Transactions of A. Razmadze
Mathematical Institute
Vol. **173** (2019), 39–46

ADD-IN FOR SOLVERS OF UNCONSTRAINED MINIMIZATION TO ELIMINATE LOWER BOUNDS OF VARIABLES BY TRANSFORMATION

K. GELASHVILI

Abstract. The paper considers the issue of implementation of a software add-in, which transforms minimization problems with constraints only on lower bounds of variables to unconstrained ones. The add-in eliminates lower bounds of variables through their transformation, without creation of new auxiliary variables. The add-in can be included in any algorithm of unconstrained minimization, which has certain design. At the same time, any number of suitable transformations of variables can be included into the add-in.

The add-in is implemented in C++ and consists of three components: small collection of transformations of variables; small collection of minimization test-problems with constraints only on lower bounds of variables; stopping conditions of minimization algorithms and program-driver.

In the numerical experiments, mainly CG-DESCENT-C-6.8 and l-bfgs were used. Results show that empowering unconstrained minimization algorithms by such add-in is a promising direction, especially, for symmetric matrix games.

Materials of experiments are uploaded at GitHub: https://github.com/kobage.

1. Introduction

Let us consider the minimization problem with constraints only on lower bounds of the variables:

$$\min_{x\in\mathbb{R}^n} f(x) \text{ subject to } x_i \geq l_i, \quad i\in\mathcal{I}, \tag{1}$$

where $f:\mathbb{R}^n\to\mathbb{R}$ is a smooth function, $\mathcal{I}$ is a subset of $1,2,\ldots,n$. $\forall i\in\mathcal{I}$, $l_i\in\mathbb{R}$ is a lower bound of x_i variable.

Problems of practical interest are often modelled in the form (1) (see [3]). Symmetric matrix games belong to this set of problems. Usually, such kind of constraints arise when variables represent certain physical quantities.

Problem (1) contains the constraints only on lower bounds of variables and represents particular case of bound constrained or box-constrained optimization problems. Because of the relevance of box-constrained optimization, investigation of their numerical algorithms also is a relevant issue.

From the point of view of citation, L-BFGS-B (see. [10]), based on the gradient projection method, is the most remarkable approach. On the other hand, this method is neither unique, nor the fastest (see [6]), and there are another alternatives. In the presented paper, one of the oldest approaches is promoted. This approach is the most natural from the point of view of mathematics and implements transformations of variables: problem (1) will be transformed into an unconstrained one by transforming the variables. After this, any unconstrained minimization algorithm can be used.

The idea of application of unconstrained minimization algorithms in constrained optimization via transformation of variables was proposed in 1966 by M.J. Box (see [1]). After that, we can mention several papers that attracted attention, certain kind of constraints are named after M.J. Box, but in practical usage this approach did not become competitive. Among the possible reasons we can emphasize the following:

- In [1], there is no emphasis on a particular type of restrictions, therefore it is difficult to systematize the approach and create a software application based on it;

2010 *Mathematics Subject Classification.* 90C06, 90C30, 90-04.

Key words and phrases. Box-constrained optimization; Elimination of lower bounds; Unconstrained optimization; Symmetric matrix game; l-bfgs, CG-DESCENT; Modified heavy ball, MINOS, GUROBI.

- [1] emphasizes the possibilities of linearization of problems or maximum simplification, often by transforming to problems of linear programming. The process is accompanied by significant number of auxiliary variables.

Our approach is effective because we are trying to research an issue that is in a certain sense the inverse: elimination of bounds in problems of linear programming (in the form of symmetric matrix games) by transforming variables without creation of auxiliary variables, and transfer to unconstrained minimization problem at the expense of a simple kind of non-linearity.

From the theoretical point of view, transforming variables means the transformation of the problem itself and not the elaboration of a new method of solving. We are trying to implement practically the same idea in the form of the software add-in that can transform problems of the form (1) into unconstrained problems. As a result, after transformation it appears possible to apply powerful unconstrained minimization solvers. Though, to reach the compatibility of the add-in with an algorithm, the latter should satisfy some design requirements. In the call of the algorithm, the following parameters should be passed: the stopping condition and names of one or more implementations (value, gradient, both) of the objective function. Parameters might have the form of a function or a function object. The existence of other parameters causes no difficulties. The algorithms implemented by us get a line search algorithm as one more parameter. Obviously, after transformation and solving of the problem, it is necessary to return to the original variables.

