CEE 251L. Uncertainty, Design, and Optimization

Department of Civil and Environmental Engineering
Duke University

Homework 2 - search methods. due: Wednesday, January 21, 2026

This course considers *engineering design* as an application of *engineering analysis* in which the design specification (i.e., the design plan) is quantified by a set of *design variables*

$$\boldsymbol{v} = \left[\; v_1, \;\; v_2, \;\; v_3, \;\; \cdots, \;\; v_n \; \right]$$

and through which performance, utility, safety, equity, and sustainability can be quantified and optimized or constrained.

A design that meets performance, utility, safety, equity, and sustainability requirements can be called *functional.* In the approach adopted for this course one of these design requirements is selected to be the primary design objective, and is to be optimized. This is quantified by an *objective function*, which is a function of the design variables.

$$f(v_1, v_2, \cdots, v_n) \qquad \text{or, more compactly,} \qquad f(\boldsymbol{v}) \; ,$$

by collecting the n individual design variables into a single vector $\boldsymbol{v}$. A common objective is to simply minimize the total cost of the design, possibly including externalities. In other design problems the objective could be to maximize an aspect of performance.

The functionality of a design depends on more than the primary objective (performance, profits, etc.). Most designs must also meet a number of other criteria (e.g., reliable enough, safe enough, stable enough, strong enough, equitable enough, sustainable enough). These criteria also depend upon the values of the design variables, and are expressed as inequalities. By convention, a set of m inequality constraints can be written as

$$\begin{array}{rcl} g_1(v_1, v_2, v_3, \cdots, v_n) & \leq & 0 \\ g_2(v_1, v_2, v_3, \cdots, v_n) & \leq & 0 \\ & \vdots & \\ g_m(v_1, v_2, v_3, \cdots, v_n) & \leq & 0 \end{array} \qquad \text{or, more compactly,} \qquad \boldsymbol{g}(\boldsymbol{v}) \leq \mathbf{0} \; ,$$

by collecting the m individual inequalities into a single vector inequality. Design constraints confine the design variables to domains (or sub-spaces) of *admissible* alternatives.

In this framework, the admissible and optimal design $\boldsymbol{v}^*$ minimizes (or maximizes) the primary design objective while satisfying the design constraints:

$$\underset{v_1, v_2, \ldots, v_n}{\text{minimize}} \; f(\boldsymbol{v}) \qquad \text{such that} \qquad \boldsymbol{g}(\boldsymbol{v}) \leq \mathbf{0} \; .$$

In certain rare cases, we can write simple equations for $f(\boldsymbol{v})$ and $\boldsymbol{g}(\boldsymbol{v})$ and use calculus to derive equations for the constrained optimum. In the vast majority of practical problems, however, these equations are much too complicated to be solved with pencil and paper. In such cases computer-aided *analysis* can automate the evaluation of the objective and admissibility of any particular trial design. Further, computer-aided *optimization* allows designers to automatically iterate on candidate designs in order to converge rapidly to an admissible and possibly "optimal" solution. But don't expect too much from optimization. In very challenging design optimization problems it can be computationally impractical to converge to an admissible or feasible design that is perfectly optimal *and* robust to uncertainties.

Sometimes, feasibility suffices; being ok is enough.

1. (5 points) engineering ethics

   Engineering Ethics Question 2

2. (5 points) Martin Luther King day reflection

   On Martin Luther King day, please enjoy time reading something by Martin Luther King, listening to one of his speeches, or participating in a Martin Luther King day event. Write a paragraph about your thoughts on your experience.

3. (5 points) update your copy of `multivarious`

   VS Code > Terminal > New Terminal ... or ... Ctrl+Shift+‘ ... a.k.a. ... Ctrl+˜

   `cd` *to the directory in which you cloned* `multivarious`

   . . . for example . . . `cd ~/Desktop/UDO/Code/multivarious`

   ```
   git pull
   pip install .
   ```

4. (10 points) the Optimized Random Search (`ors`) method

   Consider iterations of the Optimized Random Search algorithm for minimizing a function of several variables.

