Proyecto Final

Código fuente

AirGPS

Model > Converters > CoordinateCircleConverter.cs

```
    class CoordinateCircleConverter
    {
        static readonly double circumference = 40030.174;

        public static Circle CoordinateToCircle(Coordinate coordinate)
        {
            double y = circumference / 360 * Math.Cos(coordinate.Latitude * Math.PI / 180);
            double x = y * coordinate.Longitude;

            return new Circle { X = x, Y = y, R = coordinate.Distance };
        }

        public static Coordinate CircleToCoordinate(Circle circle) => new Coordinate
        {
            Latitude = (180 / Math.PI) * Math.Acos(circle.Y * 360 / circumference),
            Longitude = circle.X / circle.Y,
            Distance = circle.R
        };
    }
```

Model > Genetic > Genetics.cs

```
    class Genetics
    {
        // For random numbers
        static Random Rand;

        // Get the necessary bits for a certain variable
        public static int GetMj(double a, double b, int n)
        {
            if (b - a <= 0 || n <= 0)
                throw new ArgumentException($"El dominio de la variable es inválido: a = {a}, b =
{b}, n = {n}.");

            double r = (Math.Log(b - a) + Math.Log(10) * n) / Math.Log(2);

            int integerPart = Convert.ToInt32(r);

            return r - integerPart <= 0 ? integerPart : integerPart + 1;
        }

        // Map a chromosome part that fits in 64 bits
```

```
        private static double MapValueFast(char[] chromosome, int begin, int end, double a, double
b)
        {
            ulong val = 0;
            ulong vmx = 0;
            for (int i = begin; i < end; ++i)
            {
                val = (val << 1) | ((ulong)chromosome[i] - '0');
                vmx = (vmx << 1) | 1ul;
            }

            return (ulong)((b - a) * val) / (double)vmx + a;
        }

        // Map a chromosome part that doesn't fit in 64 bits
        private static double MapValueSlow(char[] chromosome, int begin, int end, double a, double
b)
        {
            double val = 0;
            double vmx = 0;
            for (int i = begin; i < end; ++i)
            {
                val = val * 2 + (chromosome[i] - '0');
                vmx = vmx * 2 + 1;
            }

            return (b - a) * val / vmx + a;
        }

        // Get the mapped value of a chromosome's variable
        // The variable occupies from chromosome[begin] to chromosome[end - 1]
        public static double MapValue(char[] chromosome, int begin, int end, double a, double b)
        {
            return end - begin > 64 ? MapValueSlow(chromosome, begin, end, a, b) :
MapValueFast(chromosome, begin, end, a, b);
        }

        // Generate a chromosome with n random bits
        public static char[] GenerateRandomChromosome(int n)
        {
            var r = new char[n];

            for (int i = 0; i < n; ++i)
                r[i] = Rand.Next(0, 2).ToString()[0];

            return r;
        }

        // Get the mapped values of a chromosome
        public static double[] GetMappedValues(char[] chromosome, Limit[] limits)
        {
            var mappedValues = new double[limits.Length];
            for (int i = 0, j = 0; i < mappedValues.Length; ++i)
            {
```

```
                mappedValues[i] = MapValue(chromosome, j, j + limits[i].M, limits[i].A,
limits[i].B);
                j += limits[i].M;
            }

            return mappedValues;
        }

        // Check if a chromosome is valid
        public static bool CheckChromosome(char[] chromosome, Limit[] limits, Func<double, double,
bool>[] restrictions)
        {
            var mappedValues = GetMappedValues(chromosome, limits);

            for (int i = 0; i < restrictions.Length; ++i)
                if (!restrictions[i](mappedValues[0], mappedValues[1]))
                    return false;

            return true;
        }

        // Generate a valid chromosome
        public static char[] GenerateChromosome(Limit[] limits, Func<double, double, bool>[]
restrictions, CancellationToken timer)
        {
            char[] chromosome = null;
            bool stay = true;

            while (stay && !timer.IsCancellationRequested)
            {
                int n = 0;
                foreach (var l in limits) n += l.M;

                chromosome = GenerateRandomChromosome(n);
                stay = !CheckChromosome(chromosome, limits, restrictions);
            }

            if (timer.IsCancellationRequested)
                throw new TimeoutException();

            return chromosome;
        }

        // Generate a population of n valid chromosomes
        public static char[][] GeneratePopulation(Limit[] limits, Func<double, double, bool>[]
restrictions, int m, CancellationToken timer)
        {
            char[][] population = new char[m][];

            for (int i = 0; i < m; ++i)
                population[i] = GenerateChromosome(limits, restrictions, timer);

            return population;
        }
```

```
        // Mutate a random chromosome's gen and return a new one
        public static char[] Mutate(char[] chromosome)
        {
            var r = new char[chromosome.Length];

            for (int i = 0; i < r.Length; ++i)
