6.8420: Computational Design and Fabrication
Charlotte Folinus
Massachusetts Institute of Technology
Spring 2024

Homework 5: Design Optimization

Section 1: Performance Exploration

Problem 1.1: Come up with an $O(n \log n)$ algorithm for finding the Pareto front of 2D points in the `pareto_front` function in `pareto.py`. You can run `python pareto.py -n 10` to run your algorithm against random points or `python pareto.py -f data/points 0.npy` to run against the provided test cases (see the `data` folder). The code will also output an image in the output folder. Run `run.sh` to run against all test cases. Describe the algorithm in your report, justifying your runtime bounds and highlighting any edge cases you handled in your implementation. Include all the images in the write-up or include them in the zip file.

In my algorithm, I first sort all of the points in ascending order by the first column (0th objective, the "x" value), then by the second column (1st objective, the "y" value) using `np.lexsort()`. After initializing storage structure's for information about the pareto front (a list of indices corresponding to pareto-optimal points, and the XY value of the most recently-added point), I traverse the sorted array of points, starting with the lowest-x value/first entry. If this point's y-value is higher performance (lower magnitude) than the last Pareto-optimal point, then the point is nondominated. If a point is nondominated, I update the list of pareto-optimal indices as well as the variables corresponding to the XY values of the most recently-added pareto-optimal point. I explicitly considered the edge-case where two points share the same X value but differ in their Y values; I considered this edge case by checking whether the current x-value is the same as the most recently-added pareto-optimal point. If this was the case AND the current point had a better y-value, then I would remove the last point from the list of pareto-optimal points, add the current point, and update the most recently-added XY values.

The complexity-determining portion of my algorithm comes from the call to `np.lexsort`, which I believe is built on Timsort, which has an average and worst-case complexity of $O(n \log n)$ (if the data is already sorted, then Timsort can have a better complexity by making fewer comparisons, but this is not the case for us).
The output results are included in Fig. 1 and Fig. 2.