   In each iteration, call the starting point $\boldsymbol{u}^{(0)}$. The first step from the starting point is

   $$\boldsymbol{u}^{(1)} = \boldsymbol{u}^{(0)} + \boldsymbol{r},$$

   where the components of the step vector $\boldsymbol{r}$ are selected at $\boldsymbol{r}$andom. The magnitude and direction of $\boldsymbol{r}$ are random. The expected value of $||\boldsymbol{r}||$ should be about one-tenth of the “diameter” of the design space. For example, if the upper and lower bounds of (appropriately normalized) design variables are `+1` and `-1`, then the first step of the first iteration should be around `0.2` ... give or take. Since the direction of the first step of each iteration is taken at random, there’s a 50/50 chance that it is an uphill step (in the wrong direction). It’s perfectly ok for the first step to be in the wrong direction.

   The second step is a downhill double-step. If the direction of the first step is uphill, then the second step is in the opposite direction. If the direction of the first step is downhill, then the second step is in the same direction. And the length of the second step (from $\boldsymbol{u}^{(0)}$ to $\boldsymbol{u}^{(2)}$) is twice as long as the first step. The design objective corresponding to these three points $[\boldsymbol{u}^{(0)}, \boldsymbol{u}^{(1)}, \boldsymbol{u}^{(2)}]$ has values $[f^{(0)}, f^{(1)}, f^{(2)}]$.

   The three points $[\boldsymbol{u}^{(0)}, \boldsymbol{u}^{(1)}, \boldsymbol{u}^{(2)}]$ lie on the same line,

   $$\boldsymbol{u}^{(1)} = \boldsymbol{u}^{(0)} + d_1\hat{\boldsymbol{r}} \qquad \text{and} \qquad \boldsymbol{u}^{(2)} = \boldsymbol{u}^{(0)} + d_2\hat{\boldsymbol{r}} \ ,$$

   where $\hat{\boldsymbol{r}}$ is the unit vector in the direction of the first step and d_1 is the coordinate of $\boldsymbol{u}^{(1)}$ from $\boldsymbol{u}^{(0)}$ along $\hat{\boldsymbol{r}}$, $d_1 = ||\boldsymbol{u}^{(1)} - \boldsymbol{u}^{(0)}||$ and $\hat{\boldsymbol{r}} = (\boldsymbol{u}^{(1)} - \boldsymbol{u}^{(0)})/d_1$. And d_2 is the coordinate of $\boldsymbol{u}^{(2)}$ from $\boldsymbol{u}^{(0)}$ along $\hat{\boldsymbol{r}}$, $d_2 = \pm||\boldsymbol{u}^{(2)} - \boldsymbol{u}^{(0)}||$. If the first step is uphill, $d_2 < 0$, and if the first step is downhill, $d_2 > 0$.

If a parabola passing through the three coordinates $[(0, f^{(0)}),\ (d_1, f^{(1)}),\ (d_2, \boldsymbol{u}^{(2)})]$ is concave-up a super-good third step along the same line can be attempted. The coefficients $\boldsymbol{c}$ of the parabola

$$\tilde{f}(d) = c_0 + c_1 d + c_2 d^2$$

solve the linear matrix equation

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & d_1 & d_1^2 \\ 1 & d_2 & d_2^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} f^{(0)} \\ f^{(1)} \\ f^{(2)} \end{bmatrix}$$

If $c_2 > 0$ the parabola is concave-up and is minimized at $d_3 = d^* = -c_1/(2c_2)$, so,

$$\boldsymbol{u}^{(3)} = \boldsymbol{u}^{(0)} + d_3 \hat{\boldsymbol{r}} \ .$$

Quadratic steps can be *super-good* when the quadratic fit *extrapolates* to $\boldsymbol{u}^{(3)}$. Sometimes the quadratic extrapolation succeeds and $u^{(3)}$ is an excellent point. Sometimes the extrapolation overshoots and $u^{(3)}$ doesn't work out.

