TÉCNICO LISBOA

Aircraft Optimal Design

Aerospace Engineering

José Miguel Fonseca Santos

83704

2nd Assignment

08/10/2019

Contents

1 Optimality Conditions

1.1 Gradient and Hessian

The function to be evaluated is defined by the above expression:

$$f(\boldsymbol{x}) = x_1^4 - x_1^2 x_2 + x_2^2 + \frac{1}{2}x_1^2$$

In order to compute the gradient vector, it is simply necessary to calculate the partial derivatives of the function. The gradient can be interpreted as the direction and rate of fastest increase since it points this direction.

$$\nabla \cdot f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 4x_1^3 - 2x_1 x_2 + x_1 \\ -x_1^2 + 2x_2 \end{bmatrix}$$

The Hessian matrix also follows the process of the gradient, proceeding with the calculation of the second derivatives.

$$\nabla^2 \cdot f \equiv H(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 12x_1^2 - 2x_2 + 1 & .2x_1 \\ -2x_1 & 2 \end{bmatrix}$$

1.2 Local minimizer verification

The next step is to replace the point $\boldsymbol{x}* = (0, 0)$ in the gradient and Hessian expressions.

$$\nabla f(0, 0) = (0, 0)$$

$$H(0, 0) = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

The gradient equals zero. This implies that $\boldsymbol{x}*$ is a local maximum or a local minimum, since there is no single direction of increase. If the Hessian is positive definite at $\boldsymbol{x}*$, then $f(\boldsymbol{x}*)$ attains an isolated local minimum. Since the determinants of all upper-left sub-matrices are positive, the Hessian matrix is confirmed to be positive definite and $f(\boldsymbol{x}*)$ a global minimum.

1.3 *Contour* and *quiver* plots

With the aid of MATLAB the *contour* and *quiver* plots are shown in figures 1 and 2.

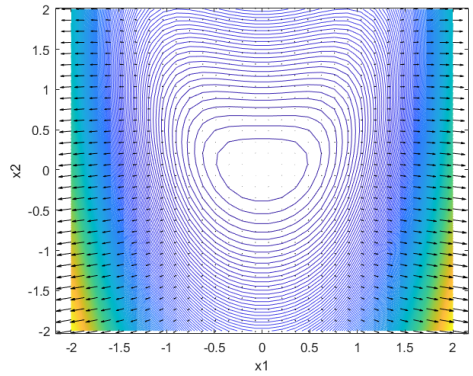

Figure 1: *Contour* and *quiver* plot

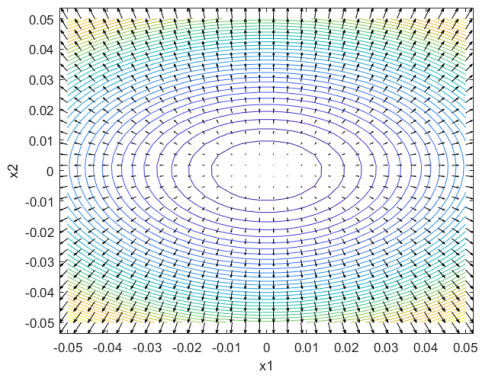

Figure 2: Domain zoomed around (0,0)

Regarding the *contour* plot, one can easily verify that (0,0) is a minimum due to the number of contour levels decreasing when around (0,0). The *quiver* plot shows the direction and magnitude of the gradients. Near (0,0) these vectors are 0 and point outwards, which gives graphical confirmation about (0,0) being a minimum.

2 Unconstrained Minimization

The Goldstein-Price Function is a continuous, non-convex, differentiable function usually evaluated on $x \in [-2, 2]$ and $y \in [-2, 2]$. It is a useful function to evaluate characteristics of optimization algorithms, such as the convergence rate, precision, and robustness. It is defined by the above expression:

$$f(x_1, x_2) = [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)]$$
$$[30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)]$$

Using MATLAB the *contour* and *surf* fucntions were used in the domain referred above. In addition, the plots obtained were verified.

