

# UTM Short course on MULTIDISCIPLINARY DESIGN OPTIMIZATION

Prof. Joseph Morlier

July 20, 2025

## Abstract

This lecture note aims at giving solid mathematical foundations to an engineering design optimization course

The codes and notebooks will be available at:

<https://github.com/jomorlier/mdoatutm>

## Contents

<b>1</b>	<b>Introduction to MDO</b>	<b>2</b>
<b>2</b>	<b>Warm up</b>	<b>3</b>
2.1	Scalar product? . . . . .	3
2.2	Revisiting multivariable calculus . . . . .	3
2.3	Bilinear form? . . . . .	4
2.4	Quadratic form? . . . . .	4
2.5	Check with Python? . . . . .	5
2.6	go deeper . . . . .	6
<b>3</b>	<b>Introduction to Optimization</b>	<b>7</b>
3.1	All starts with Gradient . . . . .	8
3.2	Introduction to Unconstrained Optimization . . . . .	10
3.3	On smoothness . . . . .	10
3.4	Conditions for local minima . . . . .	10
3.5	FONC . . . . .	11
3.6	SONC . . . . .	12
3.7	SOSC . . . . .	12
<b>4</b>	<b>Constrained optimization</b>	<b>13</b>
4.1	CasADi . . . . .	13
4.2	Topology example . . . . .	13
4.3	The adjoint method . . . . .	15
4.4	MDAO A short story . . . . .	16
4.5	A list of open source optimizers . . . . .	19
<b>5</b>	<b>References</b>	<b>19</b>

# 1 Introduction to MDO

Aeronautical science is in essence a multidisciplinary field. From an engineer point of view, an airplane is only a trade-off between several disciplines such as: structure, propulsion, aerodynamics, control etc ... If we give too much weight to the designer of one discipline, we will end up with exotic design.

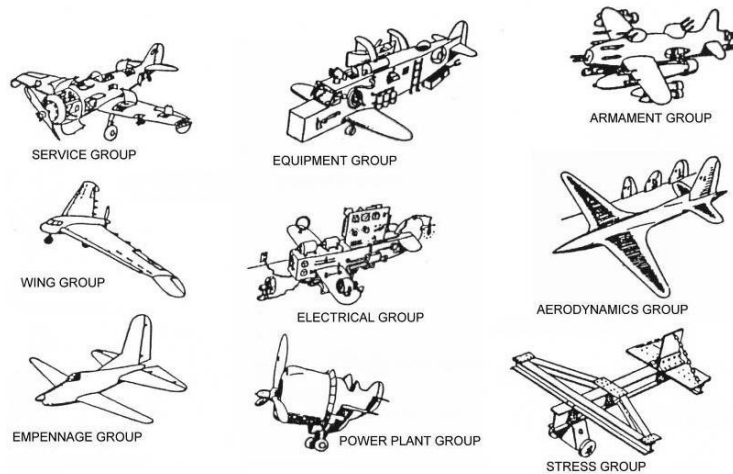


Figure 1: Each department building his own aircraft. Can we do better?

MDO is a design methodology that optimizes a complex system by considering multiple disciplines simultaneously. In the industry, design teams typically work within their specific disciplines or subsystems, optimizing their respective components locally before handing the design off to the next team. This sequential optimization approach is mathematically bound to result in suboptimal designs.

So all the expertise of an aeronautical engineer is to make compromises to get the "good" design, and this has been the case for decades. But, How can we automate this process?

MDO means: Multidisciplinary D... Optimization?  
But the D is for... ?  
D is for Design!!!

This is indeed the core of the methodology and can be used in the field of aeronautics, automotive, or consumer electronics etc. The industrial success of the structural optimization methodologies in the 1970s motivated the emergence of multidisciplinary optimization in the 1980s. Jaroslaw Sobieski, in particular, defended the decomposition methods specially designed for MDO applications. Following Sobieski, many researchers have proposed different MDO formulations. In 2013, the MDOLab led by UoM Professor Joaquim Martins proposed an exhaustive review of possible formulations in the AIAA journal. The largest recent application remains in the field of aeronautical design. For example, the Blended Wing Body has used MDO techniques extensively as part of the conceptual design level. The considered disciplines in this project were: aerodynamics, structural mechanics, propulsion and economy.

## 2 Warm up

### 2.1 Scalar product?

Let  $\mathbf{x}$  defined as  $[x_1 \ x_2 \ \dots \ x_m]^T$  and  $\mathbf{y}$  as  $[y_1 \ y_2 \ \dots \ y_m]^T$ . The scalar product  $\langle \mathbf{x}, \mathbf{y} \rangle$  of two vectors  $\mathbf{x}$  and  $\mathbf{y}$  of dimensions  $(m \times 1)$  is the scalar that is obtained by summing the products of the respective components in a given basis:

$$\langle \mathbf{x}, \mathbf{y} \rangle = x_1 y_1 + x_2 y_2 + \dots + x_m y_m = \mathbf{x}^T \mathbf{y} = \mathbf{y}^T \mathbf{x}$$

The norm of a vector can be defined as  $\|\mathbf{x}\| = \sqrt{\mathbf{x}^T \mathbf{x}}$ .

It is possible to show the Schwarz' inequality hold.  $|\mathbf{x}^T \mathbf{y}| \leq \|\mathbf{x}\| \|\mathbf{y}\|$

### 2.2 Revisiting multivariable calculus

Let  $f(x) = Ax$  ( mapping  $\mathbf{R}^n \rightarrow \mathbf{R}^m$  ) where  $A$  is a constant  $m \times n$  matrix. Then,

$$df = d(Ax) = dAx + Adx = Adx \implies f'(x) = A$$

We have  $dA = 0$  here because  $A$  does not change when we change  $x$ . Consider the function  $f(x) = Ax$  where  $A$  is a constant  $m \times n$  matrix. Then, by applying the distributive law for matrix-vector products, we have

$$\begin{aligned} df &= f(x + dx) - f(x) = A(x + dx) - Ax \\ &= Ax + Adx - Ax = Adx = f'(x)dx \end{aligned}$$

Therefore,  $f'(x) = A$ .

Let  $f(x) = Ax$  (mapping  $\mathbf{R}^n \rightarrow \mathbf{R}^m$  ) where  $A$  is a constant  $m \times n$  matrix. Then,

$$df = d(Ax) = d\tilde{A}x + Adx = Adx \implies f'(x) = A$$

We have  $dA = 0$  here because  $A$  does not change when we change  $x$ .

Let  $f(x) = x^T Ax$  (mapping  $\mathbf{R}^n \rightarrow \mathbf{R}$  ). Then,

$$df = dx^T(Ax) + x^T d(Ax) = \underbrace{dx^T Ax}_{=x^T A^T dx} + x^T Adx = x^T (A + A^T) dx = (\nabla f)^T dx$$

and hence  $f'(x) = x^T (A + A^T)$ . (In the common case where  $A$  is symmetric, this simplifies to  $f'(x) = 2x^T A$ .) Note that here we have applied in the previous example in computing  $d(Ax) = Adx$ . Furthermore,  $f$  is a scalar valued function, so we may also obtain the gradient  $\nabla f = (A + A^T)x$  as before (which simplifies to  $2Ax$  if  $A$  is symmetric).

