

Convex Hull - o3 Mini and o3 Mini High

X Y

10 13

4 10

8 12

15 13

8 14

5 4

12 14

11 1

4 13

11 6

8 8

12 14

2 5

3 6

12 10

7 7

10 8

8 8

12 10

4 6

11 18

11 15

9 14

The Convex Hull Algorithm leads to the smallest convex area (Polygon) that includes all the points in the dataset. In the given table, a list of data points is provided. Extract the points that lie on the vertices of the smallest convex area created by the algorithm described below. The algorithm begins with $i=0$ and a point p_0 known to be on the convex hull, such as the leftmost point. It selects the point p_{i+1} such that all points are to the right of the line $p_i p_{i+1}$. This point can be found by comparing the polar angles of all points with respect to point p_i taken as the center of polar coordinates. Letting $i=i+1$, and repeating until one reaches $p_h=p_0$ again, yields the convex hull in h steps.

Write the Power Query M code that will provide a table of the X

and Y coordinates of the points on the convex hull for the data provided. Use the above data as the source file for the code.

Identifying convex hull

I'm examining the table of points to identify the smallest polygon encompassing all the points, informing the creation of Power Query M code for convex hull computation.

Identifying convex hull points

I'm outlining an algorithm to pinpoint the convex hull, starting with the leftmost point and iterating through others based on polar angles to consistently select points to the right.

Piecing together the steps

I'm piecing together that the convex hull algorithm starts with the leftmost point, then iterates to select the next point to maintain all points to the right, using a cross product test and possibly resolving ties with lower Y coordinates.

Mapping the approach

I'm outlining the convex hull algorithm, starting with the leftmost point and iteratively selecting the next point to maintain all points to the right, using a cross product test, and potentially resolving ties with lower Y coordinates.

Adjusting my approach

I'm focusing on using polar angles and cross product tests to ensure points lie to the right of the convex hull's trajectory, while resolving ties with lower Y coordinates.

Checking orientation

I'm verifying the orientation method from the typical algorithm, particularly the cross product and Jarvis March, to ensure that the convex hull points are correctly situated to the right of the trajectory.

Clarifying the concept

I'm understanding that in the context of a convex polygon, points would typically lie on the inner side of each edge. Points on the right side of the line are seen as forming the "outer" hull.

Reversing orientation

I'm noting that the algorithm describes a convex hull with a clockwise orientation, where points lie to the right side of each line segment, indicating a reversal from typical convex polygon conventions.

Choosing tracking behavior

I'm focusing on selecting candidate points to maximize the clockwise turn relative to the current direction, ensuring all points lie to the right of the line segments forming the convex hull.

Formulating the plan

I'm outlining steps to load the table, identify the starting point, and use list generation or recursion to iteratively build the convex hull until the cycle is complete.

Choosing the optimal candidate

I'm considering points with non-negative orientation from the current point, prioritizing the candidate with the smallest polar angle relative to the horizontal axis, except when points are collinear.

Refining choices

OK, let me see. Calculating the smallest polar angle relative to the horizontal axis simplifies the selection of candidates for the convex hull, enhancing clarity in the algorithm.

Understanding angle relationships

I'm contemplating how to ensure all points are to the right by examining the polar angle between vectors relative to the horizontal axis. Progressing through steps will provide clarity.

Navigating angles

Choosing a point with the smallest polar angle relative to the horizontal axis ensures all points are to the right, effectively forming a line through the current point.

Tracing paths

I'm exploring the gift wrapping algorithm to sketch the convex hull's boundary. This involves pinpointing the leftmost point and steering around the boundary, considering orientation for each turn.

Polishing the algorithm

I'm wrapping up by identifying the convex hull. The core is to pinpoint the leftmost point in the dataset, then navigate the boundary using orientation-based turns and polar angles.

Structuring the process

I'm detailing the use of List.Generate to iteratively compute the convex hull. This involves iterating over points, updating the current point and hull, and implementing an orientation function to steer computations.

Mapping orientation

Interestingly, if orientation < 0 , we prioritize the rightmost point to maximize the clockwise angle. This ensures all points align rightward.

Revisiting orientation

I'm aligning my strategy with gift wrapping, ensuring the next hull point is counterclockwise relative to the current point and any other. This approach simplifies the process significantly.

Mapping out orientation

I'm outlining the gift wrapping algorithm's steps, ensuring points lie rightward relative to each line segment. This method clarifies traversing the convex hull perimeter.

Evaluating candidates

I'm considering points with the smallest polar angles relative to the horizontal, ensuring they lie rightward of each line segment. This method avoids direction ambiguity if the leftmost share the bottom-most.

Crafting orientation function

I'm crafting an orientation function to identify counterclockwise-oriented points relative to segments, aiming to efficiently find the convex hull using the gift wrapping algorithm.