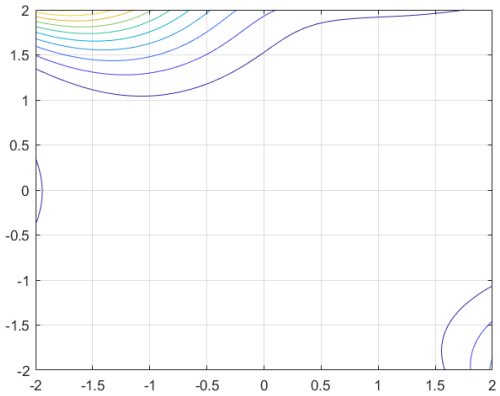

Figure 3: *Contour* - Goldstein-Price

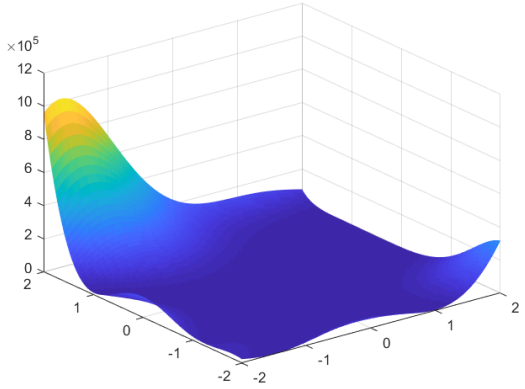

Figure 4: *Surf* - Goldstein-Price

2.1 *fminunc* parameters and default values

In this subsection, the global minimum of the Goldstein-Price Function will be calculated with the aid of the MATLAB function *fminunc*. Different algorithms will be used, as well as initial guesses, allowing the study of the consequences of these changes. To introduce some parameters and default values of these function, the tables bellow are presented.

Algorithms	**Default**
Algorithm	'quasi-newton'
MaxFunctionEvaluations	100*numberOfVariables
FiniteDifferenceType	'forward'
MaxFunctionEvaluations	100*numberOfVariables
MaxIterations	400
OptimalityTolerance	1×10^{-6}

Table 1: Default values for *fminunc*

exitFlag	**Reason**
0	MaxIterations or number of function evaluations exceeded MaxFunctionEvaluations.
1	Magnitude of gradient is smaller than the OptimalityTolerance tolerance.
2	Change in x was smaller than the StepTolerance tolerance.

Table 2: Reason behind the *exitFlag* value

The different initial values were chosen with one criteria in mind - trying to cover all the areas of the domain.

2.1.1 Quasi-Newton - Davidon–Fletcher–Powell

The first two algorithms to be used belong to the quasi-newton group, because they have small changes to the ordinary Newton methods. These methods use first order information, but build second order information, leading to an approximate Hessian. The approximate Hessian is built accounting for the curvature calculated during the most recent iteration in the DFP method. Comparing with the standard Newton method, the Quasi-Newton is faster since it does not need the second derivative of the function. Convergence rate is expected to reduce since the Hessian is merely an approximation. Varying the initial solution, the results obtained are shown in table 4. One can notice that the stopping criteria varies depending on the initial solution. For example, for an initial solution $(x_1, x_2) = (-1, 1)$, the algorithm stops due to the maximum number of function evaluations being reached. Increasing this value largely in order to get an idea of how many iterations it would take to stop with an *exitFlag* value of 1, the number of iterations obtained was 499, and a number of 1521 for the function evaluation counter.

$(\boldsymbol{x0})$	(x_1, x_2)	f(x_1, x_2)	exitFlag	Iter.	F. Counts	Gradient
$(-1, 1)$	$(0.015, -1.008)$	3.118	0	59	201	$[9.773; -11.054]$
$(-1, 0)$	$(-3.9 \times 10^{-7}, -1.0)$	3.000	1	15	51	$[-4.3 \times 10^{-5}; -9.0 \times 10^{-4}]$
$(-1, -1)$	$(-0.600, -0.400)$	30.000	1	7	42	$[-1.1 \times 10^{-4}; -3.0 \times 10^{-4}]$
$(0, 1)$	$(0.007, -1.0)$	3.013	0	55	201	$[3.649; -1.494]$
$(0, 0)$	$(-0.600, -0.400)$	30.000	1	8	36	$[-2.4 \times 10^{-5}; -3.0 \times 10^{-5}]$
$(0, -1)$	$(-4.4 \times 10^{-9}, -1.0)$	3.000	2	1	15	$[-7.0 \times 10^{-6}; -1.0 \times 10^{-6}]$
$(1, 1)$	$(1.800, 0.201)$	84.007	0	65	201	$[-10.374; 16.068]$
$(1, 0)$	$(-1.7 \times 10^{-6}, -1.0)$	3.000	1	6	57	$[0.001; -0.002]$
$(1, -1)$	$(-0.600, -0.400)$	30.000	1	8	54	$[0.005; 0.003]$

