Numerical Derivatives in Scilab

Michaël Baudin

May 2009

Abstract

This document present the use of numerical derivatives in Scilab. In the first part, we present a result which is surprising when we are not familiar with floating point numbers. In the second part, we analyse the method to use the optimal step to compute derivatives with finite differences on floating point systems. We present several formulas and their associated optimal steps. In the third part, we present the derivative function, its features and its performances.

Contents

1 Introduction 4
  1.1 Introduction .................................................. 4
  1.2 Overview ..................................................... 4

2 A surprising result 4
  2.1 Theory .......................................................... 4
    2.1.1 Taylor’s formula for univariate functions ............... 5
    2.1.2 Finite differences ......................................... 5
  2.2 Experiments .................................................... 6

3 Analysis 8
  3.1 Errors in function evaluations ................................. 8
  3.2 Various results for sin(2^{64}) ................................ 9
  3.3 Floating point implementation of the forward formula ...... 10
  3.4 Numerical experiments with the robust forward formula ..... 15
  3.5 Backward formula ............................................... 16
  3.6 Centered formula with 2 points ................................ 16
  3.7 Centered formula with 4 points ................................ 19
  3.8 Some finite difference formulas for the first derivative ..... 21
  3.9 A three points formula for the second derivative .......... 22
  3.10 Accuracy of finite difference formulas ....................... 24
  3.11 A collection of finite difference formulas .................. 26


<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Finite differences of multivariate functions</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Multivariate functions</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Numerical derivatives of multivariate functions</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Derivatives of a multivariate function in Scilab</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Derivatives of a vectorial function with Scilab</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Computing higher degree derivatives</td>
<td>35</td>
</tr>
<tr>
<td>4.6</td>
<td>Nested derivatives with Scilab</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>Computing derivatives with more accuracy</td>
<td>39</td>
</tr>
<tr>
<td>4.8</td>
<td>Taking into account bounds on parameters</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td>The derivative function</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Varying order to check accuracy</td>
<td>42</td>
</tr>
<tr>
<td>5.3</td>
<td>Orthogonal matrix</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance of finite differences</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>One more step</td>
<td>47</td>
</tr>
<tr>
<td>7</td>
<td>Automatically computing the coefficients</td>
<td>49</td>
</tr>
<tr>
<td>7.1</td>
<td>The coefficients of finite difference formulas</td>
<td>49</td>
</tr>
<tr>
<td>7.2</td>
<td>Automatically computing the coefficients</td>
<td>51</td>
</tr>
<tr>
<td>7.3</td>
<td>Computing the coefficients in Scilab</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>Notes and references</td>
<td>54</td>
</tr>
<tr>
<td>9</td>
<td>Exercises</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>Acknowledgments</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>58</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Introduction

This document is an open-source project. The \LaTeX{} sources are available on the Scilab Forge:

\url{http://forge.scilab.org/index.php/p/docnumber/}

The \LaTeX{} sources are provided under the terms of the Creative Commons Attribution ShareAlike 3.0 Unported License:

\url{http://creativecommons.org/licenses/by-sa/3.0}

The Scilab scripts are provided on the Forge, inside the project, under the \texttt{scripts} sub-directory. The scripts are available under the CeCiLL licence:

\url{http://www.cecill.info/licences/Licence_CeCILL_V2-en.txt}

1.2 Overview

In this document, we analyse the computation of the numerical derivative of a given function. Before getting into the details, we briefly motivate the need for approximate numerical derivatives.

Consider the situation where we want to solve an optimization problem with a method which requires the gradient of the cost function. In simple cases, we can provide the exact gradient. The practical computation may be performed "by hand" with paper and pencil. If the function is more complicated, we can perform the computation with a symbolic computing system (such as Maple or Mathematica). If some situations, this is not possible. In most practical situations, indeed, the formula involved in the computation is extremely complicated. In this case, numerical derivatives can provide an accurate evaluation of the gradient. Other methods to compute the gradient are based on adjoint equations and on automatic differentiation. In this document, we focus on numerical derivatives methods because Scilab provide commands for this purpose.

2 A surprising result

In this section, we present surprising results which occur when we consider a function of one variable only. We derive the forward numerical derivative based on the Taylor expansion of a function with one variable. Then we present a numerical experiment based on this formula, with decreasing step sizes.

This section was first published in [3].

2.1 Theory

Finite differences methods approximate the derivative of a given function \( f \) based on function values only. In this section, we present the forward derivative, which allows
to compute an approximation of $f'(x)$, based on the value of $f$ at well chosen points. The computations are based on a local Taylor’s expansion of $f$ in the neighbourhood of the point $x$. This assumes that $f$ is continuously derivable, an assumption which is used throughout this document.

2.1.1 Taylor’s formula for univariate functions

Taylor’s theorem is of fundamental importance because it shows that the local behaviour of the function $f$ can be known from the function and its derivatives at a single point.

**Theorem 2.1.** Assume that $f : \mathbb{R} \to \mathbb{R}$ is a continuously derivable function of one variable. Assume that $f$ is continuously differentiable $d$ times, i.e. $f \in C^d$, where $d$ is a positive integer. There exists a scalar $\theta \in [0,1]$, such that

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \ldots + \frac{1}{(d-1)!}h^{d-1}f^{(d-1)}(x) + \frac{1}{d!}h^df^{(d)}(x + \theta h),$$

(1)

This theorem will not be proved here [10].

We can expand Taylor’s formula up to order 4 derivatives of $f$ and get

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f''''(x) + O(h^5)$$

This formula can be used to derive finite differences formulas, which approximate the derivatives of $f$ using function values only.

2.1.2 Finite differences

In this section, we derive the forward 2 points finite difference formula and prove that it is an order 1 formula for the first derivative of the function $f$.

**Proposition 2.2.** Let $f : \mathbb{R} \to \mathbb{R}$ be a continuously derivable function of one variable. Therefore,

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(x) + O(h^2).$$

(4)

**Proof.** Assume that $f : \mathbb{R} \to \mathbb{R}$ is a function with continuous derivatives. If we neglect higher order terms, we have

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \mathcal{O}(h^3).$$

(5)

Therefore,

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{h}{2}f''(x) + \mathcal{O}(h^2),$$

which concludes the proof. □
Definition 2.3. (Forward finite difference for $f'$) The finite difference formula

$$Df(x) = \frac{f(x+h) - f(x)}{h}$$

is the forward 2 points finite difference for $f'$.

The following definition defines the order of a finite difference formula, which measures the accuracy of the formula.

Definition 2.4. (Order) A finite difference formula $Df$ is of order $p > 0$ for $f^{(d)}$ if

$$Df(x) = f^{(d)}(x) + O(h^p).$$

The equation 4 indicates that the forward 2 points finite difference is an order 1 formula for $f'$.

Definition 2.5. (Truncation error) The truncation error of a finite difference formula for $f^{(d)}(x)$ is

$$E_t(h) = \left|Df(x) - f^{(d)}(x)\right|$$

The equation 4 indicates that the truncation error of the 2 points forward formula is:

$$E_t(h) = \frac{h}{2}|f''(x)|,$$

The truncation error of the equation 10 depends on step $h$ so that decreasing the step reduces the truncation error. The previous discussion implies that a (naive) algorithm to compute the numerical derivative of a function of one variable is

$$f'(x) \leftarrow (f(x+h) - f(x))/h$$

As we are going to see, the previous algorithm is much more naive that it appears, as it may lead to very inaccurate numerical results.

2.2 Experiments

In this section, we present numerical experiments based on a naive implementation of the forward finite difference formula. We show that a wrong step size $h$ may lead to very inaccurate results.

The following Scilab function is a straightforward implementation of the forward finite difference formula.

```scilab
function fp = myfprime(f,x,h)
fp = (f(x+h) - f(x))/h;
endfunction
```

In the following numerical experiments, we consider the square function $f(x) = x^2$, which derivative is $f'(x) = 2x$. The following Scilab script implements the square function.
function y = myfunction (x)
    y = x*x;
endfunction

The naive idea is that the computed relative error is small when the step \( h \) is small. Because \textit{small} is not a priori clear, we take \( \epsilon_M \approx 10^{-16} \) in double precision as a good candidate for \textit{small}.

In order to compare our results, we use the \texttt{derivative} function provided by Scilab. The most simple calling sequence of this function is

\[
J = \text{derivative} \left( F, x \right)
\]

where \( F \) is a given function, \( x \) is the point where to compute the derivative and \( J \) is the Jacobian, i.e. the first derivative when the variable \( x \) is a simple scalar. The \texttt{derivative} function provides several methods to compute the derivative. In order to compare our method with the method used by \texttt{derivative}, we must specify the \texttt{order} of the method. The calling sequence is then

\[
J = \text{derivative} \left( F, x, \text{order} = o \right)
\]

where \( o \) can be equal to 1, 2 or 4. Our forward formula corresponds to order 1.

In the following script, we compare the computed relative error produced by our naive method with step \( h = \epsilon_M \) and the \texttt{derivative} function with default step and the order 1 method.

```scilab
x = 1.0;
fpref = derivative(myfunction,x);
fexact = 2.0;
e = abs(fpref-fexact)/fexact;
mprintf("Scilab f'=%e , error=%e\n", fpref,e);
h = 1.0e-16;
fp = myfprime(myfunction,x,h);
e = abs(fp-fexact)/fexact;
mprintf("Naive f'=%e , error=%e\n", fp,e);
```

When executed, the previous script prints out:

Scilab f' =2.000000e+000 , error =7.450581e-009
Naive f' =0.000000e+000 , error =1.000000e+000

Our naive method seems to be quite inaccurate and has not even 1 significant digit ! The Scilab primitive, instead, has approximately 9 significant digits.

Since our faith is based on the truth of the mathematical theory, which leads to accurate results in many situations, we choose to perform additional experiments...

Consider the following experiment. In the following Scilab script, we take an initial step \( h = 1.0 \) and then divide \( h \) by 10 at each step of a loop made of 20 iterations.

```scilab
x = 1.0;
fexact = 2.0;
fpref = derivative(myfunction,x,order=1);
e = abs(fpref-fexact)/fexact;
mprintf("Scilab f'=%e , error=%e\n", fpref,e);
h = 1.0;
for i=1:20
    h=h/10.0;
    fp = myfprime(myfunction,x,h);
```
\[ e = \frac{\text{abs}(fp - fpexact)}{fpexact}; \]

\[ \text{mprintf("Naive } f' =%e, h=%e, error=%e
\n", fp, h, e); } \]

end

Scilab then produces the following output.

\[
\text{Scilab } f' =2.000000e+000, \text{ error}=7.450581e-009 \\
\text{Naive } f' =2.100000e+000, h=1.000000e-001, \text{ error}=5.000000e-002 \\
\text{Naive } f' =2.010000e+000, h=1.000000e-002, \text{ error}=5.000000e-003 \\
\text{Naive } f' =2.001000e+000, h=1.000000e-003, \text{ error}=5.000000e-004 \\
\text{Naive } f' =2.000100e+000, h=1.000000e-004, \text{ error}=5.000000e-005 \\
\text{Naive } f' =2.000010e+000, h=1.000000e-005, \text{ error}=5.000007e-006 \\
\text{Naive } f' =2.000001e+000, h=1.000000e-006, \text{ error}=5.000000e-007 \\
\text{Naive } f' =2.000000e+000, h=1.000000e-007, \text{ error}=5.054390e-008 \\
\text{Naive } f' =2.000000e+000, h=1.000000e-008, \text{ error}=6.077471e-009 \\
\text{Naive } f' =2.000000e+000, h=1.000000e-009, \text{ error}=8.274037e-008 \\
\text{Naive } f' =2.000000e+000, h=1.000000e-010, \text{ error}=8.274037e-008 \\
\text{Naive } f' =2.000000e+000, h=1.000000e-011, \text{ error}=8.274037e-008 \\
\text{Naive } f' =2.000000e+000, h=1.000000e-012, \text{ error}=8.890058e-005 \\
\text{Naive } f' =1.998401e+000, h=1.000000e-013, \text{ error}=7.992778e-004 \\
\text{Naive } f' =1.998401e+000, h=1.000000e-014, \text{ error}=7.992778e-004 \\
\text{Naive } f' =2.220446e+000, h=1.000000e-015, \text{ error}=1.102230e-001 \\
\text{Naive } f' =0.000000e+000, h=1.000000e-016, \text{ error}=1.000000e+000 \\
\text{Naive } f' =0.000000e+000, h=1.000000e-017, \text{ error}=1.000000e+000 \\
\text{Naive } f' =0.000000e+000, h=1.000000e-018, \text{ error}=1.000000e+000 \\
\text{Naive } f' =0.000000e+000, h=1.000000e-019, \text{ error}=1.000000e+000 \\
\text{Naive } f' =0.000000e+000, h=1.000000e-020, \text{ error}=1.000000e+000 \\
\]

We see that the relative error decreases, then increases. Obviously, the optimum step is approximately \( h = 10^{-8} \), where the relative error is approximately \( e_r = 6.10^{-9} \). We should not be surprised to see that Scilab has computed a derivative which is near the optimum.

## 3 Analysis

In this section, we analyse the floating point implementation of a numerical derivative. In the first part, we take into account rounding errors in the computation of the total error of the numerical derivative. Then we derive several numerical derivative formulas and compute their optimal step and optimal error. We finally present the method which is used in the \texttt{derivative} function.

### 3.1 Errors in function evaluations

In this section, we analyze the error that we get when we evaluate a function on a floating point system such as Scilab.

Assume that \( f \) is a continuously differentiable real function of one real variable \( x \). When Scilab evaluates the function \( f \) at the point \( x \), it makes an error and computes \( \tilde{f}(x) \) instead of \( f(x) \). Let us define the relative error as

\[
e(x) = \left| \frac{\tilde{f}(x) - f(x)}{f(x)} \right|,
\]

(11)
if \( f(x) \) is different from zero. The previous definition implies:

\[
\tilde{f}(x) = (1 + \delta(x))f(x),
\]

(12)

where \( \delta(x) \in \mathbb{R} \) is such that \( |\delta(x)| = e(x) \). We assume that the relative error is satisfying the inequality

\[
e(x) \leq c(x)e_M,
\]

(13)

where \( e_M \) is the machine precision and \( c \) is a function depending on \( f \) and the point \( x \).

In Scilab, the machine precision is \( e_M \approx 10^{-16} \) since Scilab uses double precision floating point numbers. See [4] for more details on floating point numbers in Scilab.

