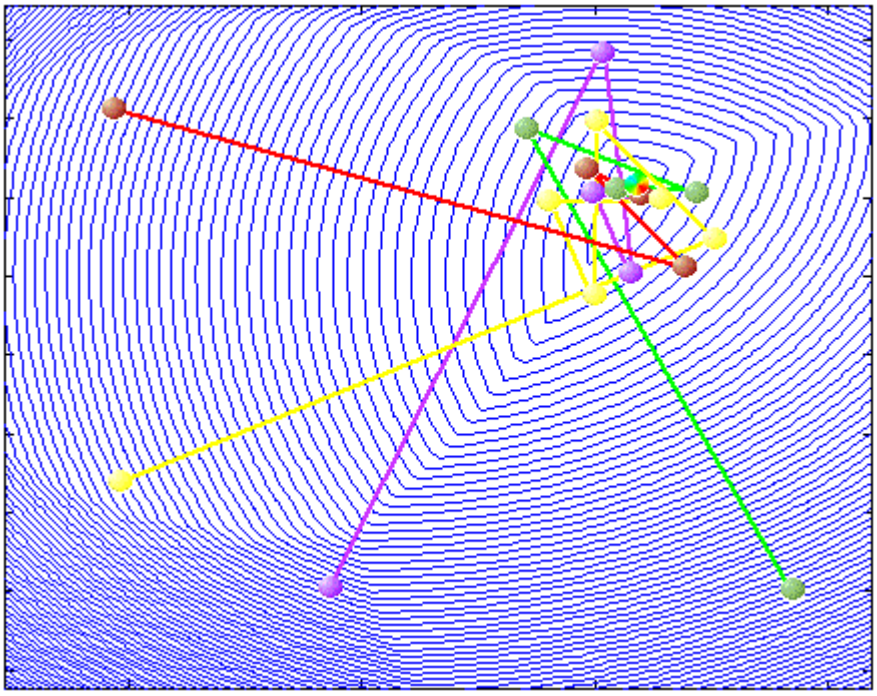
SolvOpt

The Solver For Local Nonlinear Optimization Problems

Matlab®, C and Fortran Source Codes



Alexei Kuntsevich
Franz Kappel

SolvOpt

version 1.1

is the Matlab, C and Fortran source codes

for solving nonlinear (nonsmooth) optimization problems.

SolvOpt is a freeware and comes with no warranty.

Institute for Mathematics

Karl-Franzens University of Graz

June, 1997

Contents

What is new in version 1.1

The new features of SolvOpt 1.1 are:

- the program itself calculates a suitable penalty coefficient for a set of constraints, however, still no information about Lagrange multipliers is computed,
- warning messages come every time when the program detects an irregular (complicated) case, so the user is aware of possible failures,
- the routine does not stop at a point, where no function value is available or it equals infinity, or gradient is zero, unless it fails to find a way to pass through a "bad" area,
- the code is better adapted to minimization of badly scaled, almost "flat" and extremely "steep" functions,
- a new procedure for approximating the gradients by the finite differences provides more robustness in the case, when analytically calculated gradients are not available,
- the manual is completed by including the chapters specific for a particular programming language,
- every language specific distribution package contains sample problem files illustrating minimization of a differentiable and nonsmooth objective function and solving constrained nonlinear programming problem with the use of either the intrinsic procedure for calculating a suitable penalty coefficient or a penalty coefficient given a priori by the user.

The results illustrating the performance of the program are presented on the web, but not given in this manual.

Fortran-77 version is no more supported.

The M-function `solvplot.m` for the mesh, contour and solution path plottings is no more supplied.

1 Tutorial

The program SolvOpt (*Solv*er for local *opt*imization problems) is concerned with minimization or maximization of nonlinear, possibly non-smooth objective functions and with the solution of nonlinear programming problems taking into account constraints by the so-called method of exact penalization.

Type	Problem
Unconstrained Optimization	$\min\{f(x): x \in \mathbb{R}^n\}$ (MIN) $\max\{f(x): x \in \mathbb{R}^n\}$ (MAX)
Constrained Minimization	$\min\{f(x): g(x) \le 0, h(x) = 0, x \in \mathbb{R}^n, g(x) \in \mathbb{R}^m, h(x) \in \mathbb{R}^l\}$ (NLP)

The program `solvopt` requires that the user supplies at least a routine which computes the values of the objective function at a given point. Unless the user also provides a code for computing the (sub)gradient of the objective function, these (sub)gradients are calculated by the program itself using finite differences.
For a constrained minimization problem, the user is required to supply additionally at least a routine which computes the maximal residual for a set of constraints at a given point. By analogy with the case of unconstrained optimization, the user may also provide a code for computing the (sub)gradient of a constraint function with the maximal residual at a point.

Remark. SolvOpt provides a general optimization tool applicable for a wide class of nonlinear optimization problems. However, it seems useless to apply it for solving linear and quadratic programming problems. The other optimization packages, which are particularly oriented on solution of these problems, essentially use the features of a specific class of problems and provide a higher efficiency.

Installation

Matlab

Instructions for installing toolboxes are found in the section entitled "Installing Toolboxes" in the computer-specific section of the *MATLAB User's Guide*.

With MATLAB 4.2 for MS Windows™ one has to modify the file **`MATLABRC.M`** found in the MATLAB root directory as follows. Find the section starting with

```
matlabpath([...
```

and add the lines

```
'c:\matlab\toolbox\solvopt;',...
'c:\matlab\toolbox\solvopt\uncprobs;',...
```

With MATLAB 4.2 for HP Unix™ one has to create the folders

`/users/me/`**`solvopt`**

`/users/me/`**`solvopt/uncprobs`**

copy the SolvOpt files to these folders, modify the **`.cshrc`** file in your root folder by additing the two lines

```
setenv MATLABPATH /users/me/solvopt/Matlab
setenv MATLABPATH /users/alex/solvopt/Matlab/uncprobs
```

FORTRAN

The **MS FORTRAN PowerStation 1.0** users must store the source files to a single folder and create a new project by including the program files and one of those sample problem files supplied, build an .EXE file and run a test.

With the aim to solve a new problem, edit the project by substituting a sample problem file with the problem file. Then rebuild the project and run the executable file.

One can find comprehensive instructions on building MS Fortran projects in the MS FORTRAN PowerStation 1.0 help library.

The **HP FORTRAN-90** users must store the source files to a single folder and **make** an executable sample problem file. You might need to edit the supplied **`makefile`,** if the standard FORTRAN-90 libraries are named or stored differently.

C

The **MS Visual C** users must store the source files to a single folder and create a new project by including the program files and one of those sample problem files supplied, build an .EXE file and run a test.

With the aim to solve a new problem, edit the project by substituting a sample problem file with the problem file. Then rebuild the project and run the executable file.

One can find comprehensive instructions on building MS Visual C projects in the MS VC help library.

The **HP C** users must store the source files to a single folder and **make** an executable sample problem file. You might need to edit the supplied **`makefile`,** if the standard C libraries are named or stored differently.

The **Linux C** users must install the source library according to the Linux specifications and comment out the line

```
#include <math.h>
```

in all supplied source files.

There is no sample `makefile` specific for Linux, so you have to create a one yourself.

Examples

In this section, we explain how to use SolvOpt by its application to the sample problems. We do this separately for every programming language used. Do not be confused with the fact that the Matlab, FORTRAN and C versions of the solver, which are actually identical, produce slightly different results. This is caused by the differences in the way the floating point operations are performed.

Matlab sample functions

In this section we explain how to use the M-function `solvopt` closely following the first demonstration in the M-file `solvdemo`.

Unconstrained minimization

Problem 1. Shor's piece-wise quadratic function

Find

$$\min\left\{f(x): x \in \mathbb{R}^n\right\},$$

where

$f(x) = \max\left\{\varphi_i(x):\ i = 1,\ldots,m\right\}$, $\varphi_i(x) = b_i \sum_{j=1}^{n} (x_j - a_{ij})^2$,

$x = (x_1,\ldots,x_n)$, $b = (b_i)$ is a given m-vector and $A = (a_{ij})$, $i = 1,\ldots,m$, $j = 1,\ldots,n$, is a given $m \times n$ matrix.

To solve this problem with $n = 5$ and $m = 10$ for a particular choice for b and A write an M-file that returns the function value at a point.

Solution. ***Step 1.*** *Write the following M-file:*

```
function f=shorf(x)
a=[ 0,  2,  1,  1,  3,  0,  1,  1,  0,  1;...
    0,  1,  2,  4,  2,  2,  1,  0,  0,  1;...
    0,  1,  1,  1,  1,  1,  1,  1,  2,  2;...
    0,  1,  1,  2,  0,  0,  1,  2,  1,  0;...
    0,  3,  2,  2,  1,  1,  1,  1,  0,  0];
b=[ 1; 5; 10; 2; 4; 3; 1.7; 2.5; 6; 4.5];
```

```
f=0;
 for i=1:10
    d=b(i)*sum((x-a(1:5,i)).^2);
    if d>f;f=d;end
 end
```

Next, invoke the routine `solvopt`.

Solution. ***Step2.*** *Invoke the optimization routine:*

```
x=[-1;1;-1;1;-1]; % Starting Point
[x,f]=solvopt(x,'shorf')
```

After 515 function evaluations, this produces the solution:

```
x =
  1.12437450703594
  0.979399386605537
  1.47744316500768
  0.920360602026234
  1.12420844022117
f =
  22.6001638723023
```

User-supplied gradients

SolvOpt calculates gradients by finite difference approximation (see `apprgrdn.m` in the *Reference* part). The other possibility is to supply analytically calculated partial derivatives.

