
Ministry of Education and Science of the Russian Federation
Peter the Great St. Petersburg State Polytechnic University
Institute of Computer Science and Control Systems
**Control Systems and Technology Department**

# Report for Laboratory 2

Gradient Descent Minimization Methods
Discipline: Methods of Optimization
6 December 2016

Student Group: 13541/8

Christopher W. Blake

Professor

Rodionova E.A.

St. Petersburg
2016

Christopher W. Blake


# Contents

Christopher W. Blake


# Introduction

The following objective function is provided, and a search algorithm should be used for locating the point of minimization. To determine this minimization point, both first-order and second-order gradient descent versions are utilized and compared. Additionally, different methods are used to analyze the step size. These include golden ratio search and division search for the first-order gradient descent, and traditional newton's method and dampened newton's method for the second-order gradient descent.

**Objective Function:**

$$f(x_1, x_2) = {x_1}^2 + {x_2}^2 + e^{{x_2}^2} - x_1 + 2x_2$$

**Methods**

1. First-Order Gradient Descent
    a. Golden Ratio Search
    b. Division Search
2. Second-Order Gradient Descent
    a. Traditional Newton's Method
    b. Dampened Newton's Method (Division Method)

# Convexity

Before such search methods can be performed, the objective function must be tested for convexity. The simplest method for a complex equation such as this is to check the first and second order gradient. As seen below, the second order gradient below is always positive, hence the objective function is convex.

| | Respect to $x_1$ | Respect to $x_2$ |
|---|---|---|
| First-Order Gradient: | $\nabla f(x_1) = 2x - 1$ | $\nabla f(x_2) = 2x_2 + 2x_2 e^{{x_2}^2}$ |
| Second-Order Gradient: | $\nabla^2 f(x_1) = 2$ | $\nabla^2 f(x_2) = 2 + 4{x_2}^2 e^{{x_2}^2} + 2e^{{x_2}^2}$ |

$$\nabla^2 f(x) \succ 0 \quad \text{therefore} \quad f(x) \quad \text{is convex}$$

# Method Descriptions

## First Order Gradient Descent

**Description**

The gradient of a multi-variable objective function is calculated. This gradient is used as the direction to be minimized. Next, the minimization point in this direction is calculated using either the golden radio search or the division search method. At this new position, the next direction is determined and the process is repeated. This process is repeated until the gradient is smaller than the required accuracy.

**Program Outline**

1.) Specify inputs
    a. f(x) – Objective Function
    b. eps – Accuracy Value
2.) Calculate f(x) at a first assumed point.
3.) Determine the gradient: ∇f(x).
4.) Compute the scaling factor (α) that reaches the minimum point in the gradient direction. This is performed using one of the below methods.

a. Golden Ratio Search
    i. Build a temporary objective function using the current position ($X_n$) and the gradient $\nabla f(x)$.
        1. $X_{n+1} = X_n + \alpha\nabla f(x)$
    ii. Use the golden ratio search to determine the scaling factor ($\alpha$) that produces the minimization point.
b. Division Method
    i. Begin with a default scaling factor ($\alpha$) equal to 0.8 of the previous scaling factor ($\alpha$). Establish a temporary scaling factor ($\delta$) that is equal to the current scaling factor ($\alpha$).
    ii. Compute the new position
        1. $X_{n+1} = X_n + \delta\nabla f(x)$
    iii. Check if $f(X_{n+1})$ is lower than $f(X_n)$. If it is not, divide the temporary scaling factor ($\delta$) by 2 and repeat.

5.) Move to the new position using the gradient $\nabla f(x)$ and scaling factor ($\alpha$).
    a. $X_{n+1} = X_n + \alpha\nabla f(x)$.
6.) Check if the gradient is smaller than the accuracy (eps.) If so, end the search and continue to step 7. If not, repeat steps 2 through 5.
7.) Return the results
    a. Minimization Point: x
    b. Final calculations: f(x)
    c. Number of calculations of f(x) required to reach the answer.
    d. Number of calculations of $\nabla f(x)$ required to reach the answer.

## Second-Order Gradient Descent

**Description**

The gradient and second gradient of a multi-variable objective function are calculated. These gradients are used in combination to determine the minimization direction and distance. Two versions of this method are performed. (See derivation below.)