                r[i] = chromosome[i];

            int index = Rand.Next(0, r.Length);
            r[index] = r[index] == '1' ? '0' : '1';

            return r;
        }

        // Cross two chromosome in a random position
        public static char[] Cross(char[] parent1, char[] parent2)
        {
            int chromosomeSize = parent1.Length;
            var child = new char[chromosomeSize];
            int n = Rand.Next(0, chromosomeSize);

            for (int i = 0; i < n; ++i)
                child[i] = parent1[i];

            for (int i = n; i < chromosomeSize; ++i)
                child[i] = parent2[i];

            return child;
        }

        // Calculate the z values associated with each chromosome and return an array with them and
its sum
        public static (double[], double) CalculateZValues(char[][] population, Limit[] limits)
        {
            double total = 0;
            var values = new double[population.Length];

            for (int i = 0; i < values.Length; ++i)
            {
                var mappedValues = GetMappedValues(population[i], limits);
                values[i] = -Restriction.Z(mappedValues[0], mappedValues[1]);
                total += values[i];
            }

            return (values, total);
        }

        // Calculate the percentages associated and the accumulates with each z value and return
them
        public static (double[], double[]) CalculatePercentages(double[] values, double total)
        {
            var percentages = new double[values.Length];
            var accumulates = new double[values.Length];
            double accumulate = 0.0;
```

```
            for (int i = 0; i < values.Length; ++i)
            {
                percentages[i] = values[i] / total;
                accumulate += percentages[i];
                accumulates[i] = accumulate;
            }

            return (percentages, accumulates);
        }

        // Get the best chromosomes of a population
        public static char[][] GetTheBest(char[][] population, double[] values, double[]
accumulates)
        {
            var best = new PriorityQueue();
            for (int i = 0; i < values.Length; ++i)
            {
                double r = Rand.NextDouble();
                for (int j = 0; j < values.Length; ++j)
                {
                    if (r < accumulates[j])
                    {
                        best.Push(population[j], values[j]);
                        break;
                    }
                }
            }

            return best.GetOnlyValues();
        }

        // Genetic round
        public static char[][] Round(char[][] population, Limit[] limits)
        {
            var (values, total) = CalculateZValues(population, limits);
            var (percentages, accumulates) = CalculatePercentages(values, total);

            return GetTheBest(population, percentages, accumulates);
        }

        // Regenerate the given population with best chromosomes, and mutation and crossover of
best chromosomes
        public static void RegeneratePopulation(char[][] population, char[][] best, Limit[] limits,
Func<double, double, bool>[] restrictions, CancellationToken timer)
        {
            int i;
            for (i = 0; i < best.Length; ++i)
                population[i] = best[i];

            for (int j = i; j < population.Length; ++j)
            {
                while (!timer.IsCancellationRequested)
                {
                    population[j] = Rand.Next(0, 2) == 0 ? Mutate(best[Rand.Next(0, i)]) :
Cross(best[Rand.Next(0, i)], best[Rand.Next(0, i)]);
```

```
                    if (CheckChromosome(population[j], limits, restrictions))
                        break;
                }

                if (timer.IsCancellationRequested)
                    throw new TimeoutException();
            }

        }

        // Genetic Algorithm
        public static (int, double, double, double) Calculate(Circle[] circles, int n, int rounds,
int size, double e, bool rel, Action<(int, double, double, double)> loggerTuple, Action<string>
logger,CancellationToken timer)
        {
            Rand = new Random();

            var answer = (0, 0.0, 0.0, 0.0);

            Restriction.InitializeZ(circles);

            double error = Restriction.CalculateError(e, circles.Length, rel);
            logger($"Usando error: {error}");
            var restrictions = Restriction.Generate(circles, error);
            var limits = Limit.Generate(circles, n, error, logger);


            logger($"Generando población de {size} individuos...");
            var population = GeneratePopulation(limits, restrictions, size, timer);
            logger($"Población generada de {size} individuos...");

            for (int i = 0; i < rounds && i < 100; ++i)
            {
                var best = Round(population, limits);
                RegeneratePopulation(population, best, limits, restrictions, timer);

                var values = GetMappedValues(population[0], limits);

                answer = (i, values[0], values[1], Restriction.Z(values[0], values[1]));
                loggerTuple(answer);
            }

            return answer;
        }
    }
```

