Incrementally Computing the Hypervolume of a Set of m-D Points for $m = 3$ and 4

Mila Nedić

Faculty of Mathematics and Physics, University of Ljubljana
Faculty of Computer and Information Science, University of Ljubljana
Supervisor: Tea Tušar, Jožef Stefan Institute

May 21, 2024

1 Introduction

The hypervolume of a set of points in m dimensions is informally defined as the volume of the portion of the space which is determined by this set of points and bounded by a reference point [1]. In multi-objective optimization, it is often used to assess the quality of sets of solutions to multi-objective optimization problems [2]. As computing the hypervolume and hypervolume-based metrics in more than two dimension is computationally expensive, providing better algorithms has been of interest to researchers for many years [3].

The goal of this project is to transfer the currently most efficient algorithms for the hypervolume indicator in three and four dimensions (available in [4]), which are written in the programming language C, into Python. It is also in our interest to provide an extension to the Python `moarchiving` library (available in [5]), which contains the implementation of the hypervolume computation in two dimensions. This extension will enable the user to pick either the original two-dimensional or the new three- and four-dimensional version, depending on their needs.

2 Background

Given a set of points $X \subset \mathbb{R}^m$ and a reference point $r \in \mathbb{R}^m$, the *hypervolume* is defined as

$$H(X) = \lambda\Big(\bigcup_{x \in X} [x, r]\Big)$$

where $[x, r] = \{y \in \mathbb{R}^m : x_i \leq y_i \wedge y_i \leq r_i \forall i = 1, \ldots, m\}$ and $\lambda(\cdot)$ denotes the Lebesgue measure. The *hypervolume contribution* of a point $x \in \mathbb{R}^m$ to a set $X \subset \mathbb{R}^m$ is defined as

$$H(x, X) = H(X \cup \{x\}) - H(X \setminus \{x\}).$$

An example case for the hypervolume and the hypervolume contribution in two dimensions can be seen in Figure 1. The hypervolume, defined by the point set $X = \{x_1, x_2, x_3\}$ and the reference point r, is shown in the left picture. The hypervolume contribution of the point x_4 to X is shown in the right picture (colored with light gray).

3 Methods

The main idea is to extend the `moarchiving` Python library with the functionality to compute the hypervolume of three- and four-dimensional sets. This means implementing the following methods: `add`, `add_list`, `contributing_hypervolume`, `copy`, `distance_to_hypervolume_area`, `dominates`, `dominators`, `distance_to_pareto_front`, `hypervolume`, `hypervolume_improvement`, `in_domain`, `infos` and `remove`.

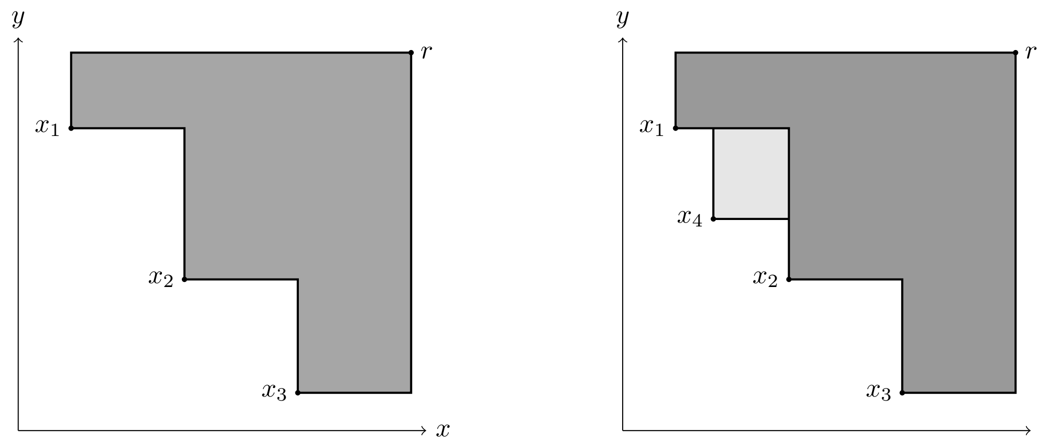

Figure 1: Example of the hypervolume indicator (left) and the hypervolume contribution (right) in two dimensions.

4 Implementation

This section gives overview of the implementation of the hypervolume in three and four dimensions. All of the auxiliary functions as well as the main functions for computing the hypervolume in three and four dimensions are written in the file `hv_plus.py`. Test cases for functions defined in `hv_plus.py` are available in `hv_plus_test.py`. A detailed overview of the obtained results is presented in Section 5.

4.1 Computation of the hypervolume in three dimensions

The hypervolume in three dimensions is computed by running the following functions: `setup_cdllist`, `preprocessing` and `hv3dplus`.

The `setup_cdllist` function takes `data` (a Python list representing the set of points in three or four dimensions, depending on the problem - in this case data is a list of three dimensional points), `n` (the number of points), `d` (the dimensionality of the problem, in this case `d = 3`) and `ref` (representing the reference point, which is also given as a Python list) as input arguments. It sets up a circular doubly-linked list of class `DLNode`, which is the class used for representing the point. Nodes are ordered in ascending lexicographic order of coordinates z, y and x. Each node has a pointer to the previous and next node in the list.

