

# Checking Gradients

## 1. Introduction

We wish to minimize a function  $F : \mathbb{R}^n \mapsto \mathbb{R}$ . Many efficient algorithms demand that the user provides a subprogram, which – for a given  $\mathbf{x}$  – returns both the function value  $F(\mathbf{x})$  and the gradient  $\mathbf{g}(\mathbf{x}) \in \mathbb{R}^n$ , defined by

$$\mathbf{g} = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{bmatrix}. \quad (1.1)$$

There are special algorithms for cases where  $F(\mathbf{x})$  is derived from some norm of a vector valued function  $\mathbf{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$ , more specifically,

$$\begin{aligned} F(\mathbf{x}) &= \|\mathbf{f}(\mathbf{x})\|_1 = |f_1(\mathbf{x})| + \dots + |f_m(\mathbf{x})| \\ F(\mathbf{x}) &= \|\mathbf{f}(\mathbf{x})\|_2^2 = f_1(\mathbf{x})^2 + \dots + f_m(\mathbf{x})^2 && \text{(Least squares)} \\ F(\mathbf{x}) &= \|\mathbf{f}(\mathbf{x})\|_\infty = \max\{|f_1(\mathbf{x})|, \dots, |f_m(\mathbf{x})|\} && \text{(Minimax)} \end{aligned}$$

In these cases the user should provide  $F(\mathbf{x})$  and the Jacobian matrix  $\mathbf{J}(\mathbf{x}) \in \mathbb{R}^{m \times n}$ , defined by

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}, \quad (1.2)$$

i.e., the  $i$ th row in  $\mathbf{J}$  is the gradient of  $f_i$ , the  $i$ th component of  $\mathbf{f}$ .

For an efficient performance of the optimization algorithm the function and the gradients must be implemented without errors. It is not possible to check the correctness of the implementation of  $F$  (or  $\mathbf{f}$ ), but we can check the corresponding gradient (or Jacobian). This is described in Section 2, while Sections 3 and 5 are a User's guides

to a MATLAB and a Fortran77 implementation, and Sections 4 and 6 give examples.

## 2. Theoretical Background

The checking is done by difference approximation. First consider a scalar function: For given  $\mathbf{x}$  and steplength  $h$  we compute

$$\left. \begin{aligned} D_j^F &= (F(\mathbf{x} + h\mathbf{e}_j) - F(\mathbf{x})) / h \\ D_j^B &= (F(\mathbf{x}) - F(\mathbf{x} - \frac{1}{2}h\mathbf{e}_j)) / (\frac{1}{2}h) \\ D_j^E &= (D_j^F + 2D_j^B) / 3 \end{aligned} \right\}, \quad j = 1, \dots, n, \quad (2.1)$$

where  $\mathbf{e}_j$  is the  $j$ th unit vector (the  $j$ th column of  $\mathbf{I}$ ), and the superscripts stand for Forward, Backward and Extrapolated difference approximation, respectively.

We assume that  $F$  is three times continuously differentiable with respect to each of its arguments. Then a Taylor expansion from  $\mathbf{x}$  shows that

$$\begin{aligned} F(\mathbf{x}) + \eta\mathbf{e}_j &= F(\mathbf{x}) + \eta \frac{\partial F}{\partial x_j}(\mathbf{x}) + \frac{1}{2}\eta^2 \frac{\partial^2 F}{\partial x_j^2}(\mathbf{x}) + \mathcal{O}(\eta^3) \\ &= F(\mathbf{x}) + \eta g_j(\mathbf{x}) + \eta^2 S_j(\mathbf{x}) + \mathcal{O}(\eta^3) . \end{aligned} \quad (2.2)$$

Inserting this in (2.1) we see that

$$\begin{aligned} D_j^F &= g_j + hS_j + \mathcal{O}(h^2) , \\ D_j^B &= g_j - \frac{1}{2}hS_j + \mathcal{O}(h^2) , \\ D_j^E &= g_j + \mathcal{O}(h^2) . \end{aligned} \quad (2.3)$$