1. **Full Step** – (Traditional Newtons Method) An alpha value of 1 is used. No additional computations for step are required.
2. **Division Method** – (Dampened Newton's Method) – A dampening factor (alpha) is applied, verifying that the full step does not overshoot.

At this new position, the next direction and distance are determined and the process is repeated. This process is repeated until the gradient is smaller than the required accuracy.

**Derivation**

1.) Equation for next f(x) – The location of the minimization point can be written as such:

$$f(x[i+1]) = f(x[i]) + f'(x[i])\Delta x + \frac{1}{2}f''(x[i])\Delta x^2$$

2.) First derivative - The derivative of this equation is set equal to zero to find the point of minimization.

$$0 = f'(x[i+1]) = f'(x[i]) + f''(x[i])\Delta x$$

3.) Distance to min point – The first derivative is rearranged to determine the step ($\Delta x$).

$$\Delta x = \frac{-f'(x[i])}{f''(x[i])}$$

4.) Minimization point – This distance is combined with the current position to determine the minimization point.

$$x[i+1] = x[n] + \frac{-f'(x[i])}{f''(x[i])}$$

**Program Outline**

1.) Specify inputs
   a. f(x) – Objective Function
   b. eps – Accuracy Value
2.) Calculate f(x) at a first assumed point.
3.) Determine the gradient, $\nabla f(x)$, and second gradient, $\nabla^2 f(x)$.
4.) Compute the full step for traditional newtons method (full step) scaling factor (α) directions that reaches the minimum.
   a. Compute the new position
      i. fullStep = -f'(x1) / f''(x1)
      ii. α = 1
5.) If using the dampened newtons method, determine the scaling factor (α) directions that reaches the minimum. If not, skip to step 6. The division method is used.
   a. Begin with a default scaling factor (α) equal to 0.8 of the previous scaling factor (α). Establish a temporary scaling factor (δ) that is equal to the current scaling factor (α).
   b. Compute the new position
      i. $X_{n+1} = X_n + \delta * \text{fullStep}$
   c. Check if $f(X_{n+1})$ is lower than $f(X_n)$. If it is not, divide the temporary scaling factor (δ) by 2 and repeat.
6.) Move to the new position using step size (α).
   a. $X_{n+1} = X_n + \alpha * \text{fullStep}$
7.) Check if the gradient is smaller than the accuracy (eps.) If so, end the search and continue to step 7. If not, repeat steps 2 through 5.
8.) Return the results
   a. Minimization Point: [x1, x2]
   b. Final calculations: f(x1, x2)
   c. Number of calculations of f(x) required to reach the answer.
   d. Number of calculations of $\nabla f(x)$ required to reach the answer
   e. Number of calculations of $\nabla^2 f(x)$ required to reach the answer.