If the minimum of $\left[f^{(0)},\ f^{(1)},\ f^{(2)},\ f^{(3)}\right]$ is smaller than the best value found in all prior iterations, the starting point for the next iteration is the point that has the lowest objective function value.

$$\boldsymbol{u}^{(0)} = u^{(i^*)} \qquad \text{where} \qquad i^* = \text{argmin}\left[f^{(0)},\ f^{(1)},\ f^{(2)},\ f^{(3)}\right]$$

Otherwise, the algorithm proceeds to the next iteration with the same values in $\boldsymbol{u}^{(0)}$.

The magnitudes of the random steps ($\boldsymbol{r}$) naturally fluctuate randomly from one iteration to the next. The algorithm intentionally reduces the expected magnitude as the solution iteratively improves. Tiny steps and huge steps are very unlikely.

This problem asks you to graphically carry out five iterations of the ORS method. Use a ruler to measure and draw each iteration using the contour plot given on page 5. Start at any vertex of the triangle.

No numbers are involved. So use your best judgment as to how the super-good step (step (3) for $\boldsymbol{u}^{(3)}$) would be carried out. For example, if the "middle point" of $[\boldsymbol{u}^{(0)}, \boldsymbol{u}^{(1)}, \boldsymbol{u}^{(2)}]$ is at a lower contour than the other two points think about how an interpolation to the minimum of a quadratic would lie within the range of these three points.

Quadratic extrapolations might be tricky to do graphically. Feel encouraged to give it your best shot, or to avoid quadratic extrapolations for this assignment. Just know that the potential for *super-good* quadratic *extrapolations* is the secret sauce of the ORS method.

Within an iteration, is the best point of the three points (almost) always improved? (yes/no)

To hand in:

A copy of the figure on page 5 annotated with five iterations of the ORS method.

5. (15 points) the Nelder-Mead Simplex (`nms`) method

Consider iterations of the Nelder-Mead algorithm for minimizing a function of several variables.

(a) Using the figure on page 5, trace out five iterations of the Nelder-Mead algorithm. Each iteration starts with a reflection. Use a straight-edge to draw your triangle-shaped simplexes. Within each triangle write "R" for reflection; "RE" for reflection+extension; "CI" for inside contraction; "CO" for outside contraction; and "S" for shrink.

(b) Within an iteration, is the best point of a "simplex" ever updated? (yes/no)

(c) If a "reflection" point is selected, does the "volume" of the "simplex" remain unchanged? (In 2D, the "simplex" is the triangle and the "volume" of the "simplex" is the *area* of the *triangle.*) (yes/no)

(d) Suppose that after several consecutive contraction steps of the Nelder-Mead method, the three vertices of a 2D simplex become nearly co-linear (or the four vertices of a 3D simplex become nearly co-planar). What implication would this have for subsequent steps?

(e) When enforcing constraint equations via a penalty function, if the optimization algorithm converges to an infeasible point, should you re-try the optimization with a larger or a smaller penalty factor?