The base ten logarithm of \( c \) approximately measures the number of significant digits which are lost in the computation. For example, assume that, for some \( x \in \mathbb{R} \), we have \( e_M \approx 10^{-16} \) and \( c(x) = 10^5 \). Then the relative error in the function value is lower than \( c(x)e_M = 10^{-16} + 5 = 10^{-11} \). Hence, five digits have been lost in the computation.

The function \( c \) depends on the accuracy of the function, and can be zero, small or large.

- At best, the compute function value is exactly equal to the mathematical value. For example, the function \( f(x) = (x - 1)^2 + 1 \) is exactly evaluated as \( f(x) = 1 \) when \( x = 1 \). In other words, we may have \( c(x) = 0 \).

- In general, the mathematical function value is between two consecutive floating point numbers. In this case, the relative error is bounded by the unit roundoff \( u = \frac{e_M}{2} \). For example, the operators +, -, *, / and \texttt{sqrt} are guaranteed to have a relative error no greater than \( u \) by the IEEE 754 standard [21]. In other words, we may have \( c(x) = \frac{1}{2} \).

- At worst, there is no significant digit in \( \tilde{f}(x) \). This may happen for example when some intermediate algorithm used within the function evaluation (e.g. the range reduction algorithm) cannot get a small relative error. An example of such a situation is given in the next section. In other words, we may have \( c(x) \approx 10^{16} \).

3.2 Various results for \( \sin(2^{64}) \)

In this section, we compute the result of the computation \( \sin(2^{64}) \) on various computation softwares on several operating systems. This particular computation is inspired by the work of Soni and Edelman[18] where the authors performed various comparisons of numerical computations across different softwares. Here, the particular input \( x = 2^{64} \) has been chosen because this number can be exactly represented as a floating point number.

In order to get all the available precision, we often have to configure the display, so that all digits are printed. For example, in Scilab, we must use the \texttt{format} function, as in the following session.
<table>
<thead>
<tr>
<th>Software</th>
<th>Operating System</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wolfram Alpha</td>
<td>Web Service</td>
<td>0.0235985...</td>
</tr>
<tr>
<td>Octave 3.2.4</td>
<td>Win. XP 32 bits</td>
<td>0.247260646...</td>
</tr>
<tr>
<td>Matlab 7.7.0 R 2008</td>
<td>Win. XP 32 bits</td>
<td>0.0235985...</td>
</tr>
<tr>
<td>Scilab 5.2.2</td>
<td>Win. XP 32 bits</td>
<td>0.2472606463...</td>
</tr>
<tr>
<td>Scilab 5.2.2</td>
<td>Linux 32 bits glibc 2.10.1</td>
<td>-0.35464734997...</td>
</tr>
<tr>
<td>Scilab 5.2.2</td>
<td>Linux 64 bits eglibc 2.11.2-1</td>
<td>0.0235985...</td>
</tr>
</tbody>
</table>

Figure 1: A family of results for \( \sin(2^{64}) \).

```plaintext
--> format("e",25)
--> sin(2^64)
ans =
 2.472606463094176865D-01
```

In Matlab, for example, we use the `format long` statement.

We used Wolfram Alpha [15] in order to compute the exact result for this computation. The results are presented in figure 1.

This table can be compared with the Table 20, p. 28 in [18].

One of the reasons behind these discrepancies may be the cumulated errors in the range reduction algorithm. Anyway, this value of \( x \) is so large that a small change in \( x \) induces a large number of cycles in the trigonometric circle: this is not a "safe" zone of computation for sine.

It can be proved that the condition number of the sine function is

\[
\begin{vmatrix}
\cos(x) \\
\sin(x)
\end{vmatrix}.
\]

Therefore, the sine function has a large condition number

- if \( x \) is large,
- if \( x \) is an integer multiple of \( \pi \) (where \( \sin(x) = 0 \)).

The example presented in this section is rather extreme. For most elementary function and for most inputs \( x \), the number of significant binary digits is in the range [50, 52]. But there are many situations where this accuracy is not achieved.

### 3.3 Floating point implementation of the forward formula

In this section, we derive the floating point implementation of the forward formula given by

\[
Df(x) = \frac{f(x + h) - f(x)}{h}.
\]  \( (14) \)

In other words, given \( x \) and \( f \), we search the step \( h > 0 \) so that the error in the numerical derivative is minimum.

In the IEEE 754 standard[21, 9], double precision floating point numbers are stored as 64 bits floating point numbers. More precisely, these numbers are stored
with 52 bits in the mantissa, 1 sign bit and 11 bits in the exponent. In Scilab, which uses double precision numbers, the machine precision is stored in the global variable %eps, which is equal to $\epsilon_M = \frac{1}{2^{52}} = 2.220.10^{-16}$. This means that, any value $x$ has 52 significants binary digits, corresponds to approximately 16 decimal digits. If IEEE 754 single precision floating point numbers were used (i.e. 32 bits floating point numbers with 23 bits in the mantissa), the precision to use would be $\epsilon_M = \frac{1}{2^{23}} \approx 10^{-7}$.

We can, as Dumontet and Vignes[6], consider the forward difference formula very closely. Indeed, there are many sources of errors which can be considered:

- the point $x$ is represented in the machine by $\tilde{x}$,
- the step $h$ is represented in the machine by $\tilde{h}$,
- the point $\tilde{x} + \tilde{h}$ is computed in the machine as $\tilde{x} \oplus \tilde{h}$, where the $\oplus$ operation is the addition,
- the function value of $f$ at point $x$ is computed by the machine as $\tilde{f}(\tilde{x})$,
- the function value of $f$ at point $x + h$ is computed by the machine as $\tilde{f}(\tilde{x} \oplus \tilde{h})$,
- the difference $f(x + h) - f(x)$ is computed by the machine as $\tilde{f}(\tilde{x} \oplus \tilde{h}) \ominus \tilde{f}(\tilde{x})$, where the $\ominus$ operation is the subtraction,
- the factor $(f(x + h) - f(x))/h$ is computed by the machine as $(\tilde{f}(\tilde{x} + \tilde{h}) \ominus \tilde{f}(\tilde{x})) \oslash \tilde{h}$, where the $\oslash$ operation is the division.

All in all, the forward difference formula

$$Df(x) = \frac{f(x + h) - f(x)}{h}$$

is computed by the machine as

$$\tilde{D}f(x) = (\tilde{f}(\tilde{x} \oplus \tilde{h}) \ominus \tilde{f}(\tilde{x})) \oslash \tilde{h}.$$ (16)

For example, consider the error which is associated with the sum $\tilde{x} \oplus \tilde{h}$. If the step $\tilde{h}$ is too small, the sum $\tilde{x} \oplus \tilde{h}$ is equal to $\tilde{x}$. On the other side, if the step $\tilde{h}$ is too large then the sum $\tilde{x} \oplus \tilde{h}$ is equal to $\tilde{h}$. We may require that the step $\tilde{h}$ is in the interval $[2^{-52}\tilde{x}, 2^{52}\tilde{x}]$ so that $\tilde{x}$ are not too far away from each other in magnitude. We will discuss this assumption later in this chapter.

Dumontet and Vignes show that the most important source of error in the computation is the function evaluation. That is, the addition $\oplus$, subtraction $\ominus$ and division $\oslash$ operations and the finite accuracy of $\tilde{x}$ and $\tilde{h}$, produce most of the time a much lower relative error than the error generated by the function evaluation.

With a floating point computer, the total error that we get from the forward difference approximation 14 is (skipping the multiplication constants) the sum of two terms :

- the truncation error caused by the term $\frac{h}{2}f''(x)$,
• and the rounding error $\epsilon_M|f(x)|$ on the function values $f(x)$ and $f(x + h)$.

Therefore, the error associated with the forward finite difference is

$$E(h) = \frac{\epsilon_M|f(x)|}{h} + \frac{h}{2}|f''(x)|$$

(17)

The total error is then the balance between the positive functions $\frac{\epsilon_M|f(x)|}{h}$ and $\frac{h}{2}|f''(x)|$.

• When $h \to \infty$, the error is dominated by the truncation error $\frac{h}{2}|f''(x)|$.

• When $h \to 0$, the error is dominated by the rounding error $\frac{\epsilon_M|f(x)|}{h}$.

The following Scilab script allows to create the plot of the function $E(h)$ which is presented in figure 2. The graph is plot in logarithmic scale for the function $f(x) = x^2$. When this function is considered at point $x = 1$, we have $f(x) = 1$ and $f''(x) = 2$.

```scilab
function e = totalerror ( h )
    f = 1
    fpp = 2
    e = %eps * f / h + h * fpp / 2.0
endfunction

n = 1000;
x = linspace (-16,0,n);
y = zeros (n,1);
for i = 1:n
    h = 10^x(i));
    y(i) = log10 (totalerror ( h ));
end
plot ( x , y )
```

**Proposition 3.1.** Let $f : \mathbb{R} \to \mathbb{R}$ be a continuously derivable function of one variable. Consider the forward finite difference of $f$ defined by (7). Assume that the associated error implied by truncation and rounding is defined by

$$E(h) = \frac{\epsilon_M|f(x)|}{h} + \frac{h}{2}|f''(x)|.$$  

(18)

Then the unique step which minimizes the error is

$$\bar{h} = \sqrt{\frac{2\epsilon_M|f(x)|}{|f''(x)|}}.$$  

(19)

Furthermore, assume that $f$ satisfies

$$|f(x)| \approx 1 \text{ and } \frac{1}{2}|f''(x)| \approx 1.$$  

(20)

Therefore, the approximate optimal step is

$$\bar{h} \approx \sqrt{\epsilon_M},$$  

(21)

where the approximate error is

$$E(\bar{h}) \approx 2\sqrt{\epsilon_M}.$$  

(22)
Figure 2: Total error of the numerical derivative as a function of the step in logarithmic scale - Theory.
Proof. The total error is minimized when the derivative of the function $E$ is zero. The first derivative of the function $E$ is

$$E'(h) = -\frac{\epsilon_M|f(x)|}{h^2} + \frac{1}{2}|f''(x)|. \quad (23)$$

The second derivative of $E$ is

$$E''(h) = 2\frac{\epsilon_M|f(x)|}{h^3}. \quad (24)$$

If we assume that $f(x) \neq 0$, then the second derivative $E''(h)$ is strictly positive, since $h > 0$ (i.e. we consider only non-zero steps). This first derivative $E'(h)$ is zero if and only if

$$-\frac{\epsilon_M|f(x)|}{h^2} + \frac{1}{2}|f''(x)| = 0 \quad (25)$$

Therefore, the optimal step is 19. If we make the additional assumptions 20, then the optimal step is given by 21. If we plug the equality 21 into the definition of the total error 17 and use the assumptions 20, we get the error as in 22, which concludes the proof.

The previous analysis shows that a more robust algorithm to compute the numerical first derivative of a function of one variable is:

```python
h = sqrt(%eps)
f_p = (f(x+h)-f(x))/h
```

In order to evaluate $f'(x)$, two evaluations of the function $f$ are required by formula 14 at points $x$ and $x + h$. In practice, the computational time is mainly consumed by the evaluation of $f$. The practical computation of 21 involves only the use of the elementary function $\sqrt{\epsilon}$, which is negligible.

In Scilab, we use double precision floating point numbers so that the rounding error is

$$\epsilon_M \approx 10^{-16}. \quad (26)$$

We are not concerned here with the exact value of $\epsilon_M$, since only the order of magnitude matters. Therefore, based on the simplified formula 21, the optimal step associated with the forward numerical difference is

$$h \approx 10^{-8}. \quad (27)$$

This is associated with the approximate error

$$E(h) \approx 2.10^{-8}. \quad (28)$$
3.4 Numerical experiments with the robust forward formula

We can introduce the accuracy of the function evaluation by modifying the equation 19. Indeed, if we take into account for 12 and 13, we get:

\[
\bar{h} = \sqrt{\frac{2c(x)\epsilon_M|f(x)|}{|f''(x)|}}
\]

(29)

\[
\bar{h} = \sqrt{\frac{2c(x)|f(x)|}{|f''(x)|}} \sqrt{\epsilon_M}.
\]

(30)

In practice, it is, unfortunately, not possible to compute the optimum step. It is still possible to analyse what happens in simplified situations where the exact derivative is known.

We now consider the function \( f(x) = \sqrt{x} \), for \( x \geq 0 \) and evaluate its numerical derivative at the point \( x = 1 \). In the following Scilab functions, we define the functions \( f(x) = \sqrt{x}, f'(x) = 1/2x^{-1/2} \) and \( f''(x) = -1/4x^{-3/2} \).

```scilab
function y = mysqrt ( x )
y = sqrt(x)
endfunction
function y = mydsqrt ( x )
y = 0.5 * x^( -0.5)
endfunction
function y = myddsqrt ( x )
y = -0.25 * x^( -1.5)
endfunction
```

The following Scilab functions define the approximate step \( h \) defined by \( h = \sqrt{\epsilon_M} \) and the optimum step \( h \) defined by 29.

```scilab
function y = step_approximate ( )
y = sqrt(%eps)
endfunction
function y = step_exact ( f , fpp , x )
y = sqrt( 2 * %eps * abs(f(x)) / abs(fpp(x)))
endfunction
```

The following functions define the forward numerical derivative and the relative error. The relative error is not defined for points \( x \) so that \( f'(x) = 0 \), but we will not consider this situation in this experiment.

```scilab
function y = forward ( f , x , h )
y = ( f(x+h) - f(x))/h
endfunction
function y = relativeerror ( f , fprime , x , h )
    expected = fprime ( x )
    computed = forward ( f , x , h )
    y = abs ( computed - expected ) / abs( expected )
endfunction
```

The following Scilab functions plots the relative error for several steps \( h \) from \( h = 10^{-16} \) to \( h = 1 \). The resulting data is plot in logarithmic scale.

```scilab
function drawrelativeerror ( f , fprime , x , mytitle )
n = 1000
```
logharray = linspace (-16,0,n)
for i = 1:n
  h = 10^(-logharray(i))
  logearray(i)=log10(relativeerror(f,fprime,x,h))
end
plot ( logharray , logearray )
xtitle(mytitle,"log(h)","log(E)")
endfunction

We now use the previous functions and execute the following Scilab statements.

```scilab
x = 1.0;
drawrelativeerror ( mysqrt , mydsqrt , x ,
    "Relative error of numerical derivative in x=1.0");
h1 = step_approximate ( );
mpprintf ( "Step Approximate = %e\n", h1)
h2 = step_exact ( mysqrt , myddsqrt , x );
mpprintf ( "Step Exact = %e\n", h2)
```

The previous script produces the following output:

```
Step Approximate = 1.490116e-008
Step Exact = 4.214685e-008
```

and plots the relative error presented in the figure 3.