Model > Genetic > Limit.cs

```
    class Limit
    {
        public double A { set; get; }
        public double B { set; get; }
        private int m;
        public int M
        {
            get => m;
            set { m = value < 0 ? -value : value; }
        }

        // Generar limites
        public static Limit[] Generate(Circle[] circles, int n, double error, Action<string>
logger)
        {
            try
            {
                // Hot fix
                var original = new double[circles.Length];

                for (int i = 0; i < circles.Length; ++i) {
                    original[i] = circles[i].R;
                    circles[i].R = Math.Sqrt(Help.Square(circles[i].R) + error);
                }

                var commonArea = Circle.GetIntersectionArea(new List<Circle>(circles));

                for (int i = 0; i < circles.Length; ++i)
                    circles[i].R = original[i];

                // ---

                double ax = commonArea.LeftDown.X;
                double ay = commonArea.LeftDown.Y;

                double bx = commonArea.RightUp.X;
                double by = commonArea.RightUp.Y;

                logger($"Límites en X: ({ax}, {bx}).");
                logger($"Límites en Y: ({ay}, {by}).");

                int mx = Genetics.GetMj(ax, bx, n);
                int my = Genetics.GetMj(ay, by, n);

                Limit[] limits = {
                    new Limit() { A = ax, B = bx, M = mx },
                    new Limit() { A = ay, B = by, M = my }
                };

                return limits;
            }
            catch (Exception e)
```

```
            {
                logger(e.Message);
                throw e;
            }
        }
    }
```

Model > Genetic > Restriction.cs

```
    class Restriction
    {
        // Represents the objective function
        public static Func<double, double, double> Z = null;

        // Create a new objetive function
        public static void InitializeZ(Circle[] circles)
        {
            Z = (double x, double y) => {
                double result = 0;

                foreach(var circle in circles)
                    result += Help.Square(Help.Square(x - circle.X) + Help.Square(y - circle.Y) -
Help.Square(circle.R));

                return result;
            };
        }

        // Create a new restriction
        static Func<double, double, bool> CreateCircleRestriction(Circle circle, double error) =>
            (double x, double y) => Help.Square(x - circle.X) + Help.Square(y - circle.Y) -
Help.Square(circle.R) <= error;

        // Error calculation (Modified to meters)
        public static double AbsoluteError(double e, int numOfRestrictions) => Math.Sqrt(Z(0, 0)) *
e / (numOfRestrictions * 10E+6);
        public static double RelativeError(double e, int numOfRestrictions) => Z(0, 0) * e /
(numOfRestrictions * 10E+6);
        public static double CalculateError(double e, int length,bool rel) => rel ?
RelativeError(e, length) : AbsoluteError(e, length);

        // Generate all restrictions
        public static Func<double, double, bool>[] Generate(Circle[] circles, double error)
        {
            var restrictions = new Func<double, double, bool>[circles.Length];
            for (int i = 0; i < restrictions.Length; ++i)
            {
                restrictions[i] = CreateCircleRestriction(circles[i], error);
            }
            return restrictions;
        }
    }
```