To solve Problem 1 using analytically calculated gradients proceed as follows.

Solution.

Step 1. *Write an M-file* `shorf` *that supplies the value of the objective function at a given point (see above)*

Step 2. *Write an M-file that supplies the gradient of the objective function at a given point:*

```
function g=shorg(x)
a=[ 0,  2,  1,  1,  3,  0,  1,  1,  0,  1;...
    0,  1,  2,  4,  2,  2,  1,  0,  0,  1;...
    0,  1,  1,  1,  1,  1,  1,  1,  2,  2;...
```

```
    0,  1,  1,  2,  0,  0,  1,  2,  1,  0;...
    0,  3,  2,  2,  1,  1,  1,  1,  0,  0];
b=[ 1; 5; 10; 2; 4; 3; 1.7; 2.5; 6; 4.5];
f=0;
  for i=1:10
     d=b(i)*sum((x-a(1:5,i)).^2);
     if d>f;f=d;k=i;end
 end
 g=b(k)*.5*(x-a(1:5,k));
```

Next, invoke the routine `solvopt`.

Solution. ***Step3.*** *Invoke the optimization routine:*

```
x=[-1;1;-1;1;-1]; % Starting Point
[x,f]=solvopt(x,'shorf','shorg')
```

After 176 function evaluations and 59 gradient evaluations, this produces the solution:

```
x =
  1.12434198921972
  0.979451962903515
  1.47764128967286
  0.920240641355049
  1.12427862238767
f =
  22.6001630849795
```

The sample problem M-files may be structured differently. One can use an initialization function for setting the constants specific for the problem. These constants should be declared as global in the three functions, the initialization routine, the one returning the function value and the one calculating the gradient. This way, the global parameters become available from the Matlab work space after invoking the initialization function. The next section provides an example of the use of global parameters.

Constrained minimization

Consider now a constrained optimization problem and the way it can be reduced to the minimization of a non-smooth penalty function. With the first example, we consider the case, when a penalty coefficient is chosen a priori by the user. In the second one, the intrinsic adjusting procedure for a penalty coefficient is used.

Minimization of a penalty function with a penalty coefficient given by the user

If you doubt on a value of a penalty coefficient, it is better to let the program itself find a suitable one. However, there are some constrained problems, for which the exact penalty coefficients are known. The following example illustrates the solution of a one.

Problem 2. Ill-Conditioned Linear Programming Problem [Ki85]
Find

$$\min\left\{c^T x:\ x \in \mathbb{R}^n\right\}$$

subject to

$$Ax \le b\,, \quad x \ge 0\,,$$

where

$$A = \left(a_{ij}\right), \quad a_{ij} = 1/(i+j)\,, \quad i,j = 1,\ldots,n,$$

$$b_i = \sum_{j=1}^{n} 1/(i+j), \quad i = 1,\ldots,n,$$

$$c_i = -1/(i+1) - \sum_{j=1}^{n} 1/(i+j), \quad i = 1,\ldots,n\,.$$

This problem is ill-conditioned for $n \ge 5$, since A is essentially a part of Hilbert's matrix. The problem can be solved by minimization of the exact penalty function

$$c^T x + r\max\left(0,-x,Ax-b\right)$$

where r is the penalty coefficient.

For Problem 2, a suitable value of the penalty coefficient ($r = 2n$) is known. Therefore, it is worth to use this value explicitly.

Solution.

Step 1. *Write an M-file that sets the values for the constant vectors and matrices:*

```
function x=initill(n)
% INITILL sets the values for n*n matrix A and n-dimensional vectors B and C
global matrA vectB vectC
vectB=[]; vectC=[]; matrA=[];
  for i=1:n,
    x(i)=0;    vectB(i)=0;
    for j=1:n
       matrA(i,j)=1/(i+j);  vectB(i)=vectB(i)+1/(i+j);
    end
    vectC(i)=-1/(i+1)-vectB(i);
  end
  vectB=vectB';  vectC=vectC';  x=x';
```

Step 2. *Write an M-file that supplies the exact penalty function value at a point:*

```
function f=illclinf(x)
% The function ILLCLINF returns value of the exact penalty function
% at a point for the Ill-conditioned Linear Programing problem.
global matrA vectB vectC
if size(x,2)>1, x=x'; end, n=size(x,1);
f=vectC'*x + 2*n*max([0;matrA*x-vectB;-x]);
```

Step 3. *Invoke the optimization routine:*

```
x=initdual; % Get the starting point and set the globals
[x,f]=solvopt(x,'illclinf')
```

After 687 function evaluations, this produces the solution:

```
x =
  1.00009381952471
  0.99952814062634
  1.00068930581739
  0.999287744917841
  1.00050586697905
  1.0000189905597
  1.00054017839116
  1.00067368900695
  0.999251893713646
  0.999428402984036
  0.998217364046941
  1.00066418869783
```

```
  1.00070780745505
  0.999440830439142
  1.0009920171358
f =
  -20.0420010545132
```

Solution of a constrained problem by use of the exact penalty function

In this subsection, we consider a sample constraint problem and the way it can be solved by exact penalty function method and with the use of SolvOpt intrinsic procedure for adjusting a penalty coefficient.

The following general conditions are to be fulfilled:

- the problem is to be formulated in the standard form (NLP):

$$\min\left\{f(x): g(x) \le 0, h(x) = 0, x \in \mathbb{R}^n, g(x) \in \mathbb{R}^m, h(x) \in \mathbb{R}^l\right\},$$

- the maximal residual for a set of constraints at a point x is calculated as

$$\max\left\{\ \max_{i=1,\dots,m}\left[\max\left(0, g_i(x)\right)\right],\ \max_{j=1,\dots,l}\left|h_j(x)\right|\right\}, \qquad \text{(RES)}$$

- the gradient (if supplied) of a constraint function at a point is the gradient of a constraint function with the maximal residual (RES) at a point.

Problem 3. Shell Dual Problem

Find

$$\min\left\{f_0(x): x \in \mathbb{R}^n\right\}$$

subject to

$$f_i(x) \le 0, \quad i = 1,\dots,m,$$
$$x_j \ge 0, \quad j = 1,\dots,n,$$

where $n = 15$, $m = 5$, $x = \begin{pmatrix} y \\ z \end{pmatrix}$, $y \in \mathbb{R}^5$, $z \in \mathbb{R}^{10}$.

$$f_0(x) = 2d^T y^3 + y^T C y - b^T z,$$
$$f_i(x) = (Az)_i - 2(Cy)_i - 3d^T y^3 - e_i, \quad i = 1,\dots,5,$$

$d, e \in \mathbb{R}^5$ resp. $b \in \mathbb{R}^{10}$ are given vectors, and C is a given 5×5 -matrix and A is a given 5×10 -matrix. The given constrained problem can be solved by minimizing the exact penalty function

$$f_p = f_0(x) + r \max\left(0, -x_1, \ldots, -x_n, f_1(x), \ldots, f_m(x)\right),$$

where r is the penalty coefficient.

We shall consider Problem 3 with the following data:

$A^T =$

```
   -16    2   0   1   0
     0   -2   0  .4   2
  -3.5   0   2   0   0
     0   -2   0  -4  -1
     0   -9  -2   1  -2.8
     2    0  -4   0   0
    -1   -1  -1  -1  -1
    -1   -2  -3  -2  -1
     1    2   3   4   5
     1    1   1   1   1
```

$b =$

```
  -40
   -2
   -0.25
   -4
   -4
   -1
  -40
  -60
    5
    1
```

$C =$

```
   30  -20  -10   32  -10
  -20   39   -6  -31   32
  -10   -6   10   -6  -10
   32  -31   -6   39  -20
  -10   32  -10  -20   30
```

$d =$

```
    4
    8
   10
    6
    2
```

$e =$

```
  -15
  -27
  -36
  -18
  -12
```

Solution.

Step 1. *Write an M-file that sets the values for the constant vectors and matrices:*

```
function x=initdual()
global A B C D E
A= [ -16,  2,  0,  1,  0;...
       0, -2,  0, .4,  2;...
    -3.5,  0,  2,  0,  0;...
       0, -2,  0, -4, -1;...
       0, -9, -2,  1, -2.8;...
       2,  0, -4,  0,  0;...
      -1, -1, -1, -1, -1;...
      -1, -2, -3, -2, -1;...
```

```
       1,  2,  3,  4,  5;...
       1,  1,  1,  1,  1];
B= [-40; -2; -.25; -4; -4; -1; -40; -60; 5; 1];
C= [ 30, -20, -10,  32, -10;...
    -20,  39,  -6, -31,  32;...
    -10,  -6,  10,  -6, -10;...
     32, -31,  -6,  39, -20;...
    -10,  32, -10, -20,  30];
D= [ 4; 8; 10; 6; 2];
E= [ -15; -27; -36; -18; -12];
x= [1.e-4;1.e-4;1.e-4;1.e-4;1.e-4;...
    1.e-4;1.e-4;1.e-4;1.e-4;1.e-4;...
    1.e-4;   60;1.e-4;1.e-4;1.e-4];
```

Step 2. *Write an M-file that supplies the objective function value at a point:*

```
function f=dsobjf(x)
global A B C D E
x=x(:);
f=2*D'*x(1:5).^3 + (C*x(1:5))'*x(1:5) - B'*x(6:15);
```

Step 3. *Write an M-file that supplies the gradient of the objective function at a point:*

```
function g=dsobjg(x)
global A B C D E
x=x(:);g=x;
g(1:5)=6*D.*x(1:5).^2 + 2*C*x(1:5);
g(6:15)= - B;
```