## 2.3 Bilinear form?

A bilinear form in the variables  $x_i$  and  $y_j$  is the scalar

$$B = \sum_{i=1}^m \sum_{j=1}^n a_{ij} x_i y_j$$

which can be written in matrix form

$$B(x, y) = x^T \mathbf{A} y = y^T \mathbf{A}^T x$$

where  $x = [x_1 \ x_2 \ \dots \ x_m]^T$ ,  $y = [y_1 \ y_2 \ \dots \ y_n]^T$ , and  $\mathbf{A}$  is the  $(m \times n)$  matrix of the coefficients  $a_{ij}$  representing the core of the form.

What is the size of  $B(x, y)$ ?

$1 \times m \times m \times n \times n \times 1 = 1 \times 1$ ; i.e. a scalar

How can you write this  $x^T \mathbf{A} y = y^T \mathbf{A}^T x$ ?

The Bilinear form is a scalar ( $1 \times 1$ ). So I can write  $scalar = scalar^T$

HINTS: use  $(\mathbf{A}\mathbf{B})^T = \mathbf{B}^T \mathbf{A}^T$

$$x^T \mathbf{A} y = (x^T \mathbf{A} y)^T = (\mathbf{A} y)^T (x^T)^T = y^T \mathbf{A}^T x$$

Given the bilinear form, the gradient of the form with respect to  $x$  is given by

$$\text{grad}_x B(x, y) = \left( \frac{\partial B(x, y)}{\partial x} \right)^T = \mathbf{A} y$$

But How?

$$\text{grad}_x B(x, y) = \left( \frac{\partial B(x, y)}{\partial x} \right)^T = \left( \frac{\partial x^T \mathbf{A} y}{\partial x} \right)^T = \left( \frac{\partial y^T \mathbf{A}^T x}{\partial x} \right)^T = (y^T \mathbf{A}^T)^T = \mathbf{A} y$$

whereas the gradient of  $B$  with respect to  $y$  is given (same demonstration) by

$$\text{grad}_y B(x, y) = \left( \frac{\partial B(x, y)}{\partial y} \right)^T = \mathbf{A}^T x$$

## 2.4 Quadratic form?

A special case of bilinear form is the quadratic form

$$Q(x) = x^T \mathbf{A} x$$

Given the quadratic form with  $\mathbf{A}$  symmetric ( $\mathbf{A} = \mathbf{A}^T$ ), the gradient of the form with respect to  $x$  is given by

$$\text{grad}_x Q(x) = \left( \frac{\partial Q(x)}{\partial x} \right)^T = 2\mathbf{A} x$$

But How?

We therefore see that we must derive with respect to  $x$  an expression where  $x$  appears twice: once on the right, and once on the left. To visualize this, let's denote these two vectors in blue and red (this is just a play of color: whether it is written in red or blue, the vector  $x$  always represents the same thing!):

$$\frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x}) = \frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x}) + \frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x})$$

The derivative with respect to the red  $x$  does not pose a problem: you just need to apply the little math reminder framed above:

$$\frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x}) = \mathbf{A} \mathbf{x}$$

For the derivative with respect to the blue  $x$  we must manage to bring the blue  $x$  to the left. To do this, we use the fact that Quadratic form is... a scalar! We can therefore happily transpose a scalar: this does not change it!

$$\frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x}) = \frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x})^T$$

HINTS: use  $(\mathbf{A}\mathbf{B})^T = \mathbf{B}^T \mathbf{A}^T$  and  $A = A^T$  i.e.  $A$  is symmetric.

It leads to:

$$\frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x})^T = \frac{\partial}{\partial \mathbf{x}} (\mathbf{A} \mathbf{x})^T (\mathbf{x}^T)^T = \frac{\partial}{\partial \mathbf{x}} (\mathbf{x})^T \mathbf{A}^T \mathbf{x} = \mathbf{A} \mathbf{x}$$

This leads to:

$$\frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x}) = \mathbf{A} \mathbf{x} + \mathbf{A} \mathbf{x} = 2\mathbf{A} \mathbf{x}$$

## 2.5 Check with Python?

Consider a matrix of quadratic form given hereafter

$$A = \begin{bmatrix} 3 & 2 & 0 \\ 2 & -1 & 4 \\ 0 & 4 & -2 \end{bmatrix}$$

construct the quadratic form  $\mathbf{x}^T A \mathbf{x}$ .

$$\begin{aligned} \mathbf{x}^T A \mathbf{x} &= \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} 3 & 2 & 0 \\ 2 & -1 & 4 \\ 0 & 4 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ &= \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} 3x_1 + 2x_2 \\ 2x_1 - x_2 + 4x_3 \\ 4x_2 - 2x_3 \end{bmatrix} \\ &= x_1 (3x_1 + 2x_2) + x_2 (2x_1 - x_2 + 4x_3) + x_3 (4x_2 - 2x_3) \\ &= 3x_1^2 + 4x_1x_2 - x_2^2 + 8x_2x_3 - 2x_3^2 \end{aligned}$$

Fortunately, there is an easier way to calculate quadratic form. Notice that coefficients of  $x_i^2$  is on the principal diagonal and coefficients of  $x_i x_j$  are be split evenly been  $(i, j)$ - and  $(j, i)$ - entries in  $A$ .

Consider another example,

$$A = \begin{bmatrix} 3 & 2 & 0 & 5 \\ 2 & -1 & 4 & -3 \\ 0 & 4 & -2 & -4 \\ 5 & -3 & -4 & 7 \end{bmatrix}$$

All  $x_i^2$  's terms are

$$3x_1^2 - x_2^2 - 2x_3^2 + 7x_4^2$$

whose coefficients are from principal diagonal.

All  $x_i x_j$  's terms are

$$4x_1x_2 + 0x_1x_3 + 10x_1x_4 + 8x_2x_3 - 6x_2x_4 - 8x_3x_4$$

Add up together then quadratic form is

$$3x_1^2 - x_2^2 - 2x_3^2 + 7x_4^2 + 4x_1x_2 + 0x_1x_3 + 10x_1x_4 + 8x_2x_3 - 6x_2x_4 - 8x_3x_4$$

Let's verify in SymPy.

```
1 import sympy as sy
2
3 x1, x2, x3, x4 = sy.symbols('x_1 x_2 x_3 x_4')
4 A = sy.Matrix([[3,2,0,5],[2,-1,4,-3],[0,4,-2,-4],[5,-3,-4,7]])
5 x = sy.Matrix([x1, x2, x3, x4])
6 sy.expand(x.T*A*x)
```

Listing 1: Python example

The result is exactly the same as we derived

$$3x_1^2 - x_2^2 - 2x_3^2 + 7x_4^2 + 4x_1x_2 + 0x_1x_3 + 10x_1x_4 + 8x_2x_3 - 6x_2x_4 - 8x_3x_4$$

## 2.6 go deeper

Consider  $f(x) = x^T A x$  where  $x \in \mathbf{R}^n$  and  $A$  is a square  $n \times n$  matrix, and thus  $f(x) \in \mathbf{R}$ . Compute  $df$ ,  $f'(x)$ , and  $\nabla f$ .