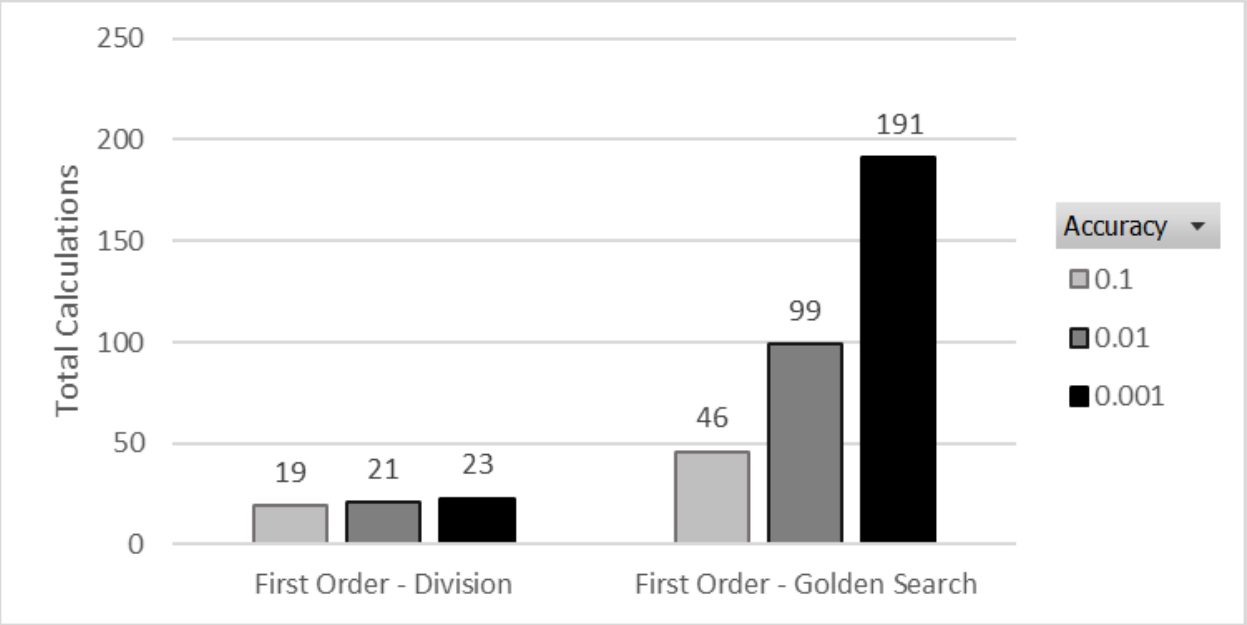
Christopher W. Blake


## Results

The results for each method is shown in the following sections. Each method was performed with different accuracy of 0.1, 0.01, and 0.001. The total number of required calculations of f(x), f'(x), and f''(x) to find the solution is used as the main comparison.

### First-Order Gradient Descent

As previously stated, the first-order gradient descent method was analyzed with both the division and the golden search method for determining the step. It can be seen in the below charts that the division method outperforms the golden search method for all accuracy requirements. Additionally, the number of required calculations is not drastically affected.