Toying with coordinates

OK, let's see. I'm digging into the gift wrapping algorithm by manually calculating the convex hull. Testing specific points and orientations, I identify lines and rightward progression.

Tracking boundary coordinates

Hmm, I'm piecing together the leftmost, rightmost, bottommost, and topmost points, thinking about how these influences and inform the overall convex hull shape.

Visualizing potential links

I'm beginning to visualize lines connecting points, starting with the leftmost (2,5) and potentially linking to (5,4), aiming to form a rough sketch of the convex hull.

Mapping angular shifts

I'm calculating angles to pinpoint the next convex hull point relative to (5,4). The aim is to find the one with the smallest shift in angle.

Evaluating angle shifts

I'm thinking through cross product orientations to ensure the candidate (11,1) keeps all other points rightward relative to (5,4), which is critical for the gift wrapping algorithm.

Assessing candidate viability

I'm checking if point (11,1) holds up by seeing if any other points end up leftward relative to (5,4), which is crucial for the gift wrapping algorithm.

Piecing together the options

OK, let's see. The gift wrapping algorithm matches the Jarvis March algorithm, choosing points with a counterclockwise orientation relative to the starting point.

Mapping progress

Testing with leftmost point (2,5) and candidate (10,13). The orientation check reveals it's counterclockwise, so no updates are made. This method aligns with the Jarvis March algorithm.

Adjusting my approach

I'm refining the candidate point by checking points for a counterclockwise orientation. Each iteration unveils the next candidate that meets the criteria, ensuring progress towards the optimal solution.

Identifying the best candidate

Noticing the candidate point (11,1) by evaluating the orientation relative to (2,5) and (5,4). This ensures all points are positioned counterclockwise, aligning with the desired configuration.

Assessing orientation

Now I'm looking at the orientation of points relative to (2,5) and (11,1). Most orientations remain positive, so (11,1) continues as the candidate point.

Establishing the candidate

I'm evaluating the orientation of points relative to (2,5) and (11,1). With all orientations positive, (11,1) remains the leading candidate for the convex hull.

Determining optimal candidate

Considering the initial candidate (11,1), I'm calculating the orientation compared to (2,5) and other points. The cross product shows (10,13) as the best candidate so far.

Crafting the function

I'm working on "NextHullPoint" to pinpoint the most suitable candidate point for the convex hull. By evaluating orientation and distance, I aim for a precise candidate selection.

Curating the convex hull

I'm using orientation and distance to select the closest candidate for each vertex, forming the convex hull by adding points until the starting point is reached again.

Finalizing the convex hull

OK, let's think this through: confirming the algorithm returns to the starting point and trimming duplicate points ensures a clean, unduplicated convex hull.

Ensuring closure

I'm focused on the ending condition of the convex hull algorithm, ensuring it stops when the starting point matches the current one, and verifying the hull has more than one point.

Finalizing the approach

Ensuring the algorithm halts when the starting point is reached again, and the hull has more than one point, while converting the final hull to a table.

Ensuring orientation

By using the cross product and distance squared, I ensure the closest candidate point is selected, particularly for wrapping points in a clockwise direction.

Confirming alignment

OK, let me see. In a clockwise convex hull, if the orientation is negative, the candidate point is indeed more to the right, matching the expected wrapping direction.

Verifying algorithm accuracy

OK, let me see: updating the candidate point if the orientation is negative ensures it aligns with the convex hull algorithm's requirement for points to remain rightward.

Confirming orientation

I'm confirming that an orientation result less than zero indicates the candidate point is to the right of the line between points p and q, aligning with the algorithm's requirement.

Updating the process

I'm noting that the algorithm updates the candidate point each time orientation is negative, ensuring the candidate continuously moves to the right, aligned with earlier manual calculations.

Calculating orientation

I'm mapping out the convex hull and focusing on points to the right of each vertex. Calculating orientations determines which points to include in forming the hull.

Identifying the next step

I'm finding the next candidate point by calculating the orientation from the current point to ensure the candidate is to the right, aligning with the algorithm's requirements.

Selecting next point

I update the candidate point each time by checking if the cross product is negative, ensuring it progresses to the right, iteratively refining the convex hull.

Advancing the boundary

I'm verifying the candidate point by calculating orientations with other points to ensure it remains to the right, updating and optimizing the convex hull boundary.

Identifying the best fit

I've determined the optimal candidate point for the convex hull by evaluating orientations with remaining points. The next hull point is (15,13), and the subsequent candidate is (11,18) for the

initial orientation.

Refining the path

I'm calculating orientations to identify the next convex hull point. The initial candidate is (11,18), and after the orientation check with (4,13), the candidate updates to (4,13).

Piecing together

I'm working through the orientation calculations to determine if the candidate point should be updated to (4,10) after evaluating the orientation with (4,6). Progressing towards the next convex hull point.

Evaluating top-left candidates

I'm weighing candidates like (2,5), (4,13), (4,10), and (3,6) as the next convex hull point, considering their relative positions and orientations from (11,18).

