```
# grad case 1 and 2 are special cases of jacobian, with a scalar rather than
# vector valued function. Case 3 differs only because of the interpretation
# that the vector result is a scalar function applied to each argument, and the
# thus the result has the same length as the argument.
# The code of grad could be consolidated to use jacobian.
# There is also some duplication in genD.

###########################################################################

# functions for gradient calculation

###########################################################################

grad <- function (func, x, method="Richardson", side=NULL,
method.args=list(), ...) UseMethod("grad")

grad.default <- function(func, x, method="Richardson", side=NULL,
method.args=list(), ...){
# modified by Paul Gilbert from code by Xingqiao Liu.
# case 1/ scalar arg, scalar result (case 2/ or 3/ code should work)
# case 2/ vector arg, scalar result (same as special case jacobian)
# case 3/ vector arg, vector result (of same length, really 1/ applied multiple times
))
f <- func(x, ...)
n <- length(x) #number of variables in argument

if (is.null(side)) side <- rep(NA, n)
else {
if(n != length(side))
stop("Non-NULL argument 'side' should have the same length as x")
if(any(1 != abs(side[!is.na(side)])))
stop("Non-NULL argument 'side' should have values NA, +1, or -1.")
}

case1or3 <- n == length(f)

if((1 != length(f)) & !case1or3)
stop("grad assumes a scalar valued function.")

if(method=="simple"){
# very simple numerical approximation
args <- list(eps=1e-4) # default
args[names(method.args)] <- method.args

side[is.na(side)] <- 1
eps <- rep(args$eps, n) * side
```

```
if(case1or3) return((func(x+eps, ...)-f)/eps)

# now case 2
df <- rep(NA,n)
for (i in 1:n) {
dx <- x
dx[i] <- dx[i] + eps[i]
df[i] <- (func(dx, ...) - f)/eps[i]
}
return(df)
}
else if(method=="complex"){ # Complex step gradient
if (any(!is.na(side))) stop("method 'complex' does not support non-NULL argument 'sid
e'.")
eps <- .Machine$double.eps
v <- try(func(x + eps * 1i, ...))
if(inherits(v, "try-error"))
stop("function does not accept complex argument as required by method 'complex'.")
if(!is.complex(v))
stop("function does not return a complex value as required by method 'complex'.")

if(case1or3) return(Im(v)/eps)
# now case 2
h0 <- rep(0, n)
g <- rep(NA, n)
for (i in 1:n) {
h0[i] <- eps * 1i
g[i] <- Im(func(x+h0, ...))/eps
h0[i] <- 0
}
return(g)
}
else if(method=="Richardson"){
args <- list(eps=1e-4, d=0.0001, zero.tol=sqrt(.Machine$double.eps/7e-7), r=4, v=2, s
how.details=FALSE) # default
args[names(method.args)] <- method.args
d <- args$d
r <- args$r
v <- args$v
show.details <- args$show.details
a <- matrix(NA, r, n)
#b <- matrix(NA, (r - 1), n)

# first order derivatives are stored in the matrix a[k,i],
# where the indexing variables k for rows(1 to r), i for columns (1 to n),
# r is the number of iterations, and n is the number of variables.
```

```
h <- abs(d*x) + args$eps * (abs(x) < args$zero.tol)
pna <- (side == 1) & !is.na(side) # double these on plus side
mna <- (side == -1) & !is.na(side) # double these on minus side

for(k in 1:r) { # successively reduce h
ph <- mh <- h
ph[pna] <- 2 * ph[pna]
ph[mna] <- 0
mh[mna] <- 2 * mh[mna]
mh[pna] <- 0

if(case1or3) a[k,] <- (func(x + ph, ...) - func(x - mh, ...))/(2*h)
else for(i in 1:n) {
if((k != 1) && (abs(a[(k-1),i]) < 1e-20)) a[k,i] <- 0 #some func are unstable near ze
ro
else a[k,i] <- (func(x + ph*(i==seq(n)), ...) -
func(x - mh*(i==seq(n)), ...))/(2*h[i])
}
if (any(is.na(a[k,]))) stop("function returns NA at ", h," distance from x.")
h <- h/v # Reduced h by 1/v.