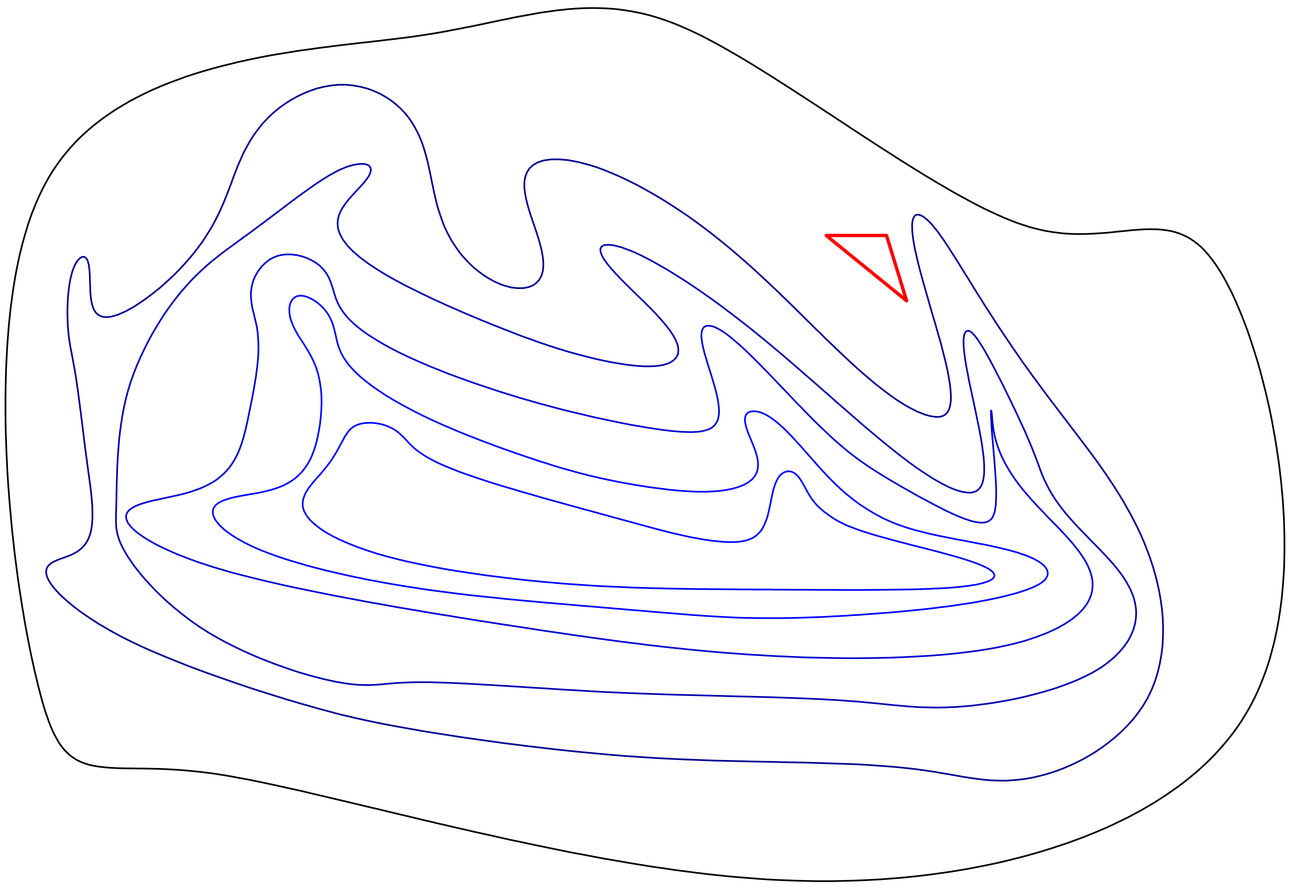
(f) If the optimization routine reaches its maximum iteration limit before converging, what changes to the optimization algorithm options:

```
%            display  tol_v  tol_f  tolG  MaxEvals Penalty  Exponent
  options = [    2    0.01   0.01   0.01  500      1.0      1.0      ];
```

would you try, and why? The meaning of the terms in this vector are explained in the document "An Example of Running Constrained Optimization Codes."

To hand in:

A copy of the figure on page 5 annotated with five iterations of the NMS method.



6. (5 points) convergence metrics and criteria

   In an iteration of the Nelder-Mead method for $n = 5$, the simplex $\boldsymbol{V}$ is:

```
84.80   84.93   84.88   84.92   85.01   85.10
14.89   15.05   14.98   15.12   14.95   15.07
 9.97   10.00    9.99   10.09    9.84    9.94
55.91   55.91   56.28   56.16   56.02   56.04
72.98   72.79   72.89   72.95   72.86   72.96
```

   with corresponding objective function values f

```
99.81   99.93  100.04  100.06  100.10  100.18
```

   Consider convergence tolerances of $\epsilon_v = 0.01$ and $\epsilon_f = 0.01$.
   (a) Does this simplex represent a solution that is converged in the design variables, v?
   (b) Does this simplex represent a solution that is converged in the design objective, f?