Now, let  $G_j$  denote the  $j$ th component of the gradient as returned from the user's subprogram, and let

$$G_j = g_j - \psi_j , \quad (2.4)$$

where  $\psi_j = \psi_j(\mathbf{x})$  is zero if the implementation is correct. Inserting this in (2.3) we get

$$\begin{aligned} \delta_j^F &\equiv D_j^F - G_j = \psi_j + hS_j + \mathcal{O}(h^2), \\ \delta_j^B &\equiv D_j^B - G_j = \psi_j - \frac{1}{2}hS_j + \mathcal{O}(h^2), \\ \delta_j^E &\equiv D_j^E - G_j = \psi_j + \mathcal{O}(h^2). \end{aligned} \quad (2.5)$$

If  $\psi_j = 0$ ,  $S_j = \frac{1}{2}\partial^2 F/\partial x_j^2 \neq 0$  and  $h$  is so small that the last term in each right-hand side of (2.5) can be neglected, then we can expect  $\delta_j^B \simeq -\frac{1}{2}\delta_j^F$  and  $\delta_j^E$  to be of the order of magnitude  $(\delta_j^F)^2$ . Also, if the approximation is recomputed with  $h$  replaced by  $\theta h$ , where  $0 < \theta < 1$ , then both  $\delta_j^F$  and  $\delta_j^B$  are reduced by a factor  $\theta$ , while  $\delta_j^E$  is reduced by a factor  $\theta^2$ .

If  $\psi_j \neq 0$  and  $h$  is sufficiently small, then the error will be recognized by  $\delta_j^F \simeq \delta_j^B \simeq \delta_j^E \simeq \psi_j$ .

The computed values are affected by rounding errors. Especially, instead of  $F(\mathbf{z})$  we get  $\text{fl}(F(\mathbf{z})) = F(\mathbf{z}) + \varepsilon$ . The best that we can hope for is that  $|\varepsilon| \leq u \cdot |F(\mathbf{x})|$ , where  $u$  is the ‘‘unit round-off’’. (In MATLAB or in Fortran with REAL\*8,  $u = 2^{-53} \simeq 10^{-16}$  on most computers). This has the consequence that for the computed difference approximations (2.5) should be replaced by

$$\begin{aligned} |\delta_j^F| &\leq |\psi_j + hS_j| + A_j h^{-1} + \mathcal{O}(u) + \mathcal{O}(h^2), \\ |\delta_j^B| &\leq |\psi_j - \frac{1}{2}hS_j| + 2A_j h^{-1} + \mathcal{O}(u) + \mathcal{O}(h^2), \\ |\delta_j^E| &\leq |\psi_j| + B_j h^{-1} + \mathcal{O}(u) + \mathcal{O}(h^2), \end{aligned} \quad (2.6)$$

where  $A_j$  and  $B_j$  are positive constants, that depend on  $F$  and  $\mathbf{x}$ , but not on  $h$ . In the case of correct implementation of the gradient (2.6) shows that for large  $h$  the errors are dominated by truncation error, while effects of rounding errors dominate if  $h$  is too small. Assuming that  $|S_j|$  and  $A_j$  are of the same order of magnitude, the smallest error with the forward and backward difference approximations is obtained with  $h \simeq \sqrt{u}|\mathbf{x}|$ , while  $h \simeq \sqrt[3]{u}|\mathbf{x}|$  minimizes  $|\delta_j^E|$ .

