – Additional information

- To run this case, you need the following applications,
  - DAKOTA.
  - OpenFOAM.
  - Onshape API
  - Python2 and python3
  - Paraview (headless mode).
  - Javascript and D3.js
  - Bash utilities.
- You can find more information about this case at the following links:
  - <https://www.mdpi.com/2311-5521/5/1/36>
  - <https://github.com/joelguerrero/cloud-based-cad-paper/>

# Additional code coupling tutorials

**These are the directories and files that will be used**

- Directories:
  - **support\_files**
  - **templatedir**
- Files:
  - *dakota\_case.in*
  - *simulator\_script*
  - *templatedir/feature\_to\_update.json.template*
  - Many files in the directory **support\_files**

# Additional code coupling tutorials

## How to run this tutorial

- This case is ready to run.
- To run it, go to the case directory:

```
$> cd $TM/dakota_onshape/API_python2/test_cases/DAKOTA_static_mixer
```

- Go to the desired sub-directory and type in the terminal

1. 

```
$> ./dakota_cleanup
```
2. 

```
$> dakota -i dakota_case.in
```

# Roadmap

- ~~1. Introduction to optimization methods~~
- ~~2. Choosing an optimization method~~
- ~~3. Optimization loop – The big picture~~
- ~~4. DAKOTA overview~~
- ~~5. Working with DAKOTA: Rosenbrock function~~
- ~~6. Working with DAKOTA: Branin function~~
- ~~7. Working with DAKOTA: Multi-objective optimization~~
- ~~8. Coupling DAKOTA and OpenFOAM: driven cavity case~~
- ~~9. Additional code coupling tutorials~~
- 10. Some kind of conclusion**

# Some kind of conclusion

- Implementing an engineering design loop is a meticulous and thoughtful process that requires careful planning.
- Always monitor and analyze your data (quantitative or qualitative) real-time.
- Validate and calibrate your design loop, be sure that is fault tolerant, accurate, and robust.
- We all want rapid iterations; however, do not sacrifice solution accuracy over solution speed. Design engineering loops are time consuming.
- Leverage your computational resources (local, remote, or on the cloud) and deploy concurrent tasks.



# Thank you for your attention

- We hope you have found this training useful, and we hope to see you in one of our advanced training sessions:
  - OpenFOAM® – Multiphase flows
  - OpenFOAM® – Naval applications
  - OpenFOAM® – Turbulence Modeling
  - OpenFOAM® – Compressible flows, heat transfer, and conjugate heat transfer
  - OpenFOAM® – Advanced meshing
  - DAKOTA – Optimization methods and code coupling
  - Python – Programming, data visualization, and exploratory data analysis
  - Python and R – Data science and big data
  - ParaView – Advanced scientific visualization and python scripting
  - And many more available on request
- Besides consulting services, we also offer '**Mentoring Days**' which are days of one-on-one coaching and mentoring on your specific problem.
- For more information, ask your trainer, or visit our website  
<http://www.wolfdynamics.com/>

**Be collaborative, be innovative, be cloud**



**wolf**dynamics

multiphysics simulations,  
optimization & data analytics

**Let's connect**



[guerrero@wolfdynamics.com](mailto:guerrero@wolfdynamics.com)



[www.wolfdynamics.com](http://www.wolfdynamics.com)