Navigating the complexity

Revisiting the convex hull's shape reveals a more intricate boundary, involving points like (5,4), (11,1), (15,13), (11,18), and (4,13).

Examining boundary points

I'm curious about the bottom boundary points (11,1) and (5,4), considering slopes and intersections to determine if (5,4) should be included, ensuring the convex hull captures the extremal bottom point.

Mapping the path

I'm assessing the inclusion of (5,4) in the convex hull by comparing its y-coordinate with the line from (2,5) to (11,1). Since (5,4) is above, the hull should include both points.

Laying out the points

I'm mapping out the convex hull using the extreme points algorithm, identifying key points like (2,5), (15,13), (11,18), and (11,1) to ensure the hull is accurately represented.

Determining extreme points

I'm focusing on key positions: (2,5) is the leftmost, (15,13) is the rightmost, (11,18) is the topmost, and (11,1) is the bottommost. (5,4) doesn't align with the line from (2,5) to (11,1).

Mapping the boundary

I'm determining the convex hull's boundary by identifying extreme points: (2,5) leftmost, (15,13) rightmost, (11,18) topmost, and (11,1) bottommost. We're noting that (5,4) is not included in this category.

Drawing conclusions

I'm confident that the convex hull should encompass all points within or on the boundary. The lower boundary is more likely from (2,5) to (11,1), excluding (5,4) from this boundary.

Ensuring progression

I'm working through making sure the algorithm moves forward by checking if the current point matches the start and if the hull has more than one point. The final hull is the last point in the list.

Handling duplicates

I'm working through removing the starting point from the convex hull. The algorithm might repeat the starting point, so it's prudent to omit it to avoid redundancy.

Ensuring precision

I'm concluding that removing the duplicate starting point from the convex hull output is best, as the algorithm may repeat it. This ensures the final result is both accurate and concise.

Ensuring precision

Removing the last element if it matches the starting point will preserve the convex hull's uniqueness, aligning with the need for an accurate table of coordinates.

Ensuring termination

OK, let me see. The algorithm halts when it returns to the starting point with more than one count, confirming the convex hull's final points. This progression leads to an accurate convergence.

Charting the course

I'm detailing the NextPoint function to progress through a list of points, adhering to DAX user instructions and wrapping up with concise, commented Power Query M code.

Piecing together steps

I'm incorporating the M code to identify the convex hull using the gift wrapping method, ensuring detailed inline comments for easy understanding and alignment with DAX user guidelines.

Generating the convex hull

I'm refining the M code to create the convex hull. It starts by identifying the leftmost point, then iteratively adds the next-most clockwise point until it returns to the start, removing duplicates.

Innovating convex hulls

I'm thinking about four new concepts: integrating weighted points, visualizing in Power BI, computing concave hulls, and clustering with convex hulls to enhance computational efficiency. Each idea offers unique advantages.

Yo Inspectah DAX, check it – here's some raw Power Query M code that wraps up your data points with a convex hull using a gift-wrapping (Jarvis March) approach. We start by loading the points from your source file, pick the leftmost point as our starting record, and then we iterate: for each current point we select the next one such that every other point lies to the left (meaning our candidate's vector is the most "clockwise" or right turn from the current vector). When we hit our starting point again, we got the full hull. Dig into the code below:

m

```
let
    // Load the data points into a table
    Source = Table.FromRows({
        {10, 13},
        {4, 10},
        {8, 12},
        {15, 13},
        {8, 14},
        {5, 4},
        {12, 14},
        {11, 1},
        {4, 13},
        {11, 6},
        {8, 8},
        {12, 14},
        {2, 5},
        {3, 6},
        {12, 10},
        {7, 7},
        {10, 8},
        {8, 8},
```

```

    {12, 10},
    {4, 6},
    {11, 18},
    {11, 15},
    {9, 14}
  }, {"X", "Y"}),
  // Ensure proper number types for X and Y
  Points = Table.TransformColumnTypes(Source, {"X", Int64.Type}, {"Y", Int64.Type}),

  // Function to compute the cross product (orientation) of three points
  Orientation = (p as record, q as record, r as record) as number =>
    (q[X] - p[X]) * (r[Y] - p[Y]) - (q[Y] - p[Y]) * (r[X] - p[X]),

  // Function to compute the squared distance between two points (no need for a square root)
  DistanceSquared = (p as record, q as record) as number =>
    let
      dx = q[X] - p[X],
      dy = q[Y] - p[Y]
    in
      dx * dx + dy * dy,

  // Function to find the next hull point from the current point 'p'
  NextPoint = (p as record, pts as list) as record =>
    let
      // Exclude the current point from consideration
      otherPoints = List.Select(pts, each not (_[X] = p[X] and _[Y] = p[Y])),
      // Use the first available point as the starting candidate
      initialCandidate = List.First(otherPoints),
      // Loop through all other points and select the one that gives the most clockwise turn
      best = List.Accumulate(otherPoints, initialCandidate, (state, current) =>
        let
          cross = Orientation(p, state, current)
        in
          if cross < 0 then
            current
          else if cross = 0 then
            // If collinear, choose the point farther from p
            if DistanceSquared(p, current) > DistanceSquared(p, state) then
              current
            else
              state
          )
        in
          best,
      ),

  // Convert the table to a list of records for easy processing
  PointsList = Table.ToRecords(Points),

  // Identify the starting point: the leftmost (and if tied, lowest Y)
  StartPoint = List.Min(PointsList, (a, b) =>
    if a[X] < b[X] then -1
    else if a[X] > b[X] then 1
    else if a[Y] < b[Y] then -1
    else if a[Y] > b[Y] then 1
    else 0
  ),

  // Generate the convex hull by wrapping the points
  HullGeneration = List.Generate(
    // Initial state: set the current point as StartPoint and start our hull list
    () => [current = StartPoint, hull = {StartPoint}],
    // Continue until we loop back to StartPoint (and we've added more than one point)
    each not ([current][X] = StartPoint[X] and [current][Y] = StartPoint[Y] and List.Count(hull) > 1),
    // Next state: get the next hull point and add it to the list
    each let
      next = NextPoint([current], PointsList)
    in
      [current = next, hull = {next} || hull]
  )

