ConvexHulls

June 3, 2016

```
In [1]: using BenchmarkTools
        using CHull2D
        using DataFrames
        using FixedSizeArrays
        using PlotlyJS
```

This notebook is the beginning of a sequence of notebooks that discuss numerical convex hull algorithms. We will focus on computing convex hulls for 2 dimensions. These are intended to give an idea of what is involved in finding the convex hull. If needed one can find other more technically complete notes (some of which are mentioned in the references) that will be helpful in dealing with more general cases. All related code to these operations can be found on github under the `CHull2D.jl` package.

1 Convex Hulls

In this notebook we present the convex hulls and discuss briefly why they matter. Additionally, we will compare the performance of two (eventually we will add a third) algorithms that are discussed in other notebooks of this series.

1.1 What is a convex set

Mathematically defined, a set X is **convex** if the mixture of any two points of the set is also in the set.

More succinctly, X is convex if $\forall \lambda \in (0, 1)$ and $\forall\, x, y \in X$ then $z \equiv \lambda x + (1-\lambda) y \in X$.

Less formally, we can think of this as the straight line drawn between two points as being completely inside the set X.

The extreme points of a convex set is the minimal set of points that can be used to generate the entire set. Mathematically, we would say that an extreme point of a convex set, S, is a point in $s \in S$ which does not lie in any open line segment joining two points of S. For shapes with finite edges, we can think about these being the vertices.

For example, the extreme points of a square in $\mathcal{R}^2$ would simply be its 4 corners. On the other hand, the extreme points of a circle are its entire circumference.

1.2 Why do we care

Convex sets are front and center for many important results within many fields – This includes our field of economics. Representing a convex set on paper –or in your mind– is simple, but representing such a set on a computer requires a little more thought. In this notebook, we will show you one way a convex set can be represented on a computer and how to generate the convex hull of a given set of points.

More generally, representing a set of points in a computer is very difficult because in order to keep track of the points the computer needs to represent every point from the set. This becomes impossible as soon as the set of points isn't discrete (we can't keep track of an infinite number of points in a computer). Luckily, convex sets can be represented using only their *extreme points* – This greatly reduces the amount of information that we need to keep.

In later notebooks, we will use the concept of a convex hull to represent sets of continuation values for dynamic games.

1.3 Numerical Convex Hull Algorithms

Numerical algorithms to generate convex hulls complete the following exercise: Given a set of points, tell me which points are necessary to represent the convex hull of that set. This exercise is simple to do on paper, you draw the least distance line possible that circles the set of points – i.e. you draw lines around the outside of the set to connect the extreme points. This task becomes slightly harder once we move to a computer because we must find a practical way to do this generally.

In this sequence of notebooks, we will cover three algorithms that are used to numerically compute the convex hull:

- The Graham Scan
- Monotone Chain
- Quickhull (This will be implemented at a later date)

At the end of this notebook we will run some simple comparisons of the three algorithms. Before doing this, we will introduce some prerequisite material that will be useful for developing these algorithms.

1.4 Useful Prerequisite Material

Here we introduce the previously mentioned prerequisite material that will simplify the exposition of the algorithms later.

In our exposition below, we will consider a set of points P. Let $a = (a_1, a_2)$, $b = (b_1, b_2)$, and $c = (c_1, c_2)$ be three points in P. We will think of these points as being ordered by (a, b, c). We will think about two vectors created by these points: $u = \vec{ab}$ and $v = \vec{ac}$. Finally, we will consider the triangle formed by the three points $\triangle abc$.

NOTE: *Some of the following material will be repetitive for some readers, feel free to skip as much of this section as you feel comfortable skipping.*

1.4.1 Angles between Points

At the base of much of what we do is determine the angle between three points. We will use this to sort points by their angle relative to some initial point and also to determine whether three points make a clockwise or counter-clockwise turn.

Consider our three points (a, b, c) and the two vectors formed by them $\vec{u}$ and $\vec{v}$. One way we could find the angle formed by the three points is to find the angle, θ, formed between our two vectors u and v. This can be computed by

$$\theta = \arccos\left(\frac{u \cdot v}{\|u\|\|v\|}\right)$$

This would tell us the exact angle. It also tells us clockwise or counter-clockwise by the angle being greater or less than 180 degrees.

