An Implementation of the DIRECT algorithm

Jörg Gablonsky
North Carolina State University
Department of Mathematics
Center for Research in Scientific Computation
Box 8205
Raleigh, NC 27695 - 8205

August 20, 1998

Contents

1 User Guide **3**
1.1 Introduction to DIRECT . . . 3
1.2 What is included in the package . . . 3
1.3 Calling DIRECT (serial Version) . . . 4
1.4 Calling DIRECT (Parallel Version) . . . 6

2 Classical 1-D Lipschitzian Optimization **7**

3 DIRECT Algorithm in 1-D **11**
3.1 Area-Dividing Strategies . . . 11
3.2 Potentially optimal intervals . . . 11

4 The multidimensional DIRECT algorithm **15**
4.1 Dividing in higher dimensions . . . 15
4.2 Potentially optimal hyper-rectangles . . . 17
4.3 Dividing according to the steepest descent direction . . . 18

5 Numerical results **19**
5.1 Test functions . . . 19
5.2 Behavior in an application . . . 23
5.2.1 The valve-train problem . . . 23
5.2.2 Results with different datasets . . . 25
5.2.3 Parallel behavior . . . 25

6 Proofs **27**
6.1 The estimation for the Lipschitz-constant . . . 27

7 Conclusions and future work **28**

1 User Guide

1.1 Introduction to DIRECT

DIRECT (*Div*iding *rec*tangles) is an algorithm for finding the minimum of a Lipschitz continuous function. DIRECT is designed to solve problems subject to box constraints. A exact mathematical definition of the problem is given below after the definition of Lipschitz continuity.

Definition 1.1 *Let $f : R^N \to R$. f is called Lipschitz continuous with Lipschitz-constant γ if and only if*

$$||f(x) - f(x')|| \leq \gamma ||x - x'|| \; \forall x, x'. \tag{1}$$

Problem
Let $f : R^N \to R$ be Lipschitz continuous with constant γ. Solve

$$\min_{x \in \Omega} f(x)$$

where

$$\Omega = \{x \in R^N | l_i \leq (x)_i \leq u_i\}$$

and

$$-\infty < l_i \leq u_i < +\infty.$$

It is important to note that we do not need the function to be continuous and we do not need existence of derivatives.

1.2 What is included in the package

The code can be found at the following WWW-address :

http://www4.ncsu.edu/eos/users/c/ctkelley/www/optimization_codes.html

If you got the file *direct.tar.gz*, uncompress it with *gunzip* and then use *tar* like this :

tar -xf direct.tar

Then you should have a subdirectory called *direct* containing the following files:

DIRect.f	the main routine
DIRserial.f	special routines for serial version of DIRECT
DIRparallel.f	special routines for parallel version of DIRECT
DIRsubrout.f	subroutines used in DIRECT
DIRmainserial.f	sample program for the serial version of DIRECT
DIRmainparallel.f	sample program for the parallel version of DIRECT
DIRtest.f	sample test-function
makefile	makefile to make the serial version of the sample program.
direct.ps	newest version of this document.

1.3 Calling DIRECT (serial Version)

In this section the calling sequence for DIRECT is described and the arguments are explained. This is done the same way as the similar user guide for IFFCO by Paul Gilmore [1]. After this the user supplied subroutines are explained.

The things to do different for the parallel version of DIRECT are explained in the next section.

- **Calling sequence**
  call Direct(fcn,x,n,eps,maxf,maxT,dwrit,fmin,l,u,cheat,kmax,writed,loadH)

- **Arguments**
  The arguments are listed in the order they appear in the calling sequence.

  - **On Entry**

    fcn is the argument containing the name of the user-supplied subroutine that returns values for the function to be minimized. *fcn* must be declared **EXTERNAL** in the calling program.

    n is an integer that represents the dimension of the problem. If $n > 12$ then the parameter *maxor* in the variable list at the beginning of DIRECT in the file DIRect.f must be set to a larger value. *maxor* is a parameter used to dimension the work arrays used in DIRECT .

    eps is a user-supplied double-precision constant used to ensure sufficient decrease in function value when a new potentially optimal interval is chosen. It is normally set to about 1.D-2, although you should try lower values if you are not happy with the results you get.

    maxf is a user-supplied integer and gives the maximum number of function evaluation to be allowed. This is only an approximate upper boundary, because the direct algorithm will finish the division of the actual hyper-rectangle. If it is set to a value higher than 90000, you need to change the parameter *Maxfunc* at the beginning of DIRECT in the file DIRect.f. *Maxfunc* is a parameter used to dimension the work arrays used in DIRECT .

    maxT is a user-supplied integer that gives the maximum number of iterations. When the maximum number of function evaluations is reached earlier, DIRECT stops before finishing all iterations. If the value is set higher then 500, you need to change the parameter *Maxdeep* at the beginning of DIRECT in the file DIRect.f. *maxdeep* is a parameter used to dimension the work arrays used in DIRECT .

    dwrit is a user-supplied integer. *dwrit* specifies the verbosity of the algorithm:

    dwrit = 1 Values of the function are written to the standard output.

    dwrit = 2 Values of the function are written to the standard output and to the file "test.dat".

    u is a user-supplied double-precision vector of length n. u is the vector containing the upper bounds for the n independent variables. The hyper-box defined

by the constraints on the variables is mapped to the unit-cube in DIRECT . DIRECT performs all calculations on points within the unit cube. The final solution is mapped back to the original hyper-box before being returned to the user.

l is a user-supplied double-precision vector of length n. l is the vector containing the lower bounds for the n independent variables.

cheat is a user-supplied integer. If set to one, the algorithm tries to "cheat". This means, if the approximated Lipschitz constant is higher than the value of *kmax*, then it is set to *kmax*. In some applications this speeded up the optimization process, but you should not use it when you start using DIRECT.

kmax is a user-supplied double-precision constant used when the algorithm is forced to "cheat". See also above.

writed is a user-supplied integer. If set to 1, the whole history of the optimization is stored, so it can be reloaded. Attention : The file which is created tends to become large!

loadh is a user-supplied integer. If set to 1, the algorithm loads the history of a run before, which was stored using the option *writed* = 1.

– **On Return**

x is the final point obtained in the optimization process. x should be a good approximation to the global minimum for the function in the hyper-box.

fmin is the value of the function at x

- **User-Supplied Functions and Subroutines**

  – The function evaluation subroutine. The name of this subroutine is determined by the user and must be declared **EXTERNAL**. The function should have the form (this is taken from the file DIRtestf.f):

```
function testf(x,n)
IMPLICIT None
integer n,ind
real*8 x(n)
real*8 testf,help
help = 0.0D0
DO 10,ind = 1,n
help = help + (x(ind)+2.D0)*(x(ind)+2.D0)
10 CONTINUE
testf = help*3
end
```

  – *DIRInitSpecific*
  This function can be found in DIRserial.f. You can include whatever application-specific initializations you have to do in this subroutine. Most of the time you will not need it.

1.4 Calling DIRECT (Parallel Version)

In this subsection the extra subroutines for the parallel version of DIRECT are explained. The subroutine *DIRInitSpecific* can be found the file DIRparallel.f together with the subroutines described below.

- *DIRSend_Common*
  If you use the parallel-version of DIRECT and you use common variables, you have to broadcast them to all processors. To do this, you should use the routine *DIRSend_Common*. Below is an example of this subroutine. The variables *test1* and *test2* are examples for a real*8 and an integer.

```
SUBROUTINE DIRSend_Common(l,u,n)
IMPLICIT None
include '/usr/include/pvm3/fpvm3.h'
Integer n
Real*8 l(n),u(n)
C+-------------------------------------------+
C— Problem-specific variables! —
C+-------------------------------------------+
Real*8 test1
Integer test2
C+-------------------------------------------+
C— PVM-specific variables! —
C+-------------------------------------------+
Integer info
CALL pvmfinitsend(PvmDataRaw,info)
IF (info .LT. 0) THEN
Write(*,*) "PVM-Error! Initializing Buffer"
STOP
END IF
CALL pvmfpack(INTEGER8,n,1,1,info)
IF (info .LT. 0) THEN
Write(*,*) "PVM-Error! Pack-fault"
STOP
END IF
CALL pvmfpack(Real8,l,n,1,info)
IF (info .LT. 0) THEN
Write(*,*) "PVM-Error! Pack-fault"
STOP
END IF
CALL pvmfpack(Real8,u,n,1,info)
IF (info .LT. 0) THEN
Write(*,*) "PVM-Error! Pack-fault"
STOP
END IF
```

```
CALL pvmfbcast(PVMALL,1001,info)
IF (info .LT. 0) THEN
Write(*,*) "PVM-Error! Broadcast-problem"
STOP
END IF
C Send problem-specific variables :
CALL pvmfinitsend(PvmDataRaw,info)
IF (info .LT. 0) THEN
Write(*,*) "PVM-Error! Initialising Buffer"
STOP
END IF
CALL pvmfpack(Integer8,test2,1,1,info)
CALL pvmfpack(Real8, test1 ,1,1,info)
CALL pvmfbcast(PVMALL,1001,info)
IF (info .LT. 0) THEN
Write(*,*) "PVM-Error! Broadcast-problem"
STOP
END IF
END
```

- *DIRSlave*
  This subroutine is the counterpart of *DIRSend_Common*. Here you have to get the common variables and unpack them.