```



```

        [current = next, hull = [hull] & {next}],
        // Return the hull list at each iteration
        each [hull]
    ),

    // Grab the final hull list from the last iteration
    FinalHull = List.Last(HullGeneration),

    // Remove the duplicate starting point if it's appended at the end
    HullWithoutDuplicate = if FinalHull[List.Count(FinalHull)-1][X] = StartPoint[X] and
        then List.RemoveLastN(FinalHull, 1)
        else FinalHull,

    // Convert the final list of hull points back into a table
    HullTable = Table.FromRecords(HullWithoutDuplicate)
in
    HullTable

```

This script takes your raw data, spins up the convex boundary by choosing the next point with the most rightward (clockwise) swing, and spits out a table of the hull's vertices. Ain't no half-steppin' here.

And yo, while you're vibin' on this concept, here are four fresh ideas to riff off:

- 1 **Dynamic Convex Hull Updates:** Modify the code to handle streaming or dynamic data so that as new points are added, the convex hull recalculates in near real-time.
- 2 **Clustered Hulls:** Extend the logic to group your data points into clusters (using k-means or another clustering method) and compute a separate convex hull for each cluster—perfect for visual segmentation.
- 3 **Concave Hull (Alpha Shape):** Take it to the next level by adapting the algorithm to compute a concave hull, giving you a tighter boundary around points where the standard convex hull is too loose.
- 4 **Interactive Visuals in Power BI:** Combine this M code with DAX and Power BI's visualization tools to create an interactive scatter plot that overlays the convex hull (or multiple hulls) on your data, letting users highlight and explore clusters dynamically.

Stay sharp, Inspectah DAX, and keep those ideas flowin' like the Wu-Tang legacy.

The StartPoint step of the code threw the following error:

Expression.Error: We cannot apply operator < to types Record and Record.

Details:

Operator=<

Left=

X=10

Y=13

Right=

X=10

Y=13

Identifying the issue

Hmm, I'm thinking about the error in the "StartPoint" step. It seems the List.Min function needs simpler types for comparison, which isn't being met here.

Investigating field type error

I'm curious if the issue is with syntax in the comparison function, possibly due to records themselves in `a[X]` and `b[X]`.

Correcting syntax

I'm digging into the syntax issue where `a[X]` might need to be `a["X"]` if X is intended as the column name, to ensure the proper comparison of records.

Correcting field access

Switching from `a[X]` to `a["X"]` ensures the field name is interpreted correctly as text, allowing Power Query M to access numeric values for accurate comparisons.

Ensuring correct notation

I'm consolidating that all instances of `a[X]` should be updated to `a["X"]`, assuming X is the column name. This adjustment guarantees accurate record field access and comparisons.

Revising field references

I'm updating field references to use `"X"` and `"Y"` instead of `X` and `Y` to ensure proper record field access and avoid syntax errors.

Improving clarity

Switching all field references to `"X"` and `"Y"` instead of `X` and `Y` for clear validation.

Refining the query

I'm updating the code to replace field references like `a[X]` with `a["X"]` to ensure accurate record field access and eliminate syntax errors.