While this approach would certainly work, it requires more operations than we need. For most of what we need to do, it will suffice to determnine whether the three points generate a clockwise or counter-clockwise turn. This can be done by using a cross product of the vectors u and v.

Wikipedia tells us,

> In computational geometry of the plane, the cross product is used to determine the sign of the acute angle defined by three points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ and $p_3 = (x_3, y_3)$. It corresponds to the direction of the cross product of the two coplanar vectors defined by the pairs of points p_1, p_2 and p_1, p_3, i.e., by the sign of the expression $P = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$. In the "right-handed" coordinate system, if the result is 0, the points are collinear; if it is positive, the three points constitute a positive angle of rotation around p_1 from p_2 to p_3, otherwise a negative angle. From another point of view, the sign of P tells whether p_3 lies to the left or to the right of line p_1, p_2.

We can use the formula they give us to compute the last (and relevant) element of the cross-product.

$$P = (b_1 - a_1)(c_2 - a_2) - (b_2 - a_2)(c_1 - a_1)$$

If P is positive then we moved counter-clockwise and if P is negative then we moved clockwise.

1.4.2 Point Relative to Triangle

Another important piece of what we do will be to determine whether a point lies inside of or outside of a triangle. We will do this by using Barycentric Coordinates.

Consider a point x and our triangle $\triangle abc$. We want to know whether $x \in \triangle abc$. The key insight will be that triangles are convex sets and so we can represent any point within the triangle as a convex combination of the three points that make up the triangle – In our case a, b, and c.

Thus $x \in \triangle abc$ if and only if $\exists\ \lambda_1, \lambda_2, \lambda_3$ where $\lambda_i \in [0, 1]$ and $\sum_i \lambda_i = 1$ such that $x = \lambda_1 a + \lambda_2 b + \lambda_3 c$.

If we use $\lambda_3 = 1 - \lambda_1 - \lambda_2$ and rearrange this equation then we get:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_1 - c_1 & b_1 - c_1 \\ a_2 - c_2 & b_2 - c_2 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} + \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

Define $T = \begin{bmatrix} a_1 - c_1 & b_1 - c_1 \\ a_2 - c_2 & b_2 - c_2 \end{bmatrix}$, then we can rearrange this again to get an expression for λ_1 and λ_2:

$$\begin{bmatrix} \lambda_1 & \lambda_2 \end{bmatrix} = T^{-1} \begin{bmatrix} x_1 - c_1 \\ x_2 - c_2 \end{bmatrix}$$

This equation pins down λ_1 and λ_2 and once we have these then we are able to determine whether the conditions on λ_i hold. If

- $\lambda_1 \in [0, 1]$
- $\lambda_2 \in [0, 1]$
- $\lambda_1 + \lambda_2 \leq 1$

then we have found the $(\lambda_1, \lambda_2, \lambda_3)$ that we need for our conditions to hold and we know $x \in \triangle abc$.

1.4.3 Wrapping a Sequence of Points

In both the monotone chain algorithm and the Graham scan, we will be required to take a sequence of sorted points and wrap our convex hull around them in the right way. We can do this by assuring that the direction of the turn between any three points in the convex hull is the same direction (worth drawing a picture to convince yourself of this fact) – We will use the counter-clockwise direction.

- Start the convex hull with the first two elements of the sequence of points.
- For each element $p_i \in P$ take the convex hull C and the number of points currently in C, $m = |C|$, as given.
- Check whether (C_{m-1}, C_m, p_i) makes up a clockwise or counter-clockwise turn.
- If this term is clockwise then C_m cannot be a memeber of the convex hull, so we discard it from C.
  - If after removing C_m from the convex hull there is only one element of the convex hull then add p_i to the convex hull and move to the next point, p_{i+1}
  - If there are more than one elements left in the convex hull then check the next element of the convex hull as well – Check whether (C_{m-2}, C_{m-1}, p_i) makes up a clockwise turn and repeat until there is only one element in the convex hull, or the last two points of the convex hull and p_i make a counter-clockwise turn.
- If the turn is counter-clockwise then add p_i to the convex hull and move to p_{i+1}.

As long as the points are ordered as described in the Graham scan algorithm or the monotone chain algorithm, this will determine the extreme points of the set.