Step 4. *Write an M-file that supplies the MAXIMAL RESIDUAL for the set of constraints at a point:*

```
function f=dscntf(x)
global A B C D E
x=x(:);
f=max([A'*x(6:15)-2*C*x(1:5)-3*D.*x(1:5).^2-E; -x]);
```

Step 5. *Write an M-file that supplies the gradient of a constraint with the MAXIMAL RESIDUAL at a point:*

```
function g=dscntg(x)
global A B C D E
x=x(:); g=zeros(size(x));
```

```
[f,k]=max([A'*x(6:15)-2*C*x(1:5)-3*D.*x(1:5).^2-E; -x]);
if f>0,
    if k>5, g(k-5)=-1;
    else,   g(6:15)=A(:,k);
            g(1:5)=-2*C(:,k);
            g(k)=g(k)-6*D(k)*x(k);
    end
end
```

Step 6. *Invoke the optimization routine:*

```
x=initdual; %Get the starting point and set the globals
[x,f]=solvopt(x,'dsobjf','dsobjg',[],'dscntf','dscntg')
```

After 1047 function evaluations and 296 gradient evaluations, this produces the solution:

```
x =
  0.300278569534156
  0.333218485705157
  0.400287704078501
  0.428099403397778
  0.224208296195057
  2.11686313222728e-009
  5.38814286825384e-008
  5.17010727665834
  9.38425731472533e-008
  3.06211609161864
 11.8350264103085
 -4.8754715160715e-009*
 -5.33007731972345e-009*
  0.103743920438453
  3.49071860248073e-007
f =
  32.3486841145684
```

* - The admissible upper bound on the maximal residual for a set of constraints is set to 10^{-8} by default. The optional parameters to the function `solvopt` and their default values are discussed in the section *The default parameter settings*.

Problem 3 can be solved without user-supplied analytical gradients by invoking the function `solvopt` as follows:

```
x=initdual; %Get the starting point and set the globals
```

```
[x,f]=solvopt(x,'dsobjf',[],[],'dscntf')
```

After 5516 function evaluations, this produces the solution:

```
x =
  0.300140731071454
  0.332763649376293
  0.400037600956474
  0.428112829612298
  0.224617359277728
  7.85469511969064e-009
  5.06335656775549e-007
  5.17065381492976
  1.74499313839886e-006
  3.06343775013392
 11.8385395874395
 -8.11639978175619e-009
  2.25878690707955e-007
  0.105231979218134
  5.85235333566764e-006
f =
  32.3487225349038
```

FORTRAN sample routines

Here we assume that MS FORTRAN 1.0 for Windows is used. Me assume this just to avoid using twice as many terms to describe what the user must do. Thus, if you find "(re)build the project", the UNIX users should understand this as "make the executable file". The instruction "edit the project" is the same as "edit the makefile" by doing whatever follows. The file extension `.f` is always used for HP F90 source files.

Unconstrained minimization

We shall consider Problem 1 (minimization of Shor's piece-wise quadratic function) with the same initial data as above.

The sample file **`shor.for`** contains the main program, the data initialization subroutine `shorinit` and the subroutines for calculating objective function values (`shorf`) and gradients (`shorg`). The main program sets the flags used, calls to the initialization subroutine and to the solver (subroutine `solvopt`) and prints the obtained solution. The subroutines `shorf` and `shorg` calculate the objective

function value, resp. the gradient, at a point. The problem data is passed to these routines from `shorinit` by use of the common block.

The flag **`flg`** controls the way the gradients are calculated. It is set to `.true.`, meaning the analytically calculated gradients are used. Turn it to `.false.` to use the finite difference approximation of the gradients.

To minimize this sample function start a new MS FORTRAN project and include the files `solvopt.for`, `shor.for`, `soptions.for` and `apprgrdn.for` to the project.

Build the project.

Note: The include file `messages.inc` must be stored in the same folder/directory as the file `solvopt.for`. The HP UNIX users must take care about the correct path to the HP F90 libraries. The provided sample **`makefile`** should be edited with this aim.

Run the project.

At the standard output screen, you will see the results of this run:

```
SolvOpt: Normal termination.

    Function Value ===== Evaluations == Iterations

  22.6001633489763        179 +   61        60


Optimum Point X:

  1.12439469387304
  .979431355218914
  1.47764382403059
  .920327774244919
  1.12425127557092
```

Constrained minimization

We shall consider Problem 3 (Shell Dual Problem) with the same initial data as above.

The sample files **`dualshl0.for`**, **`dualshl1.for`**, **`dualshl2.for`**, **`dualcnst.for`** and **`dualcnt2.for`** contain the codes providing different solution cases. With these files we illustrate

1. How to use common blocks (see the files `dualshl1.for` and `dualshl2.for`) or extra entries to a subroutine/function (see the other three files) to pass the parameters to the user's routines,
2. How to solve a constrained problem by use of the exact penalty method, when a suitable value for a penalty coefficient is known a priori (see the files `dualshl0.for`, `dualshl1.for` and `dualshl2.for`) and how to solve a constrained problem by use of the intrinsic features of the solver (see the other two files),
3. How to use analytically calculated gradients (see files `dualshl0.for`, `dualshl1.for` and `dualcnst.for`) and the finite difference approximation scheme (see the other two files).

All these sample files contain detailed comments and may be used as the patterns when coding a real optimization problem.

Let us start with the sample problem file `dualshl0.for`.

Start a new MS FORTRAN project and include the files `solvopt.for`, `dualshl0.for`, `soptions.for` and `apprgrdn.for` to the project.

Build the project and run it.

At the standard output screen, you will see the results of this run:

```
SolvOpt: Normal Termination

     Function Value ===== Evaluations == Iterations
   32.3486977724605       1106 +  318       317

 Optimum Point X:

   .299426825546335
   .333249409434943
   .400064007985528
   .428736534806681
   .224313069068149
   .223385131546603E-07
   .404749677131543E-06
   5.17763934177275
   .184116671220153E-06
   3.06147697520654
   11.8402600598917
   .243820289395243E-07
   .770361407720566E-08
   .104299707589751
   .223357529469471E-05
```

Next, edit the project by substituting the file `dualshl0.for` by the file `dualshl2.for`.

Rebuild and rerun the project.

This produces the following output:

```
SolvOpt: Normal Termination

     Function Value ===== Evaluations == Iterations
   32.3488027613418       6214 +    0       316

 Optimum Point X:
   .300652416460482
   .334052227663560
   .400675148395390
   .428306248210853
   .223898350862328
   .224165785297217E-07
   .879443464886512E-07
   5.16983078257772
   .344057985833308E-05
   3.05933154720805
   11.8358549467195
   .182872657231329E-07
   .668174738032022E-09
   .104270343767744
   .342688229519250E-06
```

Next, edit the project by substituting the file `dualshl2.for` by the file `dualcnst.for`.

Rebuild and rerun the project.

This produces the following output:

```
SolvOpt: Normal Termination

     Function Value ===== Evaluations == Iterations
   32.3486813349962       1076 +  308       306
                          1076 +  177

 Optimum Point X:
   .300324855074108
   .333292913854715
   .400173684256855
   .428061656040750
   .224106830081349
  -.290494843133336E-08
   .284518638024732E-07
```

```
   5.17038258440066
  -.807298523922867E-08
   3.06188088571973
   11.8364616839891
  -.801837606700664E-08
   .292773805446577E-09
   .103865591612960
   .143687435030575E-06
```

Next, edit the project by substituting the file `dualcnst.for` by the file `dualcnt2.for`.

Rebuild and rerun the project.

This produces the following output:

```
SolvOpt: Normal Termination

     Function Value ===== Evaluations == Iterations

   32.3488511257384       5827 +    0       294
                          3997 +    0

 Optimum Point X:
   .300043713568156
   .333294112865937
   .400890950005057
   .428629367367550
   .224695236327456
  -.680417714539980E-08
   .103801005551412E-04
   5.17113192186795
   .283637113183106E-06
   3.06147738310861
   11.8330115694190
  -.105153655627659E-08
   .657500293656519E-09
   .104665576720755
   .433417542039363E-06
```

Remark. Under the title "Evaluations", the values of the four returned counters are printed. In the first row, the numbers of objective (or penalty) function and gradient evaluations are given. In the second row, if it is printed, the numbers of constraint functions and gradient evaluations are given. These four runs illustrate that if the analytically calculated gradients are used (see the first and third runs), the solution is more precise then when making use of the gradient approximation by the finite differences.

C samples

Here we assume that MS Visual C is used. Me assume this just to avoid using twice as many terms to describe what the user must do. Thus, if you find "(re)build the project", the UNIX users should understand this as "make the executable file". The instruction "edit the project" is the same as "edit the makefile" by doing whatever follows.

Unconstrained minimization

We shall consider Problem 1 (minimization of Shor's piece-wise quadratic function) with the same initial data as above.

The sample file **`shor.c`** contains the main function and the functions for calculating objective function values and gradients. It starts with the include directives (see the note below). The user must include the header file **`solvopt.h`** to a function calling to the function `solvopt`.

Next, the program data is set to be available globally in the functions `shorf` and `shorg`. The main function follows these data specifications and is aimed to invoke the solver and to print the solution obtained. The functions `shorf` and `shorg` calculate the objective function value, resp. the gradient, at a point.

The parameter **`user_supplied_gradients`** controls the way the gradients are calculated. It is set to 1, meaning the analytically calculated gradients are used. Turn it to 0 to use the finite difference approximation of the gradients.