Table 3: Quasi-Newton DFP results

2.1.2 Quasi-Newton - Broyden–Fletcher–Goldfarb–Shanno

The BFGS is usually considered the most effective quasi-Newton method. The difference from the DFP is in the way the inverse of the Hessian approximation matrix is calculated. Results for this method are shown on table 4. The *exitFlag* value is 1 for every initial solution, as opposed to the DFP method confirming faster convergence. Concerning the value for the minimum position, only for $\boldsymbol{x0} = (-1, 1)$ the solution differed. For the DFP method, the soltution was getting close to the global minimum, whereas for the BFGS method it got a converged solution in a local minimum. One can also notice that the number of iterations and function evaluations is abusively larger for the DFP method. However, keeping the maximum number of iterations limited, it is possible to reach, in this case, good but less precise solutions. With this constrain, computational cost and time is avoided.

$(\boldsymbol{x0})$	(x_1, x_2)	f(x_1, x_2)	exitFlag	Iter.	F. Counts	Gradient
$(-1, 1)$	$(-0.600, -0.400)$	30.000	1	15	51	$[0.278; 0.162]$
$(-1, 0)$	$(-1.1 \times 10^{-6}, -1.0)$	3.000	1	8	33	$[-4.0 \times 10^{-4}; -3.5 \times 10^{-4}]$
$(-1, -1)$	$(-0.600, -0.400)$	30.000	1	7	42	$[-9.5 \times 10^{-6}; -2.1 \times 10^{-4}]$
$(0, 1)$	$(-1.6 \times 10^{-4}, -1.0)$	3.000	1	11	63	$[0.084; -0.039]$
$(0, 0)$	$(-0.600, -0.400)$	30.000	1	7	33	$[-3.3 \times 10^{-5}; -6.4 \times 10^{-5}]$
$(0, -1)$	$(-4.4 \times 10^{-9}, -1.0)$	3.000	1	1	15	$[-7.0 \times 10^{-6}; -1.0 \times 10^{-6}]$
$(1, 1)$	$(1.800, 0.200)$	84.000	1	12	42	$[-9.5 \times 10^{-6}; -2.4 \times 10^{-5}]$
$(1, 0)$	$(-6.7 \times 10^{-7}, -1.0)$	3.000	1	6	54	$[-4.9 \times 10^{-4}; -7.3 \times 10^{-4}]$
$(1, -1)$	$(-0.600, -0.400)$	30.000	1	8	48	$[0.004; 0.005]$

Table 4: Quasi-Newton BFGS

2.1.3 Steepest Descent

The last algorithm to be analyzed is the Steepest Descent Mehtod. For each iteration, the gradient vector at each point is used to obtain the search direction. The gradient points in the direction of maximum increase. The Steepest Descent searches the opposite direction of the gradient vector, that is, the direction of maximum decrease. When it is not possible to decrease further, the algorithm stops. It is expected to be inefficient since the search direction is not always the exact direction to follow, and the method "zigzags" in the design space. Convergence is guaranteed, although it may take an infinite number of iterations. The *exitFlag* returned a value of 0 for 5 of the initial solutions. Again, increasing largely the number of maximum function evaluations and iterations for $\boldsymbol{x0} = (0, 0)$, the number of iterations obtained and function evaluations was, respectively, 24 and 441.

