```
public static int FindLineCircleIntersections(Vector3 circleCenter, double radius,
                                              Vector3 lineStart, Vector3 lineEnd,
                                              out Vector3 intersection1, out Vector3 intersection2)
    {
        double dx, dy, A, B, C, det, t;

        dx = lineEnd.X - lineStart.X;
        dy = lineEnd.Y - lineStart.Y;

        A = dx * dx + dy * dy;
        B = 2 * (dx * (lineStart.X - circleCenter.X) + dy * (lineStart.Y - circleCenter.Y));
        C = (lineStart.X - circleCenter.X) * (lineStart.X - circleCenter.X) +
            (lineStart.Y - circleCenter.Y) * (lineStart.Y - circleCenter.Y) -
            radius * radius;

        det = B * B - 4 * A * C;

        if ((A <= 0.0000001) || (det < 0))
        {
            // No real solutions.
            intersection1 = new Vector3(float.NaN, float.NaN, float.NaN);
            intersection2 = new Vector3(float.NaN, float.NaN, float.NaN);
            return 0;
        }
        else if (det == 0)
        {
            // One solution.
            t = -B / (2 * A);
            intersection1 = new Vector3((float)(lineStart.X + t * dx), (float)(lineStart.Y + t * dy), lineStart.Z);
```

```
            intersection2 = new Vector3(float.NaN, float.NaN, float.NaN);
            return 1;
        }
        else
        {
            // Two solutions.
            t = (float)((-B + Math.Sqrt(det)) / (2 * A));
            intersection1 = new Vector3((float)(lineStart.X + t * dx), (float)(lineStart.Y + t * dy), lineStart.Z);
            t = (float)((-B - Math.Sqrt(det)) / (2 * A));
            intersection2 = new Vector3((float)(lineStart.X + t * dx), (float)(lineStart.Y + t * dy), lineStart.Z);
            return 2;
        }
    }

public static bool IsPointOnArc(Vector2 point, Vector2 arcCenter, double arcRadius, double startAngle, double endAngle)
    {
        double buffer = 0.000001; // Tolerance for floating point comparison

        // Check if the point is on the circle defined by the arc.
        double dist = Vector2.Distance(point, arcCenter);
        if (Math.Abs(dist - arcRadius) > buffer)
        {
            // The point is not on the arc's circle.
            return false;
        }

        // Calculate the angle of the point relative to the center of the arc.
        double angleOfPoint = Math.Atan2(point.Y - arcCenter.Y, point.X - arcCenter.X) * (180.0 / Math.PI);
        angleOfPoint = (angleOfPoint < 0) ? 360 + angleOfPoint : angleOfPoint; // Normalize angle between 0-360

        // Normalize start and end angles
        startAngle = startAngle % 360;
        endAngle = endAngle % 360;
```

```
            // Check if the point's angle is within the arc's sweep angle.
            if (startAngle < endAngle)
            {
                return startAngle <= angleOfPoint && angleOfPoint <= endAngle;
            }
            else if (startAngle > endAngle)
            {
                // The arc crosses the 0 degree line
                return angleOfPoint >= startAngle || angleOfPoint <= endAngle;
            }
            else // case where startAngle == endAngle, meaning it's a full circle or just a point
            {
                // For a full circle, any point on the circle is on the arc
                // For a point, the angle will match exactly
                return Math.Abs(startAngle - angleOfPoint) < buffer;
            }
        }


// Helper extension method to check if a point is between two other points
public static class Vector2Extensions
{
    public static bool IsBetween(this Vector2 point, Vector2 start, Vector2 end)
    {
        Vector2 startToPoint = point - start;
        Vector2 startToEnd = end - start;

        double dotProduct = (startToPoint.X * startToEnd.X) + (startToPoint.Y * startToEnd.Y);
        double squaredLength = (startToEnd.X * startToEnd.X) + (startToEnd.Y * startToEnd.Y);

        return dotProduct >= 0 && dotProduct <= squaredLength;
    }
}
```

```
/// good!!!