To minimize this sample function start a new MS VC project and include the three files `solvopt.c`, `apprgrdn.c` and `shor.c` to the project.

Build the project.

Note: Pay attention to the header files used. The HP UNIX users must take care about the correct path to the C libraries and header files. The provided sample **`makefile`** should be edited with this aim. The Linux users additionally must comment out the line **`#include <math.h>`** in all C source files supplied.

Run the project.

At the standard output screen, you will see the results of this run:

```
SolvOpt: Normal termination.
```

```
 Function Value =====  Evaluations  === Iterations
     22.6001633          179+   61          60

Optimum Point X:
      1.12439469
     0.979431355
      1.47764382
     0.920327774
      1.12425128
```

Constrained minimization

We shall consider Problem 3 (Shell Dual Problem) with the same initial data as above.

The sample file **`nlpsmple.c`** contains the main function and the functions for calculating objective function values and gradients, maximal residuals for a set of constraints and gradients for constraints. The main function is aimed to invoke the solver and to print the solution obtained.

The user may pass up to two parameters to the main function by typing their values in the command line. These parameters specify the way the gradients are calculated, analytically or by the finite differences, and the way the penalty coefficient is calculated, by taking the default value or by use of the solver intrinsic adjusting procedure. The first parameter (**`user_supplied_gradients`**) is set to 1 by default, meaning the analytically calculated gradients are used. Type 0 as the first value in the command line after the project name to use the finite difference approximation of the gradients. The default value of the second parameter (**`as_constrained_problem`**) is also 1, meaning the penalty coefficient is adjusted automatically. Type 0 as the second value in the command line after the project name to use the pre-defined penalty coefficient instead.

The functions `dsobjf` and `dsobjg` calculate the objective function value, resp. the gradient, at a point. If `as_constrained_problem=0`, these function calculate the values, resp. the gradients of the penalty function at the pre-defined value of the penalty coefficient.

The functions `dscntf` and `dscntg` calculate the maximal residual for a set of constraints, resp. the gradient of a constraint with the maximal residual, at a point.

To solve this sample constrained problem start a new MS VC project and include the three files `solvopt.c`, `apprgrdn.c` and `nlpsmple.c` to the project.

Build the project with the name **`sample`**.

Case 1.

```
> sample           equivalent to                > sample 1 1
SolvOpt warning:
Re-setting due to the use of a new penalty coefficient.
SolvOpt: Normal termination.

  Function Value =====  Evaluations  === Iterations
      32.3486813         1076+  308         306
                         1076+  177

 Optimum Point X:
      0.300324855
      0.333292914
      0.400173684
      0.428061656
       0.22410683
 -2.90494799e-009
  2.84518649e-008
       5.17038258
 -8.07298191e-009
       3.06188089
       11.8364617
 -8.01840543e-009
  2.92774205e-010
      0.103865592
  1.43687428e-007
```

Case 2.

```
> sample 1 0
SolvOpt: Normal termination.

  Function Value =====  Evaluations  === Iterations
      32.3486978         1106+  318         317
                            0+    0

 Optimum Point X:
      0.299426825
      0.333249409
      0.400064008
      0.428736535
      0.224313069
  2.23384349e-008
  4.04748235e-007
       5.17763934
  1.84117123e-007
       3.06147698
```

```
      11.8402601
  2.43819999e-008
  7.70359665e-009
     0.104299708
  2.23357491e-006
```

Case 3.

```
> sample 0          equivalent to              > sample 0 1
SolvOpt warning:
Re-setting due to the use of a new penalty coefficie
SolvOpt: Normal termination.

  Function Value =====  Evaluations  === Iterations
      32.3487289         6821+    0         346
                         4811+    0

 Optimum Point X:
      0.300166339
       0.33315548
      0.399819535
      0.427900289
      0.223878321
  1.39143231e-007
  1.27240731e-006
       5.17158045
  6.97461307e-007
       3.06243783
       11.8371186
  2.85325022e-008
  6.60984423e-008
     0.102887626
  2.38186366e-005
```

Case 4.

```
> sample 0 0
olvOpt: Normal termination.

 Function Value =====  Evaluations  === Iterations
     32.3486907         7539+    0         382
                           0+    0

Optimum Point X:
     0.300043375
     0.333455818
     0.400153582
     0.428296402
     0.224002824
```

```
 4.53772266e-008
 2.09260635e-007
      5.17314421
 1.32575802e-006
      3.06119755
      11.8374662
 5.48131163e-010
 7.01715815e-009
       0.1035913
 7.01504614e-007
```

Remark. Under the title "Evaluations", the values of the four returned counters are printed. In the first row, the numbers of objective (or penalty) function and gradient evaluations are given. In the second row, the numbers of constraint functions and gradient evaluations are given. The maximal residual for a set of constraints is calculated at every point in any case, however, the gradients for the constraints are calculated only at infeasible points (see cases 1 and 3). Since that, it is almost ever more efficient to let the solver adjust the penalty coefficient by itself.

Parameters as arguments

To minimize a function one must pass to `solvopt` at least two parameters:

i) the starting point `x` and

ii) the name of the M-file (FORTRAN subroutine, C function) that returns the objective function value at a given point.

The user may also pass to the solver the vector of optional parameters and the name of the M-file (FORTRAN subroutine, C function) that returns the gradient of an objective function at a given point.

To solve a constrained optimization problem, in addition to i) and ii), one must pass to `solvopt` at least

iii) the name of the M-file (FORTRAN subroutine, C function) that returns the maximal residual for a set of constraints at a given point

and optionally, the name of the M-file (FORTRAN subroutine, C function) that returns the gradient of a constraint function with the maximal residual at a given point.

The use of analytically calculated gradients of objective and constraint functions has been described by the examples presented in the previous section. The use of the optional parameters is illustrated below for the **Matlab** version only. For more

information see the language specific descriptions of the solver in the *Reference* part. The optional parameters **`options`** to the solver have the same meaning for every programming language used. The exception is the numbering of elements in the array `options` which is different in the C code. Namely, the indices start at 0 in C in contrary to Matlab and FORTRAN, where they start at 1.

As the starting point passed to **`solvopt.m`** is a column (row), the optimum point that the routine returns is also a column (row).

When the function is called as follows:

```
x=solvopt(x,'fun',...),
```

it returns the optimizer only. The solver returns in addition the optimum function value and the values of the optional counters in the row vector `options`, when it is called as follows:

```
[x,f,options]=solvopt(x,'fun',...).
```

We shall consider the use of the optional tuning parameters by applying the solver to Problem 1 (Shor's piece-wise quadratic function) and Problem 3 (Shell Dual Problem).

Required accuracy of the solution

The two parameters, **`options(2)`** and **`options(3)`**, set the required relative errors for the argument and the function respectively. The default values are: `options(2)=1.e-4` and `options(3)=1.e-6`.

Let us solve Problem 1 at `options(2)=1.e-6` and `options(3)=1.e-8`:

```
» options(2)=1.e-6; options(3)=1.e-8;
» x=[-1;1;-1;1;-1]; % the starting point
» [x,f,options]=solvopt(x,'shorf','shorg',options);
SolvOpt: Normal termination.

» format long
» x
x =
   1.12435189584326
   0.97946111531317
   1.47770710321265
   0.92023517830940
   1.12429094264600
```

```
» f
f =
  22.60016209626562
» options(9) % The number of iterations
ans =
    84
» options(10) % The number of function evaluations
ans =
   247
» options(11) % The number of gradient evaluations
ans =
    85
```

The minimum value 22.60016209626562 has been obtained after 247 function and 85 gradient evaluations (compare to the function value 22.6001630849795 obtained after 176 function and 59 gradient evaluations at the default values of the optional parameters).

Upper bound error for the constraints

The upper bound error for the constraints (the admissible maximal residual) is set by **`options(6)`**. By default, `options(6)=1.e-8`. One may require more or less accurate solution by changing this value. Let us illustrate this with solution of Problem 3.

```
» options=soptions; options(6)=1.e-12; x=initdual;
» [x,f,options]=solvopt(x,'dsobjf','dsobjg',options,
'dscntf','dscntg')

SolvOpt warning:
Re-setting due to the use of a new penalty coefficient.
SolvOpt: Normal termination.

» x
x =
   0.30007044533967
   0.33343008706542
   0.40014103298840
   0.42827843110839
   0.22403041097278
   0.00000000103196
```

```
   0.00000002538532
   5.17292380053550
   0.00000000801245
   3.06128343859342
  11.83770574045488
   0.00000000089718
   0.00000000032974
   0.10374292520152
   0.00000001418269
» f
f =
  32.34867973890853
» options(10:13)
ans =
        1165         336        1165         194
```

After computing 1165 objective function values, 336 gradients of the objective function, 1165 maximal residuals of the constraints and 194 gradients of a constraint function, we have obtained a very precise solution (compare to the above one obtained at the default value of `options(6)`).

Displaying intermediate results, warning and error messages

This is controlled by the optional parameter **`options(5)`**. The default value for this parameter is `0`.

One can suppress displaying the warning and error messages by setting `options(5)` to a negative value.

One can view the intermediate results at every first, second, ..., *k*-th iteration by setting `options(5)` respectively to 1,2,...,*k*. The display looks like this:

```
Iter.# ..... Function ... Step Value ... Gradient Norm
   20    3.90622e+001    2.86653e-002      9.10076e+001
```

Return code and counters

The function `solvopt` returns the five parameters, **`options(9:13)`**, in the case of constrained minimization, and the three ones, **`options(9:11)`**, otherwise. The counters are:

`options(10)`, for the number of objective function values,

`options(11)`, for the number of gradients of an objective function,

`options(13)`, for the number of gradients of a constraint with the maximal residual at a point,

`options(12)`, for the number of maximal residuals of constraints, and

computed.