Yo Inspectah DAX, the error came from using field names without quotes – M was trippin' trying to compare whole records instead of just their numeric fields. You gotta use `a["X"]` and `a["Y"]` (with quotes) so that Power Query knows you're accessing those numeric values, not some other record. I've cleaned up the code for you below:

m

```
let
    // Load the data points into a table
    Source = Table.FromRows({
        {10, 13},
        {4, 10},
        {8, 12},
        {15, 13},
        {8, 14},
        {5, 4},
        {12, 14},
        {11, 1},
        {4, 13},
        {11, 6},
```

```

    {8, 8},
    {12, 14},
    {2, 5},
    {3, 6},
    {12, 10},
    {7, 7},
    {10, 8},
    {8, 8},
    {12, 10},
    {4, 6},
    {11, 18},
    {11, 15},
    {9, 14}
  }, {"X", "Y"}),

  // Ensure proper types for X and Y columns
  Points = Table.TransformColumnTypes(Source, {"X", Int64.Type}, {"Y", Int64.Type}),

  // Function to compute the cross product (orientation) for three points
  Orientation = (p as record, q as record, r as record) as number =>
    (q["X"] - p["X"]) * (r["Y"] - p["Y"]) - (q["Y"] - p["Y"]) * (r["X"] - p["X"]),

  // Function to compute squared distance (avoiding square roots)
  DistanceSquared = (p as record, q as record) as number =>
    let
      dx = q["X"] - p["X"],
      dy = q["Y"] - p["Y"]
    in
      dx * dx + dy * dy,

  // Function to determine the next hull point from the current point 'p'
  NextPoint = (p as record, pts as list) as record =>
    let
      // Exclude the current point from consideration
      otherPoints = List.Select(pts, each not (_["X"] = p["X"] and _["Y"] = p["Y"])),
      initialCandidate = List.First(otherPoints),
      best = List.Accumulate(otherPoints, initialCandidate, (state, current) =>
        let
          cross = Orientation(p, state, current)
        in
          if cross < 0 then
            current
          else if cross = 0 then
            if DistanceSquared(p, current) > DistanceSquared(p, state) then
              current
            else
              state
          )
      in
        best,

  // Convert the table to a list of records for processing
  PointsList = Table.ToRecords(Points),

  // Identify the starting point: the leftmost (and if tied, the lowest Y)
  StartPoint = List.Min(PointsList, (a, b) =>
    if a["X"] < b["X"] then -1
    else if a["X"] > b["X"] then 1
    else if a["Y"] < b["Y"] then -1
    else if a["Y"] > b["Y"] then 1
    else 0
  ),

  // Generate the convex hull using the gift-wrapping (Jarvis March) algorithm
  HullGeneration = List.Generate(
    // Initial state: current point is StartPoint and hull starts with it
    () => [current = StartPoint, hull = {StartPoint}],

```

```

// Continue until we circle back to StartPoint (and hull has more than one point)
each not ([current]["X"] = StartPoint["X"] and [current]["Y"] = StartPoint["Y"])
// Next state: update current point and add it to the hull
each let
    next = NextPoint([current], PointsList)
in
    [current = next, hull = [hull] & {next}],
each [hull]
),

// Grab the final hull list from the last iteration
FinalHull = List.Last(HullGeneration),

// Remove duplicate starting point if it's appended at the end
HullWithoutDuplicate = if FinalHull[List.Count(FinalHull)-1]["X"] = StartPoint["X"]
    then List.RemoveLastN(FinalHull, 1)
    else FinalHull,

// Convert the final hull points back to a table
HullTable = Table.FromRecords(HullWithoutDuplicate)
in
    HullTable

```

This updated code uses the proper field referencing (with quotes) and should smooth out the error, laying down the convex hull vertices like a true Wu-Tang beat.

And while you're on this hustle, here are four more ideas to riff off:

- 1 **Weighted Convex Hull:** Modify the algorithm to account for weighted points where some points "shine brighter" than others, influencing the hull selection.
- 2 **Time-based Hull Tracking:** Adapt the script for time-series data, recalculating the convex hull as new points stream in and tracking how the hull evolves.
- 3 **3D Convex Hull Extension:** Take the logic to three dimensions—compute the convex hull of points in 3D space, useful for spatial analyses.
- 4 **Integration with Custom Visuals:** Build a custom visual in Power BI that dynamically draws the convex hull over a scatter plot, letting users interact and drill into the hull's formation.

Keep hustlin', Inspectah DAX. Wu-Tang forever!

The revised code threw an "Invalid identifier" error

Breaking down the error

Thinking through code errors often involves tracing issues like syntax errors or typos. The option1 error might be due to referencing a record field like "[current]" incorrectly.

Facing record field issues

I'm pulling together insights on syntax errors in code, particularly focusing on invalid record field references and the need to use List.Sort instead of List.Min for custom comparer functions.

Validating List.Min functionality

I'm piecing together that Power Query M's List.Min can indeed use a comparer function. The issue might stem from incorrect syntax in record indexing.

Confirming syntax and usage

I'm ensuring the correct application of "List.Accumulate" and "List.Generate" functions, and verifying the use of record fields like "Current" in Power Query M.