7. (40 points) efficiency of numerical optimization methods as compared to a gridded search

   Refer to An Example of Running Constrained Optimization Codes.

   This optimization problem features the possibility of disconnected feasible regions. The objective function in the problem below is one of Nelder's "favorite" functions.

$$\min_{v_1, v_2, v_3} \left[f(v_1, v_2, v_3) = v_1^2 + v_2^2 + v_3^2 + 10^3 \exp(-(v_1^2 + v_2^2 + v_3^2)) \right] \tag{1}$$

   such that

$$g_1(v_1, v_2, v_3) = 0.5 + \cos(\pi v_1/4) \le 0 \tag{2}$$
$$g_2(v_1, v_2, v_3) = 0.5 + \sin(\pi v_2/3) \le 0 \tag{3}$$

   To further constrain the problem, consider design variables within the bounds:
   $-10 \le v_1 \le 10$, $-10 \le v_2 \le 10$, and $-10 \le v_3 \le 10$.

   (a) (10 points) write a `Python` **function**

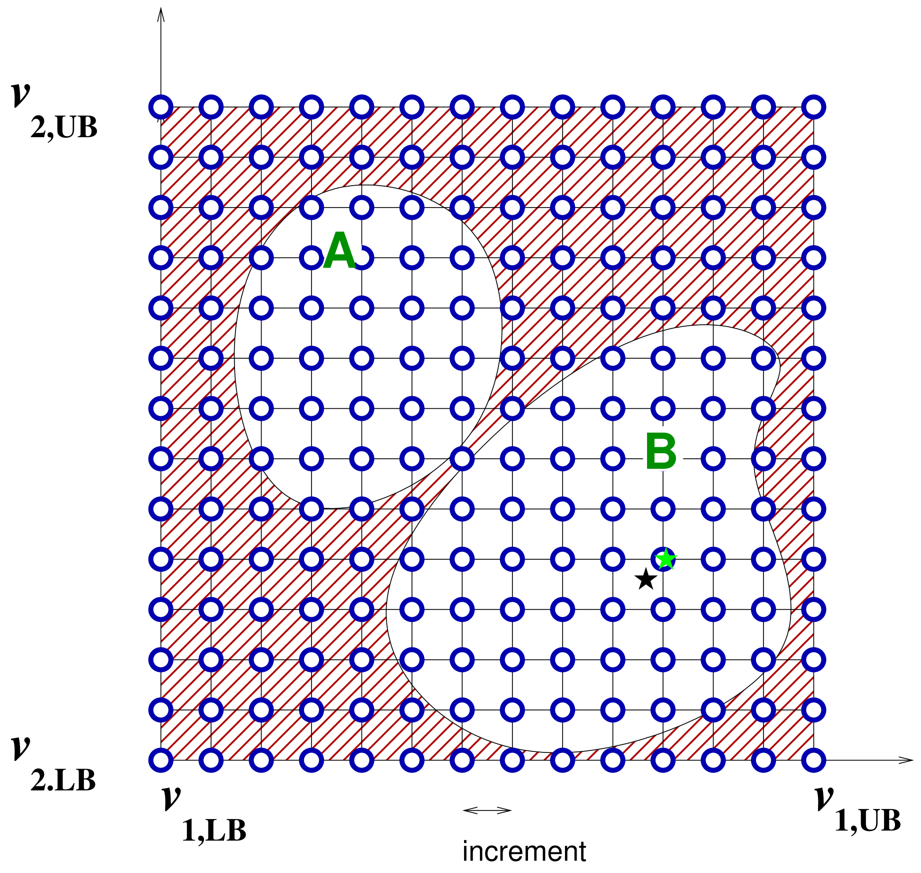
      The `Python` **function** uses a trial set of values for v_1, v_2, and v_3. In a file named
      `udo_HW2P7_2026.py` ,
      write a function named
      `udo_HW2P7_2026_analysis`
      to calculate the objective $f(v_1, v_2, v_3)$, and the constraints, $g_1(v_1, v_2, v_3)$ and $g_2(v_1, v_2, v_3)$, for a given set of values for the design variables (v_1, v_2, v_3). The function starts with the line:
      `def udo_HW2P7_2026_analysis( v, C )`
      This function uses values in a three-element vector `v` along with constants in `C`, if any, to compute a value of the objective function (the scalar `f`), and values of the constraints (the `np.array` g). In this problem, the `Python` array `g` has two elements: `[g(1), g(2)]` This `.py`-function can be as short as around seven to nine lines of code. A complete template is provided on page 7.