We can compare the figures 2 and 3 and see that, indeed, the theory produces a maximal bound for the relative error. We also see that the difference between the approximate and the exact step is small in this particular case.

### 3.5 Backward formula

Let us consider Taylor’s expansion from equation 5, and use $-h$ instead of $h$. We get

$$ f(x - h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) + O(h^3) \quad (31) $$

This leads to the **backward formula**

$$ f'(x) = \frac{f(x) - f(x - h)}{h} + O(h) \quad (32) $$

As the forward formula, the backward formula is order 1. The analysis presented for the forward formula leads to the same results and will not be repeated, since the backward formula does not increase the accuracy.

### 3.6 Centered formula with 2 points

In this section, we derive the centered formula based on the two points $x \pm h$. We give the optimal step in double precision and the associated error.

**Proposition 3.2.** Let $f : \mathbb{R} \to \mathbb{R}$ be a continuously derivable function of one variable. Therefore,

$$ f'(x) \approx \frac{f(x + h) - f(x - h)}{2h} + \frac{h^2}{6} f''(x) + O(h^3). \quad (33) $$
Figure 3: Total error of the numerical derivative as a function of the step in logarithmic scale - Numerical experiment.
Proof. The Taylor expansion of the function \( f \) at point \( x \) is
\[
f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(x) + \mathcal{O}(h^4). \tag{34}
\]
If we replace \( h \) by \(-h\) in the previous equation we get
\[
f(x - h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f'''(x) + \mathcal{O}(h^4). \tag{35}
\]
We subtract the two equations \(34\) and \(35\) and get
\[
f(x + h) - f(x - h) = 2hf'(x) + \frac{h^3}{3} f'''(x) + \mathcal{O}(h^4). \tag{36}
\]
We immediately get \(33\), which concludes the proof, or, more simply, the centered 2 points finite difference
\[
f'(x) = \frac{f(x + h) - f(x - h)}{2h} + \mathcal{O}(h^2), \tag{37}
\]
which approximates \( f' \) at order 2.

Definition 3.3. (Centered two points finite difference for \( f' \)) The finite difference formula
\[
D f(x) = \frac{f(x + h) - f(x - h)}{2h} \tag{38}
\]
is the centered 2 points finite difference for \( f' \) and is an order 2 approximation for \( f' \).

Proposition 3.4. Let \( f : \mathbb{R} \to \mathbb{R} \) be a continuously derivable function of one variable. Consider the centered 2 points finite difference of \( f \) defined by \(38\). Assume that the total error implied by truncation and rounding is
\[
E(h) = \frac{\epsilon_M |f(x)|}{h} + \frac{h^2}{6} |f'''(x)|. \tag{39}
\]
Therefore, the unique step which minimizes the error is
\[
\overline{h} = \left( \frac{3r |f(x)|}{|f'''(x)|} \right)^{1/3}. \tag{40}
\]
Assume that \( f \) satisfies
\[
|f(x)| \approx 1 \text{ and } \frac{1}{3} |f'''(x)| \approx 1. \tag{41}
\]
Therefore, the approximate step which minimizes the error is
\[
\overline{h} \approx \epsilon_M^{1/3}. \tag{42}
\]
which is associated with the approximate error
\[
E(\overline{h}) \approx \frac{3}{2} \epsilon_M^{2/3}. \tag{43}
\]
Proof. The first derivative of the error is
\[ E'(h) = -\frac{\epsilon_M |f(x)|}{h^2} + \frac{h}{3} |f'''(x)|. \] (44)
The error is minimum when the first derivative of the error is zero
\[ -\frac{\epsilon_M |f(x)|}{h^2} + \frac{h^2}{3} |f'''(x)| = 0. \] (45)
The solution of this equation is 40. By the hypothesis 41, the optimal step is given by 42, which concludes the first part of the proof. If we plug the previous equality into the definition of the total error 39 and use the assumptions 41, we get the error given by 43, which concludes the proof. \(\square\)

With double precision floating point numbers, the optimal step associated with the centered numerical difference is
\[ \bar{h} \approx 6.10^{-6}. \] (46)
This is associated with the error
\[ E(\bar{h}) \approx 5.10^{-11}. \] (47)

3.7 Centered formula with 4 points
In this section, we derive the centered formula based on the fours points \(x \pm h\) and \(x \pm 2h\). We give the optimal step in double precision and the associated error.

**Proposition 3.5.** Let \(f : \mathbb{R} \to \mathbb{R}\) be a continuously derivable function of one variable. Therefore,
\[
f'(x) = \frac{8f(x + h) - 8f(x - h) - f(x + 2h) + f(x - 2h)}{12h} + \frac{h^4}{30} f^{(5)}(x) + \mathcal{O}(h^5).
\] (48)

**Proof.** The Taylor expansion of the function \(f\) at point \(x\) is
\[
f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f^{(2)}(x) + \frac{h^3}{6} f^{(3)}(x) + \frac{h^4}{24} f^{(4)}(x) + \frac{h^5}{120} f^{(5)}(x) + \mathcal{O}(h^6).
\] (49)
If we replace \(h\) by \(-h\) in the previous equation we get
\[
f(x - h) = f(x) - hf'(x) + \frac{h^2}{2} f^{(2)}(x) - \frac{h^3}{6} f^{(3)}(x) + \frac{h^4}{24} f^{(4)}(x) - \frac{h^5}{120} f^{(5)}(x) + \mathcal{O}(h^6).
\] (50)
We subtract the two equations 49 and 50 and get

\[ f(x + h) - f(x - h) = 2hf'(x) + \frac{h^3}{3} f^{(3)}(x) + \frac{h^5}{60} f^{(5)}(x) + O(h^6). \]  

(51)

We replace \( h \) by \( 2h \) in the previous equation and get

\[ f(x + 2h) - f(x - 2h) = 4hf'(x) + \frac{8h^3}{3} f^{(3)}(x) + \frac{8h^5}{15} f^{(5)}(x) + O(h^6). \]  

(52)

In order to eliminate the term \( f^{(3)}(x) \), we multiply the equation 51 by 8 and get

\[ 8 (f(x + h) - f(x - h)) = 16hf'(x) + \frac{8h^3}{3} f^{(3)}(x) + \frac{2h^5}{15} f^{(5)}(x) + O(h^6). \]  

(53)

We subtract equations 52 and 53 and we have

\[ 8 (f(x + h) - f(x - h)) - (f(x + 2h) - f(x - 2h)) = 12hf'(x) - \frac{6h^5}{15} f^{(5)}(x) + O(h^6). \]  

(54)

We divide the previous equation by 12\( h \) and get

\[ \frac{8 (f(x + h) - f(x - h)) - (f(x + 2h) - f(x - 2h))}{12h} = f'(x) - \frac{h^4}{30} f^{(5)}(x) + O(h^5), \]  

which implies the equation 48 or, more simply,

\[ f'(x) = \frac{8f(x + h) - 8f(x - h) - f(x + 2h) + f(x - 2h)}{12h} + O(h^4), \]  

(56)

which is the centered 4 points formula of order 4.

\[ \square \]

**Definition 3.6.** (Centered 4 points finite difference for \( f' \)) The finite difference formula

\[ Df(x) = \frac{8f(x + h) - 8f(x - h) - f(x + 2h) + f(x - 2h)}{12h} \]  

(57)

is the centered 4 points finite difference for \( f' \).

**Proposition 3.7.** Let \( f : \mathbb{R} \rightarrow \mathbb{R} \) be a continuously derivable function of one variable. Consider the centered centered 4 points finite difference of \( f \) defined by 57. Assume that the total error implied by truncation and rounding is

\[ E(h) = \frac{\epsilon_M |f(x)|}{h} + \frac{h^4}{30} |f^{(5)}(x)|. \]  

(58)

Therefore, the optimal step is

\[ h = \left( \frac{15\epsilon_M |f(x)|}{2 |f^{(5)}(x)|} \right)^{1/5}. \]  

(59)
Assume that $f$ satisfies

$$|f(x)| \approx 1 \text{ and } \frac{2}{15}|f^{(5)}(x)| \approx 1,$$

(60)

Therefore, the approximate step

$$\bar{h} \approx \frac{1}{5} \epsilon_m,$$

(61)

which is associated with the error

$$E(\bar{h}) \approx \frac{5}{4} \epsilon_m^{4/5}.$$

(62)

Proof. The first derivative of the error is

$$E'(h) = -\frac{\epsilon_m |f(x)|}{h^2} + \frac{2h^3}{15} |f^{(5)}(x)|.$$

(63)

The error is minimum when the first derivative of the error is zero

$$-\frac{\epsilon_m |f(x)|}{\bar{h}^2} + \frac{2\bar{h}^3}{15} |f^{(5)}(x)| = 0.$$

(64)

The solution of the previous equation is the step 59. If we make the assumptions 60, then the optimal step is 61, which concludes the first part of the proof. If we plug the equality 61 into the definition of the total error 58 and use the assumptions 60, we get the error 62, which concludes the proof.

With double precision floating point numbers, the approximate optimal step associated with the centered 4 points numerical difference is

$$\bar{h} \approx 4.10^{-4}.$$

(65)

This is associated with the approximate error

$$E(\bar{h}) \approx 3.10^{-13}.$$

(66)

### 3.8 Some finite difference formulas for the first derivative

In this section, we present several formulas to compute the first derivative of a function of several parameters. We present and compare the associated optimal steps and optimal errors.

The figure 4 present various formulas for the computation of the first derivative of a continuously derivable function $f$. The approximate optimum step $\bar{h}$ and the approximate minimum error $E(\bar{h})$ are computed for double precision floating point numbers. We do not take into account for the scaling with respect to $x$ (see below).

The figure 5 present the optimal steps and the associated errors for various finite difference formulas.

We notice that with increasing accuracy (i.e. with order from 1 to 4), the size of the step increases, while the error decreases.
<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
<th>$\tilde{h}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward 2 points</td>
<td>$f(x + h) - f(x)$</td>
<td>$\sqrt{\epsilon_M}$</td>
</tr>
<tr>
<td>Centered 2 points</td>
<td>$f(x + h) - f(x - h)$</td>
<td>$\epsilon_M^{1/3}$</td>
</tr>
<tr>
<td>Centered 4 points</td>
<td>$-8f(x + h) - 8f(x - h) + 12f(x - 2h)$</td>
<td>$\epsilon_M^{1/5}$</td>
</tr>
</tbody>
</table>

Figure 4: Various formulas for the computation of the Jacobian of a given function $f$.

<table>
<thead>
<tr>
<th>Name</th>
<th>$h$</th>
<th>$E(h)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward 2 points</td>
<td>$10^{-8}$</td>
<td>$2.10^{-8}$</td>
</tr>
<tr>
<td>Centered 2 points</td>
<td>$6.10^{-6}$</td>
<td>$5.10^{-11}$</td>
</tr>
<tr>
<td>Centered 4 points</td>
<td>$4.10^{-4}$</td>
<td>$3.10^{-13}$</td>
</tr>
</tbody>
</table>

Figure 5: Optimal steps and error of finite difference formulas for the computation of the Jacobian of a given function $f$ with double precision floating point numbers. We do not take into account for the scaling with respect to $x$.

3.9 A three points formula for the second derivative

In this section, we present a three points formula for the second derivative of a function of one variable. We present the error analysis and compute the optimum step and minimum error.

**Proposition 3.8.** Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a continuously derivable function of one variable. Therefore,

$$f''(x) = \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} + \frac{h^2}{12} f^{(4)}(x) + \mathcal{O}(h^3). \quad (67)$$

**Proof.** The Taylor expansion of the function $f$ at point $x$ is

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f^{(2)}(x) + \frac{h^3}{6} f^{(3)}(x) + \frac{h^4}{24} f^{(4)}(x) + \mathcal{O}(h^5). \quad (68)$$

If we replace $h$ by $-h$ in the previous equation we get

$$f(x - h) = f(x) - hf'(x) + \frac{h^2}{2} f^{(2)}(x) - \frac{h^3}{6} f^{(3)}(x) + \frac{h^4}{24} f^{(4)}(x) - \frac{h^5}{120} f^{(5)}(x) + \mathcal{O}(h^6). \quad (69)$$

We sum the two equations 68 and 69 and get

$$f(x + h) + f(x - h) = 2f(x) + h^2 f''(x) + \frac{h^4}{12} f^{(4)}(x) + \mathcal{O}(h^5). \quad (70)$$

This leads to the three points finite difference formula 67, or, more simply,

$$f''(x) = \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} + \mathcal{O}(h^2). \quad (71)$$
The formula 71 shows that this three points finite difference is order 2.

**Definition 3.9.** (Centered 3 points finite difference for \( f'' \)) The finite difference formula

\[
Df(x) = \frac{f(x + h) - 2f(x) + f(x - h)}{h^2}
\]

is the centered 3 points finite difference for \( f'' \).

**Proposition 3.10.** Let \( f : \mathbb{R} \to \mathbb{R} \) be a continuously derivable function of one variable. Consider the centered centered 4 points finite difference of \( f \) defined by 72. Assume that the total error implied by truncation and rounding is

\[
E(h) = \frac{\epsilon_M |f(x)|}{h^2} + \frac{h^2}{12} |f^{(4)}(x)|.
\]

Therefore, the unique step which minimizes the error is

\[
\bar{h} = \left( \frac{12 \epsilon_M |f(x)|}{|f^{(4)}(x)|} \right)^{1/4}.
\]

Assume that \( f \) satisfies

\[
|f(x)| \approx 1 \quad \text{and} \quad \frac{1}{12} |f^{(4)}(x)| \approx 1.
\]

Therefore, the approximate step is

\[
\bar{h} \approx \epsilon_M^{1/4},
\]

which is associated with the approximate error

\[
E(\bar{h}) \approx 2 \epsilon_M^{1/2}.
\]

**Proof.** The first derivative of the error is

\[
E'(h) = -\frac{2r|f(x)|}{h^3} + \frac{h}{6} |f^{(4)}(x)|.
\]

Its second derivative is

\[
E''(h) = \frac{6r|f(x)|}{h^4} + \frac{1}{6} |f^{(4)}(x)|.
\]

The second derivative is positive, since, by hypothesis, we have \( h > 0 \). Therefore, the function \( E \) is convex and has only one global minimum. The error \( E \) is minimum when the first derivative of the error is zero

\[
-\frac{2r|f(x)|}{h^3} + \bar{h}^3 |f^{(4)}(x)| = 0.
\]

Therefore, the optimal step is given by the equation 74. By the hypothesis 75, the optimal step is given by 76, which concludes the first part of the proof. If we plug the equality 76 into the definition of the total error 73 and use the assumptions 75, we get the error 77, which concludes the proof.
With double precision floating point numbers, the optimal step associated with the centered 4 points numerical difference is
\[ h \approx 1.10^{-4}. \] (81)
This is associated with the error
\[ E(h) = 3.10^{-8}. \] (82)

### 3.10 Accuracy of finite difference formulas

In this section, we give a proposition which computes the order of magnitude of many finite difference formulas.