We can do this directly from the definition.

$$\begin{aligned} df &= f(x + dx) - f(x) \\ &= (x + dx)^T A (x + dx) - x^T A x \\ &= x^T A x + dx^T A x + x^T A dx + dx^T A dx - x^T A x \\ &= \underbrace{x^T (A + A^T) dx}_{f'(x) = (\nabla f)^T} \implies \nabla f = (A + A^T) x. \end{aligned}$$

Here, we dropped terms with more than one  $dx$  factor as these are asymptotically negligible. Another trick was to combine  $dx^T A x$  and  $x^T A dx$  by realizing that these are scalars and hence equal to their own transpose:  $dx^T A x = (dx^T A x)^T = x^T A^T dx$ . Hence, we have found that  $f'(x) = x^T (A + A^T) = (\nabla f)^T$ , or equivalently  $\nabla f = [x^T (A + A^T)]^T = (A + A^T) x$ .

It is, of course, also possible to compute the same gradient component-by-component, the way you probably learned to do in multivariable calculus. First, you would need to write  $f(x)$  explicitly in terms of the components of  $x$ , as  $f(x) = x^T A x = \sum_{i,j} x_i A_{i,j} x_j$ . Then, you would compute  $\partial f / \partial x_k$  for each  $k$ , taking care that  $x$  appears twice in the  $f$  summation. However, this approach is awkward, error-prone, labor-intensive, and quickly becomes worse as we move on to more complicated functions. It is much better, we feel, to get used to treating vectors and matrices as a whole, not as mere collections of numbers.

### 3 Introduction to Optimization

Let  $f(\mathbf{x})$  be a scalar function of a vector of variables  $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbf{R}^n$ . Numerical Optimization is

the minimization or maximization of this function  $f$  subject to constraints on  $\mathbf{x}$ . This  $f$  is a scalar function of  $\mathbf{x}$ , also known as the objective function and the continuous components  $x_i \in \mathbf{x}$  are called the decision variables.

The optimization problem is formulated in the following way:

$$\begin{array}{ll} \min_{\mathbf{x} \in \mathbf{R}^n} & f(\mathbf{x}) \\ \text{subject to} & g_k(\mathbf{x}) \leq 0, k = 1, 2, \dots, m \\ & h_k(\mathbf{x}) = 0, k = 1, 2, \dots, r \\ & m, r < n. \end{array}$$

Here,  $g_k(\mathbf{x})$  and  $h_k(\mathbf{x})$  are scalar functions too (like  $f(\mathbf{x})$ ) and are called constraint functions. The constraint functions define some specific equations and/or inequalities that  $\mathbf{x}$  should satisfy.

Optimization Algorithms for continuous variables are iterative techniques that follow the following fundamental steps: - Initialize with a guess of the decision variables  $\mathbf{x}$ , - Iterate through the process of generating a list of improving estimates, - check whether the terminating conditions are met, and the estimates will be probably stop at the solution point  $\mathbf{x}^*$ .

Most of the optimization strategies make use of either the objective function  $f(\mathbf{x})$ , the constraint functions  $g(\mathbf{x})$  and  $h(\mathbf{x})$ , the first or second derivatives of these said functions, information collected during previous iterations and/or local information gathered at the present point. As Nocedal and Wright mentions, a good optimization algorithm should have the following fundamental properties:

- **Robustness:** For all acceptable initial points chosen, the algorithm should operate well on a broad range of problems, in their particular class.
- **Efficiency:** The time complexity and the space complexity of the algorithm should be practicable
- **Accuracy:** The solution should be as precise as possible, with the caveat that it should not be too much delicate to errors in the data or to numerical rounding and/or truncating errors while it is being executed on a machine.

There might be some trade offs allowed between speed and memory, between speed and robustness, etc.

**Definition 1.1** A solution of  $f(\mathbf{x})$  is a point  $\mathbf{x}^*$  which denotes the optimum vector that solves equation (1.1), corresponding to the optimum value  $f(\mathbf{x}^*)$ .

In case of a minimization problem, the optimum vector  $\mathbf{x}^*$  is referred to as the global minimizer of  $f$ , and  $f$  attains the least possible value at  $\mathbf{x}^*$ .

To design an algorithm that finds out the global minimizer for a function is quite difficult, as in most cases we do not have the idea of the overall shape of  $f$ . Mostly our knowledge is restricted to a local portion of  $f$ .

**Definition 1.6** A feasible region is the set of those points which satisfy all the constraints provided.

**BTW, How you do Maximisation?**

We just defined a minimization problem as our optimization task. We could do the same with a maximization problem with little tweaks. The problem  $\max_{\mathbf{x} \in \mathbf{R}^n} f(\mathbf{x})$  can be formulated as:

$$\max_{\mathbf{x} \in \mathbf{R}^n} f(\mathbf{x}) = - \min_{\mathbf{x} \in \mathbf{R}^n} \{-f(\mathbf{x})\}$$

We then apply any minimization technique after setting  $\hat{f}(\mathbf{x}) = -f(\mathbf{x})$ . Further, for the inequality constraints for the maximization problem, given by  $g_k(\mathbf{x}) \geq 0$ , we set

$$\hat{g}_k(\mathbf{x}) = -g_k(\mathbf{x})$$

The problem thus has become,

$$\begin{array}{ll} \min_{\mathbf{x} \in \mathbf{R}^n} & \hat{f}(\mathbf{x}) \\ \text{subject to} & \hat{g}_k(\mathbf{x}) \leq 0, k = 1, 2, \dots, m \\ & h_k(\mathbf{x}) = 0, k = 1, 2, \dots, r \\ & m, r < n. \end{array}$$

After the solution  $\mathbf{x}^*$  is computed, the maximum value of the problem is given by:  $-\hat{f}(\mathbf{x}^*)$ .

### 3.1 All starts with Gradient

Definition 1.9 For a differentiable objective function  $f(\mathbf{x}) : \mathbf{R}^n \rightarrow \mathbf{R}$ , its gradient vector given by  $\nabla f(\mathbf{x}) : \mathbf{R}^n \rightarrow \mathbf{R}^n$ , is defined at the point  $\mathbf{x}$  in the  $n$ -dimensional space as the vector of first order partial derivatives:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{x}) \end{pmatrix}$$

Now, if  $f(\mathbf{x})$  is smooth, the gradient vector  $\nabla f(\mathbf{x})$  is always perpendicular to the contours at the point  $\mathbf{x}$ . The gradient vector is thus in the direction of the maximum increase of  $f(\mathbf{x})$ . Look at the figure below.