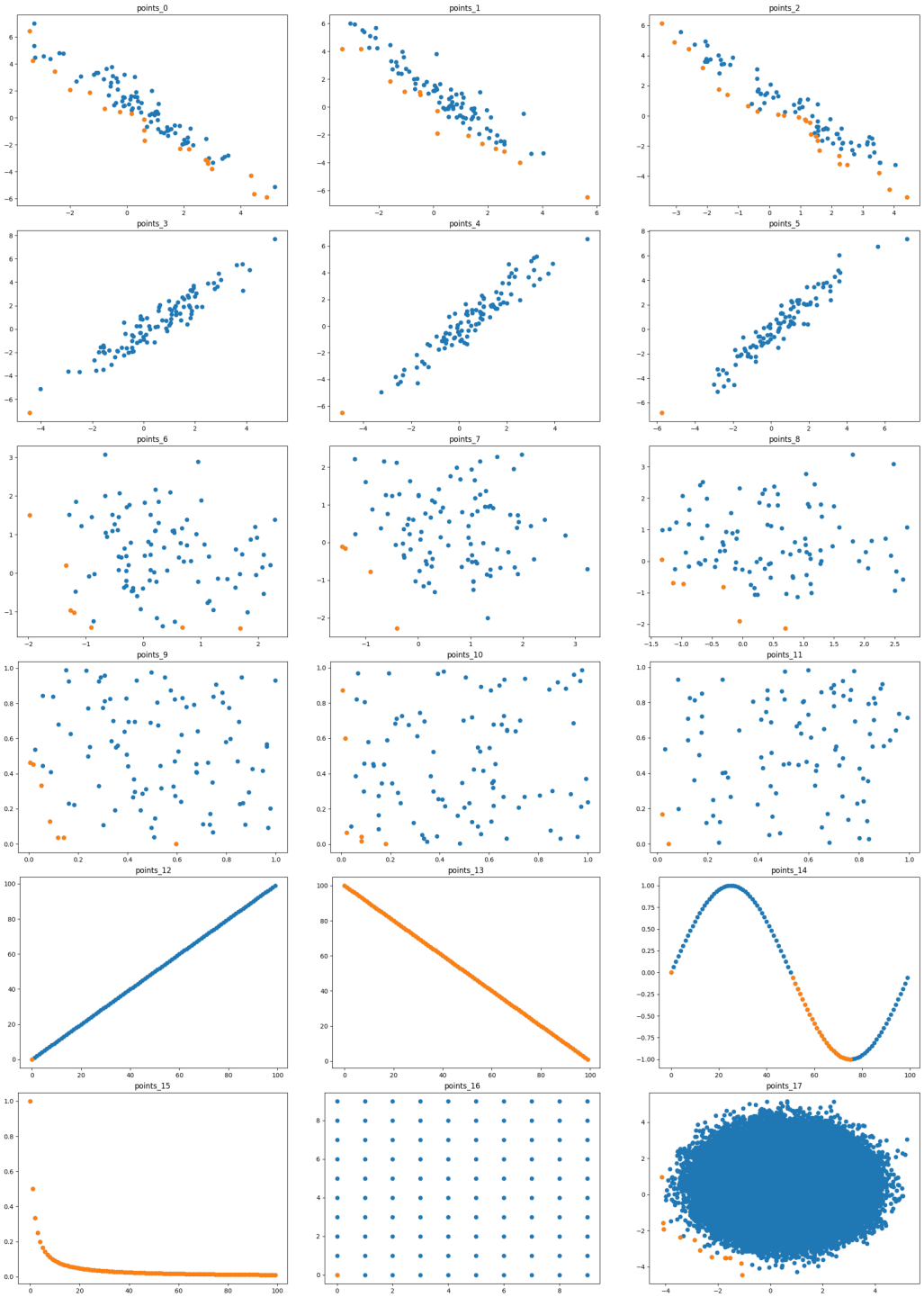


Figure 1: Auto-generated output plots for running the default test cases in `run.sh`.

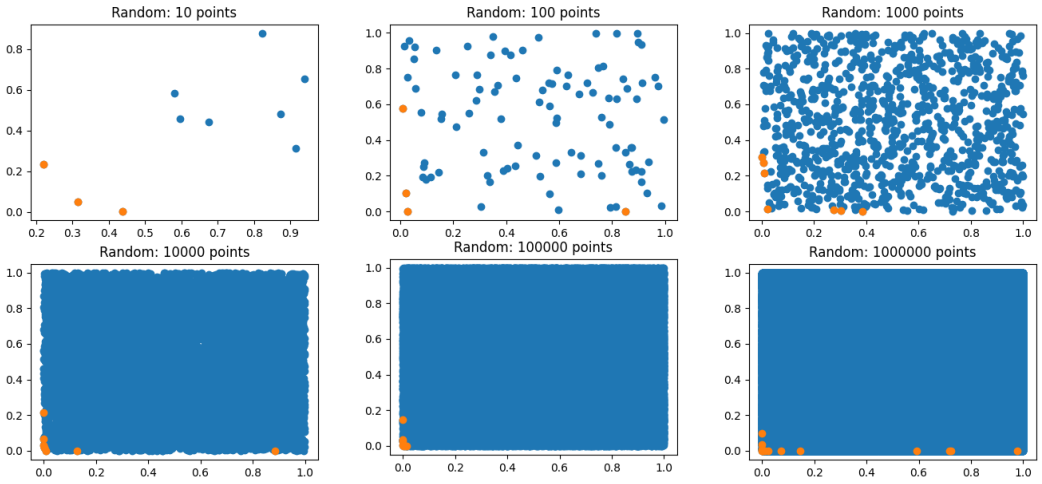


Figure 2: Auto-generated output plots for running `pareto.py` with randomly-generated points.