}

if(show.details) {
cat("\n","first order approximations", "\n")
print(a, 12)
}

#-----------------------------------------------------------------------
# 1 Applying Richardson Extrapolation to improve the accuracy of
# the first and second order derivatives. The algorithm as follows:
#
# -- For each column of the derivative matrix a,
# say, A1, A2, ..., Ar, by Richardson Extrapolation, to calculate a
# new sequence of approximations B1, B2, ..., Br used the formula
#
# B(i) =( A(i+1)*4^m - A(i) ) / (4^m - 1) , i=1,2,...,r-m
#
# N.B. This formula assumes v=2.

# -- Initially m is taken as 1 and then the process is repeated
# restarting with the latest improved values and increasing the
# value of m by one each until m equals r-1

# 2 Display the improved derivatives for each
# m from 1 to r-1 if the argument show.details=T.

# 3 Return the final improved derivative vector.
```

```
#-------------------------------------------------------------------------

for(m in 1:(r - 1)) {
    a <- (a[2:(r+1-m),,drop=FALSE]*(4^m)-a[1:(r-m),,drop=FALSE])/(4^m-1)
    if(show.details & m!=(r-1) ) {
        cat("\n","Richarson improvement group No. ", m, "\n")
        print(a[1:(r-m),,drop=FALSE], 12)
    }
}
return(c(a))
} else stop("indicated method ", method, "not supported.")
}



jacobian <- function (func, x, method="Richardson", side=NULL,
method.args=list(), ...) UseMethod("jacobian")

jacobian.default <- function(func, x, method="Richardson", side=NULL,
method.args=list(), ...){
f <- func(x, ...)
n <- length(x) #number of variables.

if (is.null(side)) side <- rep(NA, n)
else {
    if(n != length(side))
        stop("Non-NULL argument 'side' should have the same length as x")
    if(any(1 != abs(side[!is.na(side)])))
        stop("Non-NULL argument 'side' should have values NA, +1, or -1.")
}

if(method=="simple"){
    # very simple numerical approximation
    args <- list(eps=1e-4) # default
    args[names(method.args)] <- method.args

    side[is.na(side)] <- 1
    eps <- rep(args$eps, n) * side

    df <-matrix(NA, length(f), n)
    for (i in 1:n) {
        dx <- x
        dx[i] <- dx[i] + eps[i]
        df[,i] <- (func(dx, ...) - f)/eps[i]
    }
    return(df)
}
```

```
else if(method=="complex"){ # Complex step gradient
    if (any(!is.na(side))) stop("method 'complex' does not support non-NULL argument
'side'.")
    # Complex step Jacobian
    eps <- .Machine$double.eps
    h0 <- rep(0, n)
    h0[1] <- eps * 1i
    v <- try(func(x+h0, ...))
    if(inherits(v, "try-error"))
        stop("function does not accept complex argument as required by method 'comple
x'.")
    if(!is.complex(v))
        stop("function does not return a complex value as required by method 'complex
'.")

    h0[1] <- 0
    jac <- matrix(NA, length(v), n)
    jac[, 1] <- Im(v)/eps
    if (n == 1) return(jac)
    for (i in 2:n) {
        h0[i] <- eps * 1i
        jac[, i] <- Im(func(x+h0, ...))/eps
        h0[i] <- 0
    }
    return(jac)
}
else if(method=="Richardson"){
    args <- list(eps=1e-4, d=0.0001, zero.tol=sqrt(.Machine$double.eps/7e-7),
    r=4, v=2, show.details=FALSE) # default
    args[names(method.args)] <- method.args
    d <- args$d
    r <- args$r
    v <- args$v
    a <- array(NA, c(length(f),r, n) )

    h <- abs(d*x) + args$eps * (abs(x) < args$zero.tol)
    pna <- (side == 1) & !is.na(side) # double these on plus side
    mna <- (side == -1) & !is.na(side) # double these on minus side

    for(k in 1:r) { # successively reduce h
        ph <- mh <- h
        ph[pna] <- 2 * ph[pna]
        ph[mna] <- 0
        mh[mna] <- 2 * mh[mna]
        mh[pna] <- 0

        for(i in 1:n) {
```

```
            a[,k,i] <- (func(x + ph*(i==seq(n)), ...) -
            func(x - mh*(i==seq(n)), ...))/(2*h[i])
            #if((k != 1)) a[,(abs(a[,(k-1),i]) < 1e-20)] <- 0 #some func are unstable
near zero
        }
        h <- h/v # Reduced h by 1/v.
    }

    for(m in 1:(r - 1)) {
        a <- (a[,2:(r+1-m),,drop=FALSE]*(4^m)-a[,1:(r-m),,drop=FALSE])/(4^m-1)
    }
    # drop second dim of a, which is now 1 (but not other dim's even if they are 1
    return(array(a, dim(a)[c(1,3)]))
} else stop("indicated method ", method, "not supported.")