Issues related to the representation of the unconstrained minimization problem in the form, compatible to algorithm, are considered in Section 2.

The structure of the add-in and some related issues are considered in Section 3. Add-in, created in C++, consists of three components: transformations of variables which are formed as objects of the certain class and are piled in container; several test-problems with constraints only on lower bounds of variables, which are also formed as certain objects and gathered in a corresponding container; stopping conditions of minimization algorithms and program-driver.

The program-driver is responsible for choosing of the test-problem and transformation (of variables) in the interactive mode, for calling the unconstrained minimization algorithm and evaluation of the run-time, returning to the original variables and printing the results.

The add-in contains transformations of variables of the following four types: quadratic, quartic, exponential and sinusoidal. The quadratic transformation, for example, means that problem (1) is transformed to the unconstrained one via the transformation

$$x_i = l_i + t_i^2, \quad \forall i \in \mathcal{I}.$$

Adding other transformations to the add-in is a simple job.

The test-problems collection consists of 10 problems. Eight of them are taken from CUTest (see [11]). Two tests correspond to symmetric matrix games; in both cases a game is represented in the form of the minimization problem constrained only with lower bounds of the variables. In one case cubic function (with respect to original variables) is used and the other case- quadratic function.

To benchmark the add-in, two efficient algorithms of unconstrained optimization, l-bfgs (see [9]) and CG-DESCENT (see [7]), are used. CG-DESCENT-C-6.8 is taken from [8] and configured for Visual Studio (see [12]). l-bfgs is implemented by us and its design meets fully the requirements that arise when integrating the add-in into an unconstrained minimization algorithm. This implementation is relatively simple. The only modern technic is the line search procedure borrowed from [8]. But its design is object-oriented and partly generic. As a result, implementation is effective both for unconstrained minimization problems and for problems of the form (1).

The results of numerical experiments are considered in Section 4. MINOS with AMPL (see [13]) interface was used to debug the add-in and test-problems taken from CUTest. GUROBI (see [14]) with C++ interface was used to debug tests related to symmetric matrix games and to evaluate performance of the add-in. Results of experiments show that the use of unconstrained optimization algorithms for symmetric matrix games, even without pretreatment (taking into account specificity of the problem), gives impressive results. Add-in, integrated into l-bfgs, for problems constructed by randomly chosen numbers from [-1, 1] interval in most cases runs as fast as GUROBI. In some cases, when numbers are uniformly distributed, the results are even better. Using of CG-DESCENT-C-6.8 gives similar results.

The last Section 5 evaluates results, makes conclusions and considers the possible directions of future research.

Results of experiments based on codes uploaded at https://github.com/kobage are highlighting two main issues. Using unconstrained minimization algorithms via add-ins for problems, constrained only with lower bounds of variables, presented here, or similar add-ins are promising directions for future research. In the case of improvement and development of the presented here or similar add-ins, the future progress in the sphere of unconstrained minimization will have a direct impact on the progress in the area of symmetric matrix games and linear programming.

2. Some Aspects of Coding of Unconstrained Minimization Problems

Let us consider the problem of minimization of a smooth functional

$$\min f(x) \quad x \in \mathbb{R}^n,$$

with some numerical algorithm, which uses the first order derivatives. Consequently, the data needed to algorithm should be produced in the form of several functions or function objects.

For calling l-bfgs, two functions are needed. One of them takes the following parameters: the dimension of the problem and two vectors (for variable x and its gradient) by reference. It returns the value of the function at x and writes components of the gradient in the vector-parameter. This function is heavily used by l-bfgs and, therefore, it is highly recommended that its implementation takes into account each aspect related to performance speed (for example: reduce to minimum the number of calls of standard functions for simple actions, like the exponentiation, etc.). The second function is for one-time use. It sets initial values of the variable. Adequately, it receives the variable and the dimension as parameters.

To get faster code, the algorithm should distinguish the following cases: only the gradient is used; only the value is used; both are used. Consequently, it is necessary to use two more multiple-use functions. One of them will return only the function's value but will not touch the gradient. The other will not return anything, but calculates and stores the gradient. CG-DESCENT algorithm acts in this way.

