EMSM-207: RECIPES FOR ENGINEERING DESIGN OPTIMIZATION

Prof. Joseph Morlier

March 28, 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/optimaldesign

Contents

1 Warm up

1.1 Scalar product?

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

$$< \boldsymbol{x}, \boldsymbol{y} >= x_1 y_1 + x_2 y_2 + \ldots + x_m y_m = \boldsymbol{x}^T \boldsymbol{y} = \boldsymbol{y}^T \boldsymbol{x}$$

The norm of a vector can be defined as $\|x\| = \sqrt{x^T x}$.
It is possible to show the Schwarz' inequality hold. $\left|x^T y\right| \leq \|x\| \|y\|$

1.2 Revisiting multivariable calculus

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

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

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 $\mathbf{R}^n \to \mathbf{R}^m$) where A is a constant $m \times n$ matrix. Then,

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

We have $dA = 0$ here because A does not change when we change x.
Let $f(x) = x^T Ax$ (mapping $\mathbf{R}^n \to \mathbf{R}$). Then,

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

and hence $f'(x) = x^T \left(A + A^T\right)$. (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 $\nabla f = \left(A + A^T\right) x$ as before (which simplifies to $2Ax$ if A is symmetric).

1.3 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 $x = \begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix}^T, y = \begin{bmatrix} y_1 & y_2 & \dots & y_n \end{bmatrix}^T$, and $\boldsymbol{A}$ 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)?

1xm x mxn x nx1 = 1x1; i.e. a scalar

How can you write this $x^T Ay = y^T A^T x$?

The Bilinear form is a scalar (1x1). So I can write $scalar = scalar^T$
HINTS: use $(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$
$x^T \mathbf{A} y = (x^T \mathbf{A} y)^T = (\mathbf{A} y)^T (x^T)^T = y^T \mathbf{A}^T x$

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

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

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

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

1.4 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 ($A = A^T$), 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!):

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

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

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

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!

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

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

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

This leads to:

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

1.5 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 $\mathbf{x}^T A \mathbf{x}$.

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

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_ix_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.

```
import simpy as sy

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

Listing 1: Python example

The result is exactly the same as we derived

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

1.6 go deeper

Consider $f(x) = x^T Ax$ 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 ∇f.

We can do this directly from the definition.

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

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

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 $f(x) = x^T Ax = \sum_{i,j} x_i A_{i,j} x_j$. 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.

2 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:

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

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,

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

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

2.1 All starts with Gradient

Definition 1.9 For a differentiable objective function $f(\mathbf{x}) : \mathbf{R}^n \to \mathbf{R}$, its gradient vector given by $\nabla f(\mathbf{x}) : \mathbf{R}^n \to \mathbf{R}^n$, 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.

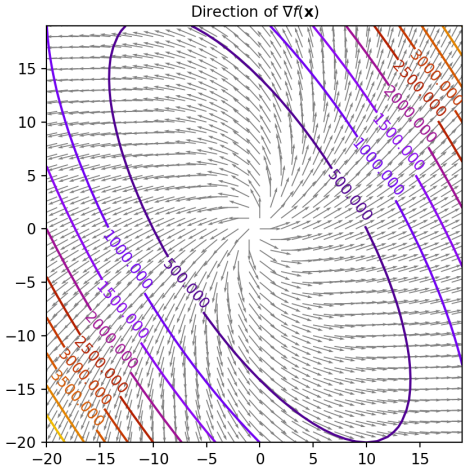

Figure 1: A nice function with his gradient. What is the meaning of the arrows ?

Definition 1.10 For a twice continuously differentiable function $f : \mathbf{R}^n \to \mathbf{R}$, 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, $\mathbf{J}\mathbf{F}(\mathbf{x}) = \begin{pmatrix} \frac{\partial \mathbf{F}}{\partial x_1} & \cdots & \frac{\partial \mathbf{F}}{\partial x_n} \end{pmatrix}$. 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