2 Classical 1-D Lipschitzian Optimization

Let $N = 1$ and let f be Lipschitz continuous with constant γ on $[a, b]$. Therefore by setting $x' = a$ and $x' = b$ in (1) we get the following inequalities :

$$\begin{aligned} f(x) &\geq f(a) - \gamma(x - a) \\ f(x) &\geq f(b) + \gamma(x - b). \end{aligned}$$

With these inequalities we can define a piecewise linear function $\hat{f}$, which lies below f.

$$\hat{f}(x) = \begin{cases} f(a) - \gamma(x - a) & \text{if } x \leq x(a, b) \\ f(b) + \gamma(x - b) & \text{otherwise} \end{cases}$$

where

$$x(a, b) = [f(a) - f(b)]/(2\gamma) + [a + b]/2.$$

Note also that $\hat{f}$ has a minimum value of

$$B(a, b) = [f(a) + f(b)]/2 - \gamma(b - a)/2.$$

Figure (1) shows this. The idea is now to divide the search area into two intervals $I_1 = [a, x(a, b)]$ and $I_2 = [x(a, b), b]$, calculate the new values of x and B for each of these two intervals and choose a new interval to divide. Obviously the interval with the lowest value of B will be chosen.

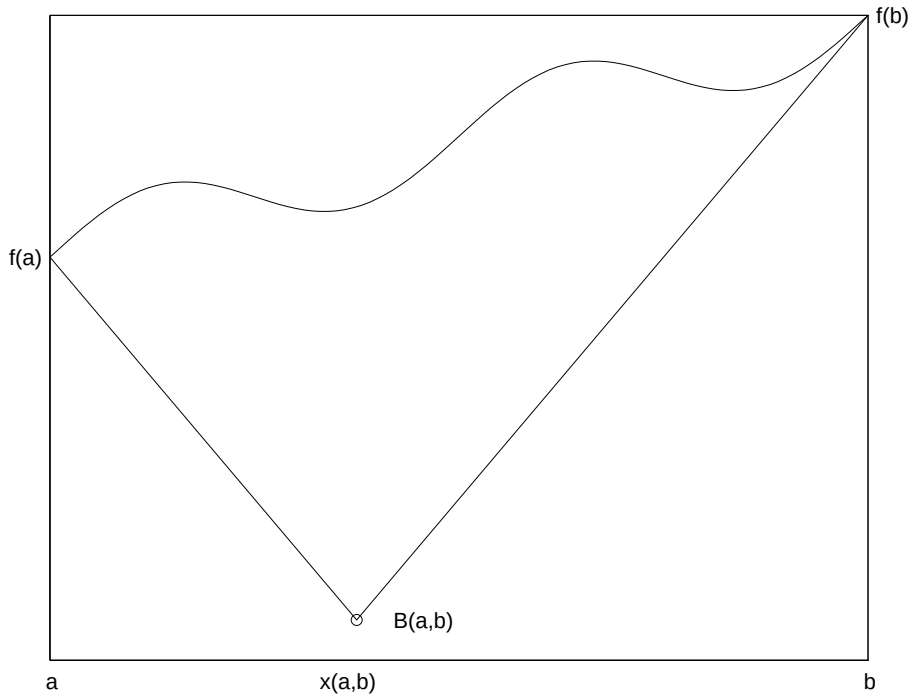

Figure 1: Example of f and $\hat{f}$

Following these ideas we get Piyavskii's [6] or also known as Shubert's [7] algorithm. They both discovered it independently. :

Algorithm 2.1 ***Piyavskii*** $(a, b, f, \gamma, numit)$

1. $n = 1, sample = 1, l_{sample} = a, u_{sample} = b$
2. *Calculate* B_1, x_1
3. *Do while* $n < numit$
   (a) $n = n + 1$
   (b) $l_n = x_n, u_n = u_{sample}, u_{sample} = x_n$
   (c) *Calculate* $B_n, x_n, B_{sample}, x_{sample}$
   (d) *Choose new interval to sample*

In this algorithm the first two steps are the initialization, where n is a counter-variable, *sample* is the interval in which the algorithm is sampling and l_{sample} and u_{sample} are the lower and upper bounds of the sampling area. This means the algorithm needs four vectors of length *numit*, where *numit* is the number of function-evaluations to be done. In the inner loop the counter-variable n is increased, after which the active interval is divided into two new intervals. In the next step the algorithm recalculates the values of x and B and then chooses a new interval to sample. This new interval will be the interval with the lowest

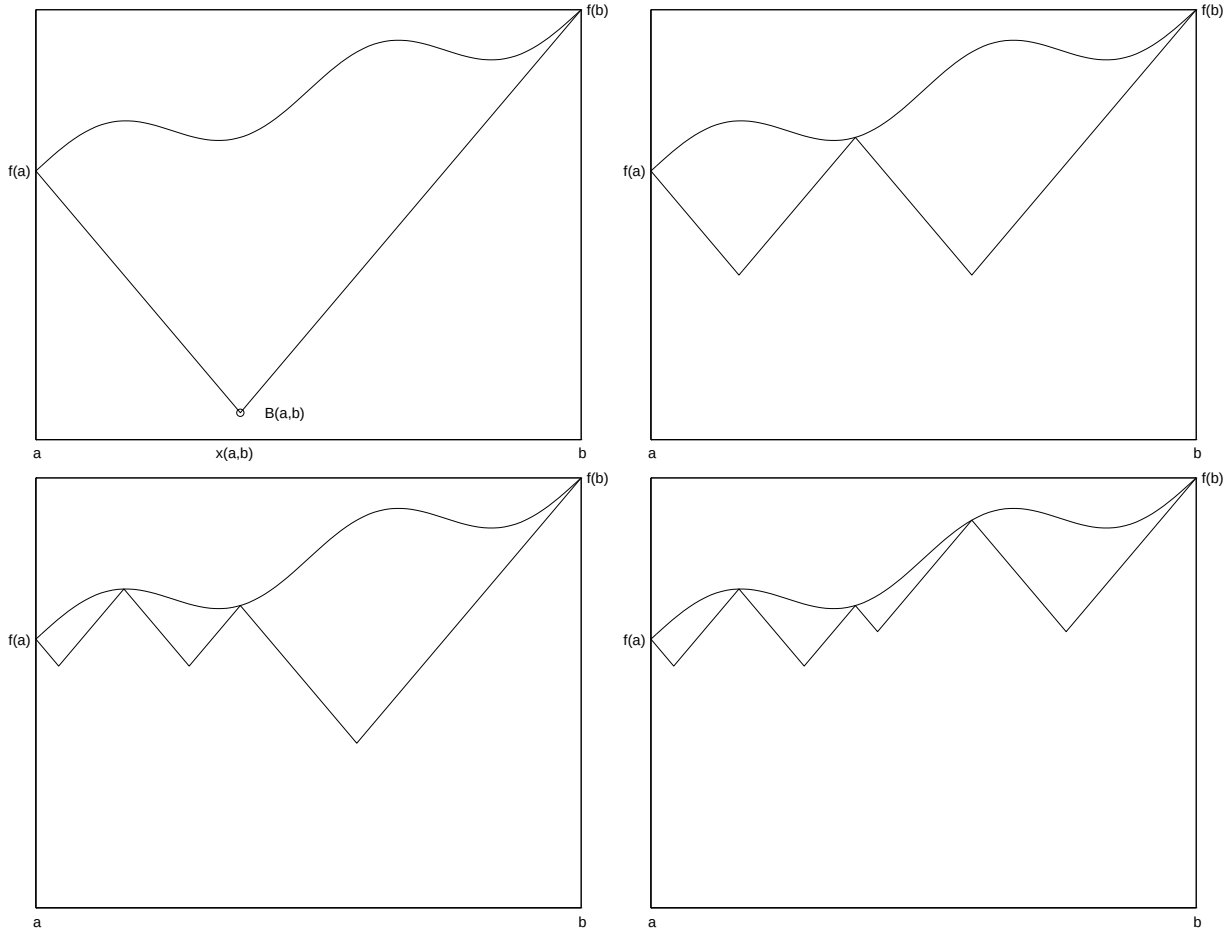

Figure 2: Example of the first 3 iterations of the Piyavskii-Algorithm

value of B. If there are more than one interval with the same value, one of these is chosen randomly. This is done until a given number of function-evaluation is done.