$(\boldsymbol{x0})$	(x_1, x_2)	f(x_1, x_2)	exitFlag	Iter.	F. Counts	Gradient
$(-1, 1)$	$(-0.599, -0.401)$	30.000	0	7	195	$[0.311; -0.852]$
$(-1, 0)$	$(8.2 \times 10^{-7}, -1.0)$	3.000	1	9	144	$[-5.8 \times 10^{-4}; -8.3 \times 10^{-4}]$
$(-1, -1)$	$(-0.600, -0.400)$	30.000	0	10	201	$[-0.600; -0.400]$
$(0, 1)$	$(-2.7 \times 10^{-5}, -1.0)$	3.000	1	8	186	$[-0.020; 0.030]$
$(0, 0)$	$(-0.567, -0.435)$	30.460	0	8	183	$[5.510; -18.704]$
$(0, -1)$	$(-4.4 \times 10^{-9}, -1.0)$	3.000	2	1	15	$[-7.0 \times 10^{-6}; -1.0 \times 10^{-6}]$
$(1, 1)$	$(1.785, 0.189)$	84.049	0	7	195	$[12.616; -25.564]$
$(1, 0)$	$(-1.4 \times 10^{-6}, -1.0)$	3.000	1	11	183	$[-6.0 \times 10^{-4}; -1.1 \times 10^{-4}]$
$(1, -1)$	$(-0.596, -0.404)$	30.007	0	10	192	$[2.223; -1.213]$

Table 5: Steepest Descent results

To have a graphical perspective of the location of the initial, intermediate and final solutions, two plots were made. The first one (figure 5) shows a 3D *surf* representation of the location of each iteration. One can notice that the first step is the bigger one since the point is located in a region where the maximum decrease value is high. After the second iteration, the steps become smaller since a low $f(\boldsymbol{x})$ value was obtained. Also, it is noticeable that, despite the initial solution being closer to the global minimum (figure 6), the final solution is a local minimum.

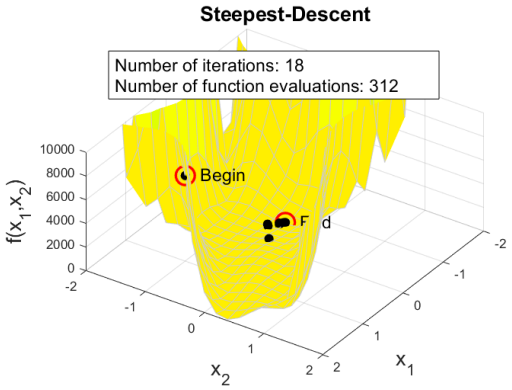

Figure 5: History of iterations in a 3D graph

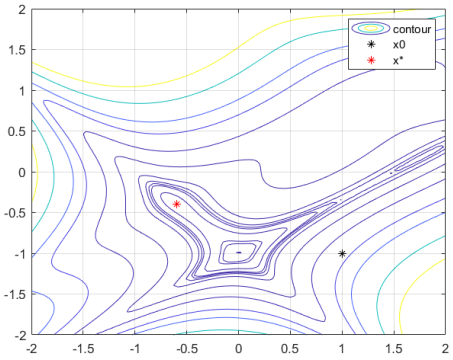

Figure 6: Initial and final solution positions

Overall, the quasi-newton methods proved to be less expensive for the BFGS, or more accurate for the DFP method, than the steepest descent method. To conclude, after these results it is possible to claim assert that the Hessian does not need to be precisely computed.

2.1.4 Providing gradient

To inspect the consequences of providing the gradient, the gradient was fed input of *fminunc*. By providing the exact derivative, the *fminunc* stops calculating the gradient using a finite difference method. So,

it is expected to reduce the time spent as well as function evaluations. This aspect is confirmed in table 6 for the *Steepest Descent* method.

Gradient	$\boldsymbol{x0}$	(x_1, x_2)	$f(x_1, x_2)$	Iter.	F. Counts	exitFlag
No	$(-1, 1)$	$(0.599, 0.401)$	30.000	7	195	0
Yes	$(-1, 1)$	$(-0.600, -0.400)$	30.000	14	100	1
No	$(0, 0)$	$(0.567, 0.435)$	30.460	8	183	0
Yes	$(-1, 1)$	$(-0.600, -0.400)$	30.000	24	147	1
No	$(1, 0)$	$(1.4 \times 10^{-6}, -1.0)$	3.000	11	183	1
Yes	$(1, 0)$	$(-1.845 \times 10^{-6}, -1.0)$	3.000	11	61	1

Table 6: Steepest Descent results

At first look, it is recognizable the effect of the input of the gradient function. For the cases where the algorithm stopped because the maximum number of iterations stopped (due to the maximum number of function evaluations being reached), the algorithm is now able to stop with an *exitFlag* value of 1. The fact that the number of interactions is smaller for the first two cases is due to the fact that the algorithm stopped before reaching a solution. One can instantly realise that, when the gradient is provided, the number of function evaluations decreases vastly, diminishing the computational power and time. To conclude, if possible, the user should always seed *fminunc* the exact gradient function. In most of the real optimization problems, it is not possible, of course, since these problems involve functions that a lot of times are not continuous, or that the user has no idea of the form it takes.