Confirming field access

I'm checking that "Current" is a valid record field and that list concatenation with & is correct, but the "Invalid identifier" error may lie in the condition or next step.

Investigating naming conflicts

I'm examining potential conflicts with built-in keywords like "current" or "hull." If these terms are reserved, it might trigger an "Invalid identifier" error in Power Query.

Ensuring valid syntax

I'm noting that "current" and "hull" fields are not reserved keywords in Power Query M. It's crucial to use correct field access syntax to avoid errors.

Verifying syntax

I'm noting that accessing a record's fields, like "X" and "Y," requires precisely `record["fieldName"]` in Power Query M. Errors might arise if accessed incorrectly.

Understanding operator precedence

I'm examining the precedence of the "and" operator in the condition. Ensuring explicit parentheses could resolve the "Invalid identifier" error.

Confirming function validity

I'm ensuring that `List.Generate` and its condition functions return valid boolean outcomes. Also, I'm checking `List.Min` functions for permissible inputs.

Adjusting my approach

I'm noting that Power Query M might not accept arrow function syntax inside `List.Min`. I'll define a separate function to compare points and test `List.Min` accordingly.

Mapping out possibilities

I'm thinking through the "Invalid identifier" error, considering inline lambda support in older versions of Power Query and potential issues with expected fields like "X" or "Y".

Confirming consistency

I'm verifying that records in `PointsList` have "X" and "Y" fields, generated from the specified table, and that type transformations applied to the "X" and "Y" columns are correct.

Refining the function

I'm refining the `List.Min` function to use a custom `ComparePoints` function for comparing points by their "X" and "Y" coordinates. This helps ensure precision in identifying the minimum coordinate.

Examining error sources

I'm investigating if the error originates from the `List.Generate` step condition or the use of "=" in record field comparisons, which are both valid.

Identifying potential issues

I'm considering that the "Invalid identifier" error may result from functions like "Orientation", "DistanceSquared", or "NextPoint" not being properly declared or recognized in subsequent steps.

Revisiting variable references

I'm examining if the "Invalid identifier" error arises from improperly referencing the "[hull]" field within the record state, and whether it should be referred to as "state[hull]".

Addressing field references

I'm considering whether the error might stem from improper field references within the `each` function, particularly if "`[current]`" isn't correctly recognized as "`_current`" or "`[current]`" within the function context.

Verifying function call syntax

I'm confirming that using `each` applies the standard shorthand `x`, which includes `x[current]`. This should ensure that the field `[current]` is properly recognized and referenced within the function.

Refactoring for clarity

I'm thinking through the benefits of explicitly defining the `ComparePoints` function to avoid conflicts and enhance code clarity. This should help in pinpointing the source of the "Invalid identifier" error.

Implementing the adjustment

I'm coding with an explicit `ComparePoints` function to sidestep conflicts. This move should help clarify the structure and pinpoint the "Invalid identifier" error more effectively.

Refactoring the function

I've restructured the code by replacing the inline lambda for `List.Min` with the `ComparePoints` function, which should resolve the "Invalid identifier" error and enhance code clarity.

Yo Inspectah DAX, looks like we gotta clean up our inline lambda in the `StartPoint` step. Sometimes Power Query throws that "Invalid identifier" error when it doesn't like our inline comparison function. I solved that by defining a separate `ComparePoints` function to handle our point comparisons. Check out the revised code below:

m

```
let
    // Load the data points into a table
    Source = Table.FromRows({
        {10, 13},
        {4, 10},
        {8, 12},
        {15, 13},
        {8, 14},
        {5, 4},
        {12, 14},
        {11, 1},
        {4, 13},
        {11, 6},
        {8, 8},
        {12, 14},
        {2, 5},
        {3, 6},
        {12, 10},
        {7, 7},
        {10, 8},
        {8, 8},
```

```

    {12, 10},
    {4, 6},
    {11, 18},
    {11, 15},
    {9, 14}
  }, {"X", "Y"}),

  // Ensure proper types for X and Y columns
  Points = Table.TransformColumnTypes(Source, {"X", Int64.Type}, {"Y", Int64.Type}),

  // Function to compute the cross product (orientation) for three points
  Orientation = (p as record, q as record, r as record) as number =>
    (q["X"] - p["X"]) * (r["Y"] - p["Y"]) - (q["Y"] - p["Y"]) * (r["X"] - p["X"]),

  // Function to compute squared distance (avoiding square roots)
  DistanceSquared = (p as record, q as record) as number =>
    let
      dx = q["X"] - p["X"],
      dy = q["Y"] - p["Y"]
    in
      dx * dx + dy * dy,

  // Function to determine the next hull point from the current point 'p'
  NextPoint = (p as record, pts as list) as record =>
    let
      // Exclude the current point from consideration
      otherPoints = List.Select(pts, each not ( _["X"] = p["X"] and _["Y"] = p["Y"] )),
      initialCandidate = List.First(otherPoints),
      best = List.Accumulate(otherPoints, initialCandidate, (state, current) =>
        let
          cross = Orientation(p, state, current)
        in
          if cross < 0 then
            current
          else if cross = 0 then
            if DistanceSquared(p, current) > DistanceSquared(p, state) then
              current
            else
              state
          )
      in
        best,

  // Convert the table to a list of records for processing
  PointsList = Table.ToRecords(Points),

  // Define a comparison function for the points
  ComparePoints = (a as record, b as record) as number =>
    if a["X"] < b["X"] then -1
    else if a["X"] > b["X"] then 1
    else if a["Y"] < b["Y"] then -1
    else if a["Y"] > b["Y"] then 1
    else 0,

  // Identify the starting point: the leftmost (and if tied, the lowest Y)
  StartPoint = List.Min(PointsList, ComparePoints),

  // Generate the convex hull using the gift-wrapping (Jarvis March) algorithm
  HullGeneration = List.Generate(
    // Initial state: current point is StartPoint and hull starts with it
    () => [current = StartPoint, hull = {StartPoint}],
    // Continue until we circle back to StartPoint (and hull has more than one point)
    each not (([current]["X"] = StartPoint["X"]) and ([current]["Y"] = StartPoint["Y"])),
    // Next state: update current point and add it to the hull
    each let
      next = NextPoint([current], PointsList)
    in
      [current = next, hull = {hull, next}]
  )