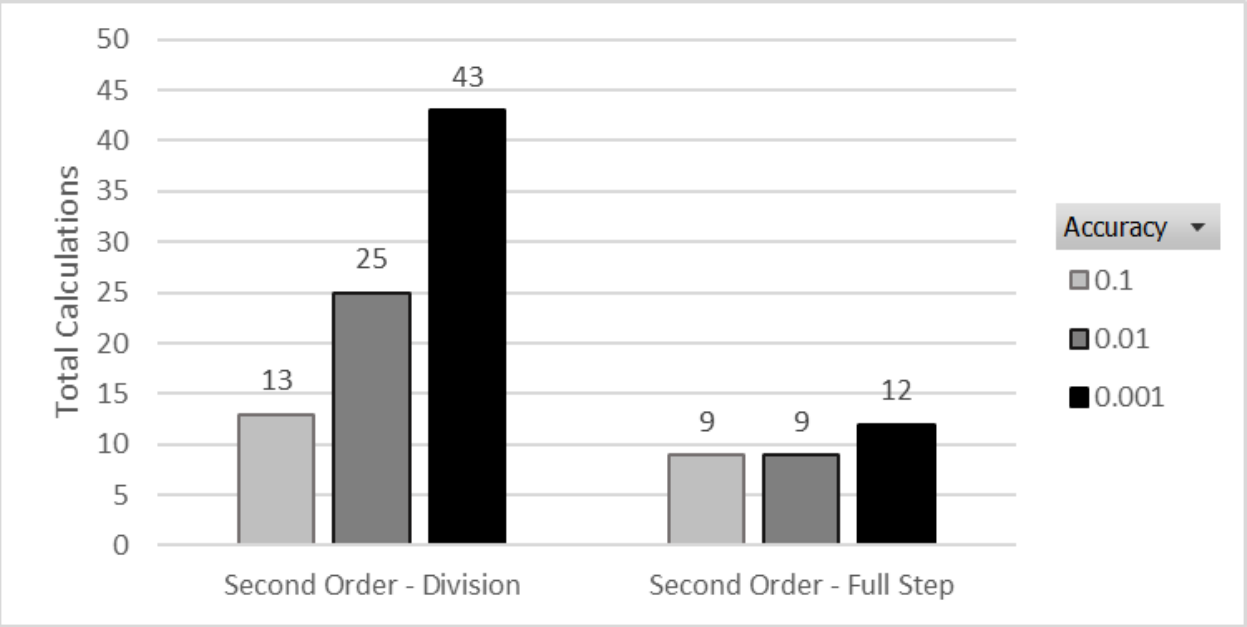



| Method | Accuracy | X1 | X2 | f(x) | Calcs f(x) | Calcs f'(x) | Calcs f''(x) | Total Calcs |
|---|---|---|---|---|---|---|---|---|
| First Order – Division | 0.1 | 0.5 | -0.45 | 0.28 | 12 | 7 | | 19 |
| First Order – Division | 0.01 | 0.50 | -0.450 | 0.277 | 13 | 8 | | 21 |
| First Order – Division | 0.001 | 0.4997 | -0.4497 | 0.2770 | 14 | 9 | | 23 |
| First Order - Golden Search | 0.1 | 0.51 | -0.48 | 0.28 | 41 | 5 | | 46 |
| First Order - Golden Search | 0.01 | 0.502 | -0.450 | 0.277 | 92 | 7 | | 99 |
| First Order - Golden Search | 0.001 | 0.5001 | -0.4495 | 0.2770 | 181 | 10 | | 191 |

Christopher W. Blake


## Second-Order Gradient Descent

The second-order gradient descent method was calculated using only the division method. It can be seen below that the total calculations was also not drastically affect by the increased accuracy.




| Method | Accuracy | X1 | X2 | f(x) | Calcs f(x) | Calcs f'(x) | Calcs f''(x) | Total Calcs |
|---|---|---|---|---|---|---|---|---|
| Second Order – Division | 0.1 | 0.49 | -0.44 | 0.28 | 5 | 4 | 4 | 13 |
| Second Order – Division | 0.01 | 0.497 | -0.448 | 0.277 | 9 | 8 | 8 | 25 |
| Second Order – Division | 0.001 | 0.4990 | -0.4497 | 0.2770 | 15 | 14 | 14 | 43 |
| Second Order – Full Step | 0.1 | 0.50 | -0.45 | 0.28 | 3 | 3 | 4 | 9 |
| Second Order – Full Step | 0.01 | 0.500 | -0.450 | 0.277 | 3 | 3 | 4 | 9 |
| Second Order – Full Step | 0.001 | 0.5000 | -0.4496 | 0.2770 | 4 | 4 | 4 | 12 |

Christopher W. Blake

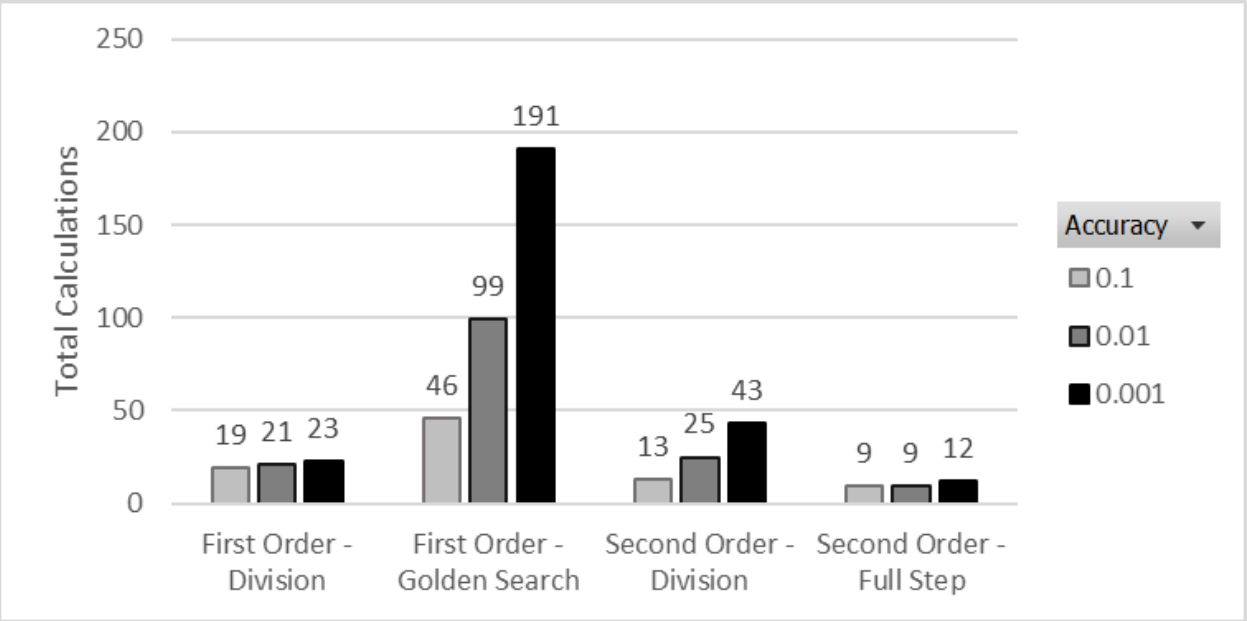

## Conclusion

From the results, it can be seen that the minimum point of the objective function is at f(0.5, -0.45) = 0.277.