Section 2: End-to-End Design and Optimization Examples

Section 2.1: Discrete Optimization with MCMC

Problem 2.1.1: Define the design and action space. Your first goal is to define how you will represent a LEGO assembly (Hint: there are two types of LEGOs, and you may consider discretizing the image into a 2D grid to help define the positions of each LEGO). The second thing you want to define is the action space (Hint: you may want to consider adding/removing a LEGO at each iteration block). Importantly, this design space does not need to ensure that anything you specify satisfies physical constraints because you can build checkers for those in the next step (remember how in the knapsack problem if we had an infeasible solution we just rejected it).

I discretized the 2D design space into a pixelized grid corresponding to one-unit in lego dimensions (a two-block would take up two pixels/units in my grid). Pixels can store whether or not a block is present. When a block is placed (or removed) in one of the pixels, the pixels corresponding to the rest of the block will also be activated (or deactivated). My action space is defined as placing a two-block, placing a four-block, or removing a block (with some probability).

At each iteration, I will decide whether to remove a block (with say 50% probability, but we can tune this parameter), to place a two-block (25%), or to place a four-block (25%). I will uniformly and at random decide whether to place the block amongst the exposed pixels on the currently-occupied layers (and the "empty" layer above them). If I am removing a block, I will uniformly and at random decide which of the already-placed blocks to remove). If we were to assemble in the order of the iterations we go through, this might require temporarily separating the structure along its layers, removing the block, and re-attaching the structure. This seems reasonable and physically feasible.

I will evaluate “[resembling] the image as closely as possible" by initially converting the input silhouette into a binary region of interest/region mask on my grid. If the pixel's central subpixel (on, say, a 5x5 subgrid) is inside the silhouette, then that pixel will be considered in the region of interest. I will store whether each pixel is within/beyond the region of interest using binary values.

Problem 2.1.2: Constraint Checking. At each iteration, after performing an action, you might end up with a design that cannot be physically realized or that does not meet the constraints of connecting to one piece. These actions should be rejected. Based on how you defined the design and action space, determine what needs to be checked and propose algorithms for checking them. Write pseudocodes for these algorithms as functions that you will call in Step 3.

I want to check that each block is either connected to a block on the row below it or hanging from a block on the row above it (though more delicate/fragile, this can still

be physically-realized). By default, the bottom-most row will be valid (supported by the ground), and I will check going from the ground-up. As soon as a design is invalid, I will stop checking the higher layers.

```
def checkConstraints():

    # Initialize design as fully-true and valid
    is_valid_design = True

    # Loop over layers, starting at second-layer from the bottom
    for layer_index in range(1, num_layers):
        # Extract current information about which blocks are in the
        ↪  current layer
        # Each block will correspond to pixel positions on the grid

        try:
            for block in layer:
                # Check attachment to lower layer

                # Get state of pixels at same X position in the
                ↪  (layer_index - 1) layer
                lower_layer_states = ...

                # Check that at least one of these is full (occupied)
                is_lower_connected = any(lower_layer_states)

                # If the block is not supported on the bottom, see if
                ↪  it hangs from the top
                if not(is_lower_connected):
                    # Get state of pixels at same X position in the
                    ↪  (layer_index + 1) layer
                    upper_layer_states = ...

                    # Check that at least one of these is full
                    ↪  (occupied)
                    is_upper_connected = any(upper_layer_states)

                    # If upper is not connected, then the design is
                    ↪  invalid
                    is_valid_design = is_upper_connected

                if not(is_valid_design):
                    break
```

```
            return is_valid_design

        except:
            return is_valid_design
```