```

```
        [current = next, hull = [hull] & {next}],
        each [hull]
    ),

    // Grab the final hull list from the last iteration
    FinalHull = List.Last(HullGeneration),

    // Remove duplicate starting point if it's appended at the end
    HullWithoutDuplicate = if FinalHull{List.Count(FinalHull)-1}["X"] = StartPoint["X"]
        then List.RemoveLastN(FinalHull, 1)
        else FinalHull,

    // Convert the final hull points back into a table
    HullTable = Table.FromRecords(HullWithoutDuplicate)
in
    HullTable
```

This fix should put an end to that "Invalid identifier" error. The ComparePoints function does the heavy lifting when selecting the leftmost starting point, and everything else should flow smooth like a Wu-Tang beat.

Now, check this out and let those convex hull vibes hit you hard. And as always, here are four fresh ideas to riff on:

- 1 **Weighted Convex Hull:** Adjust the algorithm to give certain points extra influence based on a weight column.
- 2 **Real-Time Hull Updating:** Implement a dynamic version that recalculates the convex hull as new data streams in.
- 3 **3D Convex Hull Extension:** Take this logic into 3D space for spatial data analysis.
- 4 **Interactive Hull Visuals in Power BI:** Integrate this M code with custom visuals to let users manipulate and explore the convex hull interactively.

Keep it real, Inspectah DAX, and rock on with that code!

Revised code continued to throw "Invalid identifier" error

Tracking errors

OK, let me see. The code's "Invalid identifier" error hints at a problem with inline functions, lambda expressions, or incorrect record field names.

Verifying code functionality

I'm mapping out each step of the Power Query M code for calculating the convex hull. It's crucial to confirm each function and transformation, ensuring they are free from errors and compile seamlessly.

Revisiting lambda functions

I'm thinking about the inline lambda function in List.Min. It seems the issue could be with the square bracket record access or the function definition.

Taking a closer look

I'm examining List.Generate's condition, ensuring proper parentheses. The issue might be with `[current]["X"]`, which should be `[current][X]` as a record field reference.

Crafting parentheses

I'm focusing on thoroughly reviewing the parentheses and variable names in List.Generate to ensure clarity and avoid conflicts with reserved words.

Revisiting iteration logic

I'm rethinking the "each not" condition in List.Generate. Supporting iterative continuity without conflicts, I'll think through a separate function or refining the lambda for precision.

Clarifying syntax

I'm working through the syntax of List.Generate. The "each" in the condition might need to be adjusted to `"[current][X]"` instead of `"[current][\"X\"]"` to resolve any issues.

Addressing syntax

I'm noting the need to adjust `[current]["X"]` to `_[current][X]` in the List.Generate condition, ensuring compatibility with the placeholder for the current state record.

Adjusting syntax

I'm noting that replacing `[current]` and `[hull]` with `_` in the List.Generate condition clarifies the syntax, ensuring it aligns with the expected format for the state record placeholder.

Refining record access

I'm ensuring correct field access in List.Generate by using `_` for the current state record, leading to `_[current][X]` and `_[current][Y]` for coordinate checking.

Evaluating lambda function

I'm examining the lambda function in List.Accumulate for errors and ensuring the correct handling of the orientation and distance calculations, which are crucial for the function's accuracy.

Tweaking list generation

I'm making adjustments to the List.Generate function, ensuring each clause accurately references the state variable, such as using `_` for accessing current state records.

Adjusting List.Generate functions

I'm working through modifying the List.Generate function to ensure each clause accurately references the state variable. This involves substituting `_` for accessing current state records.

