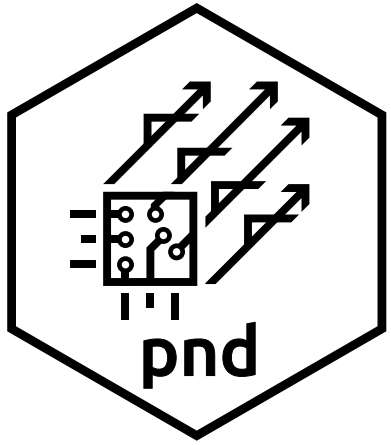
Make a difference: fast and accurate numerical derivatives with **pnd**



Based on the working paper:
Kostyrka, A. V. (2025). What are you doing, step size:
A survey of step-size selection methods for numerical derivatives

---

Andreï V. Kostyrka

uni.lu | UNIVERSITÉ DU LUXEMBOURG
Department of Economics and Management (DEM)

11èmes Rencontres R
Université de Mons
20 mai 2025

Presentation structure

1. Motivation and use cases

2. Approximations of analytical derivatives

3. Step-size selection algorithms

4. Showcasing pnd

Motivation and use cases

Contribution

1. I wrote an R package – **pnd** – for fast, **p**arallelised **n**umerical **d**ifferentiation
   - First open-source parallel Jacobians, Hessians and higher-order-accurate gradients in R
   - I implemented 6 algorithms for step-size selection and benchmarked their performance
   - You will see this benchmark
2. I am currently working on 3 papers on the topic: a survey and two algorithmic ones
   - Working paper: Kostyrka, A. V. Step size selection in numerical differences using a regression kink. *Department of Economics and Management discussion paper 2025-09*, University of Luxembourg. `https://hdl.handle.net/10993/64958`

Motivation

- Researchers rely on optimisers, algorithms, black boxes etc., and the end result depends on the solver quality
- Most popular modern optimisation techniques use numerical gradients for minimisation or maximisation

However, most software implementations yield **inaccurate** and **slow** numerical derivatives.

Consequences: inexact solutions, negative variances, invalid statistical inference etc.

Example from a financial application

AR(1)-GARCH(1, 1) model for NASDAQ log-returns, 1990–1994:

$$r_t = \mu + \rho r_{t-1} + \sigma_t U_t, \quad \sigma_t^2 = \omega + \alpha U_{t-1}^2 + \beta \sigma_{t-1}^2$$