In Figure (2) an example of the first 3 iterations is shown and it can be seen that the piecewise linear function $\hat{f}$ becomes a better approximation of the real function in every iteration. Because of the look, $\hat{f}$ is sometimes called saw-tooth cover [2]. However, there are two problems to deal with :

- In higher dimensions this algorithm has the need to store 2^N corners for each Hyper-rectangle, where N is the dimension of the problem and more restrictive, has to evaluate the functions at all these points.
- In applications the Lipschitz-constant is normally not known if the function itself is a complicated problem or a simulation.

3 DIRECT Algorithm in 1-D

These problems are addressed by the DIRECT -Algorithm, where DIRECT stands for *di*viding *rect*angles. This algorithm was developed by D.R. Jones, C.D. Perttunen and B.E. Stuckman [3].

We start again with the inequality (1) in its one-dimensional formulation:

$$|f(x) - f(x')| \leq \gamma |x - x'| \; \forall x, x' \in [a, b]$$

Let $c = (a + b)/2$ and set $x' = c$ in (1). Then $\forall x \in [a, b]$

$$x \geq c : f(c) - \gamma(x - c) \leq f(x) \leq f(c) + \gamma(x - c)$$
$$x < c : f(c) + \gamma(x - c) \leq f(x) \leq f(c) - \gamma(x - c).$$

These two inequalities define an area in which the function has to lay. Furthermore we get a lower bound $D(a, b)$ for f in $[a, b]$ by setting $x = a$ or $x = b$ in these inequalities:

$$D(a, b) = f(c) - \gamma(b - a)/2.$$

In figure (3) this area (shaded) is shown for the function

$$f(x) = \sin((x - \frac{1}{2})4\pi) + 6x + 2, \; \forall x \in [0; 1].$$

3.1 Area-Dividing Strategies

One major point for the Shubert and the DIRECT algorithm is how to divide the interval into subintervals. The DIRECT algorithm divides the interval into three subintervals. We need three so that we can save the properties that the point we already have evaluated is a middle point of one of these intervals. This means in every step we have to evaluate the function at two new points. Figure (4) is a comparison of the two dividing strategies.

3.2 Potentially optimal intervals

The next questions is how to decide which interval to choose and if this can be done without knowing the Lipschitz-constant γ. Therefore suppose we have already partitioned the minimization area in M intervals $[a_i, b_i]$ with center c_i. Create a graph with the length $(b_i - a_i)/2$ of each of these intervals on the x-axis and $f(c_i)$ on the y-axis, as done in figure (5). If you lay a line with slope γ through each of these points, the intersection of this line and the y-axis is the point $(0, D(a_i, b_i))$, where $D(a_i, b_i)$ is the lower bound for the function in this interval. Therefore it is clear that the interval with the lowest value of $D(a_i, b_i)$ will be chosen as the one in which the algorithm will sample in the next iteration. If a line with slope γ is vertically translated from below the set of data-points, the first data-point which intersects the line will be the corresponding data-point of this interval. Note that there can be more than one interval with the same value of $D(a_i, b_i)$, but they will all lie on one line.