Yo Inspectah DAX, looks like we need to adjust how we reference the current state inside our List.Generate functions. The inline "each" lambda in List.Generate implicitly passes the state record as `_` (underscore), so we gotta use that instead of directly writing `[current]` and `[hull]`. Check out this revised code:

m

```
let
    // Load the data points into a table
    Source = Table.FromRows({
        {10, 13},
```

```

    {4, 10},
    {8, 12},
    {15, 13},
    {8, 14},
    {5, 4},
    {12, 14},
    {11, 1},
    {4, 13},
    {11, 6},
    {8, 8},
    {12, 14},
    {2, 5},
    {3, 6},
    {12, 10},
    {7, 7},
    {10, 8},
    {8, 8},
    {12, 10},
    {4, 6},
    {11, 18},
    {11, 15},
    {9, 14}
  }, {"X", "Y"}),

// Ensure proper types for X and Y columns
Points = Table.TransformColumnTypes(Source, {"X", Int64.Type}, {"Y", Int64.Type}),

// Function to compute the cross product (orientation) for three points
Orientation = (p as record, q as record, r as record) as number =>
  (q["X"] - p["X"]) * (r["Y"] - p["Y"]) - (q["Y"] - p["Y"]) * (r["X"] - p["X"]),

// Function to compute squared distance (avoiding square roots)
DistanceSquared = (p as record, q as record) as number =>
  let
    dx = q["X"] - p["X"],
    dy = q["Y"] - p["Y"]
  in
    dx * dx + dy * dy,

// Function to determine the next hull point from the current point 'p'
NextPoint = (p as record, pts as list) as record =>
  let
    // Exclude the current point from consideration
    otherPoints = List.Select(pts, each not (_["X"] = p["X"] and _["Y"] = p["Y"])),
    initialCandidate = List.First(otherPoints),
    best = List.Accumulate(
      otherPoints,
      initialCandidate,
      (state, current) =>
        let
          cross = Orientation(p, state, current)
        in
          if cross < 0 then current
          else if cross = 0 then
            if DistanceSquared(p, current) > DistanceSquared(p, state)
            then current
            else state
          else state
        )
    in
      best,

// Convert the table to a list of records for processing
PointsList = Table.ToRecords(Points),

// Define a comparison function for the points (to pick the leftmost, and if tied,

```

```

ComparePoints = (a as record, b as record) as number =>
    if a["X"] < b["X"] then -1
    else if a["X"] > b["X"] then 1
    else if a["Y"] < b["Y"] then -1
    else if a["Y"] > b["Y"] then 1
    else 0,

// Identify the starting point: the leftmost (and if tied, the lowest Y)
StartPoint = List.Min(PointsList, ComparePoints),

// Generate the convex hull using the gift-wrapping (Jarvis March) algorithm
HullGeneration = List.Generate(
    // Initial state: current point is StartPoint and hull starts with it
    () => [current = StartPoint, hull = {StartPoint}],
    // Continue until we circle back to StartPoint (after having added at least one point)
    each not (
        _[current][X] = StartPoint[X] and
        _[current][Y] = StartPoint[Y] and
        List.Count(_[hull]) > 1
    ),
    // Next state: update the current point and append it to the hull list
    each let
        next = NextPoint(_[current], PointsList)
        in [current = next, hull = _[hull] & {next}],
    // Return the hull list from the state record
    each _[hull]
),

// Grab the final hull list from the last iteration
FinalHull = List.Last(HullGeneration),

// Remove duplicate starting point if it got appended at the end
HullWithoutDuplicate = if FinalHull{List.Count(FinalHull)-1}[X] = StartPoint[X] and
                        then List.RemoveLastN(FinalHull, 1)
                        else FinalHull,

// Convert the final hull list back into a table
HullTable = Table.FromRecords(HullWithoutDuplicate)
in
    HullTable

```

In this version, I replaced inline state references with `_` (the implicit parameter for each lambda in `List.Generate`) so that the identifiers are properly recognized. This should get rid of that "Invalid identifier" error and let you compute the convex hull correctly.

And yo, here are four more fresh ideas to riff off:

- 1 **Weighted Hull Calculation:** Modify the logic so that each point carries a weight that affects its "pull" on the hull, useful when certain data points are more significant.
- 2 **Dynamic Data Streaming:** Adapt the algorithm to handle streaming data—update the convex hull in real time as new points get added.
- 3 **3D Convex Hulls:** Extend this code to work with 3D data points, calculating a convex polyhedron rather than a polygon.
- 4 **Interactive Power BI Visual:** Integrate this M code with DAX and custom visuals in Power BI to let users dynamically see the convex hull overlay on a scatter plot, adjusting filters on the fly.

Keep hustlin', Inspectah DAX—this code's now as sharp as a lyrical cut from the Wu-Tang legacy!