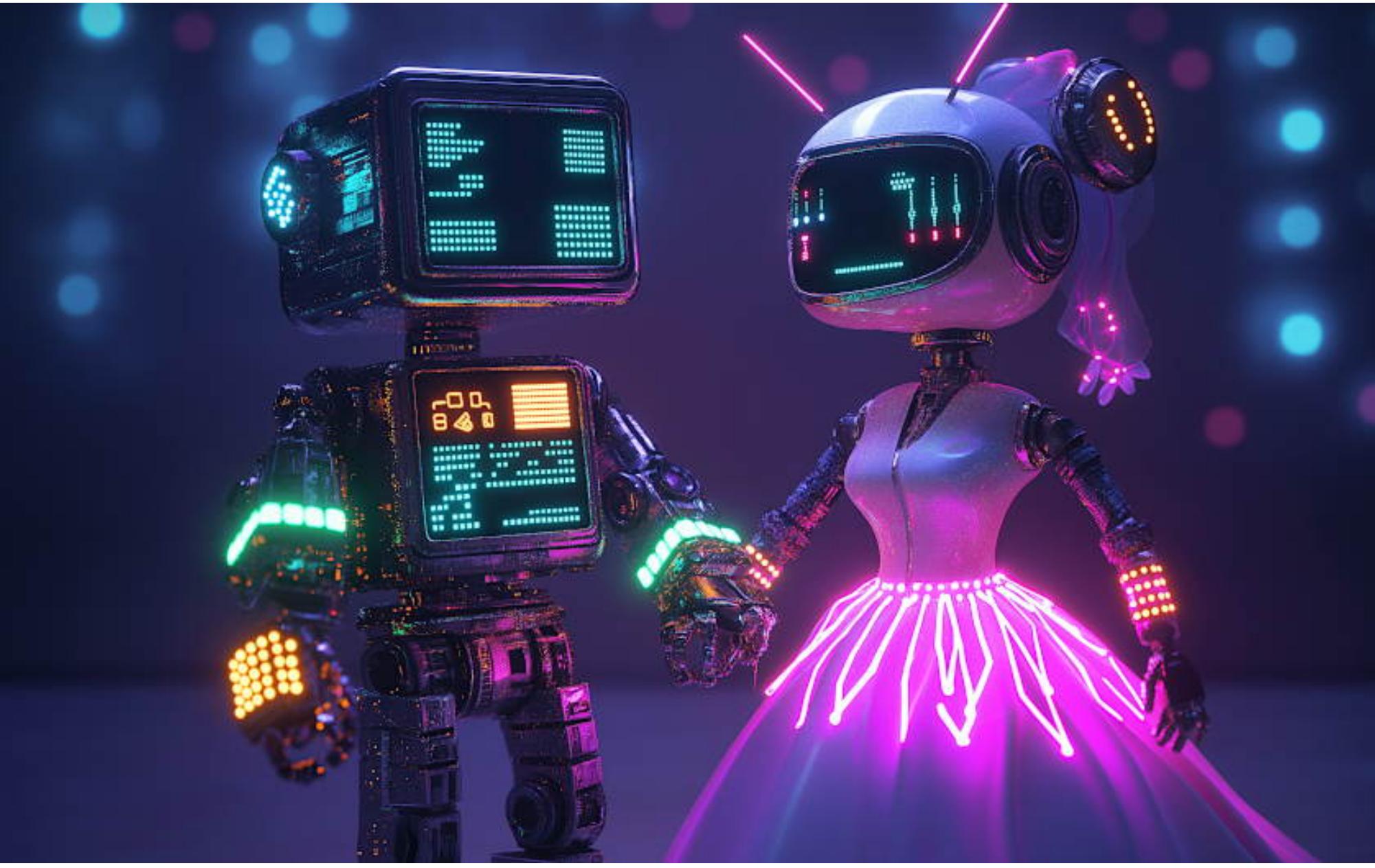
Coefficient	Est.	t-stat `rugarch`	t-stat `fGarch`
μ	0.0007	2.34	2.31
ρ	0.24	7.77	7.73
$\omega \times 10^3$	0.0098	NaN or 65 (default / fallback)	3.09
α	0.13	11.1	4.27
β	0.73	39.6	10.9

NaN due to negative variance!

Existing literature / software

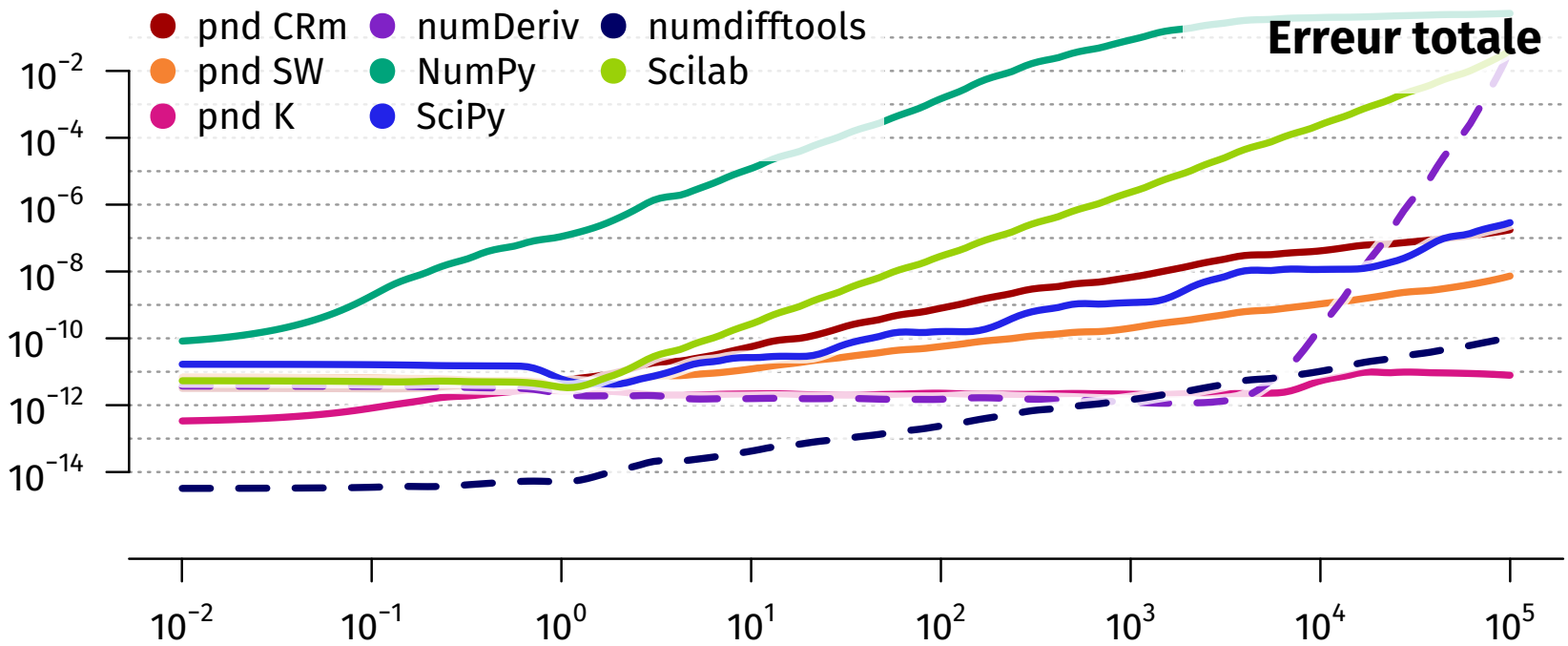
- Gilbert & Varadhan (2019). **numDeriv**: Accurate Numerical Derivatives.
  `cran.r-project.org/package=numDeriv`
  - Non-parallel version without vignettes or derivations
- Gerber & Furrer (2019). **optimParallel**: An R Package Providing a Parallel Version of the L-BFGS-B Optimization Method. *The R Journal* 11 (1).
  `cran.r-project.org/package=optimParallel`
  - Limited to the built-in `optim(method = "L-BFGS-B")`
- Algorithms for numerical derivatives from the 1970s have remained dormant… until now