**Conclusion 1:** From the various methods, the most effective solution is the full step second order gradient descent method. This method, required the least number of calculations of f(x), f'(x), and f''(x) in order to find the solution. However, it does require that a function to be second-order differentiable.

**Conclusion 2:** Looking only at the first-order gradient descent data, it can be seen that the division method significantly outperforms the golden ratio search. This is likely because the golden ratio search is more accurate per cycle than required.

Below is a chart and table comparing the various methods. The highlighted value is the cell with the least number of calculations for a given accuracy requirement.




| Method | Accuracy | X1 | X2 | f(x) | Calcs f(x) | Calcs f'(x) | Calcs f''(x) | Total Calcs |
|---|---|---|---|---|---|---|---|---|
| First Order - Golden Search | 0.1 | 0.51 | -0.48 | 0.28 | 41 | 5 | | 46 |
| First Order - Division | 0.1 | 0.50 | -0.45 | 0.28 | 12 | 7 | | 19 |
| Second Order - Full Step | 0.1 | 0.50 | -0.45 | 0.28 | 3 | 3 | 3 | 9 |
| Second Order - Division | 0.1 | 0.49 | -0.44 | 0.28 | 5 | 4 | 4 | 13 |
| First Order - Golden Search | 0.01 | 0.502 | -0.450 | 0.277 | 92 | 7 | | 99 |
| First Order - Division | 0.01 | 0.500 | -0.450 | 0.277 | 13 | 8 | | 21 |
| Second Order - Full Step | 0.01 | 0.500 | -0.450 | 0.277 | 3 | 3 | 3 | 9 |
| Second Order - Division | 0.01 | 0.497 | -0.448 | 0.277 | 9 | 8 | 8 | 25 |
| First Order - Golden Search | 0.001 | 0.5001 | -0.4494 | 0.2770 | 181 | 10 | | 191 |
| First Order - Division | 0.001 | 0.4997 | -0.4497 | 0.2770 | 14 | 9 | | 23 |
| Second Order - Full Step | 0.001 | 0.5000 | -0.4496 | 0.2770 | 4 | 4 | 4 | 12 |
| Second Order - Division | 0.001 | 0.4990 | -0.4497 | 0.2770 | 15 | 14 | 14 | 43 |