If the finite difference approximation is used for the gradients of an objective function or(and) constraints, the respective counter(s) for the number of gradients evaluations returns zero.

The solver returns the termination code in **`options(9)`**. If it is positive, it provides the number of iterations made. If negative, the termination was abnormal. The negative returning value of `options(9)` points to a (possible) reason of the abnormal termination. The abnormal termination codes are discussed in the language specific descriptions of the solver in the *Reference* part.

Default parameter settings

The `options` vector contains parameters used in the optimization routine. If, at the call to the routine, the `options` vector is empty, all the parameters are set to the default values. If `options` is passed and has fewer than 8 elements, the remaining elements assume their default values. If the value specified for a particular parameter is out of the range allowed, the parameter takes its default value or the nearest bound value. The user may choose a value for any of the first 8 elements, however, one has to know exactly how the algorithm works if one modifies the values for `options(7)` and `options(8)`.

***Note*: In the C version, the index starts at 0.**

No	Functionality	Default	Description
1	Optimization mode	`-1`	Controls the specific optimization mode and provides a factor for the first trial step size (minimization if `sign(options(1))=-1`, maximization if `sign(options(1))=1`).

			For a constrained problem, maximization is not allowed.
2	Required accuracy for `x`	`1.e-4`	This parameter is used as the bound δ_x in the termination criterion which uses the arguments. This criterion is satisfied if the relative error in the coordinates of the points obtained in two successive iterations is less than δ_x, i.e., $\left\|(x_k)_i - (x_{k+1})_i\right\| \le \delta_x \left\|(x_{k+1})_i\right\|$, $i = 1,\ldots,n$. The range of admissible values for this parameter is [1.e-12,1].
3	Required accuracy for `f`	`1.e-6`	This parameter is used as the bound δ_f in the termination criterion which uses the values of the objective function. This criterion is satisfied if the relative error in the function values obtained in two successive iterations is less than δ_f, i.e., $\left\|f(x_k) - f(x_{k+1})\right\| \le \delta_f \left\|f(x_{k+1})\right\|$. The range of admissible values for this parameter is [1.e-12,1].
4	Maximum number of iterations	`15000`	The algorithm stops if the number of iterations exceeds the value of this parameter. A corresponding message is displayed.
5	Display	`0`	Controls the output during the optimization process: `options(5)=0` means that no intermediate results are displayed, `options(5)=N` means that the results for each N-th iteration are displayed, `options(5)=-1` suppresses error and warning messages.
6	Required accuracy for constraints	`1.e-8`	This is the upper bound admissible value for the maximal residual of a set of constraints at a point. The program can stop normally, if a current point is feasible to the extend of this value. The lower bound for this parameter is 1.e-12.
7	Coefficient for space dilation	`2.5`	Controls the space dilation. The default value for the space dilation coefficient is `2.5`. The lower bound for

			this parameter is 1.5.
8	Difference approximation of gradients	`1.e-11`	Lower bound for the stepsize used in the difference approximation of gradients. The range of admissible values for this parameter is [1.e-12,1].

Recommendations

Note: Everywhere an element of the array `options` is mentioned, the C users must take into account that the index starts at 0 in C arrays, therefore, subtract a unit from the index value.

When one uses the SolvOpt in order to minimize (maximize) a function with more than one local minimum (maximum), one has to take into account the fact that SolvOpt chooses the step size automatically by an adaptive procedure. The first trial step size depends on the Euclidean norm of the gradient calculated at the starting point. Because of this, it may happen that the optimization routine finds a local optimum that is not the nearest to the starting point. If it is necessary to find the nearest local optimum, the user has to adjust the first step size by setting `options(1)` to a real number with absolute value less than `1`.

With a starting point close to the optimizer, one may enlarge the space dilation coefficient (`options(7)`) to achieve more rapid convergence. There is no limit on this coefficient. However, the efficient range for most cases is between 2 and 4. However, we strongly recommend using the default values for `options(7)`.

If the objective function cannot be calculated exactly, in particular if the values of the objective function are perturbed by bounded noise, the user must set `options(8)` to the appropriate value to reduce the influence of noise on the gradient calculated by the difference approximation.

Solving a constrained problem one may require a highly accurate result in terms of the fulfillment of constraints. However, this may cause a very long run, especially in the case of equality constraints. We suggest to use the default value for the upper bound on the maximal admissible residual for the constraints and rerun the program at a smaller value of this parameter if desperately needed and after obtaining a solution at the default value of `options(6)`.

SolvOpt is not designed to minimize (maximize) functions of scalars. The lowest possible dimension for the argument is 2. If the starting point is a scalar, an error message is displayed and the return code, `options(9)`, is set to `-1`.

If the return code is negative, meaning the abnormal termination, the obtained solution possibly does not provide the optimum. The displayed message points to the reason of the abnormal termination. If it contains the suggestion on re-running the solver from the obtained point, it is ever worth to do. Often, a termination warning message comes because of detecting a "bad" area, which has been passed by the solver successfully. Nevertheless, you must not fully trust the solution, if a termination warning comes. The termination code and warning messages are discussed in the *Reference* part.

It is almost ever more efficient to provide analytically calculated gradients for an objective function and constraints. The user must take care of possible errors in the user-supplied routines, especially the gradient calculating functions. We do not provide a check utility for the gradients calculated analytically, however, there are many such utilities available freely at public domain sites.

2 Description of the algorithm

This chapter provides a detailed description for a version of Shor's minimization method with space dilation [Sh85].

Shor's *r*-algorithm seems to be one of the most efficient methods for the minimization of non-smooth (i.e., almost-differentiable) functions. However, a serious problem for Shor's algorithm is the design of an efficient stopping criterion, a difficulty common to all algorithms for optimization of non-smooth functions. A further difficulty is the choice of initial step size. Below we describe in detail the modifications and the additions which resulted in a robust and efficient algorithm.

The introduction to the algorithm starts with a description of Shor's original method with space dilation along the difference of two successive subgradients.

Shor's *r*-algorithm

The main idea of the algorithm is to make steps in the direction opposite to a subgradient at the current point. However, the steps are to be made in the transformed space. The way to perform this transformation is quite simple. At each iteration one calculates the difference between a subgradient at the current point and that calculated at the previous step. The direction obtained is used to perform dilation of the space with a given (inour case) a priori as coefficient $\alpha > 1$ (`options(7)`).

Let $f(\cdot)$ be an almost-differentiable convex function defined on $\mathbb{R}^n$ which is differentiable on its domain except on a set of measure zero. Let us denote an almost-gradient of $f(\cdot)$ at the point x by $g_f(x)$. Consider the following iterative algorithm for minimization of the function $f(\cdot)$.

Assume that after k iterations one has obtained the point x_k, the space transformation matrix B_k and the subgradient $\tilde{g}_k$ of the function $\varphi_k(y) = f(B_k y)$ at the point $\tilde{y}_k = B_k^{-1} x_{k-1}$.

At the $(k+1)$-th iteration the following calculations have to be performed:

(1) Calculate $g_f(x_k)$, a subgradient f at x_k.

(2) Calculate $g_k^* = B_k^T g_f(x_k)$, a subgradient of φ_k at $y_k = B_k^{-1} x_k$.

(3) Calculate $r_k = g_k^* - \tilde{g}_k$, the difference of the two subgradients of φ_k at y_k and $\tilde{y}_k$.

(4) Set $\xi_{k+1} = r_k / \|r_k\|$. The normalized vector ξ_{k+1} is the direction of the next space dilation to be performed.

(5) Calculate $B_{k+1} = B_k R_\beta(\xi_{k+1})$, where $\beta = 1/\alpha$. The matrix $R_\beta(\xi_{k+1})$ is the inverse of $R_\alpha(\xi_{k+1})$, the matrix of the space dilation in the direction ξ_{k+1} with coefficient α given by

$$R_\alpha(\xi_{k+1})x = x + (\alpha - 1)(x^T \xi_{k+1})\xi_{k+1}, \qquad x \in \mathbb{R}^n.$$

(6) Calculate $\tilde{g}_{k+1} = B_{k+1}^T g_f(x_k)$, a subgradient of the function $\varphi_{k+1}(y) = f(B_{k+1} y)$ at the point $\tilde{y}_{k+1} = B_{k+1}^{-1} x_k$.

(7) Choose a step size h_{k+1}.

(8) Set $x_{k+1} = x_k - h_{k+1} B_{k+1} \tilde{g}_{k+1}$.

(9) Check the stopping criterion and stop if it is satisfied. Otherwise proceed to the next iteration.

To turn the *r*-algorithm as presented above into an efficient and robust optimization routine, one has to find solutions to the following problems:

- Initialization and re-initialization of the space transform matrix B_k and initialization of the step size h_k.
- Choice of the step size h_{k+1} to optimize the efficiency of space dilation in the direction of two consecutive subgradients (of the transformed function φ_k).
- Construction of a stopping criterion which does not need information on gradients.

In the following subsections we describe the solutions to these problems implemented in SolvOpt. The solutions are heuristic.

