Jarvis march (Gift Wrapping) Convex Hull algorithm

Author: Caio Valente (https://github.com/valentecaio/geometry)

Date: 2023-10-05

Introduction

A convex hull is a fundamental concept in computational geometry that represents the smallest convex shape enclosing a set of points in a plane. This convex shape, often referred to as the "hull," encompasses the outermost boundary of the point set, forming a closed, convex polygon. The study of convex hull algorithms revolves around developing efficient methods to calculate this polygon, which plays a crucial role in a wide range of applications, from geographic information systems to robotics and computer graphics.

The objective of this project is to study, implement and analyse one of the most known "Convex Hull" algorithms, called Jarvis march, or Gift Wrapping. We will implement it in the Lua language.

Jarvis March (Gift Wrapping) algorithm

The Jarvis March algorithm, also known as the Gift Wrapping algorithm, is a straightforward and intuitive method for finding the convex hull of a set of points in the plane. It was introduced by R. A. Jarvis in 1973 and is based on the idea of selecting points on the convex hull by imagining them as the vertices of a gift wrapping around the points.

This algorithm has a time complexity of O(nh), where n is the number of input points, and h is the number of points on the convex hull. In the worst case, where all points are on the convex hull, the algorithm becomes O(n^2). However, in practice, the number of points on the convex hull (h) is often much smaller than n, making the algorithm relatively efficient.

It starts by selecting an initial point as a reference. This point must be a point that is known to be on the convex hull. The point with the lowest y-coordinate is a good candidate, because it is known to be on the convex hull in any set of points. If there are multiple points with the same lowest y-coordinate, the one with the lowest x-coordinate is chosen. I refer to this point as the pivot.

The algorithm then sorts the points based on the polar angle from the pivot, which is the angle between the x-axis and the line from the pivot to the point. We simplify this by using the cross product, which is positive if the point is to the left of the pivot, negative if it is to the right, and zero if it is collinear.

The points are then iterated over in sorted order. For each point, we check if the last two points in the convex hull and the current point form a counter-clockwise turn. If they do, the current point is added to the convex hull. If they do not, the last point in the convex hull is removed and the check is repeated until the current point can be added to the convex hull.

The algorithm terminates when the last point in the convex hull is the pivot. Since the pivot is also the first point on the hull, the lines now form a closed polygon and the hull is complete.

Dependencies

We will use the Python library MatPlotLib for plotting results and a Lua script (matplotlua) for parsing Lua data to matplotlib. As a consequence, this project needs python and matplotlib to run. We also need the lua-cjson Lua package for parsing json data. Our main.lua script uses a Lua library called argparse to parse arguments from the command line.

The script includes an option to generate gif animations of the step-by-step execution of algorithms. This option requires the ImageMagick tool to work and may only work on Linux environments.

```
$ pip install matplotlib               # install python deps
$ sudo apt install lua-cjson           # install lua deps
$ sudo luarocks install argparse
$ sudo apt install imagemagick-6.q16   # optional, for step-by-step animations
```

Project Structure

```
.
├── convex_hull/
│   ├── convex_hull.lua   -- Contains the developed Convex Hull algorithms.
│   ├── datasets/         -- Contains two predefined input datasets.
│   ├── figures/          -- Output directory for plot PNG figures and GIF animations.
│   │                        It also contains some example animations.
│   └── main.lua          -- Script to test the developed algorithms, compare them and
│                            analyse their results.
├── matplotlua/           -- Our plot library.
│   ├── matplotlua.lua    -- Lua module to handle plot data and parse it to JSON.
│   └── matplotlua.py     -- Python script that parses JSON data and plots it.
│
└── utils/
   └── utils.lua          -- Contains useful methods reused by other modules.
```