Now, consider problems where  $F(\mathbf{x})$  is some norm of a vector function  $\mathbf{f}(\mathbf{x})$ . In this case it is relevant to check the implementation of the Jacobian  $\mathbf{J}(\mathbf{x})$ , (1.2). The  $i$ th row in  $\mathbf{J}$  is the gradient of  $f_i$ , the  $i$ th component of  $\mathbf{f}$ , and a straightforward generalization of (2.1) is

$$\left. \begin{aligned} D_{ij}^F &= (f_i(\mathbf{x} + h\mathbf{e}_j) - f_i(\mathbf{x}))/h \\ D_{ij}^B &= (f_i(\mathbf{x}) - f_i(\mathbf{x} - \frac{1}{2}h\mathbf{e}_j))/(\frac{1}{2}h) \\ D_{ij}^E &= (D_{ij}^F + 2D_{ij}^B)/3 \end{aligned} \right\}, \quad \begin{cases} i=1, \dots, m \\ j=1, \dots, n \end{cases}, \quad (2.7)$$

leading to

$$\begin{aligned} \delta_{ij}^F &\equiv D_{ij}^F - J(i, j) = \psi_{ij} + hS_{ij} + \mathcal{O}(h^2), \\ \delta_{ij}^B &\equiv D_{ij}^{(2)} - J(i, j) = \psi_{ij} - \frac{1}{2}hS_{ij} + \mathcal{O}(h^2), \\ \delta_{ij}^E &\equiv D_{ij}^{(3)} - J(i, j) = \psi_{ij} + \mathcal{O}(h^2), \end{aligned} \quad (2.8)$$

where  $J(i, j)$  is the  $(i, j)$ th element in the implemented Jacobian,  $\psi_{ij}$  is its error and  $S_{ij} = \frac{1}{2}\partial^2 f_i/\partial x_j^2$ .

### 3. User’s Guide to checkjacob

The gradient checker is implemented in MATLAB as `checkjacob.m`. A typical call is

```
[maxJ, error, index] = checkjacob(fun, fpar, x, h)
```

#### Input Parameters

**fun** : String with the name of the function that defines **f** and **J**. The function must have the declaration  

```
function [f, J] = fun(x, fpar)
```

**f** and **J** should be the vector function and its Jacobian evaluated at **x**.

**fpar** : Parameters of the function. May be empty.

**x** : The point where we wish to check the Jacobian.

**h** : Steplength used in difference approximation to **J(x)**.

#### Output Parameters

**maxJ** : Largest absolute value of the elements in the Jacobian.

**error** : Vector with three elements, cf. (2.8), **error(1)** =  $\max_{ij} |\delta_{ij}^F|$   
**error(2)** =  $\max_{ij} |\delta_{ij}^B|$     **error(3)** =  $\max_{ij} |\delta_{ij}^E|$

**index** : 3\*2 array, with **index(k, :)** giving the position in **J**, where **error(k)** occurs.

## 4. Examples

First, consider the modified *Rosenbrock* function [2, Example 3.11],

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 10(x_2 - x_1^2) \\ 1 - x_1 \\ \lambda \end{bmatrix}, \quad \mathbf{J}(\mathbf{x}) = \begin{bmatrix} -20x_1 & 10 \\ -1 & 0 \\ 0 & 0 \end{bmatrix}. \quad (4.1)$$

It can be implemented as

```
function [f,J] = fun1(x,lam)
f = [10*(x(2) - x(1)^2); 1-x(1); lam];
J = [-20*x(1) 10; -1 0; 0 0];
```

The command

```
[mJ err indx] = checkjacobi('fun1', 10, [-1.2 1], 1e-5)
```

gives the results

```
mJ = 24   err = -1.0000e-04   indx = 1 1
      5.0000e-05             1 1
      5.9211e-11             1 2
```

This is in perfect agreement with the theory:  $f_2(\mathbf{x})$  is linear and  $f_3(\mathbf{x})$  is a constant, so all the difference approximations are exact. This is also the case for  $(\mathbf{J})_{1,2}$ , while

$$D_{1,1}^F = \frac{10(x_2 - (x_1+h)^2) - 10(x_2 - x_1^2)}{h} = -20x_1 - 10h,$$

leading to  $\delta_{1,1}^F = -10h$ . Similarly we find  $\delta_{1,1}^B = 5h$  and  $\delta_{1,1}^E = 0$ . The value **err**(3) is the effect of rounding errors.