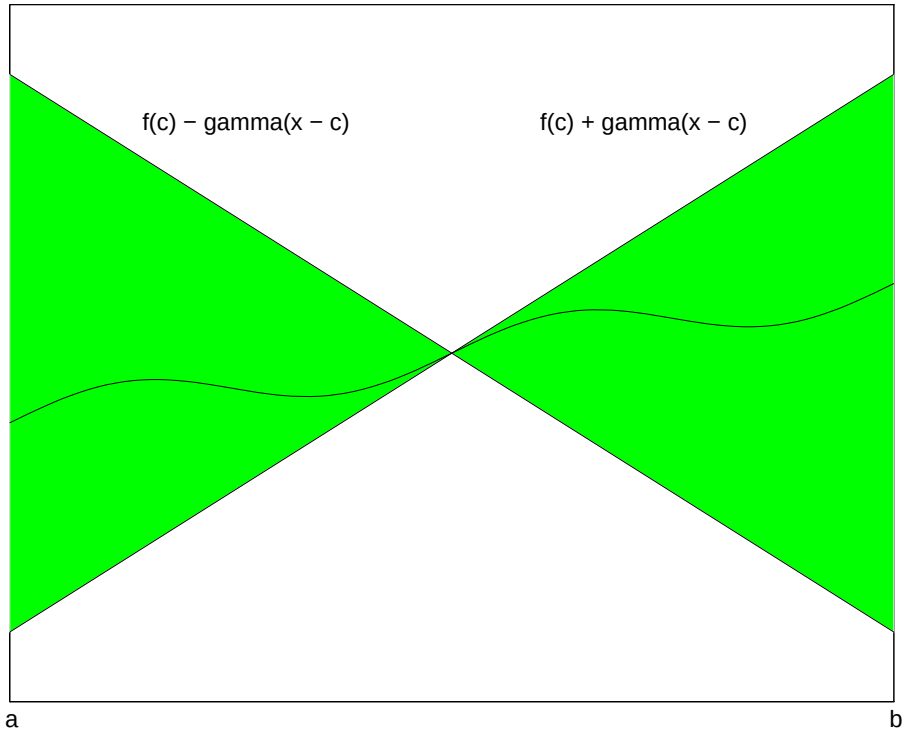

Figure 3: $f(x) = \sin((x - \frac{1}{2})4\pi) + 6x + 2$

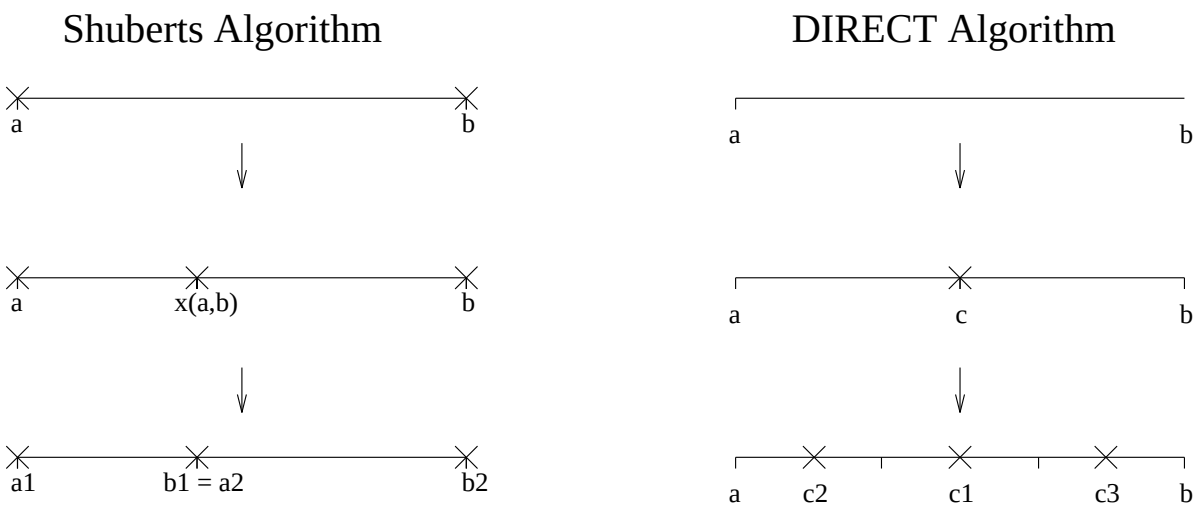

Figure 4: Dividing strategies

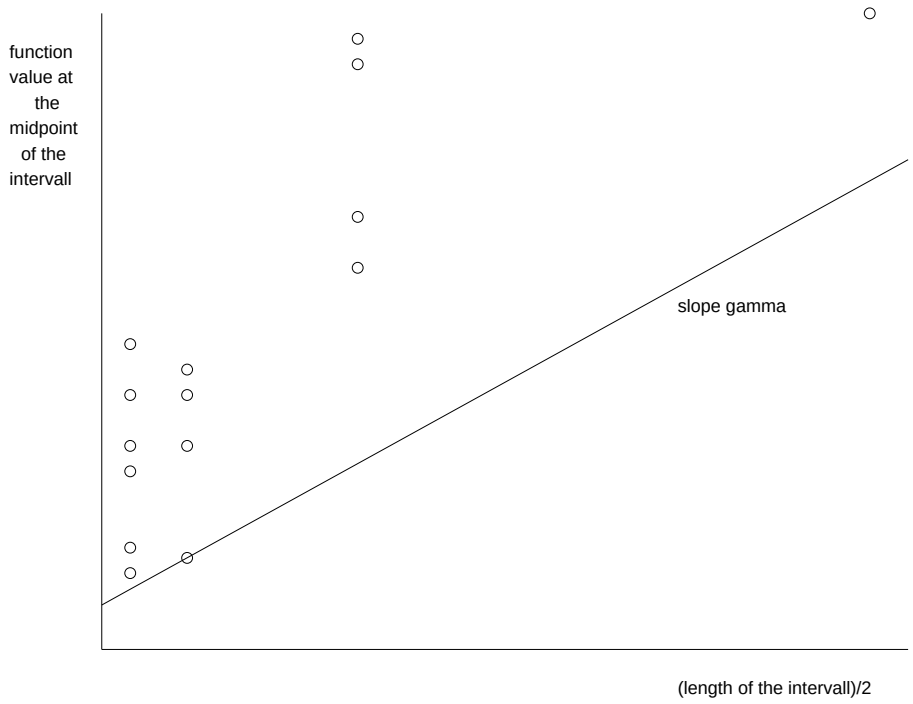

Figure 5: Interval selection graph

As stated before, in many applications the Lipschitz constant γ is not known. Therefore we try to get an estimate for the Lipschitz constant based on the known data. This is done by looking at the convex hull of the data-points mentioned before. For all data-points at the lower side of the convex hull there exists some constant $\tilde{K}_i$ such that the corresponding interval would be the choice if $\gamma = \tilde{K}_i$. It can be easily seen that from several intervals with the same length only the one with the lowest function-value at the middle point can be chosen. A formal definition of this is given in definition 3.1 and an example is given in figure (6).

Definition 3.1 *Let $\epsilon > 0$ be a positive constant and f_{min} be the current best function value. Interval j is said to be potentially optimal if there exists some rate of change constant $\tilde{K} > 0$ such that*

$$f(c_j) - \tilde{K}[(b_j - a_j)/2] \leq f(c_i) - \tilde{K}[(b_i - a_i)/2], \forall i \quad (2)$$

$$f(c_j) - \tilde{K}[(b_j - a_j)/2] \leq f_{min} - \epsilon|f_{min}|. \quad (3)$$

In the definition inequality (2) expresses the property described before and inequality (3) forces to have the possibility of a sufficient decrease in the interval. ϵ was set in all my calculations to 10^{-2}, which works fine, but in [3] it is proposed to set ϵ between 10^{-3} and 10^{-7}.

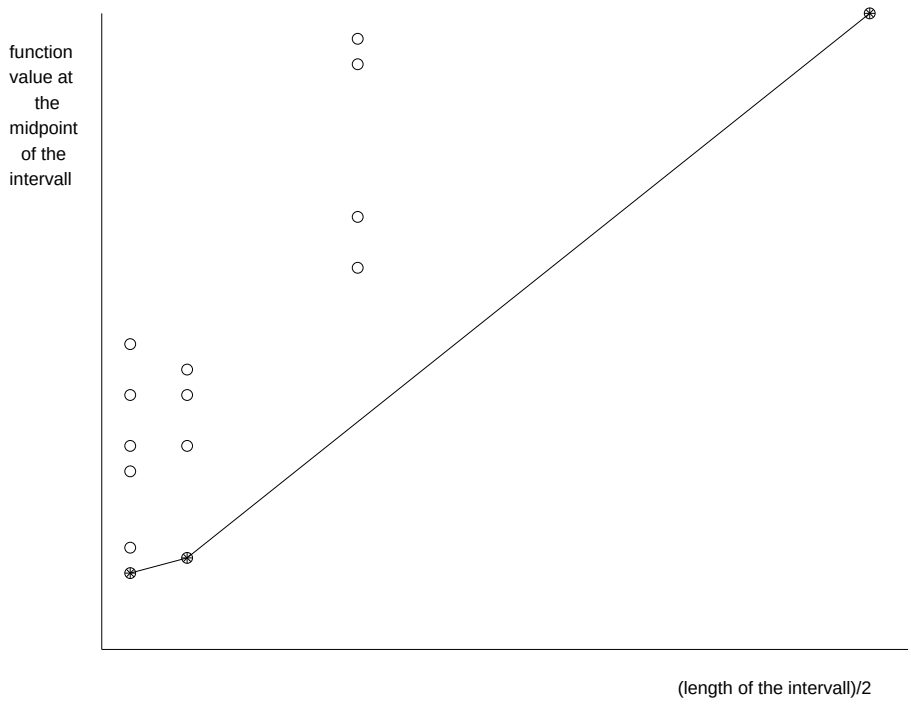

Figure 6: Potentially optimal intervals

Taken all this together, we get the one-dimensional DIRECT algorithm.

Algorithm 3.1 ***DIRECT -1D*** $(a, b, f, \epsilon, numit, numfunc)$

1. $m = 1, c_1 = (a+b)/2$
2. *evaluate* $f(c_1), f_{min} = f(c_1), t = 0$
3. *Do while* $t < numit$ *and* $m < numfunc$
   (a) *Identify the set* S *of potentially optimal intervals*
   (b) *DO while* $S \neq \emptyset$
      i. *Take* $j \in S$
      ii. *Sample new points* (c_{m+1}, c_{m+2})*, update borders*
      iii. *Evaluate* $f(c_{m+1}), f(c_{m+2})$*, update* f_{min}
      iv. *Set* $m = m+2, S = S \setminus \{j\}$
   (c) $t = t+1$

In the algorithm again the first two steps are the initialization. The variable m is a counter for the number of function evaluations done and t is a counter for the number of iterations. We do not have a better termination criteria than to stop after $numit$ iterations or after $numfunc$ function evaluations. The identification of the potentially optimal intervals is done using the definition 3.1. Note also that there are two possibilities of parallelism here. These are the inner loop and the function evaluations inside the inner loop.

4 The multidimensional DIRECT algorithm

For the multidimensional DIRECT algorithm we make the assumption that Ω is the N-dimensional unit hypercube.

We make this assumption for two reasons. The first is that we want the function to be scaled. The second reason is that with this assumption the hyper-rectangles we have to deal with can only have side-length of a power of $\frac{1}{3}$. These values can be calculated once, stored in an array and only the pointer to the position in this array have to be stored.

The main difference in higher dimension compared to the 1-D case is the problem of the division of the search space. For better understanding we start with a hypercube and describe then the more general case of a hyper-rectangle.

4.1 Dividing in higher dimensions

(a) Dividing of a hypercube

Let c be the center point of the hypercube. Sample the points $c \pm \delta e_i$, where δ equals 1/3 of the side-length of the cube and e_i is the i-th Euclidean base-vector. Define w_i by

$$w_i = \min\{f(c + \delta e_i), f(c - \delta e_i)\}.$$

Divide then in the order given by the w_i, starting with the lowest w_i. This means the hypercube is first divided along the direction with the lowest w_i, the remaining area is then divided along the direction of the second lowest w_i and so on until the hypercube is divided in all directions.

(b) Dividing of a hyper-rectangle

In this case we only divide along the longest sides of the hyper-rectangle, which assures us that we get a decrease in the maximal side-length of the hyper-rectangle.

Figure 7 is an example of the division of a hypercube. Here

$$\begin{aligned} w_1 &= \min\{5, 8\} = 5 \\ w_2 &= \min\{6, 2\} = 2. \end{aligned}$$

Therefore we divide first along the x_2-axis and then in the second step the remaining rectangle along the x_1-axis. Figure 8 shows the next step in the algorithm. The shaded areas are the areas which will be divided in this iteration. The second box shows that the bigger area is only divided once, because it is a rectangle and therefore it is divided along the longest side. The smaller area is a cube and therefore gets divided twice in the same way as seen before.

Therefore we get the following algorithm DIVIDE.

