Program.cs

```
using System;
using MathNet.Numerics.LinearAlgebra.Double;
using static System.Console;
using static System.Globalization.CultureInfo;
using static System.Math;
using static MathNet.Numerics.Generate;
using MMP.Tools;
using static MMP.Tools.GnuPlot;

using funstr = MMP.Tools.Generator<double, double>;

namespace MMP
{
    public class Ellipse
    {
        double x0, y0, a, b, phi;

        public Ellipse(double x0 = 0, double y0 = 0, double a = 1, double b = 1, double phi = 0)
        {
            this.x0 = x0; this.y0 = y0;
            this.a = a; this.b = b;
            this.phi = phi;
        }
        public Tuple<double[], double[]> Create(double[] I)
        {
            var v = DenseVector.OfArray(I);
            var x = v.Map(t => x0 + a * Cos(t));
            var y = v.Map(t => y0 + b * Sin(t));
            if (phi != 0.0)
            {
                var Rot = DenseMatrix.CreateDiagonal(2, 2, Cos(phi));
                Rot[1, 0] = Sin(phi); Rot[0, 1] = -Rot[1, 0];
                var xy = Rot * DenseMatrix.OfRowVectors(x - x0, y - y0);
                x = xy.Row(0) + x0;
                y = xy.Row(1) + y0;
            }
            return new Tuple<double[], double[]>(x.AsArray(), y.AsArray());
        }
    }


    class Program
    {
        static Terminal win = new Terminal(1200, 600);
        static Terminal png = new Terminal(1200, 600, Terminal.Enum.png);
        static Terminal eps = new Terminal(Terminal.Enum.eps);
        static Terminal tex = new Terminal(Terminal.Enum.epslatex);
        static Terminal svg = new Terminal(Terminal.Enum.svg);
        static Terminal pdf = new Terminal(Terminal.Enum.pdf);

        static void test_expressions()
        {
            Set = "yrange [-0.5:1]";
            Plot("sin(x)/x,1,1-x**2/6,1-x**2/6+x**4/120");
            WaitForKey();
            Reset();
        }
```

```
static void test_graphs()
{
    var x = LinearSpaced(51, 0, 2 * PI);
    GraphList<double> glist =
        new GraphList<double>(x,
        new funstr(Sin, "sin(x)"),
        new funstr(t=>Sin(2*t), "sin(2*x)"));
    Plot(glist);
    WaitForKey();
}

static void test_curves()
{
    var x = LinearSpaced(51, 0, 2 * PI);
    Ellipse circle1 = new Ellipse();
    Ellipse circle2 = new Ellipse(0.5, 0.5);
    Ellipse ellipse = new Ellipse(0.25, 0.25, 2, 0.25, PI / 4);

    CurveList<double> clist =
        new CurveList<double>(x,
        new Generator<double>(circle1.Create, "Circle_1"),
        new Generator<double>(circle2.Create, "Circle_2"),
        new Generator<double>(ellipse.Create, "Ellipse"));

    Set = "size ratio - 1";
    Terminal[] term = { png, eps, tex, svg, pdf, win };
    Terminals = term;
    Plot(clist);
    WaitForKey();
}
static void Main(string[] args)
```

```
        {

            DefaultThreadCurrentCulture = InvariantCulture;
            test_expressions();
            test_graphs();
            test_curves();

        }
    }
}
```

Vectorize.cs

```
using System;

namespace MMP.Tools
{
    public class Vectorize<T1, T2>
    {
        Func<T1, T2> f;
        public Vectorize(Func<T1, T2> f)
        { this.f = f; }
        public T2[] F(T1[] x)
        {
            T2[] y = new T2[x.Length];
            for (int i = 0; i < x.Length; i++)
            {
                y[i] = f(x[i]);
            }
            return y;
        }
        public static implicit operator Vectorize<T1,T2>(Func<T1, T2> f)
        {
            return new Vectorize<T1, T2>(f);
        }
        public static implicit operator Func<T1[], T2[]>(Vectorize<T1, T2> v)
        {
            return v.F;
        }
    }
}
```