Problem 2.1.3: Write MCMC pseudocode for optimizing the LEGO assembly given a 2D shape (Hint: consider using annealing).

```
def runMcmc():
# Determine corresponding pixel-grid of input silhouette
x_goal = pixelizeInputVector()

# Start with an initially-empty grid
x = zeros((grid_size, grid_size))
last_pixel_comparison = inf

for k in range (0, k_max):
    # Update temperature
    T = log(k/k_max)

    # Pick an action (0 = remove block, 1 = two-block, 2 =
    ↪  four-block)
    action = randomly_select_action()

    # Based on action, generate and check constraints of new design
    x_new = generateNewDesign(x, action)
    is_valid_design = checkConstraints(x_new)

    if is_valid_design:

        # Evaluate pixel similarity (number of pixels out of place)
        pixel_comparison = sum(x_new - x_goal)

        # Change to this design with some probability
        # If the new design is better, always change
        if pixel_comparison < last_pixel_comparison
            x = x_new
            last_pixel_comparison = pixel_comparison

        else:
            p_change = exp(-(pixel_comparison -
            ↪  last_pixel_comparison)/T)
```

```
            # Change to new design with probability p_change
            accept_new_design = 1 if change, 0 if keep

            if accept_new_design:
                x = x_new
                last_pixel_comparison = pixel_comparison

return x
```

Problem 2.1.4: Note that this sequence of steps guides you in writing a solution where many of the actions you take will be rejected. This is fine, but it means the algorithm may take a very long time to converge. At a high level, discuss potential solutions for this. Note that your solutions don't have to be perfect, and they can involve trade-offs (e.g., you can suggest something that will converge faster but will cover a smaller design space). Please make sure that you emphasize these trade-offs, however.

One solution would be to explicitly optimize within each layer before continuing to the upper layers. We would potentially lose the ability to have "hanging" LEGOs (hanging from the upper layer), but it would allow us to solve a series of much smaller optimization problems. This might allow us to make many fewer actions.
We could also design over a two-unit grid, which would impact the resolution we are able to achieve. For example, in this setup, a two-block could either be located on the left-hand half of a four-block, on the right-hand half, or centered on the four-block; the two-block could not overhang by a single unit of the four-block.

Section 2.2: Continuous Optimization for 3D Printing

Problem 2.2.1: Define the Design Space. Your first goal is to define how you will represent the space, which includes the set of variables and constraints. The variables should define both the information of the central rod and the 3 connecting rods. Hint: consider the relationship between the position of the central rod and the center of mass of the object. Consider the two constraints for how each of the three connecting rods must attach to the central rod: you want to consider the draft angle to avoid support material and you must avoid collisions.

Design variables: I defined a three-variable design space (Fig. 3) that is fully-defined by the 3D location of the central support node (node that connects the three angled rods to the vertically-oriented support rod).

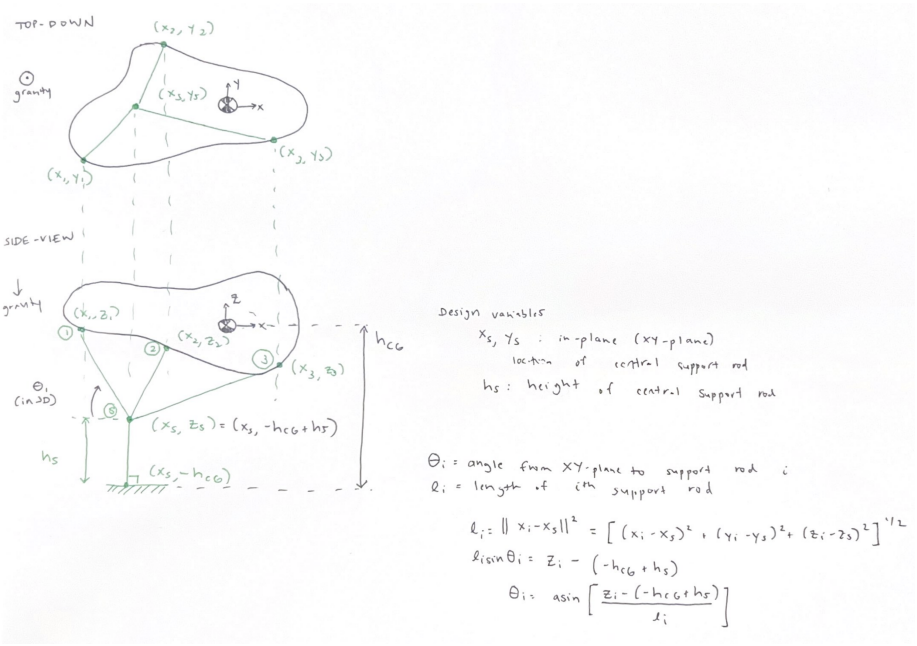


Figure 3: Labeled sketch of design space and definitions of design variables, support rod lengths, and rotation angles.