Algorithm 4.1 ***DIVIDE***

1. *Identify the set I of dimensions with maximum side length d, set $\delta = d/3$*
2. *Sample f at $c \pm \delta e_i, i \in I$*
3. *Calculate w_i and divide according to the values of the w_i*

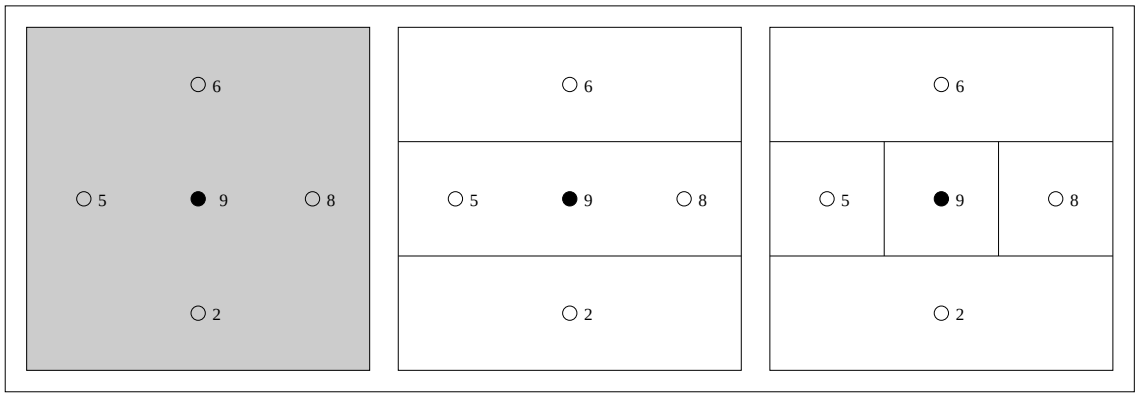

Figure 7: Dividing of a Hypercube

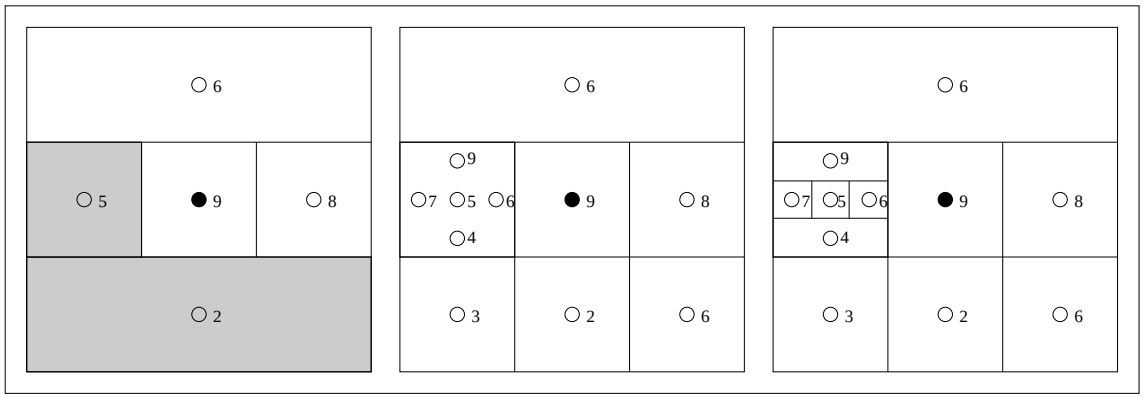

Figure 8: Next step in the algorithm

4.2 Potentially optimal hyper-rectangles

The definition of potentially optimal hyper-rectangles does not differ much from definition 3.1. The biggest difference is that instead of the interval length as in the one-dimensional case we take the length of the longest side of the hyper-rectangle. The explanations for the one dimensional case are also true for the multidimensional case.

Definition 4.1 *Let $\epsilon > 0$ be a positive constant and f_{min} be the current best function value. A hyper-rectangle j is said to be potentially optimal if there exists some $\tilde{K} > 0$ such that*

$$\begin{aligned} f(c_j) - \tilde{K}d_j &\leq f(c_i) - \tilde{K}d_i, \forall i \\ f(c_j) - \tilde{K}d_j &\leq f_{min} - \epsilon|f_{min}|. \end{aligned}$$

With this definition we have everything we need for the `DIRECT` algorithm.

Algorithm 4.2 *`DIRECT`* $(a, b, f, \epsilon, numit, numfunc)$

1. *Normalize the search space to be the unit hypercube with center point c_1*
2. *evaluate $f(c_1), f_{min} = f(c_1), t = 0, m = 1$*
3. *Do while $t < numit$ and $m < numfunc$*
   (a) *Identify the set S of potentially optimal hyper-rectangles*
   (b) *DO while $S \neq \emptyset$*
      i. *Take $j \in S$*
      ii. *Sample new points, evaluate f at the new points and divide the hyper-rectangle with **Divide***
      iii. *Update $f_{min}, m = m + \Delta m$*
      iv. *Set $S = S \setminus \{j\}$*
   (c) $t = t + 1$

Here again the first two steps are the initialization. The identification of the set of potentially optimal hyper-rectangles is done again using the definition. Note that there are again the two stages of possible parallelism, the inner loop and the function evaluations inside the inner loop.

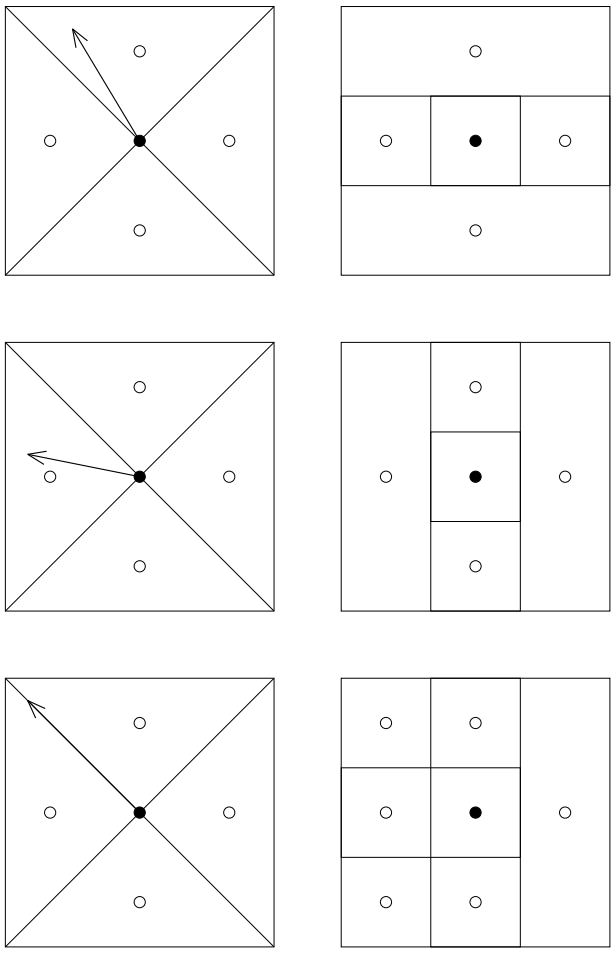

Figure 9: Dividing of a Hypercube

4.3 Dividing according to the steepest descent direction

Figure 9 shows another idea for dividing the hypercube. The arrow is here the steepest descent direction, approximated by central differences. This involves no new function-evaluations, all values are already known. The strategy now is that the hypercube is divided into parts by hyper-planes defined by the middle point and a vector pointing to a corner, which is orthogonal to the hyper-plane. A formal definition of this is given in definition 4.2.

Definition 4.2 *Let C be set set of coordinate-vectors of the corner-points of a hypercube and 0 be the middle point. Then each of the dividing hyper-planes $P(z)$ are defined by*

$$P(z) = \{x \in \mathbb{R} | x^T z = 0\} \; \forall z \in C$$

and the set D of the dividing hyper-planes is defined by

$$D = \{P(z) | z \in C\}$$

The idea is now to divide the hypercube in the algorithm such that the area, in which the steepest descent direction is, will be the biggest. If the steepest descent direction directly points towards a corner, we will sample extra points.