# Appendix A: C# Program

## Main

```
        static void Main(string[] args)
        {
            #region Intro
            Console.WriteLine(@"
//////////////////////////////////////
Task: Lab 2 - Gradient Descent
Written By: Christopher W. Blake
Date: 22 Nov. 2016
//////////////////////////////////////
Description:
Creates 2 different seach algorithms for finding
the minimum of a given function f(x) in a range. It then tests
these results and prints them to the console.

1. First-order gradient descent method.
   (one – dimensional minimization method for choosing the step)

2. Second-order gradient descent method
   (division method for choosing the step)

Verification: f(0.5, -0.44963) = 0.27696
//////////////////////////////////////
");
            #endregion

            //Required accuracy values
            List<double> epsValues = new List<double> { 0.1, 0.01, 0.001 }; //accuracy

            //Objective function
            Func<DV,D> f = delegate (DV x)
            {
                D x1 = x[0];
                D x2 = x[1];
                return x1 * x1 + x2 * x2 + AD.Exp(x2 * x2) - x1 + 2 * x2;
                //return AD.Pow(x1-7, 2) + AD.Pow(x2-3, 2);
            };

            #region 1.)  First Order, One-Dimensional Method
            //Show the table header
            Console.WriteLine("----- Gradient Search, First Order, One-Dimensional Method -----");
            Console.WriteLine("     eps      X1      X2    f(x)   Calcs F   Calcs Gr");
            foreach (double eps in epsValues)
            {
                //Get solution
                int calcsF;
                int calcsGradient;
                DV startPoint = new DV(new D[] {0,0});
                DV xMin = Optimization.GradientDescent.FirstOrder_OneDimensionalMethod(f, startPoint,
eps, out calcsF, out calcsGradient);

                //determine number of decimal places to show
                int dp = BitConverter.GetBytes(decimal.GetBits((decimal)eps)[3])[2] + 1;

                //Show on console
                Console.WriteLine("{0,8}{1,8:F"+dp+"}{2,8:F" + dp + "}{3,8:F" + dp + "}{4,8}{5,8}",
eps, (double)xMin[0], (double) xMin[1], (double) f(xMin), calcsF, calcsGradient);

            }
            #endregion

            #region 2.) First Order, Division Method
            //Show the table header
            Console.WriteLine();
            Console.WriteLine("----- Gradient Search, First Order, Division Method -----");
            Console.WriteLine("     eps      X1      X2    f(x)   Calcs F   Calcs dF");
            foreach (double eps in epsValues)
            {
                //Get solution
                int calcsF;
                int calcsGradient;
                DV startPoint = new DV(new D[] { 0, 0 });
```

```
                DV xMin = Optimization.GradientDescent.FirstOrder_DivisionMethod(f, startPoint, eps,
out calcsF, out calcsGradient);

                //determine number of decimal places to show
                int dp = BitConverter.GetBytes(decimal.GetBits((decimal)eps)[3])[2] + 1;

                //Show on console
                Console.WriteLine("{0,8}{1,8:F" + dp + "}{2,8:F" + dp + "}{3,8:F" + dp + "}{4,8}{5,8}",
eps, (double)xMin[0], (double)xMin[1], (double)f(xMin), calcsF, calcsGradient);

            }
            #endregion

            #region 3.) Second Order - Newtons Method, FullStep
            //Show the table header
            Console.WriteLine();
            Console.WriteLine("----- Gradient Search, Second Order, FullStep -----");
            Console.WriteLine("     eps      X1      X2    f(x)   Calcs F  Calcs Gr  Calcs Hess");

            //Show Results for each accuracy
            foreach (double eps in epsValues)
            {
                //Get solution
                int calcsF;
                int calcsGradient;
                int calcsHessian;
                DV startPoint = new DV(new D[] {0, 0});
                DV xMin = Optimization.GradientDescent.SecondOrder_FullStep(f, startPoint, eps, out
calcsF, out calcsGradient, out calcsHessian);

                //determine number of decimal places to show
                int dp = BitConverter.GetBytes(decimal.GetBits((decimal)eps)[3])[2] + 1;

                //Show on console
                Console.WriteLine("{0,8}{1,8:F" + dp + "}{2,8:F" + dp + "}{3,8:F" + dp +
"}{4,8}{5,8}{6,8}", eps, (double)xMin[0], (double)xMin[1], (double)f(xMin), calcsF, calcsGradient,
calcsHessian);

            }
            #endregion

            #region 4.) Second Order - Newtons Method, Division Method
            //Show the table header
            Console.WriteLine();
            Console.WriteLine("----- Gradient Search, Second Order, Division Method -----");
            Console.WriteLine("     eps      X1      X2    f(x)   Calcs F  Calcs Gr  Calcs Hess");

            //Show Results for each accuracy
            foreach (double eps in epsValues)
            {
                //Get solution
                int calcsF;
                int calcsGradient;
                int calcsHessian;
                DV startPoint = new DV(new D[] { 0, 0 });
                DV xMin = Optimization.GradientDescent.SecondOrder_DivisionMethod(f, startPoint, eps,
out calcsF, out calcsGradient, out calcsHessian);

                //determine number of decimal places to show
                int dp = BitConverter.GetBytes(decimal.GetBits((decimal)eps)[3])[2] + 1;

                //Show on console
                Console.WriteLine("{0,8}{1,8:F" + dp + "}{2,8:F" + dp + "}{3,8:F" + dp +
"}{4,8}{5,8}{6,8}", eps, (double)xMin[0], (double)xMin[1], (double)f(xMin), calcsF, calcsGradient,
calcsHessian);

            }
            #endregion

            //Wait for use to click something to exit
            Console.ReadKey();

        }
```