3 Constrained Minimization

3.1 Feasible region

The constrain region is easily identified as circle centered in $(0, 0)$ with a radius of 3. All the contour lines have the same equation $x_2 = x_1 + b$ and the minimum will be located in the fourth quadrant, at an angle of $\frac{7}{4}\pi$. After some algebra, the location of this point is identified, $(x_1, x_2) = (2.12, 2.12)$ and $f(x_1, x_2) = -4.2426$.

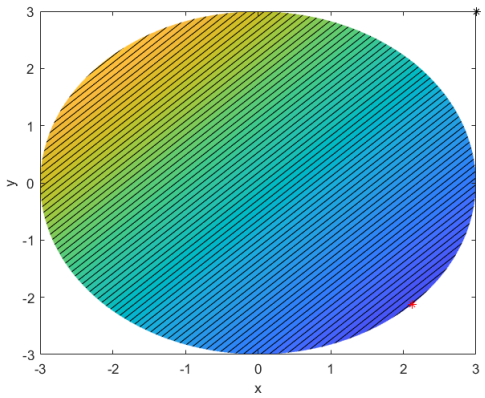

Figure 7: Feasible region

3.2 Constrained minimization method

The MATLAB function to be used is *fmincon* with several different algorithms. There are no changes in its parameters apart from the change on the algorithm itself so its default parameters are constant and equal to:

Algorithm	'interior-point'
MaxIterations	100
MaxFunctionsEvaluations	3000
OptimalityTolerance	1e-6
FunctionTolerance	1e-6

Table 7: Default values for *fmincon*

3.2.1 Interior Point

Analysing the results obtained for the Interior Point method one can conclude that the minimum obtained is the same from previous section. Concerning the number of iterations, as opposed to expected, the less number of iterations required happened for the case where the initial solution was the furthest. This means that the initial solution proximity to the actual solution doesn't affect the required computational effort, at least for this example. This fact might be related to the *Stepsize* verified, since it has the higher value for the first iteration. The *flagValue* was omitted in the table due to lack of space but it was 1 for each initial solution, meaning that *fmincon* converged to a local minimum.

3.2.2 SQP

The first thing one can notice is that the number of iterations for the SQP is less than for the Interior Point method. Even though the difference is small in this optimization problem, bare in mind that this is a simple problem that doens't require more than 10 iterations. This small difference might become a big advantage in more massive problems. Once again, the fastest solution occurs for the furthest initial guess, having again an initial *Stepsize* three times higher than for the other initial solutions.

3.2.3 Active-Set

In the Active-Set method, for the first time, the furthest solution is not the one that converges faster. However, the closest one is the one that takes the maximum number of iterations. This is due to the Hessian matrix being changed twice and the infeasible start point. Furthermore, as expected, the *Max-constrain* parameter decreases as iteration number advances, reaching a value close to zero at the end. The number of iterations is similar to the SQP method.

3.2.4 Providing Gradient

In a similar way to what was done in the previous section, the specific gradient function was seeded to the *fmincon* function in order to study the effect on the computational cost and time. The results are not displayed due to the similarity in the results. One can assume that due to the simplicity of the function evaluated in comparison with the Goldstein-Price function. Thus, it is redundant if the user seeds the gradient since it does not make much of a difference for simple problems.