The `preprocessing` function then takes the output of the `setup_cdllist` and sets up pointers to the outer delimiters of each point, denoted by `closest[0]` and `closest[1]` in x and y coordinate, respectively. It is important to note that reprocessing doesn't produce an output, it only correctly sets up the closest nodes in x and y coordinates. Points whose projections onto the xy-plane are dominated, are deleted in the preprocessing step.

Finally, the `hv3dplus` function is called to compute the hypervolume in three dimensions. The input is the head node of the circular doubly-linked list. Firstly, `restart_list_y` is called on the head node, which resets the `cnext` pointers for the y-dimension. Secondly, the next node from the circular doubly-linked list is obtained by defining `p = p.next[2].next[2]`. While `p` is different from the last node in the circular doubly-linked list (assuming `p` is non-dominated), area is computed by calling `compute_area_simple`. If `p` is dominated by any point in the list, the `remove_from_p` function is called to update `next` and `prev` pointers of `p`. Lastly, the volume is accumulated by multiplying the area with the height of the current slice.

4.2 Computation of the hypervolume in four dimensions

The hypervolume in four dimensions is computed by firstly calling the `setup_cdllist` and then calling the `hv4dplusR` function. The later function takes head node of the circular doubly-linked list as an input argument and computes the hyperovlume in four dimensions by iteratively computing the hypervolume in three dimensions.

Algorithm 1 HV3D+: HYPERVOLUME COMPUTATION

Require: head (starting node of Q - a circular doubly-linked list sorted in ascending lexicographic order of coordinates z, y and x)
Ensure: $H(X)$ (hypervolume in 3-D of X)

```
p = restart_list_y(head)
p = p.next[2].next[2]
stop = head.prev[2]
while p != stop do
  if p.ndomr < 1 (p is non-dominated) then
    p.cnext[0] = p.closest[0]
    p.cnext[1] = p.closest[1]
    area += compute_area_simple(p.x, 1, p.cnext[0], p.cnext[0].cnext[1])
    p.cnext[0].cnext[1] = p
    p.cnext[1].cnext[0] = p
  else
    remove_from_z(p)
  end if
  volume += area * (p.next[2].x[2] - p.x[2])
  p = p.next[2]
end while
return volume
```

Algorithm 2 HV4D+: HYPERVOLUME COMPUTATION

Require: head (starting node of Q - a circular doubly-linked list sorted in ascending lexicographic order of coordinates w, z, y and x)
Ensure: $H(X)$ (hypervolume in 4-D of X)

```
p = p.next[3].next[3]
stop = head.prev[3]
while p != stop do
  restart_base_setup_z_and_closest(head, p)
  add_to_z(p)
  update_links(head, p, p.next[2])
  volume = hv3dplus(head)
  height = p.next[3].x[3] - p.x[3]
  hv += volume * height
  p = p.next[3]
end while
return hv
```

5 Results

In this section, various tests are presented in order to showcase the correctness and time-efficiency of our implemented algorithms.

5.1 Results for `hv3dplust`

Currently not working as intended.

5.2 Results for `hv4dplusR`

5.2.1 Correctness

To determine whether our implementation returns a result equal to the one computed with the original C code, multiple tests have been conducted. All of the test mentioned in subsection here are available in `hv4d_test.py`. To see the results, simply run the provided Python file.

The first example case for computing the hyperovlume in four dimensions constists of the following set of ten non-dominated points:

$$\texttt{data} = \begin{bmatrix} 1 & 2 & 3 & 1.0 \\ 4 & 5 & 6 & 0.5 \\ 7 & 8 & 9 & 0.7 \\ 2 & 1 & 0 & 0.6 \\ 3 & 4 & 5 & 0.8 \\ 6 & 7 & 8 & 0.3 \\ 9 & 1 & 2 & 0.9 \\ 5 & 6 & 7 & 0.2 \\ 8 & 9 & 1 & 0.4 \\ 0 & 1 & 2 & 0.1 \end{bmatrix}$$

with reference point $\texttt{ref} = [10, 10, 10, 10]$. The computed hypervolume is 8143.6 and equals to the output given by the original C code.

The second example consists of the following set of ten non-dominated points:

$$\texttt{data} = \begin{bmatrix} 1 & 10 & 20 & 30 \\ 2 & 9 & 25 & 29 \\ 3 & 8 & 30 & 28 \\ 4 & 7 & 35 & 27 \\ 5 & 6 & 40 & 26 \\ 6 & 5 & 18 & 35 \\ 7 & 4 & 22 & 34 \\ 8 & 3 & 28 & 35 \\ 9 & 2 & 16 & 40 \\ 10 & 1 & 15 & 45 \end{bmatrix} \tag{1}$$

with reference point $\texttt{ref} = [11, 11, 41, 46]$. The computed hypervolume is 15625 and, again, equals the output given by the original C code.