Model > Data > Area.cs

```
public class Area
{
    public Point LeftDown { get; set; }
    public Point RightUp { get; set; }

    /// Static methods

    // Calculate a rectangular area that enclose all the points
    public static Area CalculateArea(List<Point> points)
    {
        double minX = points[0].X;
        double maxX = points[0].X;

        double minY = points[0].Y;
        double maxY = points[0].Y;

        foreach (var p in points)
        {
            if (minX > p.X)
                minX = p.X;

            if (maxX < p.X)
                maxX = p.X;

            if (minY > p.Y)
                minY = p.Y;

            if (maxY < p.Y)
                maxY = p.Y;
        }

        return new Area
        {
            LeftDown = new Point(minX, minY),
            RightUp = new Point(maxX, maxY)
        };
    }

    // Get the limits of the intersection in an axis
    // Throws Exception if there are not intersection
    static (double, double) GetLimitsOfAxis(List<Area> areas, string axis)
    {
        var end = axis == "X" ? areas[0].RightUp.X : areas[0].RightUp.Y;
        for (int i = 1; i < areas.Count; ++i)
        {
            var aux = axis == "X" ? areas[i].RightUp.X : areas[i].RightUp.Y;
            if (aux < end)
                end = aux;
        }

        var begin = axis == "X" ? areas[0].LeftDown.X : areas[0].LeftDown.Y;
        for (int i = 1; i < areas.Count; ++i)
```

```
            {
                var aux = axis == "X" ? areas[i].LeftDown.X : areas[i].LeftDown.Y;
                if (aux > begin)
                    begin = aux;
            }

            if (begin >= end)
                throw new Exception($"No existe una intersección común en las áreas a lo largo de
{axis}");

            return (begin, end);
        }

        // Get the intersection of all areas
        // Throw Exception if there are not intersection
        public static Area GetIntersectionOfAllAreas(List<Area> areas)
        {
            var (xBegin, xEnd) = GetLimitsOfAxis(areas, "X");
            var (yBegin, yEnd) = GetLimitsOfAxis(areas, "Y");

            return new Area
            {
                LeftDown = new Point(xBegin, yBegin),
                RightUp = new Point(xEnd, yEnd)
            };
        }
    }
```