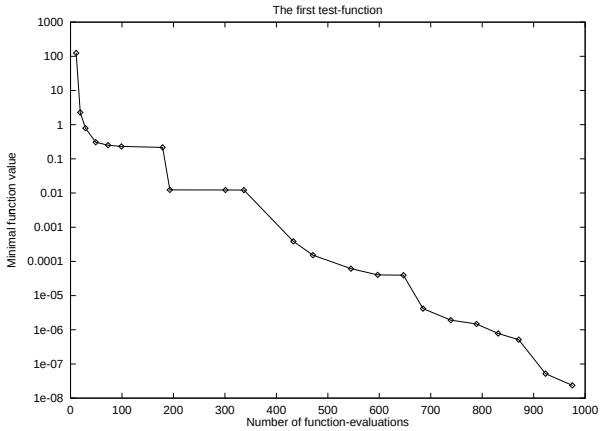

Figure 10: Plot of the reduction of the function value by DIRECT

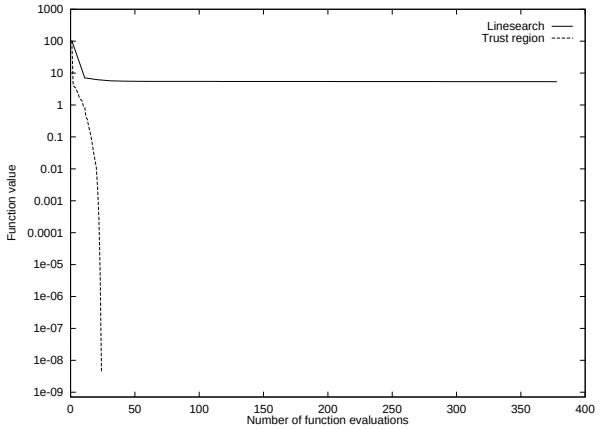

Figure 11: Plot of the reduction of the function value by steepest-descent and dog-leg trust region

5 Numerical results

5.1 Test functions

In this section I present some numerical results and compare them with a steepest descent line search and a dog-leg trust region algorithm as described in [4]. In the DIRECT algorithm I set $\epsilon = 10^{-2}$ and sampled in the first problem in the area $[-5;5]^2$, in the second problem in the area $[-5;5]^5$ and in the last problem in $[-5;4]^2$. For the steepest descent algorithm I set $\alpha = 10^{-4}$ (measures sufficient decrease) and the termination tolerances to 10^{-6}. For the dog-leg trust region algorithm I used the same termination tolerances, $\mu_0 = 10^{-4}, \mu_{low} = .25, \mu_{high} = .75$ and the start trust region radius $\Delta = 1$.

For these comparisons I have three different test functions :

$$f_1(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \quad (4)$$

$$f_2(x) = \sum_{i=1}^{5} x_i^2 \quad (5)$$

$$f_3(x) = x_1^2 + x_2^2 + 0.3z\sin(13\pi z)^2 \quad (6)$$

$$, \text{ where } z = x_1^2 + x_2^2 + 0.0001 \quad (7)$$

For the first function (4) it can be seen in the figures 10 and 11 that the dog-leg trust region algorithm is much faster than the DIRECT algorithm, but the steepest descent algorithm does not converge at all (even if we do more function evaluations).

In figure 12 you can see where the algorithm DIRECT has sampled points. Note that the special structure of the function (it is called the "Banana-function") can be seen by looking at this graph.

In figures 13 and 14 a comparison of the methods for the problem (5) is shown. Here it can be easily seen that the DIRECT algorithm should not be used for "easy" problems, where

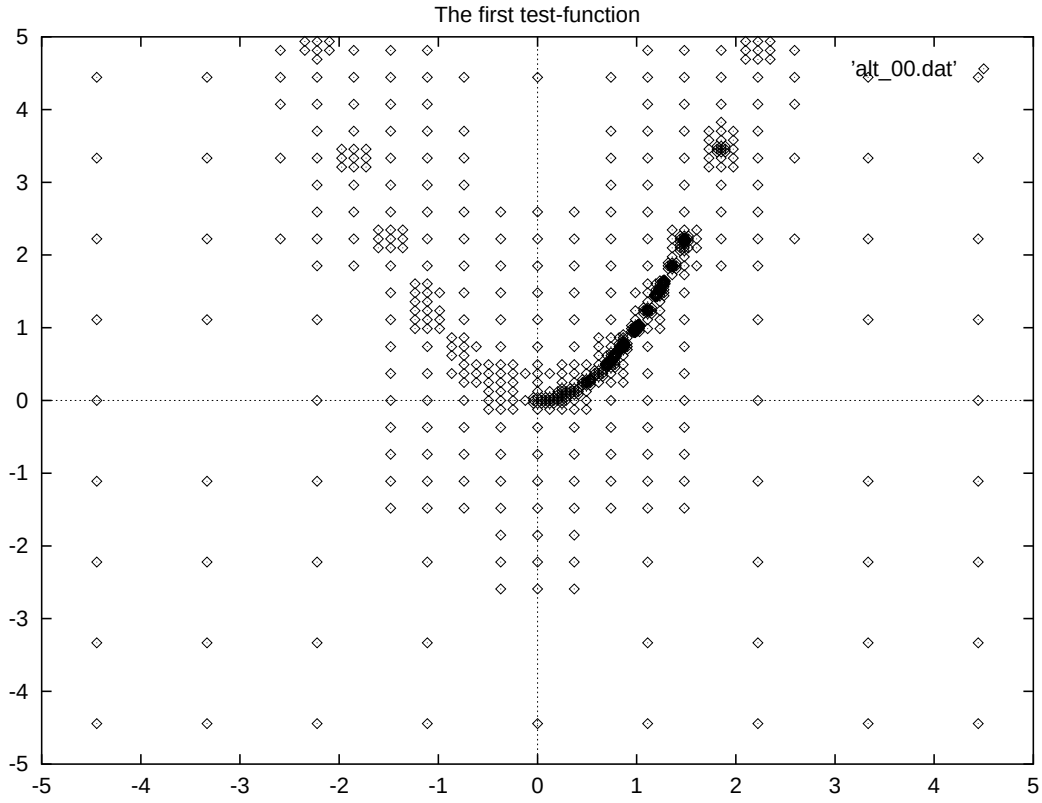

Figure 12: Plot of the sample points for (4)

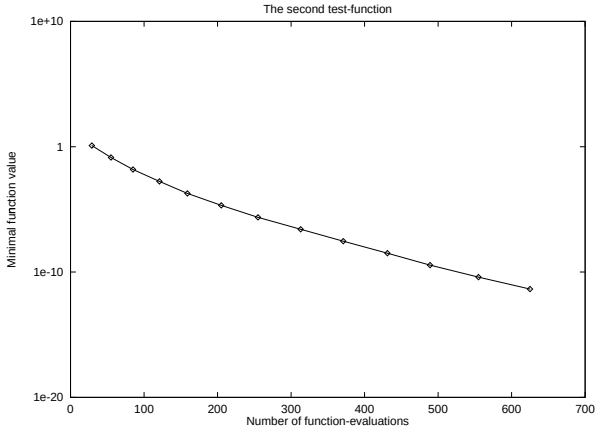

Figure 13: Plot of the reduction of the function value by `DIRECT`

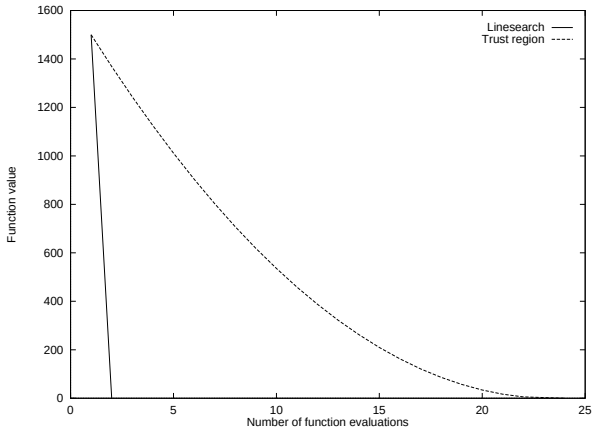

Figure 14: Plot of the reduction of the function value by steepest-descent and dog-leg trust region