Marrying numDeriv + optimParallel functionality



Selling point of pnd

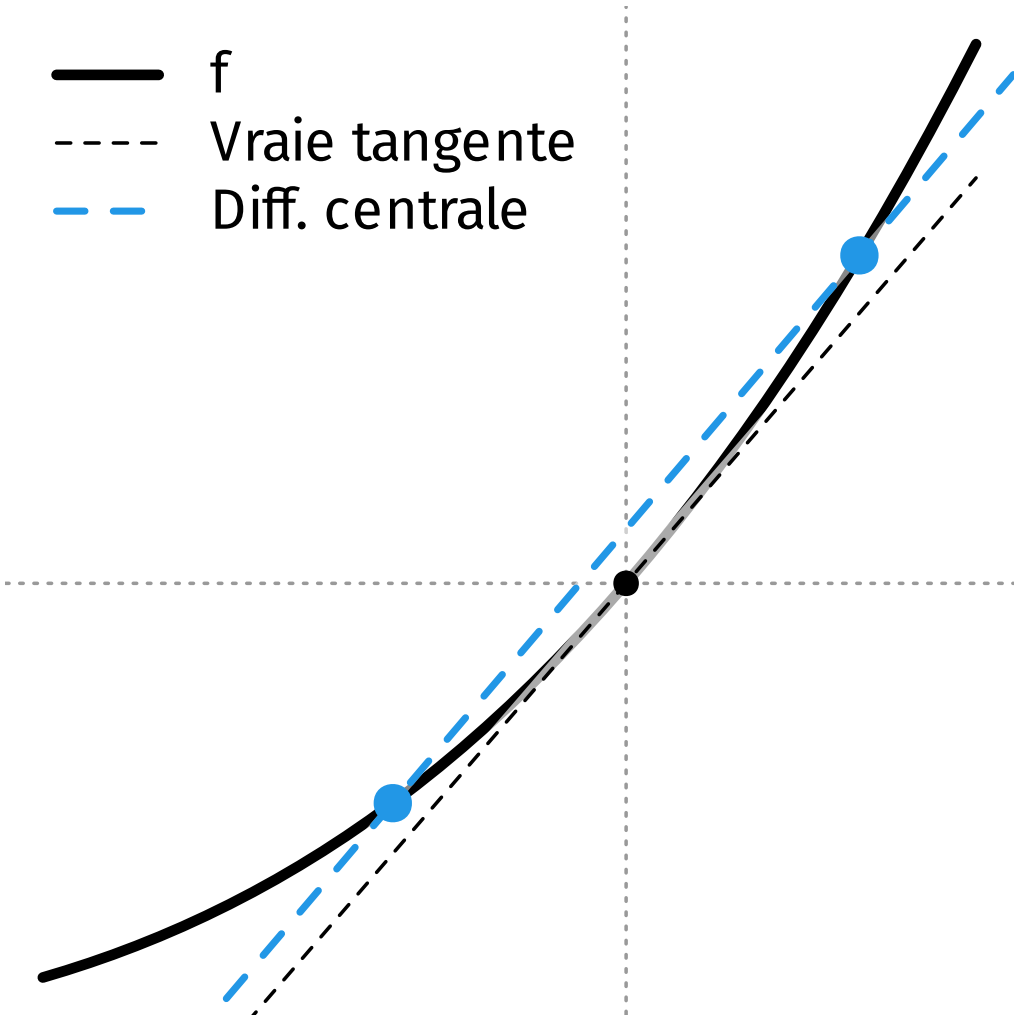
Compare the software: numerical derivative error for $f(x) = \sin x$ on an exponentially spaced grid between 0.01 and 10 000.