Model > Data > Circle.cs

```
    public class Circle
    {
        // Properties
        public Point Center { get; set; }
        public double X
        {
            get => Center.X;
            set { Center.X = value; }
        }
        public double Y
        {
            get => Center.Y;
            set { Center.Y = value; }
        }
        private double r;
        public double R
        {
            get => r;
            set { r = value < 0 ? -value : value; }
        }

        // Constructors
        public Circle()
```

```
        {
            Center = new Point(0, 0);
        }

        // Check if the given points are contained in the circle
        public bool ContainsAll(List<Point> points)
        {
            foreach (Point p in points)
                if (!Contains(p))
                    return false;

            return true;
        }

        // Check if the point is contained in the circle
        public bool Contains(Point p) => Help.Square(p.X - X) + Help.Square(p.Y - Y) <=
Help.Square(R);

        public List<Point> GetCardinalPoints()
        {
            return new List<Point>
            {
                new Point(X, Y + R),
                new Point(X, Y - R),
                new Point(X + R, Y),
                new Point(X - R, Y)
            };
        }

        /// Static methods

        // Computes the distance between two circles' centers
        public static double DistanceBetweenTwoCenters(Circle c1, Circle c2)
            => Point.DistanceBetweenTwoPoints(c1.X, c1.Y, c2.X, c2.Y);

        // Check if inner circle is enclosed in outer circle
        static bool IsEnclosedIn(Circle ic, Circle oc) => oc.R >=
Circle.DistanceBetweenTwoCenters(ic, oc) + ic.R;

        // Remove all enclosing circles
        static List<Circle> RemoveEnclosingCircles(List<Circle> circles)
        {
            // De-duplicate list and order it by radius
            var orderedList = new List<Circle>(new HashSet<Circle>(circles, new
CircleHashComparer())).OrderBy(circle => circle.R).ToList();
            var smallest = orderedList[0];
            orderedList.RemoveAt(0);

            // Quit all circles enclosing smallest
            var r = orderedList.Where(circle => !IsEnclosedIn(smallest, circle)).ToList();
            r.Insert(0, smallest);

            return r;
        }
```

```
        // Get the intersection points between a line and a circle
        // Throws Exception if line doesn't insersect with circle
        static List<Point> IntersectionLineCircle(Line line, Circle circle)
        {
            double cprime = line.C - line.A * circle.X - line.B * circle.Y;

            var aa = Help.Square(line.A) + Help.Square(line.B);
            Func<double, double> bb = n => -2 * n * cprime;
            Func<double, double> cc = n => cprime * cprime - Help.Square(circle.R * n);

            var (e1, e2) = Help.QuadraticEquation(aa, bb(line.A), cc(line.B));
            var (n1, n2) = Help.QuadraticEquation(aa, bb(line.B), cc(line.A));

            var x1 = e1 + circle.X;
            var x2 = e2 + circle.X;
            var y1 = n1 + circle.Y;
            var y2 = n2 + circle.Y;

            if (x1 == x2 && y1 == y2)
                return new List<Point> { new Point(x1, y2) };

            return new List<Point>
            {
                new Point(x1, y1),
                new Point(x2, y2)
            };
        }

        // Get the intersection point(s) between two circles
        // Throws Exception if the circles don't intersect themselves
        static List<Point> IntersectionCircleCircle(Circle c1, Circle c2)
        {
            var line = new Line
            {
                A = 2 * (c2.X - c1.X),
                B = 2 * (c2.Y - c1.Y),
                C = (Help.Square(c1.R) - Help.Square(c2.R)) - (Help.Square(c1.X) -
Help.Square(c2.X)) - (Help.Square(c1.Y) - Help.Square(c2.Y))
            };

            return IntersectionLineCircle(line, c1);
        }

        // Calculate the insersection area of two circles
        // Throws ArgumentOutOfRangeException if the circles don't intersect themselves
        static Area GetAreaFromCircles(Circle c1, Circle c2)
        {
            try
            {
                var points = new List<Point>();

                points.AddRange(c1.GetCardinalPoints().Where(point =>
c2.Contains(point)).ToList());
                points.AddRange(c2.GetCardinalPoints().Where(point =>
c1.Contains(point)).ToList());
```

```
                points.AddRange(IntersectionCircleCircle(c1, c2));

                return Area.CalculateArea(points);
            }
            catch (Exception)
            {
                throw new ArgumentOutOfRangeException($"Los círculos: {{{{{c1.X}, {c1.Y}, {c1.R}}}, {{{c2.X}, {c2.Y}, {c2.R}}}}} no se intersectan en ningún punto.");
            }
        }

        // Get all intersections areas of the circles
        // Throws ArgumentOutOfRangeException if any pair of circles don't intersect themselves
        static List<Area> GetIntersectionAreas(List<Circle> circles)
        {
            var areas = new List<Area>();

            for (int i = 0; i < circles.Count; ++i)
                for (int j = i + 1; j < circles.Count; ++j)
                    areas.Add(GetAreaFromCircles(circles[i], circles[j]));

            return areas;
        }

        // Get the rectangle intersection area of circles
        // Throw Exception if a pair of circles don't intersect or doesn't exists an area of intersection
        public static Area GetIntersectionArea(List<Circle> circles)
        {
            Point leftDown, rightUp;

            var noEnclosingCircles = RemoveEnclosingCircles(circles);

            if (noEnclosingCircles.Count == 1)
            {
                var final = noEnclosingCircles[0];

                leftDown = new Point(final.X - final.R, final.Y - final.R);
                rightUp = new Point(final.X + final.R, final.Y + final.R);

                return new Area { LeftDown = leftDown, RightUp = rightUp };
            }

            var areas = GetIntersectionAreas(noEnclosingCircles);

            return Area.GetIntersectionOfAllAreas(areas);
        }

    }
```

Model > Data > Coordinate.cs

```
    public class Coordinate
    {
        public double Latitude { get; set; }
        public double Longitude { get; set; }

        private double distance;
        public double Distance
        {
            get => distance;
            set
            {
                distance = value < 0.0 ? -value : value;
            }
        }

        public Coordinate() { Latitude = 0; Longitude = 0; Distance = 0; }
        public Coordinate(double latitude, double longitude, double distance) { Latitude =
latitude; Longitude = longitude; Distance = distance; }
    }
```