**Proposition 3.11.** Let \( f : \mathbb{R} \to \mathbb{R} \) be a continuously derivable function of one variable. We consider the derivative \( f^{(d)} \), where \( d \geq 1 \) is a positive integer. Assume that the derivative \( f^{(d)} \) is approximated by a finite difference formula. Assume that the rounding error associated with the finite difference formula is
\[ E_r(h) = \epsilon M |f(x)| h^d. \] (83)
Assume that the associated truncation error is
\[ E_t(h) = \frac{h^p}{\beta} |f^{(d+p)}(x)|, \] (84)
where \( \beta > 0 \) is a positive constant, \( p \geq 1 \) is a strictly positive integer associated with the order of the finite difference formula. Therefore, the unique step which minimizes the total error is
\[ \bar{h} = \left( \frac{\epsilon M |f(x)|}{\epsilon M - \frac{d}{p} \beta |f^{(d+p)}(x)|} \right)^{\frac{1}{d+p}}. \] (85)
Assume that the function \( f \) is so that
\[ |f(x)| \approx 1 \text{ and } \frac{1}{\beta} |f^{(d+p)}(x)| \approx 1. \] (86)
Assume that the ratio \( d/p \) has an order of magnitude which is close to 1, i.e.
\[ \frac{d}{p} \approx 1. \] (87)
Then the unique approximate optimal step is
\[ \bar{h} \approx \epsilon M^\frac{1}{d+p}, \] (88)
and the associated error is
\[ E(\bar{h}) \approx 2 \epsilon M^\frac{p}{d+p}. \] (89)
This proposition allows to compute the optimum step much faster than with a case by case analysis. The assumptions might seem to be strong at first, but, as we have already seen, are reasonable in practice.

\textbf{Proof.} The total error is

\[ E(h) = \frac{\epsilon_M |f(x)|}{h^d} + \frac{h^p}{\beta} |f^{(d+p)}(x)|. \]  

(90)

The first derivative of the error \( E \) is

\[ E'(h) = -d \frac{\epsilon_M |f(x)|}{h^{d+1}} + p h^{p-1} \frac{f^{(d+p)}(x)}{\beta}. \]  

(91)

The second derivative of the error \( E \) is

\[ E''(h) = \begin{cases} 
  d(d + 1) \frac{\epsilon_M |f(x)|}{h^{d+2}}, & \text{if } p = 1 \\
  d(d + 1) \frac{\epsilon_M |f(x)|}{h^{d+2}} + p(p - 1) h^{p-2} \frac{f^{(d+p)}(x)}{\beta}, & \text{if } p \geq 2 
\end{cases} \]  

(92)

Therefore, whatever the value of \( p \geq 1 \), the second derivative of the error \( E \) is positive. Hence, the function \( E \) is convex for \( h > 0 \). This implies that there is only one global minimum, which is the solution of the equation \( E'(h) = 0 \). The optimum step \( h \) satisfies the equation

\[ -d \frac{\epsilon_M |f(x)|}{h^{d+1}} + p h^{p-1} \frac{f^{(d+p)}(x)}{\beta} = 0. \]  

(93)

This leads to the equation 85. Under the assumptions on the function \( f \) given by 86 and on the factor \( \frac{d}{p} \) given by 87, the previous equality simplifies into

\[ h = \frac{\epsilon_M}{d^{\frac{1}{p}}}, \]  

(94)

which proves the first result. The same assumptions simplify the approximate error into

\[ E(h) \approx \frac{\epsilon_M}{h^d} + h^p. \]  

(95)

If we introduce the optimal step 94 into the previous equation, we get

\[ E(h) \approx \frac{\epsilon_M}{d^{\frac{p}{p}}} + \frac{\epsilon_M}{d^{\frac{p}{p}}} \]  

(96)

\[ \approx \frac{\epsilon_M}{d^{\frac{p}{p}}} + \frac{\epsilon_M}{d^{\frac{p}{p}}} \]  

(97)

\[ \approx 2 \frac{\epsilon_M}{d^{\frac{p}{p}}}, \]  

(98)

which concludes the proof. \( \Box \)
Example 3.1 Consider the following centered 3 points finite difference for $f''$

$$f''(x) = \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} + \frac{h^2}{12} f^{(4)}(x) + O(h^3). \tag{99}$$

The error implied by truncation and rounding is

$$E(h) = \frac{\epsilon M |f(x)|}{h^2} + \frac{h^2}{12} |f^{(4)}(x)|, \tag{100}$$

which can be interpreted in the terms of the proposition 3.11 with $d = 2$, $p = 2$ and $\beta = 12$. Then the unique approximate optimal step is

$$\bar{h} \approx \frac{\epsilon}{1!} M,$$ 

and the associated approximate error is

$$E(\bar{h}) \approx 2\epsilon M.$$

This result corresponds to the proposition 3.10, as expected.

3.11 A collection of finite difference formulas

In this section, we present some finite difference formulas which compute various derivatives with various orders of precision. For each formula, the optimum step and the minimum error is presented, under the assumptions of the proposition 3.11.

- First derivative : forward 2 points

$$f'(x) = \frac{f(x + h) - f(x)}{h} + O(h) \tag{103}$$

Optimal step : $h \approx \epsilon^{1/2} M$ and error $E \approx \epsilon^{1/2} M$.  
Double precision $h \approx 10^{-8}$ and $E \approx 10^{-8}$.

- First derivative : backward 2 points

$$f'(x) = \frac{f(x) - f(x - h)}{h} + O(h) \tag{104}$$

Optimal step : $h \approx \epsilon^{1/2} M$ and error $E \approx \epsilon^{1/2} M$.  
Double precision $h \approx 10^{-8}$ and $E \approx 10^{-8}$.

- First derivative : centered 2 points

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} + O(h^2) \tag{105}$$

Optimal step : $h = \epsilon^{1/3} M$ and error $E \approx \epsilon^{2/3} M$.  
Double precision $h \approx 10^{-5}$ and $E \approx 10^{-10}$. 

26
• First derivative : double forward 3 points

\[ f'(x) = \frac{-f(x + 2h) + 4f(x + h) - 3f(x)}{2h} + \mathcal{O}(h^2) \] (106)

Optimal step : \( h \approx \epsilon^{1/3}_M \) and error \( E \approx \epsilon^{2/3}_M \).
Double precision \( h \approx 10^{-5} \) and \( E \approx 10^{-10} \).

• First derivative : double backward 3 points

\[ f'(x) = \frac{f(x - 2h) - 4f(x + h) + 3f(x)}{2h} + \mathcal{O}(h^2) \] (107)

Optimal step : \( h \approx \epsilon^{1/3}_M \) and error \( E \approx \epsilon^{2/3}_M \).
Double precision \( h \approx 10^{-5} \) and \( E \approx 10^{-10} \).

• First derivative : centered 4 points

\[ f'(x) = \frac{1}{12h} \left( -f(x + 2h) + 8f(x + h) - 8f(x - h) + f(x - 2h) \right) + \mathcal{O}(h^4) \] (108)

Optimal step : \( h \approx \epsilon^{1/5}_M \) and error \( E \approx \epsilon^{4/5}_M \).
Double precision \( h \approx 10^{-3} \) and \( E \approx 10^{-12} \).

• Second derivative : forward 3 points

\[ f''(x) = \frac{f(x + 2h) - 2f(x + h) + f(x)}{h^2} + \mathcal{O}(h) \] (109)

Optimal step : \( h \approx \epsilon^{1/3}_M \) and error \( E \approx \epsilon^{1/3}_M \).
Double precision \( h \approx 10^{-6} \) and \( E \approx 10^{-6} \).

• Second derivative : centered 3 points

\[ f''(x) = \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} + \mathcal{O}(h^2) \] (110)

Optimal step : \( h \approx \epsilon^{1/4}_M \) and error \( E \approx \epsilon^{1/2}_M \).
Double precision \( h \approx 10^{-4} \) and \( E \approx 10^{-8} \).

• Second derivative : centered 5 points

\[ f''(x) = \frac{1}{12h^2} \left( -f(x + 2h) + 16f(x + h) - 30f(x) + 16f(x - h) - f(x - 2h) \right) + \mathcal{O}(h^4) \] (111)

Optimal step : \( h \approx \epsilon^{1/6}_M \) and error \( E \approx \epsilon^{2/3}_M \).
Double precision \( h \approx 10^{-2} \) and \( E \approx 10^{-10} \).
• Third derivative : centered 4 points

\[
f^{(3)}(x) = \frac{1}{2h^3} (f(x + 2h) - 2f(x + h) + 2f(x - h) - f(x - 2h)) + O(h^2) \]

(112)

Optimal step : \( h \approx \epsilon_1^{1/5} M \) and error \( E \approx \epsilon_2^{2/5} M \).

Double precision \( h \approx 10^{-3} \) and \( E \approx 10^{-6} \).

• Fourth derivative : centered 5 points

\[
f^{(4)}(x) = \frac{1}{h^2} (f(x + 2h) - 4f(x + h) + 6f(x) - 4f(x - h) + f(x - 2h)) + O(h^2) \]

(113)

Optimal step : \( h \approx \epsilon_1^{1/6} M \) and error \( E \approx \epsilon_1^{1/3} M \).

Double precision \( h \approx 10^{-2} \) and \( E \approx 10^{-5} \).

Some of the previous formulas will be presented in the context of Scilab in the section 4.3.

4 Finite differences of multivariate functions

In this section, we analyse methods to approximate the derivatives of multivariate functions with Scilab. In the first part, we present the gradient and Hessian of a multivariate function. Then we analyze methods to compute the derivatives of multivariate functions with finite differences. We present Scilab functions to compute these derivatives. By composing the finite difference operators, it is possible to approximate higher degree derivatives and we present how to use this method with Scilab. Finally, we present Richardson’s method to approximate derivatives with more accuracy and discuss methods to take bounds into account.

4.1 Multivariate functions

In this section, we present formulas which allow to compute the numerical derivatives of multivariate function.

Assume that \( n \) is a positive integer representing the dimension of the space. Assume that \( f \) is a multivariate continuously differentiable function : \( f : \mathbb{R}^n \to \mathbb{R} \). We denote by \( \mathbf{x} \in \mathbb{R}^n \) the current vector with \( n \) dimensions. The \( n \)-vector of partial derivatives of \( f \) is the gradient of \( f \) and will be denoted by \( \nabla f(\mathbf{x}) \) or \( \mathbf{g}(\mathbf{x}) \):

\[
\nabla f(\mathbf{x}) = \mathbf{g}(\mathbf{x}) = \left( \frac{\partial f}{\partial x_1} \right) : \left( \begin{array}{c} \vdots \frac{\partial f}{\partial x_n} \end{array} \right). \]

\[ \] (114)

Consider the function \( f : \mathbb{R}^n \to \mathbb{R}^m \), where \( m \) is a positive integer. Then the partial derivatives form a \( n \times m \) matrix, which is called the Jacobian matrix. In this
document, we will consider only the case \( m = 1 \), but the results which are presented can be applied directly to each component of \( f(x) \). Hence, the case \( m > 1 \) does not introduce any new problem and we will not consider it in the remaining of this document.

Higher derivatives of a multivariate function are defined as in the univariate case. Assume that \( f \) has continuous partial derivatives \( \partial f/\partial x_i \) for \( i = 1, \ldots, n \) and continuous partial derivatives \( \partial^2 f/\partial x_i \partial x_j i, j = 1, \ldots, n \). Then the Hessian matrix of \( f \) is denoted by \( \nabla^2 f(x) \) of \( H(x) \):

\[
\nabla^2 f(x) = H(x) = \begin{pmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2}
\end{pmatrix}.
\]

The Taylor-series expansion of a general function \( f \) in the neighbourhood of a point \( x \) can be derived as in the univariate case presented in the section 2.1.1. Let \( x \in \mathbb{R}^n \) be a given point, \( p \in \mathbb{R}^n \) a vector of unit length and \( h \in \mathbb{R} \) a scalar. The function \( f(x + hp) \) can be regarded as a univariate function of \( h \) and the univariate expansion can be applied directly:

\[
f(x + hp) = f(x) + hg(x)^T p + \frac{1}{2} h^2 p^T H(x)p + \ldots + \frac{1}{(n-1)!} h^{n-1} D^{n-1} f(x) + \frac{1}{n!} h^n D^n f(x + \theta h p),
\]

for some \( \theta \in [0,1] \) and where

\[
D^s f(x) = \sum_{i_1=1} \sum_{i_2=1} \cdots \sum_{i_s=1} p_{i_1} p_{i_2} \cdots p_{i_s} \frac{\partial^s f(x)}{\partial x_{i_1} \partial x_{i_2} \ldots \partial x_{i_s}}.
\]

We can expand Taylor’s formula, keep only the first three terms of this expansion and get:

\[
f(x + hp) = f(x) + hg(x)^T p + \frac{1}{2} h^2 p^T H(x)p + \mathcal{O}(h^3).
\]

The term \( hg(x)^T p \) is the directional derivative of \( f \) and is an order 1 term which drives the rate of change of \( f \) at the point \( x \). The order 2 term \( p^T H(x)p \) is the curvature of \( f \) along \( p \). A direction \( p \) such that \( p^T H(x)p > 0 \) is termed a direction of positive curvature.