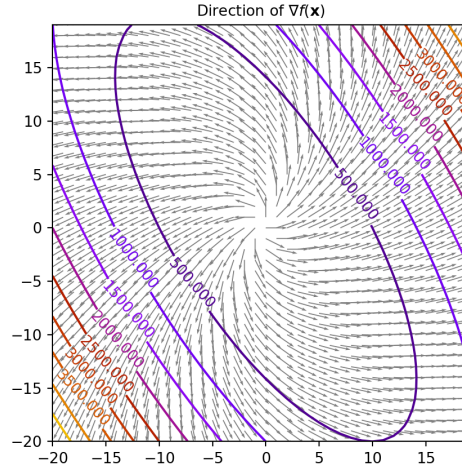


Figure 2: A nice function with his gradient. What is the meaning of the arrows ?

Definition 1.10 For a twice continuously differentiable function  $f : \mathbf{R}^n \rightarrow \mathbf{R}$ , its Hessian matrix given by  $\mathbf{H}(f(\mathbf{x}))$  is defined at the point  $\mathbf{x}$  in the  $n \times n$ -dimensional space as the matrix of second order partial derivatives:

$$\mathbf{H}f(\mathbf{x}) = \frac{\partial^2 f}{\partial x_i \partial x_j} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(\mathbf{x}) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_2^2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_n \partial x_2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(\mathbf{x}) \end{pmatrix}$$

One important relation that we will keep in mind is that the Hessian matrix is the Jacobian of the gradient vector of  $f(\mathbf{x})$ , where the Jacobian matrix of a vector-valued function  $\mathbf{F}(\mathbf{x})$  is the matrix of all its first order partial derivatives, given by,  $\mathbf{JF}(\mathbf{x}) = \begin{pmatrix} \frac{\partial \mathbf{F}}{\partial x_1} & \cdots & \frac{\partial \mathbf{F}}{\partial x_n} \end{pmatrix}$ . The relation is as followed:



$$\mathbf{H}f(\mathbf{x}) = \mathbf{J}(\nabla f(\mathbf{x}))$$

Let us consider an example now.

Example 1.1 Let an objective function be  $f(\mathbf{x}) = 2x_1x_2^3 + 3x_2^2x_3 + x_3^3x_1$ . We will find out the gradient vector  $\nabla f(\mathbf{x})$  and the Hessian matrix  $\mathbf{H}f(\mathbf{x})$  at the point  $\mathbf{p} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ .

The gradient vector is  $\nabla f(\mathbf{x}) = \begin{pmatrix} 2x_2^3 + x_3^3 \\ 6x_1x_2^2 + 6x_2x_3 \\ 3x_2^2 + 3x_3^2x_1 \end{pmatrix}$ . So  $\nabla f(\mathbf{x}) \Big|_{\mathbf{p}} = \begin{pmatrix} 43 \\ 60 \\ 39 \end{pmatrix}$ . The Hessian matrix is therefore given by,  $\mathbf{H}f(\mathbf{x}) = \begin{pmatrix} 0 & 6x_2^2 & 3x_3^2 \\ 6x_2^2 & 12x_1x_2 + 6x_3 & 6x_2 \\ 3x_3^2 & 6x_2 & 6x_3x_1 \end{pmatrix}$  and at point

$\mathbf{p}, \mathbf{H}f(\mathbf{x}) \Big|_{\mathbf{p}} = \begin{pmatrix} 0 & 24 & 27 \\ 24 & 42 & 12 \\ 27 & 12 & 18 \end{pmatrix}$ . We will try to work out the same example with Python scripting now. For that we need an extra package called autograd, besides the numpy package. The autograd package is used for automatically differentiating native Python and Numpy code. Fundamentally autograd is used in gradient-based optimization. First pip install the autograd package

```
1 import autograd.numpy as au
2 from autograd import grad, jacobian
3
4 p = np.array([1, 2, 3], dtype=float)
5 def f(x): # Objective function
6     return 2*x[0]*x[1]**3+3*x[1]**2*x[2]+x[2]**3*x[0]
7 grad_f = grad(f) # gradient of the objective function
8 hessian_f = jacobian(grad_f) # Hessian of the objective function
9
10 print("gradient vector:", grad_f(p))
11 print("Hessian matrix:\n", hessian_f(p))
```

Listing 2: Python example

```
gradient vector: [43. 60. 39.]
Hessian matrix:
[[ 0. 24. 27.]
 [24. 42. 12.]
 [27. 12. 18.]]
```

All these steps can be automated using JAX. Even more interesting is how we can compute the Hessian of a function by computing the Jacobian twice; this is what makes JAX so powerful! We see that the function hessian takes in a function and returns a function as well.

```
1 def circle(x): return x[0]**2 + x[1]**2
2 def hessian(f): return jacfwd(jacrev(f))
3 H = hessian(circle)
4 H(np.array([1.0, 2.0]))
```

Listing 3: Python example

```
[[2.00, 0.], [0.02, 0.]]
```

It should be noted that the gradients are computed with automatic differentiation, which is much more accurate and efficient compared to finite differences.

## 3.2 Introduction to Unconstrained Optimization

This chapter introduces what exactly an unconstrained optimization problem is. Studying the first order and second order necessary and sufficient conditions for local minimizer in an unconstrained optimization tasks. Examples have been supplied too in view of understanding the necessary and sufficient conditions better. The Python package `scipy.optimize`, which will form an integral part in solving many optimization problems in the later chapters of this book, is introduced too. The chapter ends with an overview of how an algorithm to solve unconstrained minimization problem works, covering briefly two procedures: line search descent method and trust region method.

As we have discussed in the first chapter, an unconstrained optimization problem deals with finding the local minimizer  $\mathbf{x}^*$  of a real valued and smooth objective function  $f(\mathbf{x})$  of  $n$  variables, given by  $f : \mathbf{R}^n \rightarrow \mathbf{R}$ , formulated as,

$$\min_{\mathbf{x} \in \mathbf{R}^n} f(\mathbf{x})$$

with no restrictions on the decision variables  $\mathbf{x}$ . We work towards computing  $\mathbf{x}^*$ , such that  $\forall \mathbf{x}$  near  $\mathbf{x}^*$ , the following inequality is satisfied:

$$f(\mathbf{x}^*) \leq f(\mathbf{x})$$

*A priori* we don't know the type of the objective function (unimodal, multimodal, noisy).

## 3.3 On smoothness

In terms of analysis, the measure of the number of continuous derivative a function has, characterizes the smoothness of a function.

Definition 2.1 A function  $f$  is smooth if it can be differentiated everywhere, i.e, the function has continuous derivatives up to some desired order over particular domain <https://mathworld.wolfram.com/SmoothFunction.html>

Some examples of smooth functions are,  $f(x) = x$ ,  $f(x) = e^x$ ,  $f(x) = \sin(x)$ , etc. To study the local minima  $\mathbf{x}^*$  of a smooth objective function  $f(\mathbf{x})$ , we emphasize on Taylor's theorem for a multivariate function, thus focusing on the computations of the gradient vector  $\nabla f(\mathbf{x})$  and the Hessian matrix  $\mathbf{H}f(\mathbf{x})$ .

## 3.4 Conditions for local minima

The following conditions are necessary for  $\mathbf{x}$  to be at a local minimum of  $f$  :

- $\nabla f(\mathbf{x}) = 0$ , the first-order necessary condition (FONC)
- $\nabla^2 f(\mathbf{x})$  is positive semidefinite (for a review of this definition, see appendix C.6), the second-order necessary condition (SONC)

The FONC tells us that the function is not changing at  $x$ . The next figure shows examples of multivariate functions where the FONC is satisfied. The SONC tells us that  $\mathbf{x}$  is in a bowl.