1.5 Pruning: Akl–Toussaint Heuristic

Since it is applicable to the algorithms we discuss in the other notebooks, we present the *Akl-Toussaint Heuristic*. This tool will allow us to reduce the number of points that we need to consider when computing the convex hull by removing points that we know are not elements of the convex hull. This is important because the algorithms that we present are all $O(n \log n)$ complexity meaning that the number of points that we are passing to our actual algorithms will have a significant effect on compute time.

This was introduced in work done by Selim Akl and G. T. Toussaint in their 1978 paper, *"A fast convex hull algorithm."* Their key insight is that if you draw a quadrilateral using four points that you know are elements of the convex hull, then you are able to immediately disregard a large number of points that you know are not in the convex hull. The four points they propose are the maximal and minimal x and y coordinates.

1.6 Algorithm Comparisons

These comparisons were run by drawing np points from a standard normal distribution and then computing the convex hull nr times to make sure each algorithm had a chance to perform well. There are obvious short comings of this comparison because it only is comparing them for a specific type of set – a set with many points but few extreme points. More careful analysis has been done by others, though I may eventually edit this notebook to include analysis on sets with different properties. The code to do this benchmark can be found in the file `benchmark.jl`.

The first thing to notice it that the algorithms scale just slower than linearly – This is because of the complexity $O(n \log n)$ which is slightly more than linear.

One thing that I found surprising was that the Akl-Toussaint heuristic did not speed things up until the number of points were very large. It was more powerful when applied to the Graham scan because the Graham scan is slightly slower than the monotone chain algorithm for large numbers of points.

TODO: Try running other experiments like taking sets in which every point is an extreme point etc...

Algorithm (ms)	25 Points	250 Points	2500 Points	25000 Points	250000 Points	2500000 Points
GrahamScan (AT)	0.005161	0.014266	0.068315	0.633169	6.341886	61.155272
GrahamScan (NAT)	2.028281	1.942436	1.711961	7.020062	94.541751	1138.330192
MonotoneChain (AT)	0.576907	0.583223	0.518355	0.629892	5.062277	53.343085
MonotoneChain (NAT)	0.456337	0.473757	0.625874	2.52919	29.210234	300.650113

1.7 References

- http://geomalgorithms.com/a10-_hull-1.html
- https://github.com/intdxdt/convexhull.jl
- Wikipedia – Specific articles are cited within text
- http://keithbriggs.info/documents/Skarakis_MSc.pdf
- http://www.toptal.com/python/computational-geometry-in-python-from-theory-to-implementation

GrahamScan

June 3, 2016

```
In [1]: using CHull2D
        using FixedSizeArrays
        using PyPlot
```

```
INFO: Recompiling stale cache file /home/chase/.julia/lib/v0.4/PyPlot.ji for module PyPl
INFO: Recompiling stale cache file /home/chase/.julia/lib/v0.4/PyCall.ji for module PyCa
INFO: Recompiling stale cache file /home/chase/.julia/lib/v0.4/LaTeXStrings.ji for modul
```

1 Graham Scan Algorithm

This notebook is a piece of a sequence of notebooks that discuss numerical convex hull algorithms. The goal of these algorithms is to take a set of points P and output a set of extreme points EP such that every point in P is contained in co(EP).

In this notebook we present the Graham Scan algorithm.

1.1 Introduction

This algorithm was initially introduced by Ronald Graham in 1972 in "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set." The algorithm will have $O(n \log(n))$ complexity mostly due to the required sorting done within the algorithm. As you will observe, the way in which the algorithm traverses the points will lead for it to be a better fit for sets of points which have many extreme points.

1.2 The Algorithm

The algorithm has two main parts:

- First, all points must be sorted relative to some initial point which is known to be an extreme point.
- Second, move along the points and make sure that the "turns" between points are all the same direction.

1.2.1 Part 1: The Sort

In our implementation, we will use the point with the minimum y value as the initial point. Additionally, our rotation direction will be counter-clockwise.

Call our initial point a. What we really need to know is whether for any two points $b, c \in P$ whether the angle formed by $\vec{ab}$ and $\vec{bc}$ is greater than or less than 180 degrees. This can be determined by using the cross-product. If the cross-product is negative then we have moved clockwise; if the cross-product is positive then we have moved counter-clockwise (as desired). This leaves us to simply sort by comparing the appropriate cross-product between vectors – It turns out we can directly write this as:

$$\vec{ab} \times \vec{bc} = (b_1 - a_1)(c_2 - a_2) - (b_2 - a_2)(c_1 - a_1)$$

We wrote a function to do this in the ConvexHulls notebook.