(b) (15 points) constrained optimization using a gridded search

A continuous parameter space (v_1, v_2) can be sampled into a finite number of points (shown as blue circles in the figure below) on a grid from $v_{1,\mathsf{LB}}$ to $v_{1,\mathsf{UB}}$ and from $v_{2,\mathsf{LB}}$ to $v_{2,\mathsf{UB}}$. The cost $f(v_1, v_2)$ and constraints $\mathbf{g}(v_1, v_2)$ can be evaluated at every point on the grid. By keeping track of the best feasible point as each point is evaluated, the optimal solution from within the set of grid points can be identified (the green star). In most cases, the true optimal value (black star) is off of the grid. A finer grid would contain a point closer to the true optimal, but would also involve more evaluations of f and g. This idea raises the idea of a method to adaptively refine the grid around (one or more) approximate optimal solutions.

The figure below is representative of problems that have discontinuous feasible regions. A local minimizing routine starting within region A might not move into region B as this would require moving through the infeasible space (hashed). Numerical optimization methods could move from one feasible region into another, if the step size of the update of the design variables is larger than the distance between adjacent feasible regions.

In 3D, the gridded space would be a cube divided into little blocks. The feasible regions would be 3D objects within the 3D grid. The grid shown below is a regular grid in which all values of v_i are uniformly spaced with the same increment.



Write a `Python` **script**.

The `Python` **script**, called `udo_HW2P7_2026.py`, is provided for you on the next page. (You're welcome.) It is a `.py`-file containing `Python` commands that use the analysis function to find the optimal value of the design variables. It implements a method that is terribly inefficient, but is easy-to-understand, easy-to-program, and is guaranteed to find a value that is close to the global optimal value.

The first block of code on page 7 and the first three lines of the second block of code provide a a sketch of the function `udo_HW2P7_2026_analysis():`. Add your code to compute `f` and `g` in the space provided.

Given vectors of values for `v1`, `v2`, and `v3`, the script uses a set of "nested for-loops" to evaluate your function

```
 f, g = udo_HW2P7_2026_analysis( v , C )
```

for *each* and *every* combination of design variables (lines 28-62). In a gridded search, the total number of function evaluations is the product of the length of each parameter-set vector. The script computes the objective `f` and constraints `g` by evaluating the `analysis` function (line 39) and checks to see if `all` elements of `g` are less than zero and if `f` is less than the smallest value of `f` found so far (line 33). If all elements of `g` are less than zero (feasible) and `f` is less than the best value computed so far, the script updates `f_opt_gs = f;` and `v_opt_gs = v;` (lines 39-43). If not, the script simply goes on to the next combination of design variables. After *all combinations* have been tried, the optimal values for the design variables will be `v_opt_gs` and the associated value of the objective function will be `f_opt_gs` (lines 63-66).

Since this method evaluates lots and lots of alternative designs, you may expect the script to take some time to run. The script keeps track of how much time it takes using the commands on line 26 and lines 54 to 59.

