Greičiausiojo nusileidimo (*angl. steepest descent*) metodo C (C++) realizacija

Greičiausiojo nusileidimo metodo realizacija

- Iš http://www.mymathlib.com/optimization/nonlinear/unconstrained/steepest_descent.html
- Parsisiunčiame *steepest_descent.c*

steepest_descent.c (1)

```
////////////////////////////////////////////////////////////////////////////////
// File: steepest_descent.c                                                   //
// Routines:                                                                  //
//    Steepest_Descent                                                        //
////////////////////////////////////////////////////////////////////////////////
#include <stdlib.h>  // required for malloc(), free(), and NULL
#include <stdio.h>
// Externally Defined Routines

double Vector_Max_Norm(double v[], int n);

int Minimize_Down_the_Line(double (*f)(double *), double x[], double fx,
                         double *p, double v[], double y[], double cutoff,
                         double cutoff_scale_factor, double tolerance, int n);

void Copy_Vector(double *d, double *s, int n);
```

steepest_descent.c (2)

```
////////////////////////////////////////////////////////////////////////////////
//  int  Steepest_Descent(double (*f)(double *),                              //
//          void (*df)(double *, double *),                                   //
//          int (*stopping_rule)(double*, double, double*, double, double*,   //
//                                                                int, int), //
//          double a[], double *fa, double *dfa, double cutoff,               //
//          double cutoff_scale_factor, double tolerance, int n)              //
//                                                                            //
//  Description:                                                              //
//     Given a nonlinear function f:R^n -> R, the optimal steepest descent    //
//     method searches for the minimum of f by starting at a predetermined    //
//     point x and then following the line from x in the direction opposite   //
//     to that given by grad f evaluated at x until a minimum on the line is  //
//     reached.  The process is repeated until halted by the stopping rule.   //
//     The steepest descent method is best used when the point x is far away  //
//     from the minimum.  Near the minimum, the norm of grad f is small and   //
//     convergence is slow and another method should be used accelerate       //
//     convergence to the final approximation of the minimum.                 //
//                                                                            //
//     It is possible that if the initial point happens to be the location of //
//     a local maximum or a saddle point, the gradient will vanish and the    //
//     procedure will fail.                                                   //
//     It is also possible that the method will converge to a saddle point    //
//     so that the result should be checked and if it is a saddle point then  //
//     try a different initial point.
```

steepest_descent.c (3)

```
//  Arguments:                                                              //
//     double (*f)(double *)                                                //
//        Pointer to a user-defined function of a real n-dimensional vector //
//        returning a real number (type double).                            //
//     void   (*df)(double *, double *)                                     //
//        Pointer to a user-defined function which returns the gradient of the//
//        function f, above, in the second argument evaluated at the point  //
//        in the first argument.  I.e. df(x,dfx), where dfx = grad f(x).    //
//     int (*stopping_rule)(double*, double, double*, double, double*, int, //
//                                                                    int)  //
//        Pointer to a user-defined function which controls the iteration of //
//        the steepest descent method.  If the stopping rule is non-zero the //
//        process is halted and if zero then the process continues iterating. //
//        Steepest descent converges slowly in the neighborhood of an       //
//        extremum and another method is then called to converge to the final //
//        solution.
//        The arguments are: original point, the value of the function f    //
//        evaluated at the original point, the new point, the value of the  //
//        function evaluated at the new point, the gradient of the function //
//        at the new point, the total number of iterations performed and the //
//        dimension, n, if a point and the gradient.                        //
```

Mano (user-defined) funkcijos

```
// Apskaiciuoja Six-hump Camel Back funkcijos reiksme taske x
double SixHumpCamelBack(double *x){
    return (4-2.1*x[0]*x[0]+x[0]*x[0]*x[0]*x[0]/3)*x[0]*x[0] + x[0]*x[1] +
    (-4+4*x[1]*x[1])*x[1]*x[1];
}
// Apskaiciuoja Six-hump Camel Back gradiento reiksme taske x
void SixHumpCamelBackGradient(double *x, double *fGrad){
    fGrad[0] = 8*x[0]-8.4*x[0]*x[0]*x[0]+2*x[0]*x[0]*x[0]*x[0]*x[0]+x[1];
    fGrad[1] = x[0]-8*x[1]+16*x[1]*x[1]*x[1];
}
// Algoritmo sustojimo salygas kontroliuojanti funkcija
int StoppingRule(double* a, double fa, double* x, double fx, double* dfa, int
iteration, int n){
    double fEps = abs(fx - fa); // Funkcijos reiksmiu skirtumas
    double xa[n];
    for(int i = 0; i < n; ++i) xa[i] = x[i]-a[i];
    double xEps = Vector_Max_Norm(xa, 2); // Argumento skirtumo norma
    double dfaEps = Vector_Max_Norm(dfa, 2); // Gradiento norma
    iteration++; // Iteraciju skaitiklis
    if(iteration > 3)
        return -6;
    else
        return 0;
}
```

steepest_descent.c (4)

```
//      double *a                                                              //
//         On input, a is the initial point to start the descent.  On output,` //
//         a is the final point before the iteration is halted.                //
//      double *fa                                                             //
//         On input and output, *fa is the value of f(a[]).                    //
//      double *dfa                                                            //
//         Working storage used to hold the gradient of f at a. Should be      //
//         dimensioned at least n in the calling routine.                      //
//      double cutoff                                                          //
//         The upper bound of the parameter p during the line search where     //
//         the line is x - pv, v is the gradient of f at x, 0 <= p <= cutoff.  //
//      double cutoff_scale_factor                                             //
//         A parameter which limits the displacement in any single step during //
//         the parabolic extrapolation phase of the line search.               //
//      double tolerance                                                       //
//         A parameter which controls the termination of the line search.      //
//         The line search is terminated when the relative length of the       //
//         interval of uncertainty to the magnitude of its mid-point is less   //
//         than or equal to tolerance.  If a nonpositive number is passed,     //
//         tolerance is set to sqrt(DBL_EPSILON).                              //
//      int    n                                                               //
//         On input, n is the dimension of a and dfa.
```