Graphics.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;

namespace MMP.Tools
{

    public class Terminal
    {
        public enum Enum
        {
            wxt, png, pdf, svg, canvas, eps, epslatex
        }

        static Dictionary<Enum, string[]> dict;
        string FileName { get; set; }

        static Terminal()
        {
            int n = Enum.GetNames(typeof(Enum)).Length;
            dict = new Dictionary<Enum, string[]>(n);
            dict.Add(Enum.canvas, new string[] { "canvas" });
            dict.Add(Enum.pdf, new string[] { "pdf", "pdf" });
            dict.Add(Enum.png, new string[] { "png", "png" });
            dict.Add(Enum.svg, new string[] { "svg", "svg" });
            dict.Add(Enum.wxt, new string[] { "wxt" });
            dict.Add(Enum.eps, new string[] { "postscript eps enhanced color", "eps" });
            dict.Add(Enum.epslatex, new string[] { "epslatex color colortext", "tex" });
        }

        public uint Width, Height;
        Enum terminal;
        public Terminal(uint w, uint h, Enum t = Enum.wxt)
        {
            Width = w; Height = h;
            terminal = t;
            FileName = "pic";
        }

        public Terminal(Enum t = Enum.wxt)
        {
            Width = 0; Height = 0;
            terminal = t;
            FileName = "pic";
        }

        public override string ToString()
        {
            string sz = "", fn = "";
            if (Width != 0 && Height != 0)
                sz = string.Format($" size {Width}, {Height}");
            if (terminal != Enum.wxt)
                fn = $"set output '{FileName}.{dict[terminal][1]}'";
            return $"set terminal {dict[terminal][0]}{sz}{Environment.NewLine}{fn}";
        }
    }
```

```csharp
public static class GnuPlot
{
    public static string PathToGnuplot = @"C:\Program Files (x86)\gnuplot\bin";
    private static Process ExtPro;
    private static StreamWriter GnupStWr;

    private static List<string> stringBuffer = new List<string>();
    private static List<Terminal> terminals = new List<Terminal>();
    public static IEnumerable<Terminal> Terminals
    {
        get
        {
            if (terminals.Count == 0)
                terminals.Add(new Terminal());
            return terminals;
        }
        set
        {
            terminals.AddRange(value);
        }
    }
    public static string Set
    {
        get
        {
            return string.Concat(stringBuffer);
        }
        set
        {
            stringBuffer.Add("set " + value + Environment.NewLine);
        }
    }

    public static void Write(string s)
    {
        GnupStWr.Write(s);
        Console.Write(s);
    }

    public static void WriteLine(string s)
    {
        GnupStWr.WriteLine(s);
        Console.WriteLine(s);
    }
    public static void Write(object o)
    {
        GnupStWr.Write(o.ToString());
        Console.Write(o.ToString());
    }

    public static void WriteLine(object o)
    {
        GnupStWr.WriteLine(o.ToString());
        Console.WriteLine(o.ToString());
    }

    public static void WriteLine()
    {
        GnupStWr.WriteLine();
        Console.WriteLine();
    }
```

```
public static void Reset()
{
    stringBuffer.Clear();
    terminals.Clear();
    GnupStWr.WriteLine("reset");
}

public static void WaitForKey()
{
    Console.WriteLine();
    Console.WriteLine("Press any key to next plot ... ");
    Console.ReadKey();
}


static GnuPlot()
{
    if (PathToGnuplot[PathToGnuplot.Length - 1].ToString() != @"\")
        PathToGnuplot += @"\";
    ExtPro = new Process();
    ExtPro.StartInfo.FileName = PathToGnuplot + "gnuplot.exe";
    ExtPro.StartInfo.UseShellExecute = false;
    ExtPro.StartInfo.RedirectStandardInput = true;
    ExtPro.Start();
    GnupStWr = ExtPro.StandardInput;
}

public static void Plot(string funs)
{
    GnuPlot.Write(Set);
    foreach (var terminal in Terminals)
```

```
        {
            GnuPlot.WriteLine(terminal);
            GnuPlot.WriteLine("plot " + funs);
            GnupStWr.Flush();
        }
    }

    public static void Plot<T>(params CurveList<T>[] curves)
    {
        GnuPlot.Write(Set);

        using (StreamWriter outputFile = new StreamWriter("data"))
            for (int i = 0; i < curves.Length; i++)
            {
                //var T = DenseMatrix.OfColumnArrays(curves[i].ToArray());
                //DelimitedWriter.Write(outputFile, T);

