|  |  |  |  |
| --- | --- | --- | --- |
| static double[] GaussMethod(double[,] a, int row)  {  double[] x; // roots array  // If recursion is over then return  if (row == a.GetUpperBound(0))  {  x = new double[a.GetLength(0)];  x[row] = a[row, row + 1] / a[row, row];  return x;  }  // Reduce roots count  for (var i = row + 1; i < a.GetLength(0); i++)  {  var k = a[i, row] / a[row, row];  for (var j = row; j < a.GetLength(1); j++)  a[i, j] -= a[row, j] \* k;  }  // Get roots of submatrix  x = GaussMethod(a, row + 1); // Call recursion  // Get current root  x[row] = a[row, a.GetUpperBound(1)];  for (var j = row + 1; j < a.GetLength(0); j++)  x[row] -= x[j] \* a[row, j];  x[row] /= a[row, row];  // Return roots of current matrix  return x;  } | static List<double> CompletePivoting(double[,] a)  {  // If recursion is over then return list of single root  if (a.GetLength(0) == 1)  return new List<double> { a[0, 1] / a[0, 0] };  // Search maximum absolute element  int iMax = 0, jMax = 0;  for (int i = 0; i < a.GetLength(0); i++)  for (int j = 0; j < a.GetLength(0); j++)  if (Math.Abs(a[i, j]) > Math.Abs(a[iMax, jMax]))  {  iMax = i;  jMax = j;  }  // Get submatrix of current matrix without pivot  double[,] b = new double[a.GetLength(0) - 1, a.GetLength(1) - 1];  for (int i = 0; i < b.GetLength(0); i++)  for (int j = 0; j < b.GetLength(1); j++)  {  var k = a[i < iMax ? i : i + 1, jMax] / a[iMax, jMax];  b[i, j] = a[i < iMax ? i : i + 1, j < jMax ? j : j + 1] - a[iMax, j < jMax ? j : j + 1] \* k;  }  // Get roots of submatrix  List<double> x = CompletePivoting(b); // Call recursion  // Get pivot root  double root = a[iMax, a.GetUpperBound(1)];  for (int j = 0; j < a.GetLength(0) - 1; j++)  root -= a[iMax, j < jMax ? j : j + 1] \* x[j];  x.Insert(jMax, root / a[iMax, jMax]);  // Return roots of current matrix  return x;  } | static double[] DirectIteration(double[,] a, double eps)  {  // Get matrix norm  double q = 0;  for (int i = 0; i < a.GetLength(0); i++)  for (int j = 0; j < a.GetLength(1); j++)  if (i != j)  {  a[i, j] /= a[i, i];  if (j < a.GetLength(0))  q += Math.Pow(a[i, j], 2);  }  q = Math.Sqrt(q);  // Get least deviation  eps = eps \* (1 - q) / q;  // Initial roots  double[] x0, x = new double[a.GetLength(0)];  for (var i = 0; i < a.GetLength(0); i++)  x[i] = a[i, a.GetUpperBound(1)];  do  {  x0 = (double[])x.Clone();  // Evaluate each new more exact root  for (var i = 0; i < a.GetLength(0); i++)  {  x[i] = a[i, a.GetUpperBound(1)];  for (var j = 0; j < a.GetLength(0); j++)  if (i != j)  x[i] -= x0[j] \* a[i, j];  }  //x[i] /= a[i, i];  // Calculate error for each root  for (var i = 0; i < x.Length; i++)  x0[i] = Math.Abs(x0[i] - x[i]);  }  while (x0.Max() > eps);  return x;  } | static double[] GaussSeidelMethod(double[,] a, double eps)  {  // Get matrix norm  double q = 0;  for (int i = 0; i < a.GetLength(0); i++)  for (int j = 0; j < a.GetLength(1); j++)  if (i != j)  {  a[i, j] /= a[i, i];  if (j < a.GetLength(0))  q += Math.Pow(a[i, j], 2);  }  q = Math.Sqrt(q);  // Get least deviation  eps = eps \* (1 - q) / q;  double error;  // Initial roots  double[] x = new double[a.GetLength(0)];  for (var i = 0; i < a.GetLength(0); i++)  x[i] = a[i, a.GetUpperBound(1)];  do  {  error = 0;  // Evaluate each new more exact root  for (var i = 0; i < a.GetLength(0); i++)  {  // Store previous value of x[i]  var prevXi = x[i];  // Recalculate x[i]  x[i] = a[i, a.GetUpperBound(1)];  for (var j = 0; j < a.GetLength(0); j++)  if (i != j)  x[i] -= x[j] \* a[i, j];  //x[i] /= a[i, i];  // Calculate error  error = Math.Max(error, Math.Abs(prevXi - x[i]));  }  }  while (error > eps);  return x;  } |
| static IEnumerable<double> GaussJordanMethod(double[,] a)  {  for (var i = 0; i < a.GetLength(0); i++)  {  // Divide current row by diagonal coefficient  for (var j = i + 1; j < a.GetLength(1); j++)  a[i, j] /= a[i, i];  // Eliminate diagonal coefficient from other rows  for (var k = 0; k < a.GetLength(0); k++)  for (var j = i + 1; j < a.GetLength(1); j++)  if (k != i)  a[k, j] -= a[i, j] \* a[k, i];  }  // Return right-hand free coefficients  for (var i = 0; i < a.GetLength(0); i++)  yield return a[i, a.GetUpperBound(1)];  } |