Solid: 2 evaluations, dashed: >2 evaluations (incomparable).

Approximations of analytical derivatives

Derivative estimation via central differences



- $f(x) = x^3, x_0 = 1$
- $f'(x_0) = 3$
- Step size $h = 0.2$
- $f'_{CD}(x_0, 0.2) = 3.04$
  Error $\approx 1.3\%$

Higher-order accuracy of first derivatives

Better accuracy is achievable with more function evaluations.

Carefully choose the coefficients to eliminate the undesirable terms:

$$f' = \underbrace{\frac{-f(x-h) + f(x+h)}{2h}}_{f'_{\text{CD},2}} + O(h^2)$$

$$f' = \underbrace{\frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h}}_{f'_{\text{CD},4}} + O(h^4)$$

- `pnd::fdCoef()` computes stencils and weights for arbitrary derivative orders and accuracy orders
- These 4 evaluations can and should be parallelised

Efficient parallelisation of gradients

Example: $\nabla f(x_{3\times 1})$, evaluation grid $\{x \pm h, x \pm 2h\}$ for 4^{th}-order accuracy. Total: 12 evaluations.

	$w_1 = \frac{1}{12}$	$w_2 = -\frac{8}{12}$	$w_3 = \frac{8}{12}$	$w_4 = -\frac{1}{12}$
$x^{(1)}$	$f(x - 2h_1)$	$f(x - h_1)$	$f(x + h_1)$	$f(x + 2h_1)$
$x^{(2)}$	$f(x - 2h_2)$	$f(x - h_2)$	$f(x + h_2)$	$f(x + 2h_2)$
$x^{(3)}$	$f(x - 2h_3)$	$f(x - h_3)$	$f(x + h_3)$	$f(x + 2h_3)$

- Create a list of length 12 containing $x + b_j h_i$
- Apply f **in parallel** to the list items, assemble $\left\{\{f(x + b_j h_i)\}_{i=1}^{3}\right\}_{j=1}^{4}$ in a matrix
- Compute weighted row sums