}
```

```
# Define the function
f <- function(x) {
  return(x^2)
}

# Compute the gradient at x=3 using Richardson method
result <- grad(f, 3, method="Richardson")

# Print the result
print(result)
```

```
## [1] 6
```

```
# Generic parallel lapply based on ctmmweb/R/5_parallel.R
############################################################
# If cores=1, vanilla lapply
# If UNIX & cores>1, mclapply
# If windows & fast, vanilla lapply
# If windows & !fast, parLapplyLB


# detect relevant core number given fast boolean
detectCores <- function(...,fast=TRUE)
{
  if(fast && .Platform$OS.type=="windows") { return(1) } # Windows cannot fork
  else { return(parallel::detectCores(logical=FALSE,...)) }
}
```

```
# resolve number of cores to use given user input
# NULL uses all cores
# non-positive values reserves that many cores
resolveCores <- function(cores=1,fast=TRUE)
{
  if(is.null(cores) || is.na(cores)) { cores <- detectCores(fast=fast) }
  else if(cores<1) { cores <-  max(1,detectCores(fast=fast) + cores) }
  # Windows can't fork
  if(fast && .Platform$OS.type=="windows") { cores <- 1 }

  return(cores)
}


# smart parallel lapply
plapply <- function(X,FUN,...,cores=1,fast=TRUE)
{
  WINDOWS <- (.Platform$OS.type=="windows")
  cores <- resolveCores(cores,fast=fast)
  cores <- min(cores,length(X)) # cap cores
  cores <- max(1,cores)

  if(cores==1 || (fast && WINDOWS)) { return(lapply(X,FUN,...)) }
  else if(!WINDOWS) { return(parallel::mclapply(X,FUN,...,mc.cores=cores)) }

  ### Windows parallel code below ###
  win_init = expression({requireNamespace("ctmm",quietly=TRUE)})

  cl <- parallel::makeCluster(cores,outfile="")
  # have to export parameter too because it's not available in remote
  parallel::clusterExport(cl,c("win_init"),envir=environment())
  parallel::clusterEvalQ(cl,eval(win_init))
  RESULT <- parallel::parLapplyLB(cl,X,FUN)
  parallel::stopCluster(cl)

  return(RESULT)
}
```

```
parallel_grad <- function(func, x, method="Richardson", side=NULL,
                          method.args=list(), ..., cores=4, fast=TRUE){
    f <- func(x, ...)
    n <- length(x) #number of variables in argument

    if (is.null(side)) side <- rep(NA, n)
    else {
        if(n != length(side))
```

```
            stop("Non-NULL argument 'side' should have the same length as x")
        if(any(1 != abs(side[!is.na(side)])))
            stop("Non-NULL argument 'side' should have values NA, +1, or -1.")
    }

    case1or3 <- n == length(f)

    if((1 != length(f)) & !case1or3)
        stop("grad assumes a scalar valued function.")

    else if(method=="Richardson"){
        args <- list(eps=1e-4, d=0.0001, zero.tol=sqrt(.Machine$double.eps/7e-7), r=4
, v=2, show.details=FALSE)
        args[names(method.args)] <- method.args
        d <- args$d
        r <- args$r
        v <- args$v
        show.details <- args$show.details
        a <- matrix(NA, r, n)

        h <- abs(d*x) + args$eps * (abs(x) < args$zero.tol)
        pna <- (side == 1) & !is.na(side)
        mna <- (side == -1) & !is.na(side)

        param_list <- list()
        for(k in 1:r) {
            for(i in 1:n) {
                param_list <- c(param_list, list(list(k=k, i=i)))
            }
        }

        results <- plapply(param_list, function(params) {
            k <- params$k
            i <- params$i
            ph <- mh <- h
            ph[pna] <- 2 * ph[pna]
            ph[mna] <- 0
            mh[mna] <- 2 * mh[mna]
            mh[pna] <- 0

            if(case1or3) return(list(k=k, i=i, result=(func(x + ph, ...) - func(x - m
h, ...))/(2*h)))
            else {
                if((k != 1) && !is.na(a[(k-1),i]) && (abs(a[(k-1),i]) < 1e-20)) retur
n(list(k=k, i=i, result=0))
                else return(list(k=k, i=i, result=(func(x + ph*(i==seq(n)), ...) - fu
nc(x - mh*(i==seq(n)), ...))/(2*h[i])))
```

```
            }
        }, cores=cores)

        for(res in results) {
            a[res$k, res$i] <- res$result
        }

        if(show.details) {
            cat("\n", "first order approximations", "\n")
            print(a, 12)
        }

        for(m in 1:(r - 1)) {
            a <- (a[2:(r+1-m),,drop=FALSE]*(4^m)-a[1:(r-m),,drop=FALSE])/(4^m-1)
            if(show.details & m!=(r-1) ) {
                cat("\n", "Richardson improvement group No. ", m, "\n")
                print(a[1:(r-m),,drop=FALSE], 12)
            }
        }

        return(c(a))
    }  else stop("indicated method ", method, "not supported.")