How to execute tests

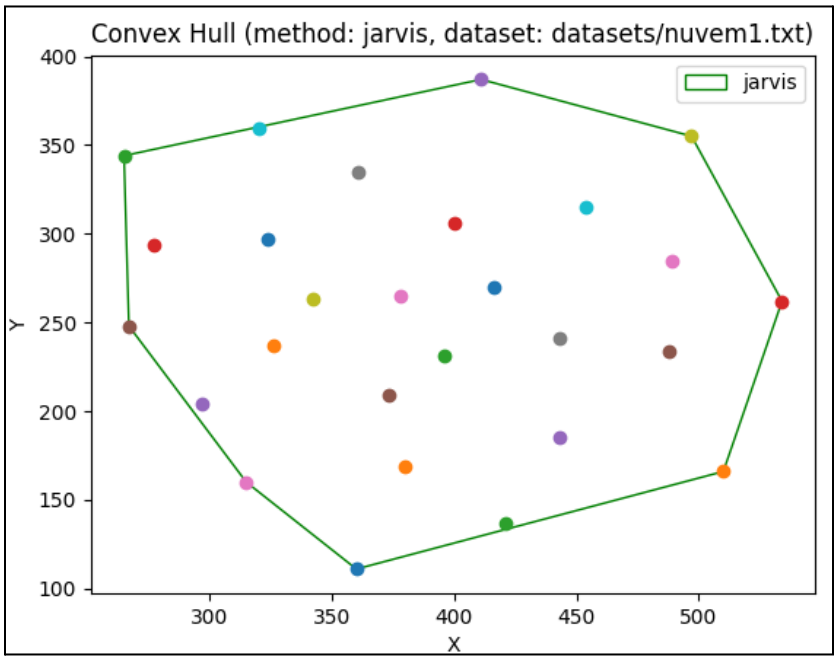
We just need to execute the convex_hull/main.lua file using the Lua interpreter:

```
# lua main.lua -h
Usage: main.lua [-h] [-a <alg>] [-i <input>] [-n <npoints>]
       [-d <delay>] [--dir <dir>] <command>

Commands:
  algorithm    Generate and plot the convex hull for a given algorithm.
  compare      Visually compare the results of all algorithms runnig with the same dataset.
  complexity   Plot the execution time of the algorithms for different input sizes.
  hull-size    Plot the size of the convex hull for different input sizes.
  gif          Generate step-by-step gif image for a given algorithm.

Arguments:
   command               One of the commands above.

Options:
   -h, --help            Show this help message and exit.
      -a <alg>,          Algorithm name, one of [ jarvis | skala ]
   --alg <alg>
        -i <input>,      Data Set input file path. Omit this option to use a random dataset.
   --input <input>
          -n <npoints>,  Number of points in the random dataset. (default: 100)
   --npoints <npoints>
        -d <delay>,      Delay of gif image transition in ms. (default: 50)
   --delay <delay>
   --dir <dir>           Output directory for figures. (default: figures/)
```

To study the Gift Wrapping algorithm, we will pass the argument -a jarvis to the commands algorithm, complexity and gif.