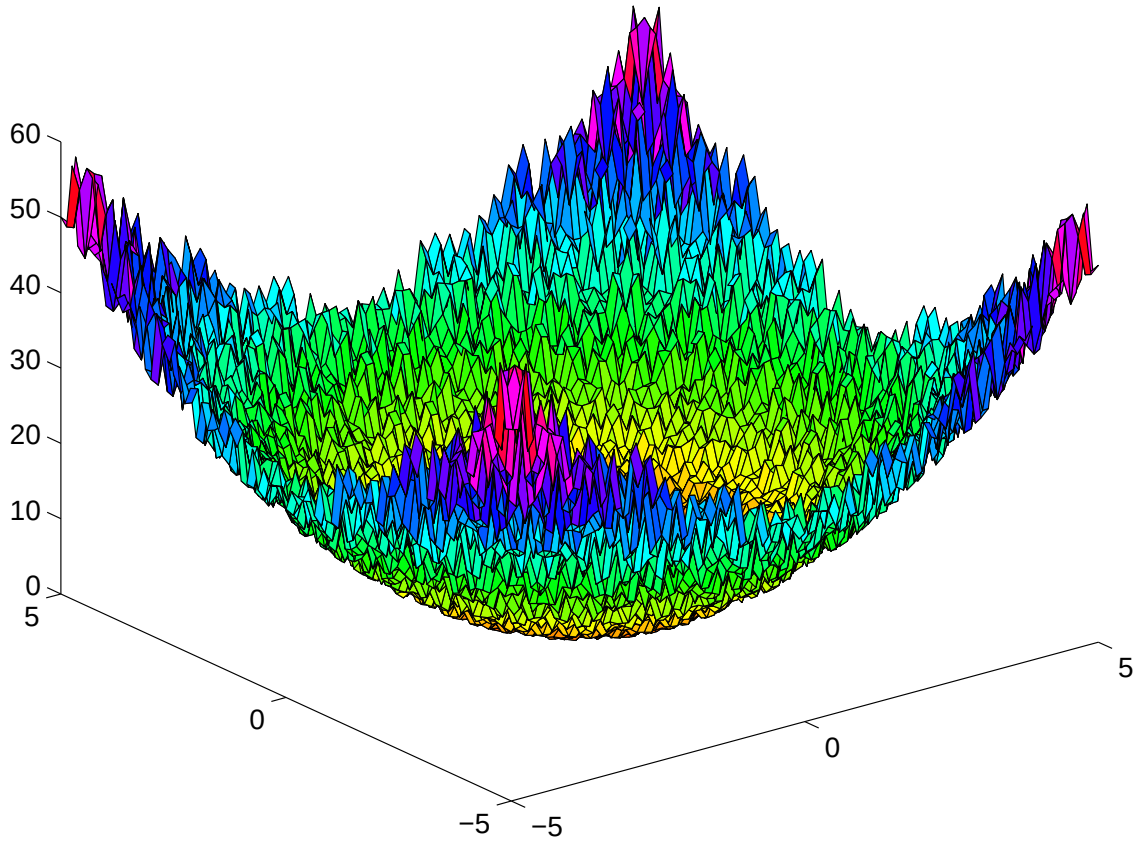

Figure 15: Plot of the third test function)

the standard algorithm solve the problem very fast.

Figure 15 shows a plot of the graph of the third test function. This function has a lot of local minima, with which the dog-leg algorithm can not handle with, as shown in figure 17. The steepest descent algorithm can solve this problem. In figure 18 the plot of the sample points for the first nine iterations and the last iteration can be seen. In the plot of the last iteration the concentration in the area of interest can be seen, but also that the algorithm still exploits other areas. It can also be seen that there are no "big" areas which are not tested yet.

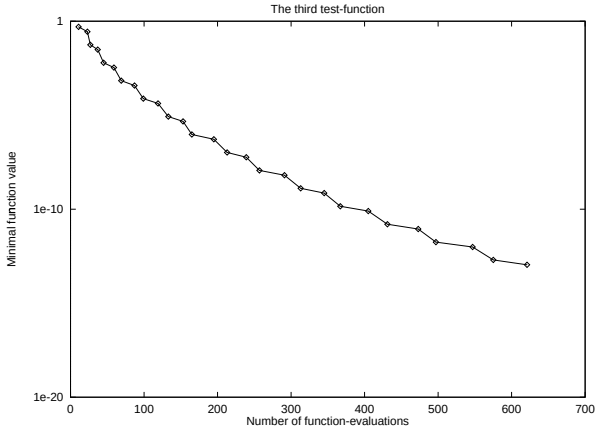

Figure 16: Plot of the reduction of the function value by DIRECT

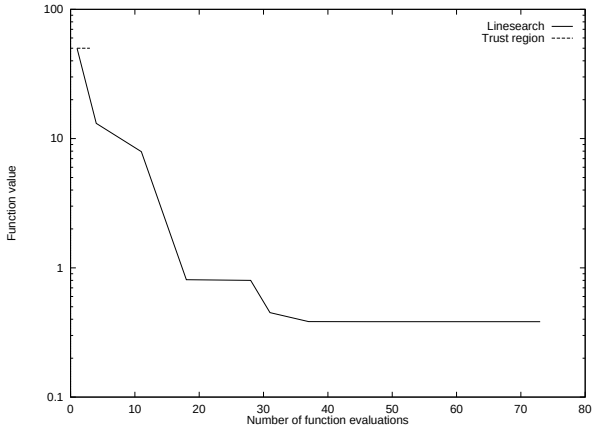

Figure 17: Plot of the reduction of the function value by steepest-descent and dog-leg trust region

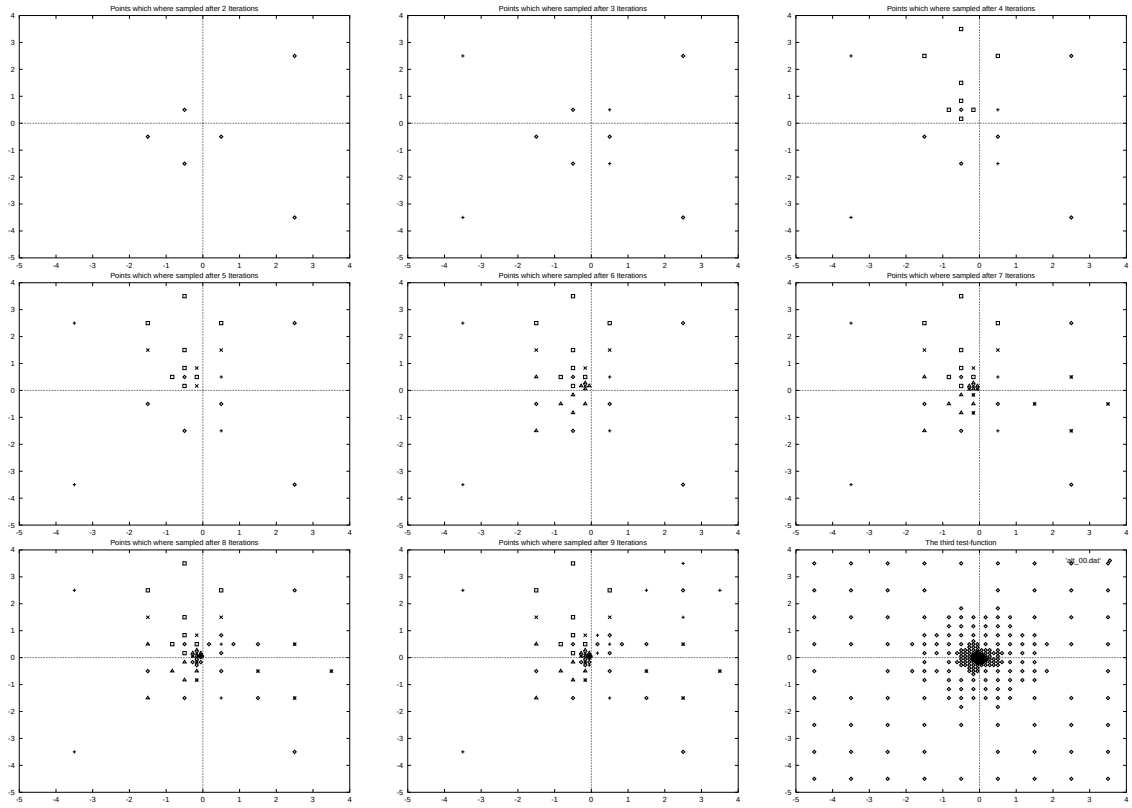

Figure 18: Plot of the sample points for (6)

Figure 19: A typical push-rod valve train

5.2 Behavior in an application

5.2.1 The valve-train problem

Optimal design of high-speed valve-trains requires the use of an accurate analytical model. While the governing differential equations are important, the coefficients (or parameters) used in these equations are equally important. Since many of the parameters are difficult to measure directly, parameter identification based on experimental data is required to assure model accuracy.

This is the background for the simulation, in which the `DIRECT` algorithm was used and compared with IFFCO. Here we are looking at push-rod engines (not overhead cams), used in NASCAR racing. Figure 19 shows a schematic of a typical push-rod type valve train. These engines run at 9000 RPM for about 500 miles and the valve bounce is an important failure mode, because it is a major limit on the engine speed and therefore a major limit on the overall performance of the engine. The model is described in [5].

There are 12 parameters to be identified. These are

- The stiffness coefficients $K_j, j = 1, \ldots, 5$
- The damping coefficients $C_j, j = 1, \ldots, 5$
- T_f friction torque at rocker arm
- F_f friction force at valve stem

where the coefficients represent the following

- K_1 and C_1 cam and lifter

Figure 20: Schematic of the valve train model

Figure 21: Cost function variation with two parameters

- K_2 and C_2 push-rod
- K_3 and C_3 rocker arm
- K_4 and C_4 valve and valve seat
- K_5 and C_5 cam lifter and head.

A schematic of the model used can be seen in figure 20. Figure 21 shows three dimensional sections of the optimization landscape. It can be seen, that there are several local minima and that the function makes big jumps. Standard optimization algorithms do not work for this problem.