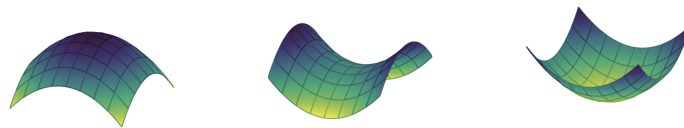


Figure 3: 3 kinds of zone where the gradient is zero?

- A local maximum. The gradient at the center is zero, but the Hessian is negative definite.
- A saddle. The gradient at the center is zero, but it is not a local minimum.
- A bowl. The gradient at the center is zero and the Hessian is positive definite. It is a local minimum.

The FONC and SONC can be obtained from a simple analysis. In order for  $\mathbf{x}^*$  to be at a local minimum, it must be smaller than those values around it:

$$f(\mathbf{x}^*) \leq f(\mathbf{x} + h\mathbf{y}) \Leftrightarrow f(\mathbf{x} + h\mathbf{y}) - f(\mathbf{x}^*) \geq 0$$

If we write the second-order approximation for  $f(\mathbf{x}^*)$ , we get:

$$f(\mathbf{x}^* + h\mathbf{y}) = f(\mathbf{x}^*) + h\nabla f(\mathbf{x}^*)^\top \mathbf{y} + \frac{1}{2}h^2\mathbf{y}^\top \nabla^2 f(\mathbf{x}^*)\mathbf{y} + O(h^3)$$

We know that at a minimum, the first derivative must be zero, and we neglect the higher order terms. Rearranging, we get:

$$\frac{1}{2}h^2\mathbf{y}^\top \nabla^2 f(\mathbf{x}^*)\mathbf{y} = f(\mathbf{x} + h\mathbf{y}) - f(\mathbf{x}^*) \geq 0$$

### 3.5 FONC

This ultimately gives  $\nabla f(\mathbf{x}^*) = 0$ , proving the first-order necessary condition. Example 2.1 The Rosenbrock function of  $n$ -variables is given by:

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

where,  $\mathbf{x} \in \mathbf{R}^n$ . For this example let us consider the Rosenbrock function for two variables, given by:

$$f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

We will show that the first order necessary condition is satisfied for the local minimizer  $\mathbf{x}^* = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ . We first check whether  $\mathbf{x}^*$  is a minimizer or not. Putting  $x_1 = x_2 = 1$  in  $f(\mathbf{x})$ , we get  $f(\mathbf{x}) = 0$ . Now, we check whether the  $\mathbf{x}^*$  satisfies the first order necessary condition. For that we calculate  $\nabla f(\mathbf{x}^*)$ .

$$\nabla f(\mathbf{x}^*) = \begin{bmatrix} -400x_1(x_2 - x_1^2) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{bmatrix}_{\mathbf{x}^*} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

So, we see that the first order necessary condition is satisfied. We can do similar analysis using the scipy.optimize package in Python. The Scipy official reference states that the scipy.optimize package provides the user with many commonly used optimization algorithms and test functions.

```
1 import numpy as np
2 import scipy
3 # Import the Rosenbrock function, its gradient and Hessian respectively
4 from scipy.optimize import rosen, rosen_der, rosen_hess
5 x_m = np.array([1, 1]) #given local minimizer
6 rosen(x_m) # check whether x_m is a minimizer
7 rosen_der(x_m) # calculates the gradient at the point x_m
```

Listing 4: Python example

0.0

the result is 0.0 . So  $\mathbf{x}^*$  is a minimizer. We then check for the first order necessary condition, using the gradient.

```
1 rosen_der(x_m) # calculates the gradient at the point x_m
```

Listing 5: Python example

array([0, 0])

This matches with our calculations and also satisfies the first-order necessary condition.

### 3.6 SONC

$$\mathbf{H}f(\mathbf{x}^*) \geq 0$$

the Hessian matrix is positive semi-definite. You don't know a priori what it is !

### 3.7 SOSC

For a real-valued smooth objective function  $f(\mathbf{x})$ , if  $\mathbf{x}^*$  is its local minimizer and  $\mathbf{H}f(\mathbf{x})$  exists and is continuous in an open neighborhood  $\subset \mathbf{R}^n$  of  $\mathbf{x}^*$  along the feasible direction  $\delta$ , then the conditions:

$$\nabla^T f(\mathbf{x}^*) \delta = 0$$

i.e.,

$$\nabla f(\mathbf{x}^*) = 0$$

and

$$\delta^T \mathbf{H}f(\mathbf{x}^*) \delta > 0$$

i.e, a positive definite Hessian matrix imply that  $\mathbf{x}^*$  is a strong local minimizer of  $f(\mathbf{x})$ .

Let us now work with a new test function called Himmelblau's function, given by,

$$f(\mathbf{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

where,  $\mathbf{x} \in \mathbf{R}^2$ .

We will check whether  $\mathbf{x}^* = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$  satisfies the second-order sufficient conditions satisfying the fact that it is a strong local minimizer. We will again use the autograd package to do the analyses for this objective function. Let us first define the function and the local minimizer as  $x^*$  in Python:

```
1 def Himm(x):
2     return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2
3
4 x_star = np.array([3, 2], dtype='float') #local minimizer
5 print("function at x_star:", Himm(x_star))
```

Listing 6: Python example

We then check whether  $x^*$  is a minimizer.

function at  $x^*$ : 0.0

Now, we calculate the gradient vector and the Hessian matrix of the function at  $x^*$  and look at the results,

```
1 # import the necessary packages
2 import autograd.numpy as au
3 from autograd import grad, jacobian
4
5 # gradient vector of the Himmelblau's function
6 Himm_grad=grad(Himm)
7 print("gradient vector at x_star:", Himm_grad(x_star))
8
9 # Hessian matrix of the Himmelblau's function
```

Listing 7: Python example

gradient vector at  $x^*$ : [0. 0.]

```

1 Himm_hess = jacobian(Himm_grad)
2 M = Himm_hess(x_star)
3 eigs = np.linalg.eigvals(M)
4 print("The eigenvalues of M:", eigs)
5 if (np.all(eigs>0)):
6     print("M is positive definite")
7 elif (np.all(eigs>=0)):
8     print("M is positive semi-definite")
9 else:
10    print("M is negative definite")

```

Listing 8: Python example

The eigenvalues of M: [82.28427125 25.71572875]  
M is positive definite

We see that  $x_1$  satisfies the second order sufficient conditions and is a strong local minimizer. We wanted to perform the analyses using autograd package instead of `scipy.optimize`, because there might be cases when we need to use test functions that are not predefined in `scipy.optimize` package, unlike the Rosenbrock function.

## 4 Constrained optimization

### 4.1 CasADi

AD is for Automatic Differentiation.