Note that cutting-and-pasting the typeset code from this `.pdf` into your `.py` file will result in a lot of spaces where there should be none, especially in the comments. These spaces will cause the code to fail, so, be prepared to look for, and remove, extra spaces.

```
import time
from datetime import datetime, timedelta

import numpy as np
from numpy import pi
from numpy import sin
from numpy import cos
from numpy import exp

from multivarious.opt import ors
from multivarious.opt import nms
from multivarious.opt import sqp
from multivarious.utl import plot_cvg_hst

def udo_HW2P7_2026_analysis( v , C ):
# f, g = udo_HW2P7_2025_analysis( v , C )
# Evaluate an objective function f(v) and constraints g(v) for problem #7

    v1 = v[0]
    v2 = v[1]
    v3 = v[2]

    ss = v1**2 + v2**2 + v3**2 # sum-squared of v1, v2, v3
```

...add your equations for f and g here ...

```
    return f, g

# ---------------------------------------------------------- udo_HW2P7_2026_analysis

C = 1 # just a place holder, not used in this problem

v_lb = np.array([ -10.0, -10.0, -10.0 ]) # lower bounds for each design variable
v_ub = np.array([  10.0,  10.0,  10.0 ]) # upper bounds for each design variable
increment = 0.1                          # smaller increment, more values to try

if 0: # --- do HW 2  Problem 7b: a gridded paramter search ...
    v1_set = np.arange( v_lb[0], v_ub[0], increment ) # the set of values for v1
    v2_set = np.arange( v_lb[1], v_ub[1], increment ) # the set of values for v2
    v3_set = np.arange( v_lb[2], v_ub[2], increment ) # the set of values for v3

    N_v1 = len(v1_set)                 # the number of v1 values in v1_set
    N_v2 = len(v2_set)                 # the number of v2 values in v2_set
    N_v3 = len(v3_set)                 # the number of v3 values in v3_set

    NumberOfAnalyses = N_v1 * N_v2 * N_v3

    f_opt_gs = 1e9  # initialize a value for objective   f ... some big number
    g_opt_gs = 1e9  # initialize a value for constraints g ... some big number

    function_evals = 0
    start_time = time.time()

    for i1 in range(N_v1):                # loop over values of v1
        for i2 in range(N_v2):            # loop over values of v2
            for i3 in range(N_v3):        # loop over values of v3

                # trial values for v1,v2,v3
                v = np.array([v1_set[i1], v2_set[i2], v3_set[i3]])

                f, g = udo_HW2P7_2026_analysis( v, C )  # run the analysis

                # print(f' f = {f}    g = {g}  ') # for debugging

                if np.all(g<=0) and f < f_opt_gs:  # best solution, so far

                    f_opt_gs = f                   # update objective
                    v_opt_gs = v                   # update variables
                    g_opt_gs = g                   # update constraints

                function_evals = function_evals + 1
                if (function_evals % 100000 == 0): # when will this ever finish?

                    elapsed = time.time() - start_time
                    secs_left = int( (NumberOfAnalyses-function_evals)*elapsed /
                                      function_evals )
                    eta = (datetime.now() +
                              timedelta(seconds=secs_left)).strftime('%H:%M:%S')

                    if f_opt_gs > 1e10:
                        print(f" {function_evals:8d}  "
                              f" {secs_left} seconds to go  (e.t.a. {eta})")
                    else:
                        print(f" {function_evals:8d}  "
                              f" {secs_left} seconds to go  (e.t.a. {eta}) "
                              f" f_opt = {f_opt_gs:6.3f} "
                              f" max_g_opt = {np.max(g_opt_gs):6.3f}")

    print(f' Number of Analyses = {function_evals}')
    print(f' v_opt_gs = {v_opt_gs}') # the best design variables found
    print(f' f_opt_gs = {f_opt_gs}') # the associated objective function value
    print(f' g_opt_gs = {g_opt_gs}') # the associated constraint values

# time.sleep(10)

if 1: # --- do HW 2 Problem 7c: now try the ors, nms, and sqp methods

    v_init = np.array([ 1, 1, 1 ])   # an initial guess for v1, v2, and v3

#             msg  tol_v                        tol_f tol_g  MaxIter Penalty
    opts = [   1,  increment/(v_ub[0]-v_lb[0]), 1e-2, 1e-4,  1000,   100 ]

    v_opt, f_opt, g_opt, cvg_hst, _,_ =
                      ors( udo_HW2P7_2026_analysis, v_init, v_lb,v_ub, opts, C )

    plot_cvg_hst( cvg_hst , v_opt, opts, pdf_plots = True )
```

"Consider everything. Keep the good. Avoid evil whenever you notice it."