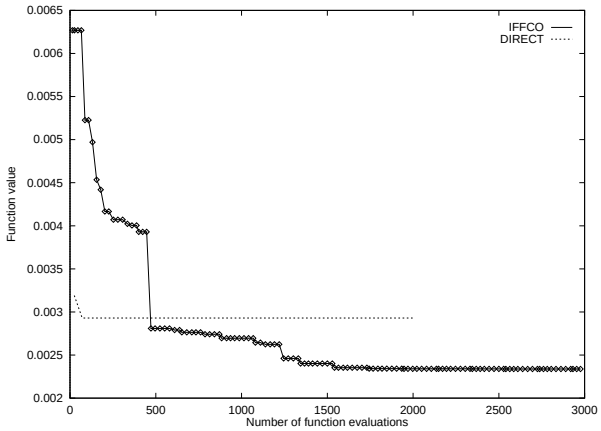

Figure 22: Comparison of `DIRECT` and IFFCO

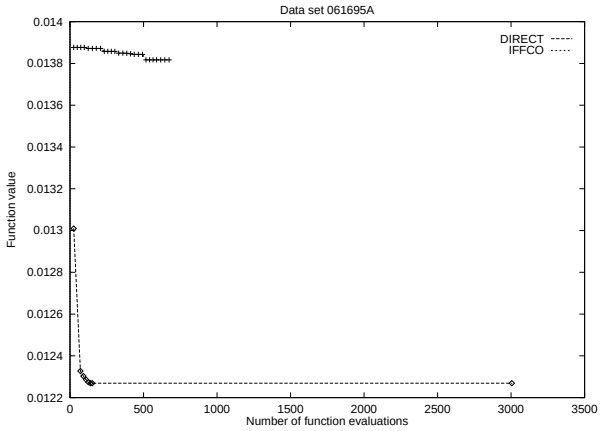

Figure 23: Comparison of `DIRECT` and IFFCO, harder problem

5.2.2 Results with different datasets

In figures 22, 23 and 24 a comparison with the algorithm IFFCO can be seen. IFFCO needs a starting point, which is given by the mechanical engineers who developed the simulation. These starting points for the different datasets seem not to be very good, so that `DIRECT` gets much better results in the starting phase and in the second problem it finds a much better solution, where IFFCO stays in a local minimum. Later in the process `DIRECT` does no improvement for a long time, whereas IFFCO gets better results. One of the problems of `DIRECT` is that there can be no improvement in the function-value for many iterations and then all of a sudden a jump, which seems not to be predictable. Also it can be seen from these results, that it might be a good idea to use `DIRECT` as a starting point generator for IFFCO.

5.2.3 Parallel behavior

In this section the parallel behavior of the actual implementation of the `DIRECT` algorithm is described. Only the function evaluations were parallelized. In Table 1 I show timings

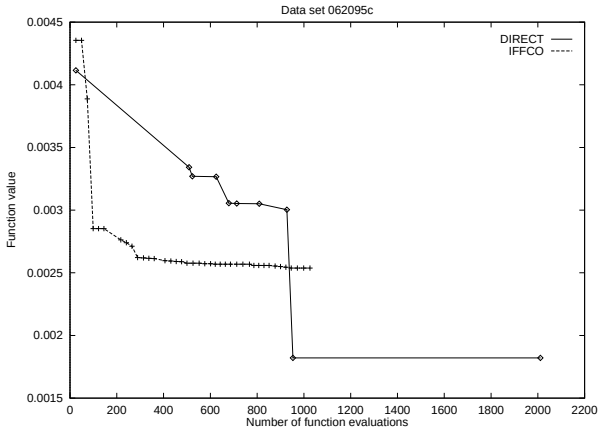

Figure 24: Comparison of `DIRECT` and IFFCO, third dataset

Number of processors	Time (in Seconds)	FLOPS (total)	FLOPS (per proc.)
2	3862.49	3.13 x 10^6	1.56 x 10^6
4	2015.20	5.59 x 10^6	1.40 x 10^6
8	1120.87	9.62 x 10^6	1.20 x 10^6
16	723.28	14.12 x 10^6	0.88 x 10^6

Table 1: Run-time for different numbers of processors

for different numbers of processors. These timings were done on a 32 processor CRAY T3D ©located at the North Carolina Supercomputing Center in the Research Triangle Park, North Carolina using 'ja'. The program did 151 function evaluations and reduced the function value from 0.138895E-01 to 0.122690E-01. It can be seen, that for this particular problem a good number of processors to use is 8. This can be explained by the dimension of the problem, which is 12. Therefore by dividing a hypercube 24 function-evaluations are done. This number is divided evenly by 8, so that me get a maximum performance, because the idle time of the processors is reduced. Table 2 shows the time spend performing different task types as given by APPRENTICE ©, a program to analyze the parallel performance of T3D programs. This program also gives a lot of data about how single subroutines behave.

Number of processors	"work" instructions	loading & data caches	waiting on PVM	read & write	other
2	9.46%	11.13%	9.49%	0.03%	69.89%
4	8.47%	9.91%	19.09%	0.02%	62.51%
8	7.29%	8.51%	30.42%	0.02%	53.77%
16	5.35%	6.23%	48.94%	0.04%	39.45%

Table 2: Where the program spends its time

6 Proofs

6.1 The estimation for the Lipschitz-constant

Let f be Lipschitz-continuous with smallest possible Lipschitz-constant γ and be defined on $J_0 = [0,1]^N$, where N is the dimension of the problem. Let I_n be all indices in the $n-$th iteration of `DIRECT` . c_i is the length of the shortest side of the i-th set J_i created by `DIRECT` with middle point x_i. $c_i = 1/3^k$ by definition. Therefore $\forall i, j \in I_n$ we can say

$$\| x_i - y \| \geq c_i/2 \; \forall y \in [0,1]/J_i$$

$$\Rightarrow \| x_i - x_j \| \geq \frac{c_i + c_j}{2} \geq \frac{|c_i - c_j|}{2}$$

$$\Rightarrow \frac{2}{|c_i - c_j|} \geq \frac{1}{\| x_i - x_j \|}$$

We know that

$$\gamma = \max_{x \in J_0} \max_{y \in J_0} \frac{|f(x) - f(y)|}{\| x - y \|} \geq \max_{i \in J_n} \max_{j \in J_n} \frac{|f(x_i) - f(x_j)|}{\| x_i - x_j \|} = K_n.$$

We also know that

$$\hat{K}_n = \max_{i \in J_n} \max_{j \in J_n} 2 \frac{|f(x_i) - f(x_j)|}{|c_i - c_j|} \geq \max_{i \in J_n} \max_{j \in J_n} \frac{|f(x_i) - f(x_j)|}{\| x_i - x_j \|} = K_n.$$

So we get $\gamma \geq K_n$ and $\hat{K}_n \geq K_n$. For $n \to \infty$ $K_n \to \gamma$ so we get that $\exists n_0 \in \mathbb{N}$:

$$\forall n \geq n_0 : \; \hat{K} \geq \gamma.$$

So we can say that the approximate Lipschitz-constant $\hat{K}$ used in `DIRECT` is at least in the end an overestimate for the Lipschitz-constant γ. It can also be seen that this estimate can be calculated with much less computations than K_n, because no norms need to be computed.

7 Conclusions and future work

Conclusions

- `DIRECT` exploits the search space well, it has a good global behavior, but a bad local behavior
- `DIRECT` can do many function evaluations without improvement
- `DIRECT` may be a good starting-point generator

Future work

- Find better termination criteria
- Explain good global behavior
- Find estimates for maximum number of function evaluations (for special families of functions)
- Find error estimates
- Proof convergence ?

References

[1] P. Gilmore. IFFCO: Implicit Filtering for Constrained Optimization. Technical Report CRSC-TR93-7, Center for Research in Scientific Computation, North Carolina State University, May 1993. available by anonymous ftp from math.ncsu.edu in pub/kelley/iffco/ug.ps.

[2] Reiner Horst, Panos M. Pardalos, and Nguyen V. Thoai. *Introduction to Global Optimization.* Nonconvex Optimization and its applications. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1995.

[3] D.R. Jones, C.D.Perttunen, and B.E.Stuckmann. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, 79:157, October 1993.

[4] C. T. Kelley. *Iterative Methods for Optimization.* 1998. Department of Mathematics, North Carolina State University, to be published by SIAM in 1999.

[5] D. Kim and J.W. David. A combined model for high-speed valve train dynamics, partly linear and partly nonlinear. *SAE Technical Paper Series 901726.*

[6] S. A. Piyawskii. An algorithm for finding the absolute extremum of a function. *USSR Computational Mathematics and Mathematical Physics*, 12:57–67, 1972.

[7] B. Schubert. A sequential method seeking the global maximum of a function. *SIAM J. Numer. Anal.*, 9:379–388, 1972.