In our implementations, the special structure is made for test problems. The structure of objects is storing certain information about the objective function: name, dimension, and names of the above-mentioned two functions (calculating gradient and value and initialising variables). The data of test problems are stored in containers of the corresponding type.

The container of test functions of unconstrained minimization is not a part of the add-in. Neither codes of objective functions of the test problems. They are intended to test unconstrained minimization algorithms and are enclosed to both directories (with prefix LB denoting the lower bound) uploaded at GitHub. Unconstrained minimization algorithms implemented by us have some characteristics in common. They have similar design (to respond to certain requirements) and have the same collections of test-problems. These 44 tests are chosen from the 145, which were used in [7]. But they were used with higher dimensions. The-above mentioned GitHub web-page contains some material, uploaded earlier. The Test-problems collection is among them. The recently uploaded collection of test-problems is essentially refreshed. Implementations of many functions are improved, noticed drawbacks are corrected.

It is worth to note that the performance of the CG-DESCENT-C-6.8 became more efficient on the improved test-problems, than performance of the l-bfgs. This is related to the fact that l-bfgs needs less iterations, than CG-DESCENT. Because of the same reason, l-bfgs is more effective on symmetric matrix games, where calculations of values and gradients are time-consuming.

3. The Structure of the add-in and Issue of its Integration into Unconstrained Optimization Algorithms

The minimization problem with the constraints only on lower bounds of variables contains more data, than unconstrained one, so it has different representation.

After transformations of variables, it is necessary to collect data of the transformed problem. The main challenge lies in efficient implementation of derivatives of the composite function. After solving the transformed problem, it is necessary to return to the original variables.

In mathematics, transformation of variables is a natural and simple task. In order for its equivalent in software also to be natural and simple, it is necessary to elaborate some implementation related separate details:

- for the given test-problem, replacing one transformation of variables with another should be simply feasible;
- adding new transformation of variables into the add-in should be simply feasible;
- adding new test-problems into the add-in should be simply feasible; the structure of test-problems should not be complicated;
- test-problems with different content have different stopping conditions. Therefore, minimization algorithms should allow changing of stopping conditions and accept them as parameters.

The Add-in is implemented in C++ and consists of three components: transformations of variables, which are presented as objects of certain class and are gathered in a container; several minimization test-problems with constraints only on lower bounds of variables, also presented as certain objects and gathered in the corresponding container; stopping conditions of algorithms and program-driver. The program-driver provides choosing of a test-problem and a transformation in the interactive mode, calling unconstrained minimization algorithm, evaluation of the performance time, returning to the original variables and printing the results.

The add-in contains transformations of four types: quadratic: $x_i = l_i + t_i^2$, $\forall i \in \mathcal{I}$; quartic: $x_i = l_i + t_i^4$, $\forall i \in \mathcal{I}$; exponential: $x_i = l_i + e^{t_i}$, $\forall i \in \mathcal{I}$; sinusoidal: $x_i = l_i + 1 + \sin(t_i)$, $\forall i \in \mathcal{I}$. The exponential transformation is not completely correct, because x_i never becomes equal to l_i.

Sinusoidal transformation is correct for test-problems, corresponding to symmetric matrix games, because variables represents probabilities in this case. Besides these four types, it is very simple to add another transformations into the add-in. It should be noted that we are using uniform transformations, the same for each coordinate. Quadratic and sinusoidal transformations, which are the most efficient in numerical experiments, are not invertible. Theoretically, this fact may lead to the emergence of false critical points (at which derivative of the composite function is equal to zero). Practically, this fact causes no problems - because of rounding errors, the possibility of coinciding of a double precision real number and zero is extremely small. The software equivalent of the transformation of variables is an aggregation of three functions: a transformation, its inverse (or its branch) and its derivative

```
double quadraticSubstitution(double v){
        return v*v;
}
double quadraticSubstitutionPrime(double v){
        return 2 * v;
}
double inverseQuadraticSubstitution(double v){
        return sqrt(v);
}
```

In order to store the transformation, we make such a structure that its objects represent software equivalent of transformations:

```
struct lb_substiturtion
{
    double(*substitution) (double);
    double(*inverseSubstitution)(double);
    double(*substitutionPrime)(double);
    string substitutionName;
    void show(void)
    {
         std::cout << "Substitution:  " << substitutionName << std::endl;
    }
};
```

In the following vector

$$\text{vector} < \text{lb\_substiturtion} > \text{lb\_substiturtionsVector};$$

transformations are added in the standard way. After variables transformation, the objective function becomes a composite function. In C++, implementation of the objective function (or function object) should have access to the transformation. Let us consider one possible approach, simple and yet flexible.