Another test has been performed to make sure that the results of the `hv4dplusR` function gets correctly multiplied by a constant factor if we scale all of the points from the original data by some constant. Points in the datset (1) are multiplied by factors 10, 0.01 and 0.01 and the computed hypervolume is 15620000, 1.15625 and 0.00015625, respectively.

The third example consists of the following set of ten non-dominated points:

$$\texttt{data} = \begin{bmatrix} 4 & 1 & 3 & 35 \\ 4 & 2 & 39 & 26 \\ 2 & 2 & 24 & 38 \\ 6 & 9 & 21 & 36 \\ 4 & 5 & 15 & 26 \\ 5 & 7 & 12 & 46 \\ 8 & 3 & 22 & 46 \\ 3 & 6 & 13 & 35 \\ 3 & 5 & 26 & 45 \\ 1 & 9 & 5 & 29 \end{bmatrix} \tag{2}$$

with reference point $\texttt{ref} = [11, 11, 41, 51]$. The computed hypervolume is 62133 and equals to the output given by the original C code.

Next, we want to see what happens when adding multiple dominated points to the dataset. firstly, we add the point $(10, 10, 40, 50)$, which is dominated by all other points, to the dataset (2) $(n = 11)$ while the reference point remains unchanged. The computed hypervolume also remains unchanged and equals to 62133 as before.

Secondly, we add the point $(9, 4, 23, 47)$ to the dataset (the total number of points in now $n = 13$), which is dominated by $(8, 3, 22, 46)$. As expected, the hyperovlume is again equal to 62133.

Thirdly, we add $(4, 7, 14, 36)$ to the dataset, which is dominated by $(3, 6, 13, 35)$. Again, the hypervolume is 62133.

We can conclude that, by iteratively adding dominated points to the list, the hypervolume remains constant.

5.2.2 Time-efficiency

An important factor when it comes to designing and programming algorithms is computational complexity, i.e. how the algorithms scale when increasing the size of the input.

To show that our implementation is time-efficient, examples of varying size in the number of points (denoted by n) are created, and for each size, ten calculations of the hypervolume in four dimensions are executed. For each of the ten calculations (for a fixed n), the executing time of computing the hypervolume with `hv4dplusR` is stored. Then, the average time for each value of n is calculated. Average times needed for computing the hypervolume in four dimensions in logartihmic scale can be seen in Figure 2. These results are very similar to the ones presented in [1].

Details of the implementation can be found in `hv4d_test_time.py`. Points are randomly generated within the $[0, 1]$ interval for each dimension and the reference point is chosen to be $[1, 1, 1, 1]$.

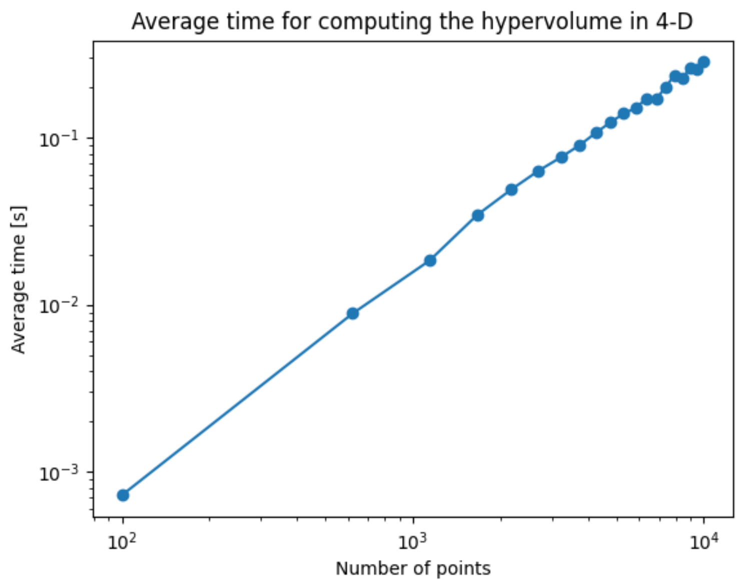

Figure 2: Average time for computing the hypervolume in four dimensions using an average of 10 calculations (both axes are in logarithmic scale).

6 Conclusion and further work

References

[1] A. P. Guerreiro and C. M. Fonseca, "Computing and updating hypervolume contributions in up to four dimensions," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 449–463, 2018.

[2] E. Zitzler, L. Thiele, M. Laumanns, C. Fonseca, and V. da Fonseca, "Performance assessment of multiobjective optimizers: an analysis and review," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 2, pp. 117–132, 2003.

[3] A. P. Guerreiro, C. M. Fonseca, and L. Paquete, "The hypervolume indicator: Computational problems and algorithms," *ACM Comput. Surv.*, vol. 54, jul 2021.

[4] A. P. Guerreiro, "Computing and updating hypervolume contributions in up to four dimensions." https://github.com/apguerreiro/HVC, 2016. Accessed: 2024-04-04.

[5] N. Hansen, "A bi-objective nondominated archive class." https://github.com/CMA-ES/moarchiving, 2018. Accessed: 2024-04-04.