1.2.2 Part 2: Finding Points

As previously mentioned, we will begin with the point which has the smallest y value. We will then work through our ordered points three at a time.

Imagine we have three points p_{i-1}, p_i, p_{i+1}.

- If these points constitute a counter-clockwise turn as desired then we proceed to compare points p_i, p_{i+1}, p_{i+2}.
- If these points constitute a clockwise turn then we know that p_i is not an element of the extreme points and can discard it. We would then "take a step backwards" and compare $p_{i-2}, p_{i-1}, p_{i+1}$

We follow until we reach the final point (which is also our starting point).

This is the `wrappoints` function from the ConvexHulls notebook.

1.2.3 Summary of Algorithm

To summarize the algorithm we could write:

1. Begin with a point that we know is a member of the extreme points – for example, the point with the minimum y value. Call this point p_0.
2. Sort all points by their polar angle with respect to p_0
3. Given p_{i-1}, p_i, p_{i+1}:
    - If points move clockwise, then p_i is not an element of the extreme points and we remove it from the set of points. We then consider $p_{i-1}, p_{i+1}, p_{i+2}$.
    - If points move counter-clockwise, then move forward to next step by consider p_i, p_{i+1}, p_{i+2}.
4. Continue until reach the point p_0 again.

1.3 Implementation

Below we implement the algorithm.

```
In [2]: function _grahamscan{T<:Real}(p::Vector{Point{2, T}})

            # Get yminind
            xminind = indmin(p)
            xmin = p[xminind]

            # Create function to sort by angles
            lt(a, b) = xmin == a ? true : xmin == b ? false : ccw(xmin, a, b)
            psort = sort!(p, lt=lt)
            # Add the starting point at end so we go full circle
            push!(psort, xmin)

            # Call function that works around points
            ep = wrappoints(psort)

            return ep[1:end-1]
        end
```

```
Out[2]: _grahamscan (generic function with 1 method)
```