3.2.5 Results obtained for each method

$x0$	Iter	FEval	$f(x)$	Stepsize
(0,0)	0	3	0	
	1	7	-1	0.707
	2	11	-2.4695	1.039
	3	14	-4.1516	1.189
	4	17	-4.1834	0.022
	5	20	-4.2224	0.027
	6	23	-4.2424	0.014
	7	26	-4.2426	1.73e-4
	8	29	-4.2426	1.402e-6
(3,3)	0	3	0	
	1	6	-2	1.838
	2	10	-4.9624	3.456
	3	13	-4.2385	0.537
	4	16	-4.1497	0.128
	5	19	-4.2219	0.063
	6	22	-4.2425	0.017
	7	25	-4.2426	7.88e-5
	8	28	-4.2426	1.73e-6
(9,-9)	0	3	-18	
	1	6	-9.4552	6.042
	2	9	-5.6234	2.709
	3	12	-4.3349	0.911
	4	15	-4.2266	0.076
	5	18	-4.2424	0.011
	6	21	-4.2426	1.55e-4
	7	24	-4.2426	1.403e-6

Table 8: Interior Point

$x0$	Iter	FEval	$f(x)$	Stepsize
(0,0)	0	3	0	
	1	6	-2	1.414
	2	10	-2.4372	0.3094
	3	13	-4.5293	1.479
	4	16	-4.2511	0.1961
	5	19	-4.2426	0.0064
	6	22	-4.2426	6.81e-6
(3,3)	0	3	0	
	1	6	-2	1.768
	2	12	-5	2.951
	3	15	-4.3492	0.8195
	4	18	-4.2792	0.4
	5	21	-4.2438	0.0711
	6	24	-4.2426	1.3e-3
	7	27	-4.2426	6.92e-5
(9,-9)	0	3	-18	
	1	6	-9.5	6
	2	9	-5.6974	2.689
	3	12	-4.4283	0.897
	4	15	-4.2465	0.128
	5	18	-4.2426	2.7e-3
	6	21	-4.2426	1.26e-6

Table 9: SPQ iteration

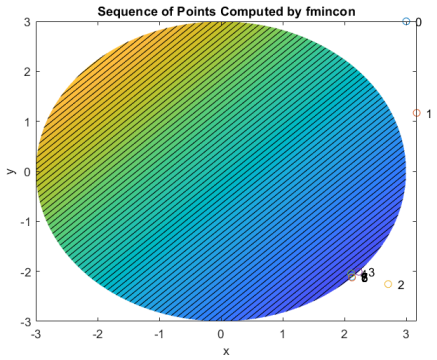

Figure 8: Interior Point solution history

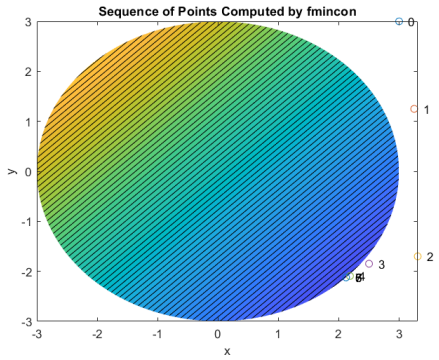

Figure 9: SQP solution history

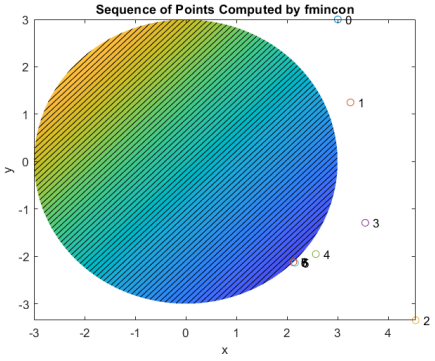

Figure 10: Active-Set solution history

$x0$	Iter	FEval	$f(x)$	Maxconstraint	Procedure
(0,0)	0	3	0	-9	
	1	6	-2	-7	
	2	10	-3.75	-1.969	Hessian modified twice
	3	13	-4.275		0.1378
	4	16	-4.2428	5.2e-4	
	5	19	-4.2426	7.5e-9	Hessian modified
(3,3)	0	3	0	9	Infeasible start point
	1	6	-2	3.125	
	2	9	-7.8823	22.78	
	3	12	-4.8327	5.212	
	4	15	-4.5158	1.387	
	5	18	-4.29	0.2023	
	6	21	-4.2429	0.00127	
	7	24	-4.2426	1.04e-5	Hessian modified
	8	27	-4.2426	2.6e-7	Hessian modified
(9,-9)	0	3	-18	153	Infeasible start point
	1	6	-9.5	36.13	
	2	9	-5.6974	7.23	
	3	12	-4.4284	0.8052	
	4	15	-4.2465	0.01653	
	5	18	-4.2426	7.58e-6	
	6	21	-4.2426	1.64e-13	Hessian modified