Next consider the scalar problem ( $n=2$ )

$$F(\mathbf{x}) = \cos x_1 + e^{2x_2}, \quad \mathbf{g}(\mathbf{x}) = \begin{bmatrix} -\sin x_1 \\ 2e^{2x_2} \end{bmatrix}, \quad (4.2)$$

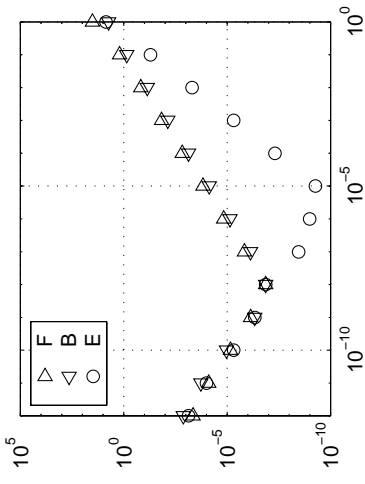
implemented as follows (note the sign error in  $g_1$  and that the gradient must be returned as a row vector: the only row in  $\mathbf{J}$ ),

```
function [f,g] = fun2(x,p)
e = exp(2*x(2));
f = cos(x(1)) + e;   g = [-sin(x(1)) 2*e];
```

The commands

```
h = 1e-3; [mJ err indx] = checkjacobi('fun2', [], [1 1], h)
gives the results
      mJ = 1.4778e+01   err = -1.6832e+00   indx = 1 1
      -1.6828e+00             1 1
      -1.6829e+00             1 1
```

The three **err**-values are almost identical, indicating an error in the implementation, and the second column in **indx** shows that  $G_1$  is the erroneous element. (To be more precise: the element with largest  $|\psi_j|$ , cf. (2.4)). After correcting this element we called `checkjacobi('fun2', [], [1 1], h)` for  $h = 1, 10^{-1}, \dots, 10^{-12}$ , and found the following results for the absolute values of **err**.



In this double logarithmic plot the difference between **|err**(1) and **|err**(2) is  $\log \frac{1}{2}$ . For large values of  $h$  both of these errors are proportional with  $h$ , but for  $h \lesssim 10^{-8}$  the errors grow – rounding errors dominate. The errors of the extrapolated approximations are proportional with  $h^2$  for  $h \gtrsim 10^{-5}$  and with  $h^{-1}$  for smaller steps. This behaviour agrees with the discussion in Section 2.

Finally, consider the *Branin* function

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 1 - 2x_2 + \frac{1}{20} \sin 4\pi x_2 - x_1 \\ x_2 - \frac{1}{2} \sin 2\pi x_1 \end{bmatrix}, \quad (4.3)$$

implemented as

```
function [f,J] = branin(x,par)
% Function
f = [1-2*x(2)+.05*sin(4*pi*x(2))-x(1)
     x(2)-.5*sin(2*pi*x(1))];
if nargin > 1
    % Jacobian
    J = [-1 (-2 + .2*pi*cos(4*pi*x(2)))
         -pi*cos(2*pi*x(1)) 1];
end
```

The command

```
[mJ err indx] = checkjacobi('branin', [], [1 1], 1e-5)
```

gives the results

```
mJ = 3.1416e+00  err = 2.0427e-09  indx = 2  1
      5.6612e-10
      1.0583e-09  2  1
                  2  1
```

These results do not agree with our expectations, but can easily be explained: `indx` shows that the worst error is in the approximation to

$$\frac{\partial f_2}{\partial x_1}(\mathbf{x}) = -\pi \cos(2\pi x_1).$$

The corresponding  $S_{ij}$  is

$$S_{2,1} = \frac{1}{2} \frac{\partial^2 f_2}{\partial x_1^2}(\mathbf{x}) = \pi^2 \sin(2\pi x_1),$$

which is zero for  $x_1 = 1$ . Therefore, both  $\delta_{2,1}^F$  and  $\delta_{2,1}^B$  are  $\mathcal{O}(h^2)$  – verified by `err(2)  $\simeq \frac{1}{4}$ err(1) – and the extrapolated approximation is not much better.`