In the particular case of a function of two variables, the previous general formula
can be written in integral form:

\[
f(x_1 + h_1, x_2 + h_2) = f(x_1, x_2) + h_1 \frac{\partial f}{\partial x_1} + h_2 \frac{\partial f}{\partial x_2} + \frac{h_1^2}{2} \frac{\partial^2 f}{\partial x_1^2} + h_1 h_2 \frac{\partial^2 f}{\partial x_1 \partial x_2} + \frac{h_2^2}{2} \frac{\partial^2 f}{\partial x_2^2} + \frac{h_1^3}{6} \frac{\partial^3 f}{\partial x_1^3} + h_1 h_2^2 \frac{\partial^3 f}{\partial x_1 \partial x_2^2} + \frac{h_2^3}{6} \frac{\partial^3 f}{\partial x_2^3} + \frac{h_1 h_2}{24} \frac{\partial^4 f}{\partial x_1^2 \partial x_2^2} + \frac{h_1^2 h_2}{6} \frac{\partial^4 f}{\partial x_1^3 \partial x_2} + \frac{h_1^3 h_2}{4} \frac{\partial^4 f}{\partial x_1^4} + \frac{h_2^3}{24} \frac{\partial^4 f}{\partial x_2^4} + \ldots + \sum_{m+n=p} \frac{h_1^m h_2^n}{m! n!} \int_0^1 \frac{\partial^p f}{\partial x_1^m \partial x_2^n} (x_1 + th_1, x_2 + th_2) p(1-t)^{p-1} dt, \]

where the terms associated with the partial derivatives of degree \( p \) have the form

\[
\sum_{m+n=p} \frac{h_1^m h_2^n}{m! n!} \frac{\partial^p f}{\partial x_1^m \partial x_2^n}. \tag{120}
\]

### 4.2 Numerical derivatives of multivariate functions

The Taylor-series expansion of a general function \( f \) allows to derive approximation of the function in a neighbourhood of \( x \). Indeed, if we keep the first term in the expansion, we get

\[
f(x + hp) = f(x) + hg(x)^T p + \mathcal{O}(h^2). \tag{121}
\]

This formula leads to an order 1 finite difference formula for the multivariate function \( f \). We emphasize that the equation 121 is a univariate expansion in the direction \( p \). This is why the univariate finite difference formulas can be directly applied for multivariate functions. Let \( h_i \) be the step associated with the \( i \)-th component of \( x \), and let \( e_i \in \mathbb{R}^n \) be the vector \( e_i = ((e_i)_1, (e_i)_2, \ldots, (e_i)_n)^T \) with

\[
(e_i)_j = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j, \end{cases} \tag{122}
\]

for \( j = 1, \ldots, n \). Then,

\[
f(x + h_i e_i) = f(x) + h_i g(x)^T e_i + \mathcal{O}(h^2). \tag{123}
\]

The term \( g(x)^T e_i \) is the \( i \)-th component of the gradient \( g(x) \), so that \( g(x)^T e_i = g_i(x) \). Therefore, we can approximate the gradient of the function \( f \) by the finite difference formula

\[
g_i(x) = \frac{f(x + h_i e_i) - f(x)}{h_i} + \mathcal{O}(h). \tag{124}
\]

The previous formula is a multivariate finite difference formula of order 1 for the gradient of the function \( f \). It is the direct analog of univariate finite differences formulas that we have previously analyzed.
Similarly to the univariate case, the centered 2 points multivariate finite difference for the gradient of $f$ is

$$g_i(x) = \frac{f(x + h_i e_i) - f(x - h_i e_i)}{h_i} + O(h^2) \quad (125)$$

and the centered 4 points multivariate finite difference for the gradient of $f$ is

$$g_i(x) = \frac{8f(x + h_i e_i) - 8f(x - h_i e_i) - f(x + 2h_i e_i) + f(x - 2h_i e_i)}{12h_i} + O(h^4). \quad (126)$$

We have already noticed that the previous formulas are simply the univariate formula in the direction $h_i e_i$. The consequence is that the evaluation of the gradient vector $g$ requires $n$ univariate finite differences.

4.3 Derivatives of a multivariate function in Scilab

In this section, we present a function which computes the Jacobian of a multivariate function $f$.

The following `derivativeJacobianStep` function computes the approximate optimal step for some of the formulas for the first derivative. The function takes the formula name `form` as input argument and returns the approximate (scalar) optimal step $h$.

```plaintext
function h = derivativeJacobianStep(form)
    select form
    case "forward2points" then // Order 1
        h = %eps^(1/2)
    case "backward2points" then // Order 1
        h = %eps^(1/2)
    case "centered2points" then // Order 2
        h = %eps^(1/3)
    case "doubleforward3points" then // Order 2
        h = %eps^(1/3)
    case "doublebackward3points" then // Order 2
        h = %eps^(1/3)
    case "centered4points" then // Order 4
        h = %eps^(1/5)
    else
        error(msprintf("Unknown formula %s",form))
    end
endfunction
```

The following `derivativeJacobian` function computes an approximate Jacobian. It takes as input argument the function $f$, the vector point $x$, the vector step $h$ and the formula `form` and returns the approximate Jacobian $J$.

```plaintext
function J = derivativeJacobian(f,x,h,form)
    n = size(x,"*")
    D = diag(diag(h))
    for i = 1 : n
        d = D(:,i)
        select form
        case "forward2points" then // Order 1
```
\[ J(i) = \frac{(f(x+d)-f(x))}{h(i)} \]

\[ \text{case } "backward2points" \text{ then } // \text{ Order 1} \]
\[ J(i) = \frac{(f(x)-f(x-d))}{h(i)} \]

\[ \text{case } "centered2points" \text{ then } // \text{ Order 2} \]
\[ J(i) = \frac{(f(x+d)-f(x-d))}{2 \times h(i)} \]

\[ \text{case } "doubleforward3points" \text{ then } // \text{ Order 2} \]
\[ J(i) = \frac{(-f(x+2d)+4f(x+d)-3f(x))}{2 \times h(i)} \]

\[ \text{case } "doublebackward3points" \text{ then } // \text{ Order 2} \]
\[ J(i) = \frac{(f(x-2d)-4f(x-d)+3f(x))}{2 \times h(i)} \]

\[ \text{case } "centered4points" \text{ then } // \text{ Order 4} \]
\[ J(i) = \frac{(-f(x+2d)+8f(x+d)-8f(x-d)+f(x-2d))}{12 \times h(i)} \]

\[ \text{else} \]
\[ \text{error(msprintf("Unknown formula %s",form))} \]
\[ \text{end} \]

end

e ndfunction

In the previous function, the statement \( D=\text{diag}(h) \) creates a diagonal matrix \( D \) where the diagonal entries are equal to the vector \( h \). Therefore, the \( i \)-th column of \( D \) is equal to \( h_i e_i \), as defined in the previous section.

We now experiment our approximate Jacobian function. The following function \text{quadf} \ computes a quadratic function.

\[
\text{function } f = \text{quadf} \text{ ( x )} \\
f = x(1)^2 + x(2)^2
\]
endfunction

The \text{quadJ} function computes the exact Jacobian of \text{quadf}.

\[
\text{function } J = \text{quadJ} \text{ ( x )} \\
J(1) = 2 \times x(1) \\
J(2) = 2 \times x(2)
\]
endfunction

In the following session, we compute the exact Jacobian matrix at the point \( x = (1,2)^T \).

\[ \text{-->x} = [1;2]; \]
\[ \text{-->J} = \text{quadJ} \text{ ( x )} \]
\[ J = \]
\[ 2. \]
\[ 4. \]

In the following session, we compute the approximate Jacobian matrix at the point \( x = (1,2)^T \).

\[ \text{-->form} = "\text{forward2points}"; \]
\[ \text{-->h} = \text{derivativeJacobianStep}(\text{form}) \]
\[ h = \]
\[ 0.0007401 \]
\[ \text{-->h} = h*\text{ones}(2,1) \]
\[ h = \]
\[ 0.0007401 \]
\[ 0.0007401 \]
\[ \text{-->Japprox} = \text{derivativeJacobian}(\text{quadf},\text{x},h,\text{form}) \]
\[ Japprox = \]
\[ 2. \]
\[ 4. \]
Although the `derivativeJacobian` function has interesting features, there are some limitations.

- We cannot compute the Jacobian matrix of a function which returns a m-by-1 vector: only scalar functions can be differentiated.
- We cannot differentiate a function \( f \) which requires extra-arguments.

Both these limitations are addressed in the next section.

### 4.4 Derivatives of a vectorial function with Scilab

In this section, we present a Scilab script which computes the Jacobian matrix of a vectorial function. This script will be used in the section 4.6, where we compose derivatives.

In order to manage extra-arguments, we will make so that the function to be differentiated can be either

- a function, with calling sequence \( y = f(x) \),
- a list \((f,a_1,a_2,\ldots)\). In this case, the first element in the list is the function to be differentiated with calling sequence \( y = f(x,a_1,a_2,\ldots) \), and the remaining arguments \( a_1,a_2,\ldots \) are automatically appended to the calling sequence.

Both cases are managed by the following `derivativeEvalf` function, which evaluates the function `__derEvalf__` at the given point \( x \).

```scilab
function y = derivativeEvalf(__derEvalf__,x)
  if ( typeof(__derEvalf__)=="function" ) then
    y = __derEvalf__(x)
  elseif ( typeof(__derEvalf__)=="list" ) then
    __f_fun__ = __derEvalf__(1)
    y = __f_fun__(x, __derEvalf__(2:$))
  else
    error(msprintf("Unknown function type %s", typeof(f)))
  end
endfunction
```

The complicated name `__derEvalf__` has been chosen in order to avoid conflicts between the name of the argument and the name of the user-defined function. Indeed, such a conflict may produce an infinite recursion. This topic is presented in more depth in [5].

The following `derivativeJacobian` function computes the Jacobian matrix of a given function `__derJacf__`.

```scilab
function J = derivativeJacobian(__derJacf__,x,h,form)
  n = size(x,"*")
  D = diag(h)
  for i = 1 : n
    d = D(:,i)
    select form
      case "forward2points" then // Order 1
        y(:,1) = -derivativeEvalf(__derJacf__,x)
        y(:,2) = derivativeEvalf(__derJacf__,x+d)
    end
  endfunction
```

33
The following quadf function takes as input argument a 3-by-1 vector and returns a 2-by-1 vector.

```matlab
function y = quadf ( x )
    f1 = x(1)^2 + x(2)^3 + x(3)^4
    f2 = exp(x(1)) + 2* sin(x(2)) + 3* cos(x(3))
    y = [f1;f2]
endfunction
```

The quadJ function returns the Jacobian matrix of quadf.

```matlab
function J = quadJ ( x )
    J1(1) = 2 * x(1)
    J1(2) = 3 * x(2)^2
    J1(3) = 4 * x(3)^3
    //
    J2(1) = exp(x(1))
    J2(2) = 2*cos(x(2))
    J2(3) = -3*sin(x(3))
    //
    J = [J1';J2']
endfunction
```

In the following session, we compute the exact Jacobian matrix of quadf at the point \( x = (1, 2, 3)^T \).

```plaintext
-->x=[1;2;3];
-->J = quadJ ( x )
J =
    2.    12.    108.
   2.7182818 -0.8322937 -0.4233600
```

The following quadf function takes as input argument a 3-by-1 vector and returns a 2-by-1 vector.

```matlab
function y = quadf ( x )
    f1 = x(1)^2 + x(2)^3 + x(3)^4
    f2 = exp(x(1)) + 2* sin(x(2)) + 3* cos(x(3))
    y = [f1;f2]
endfunction
```

The quadJ function returns the Jacobian matrix of quadf.

```matlab
function J = quadJ ( x )
    J1(1) = 2 * x(1)
    J1(2) = 3 * x(2)^2
    J1(3) = 4 * x(3)^3
    //
    J2(1) = exp(x(1))
    J2(2) = 2*cos(x(2))
    J2(3) = -3*sin(x(3))
    //
    J = [J1';J2']
endfunction
```

In the following session, we compute the exact Jacobian matrix of quadf at the point \( x = (1, 2, 3)^T \).

```plaintext
--->x=[1;2;3];
--->J = quadJ ( x )
J =
    2.    12.    108.
   2.7182818 -0.8322937 -0.4233600
```
In the following session, we compute the approximate Jacobian matrix of the function quadf.

```matlab
--> x = [1; 2; 3];
--> form = "forward2points";
--> h = derivativeJacobianStep(form);
--> Japprox = derivativeJacobian(quadf, x, h, form)
```

```
Japprox =
2.
12.
108.
2.7182819 - 0.8322937 - 0.4233600
```

4.5 Computing higher degree derivatives

In this section, we present a result which allows to get a finite difference operator for $f''$, based on a finite difference operator for $f'$. Consider the 2 points forward finite difference operator $Df$ defined by

$$Df(x) = \frac{f(x + h) - f(x)}{h},$$

which produce an order 1 approximation for $f'$. Similarly, let us consider the finite difference operator $DDf$ defined by

$$DDf(x) = \frac{Df(x + h) - Df(x)}{h},$$

that is, the composed operator $DDf = (D \circ D)f$. It would be nice if $DD$ was an approximation for $f''$. The previous formula simplifies into

$$DDf(x) = \frac{f(x + 2h) - 2f(x + h) + f(x)}{h^2}.$$  

It is straightforward to prove that the previous formula is, indeed, an order 1 formula for $f''$, that is, $DDf$ defined by 128 is an order 1 approximation for $f''$. The following proposition presents this result in a more general framework.

**Proposition 4.1.** Let $f : \mathbb{R} \to \mathbb{R}$ be a continuously derivable function of one variable. Let $Df$ be a finite difference operator of order $p > 0$ for $f'$. Therefore the finite difference operator $DDf$ is of order $p$ for $f''$.

**Proof.** By hypothesis, $Df$ is of order $p$, which implies that

$$Df(x) = f'(x) + O(h^p).$$

Let us define $g$ by

$$g(x) = Df(x).$$
Since \( f \) is continuously derivable function, so is \( g \). Therefore, \( Dg \) is of order \( p \) for \( g' \), which implies

\[
Dg(x) = g'(x) + \mathcal{O}(h^p). \tag{134}
\]

We now plug the definition of \( g \) given by 133 into the previous equation and get

\[
DDf(x) = (Df)'(x) + \mathcal{O}(h^p) \tag{135}
\]

\[
= f''(x) + \mathcal{O}(h^p) \tag{136}
\]

which concludes the proof.