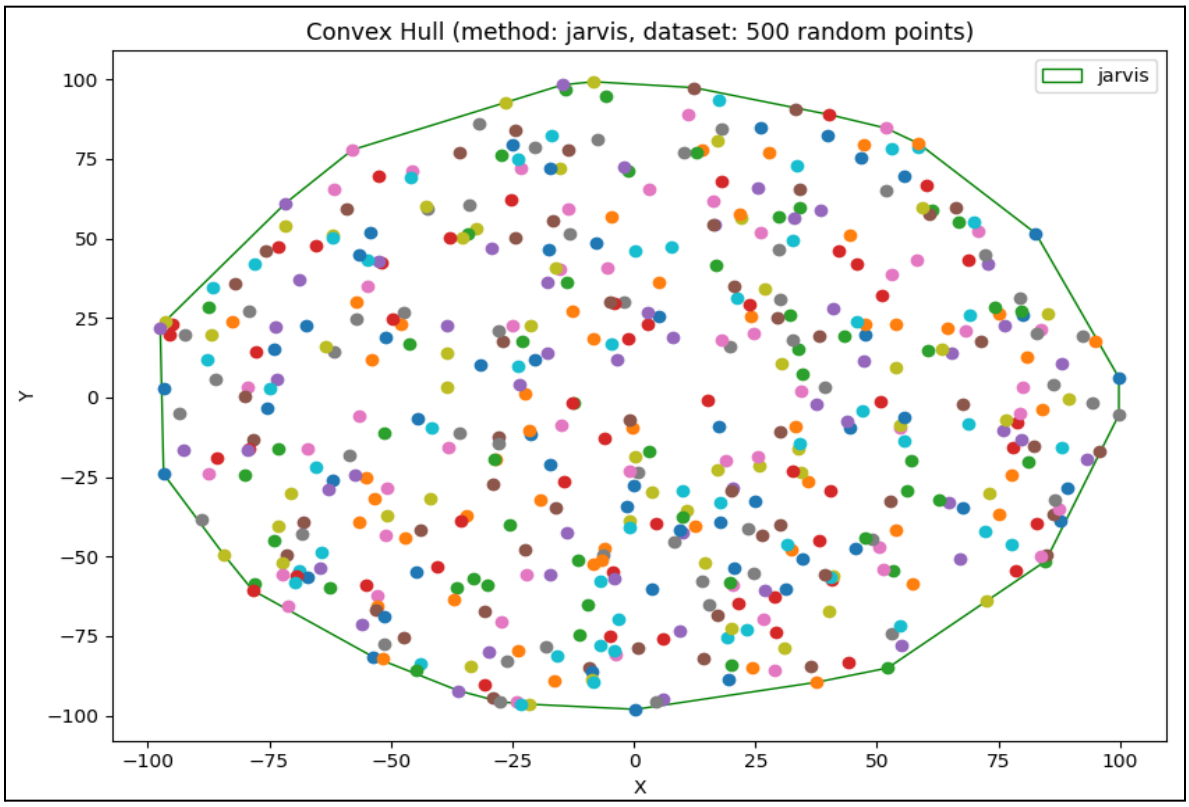
Results

1) Small predefined dataset



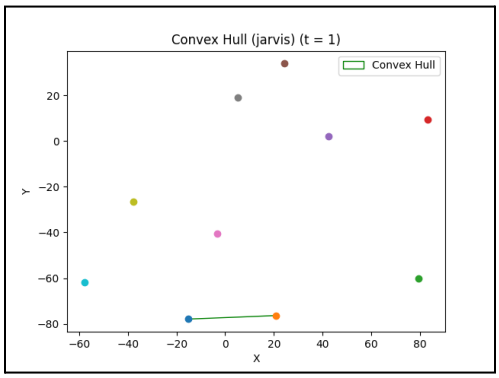
lua main.lua algorithm -a jarvis -i datasets/nuvem1.txt