Initial trial step size

The initial value $\hat{h}_1$ for trial step size has to meet the following obvious requirements: (i) it must be large enough to avoid premature termination of the process in the case where the starting point is far from the optimum, and (ii) it must be small enough in the opposite case where the starting point and the optimum are close to each other. How can we satisfy these requirements, having no other information on the objective function but the starting point and the gradient calculated at this point? In view of these conditions and our limited knowledge the following formula is suggested for the first trial step along the gradient calculated at the starting point:

$$\hat{h}_1 = \max\left(\frac{c}{\log_2\left(\|g_0\|_2 + 1\right)}, \sqrt{\delta_x}\|x_0\|_\infty\right),$$

where $c > 0$ is the factor set by `options(1)` and δ_x is the required relative error for the optimizer set by `options(2)` (see *Tutorial*).

Re-Initialization

The space transformation matrix holds the history of the optimization process. It should cause the algorithm to make steps mostly towards the optimum instead of along the current gradient, which may be almost orthogonal to the desired direction. A similar strategy is pursued in quasi-Newton methods. However, occasionally one has to forget the history of the optimization process. In particular, this might be necessary when the objective function has the shape of a ravine. There is no particular technique to recognize this. Nevertheless, any method that holds the process history has to have such a "switch" to turn back to the initial values and to start optimization from the current point in the same way as from the starting point.

Since the space dilation coefficient is larger than 1, the norm of the gradients in the transformed space decreases from iteration to iteration. If the dilations occur more or less in the same direction, which is the most preferable situation, then the norm of the transformed gradients possibly becomes close to zero only near the optimum. However, if space dilations are repeatedly made in directions which almost give an orthogonal basis of the space, then the norm of the transformed gradients may become very small also far away from the optimum. Such a situation can occur for ravine shaped functions (e.g. the Shell Dual Problem [LM78] and Shor's problem [Sh85]) and for functions which are badly scaled (as for instance Meyer's function).

The problem is to decide when one should reset the algorithm, so that it starts anew at the present point and the past history of the process does not interfere in choosing the next direction to go. The following simple criterion is based on heuristics and proved to be efficient.

Resetting of the space dilation matrix:

If at the k -th iteration the inequality

$$\|\tilde{g}_k\|_2 \le 10^{-15l^2}\|g_k\|_2$$

is satisfied, where g_k, $\tilde{g}_k$ are the gradients of the functions $f(x)$ and $\varphi(y)$, l is the number of nonzero coefficients of g_k, then assume

$B_k = I$, i.e., set the space transformation matrix to identity.

The step size strategy

Any optimization algorithm which uses the gradient of the objective function invokes at each iteration a line search procedure to determine the step size. In algorithms for smooth problems, the line search procedure usually tries to find or to approximate the optimizer of the objective function f in the chosen direction. However, in Shor's *r*-algorithm it seems to be necessary to find a step size such that the optimizer for f in the chosen direction is a point *between* the current point and the next point of the solution path. This is the most efficient way to guarantee that the direction for space dilation at the next step (the difference of the gradients of f at x_k and x_{k+1}) points in the direction where the valley of the objective function which just has been crossed descends to a local minimum or ascends to a local maximum. The step size strategy implemented in SolvOpt also avoids increasing the objective function too much in one step. However, the objective function might increase occasionally, so the algorithm is not a true descent algorithm. This fact, together with the difficulty of keeping track of the successive space transformations, is the main reason that at present there does not exist a sufficiently general proof of convergence for the *r*-algorithm.

In the description below it is understood that the task is to find a local minimum of the objective function. We assume that steps (1)-(6) of $(k+1)$-th iteration of the *r*-algorithm have been completed, so

$$x_k, \quad g_k, \quad \tilde{g}_{k+1} \quad \text{and} \quad B_{k+1}$$

are already computed. Moreover, the trial step size $\hat{h}_{k+1}$ for the line search in the $(k+1)$-th iteration has also been determined. Then we have to perform the following calculations:

(a) Set $j_{k+1}=0$, $h_{k+1}=\hat{h}_{k+1}$, $x_{k+1}^{(0)}=x_k$ and $f^{(0)}=f(x_{k+1}^{(0)})$.

(b) Calculate $\gamma=\min\left(b_1^{\max(\log_{10}(\|g_k\|+1),1)},\ 1+b_2^{p-q}\right)$, where b_1=1.15, $b_2=1+0.1/n^2$, p=20 is the starting value and q is the iteration counter reset at every re-initialization of the transformation matrix.

(c) Calculate

$$x_{k+1}^{(j_{k+1}+1)}=x_{k+1}^{(j_{k+1})}-h_{k+1}B_{k+1}\tilde{g}_{k+1},\qquad f^{(j_{k+1}+1)}=f(x_{k+1}^{(j_{k+1}+1)}).$$

(d) Set $j_{k+1}=j_{k+1}+1$.

(e) If $f^{(j_{k+1})}\geq\gamma f^{(j_{k+1}-1)}$, then set $h_{k+1}=h_{k+1}/5.1$, $j_{k+1}=0$ and continue with (c). Otherwise go to (f).

(f) If $f^{(j_{k+1})}<f^{(j_{k+1}-1)}$, check the step counter, set

$$h_{k+1}=\begin{cases}2h_{k+1}, & \text{if } j_{k+1}>20,\\ 1.5h_{k+1}, & \text{if } 20\geq j_{k+1}>10,\\ 1.05h_{k+1}, & \text{if } 10\geq j_{k+1}>2,\\ h_{k+1}, & \text{if } j_{k+1}\leq 2\end{cases}$$

and continue with (c). Otherwise go to (g).

(g) Set $x_{k+1}=x_{k+1}^{(j_{k+1}+1)}$ and

$$\tilde{j}=\left(\frac{\Delta x_{k-1}}{\hat{h}_{k-1}}+\frac{2\Delta x_k}{\hat{h}_k}+\frac{3\Delta x_{k+1}}{\hat{h}_{k+1}}\right)\Bigg/\sum_{i=1}^{\min(k,3)}i,$$

where $\Delta x_k=\|x_k-x_{k-1}\|_2$.

Check the value $\tilde{j}$ and set

$$\hat{h}_{k+2}=\begin{cases}\sqrt{\tilde{j}-j_0+1}\,\hat{h}_{k+1}, & \text{if } \tilde{j}>j_0,\\ \hat{h}_{k+1}, & \text{if } \tilde{j}=j_0,\\ \sqrt{\tilde{j}/j_0}\,\hat{h}_{k+1}, & \text{if } \tilde{j}<j_0.\end{cases}$$

where j_0 =3.3, if the analytically calculated gradients are used, and j_0 =6.3, otherwise.

Note that item (g) is reached only if $f(x_{k+1}^{(j_{k+1})}) \geq f(x_{k+1}^{(j_{k+1}-1)})$.

Termination

The stopping criterion used in SolvOpt (besides the condition that the maximum number of iterations given by `options(4)` is not exceeded) can only be seen in connection with the step size strategy implemented in SolvOpt. The efficiency of the stopping criterion can be explained by the fact that, in general, the minimum (maximum) of the objective function on the line joining x_k and x_{k+1} is attained at a point between x_k and x_{k+1} . This prevents the step size from becoming too small if x_k is not close enough to the local minimizer (maximizer). In our test the stopping criterion always guaranteed the required accuracy by choosing the parameters `options(2)` and `options(3)` (i.e., δ_x and δ_f) appropriately.

The criterion:

If both the set of inequalities

$$\left|x_{k+1}^i - x_k^i\right| \leq \delta_x \left|x_{k+1}^i\right|, \quad i = 1,\ldots,n, \qquad \text{(TERMX)}$$

and the inequality

$$\left|f(x_{k+1}) - f(x_k)\right| \leq \delta_f \left|f(x_{k+1})\right| \qquad \text{(TERMF)}$$

are fulfilled, terminate the algorithm.

Here δ_x and δ_f are the required relative errors for the argument and function value at the solution, respectively. These values are given by `options(2)` and `options(3)` (see *Default parameter settings* in *Tutorial*).

Additionally, the algorithm stops,

1) if the condition $f^{(j_{k+1})} < \gamma f^{(j_{k+1}-1)}$ is not fulfilled at the smallest possible stepsize (this may happen particularly in the case, when gradients are approximated by the finite differences) or
2) if the condition (TERMX) is fulfilled and

$$\left|f(x_{k+1})\right| \le \delta_f^2 \quad \text{and} \quad \left|f(x_{k+1}) - f(x_k)\right| \le \delta_f$$

(this avoids iddling at very small absolute function values).

Other heuristic procedures implemented

If the regular stopping criteria (TERMX) and (TERMF) are not fulfilled, the algorithm does not stop at a point, where the gradient is (approximately) zero in the case of unconstrained optimization, unless it fails to find a point with a nonzero gradient. This avoids a premature stop at a point, where the function is flat, but takes no optimum.

If the condition (TERMF) is fulfilled alone and one or more elements of the gradient calculated at a point are zero, the insensitive (with zero partial derivatives at this point) variables are shifted manually in order to find a point with a smaller function value. This is also made with the aim to avoid a premature stop at a point, where the function is flat, but takes no optimum.