## GradientDescent_FirstOrder_OneDimensionalMethod

```
        public static DV FirstOrder_OneDimensionalMethod(Func<DV, D> f, DV startPoint, double accuracy,
out int calcsF, out int calcsGradient, out DV[] x, out double[] fx)
        {
            //Counters
            calcsF = 0; //Count how many times the objective function was used.
            calcsGradient = 0; //Count how many times the gradient was calculated.

            //Define our X vector
            int maxIterations = 1000;
            x = new DV[maxIterations];
            fx = new double[maxIterations];

            //Pick an initial guess for x
            int i = 0;
            x[i] = startPoint;
            fx[i] = f(x[i]); calcsF++;

            //Loop through gradient steps until min points are found, recompute gradient and repeat.
            while (true)
            {
                //Compute next step, using previous step
                i++;

                //Return failed results
                if (double.IsNaN(x[i-1][0]) || double.IsNaN(x[i - 1][1]) ||  (i == maxIterations))
                {
                    x = x.Take(i).ToArray();
                    fx = fx.Take(i).ToArray();
                    return null;
                }

                //Step 1 - Determine the gradient
                DV gradient = 0-AD.Grad(f, x[i - 1]); calcsGradient++;
                DV direction = gradient / Math.Sqrt(AD.Pow(gradient[0], 2) + AD.Pow(gradient[1], 2));
//Normalize Gradient

                //Step 2 - Build an objective function using the gradient.
                // This objective function moves downward in the direction of the gradient.
                // It uses golden ratio optimization to find the minimum point in this direction
                DV xPrev = x[i - 1];
                Func<D, D> objFStep = delegate (D alpha)
                {
                    DV xNew = xPrev + (alpha * direction);
                    return f(xNew);
                };
                var stepSearchResults = UnimodalMinimization.goldenRatioSearch(objFStep, 0, 1,
accuracy); //alpha can only be between 0 and 1
                double step = (stepSearchResults.a + stepSearchResults.b) / 2; //The step required to
get to the bottom
                calcsF += stepSearchResults.CalculationsUntilAnswer; //The number of calculations of f
that were required.

                //Step 3 - Move to the discovered minimum point
                x[i] = x[i - 1] + (step * direction);
                fx[i] = f(x[i]); calcsF++;

                //Step 4 - Check if accuracy has been met. If so, then end.
                double magGradient = Math.Sqrt(AD.Pow(gradient[0], 2) + AD.Pow(gradient[1], 2));
                if (magGradient < accuracy)
                    break;
            }

            //Return the minimization point.
            x = x.Take(i+1).ToArray();
            fx = fx.Take(i+1).ToArray();
            return x[i];
        }
```

## GradientDescent_FirstOrder_DivisionMethod

```
public static DV FirstOrder_DivisionMethod(Func<DV, D> f, DV startPoint, double accuracy, out int
calcsF, out int calcsGradient, out DV[] x, out double[] fx)
        {
            //Counters
            calcsF = 0; //Count how many times the objective function was used.
            calcsGradient = 0; //Count how many times the gradient was calculated.

            //Define our X vector
            int maxIterations = 10000;
            x = new DV[maxIterations];
            fx = new double[maxIterations];

            //Pick an initial guess for x
            int i = 0;
            x[0] = startPoint;
            fx[0] = f(x[0]); calcsF++;

            //Loop through gradient steps until min points are found, recompute gradient and repeat.
            double alpha = 1;
            while (true)
            {
                //Compute next step, using previous step
                i++;

                //Step 1 - Determine the gradient
                DV gradient = AD.Grad(f, x[i - 1]); calcsGradient++;

                //Step 2 - Division method, to compute the new x[i] and fx[i]
                DV xPrev = x[i - 1];
                Func<D, D> objFAlpha = delegate (D a)
                {
                    DV xNext = xPrev - (a * gradient);
                    return f(xNext);
                };
                alpha = alpha * 0.8;
                double beta = UnimodalMinimization.DivisionSearch(objFAlpha, fx[i - 1], alpha, out
fx[i], ref calcsF);
                x[i] = x[i - 1] - (beta * gradient);

                //Step 3 - Check if accuracy has been met. If so, then end.
                double magGradient = Math.Sqrt(AD.Pow(gradient[0], 2) + AD.Pow(gradient[1], 2));
                if (magGradient < accuracy)
                    break;
            }

            //Return the minimization point.
            x = x.Take(i + 1).ToArray();
            fx = fx.Take(i + 1).ToArray();
            return x[i];
        }
```