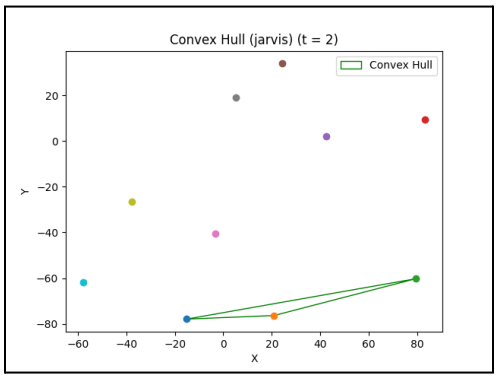
2) Big randomic dataset

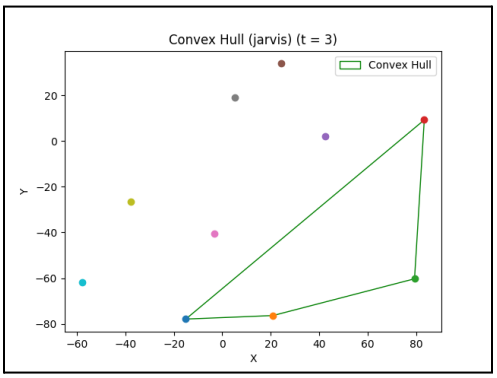


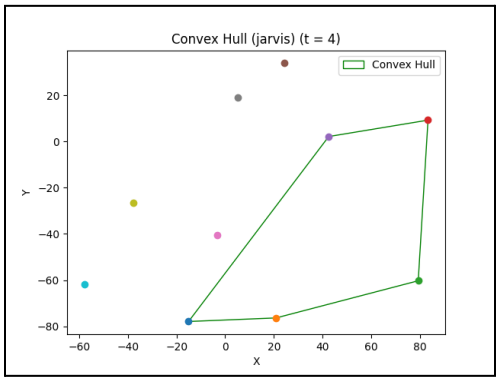
lua main.lua algorithm -a jarvis -n 500

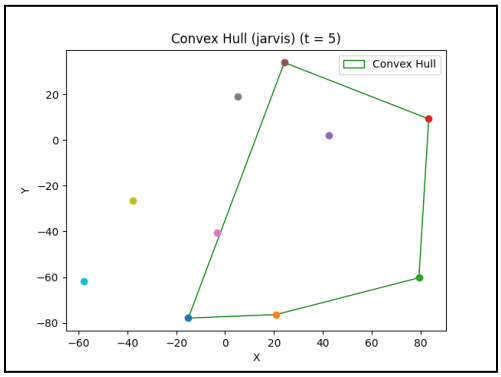
3) Step by step

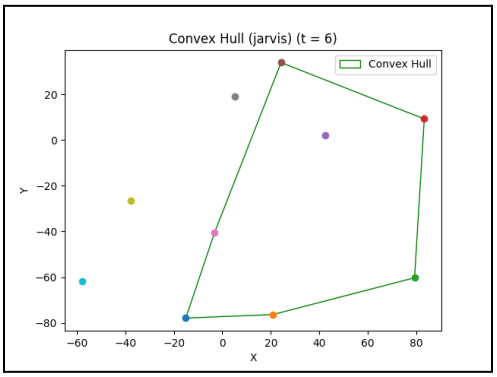


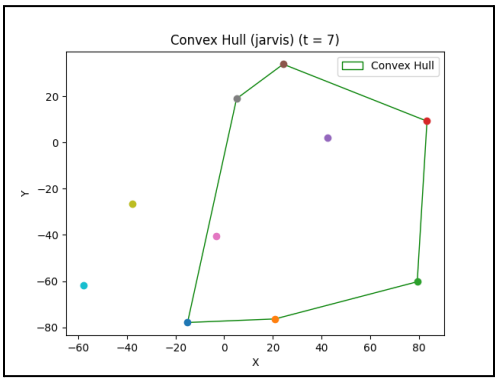


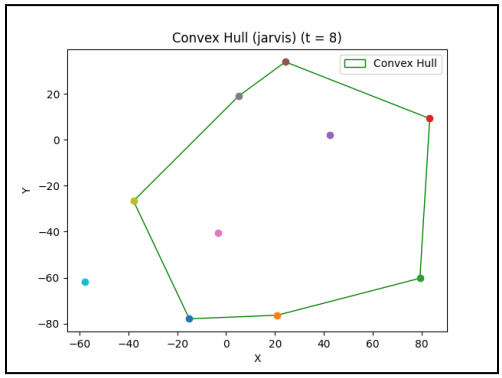


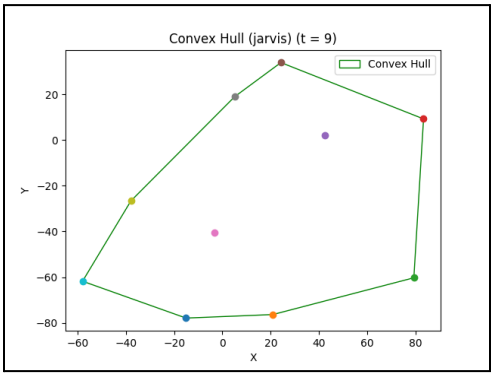


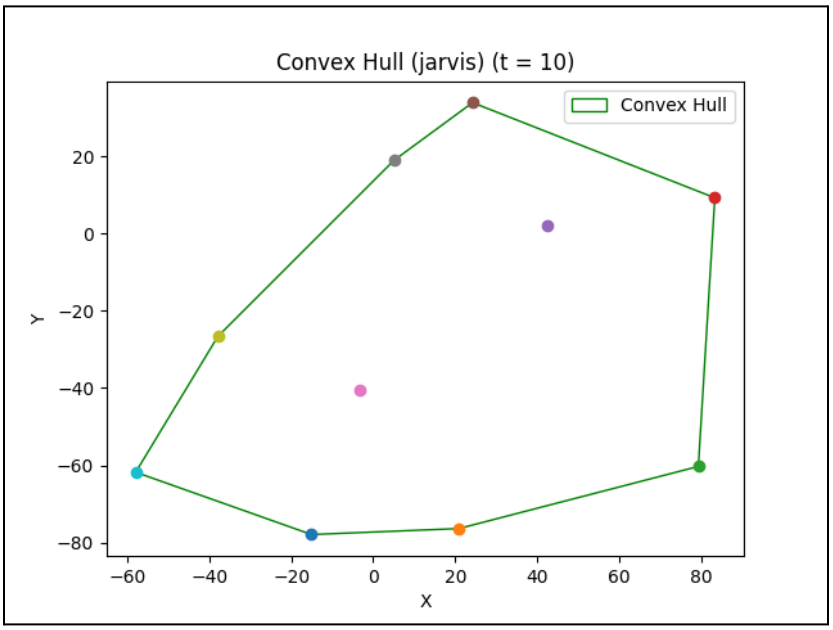












This small example illustrates how the gif wrapping operates. In the first image, we can see that the chosen pivot is the y-lowest point (dark blue), and the hull is initiated with the pivot and the first point in the sorted set (orange). It then proceeds to the next point (green), which is on the hull (orientation < 0). The same thing happens with the next point (red). However, in image 4, the algorithm tries to add the purple point to the hull. Since the next point (brown) would have an orientation > 0, it would be outside the hull. As a consequence, the purple point is removed and the brown point is added to the hull (image 5).

This example can be found in the form of a Gif animation in the convex_hull/figures directory. This directory also contains other animations. They were generated with the commands below:

```
$ lua main.lua gif -a jarvis -i datasets/nuvem1.txt
$ lua main.lua gif -a jarvis -i datasets/nuvem2.txt
$ lua main.lua gif -a jarvis -n 100 -d 20
$ lua main.lua gif -a jarvis -n 10

$ ls figures/example-jarvis-*
figures/example-jarvis-dataset1.gif
figures/example-jarvis-dataset2.gif
figures/example-jarvis-random-100-points.gif
figures/example-jarvis-random-10-points.gif
```