3 Reference

This chapter contains descriptions of the **`solvopt`** function/subroutine, auxiliary routines and sample problem files. The functions/subroutines are listed within the groups by their purpose, that is exactly in the order in which they are listed in the tables below. Information is also available through the online help facility for Matlab functions and readme files specific for a programming language. The following tables detail the routines and demonstration programs available:

Distributed source codes

Nonlinear optimization routine	
`solvopt`	The solver for local nonlinear optimization problems (Matlab, FORTRAN,C)

Utilities and auxiliary routines	
`apprgrdn`	Finite difference approximation of the gradient (Matlab, FORTRAN, C)
`soptions`	Parameter settings (Matlab, FORTRAN)

Include and header files	
`messages.inc`	Error and warning strings (FORTRAN specific)
`soptions.h`	Parameter settings (C specific)

Demonstrations and examples	
***Matlab specific*:**	
`initdual`	Constants for the Shell Dual Problem
`dualshf`	Value of the penalty function of the Shell Dual Problem
`dualshg`	Gradient of the penalty function of the Shell Dual

	Problem
dsobjf	Value of the objective function of the Shell Dual Problem
dsobjg	Gradient of the objective function of the Shell Dual Problem
dscntf	Maximal residual for a set of constraints (Shell Dual Problem)
dscntg	Gradient of the constraint function with the maximal residual at a point (Shell Dual Problem)
initill	Constants for the Ill-conditioned LP problem
illclinf	Value of the penalty function of the Ill-conditioned LP problem
illcling	Gradient of the penalty function of the Ill-conditioned LP problem
initmaxq	Constants for Lemarechal's MaxQuad function
maxquadf	Value of Lemarechal's MaxQuad function
maxquadg	Gradient of Lemarechal's MaxQuad function
shorf	Value of Shor's function
shorg	Gradient of Shor's function
solvdemo	Tutorial walk-through
testf	Interface function to Moré set of tests, returns the objective function value
testg	Interface function to Moré set of tests, returns the gradient
unctest	Minimization of supplied test functions
UNCPROBS	The directory that contains the M-codes for commonly used differentiable test functions by Moré *et al.*
	FORTRAN specific:
dualshl0 **dualshl1** **dualshl2** **dualcnst** **dualcnt2**	Sample problem files for solution of the Shell Dual Problem

shor	Sample problem file for minimization of Shor's piece-wise quadratic function
	C specific:
nlpsmple	Sample problem file for solution of the Shell Dual Problem
shor	Sample problem file for minimization of Shor's piece-wise quadratic function

The demonstration and sample files are not documented in this manual. They are partly described in the Tutorial. One can also find detailed comments in the sources files.

solvopt

Purpose

Find a local optimum for a nonlinear (non-smooth) continuous function or solve a nonlinear optimization problem in the standard form (NLP)

Usage

Matlab:

```
[x,f,options] = solvopt(x,fun,grad,options,func,gradc)
```

FORTRAN:

```
call
solvopt(n,x,f,fun,flg,grad,options,flfc,func,flgc,gradc)
```

C:

```
f = solvopt ( n,x,fun,grad,options,func,gradc );
```

Language specific descriptions

The algorithm is described in *Description of the algorithm.*

Matlab

The necessary parameters to the function are boldface, the optional ones are normalface.

x	is a vector (row or column) of the coordinates of the starting point.
fun	provides the name (valid character string) of the M-file (M-function) that returns the objective function value `f` at a point `x` (`f` may also take the values `Inf`, `-Inf`, `NaN`). Synopsis: **`f=fun(x)`**
`grad`	provides the name (valid character string) of the M-file (M-function) that returns the gradient vector `g` (row or column)

of an objective function at a point `x` (`g` may also take the values `Inf`, `-Inf`, `NaN`).

Synopsis: **`g=grad(x)`**

`func` provides the name (valid character string) of the M-file (M-function) that returns the maximal residual `fc` for a set of constraints at a point `x` (`fc` may **not** take the values `Inf`, `-Inf`, `NaN`). By passing this parameter to the solver, the user specifies the constrained optimization mode.

Synopsis: **`fc=func(x)`**

`gradc` provides the name (valid character string) of the M-file (M-function) that returns the gradient vector `gc` (row or column) of a constraint function with the maximal residual at a point `x` (`gc` may **not** take the values `Inf`, `-Inf`, `NaN`, it also may **not** be zero at an infeasible point).

Synopsis: **`gc=gradc(x)`**

`options` is a vector of the optional parameters:

`options(1)` controls the specific optimization mode and provides a factor for the first trial step size (minimization if `sign(options(1))=-1`, maximization if `sign(options(1))=1`). The default value is `-1`. *Exception*: Maximization is not allowed for a constrained problem.

`options(2)` is the relative error for the argument at the solution point in terms of ℓ_∞-norm (`=1.e-4` by default).

`options(3)` is the required relative error for the objective function value at the solution (`=1.e-6` by default).

`options(4)` is a limit for the number of iterations (`=15000` by default).

`options(5)` controls the output during the optimization process: `0` means that no intermediate results are displayed, `-1` means no output (suppresses error and warning messages), `N` means that the

results for each N-th iteration are displayed. The default value is 0.

`options(6)` is the upper bound admissible value for the maximal residual of a set of constraints. This means that if the maximal residual at a point is less then `options(6)`, the point is assumed to be feasible. The default is `1.e-8`.

Changing the following is somewhat delicate:

`options(7)` is the coefficient of space dilation (`=2.5` by default)

`options(8)` is the relative lower bound for the step size used in the difference approximation of gradients. (`=1.e-11` by default).

Values to be returned:

`x` is the solution point (row or column, depending on how the routine is called).

`f` is the value of the objective function at the solution point.

`options` returns the values of the following counters:

`options(9)`, the number of iterations made, `options(9)<0` means that the termination was abnormal and provides a specific termination code (these codes are listed below),

`options(10)`, the number of objective function evaluations,

`options(11)`, the number of gradient evaluations for the objective function,

`options(12)`, the number of constraints evaluations (equals to `options(10))`,

`options(13)`, the number of gradient evaluations for the constraints.

More information on the use of the optional parameters may be found in the section *Parameters as arguments* (see also *Default parameter settings*) of the *Tutorial*.

As the routine **solvopt** passes to both the <fun> and the <grad> M-functions only the current point x, one has to use Matlab global declaration to pass the needed parameters to these M-functions. One can find an example of the use of global constants in the section *Examples* of the *Tutorial*.

One cannot pass to **solvopt** any other string, except the names of M-functions, by arguments fun or grad. The routine **solvopt** cannot be used for optimization of expressions if they are passed to the routine as arguments.

FORTRAN

n — is the number of variables.

Type: **integer**

x — is an **n**-dimensional vector of the coordinates of the starting point at a call to the subroutine and the optimizer at a regular return.

Type: **double precision**

f — is the objective function value at the solution point at a regular return.

Type: **double precision**

fun — provides the entry name to the subroutine that calculates the objective function value f at a point x. If the absolute value of f is greater then 1.d100, it is assumed (+|-) infinity.

The actual entry name, which is passed as a parameter to the solver, must be declared as **external** in the calling routine.

Synopsis:
```
subroutine fun(x,f)
      double precision f
```

flg — is a flag pointing to the way the gradient of the objective function is calculated: .true. means that the gradients are calculated analytically by use of subroutine <**grad**>, .false. means that the gradients are approximated by the finite differences.

Type: **logical**

grad — provides the entry name to the subroutine that calculates the gradient g of the objective function value at a point x.

The actual entry name, which is passed as a parameter to the solver, must be declared as **external** in the calling routine.

If analytically calculated gradients are not supplied, the entry name **null** is used at a call to the solver.

Synopsis: **subroutine *grad*(x,g)**
```
double precision g(*)
```

flfc is a flag used to indicate the constrained optimization mode: `.true.` means that the problem is constrained and the maximal residual of a set of constraints is calculated in the subroutine <**func**>, `.false.` means that the problem is unconstrained or the user-defined penalty function is minimized.

Type: **logical**

func provides the entry name to the subroutine that calculates the maximal residual `fc` for a set of constraints at a point `x`.

The actual entry name, which is passed as a parameter to the solver, must be declared as **external** in the calling routine.

If the problem is unconstrained, the entry name **null** is used at a call to the solver.

Synopsis: **subroutine *func*(x,fc)**
```
double precision fc
```

flgc is a flag pointing to the way the gradients of constraints are calculated: `.true.` means that the gradients are calculated analytically by use of subroutine <**gradc**>, `.false.` means that the gradients are approximated by the finite differences.

`flgc` may not take the value `.true.`, if `flfc=.false.`.

Type: **logical**

gradc provides the entry name to the subroutine that calculates the gradient `gc` of the constraint function with the maximal residual at a point `x.`

The actual entry name, which is passed as a parameter to the solver, must be declared as **external** in the calling routine.

If analytically calculated gradients are not supplied, the entry

name **null** is used at a call to the solver.

Synopsis: **subroutine *gradc*(x,gc)**
`double precision gc(*)`

options is a vector of the optional parameters. See the description of the elements in the *Matlab* specific section.

Type: **double precision**

More information on the use of the optional parameters may be found in the section *Parameters as arguments* (see also *Default parameter settings*) of the *Tutorial*.

Note: The user must take care about possible domain errors in the FORTRAN intrinsic math-library functions `log`, `log10`, `sqrt`, etc., if they are used in the user's code, to prevent an abnormal stop.

C

Synopsis:

```
double solvopt( unsigned short n,
                double x[],
                double far fun(),
                void far grad(),
                double options[],
                double far func(),
                void far gradc() )
```

The function **solvopt** returns the objective function value at the solution.

n is the number of variables.

x is an **n**-dimensional vector of the coordinates of the starting point at a call to the subroutine and the optimizer at a regular return.

fun provides the entry to the function that calculates the objective function value `f` at a point `x`. If the absolute value of `f` is greater then `1.e100`, it is assumed (+|-) infinity.

The actual function name, which is passed as a parameter to the solver, must be declared as **far** in the calling routine.
Synopsis: **double *fun*(double x[])**

grad provides the entry to the function that calculates the gradient g of the objective function value at a point x.

The actual function name, which is passed as a parameter to the solver, must be declared as **far** in the calling routine.