```
import autograd.numpy as au
from autograd import grad, jacobian

p = np.array([1, 2, 3], dtype=float)
def f(x): # Objective function
    return 2*x[0]*x[1]**3+3*x[1]**2*x[2]+x[2]**3*x[0]
grad_f = grad(f) # gradient of the objective function
hessian_f = jacobian(grad_f) # Hessian of the objective function

print("gradient vector:",grad_f(p))
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.]]
```

2.2 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 $f : \mathbf{R}^n \to \mathbf{R}$, 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).

2.3 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 $\mathbf{H}f(\mathbf{x})$.

2.4 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 x is in a bowl.

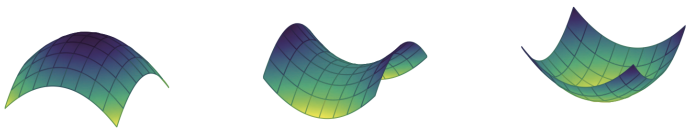

Figure 2: 3 kinds of zone where the gradient is zero?

- 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:

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

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

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

2.5 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:

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

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

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

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

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

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.

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

Listing 3: 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.

```

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

Listing 4: Python example

```
array([0, 0])
```

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

2.6 SONC

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

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

2.7 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 δ, then the conditions:

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

i.e,

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

and

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

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,

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

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:

```
def Himm(x):
    return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2

x_star = np.array([3, 2], dtype='float') #local minimizer
print("function at x_star:", Himm(x_star))
```

Listing 5: 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,

```
# import the necessary packages
import autograd.numpy as au
from autograd import grad, jacobian

# gradient vector of the Himmelblau's function
Himm_grad=grad(Himm)
print("gradient vector at x_star:", Himm_grad(x_star))

# Hessian matrix of the Himmelblau's function
```

Listing 6: Python example

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

```
Himm_hess = jacobian(Himm_grad)
M = Himm_hess(x_star)
eigs = np.linalg.eigvals(M)
print("The eigenvalues of M:", eigs)
if (np.all(eigs>0)):
    print("M is positive definite")
elif (np.all(eigs>=0)):
    print("M is positive semi-definite")
else:
    print("M is negative definite")
```

Listing 7: Python example

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

We see that x1 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.

3 Constrained optimization

3.1 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

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

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 (= nelx × nely) 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
$c(x) = U^TKU = \sum_{e=1}^{N} E_e\,(x_e)^p\,u_e^Tk_0u_e$

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

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

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

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

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

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.

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

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

We adjoint the constraint rewriting:

$$\Phi = \Phi + \lambda^T(KU - F)$$

3.2 The adjoint method

- A general function and a general residual:

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

- Step 1: differentiate using the chainrule

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

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

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

- Step 3: Adjoint problem

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

- Final sensitivity

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

- Example problem - Linear compliance

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

- The 4 required terms become

$$\begin{array}{ll} \frac{\partial \Phi}{\partial \rho_e} = \boldsymbol{u}^T \frac{\partial \boldsymbol{K}}{\partial \rho_e} \boldsymbol{u} & \frac{\partial \Phi}{\partial \boldsymbol{u}} = 2\boldsymbol{F} \\ \frac{\partial \boldsymbol{R}}{\partial \rho_e} = \frac{\partial \boldsymbol{K}}{\partial \rho_e} \boldsymbol{u} & \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{u}} = \boldsymbol{K} = \boldsymbol{K}^T \end{array}$$

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

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

- The sensitivity now reads

$$\frac{\mathrm{d}\Phi}{\mathrm{d}\rho_e} = \boldsymbol{u}^T \frac{\partial \boldsymbol{K}}{\partial \rho_e} \boldsymbol{u} - 2\boldsymbol{u}^T \frac{\partial \boldsymbol{K}}{\partial \rho_e} \boldsymbol{u} = -\boldsymbol{u}^T \frac{\partial \boldsymbol{K}}{\partial \rho_e} \boldsymbol{u}$$
$$\text{with: } \frac{\partial \boldsymbol{K}}{\partial \rho_e} = p\rho^{p-1} \left(E_{\max} - E_{\min}\right) \boldsymbol{K}_0$$

- Note: this is a negative scaled strain energy

3.3 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.

4 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