1.4 Example Use

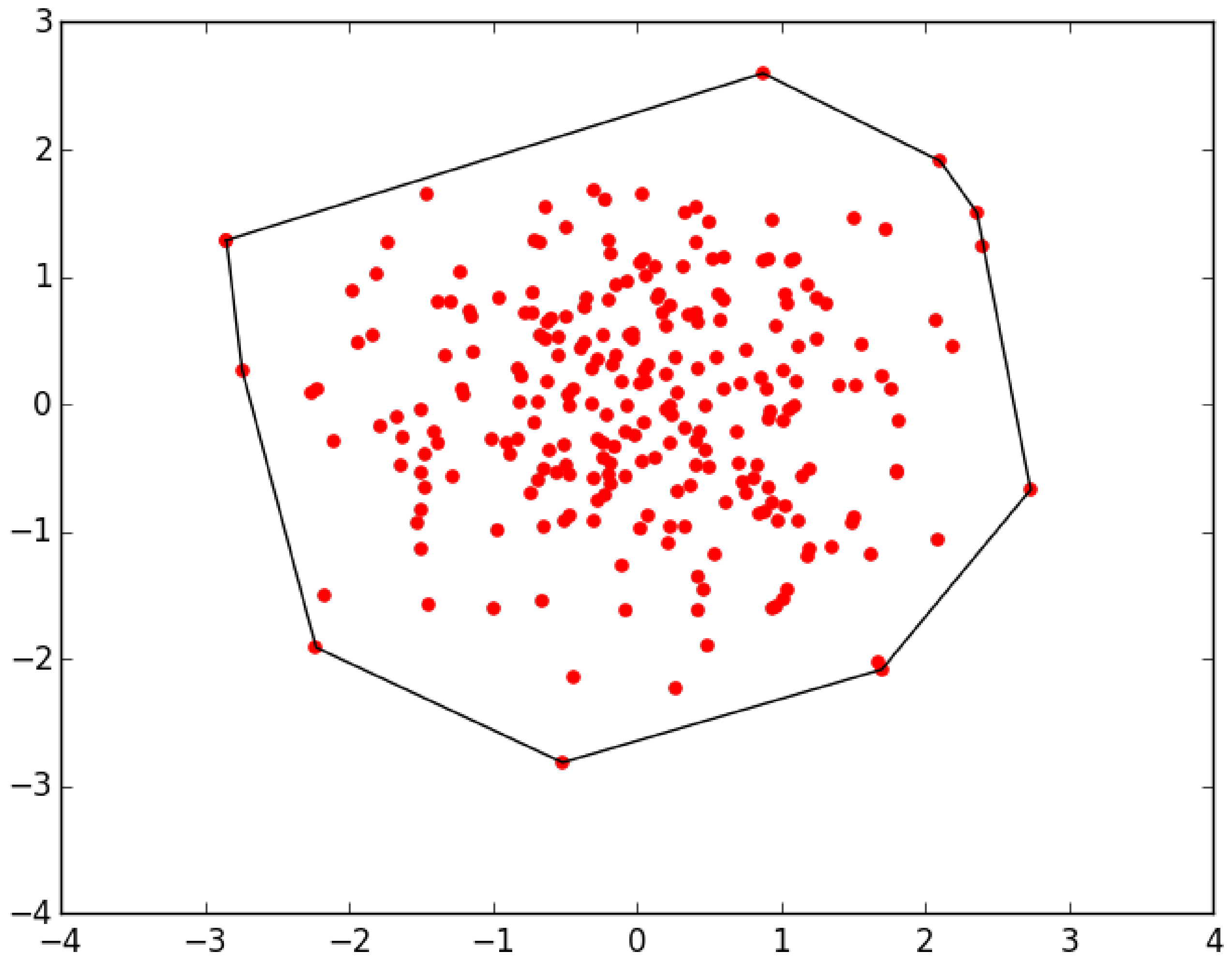
We can now use the Graham scan algorithm to get the Convex Hull and plot it.

```
In [3]: srand(1112016)
        p = [Point(randn(), randn()) for i=1:250]
        ep = _grahamscan(p)

        # Add the first point to end to make plot pretty
        push!(ep, ep[1]);
```

```
In [7]: fig, ax = subplots()

        ax[:scatter]([el[1] for el in p], [el[2] for el in p], color="r")
        ax[:plot]([el[1] for el in ep], [el[2] for el in ep], color="k")
```



```
Out[7]: 1-element Array{Any,1}:
         PyObject <matplotlib.lines.Line2D object at 0x7f55a1af3e10>
```

1.5 References

1.5.1 Academic Papers

- Graham, R.L. (1972). An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. Information Processing Letters 1, 132-133
- Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", Communications of the ACM 7 (6): 347–348

1.5.2 Useful Wikipedia Pages

- https://en.wikipedia.org/wiki/Graham_scan
- https://en.wikipedia.org/wiki/Heapsort
- https://en.wikipedia.org/wiki/Cross_product

MonotoneChain

June 3, 2016

```
In [1]: using CHull2D
        using FixedSizeArrays
        using PyPlot
```

1 Monotone Chain Algorithm

This notebook is a piece of a sequence of notebooks that discuss numerical convex hull algorithms. The goal of these algorithms is to take a set of points P and output a set of extreme points EP such that every point in P is contained in $\text{co}(EP)$.

In this notebook we present the Monotone Chain algorithm – also known as Andrew's algorithm.

1.1 Introduction

This algorithm was first introduced by A.M. Andrew in 1979 in the paper "*Another Efficient Algorithm for Convex Hulls in Two Dimensions*." This algorithm also has $O(n \log n)$ complexity which again is mostly due to sorting. The algorithm will compute the convex hull by computing two pieces: an "upper" and "lower" hull. The upper hull is the piece of the hull you would see looking down from above and the lower hull is the piece of the hull you would see looking up from below. One benefit this algorithm has over the Graham scan is that it will only require us to sort according to x-value instead of by angle – This is a computationally simpler problem.

1.2 Algorithm

TODO: Reread some of this and rephrase some steps.