Model > Data > Line.cs

```
    class Line
    {
        public double A { get; set; }
        public double B { get; set; }
        public double C { get; set; }
    }
```

Model > Data > Point.cs

```
    public class Point
    {
        public double X { get; set; }
        public double Y { get; set; }

        public Point(double x, double y) { X = x; Y = y; }

        public static double DistanceBetweenTwoPoints(double px, double py, double qx, double qy)
            => Math.Sqrt(Help.Square(px - qx) + Help.Square(py - qy));

        public static double DistanceBetweenTwoPoints(Point p, Point q)
            => Math.Sqrt(Help.Square(p.X - q.X) + Help.Square(p.Y - q.Y));
    }
```

Model > Helpers > Help.cs

```
class Help
{
    // Simplification of a * a
    public static double Square(double a) => a * a;

    // Resolves ax^2 + bx + c = 0
    public static (double, double) QuadraticEquation(double a, double b, double c)
    {
        double discriminant = b * b - 4 * a * c;
        if (discriminante < 0)
            throw new Exception("El discriminante no es positivo.");

        double x1 = ((-b) + Math.Sqrt(discriminant)) / (2 * a);
        double x2 = ((-b) - Math.Sqrt(discriminant)) / (2 * a);

        return (x1, x2);
    }
}
```

Model > Helpers > Pair.cs

```
public struct Pair
{
    public char[] Item { get; set; }
    public int Repetitions { get; set; }
    public double Weight { get; set; }

    public bool EqualsTo(char[] item)
    {
        if (Item.Length != item.Length) return false;

        for (int i = 0; i < item.Length; ++i)
            if (Item[i] != item[i])
                return false;

        return true;
    }
}
```

Model > Helpers > PriorityQueue.cs

```
public class PriorityQueue
{
    public List<Pair> queue;

    // Get each Pair.Item of each queue element and return them
    public char[][] GetOnlyValues()
    {
        char[][] result = new char[queue.Count][];
```

```
            for (int i = 0; i < result.Length; ++i)
                result[i] = queue[i].Item;

            return result;
        }

        // Constructor
        public PriorityQueue()
        {
            queue = new List<Pair>();
        }

        // Push an element, ordering first by repetitions, then by weight
        public void Push(char[] item, double weight)
        {
            // First element to be inserted
            if (queue.Count == 0)
            {
                queue.Add(new Pair { Item = item, Repetitions = 1, Weight = weight});
                return;
            }

            // Search the element in the queue
            int i = SearchFor(item);

            // If exists, update the repetitions or add a new element if not
            if (i >= 0)
                queue[i] = new Pair { Item = queue[i].Item, Repetitions = queue[i].Repetitions + 1,
Weight = queue[i].Weight };
            else
            {
                queue.Add(new Pair { Item = item, Repetitions = 1, Weight = weight });
                i = queue.Count - 1;
            }

            Shift(i);
        }

        // Search for the item inside the queue and returns its index
        // Return -1 if not
        public int SearchFor(char[] item) => queue.FindIndex(e => e.EqualsTo(item));

        // Shift the queue starting from i
        void Shift(int i)
        {
            for (int j = i - 1; j >= 0; --j)
            {
                if (queue[i].Repetitions < queue[j].Repetitions)
                    break;

                if (queue[i].Repetitions > queue[j].Repetitions)
                {
                    Swap(i, j);
                    i = j;
                    continue;
```

```
            }

            if (queue[i].Weight <= queue[j].Weight)
                continue;

            Swap(i, j);
            i = j;
        }
    }

    // Swap i with j elements in the queue
    void Swap(int i, int j)
    {
        Pair aux = queue[i];
        queue[i] = queue[j];
        queue[j] = aux;
    }
}
```

Integrantes:

- ➔ Joel Harim Hernandez Javier (3CM8)
- ➔ Eli Sanchez Martinez (3CM8)
- ➔ Brenda Mariana Hernández Morales (3CM7)
- ➔ Diana Laura Jiménez López (3CM7)

El código completo se puede encontrar en Github:
https://github.com/JoelHernandez343/TrilateracionGpsGeneticos