Table 10: Active set

4 Sensitivity Analysis: Finite-Differences

4.1 Finite-differences Taylor Series formulas deduction

The idea of finite difference methods is quite simple, since it corresponds to an estimation of a derivative by the ratio of two or more differences according to the theoretical definition of the derivative and the Taylor series expansion:

$$\begin{cases} f_{i-2} = f(x-2\Delta x) = f(x) - f^{(1)}2\Delta x + 2f^{(2)}(x)\Delta x^2 - \frac{4}{3}f^{(3)}(x)\Delta x^3 + \frac{2}{3}f^{(4)}\Delta x^4 + \cdots \\ f_{i-1} = f(x-\Delta x) = f(x) - f^{(1)}\Delta x + \frac{1}{2}f^{(2)}(x)\Delta x^2 - \frac{1}{6}f^{(3)}(x)\Delta x^3 + \frac{1}{24}f^{(4)}\Delta x^4 + \cdots \\ f_i = f(x) \\ f_{i+1} = f(x-\Delta x) = f(x) + f^{(1)}\Delta x + \frac{1}{2}f^{(2)}(x)\Delta x^2 + \frac{1}{6}f^{(3)}(x)\Delta x^3 + \frac{1}{24}f^{(4)}\Delta x^4 + \cdots \\ f_{i-2} = f(x-2\Delta x) = f(x) + f^{(1)}2\Delta x + 2f^{(2)}(x)\Delta x^2 + \frac{4}{3}f^{(3)}(x)\Delta x^3 + \frac{2}{3}f^{(4)}\Delta x^4 + \cdots \end{cases}$$

To obtain a second order central differences approximation the following steps are required to obtain an expression.

$$0^{''}(x) + f^{'}(x) + 0(x) = a_1 f_{i-1} + a_2 f_i + a_3 f_{i+1} \Leftrightarrow$$

$$\Leftrightarrow f^{'} = a_1[f(x) - f^{'}\Delta x + \frac{1}{2}f^{''}(x)\Delta x^2 + \epsilon(\Delta x^3)] + a_2 f(x) + a_3[f(x) + f^{'}\Delta x + \frac{1}{2}f^{''}(x)\Delta x^2 + \epsilon(\Delta x^3)]$$

$$\begin{cases} 0 = a_1 + a_2 + a_3 \\ 1 = -a_1\Delta x + a_3\Delta x \\ 0 = a_1\frac{1}{2}\Delta x^2 + a_3\frac{1}{2}\Delta x^2 \end{cases} \Leftrightarrow \begin{cases} a_1 = -\frac{1}{2\Delta x} \\ a_2 = 0 \\ a_3 = \frac{1}{2\Delta x} \end{cases}$$

The expression obtained has an associated error proportional to h^2 as expected:

$$f^{(1)}(x) = \frac{f_{i+1} - f_{i-1}}{2\Delta x} + O(h^2)$$

For the first order forward differences the method is the same as the one present above.

$$f^1(x) = \frac{f_i - f_{i-1}}{\Delta x} + O(h)$$

As expected, the error associated to the truncation of terms in the Taylor series expression is proportional to h.

For the fourth order central differences expression the steps follow the basis of last two examples, so they will be omitted. The expression for the fourth order central differences is:

$$f^{(1)} = \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12\Delta x}$$

4.2 Calculation of the analytical derivative

In the next subsection, the function to be evaluated is:

$$f(x) = \frac{e^x}{\sqrt{sin^3(x) + cos^3(x)}} \tag{1}$$

Solving analytically and confirming using MATLAB, the expression obtained is:

$$\frac{df}{dx} = \frac{e^x}{\sqrt{sen(x)^3 + cos(x)^3}} - \frac{e^x(3cos(x)sen(x)^2 - 3cos(x)^2 sen(x)}{2((sen(x)^3 + cos(x)^3)^{\frac{3}{2}}} \tag{2}$$

The value for $\frac{df}{dx}|_{x=1.5} = 4.0534$ is easily computed.

4.3 Relative error analyses

The following graph shows the evolution of the relative error with the step size (h) for the first order forward difference, second and fourth order central differences.