- First, sort points by their x-coordinates in ascending order. Let x_{min} be the point with the smallest x-value, and likewise define x_{max} as the point with the largest x-value.
- Second, split points into two categories based on whether they are above or below the line that runs from x_{min} to x_{max}. The points above the line (including endpoints of line) will be referred to as x_U and points below (including endpoints of line) will be referred to as x_L.
- Third, beginning with x_{min} and iterating through points in x_L in ascending order… This will build our lower hull
- Fourth, beginning with x_{max} and iterating through points in x_U in descending order… This will build our upper hull

1.2.1 Part 1: Sort

Since we sort by x-coordinate, it is easy to just call out to the sorting function provided by your programming language – Here we will simply call Julia's `sort`.

1.2.2 Part 2: Split into two categories

We now need to split the points into our two categories: x_L and x_U. We can do this by taking each x-coordinate and computing the y-coordinate that corresponded to being on the line between x_L and x_U. If the y-coordinate of the point is above the line then it belongs to x_U and if it is below then it belongs to x_L.

We write a function which splits the points into x_L and x_U below.

```
In [2]: function split_xL_xU{T<:Real}(psort::Vector{Point{2, T}})
            # Allocate space to identify where points belong
            # We make endpoints true because we want them in both sets
            npts = length(psort)
            xL_bool = Array(Bool, npts)
            xL_bool[1] = true; xL_bool[end] = true
            xU_bool = Array(Bool, npts)
            xU_bool[1] = true; xU_bool[end] = true

            # Get the endpoints and slope
            pmin = psort[1]; pmax = psort[end]
            ls = LineSegment(pmin, pmax)

            # We check whether it is above or below the line between
            # the points, but could also check whether it was clockwise
            # or counter-clockwise (above/below seems like less operations)
            # Iterate through all points except endpoints
            @inbounds for i=2:npts-1
                # Pull out x and y values for ith point
                p_i = psort[i]
                xi, yi = p_i[1], p_i[2]

                # Compute where y on line would be
                yi_line = evaluatey(ls, xi)

                if yi_line <= yi
                    xL_bool[i] = false
                    xU_bool[i] = true
                else
                    xL_bool[i] = true
                    xU_bool[i] = false
                end
            end
            xL = psort[xL_bool]
            xU = psort[xU_bool]

            return xL, xU
        end

Out[2]: split_xL_xU (generic function with 1 method)
```

1.2.3 Part 3 & 4: Finding Upper and Lower Hulls

In this step, we will use an approach that is similar to the Graham scan algorithm. If we are finding the upper (lower) hull then we will begin with x_{min} (x_{max}) and work our way towards x_{max} (x_{min}).

Just as in the Graham scan, we will need to identify the direction of the turn generated by three points. Imagine we have three points p_{i-1}, p_i, p_{i+1}.

- If these points constitute a counter-clockwise turn as desired then we proceed to compare points p_i, p_{i+1}, p_{i+2}.
- If these points constitute a clockwise turn then we know that p_i is not an element of the extreme points and can discard it. We would then "take a step backwards" and compare $p_{i-2}, p_{i-1}, p_{i+1}$

We follow until we reach the end point which will be x_{max} when we are finding the lower hull and x_{min} when we are finding the upper hull.

Notice that this is the same function from the convex hull notebook that we used in the Graham scan algorithm.

1.3 Implementation

We can combine the previous steps into this short function which will compute the convex hull using the monotone chain algorithm.

```
In [3]: function _monotonechain{T<:Real}(p::Vector{Point{2, T}})

            # Sort points
            psort = sort!(p)

            # Split into upper and lower
            xl, xu = split_xL_xU(psort)

            lh = wrappoints(xl)
            uh = wrappoints(reverse!(xu))

            return [lh[1:end-1]; uh[1:end-1]]
        end
```

```
Out[3]: _monotonechain (generic function with 1 method)
```