Changing the argument so that at least one component of  $\mathbf{x}$  is non integer gives the expected performance:

```
[mJ err indx] = checkjacobi('branin', [], [1 1.1], 1e-5)
```

gives

```
mJ = 3.1416e+00  err = -3.7547e-05  indx = 1  2
      1.8773e-05
      1.0620e-09  1  2
                  2  1
```

## 5. User's Guide to CHKJAC

The gradient checker is implemented in Fortran77 as `chkjac.f`. A typical call is

```
CALL CHKJAC(FDF,N,M,X,DF,F,DX,W)
```

### Input Parameters

**FDF** Name of the subroutine that defines **f** and **J**. The function must have the declaration

```
SUBROUTINE FDF(N,M,X,DF,F)
  DOUBLE PRECISION X(N),DF(M,N),F(M)
```

**F** and **DF** should be the vector function and its Jacobian evaluated at **X**.

The name of this subroutine can be chosen freely by the user. It must appear in an **EXTERNAL** statement in the calling program.

**N** **INTEGER**. Must be set to  $m$ , the number of unknowns. It must be positive. Is not changed.

**M** **INTEGER**. Must be set to  $m$ , the number of functions. It must be positive. Is not changed.

**X** **DOUBLE PRECISION ARRAY** with  $N$  elements.

The point where we wish to check the Jacobian. Is not changed.

**DX** **DOUBLE PRECISION**. Steplength  $h$ .

### Output Parameters

- DF** DOUBLE PRECISION ARRAY with  $M$  rows and  $N$  columns. Jacobian  $\mathbf{J}(\mathbf{X})$ .
- F** DOUBLE PRECISION ARRAY with  $M$  elements. Vector function  $\mathbf{f}(\mathbf{X})$ .
- W** DOUBLE PRECISION ARRAY with at least  $10+N+M(3+N)$  elements. Work space. Results of the check is returned in the first 10 elements as follows,
- $W(1)$  Maximum element in  $|\mathbf{DF}|$ .
  - $W(2), W(5), W(6)$  Maximum error in forward difference approximation and its row and column index.
  - $W(3), W(7), W(8)$  Maximum error in backward difference approximation and its row and column index.
  - $W(4), W(9), W(10)$  Maximum error in extrapolated approximation and its row and column index.

### Use of other Subprograms

CHKJAC calls the BLAS [1] Level 1 subroutines and functions `daxpy`, `dcopy`, `dscal` and `idamax`. Copies of these were obtained from <http://www.netlib.org/blas/blas.tgz> and are included in the file `chkjac.f`. At line 34 you can find instructions about how to modify the file if BLAS is available.