The NLP solvers distributed with or interfaced to CasADi solves parametric NLPs of the following form:

$$\begin{aligned}
 &\underset{x}{\text{minimize:}} && f(x, p) \\
 &\text{subject to:} && \begin{aligned} x_{lb} &\leq x && \leq x_{ub} \\ g_{lb} &\leq g(x, p) && \leq g_{ub} \end{aligned}
 \end{aligned}$$

where  $x \in \mathbf{R}^{n_x}$  is the decision variable and  $p \in \mathbf{R}^{n_p}$  is a known parameter vector. An NLP solver in CasADi is a function that takes the parameter value ( $p$ ), the bounds ( $lb_x, ub_x, lb_g, ub_g$ ) and a guess for the primal-dual solution ( $x_0, lam_x, lam_g$ ) and returns the optimal solution. Unlike integrator objects, NLP solver functions are currently not differentiable functions in CasADi.

There are several NLP solvers interfaced with CasADi. The most popular one is IPOPT, an open-source primal-dual interior point method which is included in CasADi installations. Others, that require the installation of third-party software, include SNOPT, WORHP and KNITRO. Whatever the NLP solver used, the interface will automatically generate the information that it needs to solve the NLP, which may be solver and option dependent.

Typically an NLP solver will need a function that gives the Jacobian of the constraint function and a Hessian of the Lagrangian function ( $L(x, \lambda) = f(x) + \lambda^T g(x)$ ) with respect to  $x$ . It is important to point out that CasADi is not a conventional AD tool, that can be used to calculate derivative information from existing user code with little to no modification. If you have an existing model written in C++, Python or MATLAB/Octave, you need to be prepared to reimplement the model using CasADi syntax.

### 4.2 Topology example

An engineering example with quadratic form is introduced, so called topology optimization problem based on the powerlaw approach, where the objective is to minimize Compliance (is the opposite of stiffness) can be written as

$$\left. \begin{array}{l} \min_{\mathbf{x}} : c(\mathbf{x}) = \mathbf{U}^T \mathbf{K} \mathbf{U} = \sum_{e=1}^N (x_e)^p \mathbf{u}_e^T \mathbf{k}_0 \mathbf{u}_e \\ \text{ect to} : \frac{V(\mathbf{x})}{V_0} = f \\ : \mathbf{K} \mathbf{U} = \mathbf{F} \\ : \mathbf{0} < \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{1} \end{array} \right\},$$

where  $\mathbf{U}$  and  $\mathbf{F}$  are the global displacement and force vectors, respectively,  $\mathbf{K}$  is the global stiffness matrix,  $\mathbf{u}_e$  and  $\mathbf{k}_e$  are the element displacement vector and stiffness matrix, respectively,  $\mathbf{x}$  is the vector of design variables,  $\mathbf{x}_{\min}$  is a vector of minimum relative densities (non-zero to avoid singularity),  $N$  ( $= n_{\text{elx}} \times n_{\text{ely}}$ ) is the number of elements used to discretize the design domain,  $p$  is the penalization power (typically  $p = 3$ ),  $V(\mathbf{x})$  and  $V_0$  is the material volume and design domain volume, respectively and  $f$  (volfrac) is the prescribed volume fraction.

A practical way to solve this is to use a trick on the Young's modulus:

$$E_e = E_{\min} + x_e^p (E_0 - E_{\min})$$

so that we can rewrite

$$c(x) = \mathbf{U}^T \mathbf{K} \mathbf{U} = \sum_{e=1}^N E_e (x_e)^p \mathbf{u}_e^T \mathbf{k}_0 \mathbf{u}_e$$

$$C = \mathbf{f}^T \mathbf{u} = \mathbf{u}^T \mathbf{K} \mathbf{u}$$

The cheap derivatives (use chain rule) can be written as:

$$\frac{dC}{dx} = 2\mathbf{u}^T K \frac{d\mathbf{u}}{dx} + \mathbf{u}^T \frac{dK}{dx} \mathbf{u}$$

But since  $\mathbf{K} \mathbf{u} = \mathbf{f}$  if  $\mathbf{f}$  does not depend on  $x$

$$\begin{aligned} K \frac{d\mathbf{u}}{dx} &= -\frac{dK}{dx} \mathbf{u} \\ \frac{dC}{dx} &= -\mathbf{u}^T \frac{dK}{dx} \mathbf{u} \end{aligned}$$

Knowing displacements you also know the gradients !!!

Since the design variable will be controlling the stiffness of one element, we can replace  $K$  by the element stiffness matrix and replace the global displacement vector by the element nodal displacements.

$$\text{Recall } \frac{dC}{dx} = -\mathbf{u}^T \frac{dK}{dx} \mathbf{u}$$

we can write for density variables

$$\frac{dC}{d\rho'} \propto -\mathbf{u}^T \rho^{p-1} K' \mathbf{u}$$

The equation gives us the directive that we want to increase the density of elements with high strain energy (or high stresses) and decrease it for elements with low strain energy (low stresses).

In practice, we use the Sensitivity analysis and especially the adjoint method

$$\Phi = \Phi(\mathbf{U}(\boldsymbol{\rho})), \quad \mathbf{K}(\boldsymbol{\rho}) \mathbf{U} = \mathbf{F}$$

We adjoint the constraint rewriting:

$$\Phi = \Phi + \lambda^T (\mathbf{K} \mathbf{U} - \mathbf{F})$$

### 4.3 The adjoint method

- A general function and a general residual:

$$\Phi = \Phi(\boldsymbol{\rho}, \mathbf{u}(\boldsymbol{\rho})), \quad \mathbf{R}(\boldsymbol{\rho}, \mathbf{u}(\boldsymbol{\rho})) = \mathbf{0}$$

- Step 1: differentiate using the chainrule

$$\frac{d\Phi}{d\rho_e} = \frac{\partial\Phi}{\partial\rho_e} + \frac{\partial\Phi}{\partial\mathbf{u}} \frac{\partial\mathbf{u}}{\partial\rho_e} \quad \frac{d\mathbf{R}}{d\rho_e} = \frac{\partial\mathbf{R}}{\partial\rho_e} + \frac{\partial\mathbf{R}}{\partial\mathbf{u}} \frac{\partial\mathbf{u}}{\partial\rho_e} = \mathbf{0}$$

- Problem term - must be eliminated! - Use the residual eqs.:  $\frac{\partial\mathbf{u}}{\partial\rho_e} = -\left(\frac{\partial\mathbf{R}}{\partial\mathbf{u}}\right)^{-1} \frac{\partial\mathbf{R}}{\partial\rho_e}$
- Step 2: Insert trouble term into derivative

$$\frac{d\Phi}{d\rho_e} = \frac{\partial\Phi}{\partial\rho_e} + \underbrace{\frac{\partial\Phi}{\partial\mathbf{u}} \left(-\frac{\partial\mathbf{R}}{\partial\mathbf{u}}\right)^{-1}}_{\boldsymbol{\lambda}^T} \frac{\partial\mathbf{R}}{\partial\rho_e}$$

- Step 3: Adjoint problem

$$\boldsymbol{\lambda}^T = -\frac{\partial\Phi}{\partial\mathbf{u}} \left(\frac{\partial\mathbf{R}}{\partial\mathbf{u}}\right)^{-1} \Rightarrow \frac{\partial\mathbf{R}^T}{\partial\mathbf{u}} \boldsymbol{\lambda} = -\frac{\partial\Phi}{\partial\mathbf{u}}$$