Time complexity analysis

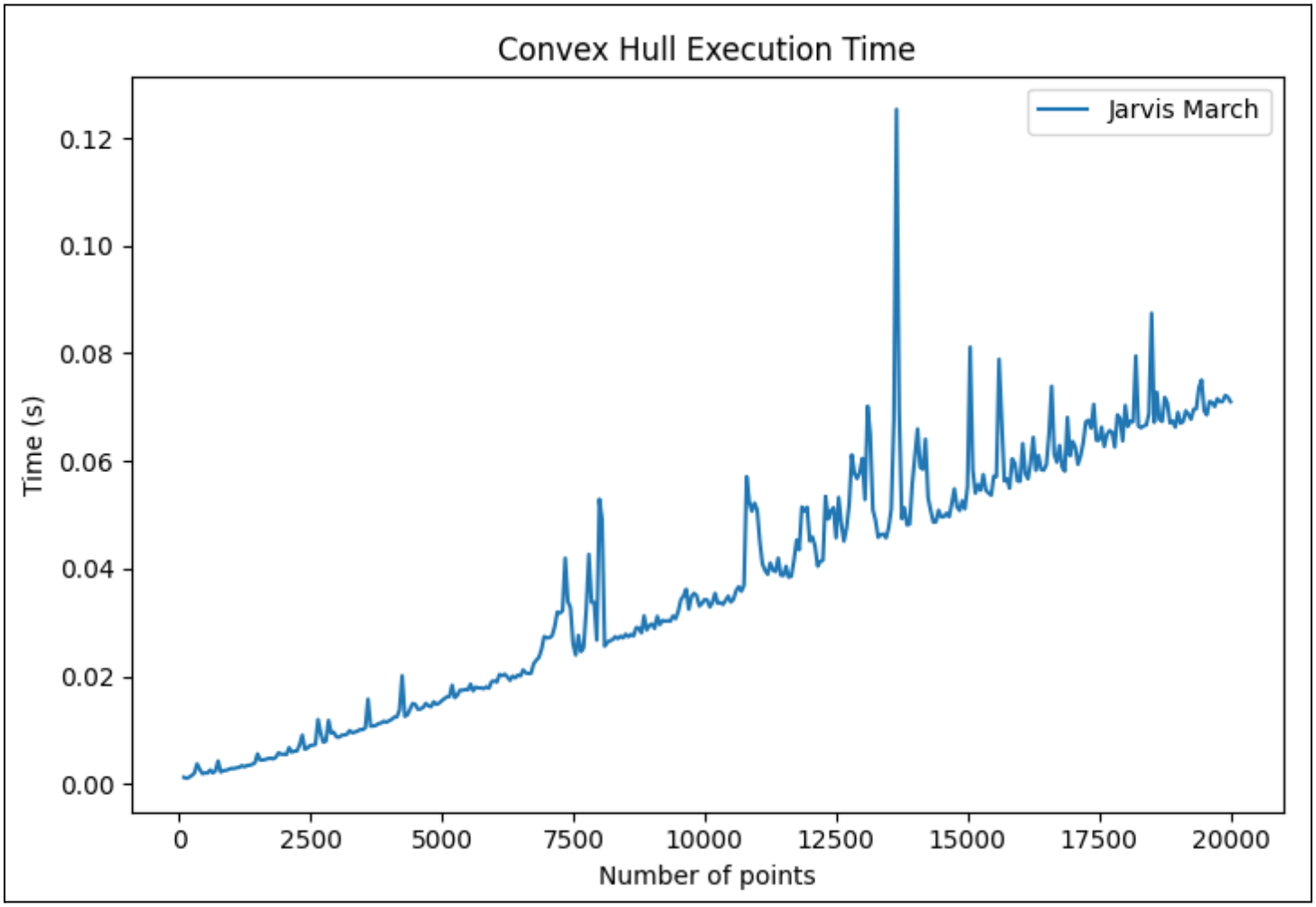
To analyse the time complexity of this algorithm, we will measure its execution time for different dataset sizes.

These parameters can be changed in the main.lua script, at line 11:

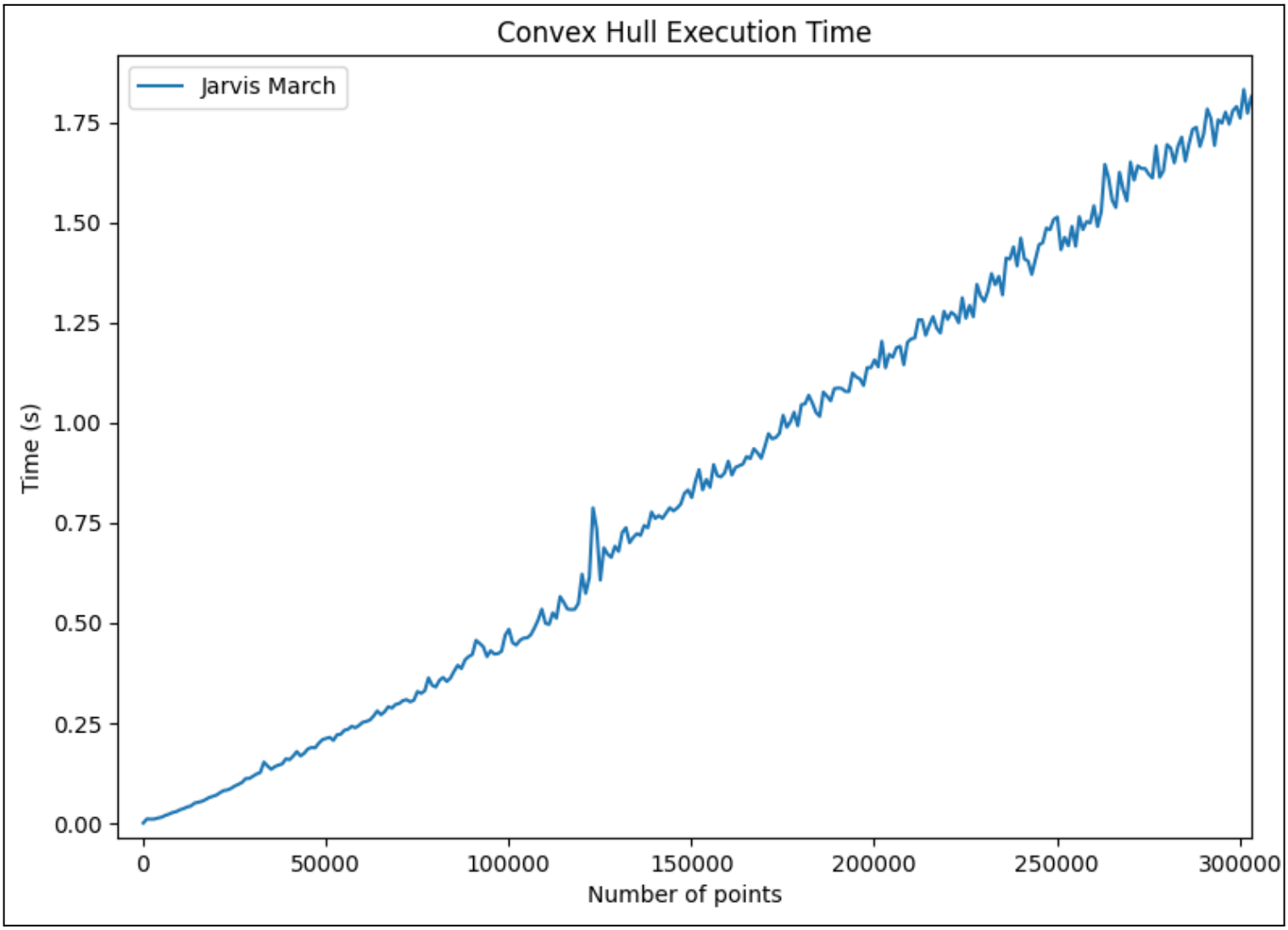
```
local min_input_size = 10
local max_input_size = 50000
local step = 50
```

Then, run the script using the "complexity" command:

```
$ lua main.lua complexity -a jarvis
```