If analytically calculated gradients are not supplied, the function name **null_entry** is used at a call to the solver.
Synopsis:
void *grad*(double x[], double g[])

func provides the entry to the function that returns the maximal residual fc for a set of constraints at a point x.

The actual function name, which is passed as a parameter to the solver, must be declared as **far** in the calling routine.

If the problem is unconstrained, the function name **null_entry** is used at a call to the solver.
Synopsis: **double *func*(double x[])**

gradc provides the entry to the function that calculates the gradient gc of the constraint function with the maximal residual at a point x.

The actual function name, which is passed as a parameter to the solver, must be declared as **far** in the calling routine.

If analytically calculated gradients are not supplied, the entry name **null_entry** is used at a call to the solver.
Synopsis:
void *gradc*(double x[], double gc[])

options is a vector of the optional parameters. See the description of the elements in the *Matlab* specific section (take into account that the index starts at 0 in C, therefore, every subtract a unit from the index).

More information on the use of the optional parameters may be found in the section *Parameters as arguments* (see also *Default parameter settings*) of the *Tutorial*.

Notes:

1. The user must include the directive

   ```
   #include <solvopt.h>
   ```

   to the calling C-function.

2. The user must take care about possible domain errors in the C intrinsic math-library functions `log`, `log10`, `sqrt`, etc., if they are used in the user's code, by providing a function that process these errors and prevents an abnormal stop.

Return codes, error and warning messages

The solver prints an error message at a premature stop, unless printing messages is suppressed by setting `options(5)` (`options(4)` in C) to a negative value. It starts with the line

`SolvOpt error:`

A termination warning message is printed at an abnormal stop (when regular stopping criteria are not fulfilled or the solver detected an irregular case) and starts with the line

`SolvOpt: Termination warning:`

Warning messages are printed also during the optimization process, when it is worth to warn the user about the case. These messages start with the line

`SolvOpt warning:`

At a regular return from the solver, the program prints the message

`SolvOpt: Normal termination`

and returns the number of iteration made in `options(9)` (`options(8)` in C).

The return codes are listed below together with the corresponding error and warning messages.

-1:

Allocation Error = *number* (*FORTRAN and C only*)
Check the memory available and the parameter **n** to the solver.

No function name and/or starting point passed to the function. *(Matlab specific)*
Check the call to the function `solvopt.m`.

-2:

Improper space dimension. (*FORTRAN and C only*)
Check the parameter **n** to the solver.

Argument X has to be a row or column vector of dimension > 1. *(Matlab specific)*
Check the parameter `x` to the function `solvopt.m`. The argument must not be a scalar.

-3:

<fun> returns an empty string. (*Matlab specific*)
Check the M-function `<fun>`.

Function value does not exist (NaN is returned). (*Matlab specific*)
Function equals infinity at the point.
Choose another starting point.

-4:

<grad> returns an empty string. (*Matlab specific*)
Check the M-function `<grad>`.

Gradient does not exist (NaN is returned by <grad>). (*Matlab specific*)
Gradient equals infinity at the starting point.
Gradient equals zero at the starting point.
Choose another starting point.

-5:

<func> returns an empty string. (*Matlab specific*)
Check the M-function `<func>`.

<func> returns NaN at the point. (*Matlab specific*)
<func> returns infinite value at the point.
Check the routine calculating the maximal residual for a set of constraints.

-6:

<gradc> returns an improper vector. Check the dimension. (*Matlab specific*)
Check the M-function `<gradc>`.

<gradc> returns NaN at the point. (*Matlab specific*)
<gradc> returns infinite vector at the point.
<gradc> returns zero vector at an infeasible point.
Check the routine calculating gradients of the constraints.

-7:

Function is unbounded.

Usually, this means that the user-defined penalty coefficient is too small and the penalty function is not lower bounded.

-8:

Gradient is zero at the point, but stopping criteria are not fulfilled.

This means that the solver fails to pass through a (possibly) flat area. The returned point may not provide the optimizer.

-9:

Iterations limit exceeded.

This may be caused by setting the optional parameter `options(4)` (`options(3)` in C) to a number smaller then the default value. If the default value is used, this also may be caused by demanding a very small absolute error for the constraints or by errors in the gradient calculating routine(s). Finally, the default number of iterations may not be sufficiently large for a huge-scaled problem.

-11:

Premature stop is possible. Try to re-run the routine from the obtained point.

The solver returns the code -11 whenever detects a "ravine" with steep walls and a flat bottom, meaning the level surfaces are almost parallel and the two successive gradients are opposite to each other. Either the solver has managed to pass through a "nasty" area or it has not, the program returns an abnormal code and prints this message. This is done because of the highest difficulty of the case for the solver.

-12:

Result may not provide the optimum. The function apparently has many extremum points.

The solver returns the code -12, if the stopping criteria are fulfilled at a point, which is not the "best" recorded one and sufficiently far from the recorded point. This also may happen, if the objective function is flat at the optimum, but more likely the function has many locally extreme points.

-13:

Result may be inaccurate in the coordinates. The function is flat at the optimum.

The solver returns the code -13, if the stopping criteria are fulfilled at a point, where one or more partial derivatives are zero.

-14:

Result may be inaccurate in a function value. The function is extremely steep at the optimum

The solver returns the code -14, if the stopping criterion (TERMF) is not fulfilled at a point, but the stepsize is approximately zero. This may be caused by the use of a very large penalty coefficient and by inaccurate gradient approximation as well.

The following warning messages may be printed during a run to inform the user on irregular cases:

Ravine with a flat bottom is detected.

Re-run from recorded point.

The function is flat in certain directions.
Trying to recover by shifting insensitive variables.

Finally, the following warnings provide an additional information on a run, but do not point to anything abnormal.

Normal re-setting of a transformation matrix.

Re-setting due to the use of a new penalty coefficient.

Calls

`apprgrdn,` user-supplied routines <**`fun`**>, <**`func`**>, <**`grad`**> and <**`gradc`**>.

Examples

See *Tutorial*.

apprgrdn

Purpose

Gradient approximation by the finite differences.

Usage

Matlab:

```
g = apprgrdn ( x,f,fun,deltax,obj)
```

FORTRAN:

```
call apprgrdn (n,g,x,f,fun,deltax,obj)
```

C:

```
apprgrdn (n,g,x,f,fun,deltax,obj);
```

Description

The routine performs the finite difference approximation of the gradient `g` at a point `x`. Normally, it calculates the forward differences (the argument `deltax` must provide the signed relative stepsizes at an entry to the routine). If the argument `obj` equals `1` (`.true.`), the routine calculates the central differences for the coordinates of `x` that have small (`<1`) absolute values. It is assumed that the function value `f` is calculated at a point `x` before a call to the routine.

Arguments:

n is the space dimension (*FORTRAN and C specific*)
Type: FORTRAN: `integer`
C: `unsigned short`

g is the approximated gradient vector returned by the routine.
Type: FORTRAN: `double precision`
C: `double`

x is the current point in the `n`-dimensional space.
Type: FORTRAN: `double precision`
C: `double`

f is the function value at a point (must be evaluated before a call to the routine).
Type: FORTRAN: `double precision`
C: `double`

fun provides the entry to the function/subroutine that calculates the objective/constraint function value `f` at a point `x`.

See synopsis in the language specific description section for the routine `solvopt`.

deltax is an `n`-element array of the relative step sizes. The sign of an element points to the direction along the axis for making a step.
Type: FORTRAN: `double precision`
C: `double`

obj is a flag which allows (if `1` or `.true.`) or not (if `0` or `.false.`) calculating of central differences.
Type: FORTRAN: `logical`
C: `unsigned short`

Calls

user-supplied function/subroutine `<fun>` or `<func>` (see description of `solvopt`).

soptions

Purpose

Restore the default settings for the optional parameters.

Usage

Matlab:

```
options = soptions
```

FORTRAN

```
call  soptions ( options )
```

Description

soptions returns the default values for the optional parameters used by SolvOpt. A detailed description of the optional parameters appears in the sections *Parameter as arguments* and *Default parameter settings* of the *Tutorial* part and in the description of the routine **solvopt**.

The **M-function** returns the row
options=[-1,1.e-4,1.e-6,15000,0,1.e-8,2.5,1.e-11].
The **FORTRAN subroutine** returns the double precision 13-element array with the same values for the first 8 elements and zeros for the last 5.

solvopt.h

This header file is used with the **C version** and has the same purpose as FORTRAN subroutine `soptions`. Additionally, the function `null_entry` is declared in this header file.

`solvopt.h` must be included to the C function calling to the `solvopt`:

```
#include <solvopt.h>
```

REFERENCES

[LM78] Lemarechal C. and Mifflin R. (eds.), Nonsmooth Optimization, Pergamon Press, Oxford 1978.

[Ki85] Kiwiel K. C., Methods of Descent for Nondifferentiable Optimization, Lecture Notes in Mathematics, Vol. 1133, Springer-Verlag, Berlin 1985.

[Sh85] Shor N.Z., Minimization Methods for Non-Differentiable Functions, Springer Series in Computational Mathematics, Vol. 3, Springer-Verlag, Berlin 1985.

Acknowledgements

Since the first version of SolvOpt had become available for free downloading from the web, the authors have been receiving many comments mostly from those people doing their research in the area of optimization. The authors appreciated these fruitful and grounded critical remarks on SolvOpt. Here we express our thanks particularly to Prof. Arnold Neumaier (University of Vienna) for his valuable and helpful suggestions.

The authors acknowledge support by FWF (Austria) under grants M00331-Mat (A. K.) and F003 (A. K. and F. K.).