When working with a collection of test-problems, the same objective function may appear both in unconstrained and in constrained problems. In this case, it is convenient to obtain the representation of the constrained problem based on the unconstrained one. Let us consider the functional “hatflda”: $hatflda\,(x) = (x_1 - 1)^2 + \sum\limits_{i=2}^{4} \left(x_{i-1} - \sqrt{x_i}\right)^2$. All four variables are constrained with lower bounds: lBounds[i] = 0.0000001. We store the representation of this function with the prototype

```
double hatflda
(
    double *x,
    double *g,
    const int k
);
```

Its implementation is trivial, and we omit it. This representation is suited for the unconstrained minimization. But on its base we construct a composite function which is connected to the transformation via the function pointers:

```
double hatfldaComposite
(
    double *x,
    double *g,
    const int k
)
{
    for (int i = 0; i < k; i++)
       y[i] = lBounds[i] + substitutionName(x[i]);

    double fx = hatflda(y, g, n);
    for (int i = 1; i < k; i++)
           g[i] *= substitutionPrimeName(x[i]);
    return fx;
}
```

Here, substitutionName($x[i]$) is a pointer to the transformation, and substitutionPrimeName($x[i]$) is a pointer to its derivative. Of course, these addresses should be initialized before minimization algorithm will be run. For quadratic transformation, for example, this can be done with following assignments:

substitutionName = quadraticSubstitution; substitutionPrimeName = quadraticSubstitutionPrime;

The other possibilities also existed. For example, to attach test-problem, corresponding to symmetric matrix games, to CG-DESCENT-C-6.8, we used less elegant approach.

The representation of minimization problem (1) needs some additional code in the function, initializing the starting iterate. This portion of the code makes lower bounds and indices of variables with constraints. One more one-time function provides returning to the original variables.

The described add-in was initially debugged with the Modified Heavy Ball algorithm (see [5]) due to the following simple factor: the code of this algorithm consists of 20 lines which makes experiments using it extremely simple. At the same time, it is fast enough. In the debugging and development process, we settled on the design of the algorithms l-bfgs_LB and MHB_LB (see materials uploaded to GitHub). The add-in is fully adapted to these two algorithms. For CG-DESCENT-C-6.8, only the part, corresponding to symmetric matrix games, is adapted. At first, this emphasizes the importance of symmetric games for us. Secondly, we tried to interfere with the code of CG-DESCENT-C-6.8 as little as possible.

Integration of the add-in into the algorithm requires weak support from the algorithm which is expressed in the possibility of variation of the stopping condition. In the ideal case, a stopping condition should be one of the parameters of the algorithm. The above-mentioned is arranged in algorithms l-bfgs_LB and MHB_LB. This possibility is not defined explicitly in CG-DESCENT-C-6.8, but its structure allows users to consider implicitly the stopping condition as a parameter, using function pointers.

4. Numerical Experiments

In the numerical experiments, two groups of test-problems are used. One group consists of 8 test-problems taken from CUTest. It mainly is used to debug the add-in. MINOS with AMPL interface was used to debug the add-in, to check correctness of results and to evaluate performance characteristics of the add-in on these test-problems. Three different transformations are applicable to each test-problem (to some problems, sinusoidal transformation is applicable as well). Therefore, the number of test-problems is, really, greater. The add-in works reliably. Few exceptions take place in the case of exponential transformation which could be expected.

GUROBI with C++ interface was used to debug the add-in on symmetric matrix games and evaluate performance.

