```python
# Convex-Hull

import sys
import argparse
import math

# point tuple indexes
p_x = 0
p_y = 1

def area2(a, b, c):
    """ The area of a triangle with three points. """
    return ((b[p_x] - a[p_x])*(c[p_y] - a[p_y]) -
            (b[p_y] - a[p_y])*(c[p_x] - a[p_x]))

def is_left_of(a, b, c):
    """ Checks if the point a is to the left of the line from b to c. """
    return area2(a, b, c) > 0

def is_colinear(a, b, c):
    """ Checks if the point a is collinear with the line from a to b. """
    return area2(a, b, c) == 0

def is_right_of(a, b, c):
    """ Checks if the point a is to the right of the line from b to c. """
    return area2(a, b, c) < 0

def brute(points):
    """
    Find the convex hull of a number of points using a brute for method.
    """
    hull_pairs = []
    # for every two points
    for i in xrange(len(points)):
        for j in xrange(len(points)):
            if i != j:
                # Go through all the points and check if they're all on one side
                on_right = False # if we have points on the right
                on_left = False # if we have points on the left
                colinear = False # if we have colinear points

                # for all points that are not part of the line we're considering
                for p in points:
                    if p != points[i] and p != points[j]:
                        a = area2(p, points[i], points[j])
                        if a > 0:
                            on_left = True
                        elif a < 0:
                            on_right = True
                        else:
                            colinear = True

                # check if we have situations where all points are on one side
                # or colinear (all points are in a line so the convex hull is
                # the line)
                if (on_right and not on_left and not colinear) \
                   or (on_left and not on_right and not colinear) \
                   or (colinear and not on_right and not on_left):
                    hull_pairs.append((points[i], points[j]))

    return hull_pairs


def vector(p1, p2):
```

```
    """ Returns the vector of two points. """
    return (p2[p_x] - p1[p_x], p2[p_y] - p1[p_y])

def flip(x, y):
    return y, x

def graham_scan(points):
    n = len(points)

    # find the anchor point (O(n))
    anchor = None
    for i, p in enumerate(points):
        if anchor is None or p[p_y] < points[anchor][p_y] or \
           (p[p_y] == points[anchor][p_y] and p[p_x] < points[anchor][p_x]):
           # go ahead and set the anchor
           anchor = i

    # sort the remaining points by creating a new list of indecies that are
    # sorted (O(nlogn))
    p_srtd = [x for x in xrange(n) if x != anchor]
    p_srtd.sort(
        key=lambda x: math.atan2(*flip(*vector(points[anchor], points[x])))
    )

    # insert the anchor point
    print "anchor", anchor, points
    points.insert(0, points[anchor])

    # insert a sentinal point
    points.insert(0, points[0])
    print len(points), points

    points = [points[i] for i in p_srtd if isinstance(i, int)]

    # push the first three points onto the stack
    m = 2
    i = 3
    while i < n:
        while area2(points[m-1], points[m], points[i]) <= 0:
            print 'i', i
            print 'm', m
            if m == 2:
                points[m], points[i] = points[i], points[m]
                i += 1
            else:
                m -= 1
        m += 1
        points[m], points[i] = points[i], points[m]

        i += 1

    return points

def gift_wrap(points):
    """
    Find the convex hull for a number of points using the gift
    wrapping method, also known as the Jarvis march method.
    """
    # degenerate cases
    if len(points) == 1 or len(points) == 2:
        return points

    hull = []
    # find the leftmost point
```

```
    point_on_hull = (float('inf'), float('inf'))
    for p in points:
        if (p[p_x] < point_on_hull[p_x]) or \
           (p[p_x] == point_on_hull[p_x] and p[p_y] < point_on_hull[p_y]):
            point_on_hull = p

    i = 0 # TODO: maybe take out
    while True:
        hull.append(point_on_hull)

        # initial endpoint candidate
        endpoint = points[0]
        for j in xrange(1,len(points)):
            if endpoint == point_on_hull or \
               is_left_of(points[j], point_on_hull, endpoint):
                endpoint = points[j]
        i = i + 1
        point_on_hull = endpoint

        # end case
        if endpoint == hull[0]: # we got to the beginning of the hull
            break

    return hull

def parse_points_file(file_name):
    f = open(file_name)
    if f:
        points = []
        for l in f:
            try:
                points.append(tuple(map(int, l.split(','))))
            except:
                pass
        return points
    else:
        return []

def plot(points, hull):
    import matplotlib.pyplot as plt

    # setup the figure
    fig = plt.figure()
    ax = fig.add_subplot(111)

    # plot the points
    x, y = zip(*points)
    ax.plot(x, y, 'ro')

    # check if we're plotting hull pairs or an ordered point list
    print hull
    if isinstance(hull[0][0], tuple):
        # we have hull pairs
        # loop through the connections
        for h in hull:
            # plot the hull
            x, y = zip(*h)
            ax.plot(x, y, 'go-')
    else:
        # we have a list of hull points
        # close the hull
        hull.append(hull[0])

        # plot the hull
```

```
        x, y = zip(*hull)
        ax.plot(x, y, 'go-')

    # set additional settings and show plot
    x, y = zip(*points)
    # the fudge factor represents the spacing around the plot so it looks nicer
    fudge_factor = math.ceil((max(x) + max(y))/2 * 0.1)
    ax.grid()
    ax.set_xlim(min(x)-fudge_factor,max(x)+fudge_factor)
    ax.set_ylim(min(y)-fudge_factor,max(y)+fudge_factor)
    ax.set_title('Convex Hull')
    plt.show()

def main():
    # A dict of the supported methods
    methods = {
        'gift_wrap': gift_wrap,
        'brute': brute,
        'graham_scan': graham_scan,
    }

    # parse command line arguments
    parser = argparse.ArgumentParser(
        description="Find the convex hull of points using varius methods.")
    parser.add_argument('--graph', dest='graph',# nargs='?',
                        const=True, default=False, action='store_const',
                        help='Produce a graph.')
    parser.add_argument('method', metavar='METHOD', type=str,
                         help='The type of method to use. Supported methods are: '+\
                              ', '.join(methods))
    parser.add_argument('filename', metavar='FILE', type=str,
                        help='The file to parse the points out of. Expects a '+\
                             'file of lines with comma seperated x/y values.')
    args = parser.parse_args()

    # open the file and parse out the points
    points = parse_points_file(args.filename)

    # if we have points let's git-r-done
    if points:
        try:
            result = methods[args.method](points)
        except KeyError:
            sys.stderr.write("Invalid method selected\n")
            sys.exit(1)

        print "Convex hull: " + repr(result)

        if args.graph:
            plot(points, result)
    else:
        print "Can't parse points file."

if __name__ == "__main__":
    main()
```