N = 10 to 20k, step of 50

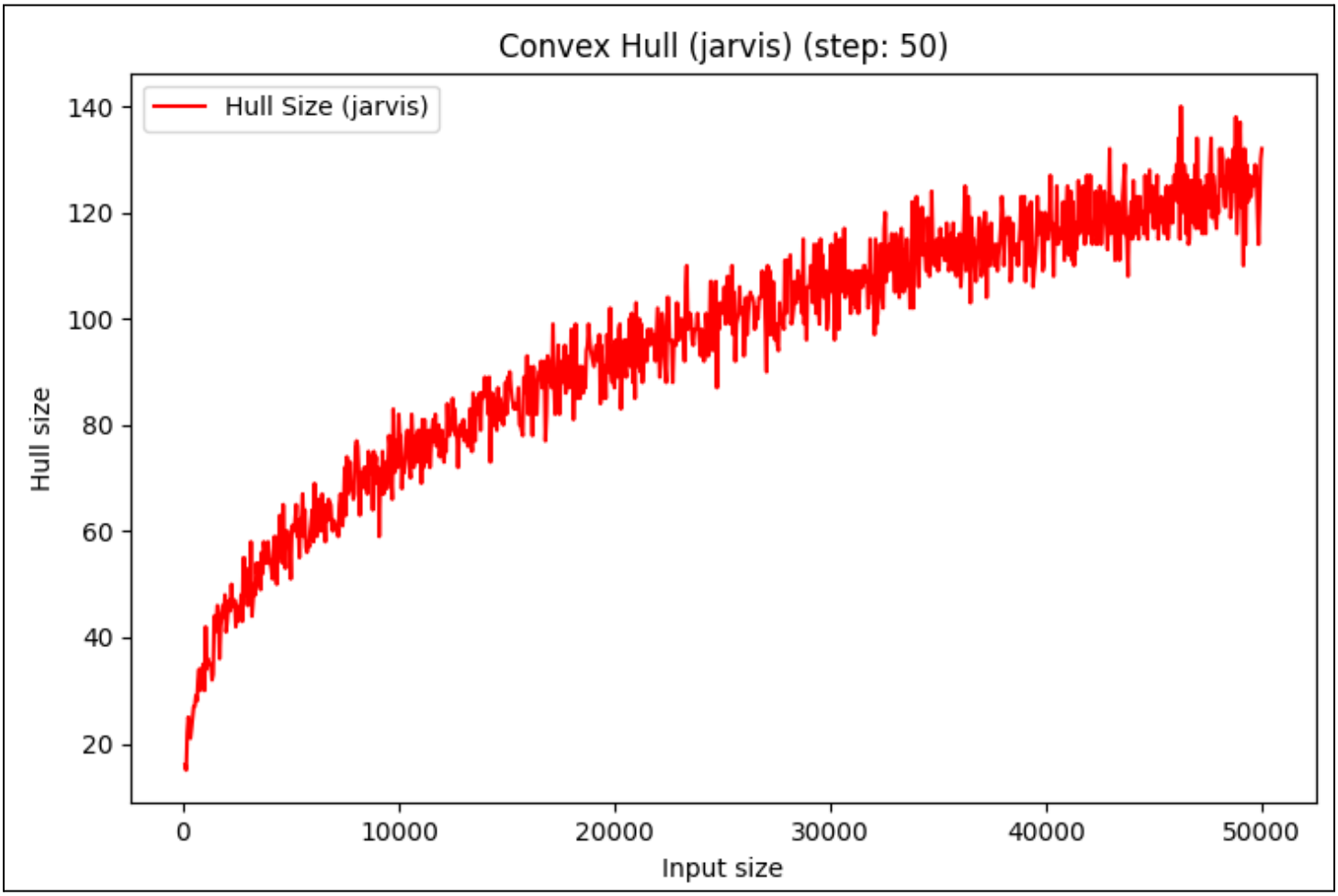


N = 10 to 300k, step of 1k

The Gift Wrapping algorithm has a complexity of O(nh), where n is the size of the input and h is the size of the output (the hull). Its worst case is $O(n^2)$, when all points are on the hull, but this condition is very unlikely to happen. In practical scenarios, the number of points on the hull is much smaller than the size of the input ($h << n$), and the algorithm performs closely to its lower bounds $O(n * \log(n))$, limited by the angle sorting operation. We can confirm this with the plotted curve.

The curve is also very noisy, due to the probabilistic generation of our input. Since we randomly generate our inputs, some sets may contain more points on their hulls than others.

Hull size growth



N = 10 to 50k, step of 50

Plotting the hull size (k) for different input sizes, we can observe that it grows proportionally to log(n), as expected.

```
$ lua main.lua hull-size -a jarvis
```