                for (int ir = 0; ir < curves[i][0].Length; ir++)
                {
                    for (int ic = 0; ic < 2 * curves[i].Count; ic++)
                    {
                        outputFile.Write(curves[i][ic][ir]);
                        outputFile.Write("\t");
                    }
                    outputFile.WriteLine();
                }

                if (i < curves.Length - 1)
                {
                    outputFile.WriteLine();
                    outputFile.WriteLine();
                    }
                }

            foreach (var terminal in Terminals)
            {
                GnuPlot.WriteLine(terminal.ToString());
                GnuPlot.Write("plot ");
                for (int ic = 0; ic < curves.Length; ic++)
                {
                    for (int i = 0; i < curves[ic].Count; i++)
                    {
                        GnuPlot.Write($"'data' i {ic} u {2 * i + 1}:{2 * i + 2}  w l t 
'{curves[ic].Titles[i]}'");
                        if (ic < curves.Length  - 1 || i < curves[ic].Count - 1)
                            GnuPlot.Write(@", \");
                        GnuPlot.WriteLine();
                    }
                }
                GnupStWr.Flush();
            }
        }


        public static void Plot<T>(params GraphList<T>[] graphs)
        {
            GnuPlot.Write(Set);
            using (StreamWriter outputFile = new StreamWriter("data"))
                for (int i = 0; i < graphs.Length; i++)
                {

                    for (int ir = 0; ir < graphs[i][0].Length; ir++)
                    {
                        for (int ic = 0; ic < graphs[i].Count; ic++)
                        {
                            outputFile.Write(graphs[i][ic][ir]);
                            outputFile.Write("\t");
                        }
                        outputFile.WriteLine();
                    }

                    if (i < graphs.Length - 1)
                    {
                        outputFile.WriteLine();
                        outputFile.WriteLine();
                    }
                }

            foreach (var terminal in Terminals)
            {
                GnuPlot.WriteLine(terminal.ToString());
                GnuPlot.Write("plot ");
                for (int ic = 0; ic < graphs.Length; ic++)
                {
                    for (int i = 1; i < graphs[ic].Count; i++)
                    {
                        GnuPlot.Write($"'data' i {ic} u {1}:{i + 1}  w l t '{graphs[ic].Titles[i-
1]}'");
                        if (ic < graphs.Length - 1 || i < graphs[ic].Count - 1)
                            GnuPlot.Write(@", \");
                        GnuPlot.WriteLine();
                    }
                }
                GnupStWr.Flush();
            }
        }

        public static void Exit()
        {
            GnuPlot.WriteLine("quit");
            GnupStWr.Flush();
        }

    }

    public class Generator<T1, T2>
    {
        public string Title { get; set; }
        public Func<T1[], T2[]> Fun { get; set; }
        public Generator(Func<T1[], T2[]> fun, string title = "")
        {
            Fun = fun;
            Title = title;
        }
        public Generator(Func<T1, T2> fun, string title = "")
        {
            Fun = (Vectorize<T1,T2>)fun;
            Title = title;
        }
    }

    public class Generator<T>
    {
        public string Title { get; }
        public Func<T[], Tuple<T[], T[]>> Fun { get; }
        public Generator(Func<T[], Tuple<T[], T[]>> fun, string title = "")
        {
            Fun = fun;
            Title = title;
        }
    }

    public class GraphList<T> : List<T[]>
    {
        public List<string> Titles = new List<string>();
        public GraphList(T[] x, params Generator<T, T>[] funs)
        {
            this.Add(x);
            for (int i = 0; i < funs.Length; i++)
            {
                this.Add(funs[i].Fun(x));
                Titles.Add(funs[i].Title);
            }
        }
    }

    public class CurveList<T> : List<T[]>
    {
        public List<string> Titles = new List<string>();
        public CurveList(T[] x, params Generator<T>[] funs)
        {
            for (int i = 0; i < funs.Length; i++)
            {
                var xy = funs[i].Fun(x);
                this.Add(xy.Item1);
                this.Add(xy.Item2);
                Titles.Add(funs[i].Title);
            }
        }
        public new int Count { get { return base.Count / 2; } }
    }

    public class CurveList<T1, T2>
    {
        public List<T2[]> List { get; }
        public CurveList(T1[] x, params Func<T1[], T2[]>[] funs)
        {
            List = new List<T2[]>();
            for (int i = 0; i < funs.Length; i++)
            {
                List.Add(funs[i](x));
            }
        }
    }

}
```