## 6. Examples II

A Fortran77 program for testing the modified Rosenbrock function (4.1) with can be implemented as follows

```

PROGRAM TCHKJ
*****
* Test CHKJAC on Modified Rosenbrock. 00.09.21
*****
IMPLICIT NONE
INTEGER M, N
DOUBLE PRECISION X(2), F(3), DF(3,2), W(27), FPAR, H
EXTERNAL FUN1
COMMON FPAR
DATA M, N, H, X / 3, 2, 1D-5, -1.2D0, 1D0 /

      FPAR = 10D0
      CALL CHKJAC(FUN1,N,M,X,DF,F,H,W)
      WRITE(7,'(A,1P1D11.4)') 'Max|DF| = ',W(1)
      WRITE(7,'(A,2X,A8,3H : ,1P1D11.4,11H at i,j =,2I4)')
+      'Max difference.', 'Forward', W(2),INT(W(5)),INT(W(6))
+      WRITE(7,'(17X,A8,3H : ,1P1D11.4,11H at i,j =,2I4)')
+      'Backward', W(3),INT(W(7)),INT(W(8))
+      WRITE(7,'(17X,A8,3H : ,1P1D11.4,11H at i,j =,2I4)')
+      'Extrap.', W(4),INT(W(9)),INT(W(10))
      STOP
      END

      SUBROUTINE FUN1(N,M,X,DF,F)
      INTEGER N,M
      DOUBLE PRECISION X(N),DF(M,N),F(M), FPAR
      COMMON FPAR
      F(1) = 10D0*(X(2) - X(1)**2)
      F(2) = 1D0 - X(1)
      F(3) = FPAR
      DF(1,1) = -20D0*X(1)
      DF(1,2) = 10D0
      DF(2,1) = -1D0
      DF(2,2) = 0D0
      DF(3,1) = 0D0
      DF(3,2) = 0D0
      RETURN
      END

```

In perfect agreement with the MATLAB version we get the results

```

Max|DF| = 2.4000E+01
Max difference, Forward : -1.0000E-04 at i,j = 1 1
Backward : 5.0000E-05 at i,j = 1 1
Extrap. : 5.9211E-11 at i,j = 1 2

*****
* Test CHKJAC on simple scalar problem. 00.09.21
*****
PROGRAM TCHKJ2
      M, N
      DOUBLE PRECISION X(2), F(1), DF(1,2), W(17), H
      EXTERNAL FUN2
      DATA M, N, H, X / 1, 2, 1D-3, 2*1D0 /

      CALL CHKJAC(FUN2, N, M, X, DF, F, H, W)
      WRITE(7, '(A,1P1D11.4)') 'Max|DF| = ', W(1)
      WRITE(7, '(A,2X,A8,3H : ,1P1D11.4,11H at i,j =,2I4)')
+ 'Max difference, ', 'Forward', W(2), INT(W(6)), INT(W(6))
+ 'Backward', W(3), INT(W(7)), INT(W(8))
+ 'Extrap.', W(4), INT(W(9)), INT(W(10))
      STOP
      END

```

c

```

SUBROUTINE FUN2(N,M,X,DF,F)
  Scalar function with gradient error
  INTEGER N,M
  DOUBLE PRECISION X(N),DF(M,N),F(M),E
  INTRINSIC COS,EXP,SIN
  E = EXP(2D0 * X(2))
  F(1) = COS(X(1)) + E
  DF(1,1) = SIN(X(1))
  DF(1,2) = 2D0 * E
  RETURN
  END

```

We get the same results as in Section 4, and after having corrected the sign in the expression for DF(1,1) we can compute errors as in the figure on page 6.

Finally, Branin's problem (4.3) can be implemented as

```

SUBROUTINE BRANIN(N,M,X,DF,F)
  INTEGER N,M
  DOUBLE PRECISION X(N),DF(M,N),F(M),PI2
  INTRINSIC ATAN,COS,SIN
  ... PI2 = 2*pi
  PI2 = 8D0*ATAN(1D0)
  F(1) = 1D0 - 2D0*X(2) + .05D0*SIN(2D0*PI2*X(2)) - X(1)
  F(2) = X(2) - .5D0*SIN(PI2*X(1))
  DF(1,1) = -1D0
  DF(1,2) = .1D0*PI2*COS(2D0*PI2*X(2)) - 2D0
  DF(2,1) = -.5D0*PI2*COS(PI2*X(1))
  DF(2,2) = 1D0
  RETURN
  END

```

c

Again the results show perfect agreement with the MATLAB results.

## References

- [1] J. Dongarra, C.B. Moler, J.R. Bunch and G.W. Stewart. (1988): *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft. **14**, 1-17.
- [2] K. Madsen, H.B. Nielsen and O. Tingleff (1999): *Methods for Non-Linear Least Squares Problems*. Lecture note, IMM, DTU.