Results of experiments show that application of unconstrained optimization algorithms through the add-in for the problems, constrained only with lower bounds of variables, is fully competitive. In particular, this concerns the theme of matrix games. Formally, only two such test-problems are used. But each of them can be considered with 4 transformations of variables. Moreover, 3 different scenarios can be used to generate random data:

```
std::uniform_real_distribution<double>
std::normal_distribution<double>
std::cauchy_distribution<double>
```

and during generating the objects of distributions, we are able to vary ranges of random numbers. For example, std::cauchy_distribution $<$ double $>$ urdi$(-1, 1)$;

All this allows us to get a fairly complete picture of the perspective of the usage of unconstrained minimization algorithms to solve symmetric matrix games.

Results of experiments show that using integrated into l-bfgs add-in with quadratic and sinusoidal transformations in most cases solves symmetric matrix games, constructed with random numbers from [-1,1], as fast as GUROBI solves problems of linear programming correspondig to the same matrix games. In some cases, when numbers are uniformly distributed, the results are even better. Using of CG-DESCENT-C-6.8 gives similar results. The advantage of GUROBI appears and increases with increasing range of random numbers.

The main goal of this paper is to present the results of developing the add-in and evaluating its performance. Symmetric matrix games represent one tool in this process. The results of numerical experiments demonstrate that matrix games and, more generally, linear programming should be among the key topics of our future research. Because of this, let us shortly consider the point of representation of symmetric matrix games as problems of unconstrained minimization.

Consider a symmetric matrix game, i.e., a game with an $n \times n$ skew matrix $A = (a_{i\,j})$:

$$a_{i\,j} = -a_{j\,i}, \quad \forall i, j \in \{1, 2, \ldots, n\}\,.$$

The price of such a game is equal to zero (see [2]), hence solving the game is equivalent to solving the following system of inequalities in $\mathbb{R}^n$:

$$\begin{cases} A_1 x \leq 0, \dots, A_n x \leq 0, \\ x_1 + \cdots + x_n = 1, \\ x_1 \geq 0, \dots, x_n \geq 0, \end{cases} \tag{2}$$

where the i-th row is denoted by A_i and $x = (x_1, \dots, x_n)^T$ is the unknown vector. This problem always has a not necessarily unique solution. If $\tilde{x} = (\tilde{x}_1, \dots, \tilde{x}_n)^T$ is the solution of (2), then for some indices, say i, $A_i \tilde{x} = 0$ is fulfilled.

The main problem in numerical solution of symmetric game lies in the third row of (2). To avoid combinatorial approach, the transformation of variables can be used. From the above-mentioned four transformations, the quadratic and sinusoidal transformations are equally effective. For simplicity, let us consider the quadratic transformation

$$x_i = t_i^2, \quad i \in \{1, \dots, n\}.$$

Now, (2) is equivalent to each of the following two unconstrained minimization problems:

$$\sum_{i=1}^{n} (A_i t \circ t < 0)?0 : (A_i t \circ t)^3 + \left| \left(\sum_{i=1}^{n} t_i^2 - 1 \right)^3 \right| \to \min, \tag{3}$$

which represents test-problem -9 of the add-in,

$$\sum_{i=1}^{n} (A_i t \circ t < 0)?0 : (A_i t \circ t)^2 + \left(\sum_{i=1}^{n} t_i^2 - 1 \right)^2 \to \min, \tag{4}$$

which represents test-problem-10. In (3) and (4), for arbitrarily taken $t = (t_1, \dots, t_n)^T$ and $s = (s_1, \dots, s_n)^T$ vectors "$\circ$" operator denotes their Hadamard product $s \circ t \equiv (s_1 t_1, \dots, s_n t_n)^T$, whilst operator "?" has the following format:

$$\text{(condition) ? (result 1) : (result 2)} \tag{5}$$

if the condition is true, then (5) equals (result 1), or (result 2) if the condition is false.

For $m \times n$ real matrix $A = (a_{ij})$, when $m \neq n$, the expression $Ax \circ x$ uniquely means the same as $A(x \circ x)$, because vectors Ax and x have different dimensions and thus their Hadamard product is not defined. Consequently, we are using the denotation $Ax \circ x$ and consider that priority of the Hadamard product is higher, than priority of matrix-vector product.

Let us note that (3) and (4) do not represent penalty functions. Because of variables' transformation, these functions are not convex. Nevertheless, they are smooth enough for minimization algorithms to work efficiently (via LB-add-in). Note also that in case of high dimensions, (3) and (4) are equally effective.