**Example 4.1** Consider the centered 2 points finite difference for \( f' \) defined by

\[
Df(x) = \frac{f(x + h) - f(x - h)}{2h}. \tag{137}
\]

We have proved in proposition 3.2 that \( Df \) is an order 2 approximation for \( f' \). We can therefore apply the proposition 4.1 with \( p = 2 \) and get an approximation for \( f'' \) based on the finite difference

\[
DDf(x) = \frac{Df(x + h) - Df(x - h)}{2h}. \tag{138}
\]

We can expand this formula and get

\[
DDf(x) = \frac{f(x + 2h) - f(x) - 2f(x) + f(x - 2h)}{4h^2}, \tag{139}
\]

\[
= \frac{f(x + 2h) - f(x) - 2f(x) + f(x - 2h)}{4h^2}, \tag{140}
\]

which is, by proposition 4.1 a finite difference formula of order 2 for \( f'' \).

In practice, it may not be required to expand the finite difference in the way of 139. Indeed, Scilab can manage callbacks (i.e. function pointers), so that it is easy to use the proposition 4.1 so that the computation of the second derivative is performed with the same source code that for the first derivative. This method is used in the *derivative* function of Scilab, as we will see in the corresponding section.

We may ask if, by chance, a better result is possible for the finite difference \( DD \). More precisely, we may ask if the order of the operator \( DD \) may be greater than the order of the operator \( D \). In fact, there is no better result, as we are going to see. In order to analyse if a higher order formula would be produced, we must explicitly write higher order terms in the finite difference approximation. Let us write the finite difference operator \( Df \) by

\[
Df(x) = f'(x) + \frac{h^p}{\beta} f^{(d+p)}(x) + \mathcal{O}(h^{p+1}), \tag{141}
\]

where \( \beta > 0 \) is a positive real and \( d \geq 1 \) is an integer. We have

\[
DDf(x) = (Df)'(x) + \frac{h^p}{\beta} (Df)'^{(d+p)}(x) + \mathcal{O}(h^{p+1}) \tag{142}
\]

\[
= \left( f''(x) + \frac{h^p}{\beta} f^{(d+p+1)}(x) + \mathcal{O}(h^{p+1}) \right) + \frac{h^p}{\beta} \left( f^{(d+p+1)}(x) + \frac{h^p}{\beta} f^{(2d+2p)}(x) + \mathcal{O}(h^{p+1}) \right) + \mathcal{O}(h^{p+1}). \tag{143}
\]
Hence
\[
DDf(x) = f''(x) + 2\frac{h^p}{\beta} f^{(d+p+1)}(x) + \frac{h^{2p}}{\beta^2} f^{(2d+2p)}(x) + O(h^{p+1}).
\] (144)

We can see that the second term in the expansion is \(2\frac{h^p}{\beta} f^{(d+1)}(x)\), which is of order \(p\). There is no assumption which may set this term to zero, which implies that \(DD\) is of order \(p\), at best.

Of course, the process can be extended in order to compute more derivatives. Suppose that we want to compute an approximation for \(f^{(d)}(x)\), where \(d \geq 1\) is an integer. Let us define the finite difference operator \(D^{(d)}f\) by recurrence on \(d\) as
\[
D^{(d+1)}f(x) = D \circ D^{(d)}f(x).
\] (145)

By proposition 4.1, if \(Df\) is a finite difference operator of order \(p\) for \(f'\), therefore \(D^{(d)}f\) is a finite difference operator of order \(p\) for \(f^{(d)}(x)\).

We present how to implement the previous composition formulas in the section 4.6. But, for reasons which will be made clear later in this document, we must first consider derivatives of multivariate functions and derivatives of vectorial functions.

### 4.6 Nested derivatives with Scilab

In this section, we present how to compute higher derivatives with Scilab, based on recursive calls.

We define a function which returns the Jacobian matrix as a column vector. Moreover, we have to create a function with a calling sequence which is compatible with the one required by \texttt{derivativeJacobian}. The following \texttt{derivativeFunctionJ} function returns the Jacobian matrix at the point \(x\).

```matlab
function h = derivativeHessianStep(form)
    select form
    case "forward2points" then // Order 1
        h = %eps^(1/3)
    case "backward2points" then // Order 1
        h = %eps^(1/3)
    case "centered2points" then // Order 2
        h = %eps^(1/4)
    case "doubleforward3points" then // Order 2
        h = %eps^(1/4)
    case "doublebackward3points" then // Order 2
        h = %eps^(1/4)
    case "centered4points" then // Order 4
        h = %eps^(1/6)
    else
        error(msprintf("Unknown formula %s",form))
    end
endfunction
```

We define a function which returns the Jacobian matrix as a column vector.
function J = derivativeFunctionJ(x,f,h,form)
    J = derivativeJacobian(f,x,h,form)
    J = J'
    J = J(:)
endfunction

Notice that the arguments x and f are switched in the calling sequence. The following session shows how the derivativeFunctionJ function changes the shape of J.

-->x=[1;2;3];
-->H = quadH ( x );
-->h = derivativeHessianStep("forward2points");
-->h = h*ones(3,1);
--> Japprox = derivativeFunctionJ (x, quadf, h, form)
Japprox =
    2.0000061
    12.000036
    108.00033
    2.7182901
    -0.8322992
    -0.4233510

The following quadH function returns the Hessian matrix of the quadf function, which was defined in the section 4.4.

function H = quadH ( x )
    H1 = [
        2 0 0
        0 6*x(2) 0
        0 0 12*x(3)^2
    ]
    //
    H2 = [
        exp(x(1)) 0 0
        0 -2*sin(x(2)) 0
        0 0 -3*cos(x(3))
    ]
    //
    H = [H1;H2]
endfunction

In the following session, we compute the Hessian matrix at the point $x = (1, 2, 3)^T$.

-->x=[1;2;3];
-->H = quadH ( x )
H =
    2. 0. 0.
    0. 12. 0.
    0. 0. 108.
    2.7182818 0. 0.
    0. -1.8185949 0.
    0. 0. 2.9699775

Notice that the rows #1 to #3 contain the Hessian matrix of the first component of quadf, while the rows #4 to #6 contain the Hessian matrix of the second component of quadf.

In the following session, we compute the approximate Hessian matrix of quadf. We use the approximate optimal step and the derivativeJacobian function, which
was defined in the section 4.4. The trick is that we differentiate `derivativeFunctionJ`, instead of `quadf`.

```plaintext
--> h = derivativeHessianStep("forward2points");
--> h = h*ones(3,1);
--> funlist = list(derivativeFunctionJ, quadf, h, form);
--> Happrox = derivativeJacobian(funlist,x,h,form)
```

\[
H_{approx} =
\begin{bmatrix}
1.9997533 & 0. & 0. \\
0. & 12.00007 & 0. \\
0. & 0. & 108.00063 \\
2.7182693 & 0. & 0. \\
0. & -1.8185741 & 0. \\
0. & 0. & 2.9699582
\end{bmatrix}
\]

Although the previous method seems interesting, it has a major drawback: it does not exploit the symmetry of the Hessian matrix, so that the number of function evaluations is larger than required. Indeed, the Hessian matrix of a smooth function \(f\) is symmetric, i.e.

\[
H_{ij} = H_{ji},
\]

for \(i, j = 1, 2, \ldots, n\). This relation comes as a consequence of the equality

\[
\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i},
\]

for \(i, j = 1, 2, \ldots, n\).

The symmetry implies that only the coefficients for which \(i \geq j\), for example, need to be computed: the coefficients \(i < j\) can be deduced by symmetry of the Hessian matrix. But the method that we have presented ignores this property. This leads to a number of function evaluations which could be divided roughly by a factor 2.

4.7 Computing derivatives with more accuracy

In this section, we present a method to compute derivatives with more accuracy. This method, known as Richardson’s extrapolation, improves the accuracy by using a sequence of steps with decreasing sizes.

We may ask if there is a general method to get an increased accuracy for a given derivative, from an existing finite difference formula. Of course, such a finite difference will require more function evaluations, which is the price to pay for an increased accuracy. The following proposition gives such a method.

**Proposition 4.2.** Assume that the finite difference operator \(Df\) approximates the derivative \(f^{(d)}\) at order \(p > 0\) where \(d, p \geq 1\) are integers. Assume that

\[
Df_h(x) = f^{(d)}(x) + \frac{h^p}{\beta} f^{(d+p)}(x) + O(h^q),
\]

where \(\beta > 0\) is a real constant and \(q\) is an integer greater than \(p\). Therefore, the finite difference operator

\[
\overline{D}f(x) = \frac{2^p Df_h(x) - Df_{2h}(x)}{2^p - 1}
\]

\[
39
\]
is an order $q$ approximation for $f^{(d)}$.

**Proof.** The proof is based on a direct use of the equation 148, with different steps $h$. With $2h$ instead of $h$ in 148, we have

$$Df_{2h}(x) = f^{(d)}(x) + \frac{2^p}{\beta} h^p f^{(d+p)}(x) + O(h^q). \quad (150)$$

We multiply the equation 148 by $2^p$ and get:

$$2^p Df_h(x) = 2^p f^{(d)}(x) + \frac{2^p}{\beta} h^p f^{(d+p)}(x) + O(h^q), \quad (151)$$

We subtract the equation 151 and the equation 150, and get

$$2^p Df_h(x) - Df_{2h}(x) = (2^p - 1) f^{(d)}(x) + O(h^q). \quad (152)$$

We divide both sides of the previous equation by $2^p - 1$ and get 149, which concludes the proof.

**Example 4.2** Consider the following centered 2 points finite difference operator for $f'$

$$Df_h(x) = \frac{f(x + h) - f(x - h)}{2h}. \quad (153)$$

We have proved in proposition 3.2 that this is an approximation for $f'(x)$ and

$$Df_h(x) = f'(x) + \frac{h^2}{6} f^{(3)}(x) + O(h^4). \quad (154)$$

Therefore, we can apply the proposition 4.2 with $d = 1$, $p = 2$, $\beta = 6$ and $q = 4$. Hence, the finite difference operator

$$\overline{D} f(x) = \frac{4Df_h(x) - Df_{2h}(x)}{3} \quad (155)$$

is an order $q = 4$ approximation for $f'(x)$. We can expand this new finite difference formula and find that we already have analysed it. Indeed, if we plug the definition of the finite difference operator 153 into 155, we get

$$\overline{D} f(x) = \frac{4f(x+h) - f(x-h)}{2h} - \frac{f(x+2h) - f(x-2h)}{4h}$$

$$= \frac{3}{12h} \left( 8(f(x+h) - f(x-h)) - (f(x+2h) - f(x-2h)) \right). \quad (156)$$

The previous finite difference operator is the one which has been presented in proposition 3.5, which states that it is an order 4 operator for $f'$.

The problem with the proposition 4.2 is that the optimal step is changed. Indeed, since the order of the modified finite difference method is changed, therefore, the optimal step is changed too. In this case, the proposition 3.11 can be applied to compute an approximate optimal step.
4.8 Taking into account bounds on parameters

The backward formula might be useful in some practical situations where the parameters are bounded. This might happen when this parameter represents a physical quantity which is physically bounded. For example, the real parameter $x$ might represent a fraction which is naturally in the interval $[0, 1]$.

Assume that some parameter $x$ is bounded in a given interval $[a, b]$, with $a, b \in \mathbb{R}$ and $a < b$. Assume that the step $h$ is given, may be by the formula $21$. If $b > a + h$, there is no problem at computing the numerical derivative with the forward formula

$$f'(a) \approx \frac{f(a + h) - f(a)}{h}.$$  \hfill (158)

If we want to compute the numerical derivative at $b$ with the forward formula

$$f'(b) \approx \frac{f(b + h) - f(b)}{h},$$  \hfill (159)

this leads to a problem, since $b + h \notin [a, b]$. In fact, any point $x$ in the interval $[b - h, b]$ leads to the problem. For such points, the backward formula may be used instead.

5 The derivative function

In this section, we present the derivative function. We present the main features of this function and show how to change the order of the finite difference method. We analyze of an orthogonal matrix may be used to change the directions of differentiation. Finally, we analyze the performances of derivative, in terms of function evaluations.

5.1 Overview

The derivative function computes the Jacobian and the Hessian matrix of a given function. We can use formulas of order 1, 2 or 4. Finally, the user can set the step used in the finite difference formula. In this section, we will analyse all these points.

The following is the complete calling sequence for the derivative function.

$$[ J , H ] = \text{derivative} \left( F , x , h , \text{order} , \text{H_form} , Q \right)$$

where the variables are

- $J$, the Jacobian vector,
- $H$, the Hessian matrix,
- $F$, the multivariate function,
- $x$, the current point,
- $\text{order}$, the order of the formula (1, 2 or 4),
- $\text{H_form}$, the Hessian matrix storage (‘default’, ‘blockmat’ or ‘hypermat’),
• \(Q\), a matrix used to scale the step.

Since we are concerned here by numerical issues, we will use the "blockmat" Hessian matrix storage.

The order 1, 2 and 4 formulas for the Jacobian matrix are implemented with formulas similar to the ones presented in figure 4, that is, the computations are based on forward 2 points (order 1), centered 2 points (order 2) and centered 4 points (order 4) formulas. The approximate optimal step \(h\) is computed depending on the formulas in order to minimize the total error.

The \texttt{derivative} function takes into account for multivariate functions, so that all points which have been detailed in section 4.2 can be applied here. In particular, the function uses modified versions 124, 125 and 126. Indeed, instead of using one step \(h_i\) for each direction \(i = 1, \ldots, n\), the same step \(h\) is used for all components.

### 5.2 Varying order to check accuracy

Since several accuracy are provided by the \texttt{derivative} function, it is easy and useful to check the accuracy of a specific numerical derivative. If the derivative varies only slightly with various formula orders, that implies that the user can be confident in its derivatives. Instead, if the numerical derivatives varies greatly with different formulas, that implies that the numerical derivative must be used with caution.

In the following Scilab script, we use various formulas to check the numerical derivative of the univariate quadratic function \(f(x) = x^2\).

```scilab
function y = myfunction3 (x)
  y = x^2;
endfunction
x = 1.0;
expected = 2.0;
for o = [1 2 4]
  fp = derivative (myfunction3,x, order = o);
  err = abs(fp-expected)/abs(expected);
  mprintf(" Order = %d, Relative error : %e\n",order,err)
end
```

The previous script produces the following output, where the relative error is printed.

- \texttt{Order = 1}, Relative error : \(7.450581 \times 10^{-09}\)
- \texttt{Order = 2}, Relative error : \(8.531620 \times 10^{-12}\)
- \texttt{Order = 4}, Relative error : \(0.000000 \times 10^{00}\)

Increasing the order produces increasing accuracy, as expected in such a simple case.