steepest_descent.c (5)

```
//  Return Values:                                                        //
//     0   Success                                                        //
//    -1   In the line search three points are collinear.                //
//    -2   In the line search the extremum of the parabola through the three  //
//         points is a maximum.                                           //
//    -3   Int the line search the initial points failed to satisfy the   //
//         condition that x1 < x2 < x3 and fx1 > fx2 < fx3.               //
//    -4   Not enough memory.                                             //
//    -5   The gradient evaluated at the initial point vanishes.          //
//   Other user specified return from stopping_rule().                    //
//                                                                        //
//  Example:                                                              //
//     extern double f(double*);                                          //
//     extern void df(double*, double*);                                  //
//     extern int Stopping_Rule(double*, double, double*, double, double*,   //
//                                                            int, int); //
//     #define N                                                          //
//     double cutoff, cutoff_scale_factor, tolerance;                     //
//     double a[N], dfa[N];                                               //
//                                                                        //
//     (your code to initialize the vector a, set cutoff, )               //
//     (cutoff_scale_factor, and tolerance.               )               //
//                                                                        //
//     err = Steepest_Descent( f, df, Stopping_Rule, a, fa, dfa, cutoff,  //
//                cutoff_scale_factor, tolerance, N);                     //
//                                                                        //
////////////////////////////////////////////////////////////////////////////
```

steepest_descent.c (6)

```
int  Steepest_Descent(double (*f)(double *), void (*df)(double *, double *),
    int (*stopping_rule)(double*, double, double*, double, double*, int, int),
                               double a[], double *fa, double *dfa, double cutoff,
                               double cutoff_scale_factor, double tolerance, int n)
{
   double* x;
   double* initial_a = a;
   double* tmp;
   double fx;
   double p;
   int err = 0;
   int iteration = 0;

   x = (double *) malloc(n * sizeof(double));
   if ( x == NULL) { err = -3; }

   df(a, dfa);
   if (Vector_Max_Norm(dfa,n) == 0.0) err = -5;
```

steepest_descent.c (7)

```
   while (!err) {
      if (Vector_Max_Norm(dfa,n) == 0.0) break;
      err = Minimize_Down_the_Line(f, a, *fa, &p, dfa, x, cutoff,
                                          cutoff_scale_factor, tolerance, n);
      if ( err ) break;
      fx = f(x);
      iteration++;
      df(x, dfa);
      err = stopping_rule( a, *fa, x, fx, dfa, iteration, n);
      *fa = fx;
      tmp = a;
      a = x;
      x = tmp;
   }
   if (a != initial_a ) {
      Copy_Vector(initial_a, a, n);
      x = a;
   }
   free(x);
   return err;
}
```

main.cpp (I)

```
int main(int argc, const char * argv[])
{
    double region[] = {-1.9, 1.9, -1.1, 1.1};
    double a[N] = {0.0, 1.0}; // N-matis Vektorius
    /*srand(time(0)); // Naudoja vis kita seed'a
    double a[N]; // N-matis Vektorius
    for(int i = 0; i < N; ++i){
        a[i] = GetRandomNumber(region[2*i], region[2*i+1]);
    }*/
    double fa = SixHumpCamelBack(a); // Funkcijos reiksme pradiniame taske a
    double dfa[N];
    SixHumpCamelBackGradient(a, dfa); // Funkcijos gradiento reiksme taske a
    double cutoff = 1.0, cutoff_scale_factor = 1.0; // Pap. parametrai
    double tolerance = 0.01;
    int err = Steepest_Descent( SixHumpCamelBack, SixHumpCamelBackGradient, StoppingRule,
    a, &fa, dfa, cutoff, cutoff_scale_factor, tolerance, N);
```

main.cpp (2)

```
    switch (err)
    {
        case 0:
            cout << "Success" << endl;
            break;
        case -1:
            cout << "In the line search three points are collinear." << endl;
            break;
        case -2:
            cout << "In the line search the extremum of the parabola through the
            three points is a maximum." << endl;
            break;
        case -3:
            cout << "Int the line search the initial points failed to satisfy
            the condition that x1 < x2 < x3 and fx1 > fx2 < fx3." << endl;
            break;
        case -4:
            cout << "Not enough memory." << endl;
            break;
        case -5:
            cout << "The gradient evaluated at the initial point vanishes." << endl;
        case -6:
            cout << "Exceed maximal number of iterations." << endl;
        break;
    }
    cout << "Greiciausio nusileidimo (angl. Steepest Descent) metodu" << endl;
    cout << "surastas sprendinys yra:" << endl;
    cout << "xMin = (" << a[0] << ", " << a[1] << ")" << endl;
    cout << "f(xMin) = " << fa << endl;
    return 0;
}
```

Kartotinių paieškų metodas (*angl. multi start*)

1. Papildyti Jūsų turimas atsitiktinės paieškos (*angl. pure random search or Monte Carlo*) programas, kad jos iš 3 geriausių surastų sprendinių atliktų lokalias paieškas - sprendinių patikslinimus naudojantis greičiausio nusileidimo metodu (*angl. steepest descent*).

   git clone https://github.com/Remziukas/SteepestDescent.git

2. Papildyti dar (bent) vienu laisvai pasirinktu lokaliosios paieškos metodu: http://www.mymathlib.com/optimization/nonlinear/unconstrained/