(c) (15 points) constrained optimization using `ors`, `nms`, and `sqp`

Solve this problem using the `Python` functions provided in multivarious*!* The methods implemented in these functions keep track of the design variables, the objective function, and the constraints. Each method implements a different approach in determining what the next guess of the optimal value of the design variables should be, based on the information it has developed so far by trying out several different values of the design variables. As information about the problem accumulates with each successive iteration, the methods converge toward an optimal solution. To use these optimization functions, use the same function `udo_HW2P7_2026_analysis():` that you wrote for part (a) and used in part (b). This will be passed to the optimization routines to indicate that the variable is a function name. These methods, like most optimization methods, require a decent initial guess at the design variables. The initial guess of the optimal design variables is specified in the `Python` array `v_init`. The lower and upper bounds of the design variables are given in `Python` arrays `v_lb` and `v_ub`.

To apply the constrained optimization methods in your `.py`-script file, add the following lines at the end of `udo_HW2P7_2026.py` :

```

if 1: # —— do HW 2 Problem 7c: now try the ors, nms, and sqp methods

    v_init = np.array([ 1, 1, 1 ])   # an initial guess for v1, v2, and v3

#              msg   tol_v                            tol_f  tol_g   MavIter  Penalty
    opts = [   1,  increment/(v_ub[0]-v_lb[0]), 1e-2, 1e-4,  1000,    100 ]

    v_opt, f_opt, g_opt, cvg_hst, _,_ =
                         ors ( udo_HW2P7_2026_analysis, v_init, v_lb,v_ub, opts, C )

    plot_cvg_hst( cvg_hst , v_opt, opts, pdf_plots = True )
```

Note here that the convergence tolerance on the variables, `tol_v`, is set to `increment / (v_ub[0] - v_lb[0])` used in the gridded search. This value is the precision of the gridded search relative to the span of the search domain. Setting `tol_v` in this way makes for a level comparison between the efficiency of the gridded search and more advanced optimization methods. Normally, smaller values of `tol_v` would be used. Try each of the three optimization functions. Try various values in the initial design variable vector `v_init` to see if the computed solution depends on the initial design variables. Keep track of the total number of function evaluations made in each of the three methods. (This information is displayed in the Terminal window.) Which (if any) of the constraint inequalities are binding the optimum point?

To hand in:

(a) your *well-commented* `.py`-function `udo_HW2P7_2026_analysis.py`
Every `Python` file that you write should be *well-commented.* In the first lines of the `.py`-file, write comments that tell a user about your `.py`-file ... what it does, how to use it, your name, your e-mail, and the date.

(b) The tables below, filled in. Run two analyses, first with `increment = 1.0` and again with `increment = 0.1`. In each row, circle the value of the constraint(s) (g_1, g_2) that bind(s) the optimum point.

Gridded Search Increment Value = 1.0
number of function evaluations = ________________________

optimal values						compute
f^*	g_1^*	g_2^*	v_1^*	v_2^*	v_3^*	time

Gridded Search Increment Value = 0.1
number of function evaluations = ________________________

optimal values						compute
f^*	g_1^*	g_2^*	v_1^*	v_2^*	v_3^*	time

Which case resulted in a more-negative (lower) f^* , and why does that make sense?

(c) To complete the table below, run three analyses using each of the three methods `ors`, `nms` and `sqp`, using different initial values of the design variables each time. In each row, circle the value of the constraint(s) (g_1, g_2) that bind(s) the optimum point.

	initial values			optimal values						number of
method	v_1	v_2	v_3	f^*	g_1^*	g_2^*	v_1^*	v_2^*	v_3^*	analyses
`ors`	5	0	0							
`ors`	0	5	0							
`ors`	0	0	2							
`nms`	5	0	0							
`nms`	0	5	0							
`nms`	0	0	2							
`sqp`	5	0	0							
`sqp`	0	5	0							
`sqp`	0	0	2							

In a sentence or two discuss the similarities and differences among the nine solutions.