| static double CompositeTrapeziumRule(Func<double, double> f, double a, double b, double eps)  {  var n = (int)(1 / Math.Sqrt(eps));  double prevResult, nextResult = double.PositiveInfinity;  do  {  prevResult = nextResult;  nextResult = (f(a) + f(b)) / 2;  double h = (b - a) / n;  for (int i = 1; i < n; i++)  nextResult += f(a + i \* h);  nextResult \*= h;  n \*= 2;  }  while (Math.Abs(nextResult - prevResult) / 3 > eps);  return nextResult;  } |
| static double NewtonRaphsonMethod(Func<double, double> f, Func<double, double> df, Func<double, double> d2f, double a, double b, double eps)  {  double x = d2f(a) \* f(a) > 0 ? a : b;  double m1 = Math.Min(Math.Abs(df(a)), Math.Abs(df(b)));  while (Math.Abs(f(x)) / m1 > eps)  x = x - f(x) / df(x);  return x;  } |
| static double ChordMethod(Func<double, double> f, Func<double, double> df, Func<double, double> d2f, double a, double b, double eps)  {  double c, x;  if (d2f(a) \* f(a) > 0)  {  c = a;  x = b;  }  else  {  c = b;  x = a;  }  double m1 = Math.Min(Math.Abs(df(a)), Math.Abs(df(b)));  while (Math.Abs(f(x)) / m1 > eps)  x = x - f(x) \* (x - c) / (f(x) - f(c));  return x;  } |
| static double MethodOfSuccessiveApproximations(Func<double, double> f, Func<double, double> df, double a, double b, double eps)  {  int sign = Math.Sign(df(a));  double m1 = Math.Min(sign \* df(a), sign \* df(b));  double M1 = Math.Max(sign \* df(a), sign \* df(b));  double q = 1 - m1 / M1;  double delta = (1 - q) / q \* eps;  double x0 = (a + b) / 2;  double x1 = x0 - sign \* f(x0) / M1;  while (Math.Abs(x0 - x1) > delta)  {  x0 = x1;  x1 = x0 - sign \* f(x0) / M1;  }  return x1;  } |
| static List<KeyValuePair<double, double>> RungeMethod(Func<double, double, double> f, double x0, double y0, double l, double eps)  {  List<KeyValuePair<double, double>> points;  double x, y = double.PositiveInfinity, yPrev;  double h = Math.Pow(eps, 0.25);  do  {  yPrev = y;  points = new List<KeyValuePair<double, double>> { new KeyValuePair<double, double>(x = x0, y = y0) };  while (x < x0 + l)  {  var k1 = h \* f(x, y);  var k2 = h \* f(x + h / 2, y + k1 / 2);  var k3 = h \* f(x + h / 2, y + k2 / 2);  var k4 = h \* f(x + h, y + k3);  y += (k1 + 2 \* k2 + 2 \* k3 + k4) / 6;  x += h;  points.Add(new KeyValuePair<double, double>(x, y));  }  h /= 2;  }  while (Math.Abs(yPrev - y) / 15 < eps);  return points;  } |
| static void LagrangeInterpolation(Func<double, double> f, double a, double b, double eps, double[] checkPoints)  {  for (var n = 3; ; n++)  {  var h = (a + b) / n;  var nodes = from i in Enumerable.Range(0, n) let x = a + i \* h select new { X = x, Y = f(x) };  var values = from checkPoint in checkPoints  select new  {  Exact = f(checkPoint),  Interpolated = nodes.Sum(Node => Node.Y \* nodes.Aggregate(1.0, (p, node) => Node.X == node.X ? p : p \* (checkPoint - node.X) / (Node.X - node.X)))  };  if (values.All(value => Math.Abs(value.Exact - value.Interpolated) < eps))  {  foreach (var value in values)  Console.WriteLine(value.Interpolated);  break;  }  }  } |
| static double CompositeSimpsonsRule(Func<double, double> f, double a, double b, double eps)  {  var n = (int)(Math.Pow(eps, -0.25));  n += n % 2;  double prevResult, nextResult = double.PositiveInfinity;  do  {  prevResult = nextResult;  nextResult = f(a) + f(b);  double h = (b - a) / n;  for (int i = 1; i < n; i++)  nextResult += (i % 2 == 0 ? 2 : 4) \* f(a + i \* h);  nextResult \*= h / 3;  n \*= 2;  }  while (Math.Abs(nextResult - prevResult) / 15 > eps);  return nextResult;  } |
| static double BisectionMethod(Func<double, double> f, double a, double b, double eps)  {  double x = (a + b) / 2;  while (b - a > 2 \* eps)  {  if (f(a) \* f(x) < 0) b = x;  else a = x;  x = (a + b) / 2;  }  return x;  } |