Constraints: I defined two types of constraints: variable side-bounds and function-valued constraints (constraint functions whose values will depend on the design variables used).

- The central support rod must be located vertically below each of the three connection points (below each z_i, or below $\min(z_1, z_2, z_3)$).
- The central support rod's in-plane location needs to fit within the bounding box of the printer build area. This could be defined as a constraint relative to the three contact node locations, or with an overall size bound on where the support rod can be located (a range (or ranges) of values for x_s and y_s).

- Overhang angles: I defined angles Θ_i from the XY-plane (from the horizontal) to the ith support rod.
  - Each $\Theta_i \geq \Theta_{support}$, where $\Theta_{support}$ is the minimum angle at which support is required.
  - Many optimization libraries require inequality constraints be written as functions with positive values when the constraint is satisfied. Rewritten, this gives a constraint $g(x) \geq \Theta_i - \Theta_{support}$, for each support rod i (three constraints).
- Collisions: The three support rods should also only intersect at the connection point with the central support rod. We can check for this by making sure that each pair of line segments (each pair of support rods) only has one intersection point (using a line-line intersection).

Clarification on assumptions: We are told that the material is infinitely strong, and the graphics show that the support rods connected to a base material. As a result, I assumed that the support rod did not need to be located directly vertically below (same XY location as) the object's center of mass. Placing the central support off-center will produce a net moment about the attachment point to the base. If the material is infinitely stiff/strong, we do not need to worry about this moment causing deformation or yielding in any of the members. Because of the base, we also do not need to worry that this moment will cause the entire system (object-stand) to tip over (provided the base extends below the object's center of mass).

Problem 2.2.2: Express this problem as an optimization problem with pseudocode. This includes expressing a cost function to minimize material usage. Assume you are calling a function that optimizes from a library. Describe what kind of function that will be.

Potential cost function: We are asked to optimize for the total material usage and provided with an input rod radius r_{input} which is the same for all rods (all rods have same cross section area). As a result, optimizing for the total material volume will be equivalent to optimizing for the total length of the rods, $l_{total} = l_1 + l_2 + l_3 + l_support = l_1 + l_2 + l_3 + h_s$.

Potential optimization algorithm: We want to use an optimization algorithm that allows us to include constraints (or, we would need to reformulate our objective function to be a penalty function to penalize invalid designs). We could use an algorithm like COBYLA or SLSQP, both of which do this and are built into optimization library's like `scipy.optimize`.