}
```

```
# Test the grad_parallel function
test_func <- function(x) {
  sum((x^2))
}

x0 <- c(1, 2, 3, 4, 5)
print(parallel_grad(test_func, x0, method="Richardson"))
```

```
## [1]  2  4  6  8 10
```

```
# Compute the gradient at x=3 using Richardson method
result <- grad(test_func, x0, method="Richardson")

# Print the result
print(result)
```

```
## [1]  2  4  6  8 10
```

```
#install.packages("microbenchmark")
library(microbenchmark)
```

```
# Test the grad_parallel function
test_func <- function(x) {
  sum(exp(x)*(x^5+x^4+x^3))
}

x0 <- c(1, 2, 3, 4, 5)

benchmark_results <- microbenchmark(
  grad_result = grad(test_func, x0, method="Richardson"),
  parallel_grad_result_1_core = parallel_grad(test_func, x0, method="Richardson", cor
es=1),
  parallel_grad_result_2_cores = parallel_grad(test_func, x0, method="Richardson", co
res=2),
  parallel_grad_result_4_cores = parallel_grad(test_func, x0, method="Richardson", co
res=4),
  times = 100
)
```

```
## Warning in microbenchmark(grad_result = grad(test_func, x0, method =
## "Richardson"), : less accurate nanosecond times to avoid potential integer
## overflows
```

```
print(benchmark_results)
```

```
## Unit: microseconds
##                           expr      min        lq      mean    median       uq
##                    grad_result  236.529  268.4065  729.8918  901.0365 1077.747
##   parallel_grad_result_1_core  312.625  336.4460  813.8795  548.6210 1260.053
##  parallel_grad_result_2_cores 2442.001 2692.2240 3169.4923 3328.6055 3507.550
##  parallel_grad_result_4_cores 3343.714 3747.2360 4584.3203 4221.9135 4496.675
##         max neval
##    2723.384   100
##    4215.579   100
##    4939.926   100
##   42432.130   100
```

```
computeDistances <- function(x) {
  N <- 100
  vectors <- matrix(runif(N * length(x)), ncol=length(x))

  distSum <- 0

  for(i in 1:(N-1)) {
    for(j in (i+1):N) {
      distSum <- distSum + sum((vectors[i,] - vectors[j,])^2)^0.5
    }
  }

  return(distSum)
}

# Initial values
x0 <- c(3, 2)

benchmark_results <- microbenchmark(
  grad_result = grad(computeDistances, x0, method="Richardson"),
  parallel_grad_result_1_core = parallel_grad(computeDistances, x0, method="Richardso
n", cores=1),
  parallel_grad_result_2_cores = parallel_grad(computeDistances, x0, method="Richards
on", cores=2),
  parallel_grad_result_4_cores = parallel_grad(computeDistances, x0, method="Richards
on", cores=4),
  times = 5
)

print(benchmark_results)
```

```
## Unit: milliseconds
##                          expr      min       lq     mean   median       uq
##                   grad_result 81.06401 81.97089 86.84258 89.18259 89.59562
##   parallel_grad_result_1_core 80.66463 81.47368 84.87190 81.50730 88.48522
##  parallel_grad_result_2_cores 63.83097 65.10907 66.48220 65.29000 65.29681
##  parallel_grad_result_4_cores 41.27273 41.50553 43.38458 43.29100 43.61486
##       max neval
##  92.39981     5
##  92.22868     5
##  72.88414     5
##  47.23877     5
```

```
x0 <- c(1, 2)

intensive_function <- function(x) {
  sum_val = 0
  for (i in 1:1000) {
    for (j in 1:1000) {
      sum_val <- sum_val + sin(x[1] * i) * cos(x[2] * j)
    }
  }
  return(sum_val)
}

# Benchmark the grad and parallel_grad functions
benchmark_results <- microbenchmark(
  grad_result = grad(intensive_function, x0, method="Richardson"),
  parallel_grad_result_1_core = parallel_grad(intensive_function, x0, method="Richard
son", cores=1),
  parallel_grad_result_2_cores = parallel_grad(intensive_function, x0, method="Richar
dson", cores=2),
  parallel_grad_result_4_cores = parallel_grad(intensive_function, x0, method="Richar
dson", cores=4),
  times = 5
)

print(benchmark_results)
```

```
## Unit: milliseconds
##                          expr      min       lq      mean    median        uq
##                   grad_result 2158.577 2159.889 2168.0697 2162.7022 2164.3951
##   parallel_grad_result_1_core 2145.522 2146.790 2149.1527 2150.0891 2150.6300
##  parallel_grad_result_2_cores 1140.481 1141.036 1146.8520 1145.7106 1146.9693
##  parallel_grad_result_4_cores  637.433  639.941  695.9555  717.8108  733.4123
##        max neval
##  2194.7854     5
##  2152.7325     5
##  1160.0621     5
##   751.1804     5
```