## GradientDescent_SecondOrder_Traditional

```
public static DV SecondOrder_FullStep(Func<DV, D> f, DV startPoint, double accuracy, out int calcsF,
out int calcsGradient, out int calcsHessian, out DV[] x, out double[] fx)
        {
            //Counters
            calcsF = 0; //Count how many times the objective function was used.
            calcsGradient = 0; //Count how many times the gradient was calculated.
            calcsHessian = 0; //Count how many times the second gradient was calculated.

            //Define our X vector
            int maxIterations = 10000;
            x = new DV[maxIterations];
            fx = new double[maxIterations];

            //Pick an initial guess for x
            int i = 0;
            x[0] = startPoint;

            //Loop through gradient steps until zeros are found
            while (true)
            {
                //Compute next step, using previous step
                i++;

                //Step 1 - Determine the gradients
                DV gradient = AD.Grad(f, x[i - 1]); calcsGradient++;
                var hess = AD.Hessian(f, x[i - 1]); calcsHessian++;

                //Loop through every entry in the DV and compute the step for each one.
                List<D> listSteps = new List<D>();
                while (true)
                {
                    try
                    {
                        int v = listSteps.Count;
                        listSteps.Add(-gradient[v] / hess[v, v]); // first-gradient divided by second-
gradient
                    }
                    catch
                    { break; }
                }
                DV fullStep = new DV(listSteps.ToArray());

                //Compute the new position using the step
                x[i] = x[i - 1] + fullStep;
                fx[i] = f(x[i]); calcsF++;

                //Check if accuracy has been met
                double magGradient = Math.Sqrt(AD.Pow(gradient[0], 2) + AD.Pow(gradient[1], 2));
                if (magGradient < accuracy)
                    break;
            }

            //Return the minimization point.
            x = x.Take(i + 1).ToArray();
            fx = fx.Take(i + 1).ToArray();
            return x[i];

        }
```

## GradientDescent_SecondOrder_Dampened

```
  public static DV SecondOrder_DivisionMethod(Func<DV, D> f, DV startPoint, double accuracy, out int
calcsF, out int calcsGradient, out int calcsHessian, out DV[] x, out double[] fx)
        {
            //Counters
            calcsF = 0; //Count how many times the objective function was used.
            calcsGradient = 0; //Count how many times the gradient was calculated.
            calcsHessian = 0; //Count how many times the second gradient was calculated.

            //Define our X vector
            int maxIterations = 10000;
            x = new DV[maxIterations];
            fx = new double[maxIterations];

            //Pick an initial guess for x
            int i = 0;
            x[i] = startPoint;
            fx[i] = f(x[i]); calcsF++;

            //Loop through gradient steps until zeros are found
            double alpha = 1;
            while (true)
            {
                //Compute next step, using previous step
                i++;

                //Step 1 - Determine the gradients
                DV gradient = AD.Grad(f, x[i - 1]); calcsGradient++;
                var hess = AD.Hessian(f, x[i - 1]); calcsHessian++;

                //Step 2 - Compute full step (alpha = 1). Loop through every entry in the DV and
compute the step for each one.
                List<D> listSteps = new List<D>();
                while (true)
                {
                    try
                    {
                        int c = listSteps.Count;
                        listSteps.Add(-gradient[c] / hess[c, c]); // first-gradient divided by second-
gradient
                    }
                    catch
                    { break; }
                }
                DV fullStep = new DV(listSteps.ToArray());

                //Step 3 - Division method, to compute the new x[i] and fx[i]
                DV xPrev = x[i - 1];
                Func<D, D> objFAlpha = delegate (D a)
                {
                    DV xNext = xPrev + (a * fullStep);
                    return f(xNext);
                };
                alpha = alpha * 0.8;
                double beta = UnimodalMinimization.DivisionSearch(objFAlpha, fx[i - 1], alpha, out
fx[i], ref calcsF);
                x[i] = x[i - 1] + (beta * fullStep);

                //Check if accuracy has been met
                double magGradient = Math.Sqrt(AD.Pow(gradient[0], 2) + AD.Pow(gradient[1], 2));
                if (magGradient < accuracy)
                    break;
            }

            //Return the minimization point
            x = x.Take(i + 1).ToArray();
            fx = fx.Take(i + 1).ToArray();
            return x[i];

        }
```