Potential setup as pseudocode, where coordinates x contain [x, y, z] locations.

```
from library_name import minimize
import numpy as np
import other_required_libraries

# Define helper functions for evaluating constraints and objective
↪  funciton
def calculateRodLength(x_i, x_support):
    # Calculates length of distance (XYZ) between central support rod
    # and contact node i

    l = np.linalg.norm(x_i - x_support)
    return l

def calculateOverhangAngle(x_i, x_support):
    # Calculates overhang angle theta_i

    rod_length = calculateRodLength(x_i, x_support)

    # Use trigonometry to solve overhang angle
    theta_i = asin( ... )
    return theta_i

def calculateIfRodsCollide (x_i, x_j, x_support):
    # Calculate line-line intersection of x_i -> x_support and
    ↪  x_j->x_support
```

```
    are_colliding_rods = 0 if no collision, 1 if collision
    return are_colliding_rods

def calculateTotalRodLength (x_1, x_2, x_3, x_support, h_cg):

    l_1 = calculateRodLength(x_1, x_support)
    l_2 = calculateRodLength(x_1, x_support)
    l_3 = calculateRodLength(x_1, x_support)
    l_support = x_support[2] + h_cg

# Define constraints
# Define overhang angle constraints
def c1():
    # Define constraint c1
    theta_1 = calculateOverhangAngle(x_1, x_support)
    c1 = theta_1 - theta_support
    return c1

def c2():
    # Define constraint c2
    theta_2 = calculateOverhangAngle(x_2, x_support)
    c2 = theta_2 - theta_support
    return c2

def c3():
    # Define constraint c3
    theta_3 = calculateOverhangAngle(x_3, x_support)
    c3 = theta_3 - theta_support
    return c3

# Define collision constraints
def c4():
    # Define constraint c4
    is_collision = calculateIfRodsCollide(x_1, x_2, x_support)
    return c4

def c5():
    # Define constraint c5
    is_collision = calculateIfRodsCollide(x_1, x_3, x_support)
    return c5

def c6():
    # Define constraint c6
    is_collision = calculateIfRodsCollide(x_2, x_3, x_support)
    return c6
```

```
# Initialize design and constraints
# Define variable side bounds
bnds = [(min, max), (min, max), (min, max)]

# Define initial design
# Could randomly select, define as midway in bounds, or use an
↪  educated "guess"
x_0 = [ ... ]

# Define constraints (include each constraint function)
cons = ({'type': 'ineq', 'fun': c1}, ...)

# Call the optimizer
res = minimize(calculateTotalRodLength, x0, bounds = bnds, constraints
↪  = cons, method='COBYLA', options={'gtol': 1e-6, 'disp': True})
```

Section 3: High-Level Understanding

Problem 3.1: Combining Objectives: Give (i) a simple example of what these objective functions could look like mathematically, (ii) a method by which they might be combined into a single objective that can be optimized to sample points along a pareto frontier, (iii) scenarios where this combination method would produce an undesirable pareto frontier, and (iv) a concrete example of this undesirable scenario.

(i) Objective functions (Fig. 4a):

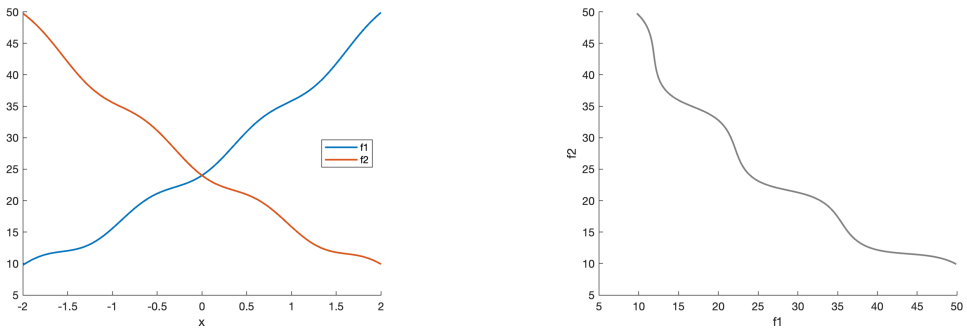
$$f_1(x) = (x+5)^2 - \cos(5(x+5))$$
$$f_2(x) = (x-5)^2 - \cos(5(x+5))$$

(ii) Method by which they might be combined: we could create a combined objective function $f_{combined}$ which is a linear combination (weighted sum) of the two objective functions. By adjusting the relative weight α, we can sample points along the (convex portion of) the pareto front:

$$f_{combined}(x) = \alpha f_1(x) + (1-\alpha) f_2(x)$$

(iii) Undesirable scenarios: this method will only give us the convex portion of the pareto front, but the full pareto front has many non-convex regions (Fig. 4b). If we are interested in the non-convex portions of the pareto front, the simple combined function will not be particulary useful.

(iv) Concrete example: these objective functions could represent things like cost or fabrication time for a production process that has some seasonal/periodic variation (periodic changes) as well long-term trends. We might be interested in designs that have a cost and fabrication time between two points on the convex portion of the pareto front.



(a) (b)

Figure 4: a) Individual objectives f_1 and f_2. b) Plot of tradespace comparing evaluated values of f_2 with f_1, revealing nonconvex regions of the pareto front.