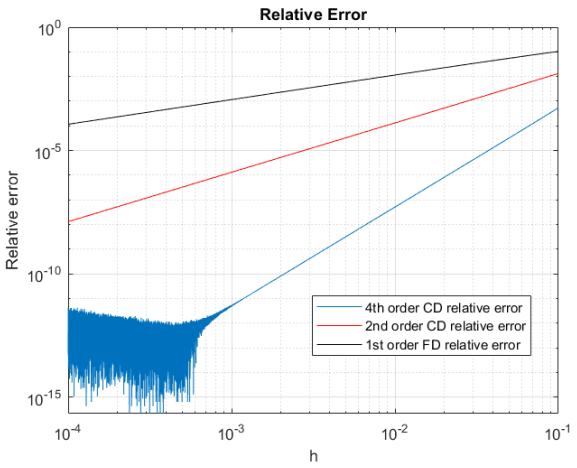

Figure 11: Evolution of the relative error in a *loglog* graph

Each relative error line has a different slope that corresponds to the order of the approximation. Thus, the fourth order displays the highest slope and, consequentially, the lowest error. It is noticeable that the fourth order shows a strange behaviour as soon as h gets close to $log(10^-3)$. This behaviour makes the error increase due to the subtractive cancellation error.

5 Sensitivity Analysis: Automatic Differentiation

This section has the main goal of obtaining the expressions of the derivatives of a function using Leibniz's chain rule. Although this approach is as accurate as an analytic method, it is potentially much easier to implement since this can be done automatically without having to obtain manually arduous differentiation.

5.1 Implementation

With the aid of TAPENADE online differentiation tool it is possible to obtain these derivatives in both modes for a function to be fed as input in C language. The input represents two functions given by:

$$\begin{cases} f_1(x_1, x_2) = 4x_1x_2 + x_2^3 \\ f_2(x_1, x_2) = cos(x_1) \end{cases}$$

The results obtained were:

```
/*        Generated by TAPENADE     (INRIA, Ecuador team)
    Tapenade 3.14 (master-db54337a6) - 29 Jul 2019 10:54
*/
/*
  Differentiation of myFunction in forward (tangent) mode:
   variations   of useful results: *f1 *f2
   with respect to varying inputs: *f1 *f2 x1 x2
   RW status of diff variables: *f1:in-out *f2:in-out x1:in x2:in
   Plus diff mem management of: f1:in f2:in
*/
void myFunction_d(double x1, double x1d, double x2, double x2d, double *f1,
        double *f1d, double *f2, double *f2d) {
    /* First function */
    *f1d = 4*(x1d*x2) + 4*(x1*x2d) + x2d*3*pow(x2, 3-1);
    *f1 = 4*x1*x2 + pow(x2, 3);
    /* Second function */
    *f2d = -(x1d*sin(x1));
    *f2 = cos(x1);
}
```

Figure 12: Differentiated code (forward mode)

```
/*        Generated by TAPENADE     (INRIA, Ecuador team)
    Tapenade 3.14 (master-db54337a6) - 29 Jul 2019 10:54
*/
#include <adBuffer.h>

/*
  Differentiation of myFunction in reverse (adjoint) mode:
   gradient     of useful results: *f1 *f2 x1 x2
   with respect to varying inputs: *f1 *f2 x1 x2
   RW status of diff variables: *f1:in-out *f2:in-out x1:incr
                x2:incr
   Plus diff mem management of: f1:in f2:in
*/
void myFunction_b(double x1, double *x1b, double x2, double *x2b, double *f1,
        double *f1b, double *f2, double *f2b) {
    *x1b = *x1b - sin(x1)*(*f2b);
    *f2b = 0.0;
    *x1b = *x1b + 4*x2*(*f1b);
    *x2b = *x2b + (3*pow(x2, 3-1)+4*x1)*(*f1b);
    *f1b = 0.0;
}
```

Figure 13: Differentiated code (adjoint mode)

5.2 Forward

Observing the output generated by TAPENADA for the forward method, one can simply identify $\frac{\partial f_2}{\partial \partial x_1}$ as $*f2d$. Thus, the input values will be $[x1d, x2d] = [1, 0]$ due to the simple cosine function input.

5.3 Backward

For the backward method, the inputs are $[f1b, f2b] = [1, 0]$.