An advanced feature is provided by the \texttt{derivative} function, namely the transformation of the directions by an orthogonal matrix \(Q\). This is the topic of the following section.

### 5.3 Orthogonal matrix

In this section, we describe the mathematics behind the orthogonal \(n \times n\) matrix \(Q\), which is an optionnal input argument of the \texttt{derivative} function. An orthogonal
matrix is a square matrix satisfying \( Q^T = Q^{-1} \).

In order to simplify the discussion, let us assume that the function is a multivariate scalar function, i.e. \( f : \mathbb{R}^n \to \mathbb{R} \). Second, we want to produce a result which does not explicitly depend on the canonical vectors \( e_i \). The goal is to be able to compute directional derivatives in directions which are combinations of the axis vectors. Then, Taylor’s expansion in the direction \( Q e_i \) yields

\[
f(x + hQe_i) = f(x) + h g(x)^T Q e_i + O(h^2).
\]

This leads to

\[
g(x)^T Q e_i = \frac{f(x + hQe_i) - f(x)}{h}.
\]  

(161)

Recall that in the classical formula, the term \( g(x)^T e_i \) can be simplified into \( g_i(x) \). But now, the matrix \( Q \) has been inserted in between, so that the direction is indeed different. Let us denote by \( q_i \in \mathbb{R}^n \) the \( i \)-th column of the matrix \( Q \). Let us denote by \( d^T \in \mathbb{R}^n \) the row vector of function differences defined by

\[
d_i = \frac{f(x + hQe_i) - f(x)}{h},
\]

for \( i = 1, \ldots, n \). The equation 161 is transformed into \( g(x)^T q_i = d_i \), or, in matrix form,

\[
g(x)^T Q = d^T.
\]  

(163)

We right multiply the previous equation by \( Q^T \) and get

\[
g(x)^T QQ^T = d^T Q^T.
\]  

(164)

By the orthogonality property of \( Q \), this implies

\[
g(x)^T = d^T Q^T.
\]  

(165)

Finally, we transpose the previous equation and get

\[
g(x) = Q d.
\]  

(166)

The Hessian matrix can be computed based on the method which has been presented in the section 4.5. Hence, the computation of the Hessian matrix can also be modified to take into account the orthogonal matrix \( Q \).

Let us consider the case where the function \( f : \mathbb{R}^n \to \mathbb{R}^m \), where \( m \) is a positive integer. We want to compute the Jacobian \( m \times n \) matrix \( J \) defined by

\[
J = \begin{pmatrix}
\frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\
\vdots & & \vdots \\
\frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n}
\end{pmatrix}.
\]  

(167)

In this case, the finite differences are defining a column vector, so that we must consider the \( m \times n \) matrix \( D \) with entries

\[
D_{ij} = \frac{f_i(x + hQe_j) - f_i(x)}{h},
\]

for \( i = 1, \ldots, m \) and \( j = 1, \ldots, n \). The Jacobian matrix \( J \) is therefore computed from

\[
J = DQ^T.
\]  

(169)
<table>
<thead>
<tr>
<th>Degree</th>
<th>Order</th>
<th>Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobian</td>
<td>1</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>Jacobian</td>
<td>2</td>
<td>$2n$</td>
</tr>
<tr>
<td>Jacobian</td>
<td>4</td>
<td>$4n$</td>
</tr>
<tr>
<td>Hessian</td>
<td>1</td>
<td>$(n + 1)^2$</td>
</tr>
<tr>
<td>Hessian</td>
<td>2</td>
<td>$4n^2$</td>
</tr>
<tr>
<td>Hessian</td>
<td>4</td>
<td>$16n^2$</td>
</tr>
</tbody>
</table>

Figure 6: The number of function evaluations for the Jacobian and Hessian matrices.

5.4 Performance of finite differences

In this section, we analyse the number of function evaluations required by the computation of the Jacobian and Hessian matrices with the \texttt{derivative} function.

The number of function evaluations required to perform the computation depends on the dimension $n$ and the number of points in the formula. The table 6 summarizes the results.

The following list analyzes the number of function evaluations required to compute the gradient of the function depending on the dimension and the order of the formula.

- The order = 1 formula requires $n + 1$ function evaluations. Indeed, the function must be evaluated at $f(x)$ and $f(x + h e_i)$, for $i = 1, \ldots, n$.

- The order = 2 formula requires $2n$ function evaluations. Indeed, the function must be evaluated at $f(x - h e_i)$ and $f(x + h e_i)$, for $i = 1, \ldots, n$.

- The order = 4 formula requires $4n$ function evaluations. Indeed, the function must be evaluated at $f(x - h e_i)$, $f(x + h e_i)$, $f(x - 2h e_i)$ and $f(x + 2h e_i)$, for $i = 1, \ldots, n$.

Consider the quadratic function in $n = 10$ dimensions
\begin{equation}
  f(x) = \sum_{i=1,10} x_i^2.
\end{equation}

In the following Scilab script, we define the function and use a global variable to store the number of function evaluations required by the \texttt{derivative} function.

```plaintext
function y = myfunction3(x)
global nbfeval
nbfeval = nbfeval + 1
y = x.' * x;
endfunction
x = (1:10).';
for o = [1 2 4]
global nbfeval
nbfeval = 0;
J = derivative(myfunction3,x,order=o);
mprintf("Order = %d, Feval : %d\n",o,nbfeval)
end
```

44
The previous script produces the following output.

Order = 1, Feval : 11  
Order = 2, Feval : 20  
Order = 4, Feval : 40

In the following example, we consider a quadratic function in two dimensions. We define the `quadf` function, which computes the value of the function and plots the input points.

```plaintext
function f = quadf ( x )
    f = x(1)^2 + x(2)^2
    plot(x(1)-1,x(2)-1,"bo")
endfunction
```

The following `updateBounds` function updates the bounds of the given graphics handle `h`. This removes the labels of the graphics: if we keep them, very small numbers are printed, which is useless. We symmetrize the plot. We slightly increase the bounds, in order to make visible points which would otherwise be at the limit of the plot. Finally, we set the background of the points to blue, so that the points are clearly visible.

```plaintext
function updateBounds(h)
    hc = h.children
    hc.axes_visible=['off' 'off','off'];
    hc.data_bounds(1,:) = -hc.data_bounds(2,:);
    hc.data_bounds = 1.1*hc.data_bounds;
    for i = 1 : size(hc.children,"*")
        hc.children(i).children.mark_background=2
    end
endfunction
```

Figure 7: Points used for the computation of the Jacobian with finite differences and order 1, order 2 and order 4 formulas.
Figure 8: Points used in the computation of the Hessian with finite differences and order 1, order 2 and order 4 formulas. The points clustered in the middle are from the numerical Jacobian.

Then, we make several calls to the derivative function, which creates the plots which are presented in figures 7 and 8.

```markdown
// See pattern for Jacobian
h = scf();
J1 = derivative(quadf,x,order = 1);
updateBounds(h);
scf();
J1 = derivative(quadf,x,order = 2);
updateBounds(h);
scf();
J1 = derivative(quadf,x,order = 4);
updateBounds(h);

// See pattern for Hessian for order 2
h = scf();
[J1, H1] = derivative(quadf,x,order = 1);
updateBounds(h);
h = scf();
[J1, H1] = derivative(quadf,x,order = 2);
updateBounds(h);
h = scf();
[J1, H1] = derivative(quadf,x,order = 4);
updateBounds(h);
```

In the following example, we compute both the gradient and the Hessian matrix in the same case as previously.

```markdown
function y = myfunction3 (x)
global nbfeval
nbfeval = nbfeval + 1
y = x.' * x;
endfunction
```
x = (1:10).';
for o = [1 2 4]
    global nbfeval
    nbfeval = 0;
    [ J , H ] = derivative(myfunction3,x,order=o);
    mprintf("Order = %d, Feval : %d\n", o, nbfeval)
end

The previous script produces the following output. Notice that, since we compute both the gradient and the Hessian matrix, the number of function evaluations is the sum of the two, although, in practice, the cost of the Hessian matrix is the most important.

Order = 1, Feval : 132
Order = 2, Feval : 420
Order = 4, Feval : 1640

6 One more step

In this section, we analyse the behaviour of derivative when the point \( x \) is either large \( x \to \infty \), when \( x \) is small \( x \to 0 \) and when \( x = 0 \). We compare these results with the numdiff function, which does not use the same step strategy. As we are going to see, both commands performs the same when \( x \) is near 1, but performs very differently when \( x \) is large or small.

We have already explained the theory of the floating point implementation of the derivative function. Is it completely bulletproof? Not exactly.

See for example the following Scilab session, where one computes the numerical derivative of \( f(x) = x^2 \) for \( x = 10^{-100} \). The expected result is \( f'(x) = 2 \times 10^{-100} \).

```scilab
--> function y = myfunction (x)
    --> y = x*x;
--> endfunction
-->fp = derivative(myfunction,1.e-100,order=1)
fp =
    0.0000000149011611938477
-->fe=2.e-100
fe =
    2.000000000000000040-100
-->e = abs(fp-fe)/fe
   =
    7.450580596923828243D+91
```

The result does not have any significant digits.

The explanation is that the step is computed with \( h = \sqrt{\varepsilon M} \approx 10^{-8} \). Then \( f(x + h) = f(10^{-100} + 10^{-8}) \approx f(10^{-8}) = 10^{-16} \), because the term \( 10^{-100} \) is much smaller than \( 10^{-8} \). The result of the computation is therefore \( (f(x + h) - f(x))/h = (10^{-16} + 10^{-200})/10^{-8} \approx 10^{-8} \).

The additional experiment

```scilab
--> sqrt( %eps )
ans =
    0.0000000149011611938477
```
allows to check that the result of the computation simply is $\sqrt{\epsilon_M}$. That experiment shows that the `derivative` function uses a wrong default step $h$ when $x$ is very small.

To improve the accuracy of the computation, one can take control of the step $h$. A reasonable solution is to use $h = \sqrt{\epsilon_M|x|}$ so that the step is scaled depending on $x$. The following script illustrates than method, which produces results with 8 significant digits.

```matlab
-->h=sqrt(eps)*1.e-100;
-->fp = derivative(myfunction,1.e-100,order=1,h=h)
fp =
   2.000000013099139394-100
-->fe=2.e-100
fe =
   2.000000000000000040-100
-->e = abs(fp-fe)/fe
   0.000000065495696770794
```

But when $x$ is exactly zero, the scaling method cannot work, because it would produce the step $h = 0$, and therefore a division by zero exception. In that case, the default step provides a good accuracy.

Another function is available in Scilab to compute the numerical derivatives of a given function, that is `numdiff`. The `numdiff` function uses the step

$$h = \sqrt{\epsilon_M(1 + 10^{-3}|x|)}.$$  \hfill (171)

In the following paragraphs, we try to analyse why this formula has been chosen. As we are going to check experimentally, this step formula performs better than `derivative` when $x$ is large.

As we can see the following session, the behaviour is approximately the same when the value of $x$ is 1.

```matlab
-->fp = numdiff(myfunction,1.0)
fp =
   2.0000000189353417390237
-->fe=2.0
fe =
   2.
-->e = abs(fp-fe)/fe
   9.468D-09
```

The accuracy is slightly decreased with respect to the optimal value $7.450581e-009$ which was produced by `derivative`. But the number of significant digits is approximately the same, i.e. 9 digits.

The goal of this step is to produce good accuracy when the value of $x$ is large, where the `numdiff` function produces accurate results, while `derivative` performs poorly.
This step is a trade-off because it allows to keep a good accuracy with large values of $x$, but produces a slightly sub-optimal step size when $x$ is near 1. The behaviour near zero is the same, i.e. both commands produce wrong results when $x \to 0$ and $x \neq 0$.

7 Automatically computing the coefficients

In this section, we present the general method which leads to finite difference formulas. We show how this method enables to automatically compute the coefficients of finite difference formulas of arbitrary degree and order.

This section is based on a paper by Eberly [7].

7.1 The coefficients of finite difference formulas

In this section, we present the general method which leads to finite difference formulas.

**Proposition 7.1.** Assume that $f : \mathbb{R} \to \mathbb{R}$ is a continuously derivable function of one variable. Assume that $f$ is continuously differentiable. Assume that $i_{\min}, i_{\max}$ are two integers, such that $i_{\min} < i_{\max}$. Consider the finite difference formula

$$Df(x) = \frac{d!}{h^d} \sum_{i=i_{\min}}^{i_{\max}} C_i f(x + ih),$$

(172)

where $d$ is a positive integer, $x$ is a real number, $h$ is a real number and the coefficient $C_i$ is a real numbers for $i = i_{\min}, i_{\min} + 1, ..., i_{\max}$. Let us introduce

$$b_n = \sum_{i=i_{\min}}^{i_{\max}} i^n C_i,$$

(173)

for $n \geq 0$. Assume that

$$b_n = \begin{cases} 
0 & \text{if } n = 0, 1, ..., d - 1, \\
1 & \text{if } n = d, \\
0 & \text{if } n = d + 1, d + 2, ..., d + p - 1. 
\end{cases}$$

(174)

Therefore, the operator $Df$ is an order $p$ finite difference operator for $f^{(d)}(x)$ and:

$$f^{(d)}(x) = \frac{d!}{h^d} \sum_{i=i_{\min}}^{i_{\max}} C_i f(x + ih) + O(h^p).$$

(175)
Proof. By Taylor’s theorem 2.1, we have
\[
f(x + ih) = \sum_{n \geq 0} i^n \frac{h^n}{n!} f^{(n)}(x)
\]
(176)
\[
= i^n \sum_{n \geq 0} \frac{h^n}{n!} f^{(n)}(x),
\]
(177)
where \(i = i_{\min}, i_{\min} + 1, \ldots, i_{\max}\). Hence,
\[
\sum_{i=i_{\min}}^{i_{\max}} C_i f(x + ih) = \sum_{i=i_{\min}}^{i_{\max}} C_i i^n \sum_{n \geq 0} \frac{h^n}{n!} f^{(n)}(x)
\]
(178)
\[
= \sum_{n \geq 0} \left( \sum_{i=i_{\min}}^{i_{\max}} i^n C_i \right) \frac{h^n}{n!} f^{(n)}(x).
\]
(179)
We introduce the equation 173 into the previous equation and get
\[
\sum_{i=i_{\min}}^{i_{\max}} C_i f(x + ih) = \sum_{n \geq 0} b_n \frac{h^n}{n!} f^{(n)}(x).
\]
(180)
We can expand the sum for \(n \geq 0\) in several parts and analyze the consequences of the equation 174.