- Final sensitivity

$$\frac{d\Phi}{d\rho_e} = \frac{\partial\Phi}{\partial\rho_e} + \boldsymbol{\lambda}^T \frac{\partial\mathbf{R}}{\partial\rho_e}$$

- Example problem - Linear compliance

$$\Phi = \mathbf{F}^T \mathbf{u} = \mathbf{u}^T \mathbf{K} \mathbf{u}, \quad \mathbf{R} = \mathbf{K}(\rho) \mathbf{u} - \mathbf{F} = \mathbf{0}$$

- The 4 required terms become

$$\begin{aligned} \frac{\partial\Phi}{\partial\rho_e} &= \mathbf{u}^T \frac{\partial\mathbf{K}}{\partial\rho_e} \mathbf{u} & \frac{\partial\Phi}{\partial\mathbf{u}} &= 2\mathbf{F} \\ \frac{\partial\mathbf{R}}{\partial\rho_e} &= \frac{\partial\mathbf{K}}{\partial\rho_e} \mathbf{u} & \frac{\partial\mathbf{R}}{\partial\mathbf{u}} &= \mathbf{K} = \mathbf{K}^T \end{aligned}$$

- The adjoint becomes (so-called self-adjoint!):

$$\mathbf{K}(\rho) \boldsymbol{\lambda} = -2\mathbf{F} \Rightarrow \boldsymbol{\lambda} = -2\mathbf{u}$$

- The sensitivity now reads

$$\begin{aligned} \frac{d\Phi}{d\rho_e} &= \mathbf{u}^T \frac{\partial\mathbf{K}}{\partial\rho_e} \mathbf{u} - 2\mathbf{u}^T \frac{\partial\mathbf{K}}{\partial\rho_e} \mathbf{u} = -\mathbf{u}^T \frac{\partial\mathbf{K}}{\partial\rho_e} \mathbf{u} \\ \text{with: } \frac{\partial\mathbf{K}}{\partial\rho_e} &= p\rho^{p-1} (E_{\max} - E_{\min}) \mathbf{K}_0 \end{aligned}$$

- Note: this is a negative scaled strain energy

## 4.4 MDAO A short story

Coupling simulation codes to perform aeroelasticity analysis (for example) is the first step of the MDO methodology. No MDO without MDA? (A stands for Analysis). A simple example of MDA consists in creating a model with a scalar input  $x$ , two "disciplines" that define the state variables ( $y_1$ ,  $y_2$ ) and finally a scalar output  $f$ . The equations for each discipline are extracted from [Gray et al, SMO 2019].

(Discipline 1):  $y_1 = y_2^2$

(Discipline 2):  $\exp(-y_1 * y_2) - x * y_2 = 0$

Here above, Discipline 1 computes  $y_1$  explicitly and Discipline 2 calculates  $y_2$  implicitly. The equation for the output of the model  $f$  is (model output):  $f = y_1^2 - y_2 + 3$

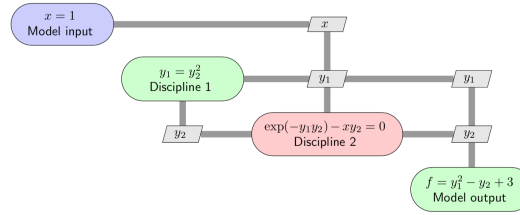


Figure 4: I/O relationship in a MDO problem

We can also depict in figure 5 an already complex 2 disciplines aeroelastic problem available in opensource at <https://github.com/mdolab/OpenAeroStruct>

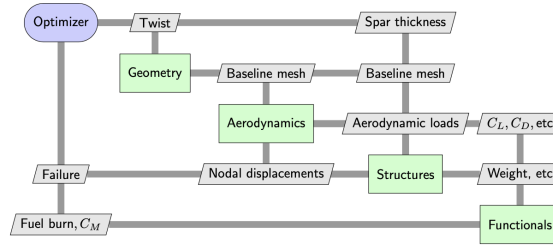


Figure 5: An aerostructure problem

For you what is the hypothesis made on the 2 meshes (fluid and structure)??

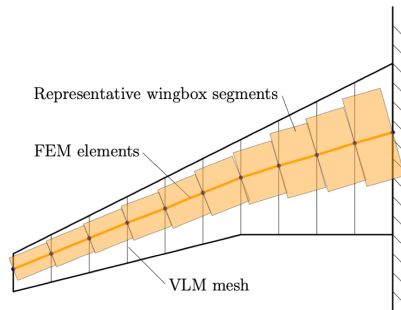


Figure 6: An aerostructure problem with associated simulation tools

The figure 4 above shows the dependencies of the variables using an XDSM diagram. 3 equations, it's a bit too much for popularization ... The resolution of the MDA is done by a fixed point approach (or equivalent). Then, above this MDA, we can consider additionally an objective function (output of the codes to be maximized or minimized) while respecting some constraints (deriving from the outputs of the different codes). That optimization can be performed via different specific MDO methodologies, which often relies on sensitivities (or gradients) computation. OpenMDAO is a computing platform that solves it for you !



At another level (system engineering) we can see that a system is composed of interacting components, and from within the optimal system design paradigm there exist two types of component interconnections: shared variables (  $\mathbf{x}_s$  ), and coupling variables (  $\mathbf{y}$  ). If it were not for these connections, the subproblems could be solved completely independently. Methods such as those discussed in this tutorial provide a means for handling connections.

Shared variables are quantities that are design variables for more than one subproblem. If a quantity is only a design variable for one subproblem, then it is called a local variable (  $\mathbf{x}_\ell$  ). A coupling variable is a quantity calculated by a subproblem analysis that is a required input parameter to at least one other subproblem analysis. If we label the subproblems  $1, 2, \dots, N$ , where the total number of subsystems is  $N$ , then we say that  $\mathbf{y}_{ij}$  is the coupling variable that is an output of subproblem  $j$  and an input to subproblem  $i$ . Coupling variables may be scalar, vector, or function valued. The variables local to subproblem  $i$  are  $\mathbf{x}_{\ell i}$ , and its shared variables are  $\mathbf{x}_{s i}$ . The collection of all design variables for subproblem  $i$  is  $\mathbf{x}_i$ .

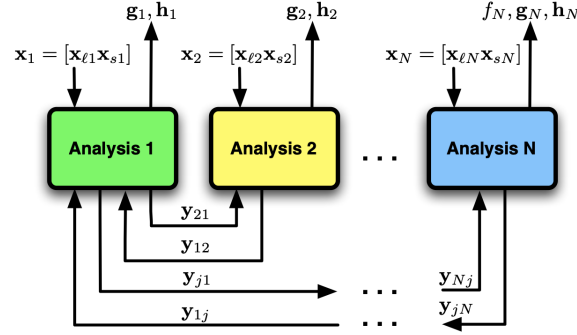


Figure 7: General representation of a partitioned system analysis, Copyright 2006 by James T. Allison

Figure 7 graphically illustrates a partitioned system analysis using the quantities just described, and also additional analysis outputs-  $\mathbf{g}_i$  and  $\mathbf{h}_i$ . For convenience in notation, vectors are assumed to be row vectors. These additional outputs are constraint or objective functions required for the optimization subproblem, but not for system analysis (as are the coupling variables). Note that not all of the depicted  $N(N - 1)$  couplings necessarily exist. Also observe that some couplings may exist in reality, but approximate models used for optimization may not account for them.