1.4 Example Usage

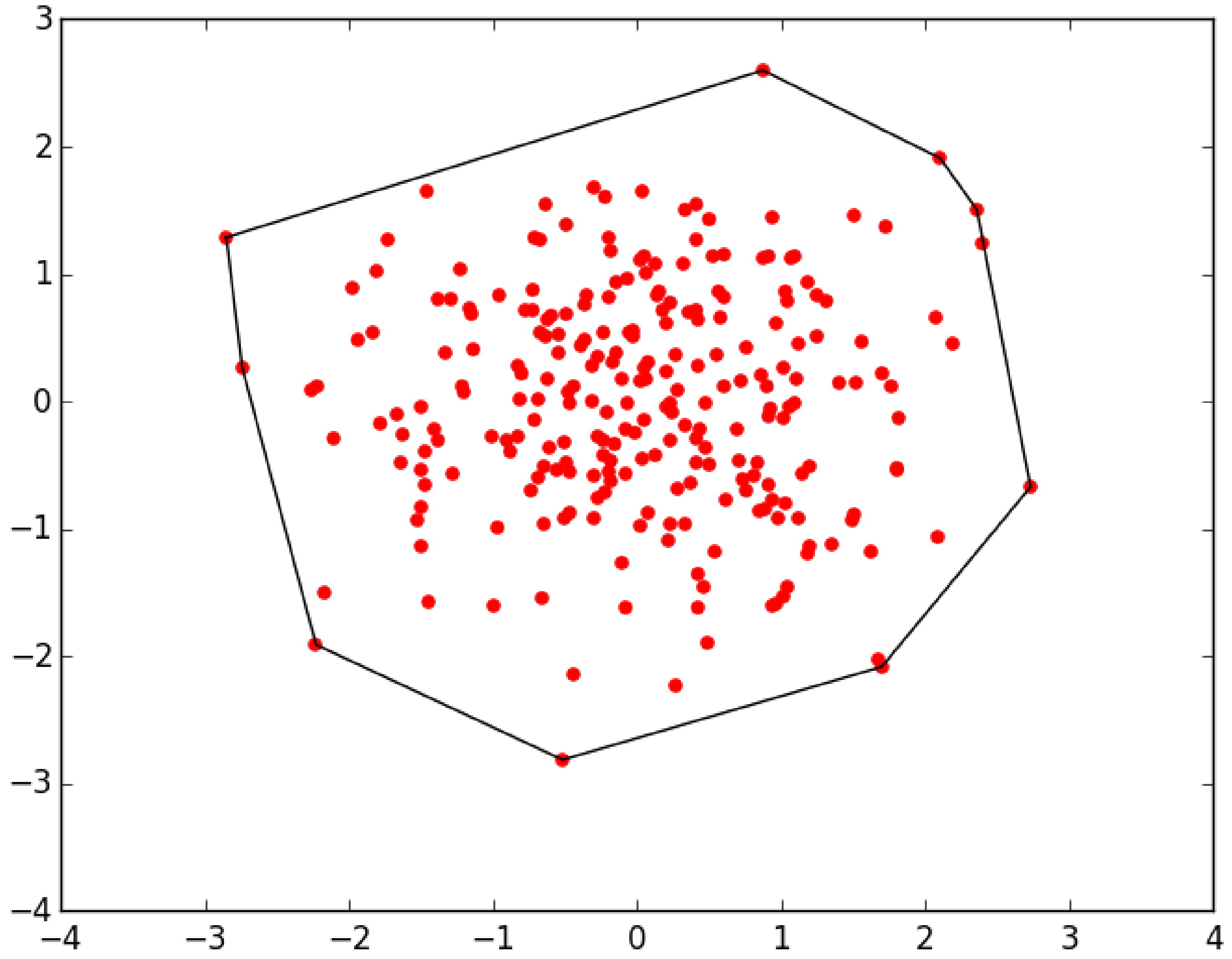
We can now do a simple example of how this function would be called.

```
In [4]: srand(1112016)
        p = [Point(randn(), randn()) for i=1:250]
        ep = _monotonechain(p)

        # Add the first point to end to make plot pretty
        push!(ep, ep[1]);
```

```
In [5]: fig, ax = subplots()

        ax[:scatter]([el[1] for el in p], [el[2] for el in p], color="r")
        ax[:plot]([el[1] for el in ep], [el[2] for el in ep], color="k")
```



```
Out[5]: 1-element Array{Any,1}:
         PyObject <matplotlib.lines.Line2D object at 0x7f5a69ccb690>

/home/chase/.julia/v0.4/Conda/deps/usr/lib/python2.7/site-packages/matplotlib/collection
  if self._edgecolors == str('face'):
```

1.5 References

- A. M. Andrew, "*Another Efficient Algorithm for Convex Hulls in Two Dimensions*", Info. Proc. Letters 9, 216-219 (1979).