- The indices \(n = 0, 1, \ldots, d - 1\) make the function \(f\) and the derivatives \(f^{(1)}, f^{(2)}, \ldots, f^{(d-1)}\) appear. By choosing \(b_n = 0\) for \(n = 0, 1, \ldots, d - 1\), we make these terms disappear.

- The indice \(n = d\) make the expression \(b_d \frac{h^d}{d!} f^{(d)}(x)\) appear. The choice \(b_d = 1\) keeps this expression in the expansion.

- The indices \(n = d + 1, d + 2, \ldots, d + p - 1\) are associated to order \(n\) terms: these expressions may be written \(b_n \mathcal{O}(h^n)\) and must be cancelled in order to have an order \(p\) finite difference formula. The choice \(b_n = 0\) for \(n = d + 1, d + 2, \ldots, d + p - 1\) make these expressions disappear.

Hence, the equation 174 implies
\[
\sum_{i=i_{\min}}^{i_{\max}} C_i f(x + ih) = \frac{h^d}{d!} f^{(d)}(x) + \sum_{n \geq d+p} b_n \frac{h^n}{n!} f^{(n)}(x)
\]
(181)
\[
= \frac{h^d}{d!} f^{(d)}(x) + \mathcal{O}(h^{d+p}).
\]
(182)
We multiply the previous equation by \(\frac{d!}{h^d}\) and immediately get the equation 175, which concludes the proof. 
\(\Box\)
7.2 Automatically computing the coefficients

In this section, we consider practical uses of the previous proposition. We consider forward, backward and centered finite difference formulas of arbitrary degree and order. We present how the coefficients of classical formulas are the solution of simple linear systems of equations. We present how the method also leads to less classical formulas.

Let us consider the proposition 172 in the situation where we are searching for a finite difference formula. In this case, we are given the degree $d$ and the order $p$, and we are searching for the $n_c = i_{\text{max}} - i_{\text{min}} + 1$ unknowns $C_i$. The equation 174 define $d + p$ linear equations: solving these equations leads to the coefficients of the finite difference formula.

This is a change with the ordinary "manual" search for finite difference formulas, where the search for the coefficients requires clever combinations of various Taylor’s expansions. Here, there is no need to be clever: the method only requires to solve a linear system of equations.

We are interested in the coefficients of forward, backward and centered finite difference formulas in the context of the proposition 7.1. We are interested in the values of $i_{\text{min}}$, $i_{\text{max}}$ and the number of unknowns $n_c$.

- A forward finite difference formula is associated with $i_{\text{min}} = 0$, $i_{\text{max}} = d + p - 1$ and $n_c = d + p$ unknowns.

- A forward finite difference formula is associated with $i_{\text{min}} = -(d + p - 1)$, $i_{\text{max}} = 0$ and $n_c = d + p$ unknowns.

- A centered finite difference formula is associated with $i_{\text{max}} = [(d + p - 1)/2]$, $i_{\text{min}} = -i_{\text{max}}$ and $n_c = d + p - 1$ if $d + p$ is even, and $n_c = d + p$ if $d + p$ is odd.

The table 9 presents the values of the parameters for classical finite difference formulas.

For example, consider the centered formula 111, and order $p = 4$ formula for $f^{(2)}$, i.e. $d = 2$. We have $(d + p - 1)/2 = 2.5$, so that $i_{\text{min}} = -2$, $i_{\text{max}} = 2$ and $n_c = 5$. The unknowns is the vector $C = (C_{-2}, C_{-1}, C_0, C_1, C_2)^T$. The equation 173 can be

<table>
<thead>
<tr>
<th>Formula</th>
<th>$d$</th>
<th>$p$</th>
<th>Type</th>
<th>$i_{\text{min}}$</th>
<th>$i_{\text{max}}$</th>
<th>$n_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>1</td>
<td>1</td>
<td>Forward</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>1</td>
<td>Backward</td>
<td>-1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>1</td>
<td>Centered</td>
<td>-1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>106</td>
<td>1</td>
<td>2</td>
<td>Forward</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>107</td>
<td>1</td>
<td>2</td>
<td>Backward</td>
<td>-2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>108</td>
<td>1</td>
<td>4</td>
<td>Centered</td>
<td>-2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>109</td>
<td>2</td>
<td>1</td>
<td>Forward</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
<td>2</td>
<td>Centered</td>
<td>-1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
written as the linear system of equations

\[
\begin{pmatrix}
1 & 1 & 1 & 1 & 1 \\
-2 & -1 & 0 & 1 & 2 \\
4 & 1 & 0 & 1 & 4 \\
-8 & -1 & 0 & 1 & 8 \\
16 & 1 & 0 & 1 & 16
\end{pmatrix}
C =
\begin{pmatrix}
0 \\
0 \\
1 \\
0 \\
0
\end{pmatrix}
\]

(183)

The solution of the previous system of equation is

\[C = \frac{1}{24}(-1, 16, -30, 16, -1)^T.\]  

(184)

The equation 172 then implies

\[
f^{(2)} = \frac{2!}{24h^2}(-f(x - 2h) + 16f(x - h) - 30f(x) + 16f(x + h) - f(x + 2h)) + \mathcal{O}(h^4),
\]

(185)

which simplifies to 111, as expected.

### 7.3 Computing the coefficients in Scilab

In this section, we present scripts which compute the coefficients of finite difference formulas of arbitrary order.

The following `derivativeIndices` computes `imin` and `imax`, given the degree `d`, the order `p` and the type of formula `form`.

```scilab
function [imin,imax] = derivativeIndices(d,p,form)
    select form
        case "forward"
            if ( p > 1 & modulo(p,2)==1 ) then
                error(msprintf("The order p must be even."))
            end
            imin = 0
            imax = d+p-1
        case "backward"
            if ( p > 1 & modulo(p,2)==1 ) then
                error(msprintf("The order p must be even."))
            end
            imin = -(d+p-1)
            imax = 0
        case "centered"
            if ( modulo(p,2)==1 ) then
                error(msprintf("The order p must be even."))
            end
            imax = floor((d+p-1)/2)
            imin = -imax
        else
            error(msprintf("Unknown form \%s",form))
    end
endfunction
```

In the following session, we experiment the `derivativeIndices` function with various values of `d`, `p` and `form`.  

52
The following `derivativeMatrixNaive` function computes the matrix which is associated with the equation 173. Its entries are \( i^n \) for \( i = 1, 2, \ldots, n_c \) and \( n = 1, 2, \ldots, n_c \).

```scilab
function A = derivativeMatrixNaive(d,p, form)
    [imin,imax] = derivativeIndices(d,p, form)
    indices = imin:imax
    nc=size(indices,"*")
    A = zeros(nc,nc)
    for irow = 1 : nc
        n = irow-1
        for jcol = 1 : nc
            i = indices(jcol)
            A(irow,jcol) = i^n
        end
    end
endfunction
```

The previous function requires two nested loops, which may be inefficient in Scilab. The following `derivativeMatrix` function uses vectorization to perform the same algorithm. More precisely, the function uses the elementwise power operator.

```scilab
function A = derivativeMatrix(d,p, form)
    [imin,imax] = derivativeIndices(d,p, form)
    indices = imin:imax
    nc=size(indices,"*")
    A = zeros(nc,nc)
    x = indices(ones(nc,1),:)
    y = ((1:nc)-1)'
    z = y(:,ones(nc,1))
    A = x.^z
endfunction
```

In the following session, we compute the matrix associated with \( d = 2 \) and \( p = 4 \), i.e. we compute the same matrix as in the equation 183.

```scilab
-->A = derivativeMatrix(2,4,"centered")
A =
    1.   1.   1.   1.   1.
  -2.  -1.   0.   1.   2.
   4.   1.   0.   1.   4.
 -8.  -1.   0.   1.   8.
 16.   1.   0.   1.  16.
```

53
Figure 10: Coefficients of various centered finite difference formulas.

The following `derivativeTemplate` function solves the linear system of equations 173 and computes the coefficients C.

```plaintext
function C = derivativeTemplate(d,p,form)
    A = derivativeMatrix(d,p,form)
    nc= size(A,"r")
    b = zeros(nc,1)
    b(d+1) = 1
    C = A\b
endfunction
```

In the following session, we compute the coefficients associated with \(d = 2\) and \(p = 4\), i.e. we compute the coefficients 184.

```plaintext
-->C = derivativeTemplate(2,4,"centered")
C =
   - 0.0416667
   0.6666667
  - 1.25
   0.6666667
   - 0.0416667
```

The previous function can produce the coefficients for any degree \(d\) and order \(p\). For example, the table 10 presents the coefficients of various centered finite difference formulas, as presented in [22].

8 Notes and references

A reference for numerical derivatives is [1], chapter 25. "Numerical Interpolation, Differentiation and Integration" (p. 875).


On the specific usage of numerical derivatives in optimization, the Gill, Murray and Wright book [8] in the section 4.6.2 "Non-derivative Quasi-Newton methods",
"Notes and Selected Bibliography for section 4.6" and the section 8.6 "Computing finite differences".

The book by Nocedal and Wright [11] presents the evaluation of sparse Jacobians with numerical derivatives in section 7.1 "Finite-difference derivative approximations".

The book by Kelley [12] presents in the section 2.3, "Computing a Finite Difference Jacobian" a scaling method so that the step is scaled with respect to $x$. The scaling is applied only when $x$ is large, and not when $x$ is small. Kelley’s book [12] presents a method based on complex arithmetic is presented and is the following.

According to an article by Shampine [17], the method is due to Squire and Trapp [19]. Assume that $f$ can be evaluated in complex arithmetic. Then Taylor’s expansion of $f$ shows that

$$f(x + ih) = f(x) + ihf'(x) - \frac{h^2}{2} f''(x) - i \frac{h^3}{6} f'''(x) + \frac{h^4}{24} f''''(x) + \ldots$$

(187)

We now take the imaginary part of both sides of the previous equation and dividing by $h$ yields

$$\frac{\text{Im}(f(x + ih))}{h} = f'(x) - \frac{h^2}{6} f'''(x) + \ldots$$

(188)

Therefore,

$$f'(x) = \frac{\text{Im}(f(x + ih))}{h} + O(h^2).$$

(189)

The previous formula is therefore of order 2. This method is not subject to subtractive cancellation.

The Numerical Recipes [14] contains several details regarding numerical derivatives in chapter 5.7 "Numerical Derivatives". The authors present a method to compute the step $h$ so that the rounding error for the sum $x + h$ is minimum. This is performed by the following algorithm, which implies a temporary variable $t$

$$t \leftarrow x + h$$
$$h \leftarrow t - h$$

In Stepleman and Winarsky’s [20] (1979), an algorithm is designed to compute an automatic scaling for the numerical derivative. This method is an improvement of the method by Dumontet and Vignes [6] (1977). The examples tested by [6] are analysed in [20], which shows that 11 digits are accurate with an average number of function evaluations from 8 to 11, instead of an average 20 in [6].

In the section 7, we presented a method to compute the coefficient of finite difference formulas. This section is based on a paper by Eberly [7], but this approach is also presented by Nicholas Maxwell in [13]. According to Maxwell, his source of inspiration is [2], where finite difference schemes for the Laplacian are derived in a similar way.

9 Exercises

**Exercise 9.1 (Using lists with derivative)** The goal of this exercise is to use lists in a practical situation where we want to compute the numerical derivative of a function. Consider the function
$f : \mathbb{R}^3 \to \mathbb{R}$ defined by

$$f(x) = p_1 x_1^2 + p_2 x_2^2 + p_3 + p_4 (x_3 - 1)^2,$$

where $x \in \mathbb{R}^3$ and $p \in \mathbb{R}^4$ is a given vector of parameters. In this exercise, we consider the vector $p = (4, 5, 6, 7)^T$. The gradient of this function is

$$g(x) = \begin{pmatrix}
2p_1 x_1 \\
2p_2 x_2 \\
2p_4 (x_3 - 1)
\end{pmatrix}.
$$

(191)

In this exercise, we want to find the minimum $x^*$ of this function with the `optim` function. The following script defines the function `cost` which allows to compute the value of this function given the point $x$ and the floating point integer `ind`. This function returns the function value $f$ and the gradient $g$.

```plaintext
function [f,g,ind] = cost ( x , ind )
    f = 4 * x(1)^2 + 5 * x(2)^2 + 6 + 7*(x(3)-1)^2
    g = [
            8 * x(1)
            10 * x(2)
        14 * (x(3)-1)
    ]
endfunction
```

In the following session, we set the initial point and check that the function can be computed for this initial point. Then we use the `optim` function and compute the optimum point $x_{opt}$ and the corresponding function value $f_{opt}$.

```plaintext
-->x0 = [1 2 3] ;
-->[f,g,ind] = cost ( x0 , 1 )
    ind = 1.
    g =  8.
         20.
         28.
    f =  58.
--->[fopt,xopt] = optim ( cost , x0 )
    xopt = - 3.677 -186
           8.165 -202
           1.
    fopt =  6.
```

Use a list and find the answer to the two following questions.

1. We would like to check the gradient of the cost function. Use a list and the `derivative` function to check the gradient of the function `cost`.

2. It would be more clear if the parameters $(p_1, p_2, p_3, p_4)$ were explicit input arguments of the cost function. In this case, the cost function would be the following.

```plaintext
function [f,g,ind] = cost2 ( x , p1 , p2 , p3 , p4 , ind )
    f = p1 * x(1)^2 + p2 * x(2)^2 + p3 + p4 * (x(3)-1)^2
    g = [
            2*p1 * x(1)
            2*p2 * x(2)
        2*p4 * (x(3)-1)
    ]
endfunction
```
Use a list and the `optim` function to compute the minimum of the `cost2` function.

10 Acknowledgments

I would like to thank Bruno Pinçon who made many highly valuable numerical comments on this document.

References


Index

derivative, 6
numdiff, 46

Centered (2 points) finite difference for $f'$, 17
Centered (3 points) finite difference for $f''$, 22
Centered (4 points) finite difference for $f'$, 19

Forward (2 points) finite difference for $f'$, 5

gradient, 27
Jacobian, 28

order, 5

Taylor formula, 4
Truncation error, 5