It is assumed here that several subsystem optimization problems have been defined, an example of which is shown in equation 1 for subproblem  $i$ . Each subproblem has its own objective, local constraints, and design variables. The objective and constraint functions require as input parameters the input coupling variables from the rest of the system, i.e., subproblem  $i$  has as input coupling variables  $\mathbf{y}_{ij}$  where  $j$  takes on all values from 1 to  $N$  except  $i$ . If the set of all subproblem labels is  $\mathcal{P} = \{1, 2, \dots, N\}$ , we can write this requirement on  $j$  as  $j \in \mathcal{P} \setminus i$ .

$$\begin{aligned} & \min_{\mathbf{x}_i} && f_i(\mathbf{x}_i, \mathbf{y}_{ij}) \\ & \text{subject to} && \mathbf{g}_i(\mathbf{x}_i, \mathbf{y}_{ij}) \leq \mathbf{0} \\ & && \mathbf{h}_i(\mathbf{x}_i, \mathbf{y}_{ij}) = \mathbf{0} \\ & \text{where} && j \in \mathcal{P} \setminus i \end{aligned}$$

With the subproblems defined, the task now is to formulate a system optimization problem that accounts for all subproblems, their interactions (coupling variables), and connections via shared variables. Equation 2 illustrates the complete system optimization formulation. Since it was assumed each subproblem has its own local objective function, the system design problem entails multiobjective optimization. The formulation in equation 2 handles the multiobjective optimization by scalarizing the objectives with a weighted sum, where  $w_i$  indicates the importance of each objective function. Another approach is to select one objective as the overall system objective, and convert the other local objectives to inequality constraints. The system optimization is performed with respect to all design variables, shared and coupled, for all subproblems. The vector of all local variables for all subproblems is  $\mathbf{x}_\ell$ , and the set of all shared variables for all subproblems is  $\mathbf{x}_s$ .

$$\begin{aligned} \min_{\mathbf{x}=[\mathbf{x}_\ell, \mathbf{x}_s]} \quad & \sum_{i=1}^N w_i f_i(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{x}) = [\mathbf{g}_1, \mathbf{g}_2, \dots, \mathbf{g}_N] \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = [\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_N] = \mathbf{0} \end{aligned}$$

Implicit in the above formulation is the task of system analysis, which involves finding the values for coupling variables that result in a consistent system. To illustrate, consider the simple coupled analysis in Figure 8.



Figure 8: Two element coupled system, Copyright 2006 by James T. Allison

Each analysis depends on the other, and we say that the system has feedback coupling. For example, if we want to calculate the value of  $y_{12}$ , we need as an input parameter the value of  $y_{21}$ . However, in order to calculate the value of  $y_{21}$ , we need as an input parameter the value of  $y_{12}$ . A solution to this chicken-and-egg dilemma is to make a guess for one of the values, iteratively solve the analyses with updated coupling variables, and end when the system has reached a stability, or 'fixed' point.

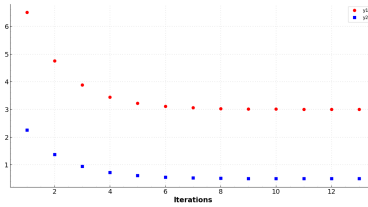


Figure 9: Convergency of the 2 element coupled system by FPI, Copyright 2006 by James T. Allison

This process will converge in many cases (see 9, but it has been shown that some systems will not converge when this process is used. Systems with information flow in only one direction are much easier to handle, since system analysis may be completed with only one pass through the subproblems.

We can solve the system optimization problem defined in the previous set of equations by completely solving the system analysis to obtain values for  $\mathbf{f}$ ,  $\mathbf{g}$ , and  $\mathbf{h}$  for every iteration of the optimization algorithm. In other words, we nest the system analysis task within the system optimization task. This approach is called the all-in-one ( AiO ) approach, and is sometimes referred to as nested analysis and design (NAND). The AiO approach is depicted graphically in Figure 10.

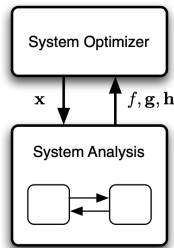


Figure 10: AiO system optimization process, Copyright 2006 by James T. Allison

Although commonly used and straightforward to implement, several problems may be encountered with AiO . If a system is strongly coupled, then the system analysis task may require numerous costly analysis iterations, resulting in a very long computation time. In addition, fixed point iteration may fail to converge. Finally, the dimension of the optimization problem may be too large to solve.

## 4.5 A list of open source optimizers

Because we have an optimization method with an objective and a single constraint (volume) it can be done with a simple specialized optimization method, called the optimality criterion method easy to code in few lines of code. Several solvers are commonly used. These solvers help in solving the optimization problem efficiently:

- 1. MMA (Method of Moving Asymptotes) – One of the most popular solvers for topology optimization, particularly effective for large-scale problems.
- 2. SLSQP (Sequential Least Squares Quadratic Programming) – A gradient-based optimizer that can handle constraints effectively.
- 3. IPOPT (Interior Point Optimizer) – A nonlinear optimization solver that works well with adjoint-based sensitivity analysis.
- 4. NLOPT (with MMA and SLSQP options) – A library of nonlinear optimizers that includes both MMA and SLSQP.
- 5. CasADi <https://web.casadi.org/docs/>
- 6 OpenMDAO <https://openmdao.github.io/PracticalMDO/intro.html>

## 5 References

- Lectures of Linear Algebra <https://github.com/weijie-chen/Linear-Algebra-With-Python/>
- Numerical optimization <https://indrag49.github.io/Numerical-Optimization>
- Numerical optimization in Julia <https://github.com/qzhu2017/Numerical-Optimization>
- Engineering Optimization <https://github.com/danielrherber/engineering-optimization-examples/>
- Engineering Design Optimization <https://mdobook.github.io>
- Matrix Calculus for Machine Learning and Beyond by Bright et al
- Linear Algebra and Its Applications by Gilbert Strang
- Linear Algebra and Its Applications by David Lay
- Introduction to Linear Algebra With Applications by DeFranza and Gagliardi
- Linear Algebra With Applications by Gareth Williams
- Algorithms for Optimization by Mykel J. Kochenderfer and Tim A. Wheeler.
- Numerical Optimization by Jorge Nocedal and Stephen J. Wright
- Design impact <https://www.youtube.com/@designimpact2178>
- What is Multidisciplinary Design Optimization?<https://medium.com/optimise-consultancy/what-is-multidisciplinary-design-optimisation-88e6e6b933b3>
- Solving optimization problem with JAX  
<https://github.com/mazy1998/Solving-Optimization-Problems-with-JAX>.