5. Conclusions

The structure of the LB-add-in is simple and flexible. It is easily joinable to unconstrained minimization algorithms; it does not matter if an algorithm has special design or it is unaware of the add-in (as in the case of CG-DESCENT-C-6.8). The add-in itself (excluding transformations and collection of test-problems) is fairly compact and in case of necessity, its implementation in other high level programming languages or developing its parallel versions is a very realistic challenge.

The add-ins presented here are aimed at problems, constrained only with lower bounds of variables. But it can be extended to box-constrained problems, as well. Suitable transformations are described in [1] and technical aspects are developed in the presented paper. Much harder part consists of benchmarking the add-in, which transforms box-constrained problems into unconstrained ones. In order to carry out a complete study, vast infrastructure (algorithms, benchmarking environment, test-problems collection) is needed (see [6]).

Intensive experiments were carried out on symmetric matrix games using the add-in. Interest in matrix games is reinforced by the fact that the canonical form of linear programming is easily reducible to symmetric matrix games. Based on the experiments, we can conclude that if we further develop the approach presented here, the future progress in the sphere of unconstrained minimization will have a direct influence on the progress in the area of symmetric matrix games and linear programming.

Experiments have also showed some weaknesses in the usage of unconstrained minimization algorithms to symmetric matrix games in case of absence of specific-based improvements. It is possible to use various heuristics as well as other themes that are obligatory in design of high-quality solvers. There is a wide choice of solvers in linear programming and among them GUROBI is one of the best. If we make comparisons using some open-source solver, the relative results of the add-in would be (see [4]) much better.

References

1. M. J. Box, A Comparison of Several Current Optimization Methods, and the use of Transformations in Constrained Problems. *The Computer Journal* **9** (1966), no. 1, 67–77.
2. G. W. Brown, J. von Neumann, Solutions of Games by Differential Equations. *Contributions to the Theory of Games*, pp. 73–79. Annals of Mathematics Studies, no. 24. Princeton University Press, Princeton, N. J. 1950.
3. D. Chen, R. J. Plemmons, Nonnegativity Constraints in Numerical Analysis. *The birth of numerical analysis*, 109–139, World Sci. Publ., Hackensack, NJ, 2010.
4. J. L. Gearhart, K. L. Adair, J. D. Durfee, K. A. Jones, N. Martin, R. J. Detry, Comparison of open-source linear programming solvers. *Sandia National Laboratories, SAND2 013-8847* (2013).
5. K. Gelashvili, I. Khutsishvili, L. Gorgadze, L. Alkhazishvili, Speeding up the Convergence of the Polyak's Heavy Ball Algorithm. *Trans. A. Razmadze Math. Inst.* **172** (2018), no. 2, 176–188.
6. W. W. Hager, H. Zhang, Recent Advances in Bound Constrained Optimization. *System modeling and optimization*, 67–82, IFIP Int. Fed. Inf. Process., 199, Springer, New York, 2006.
7. W. W. Hager, H. Zhang, The limited memory conjugate gradient method. *SIAM J. Optim.* **23** (2013), no. 4, 2150–2168.
8. W. W. Hager, H. Zhang, Source Code for CG_DESCENT Version 6.8 (C and Matlab code). March 7, 2015. https://www.math.lsu.edu/hozhang/Software.html.
9. Dong C. Liu, Jorge Nocedal, On the limited memory BFGS method for large scale optimization. *Math. Programming* **45** (1989), no. 3 (Ser. B), 503–528.
10. Ciyou Zhu, Richard H. Byrd, Lu Peihuang, Jorge Nocedal, Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Software* **23** (1997), no. 4, 550–560.
11. Cute Set (AMPL models). http://orfe.princeton.edu/ rvdb/ampl/nlmodels/cute/index.html.
12. Visual Studio Enterprise. See: https://beta.visualstudio.com/visual-studio-enterprise-vs.
13. MINOS for AMPL. https://ampl.com/products/solvers/solvers-we-sell/minos.
14. GUROBI Optimization. http://www.gurobi.com.

(Received 02.10.2018)

School of Business, Computing and Social Sciences, St. Andrew the First-Called Georgian University of the Patriarchate of Georgia, 53a Chavchavadze Ave., 0179 Tbilisi, Georgia

E-mail address: koba.gelashvili@sangu.edu.ge