        DxfDocument loaded = DxfDocument.Load("C:\\Users\\Dave\\Downloads\\new block\\new\\arcline.dxf");
        loaded.Entities.ActiveLayout = netDxf.Objects.Layout.ModelSpaceName;
        EntityCollection entitiesBlocks = loaded.Blocks[Block.DefaultModelSpaceName].Entities;

        // Specify the extension amount (e.g., 0.1 for an extension of 0.1 units)
        double extensionAmount = 0.1;

        // Lists to store intersected lines and arcs
        List<EntityObject> intersectedLines = new List<EntityObject>();
        List<EntityObject> intersectedArcs = new List<EntityObject>();

        // Iterate through lines in ModelSpace
        foreach (Line line in entitiesBlocks.OfType<Line>())
        {
            // Extend both ends of the line in 2D
            //Convert the Vector3 to a Vector2 and extend
            Vector2 startPoint2D = new Vector2(line.StartPoint.X, line.StartPoint.Y);
            Vector2 endPoint2D = new Vector2(line.EndPoint.X, line.EndPoint.Y);
            //Not used atm and not needed, a the line projects to infinity
            //startPoint2D = startPoint2D - extensionAmount * new Vector2(line.Direction.X, line.Direction.Y);
            //endPoint2D = endPoint2D + extensionAmount * new Vector2(line.Direction.X, line.Direction.Y);

            // Update the 3D line endpoints while preserving the original Z value
            //Not used atm and not needed, a the line projects to infinity
            ////line.StartPoint = new Vector3(startPoint2D.X, startPoint2D.Y, line.StartPoint.Z);
            ////line.EndPoint = new Vector3(endPoint2D.X, endPoint2D.Y, line.EndPoint.Z);
        }

        // Iterate through arcs in ModelSpace
        foreach (Arc myarc in entitiesBlocks.OfType<Arc>())
        {
            // Extend both ends of the arc
```

```
    //Not used atm, still checking projection via the required estimate of the curve
    //myarc.StartAngle -= extensionAmount;
    //myarc.EndAngle += extensionAmount;
}

// Lists to store unique intersection details
List<string> uniqueIntersections = new List<string>();

// This HashSet will store tuples of line and arc handles that have been printed to avoid duplicates
HashSet<Tuple<string, string>> printedPairs = new HashSet<Tuple<string, string>>();

// Iterate through entities to find intersections
for (int i = 0; i < entitiesBlocks.Count; i++)
{
    for (int j = i + 1; j < entitiesBlocks.Count; j++)
    {
        if (entitiesBlocks[i] is Line line1 && entitiesBlocks[j] is Line line2)
        {
            // Check for intersection between two lines
            Vector2 intersection = MathHelper.FindIntersection(
                line1.StartPoint.ToVector2(),
                line1.Direction.ToVector2(),
                line2.StartPoint.ToVector2(),
                line2.Direction.ToVector2(),
                1e-6);

            if (!double.IsNaN(intersection.X) && !double.IsNaN(intersection.Y))
            {
                // Ensure the intersection is within the bounds of line1
                bool isWithinLine1 = intersection.IsBetween(line1.StartPoint.ToVector2(), line1.EndPoint.ToVector2());

                // Ensure the intersection is within the bounds of line2
                bool isWithinLine2 = intersection.IsBetween(line2.StartPoint.ToVector2(), line2.EndPoint.ToVector2());
```

```
                    // If the intersection is within the bounds of both lines
                    if (isWithinLine1 && isWithinLine2)
                    {
                        // Create a unique intersection identifier
                        string identifier = $"{intersection.X:F3},{intersection.Y:F3}";

                        // Check for uniqueness
                        if (!uniqueIntersections.Contains(identifier))
                        {
                            intersectedLines.Add(line1);
                            intersectedLines.Add(line2);
                            uniqueIntersections.Add(identifier);

                            // Output intersection details for lines
                            Console.WriteLine($"Intersection at ({intersection.X:F3},{intersection.Y:F3}) between Line: {line1.Handle}
and Line: {line2.Handle}");
                        }
                    }
                }
            }
            else if (entitiesBlocks[i] is Arc arc && entitiesBlocks[j] is Line line)
            {
                Vector3 intersection1, intersection2;
                int intersections = FindLineCircleIntersections(
                    new Vector3((float)arc.Center.X, (float)arc.Center.Y, (float)arc.Center.Z),
                    arc.Radius,
                    line.StartPoint,
                    line.EndPoint,
                    out intersection1, out intersection2);

                // Handle the intersections...
                for (int k = 1; k <= intersections; k++)
                {
                    Vector2 intersectionPoint = k == 1 ?
```

```
                        new Vector2(intersection1.X, intersection1.Y) :
                        new Vector2(intersection2.X, intersection2.Y);

                    // Check if intersection is within arc's angles and on the line segment
                    if (IsPointOnArc(intersectionPoint, arc.Center.ToVector2(), arc.Radius, arc.StartAngle, arc.EndAngle) &&
                        intersectionPoint.IsBetween(line.StartPoint.ToVector2(), line.EndPoint.ToVector2()))
                    {
                        // Create a unique intersection identifier
                        string identifier = $"{intersectionPoint.X:F3},{intersectionPoint.Y:F3}";

                        // Check if we have already printed this pair
                        if (!uniqueIntersections.Contains(identifier))
                        {
                            // If not, print the details and add the identifier to the uniqueIntersections
                            Console.WriteLine($"Intersection at ({intersectionPoint.X:F3},{intersectionPoint.Y:F3}) between Line:
{line.Handle} and Arc: {arc.Handle}");
                            uniqueIntersections.Add(identifier);
                        }
                    }
                }
            }
            // ... [Any other intersection checks]
        }
    }

    // Save the modified DXF file
    loaded.Save("C:\\Users\\Dave\\Downloads\\new block\\new\\1a.dxf");
```