Problem 3.2: Multiple Domains and Bi-level Optimization: Your task is to describe two examples of design spaces for cyber-physical systems: one which can be treated as a single-level optimization and one that cannot, and discuss why. (Hint: consider the case of our walking robot. If we allow the geometry to vary arbitrarily, what does this imply for our motion design space? Can we constrain the design space of valid geometry in a way that makes the overall geometry + motion optimization a single-objective one?)

Single-Level Optimization: walking robots with pre-defined morphologies (geometric constraints)

- Geometry Design Space: We can start with a set of predefined robot shapes, like a quadruped or a biped. By imposing predetermined constraints on the robot's geometry, we constrain the parameter space to a predefined manifold of feasible shapes or structural configurations. This approach markedly reduces the dimensionality of the design space.
- Motion: With the geometry firmly constrained, the motion design space becomes intricately intertwined with the predefined geometric space. Each admissible geometry defines a unique subset within the motion design space, encapsulating feasible locomotion strategies, gait patterns, and kinematic configurations.
- Single-Level Optimization: Because we have defined a joint design space (joint as in "both", not as in mechanical joints!), we're able to express the geometry and the motion-related objectives together. This allows us to optimize in a single-step, without needing to optimize the gait within each geometry.

Bi-Level Optimization: walking robots without geometric restrictions

- Geometry design space: If we don't restrict what geometries can look like for our walking robot, we can have a wide variety of geometric configurations without predefined constraints on how their gaits work. This means our geometry design space includes not only things like limb lengths but also different morphologies, which would be actuated/controlled in a variety of different ways.
- Motion: The unconstrained geometric variability means we have to define a motion design space specific to each geometry. There isn't a straightforward way to express all of our permissible geometries and all of our permissible motions together.
- Bi-Level Optimization: Because of how tightly connected the motion is to the geometry, we need to use a hierarchical optimization approach, where we nest our objectives. The outer optimization loop finds an optimal geometry, while the inner loop iteratively refines motion parameters tailored to the specific geometric configuration, which allows us to eventually find a geometry and motion that work well together.

Problem 3.3: Minimizing Expensive Performance Evaluation: The optimization methods that we have discussed so far involve evaluating the performance objective many times; describe two alternatives to doing so based on the ideas described in class, and discuss when you would use each. (Hint: it may be useful to consider cases where you have some previous expertise or data from performing similar experiments.)

1. **Traditional design of experiments or sampling plans:** There are scenarios where running fabrication/testing procedures is relatively expensive but does not scale linearly (or at all) with the number of specimens made/evaluated in one testing batch. For example, testing requiring the research team to travel out-of-state or out-of-country can be expensive due to the high travel costs, but testing an additional specimen to a given visit may not meaningfully affect testing costs. Without the ability to predict performance *a priori*, one approach would be to use traditional design of experiment tools (full-factorial, split-factorial, partial-factorial, etc.) or sampling plans (random, latin hypercube sampling, etc.) to select specimens to fabricate/test. You could then select "optimal" performance by selecting the highest performing design from those tested or by fitting a surrogate model to predict the response surface.

2. **Bayesian optimization:** More often, costs are incurred both for each specimen and each batch of testing, with costs dominated by the number of specimens tested. In this case, it may be desirable to use a Bayesian optimization approach to systematically and iteratively select designs for testing, to test these designs, and then to predict the next candidate for testing. In a Bayesian approach, you begin with some coarse initial sampling before fitting a Gaussian process surrogate model for the objective function and defining an acquisition function. After defining the acquisition function, you can optimize over this function to propose the next design candidate. A common strategy is to optimize for expected improvement, which takes into consideration both the mean performance and the uncertainty associated with the surrogate model. After evaluating this design candidate, the surrogate model is updated, the acquisition function is redefined, etc... This approach can be extremely powerful and produce good results with a relatively low number of specimens, but it does require the ability to have more than one fabrication/testing cycle.