Step-size selection algorithms

Total error in numerical derivatives

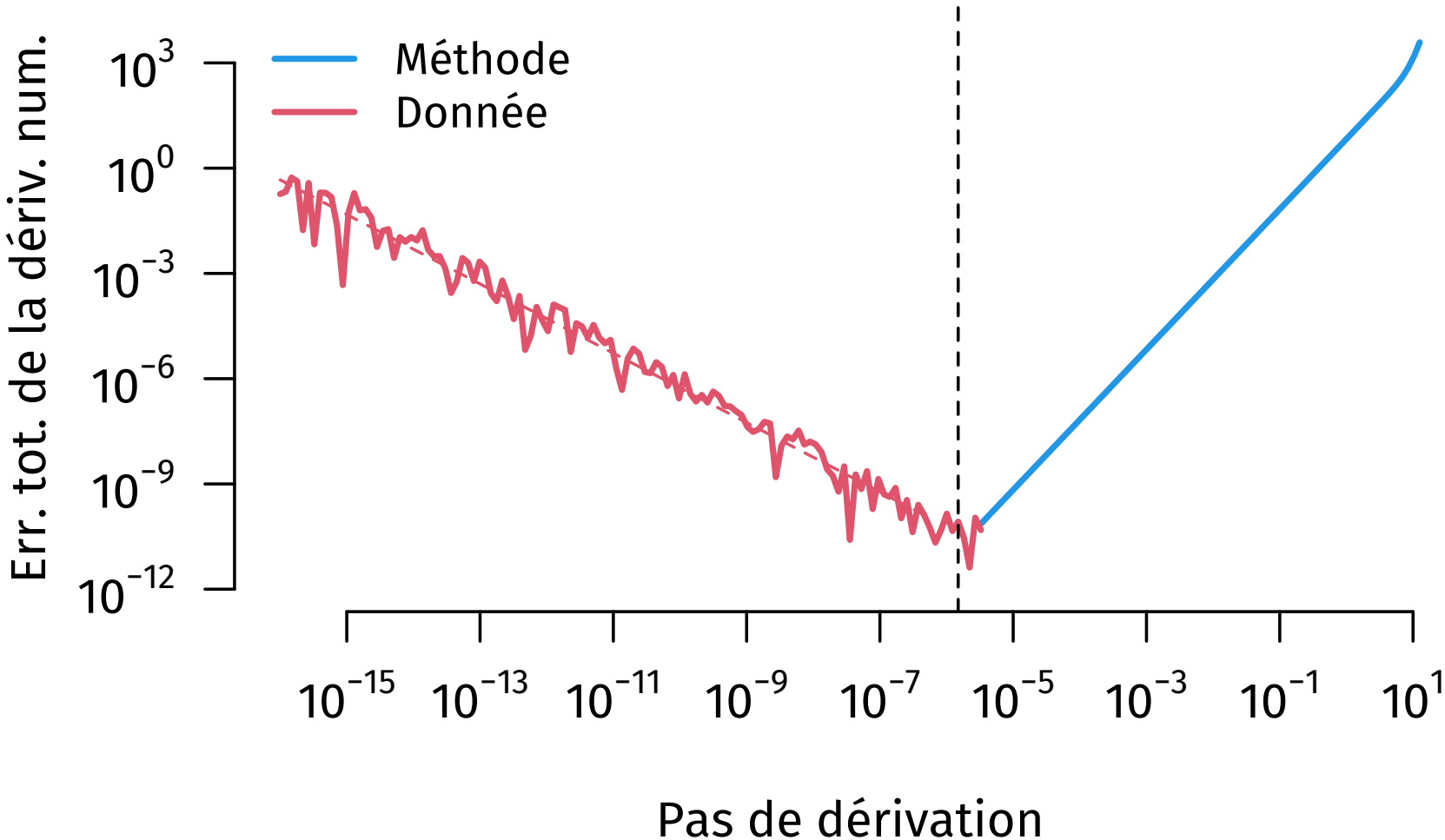
Step size selection is critical for accuracy:

- h too large $\rightarrow$ large **truncation error** from the remainder Taylor-series term (poor **mathematical** approximation)
- h too small $\rightarrow$ large **rounding error** (poor **numerical** approximation): catastrophic cancellation, division of something small by something small, machine accuracy limited by ϵ_{mach}
- h near-optimal $\rightarrow$ the two errors are balanced

One good step size with one difference is better than 3 bad step sizes with refinements and extrapolations!

Approximation error to minimise

$$f(x) := x^4 + \cos x + \exp(x-1), \quad x_0 = \pi/4, \quad f'(x_0) = ?$$



Using the analytical error estimate

Total-error function: conservative absolute-error bound.

$$E(x,h) := \underbrace{\frac{|f'''(x)|}{6}h^2}_{\text{truncation}} + \underbrace{\frac{0.5|f(x)|\epsilon_{\text{mach}}}{h}}_{\text{rounding}}, \qquad h_{\text{opt}} = \sqrt[3]{\frac{1.5|f(x)|}{|f'''(x)|}\epsilon_{\text{mach}}}$$

- Estimate $f'''(x)$ using any reasonable $\tilde{h}$ (e. g. 0.001)

```
Grad(FUN = f, x = x0, h = "plugin")
```

- Dumontet–Vignes (1977) proposed an iterative search algorithm for a better estimate of $f'''(x)$

```
Grad(FUN = f, x = x0, h = "DV")
```

Controlling the error ratio

Observation: when the truncation error and the rounding error are similar, the total error is close to minimal.

Curtis & Reid (1974) proposed choosing h such that

$$\frac{\text{over-estimated } e_{\text{trunc}}}{e_{\text{round}}} \in [10, 1000] \qquad \text{(rule of thumb: 100)}$$

$e_{\text{trunc}} \approx$ forward minus central differences (too conservative!),
$e_{\text{round}} \approx 0.5|f(x)|\epsilon_{\text{mach}}/h$. The RoT ensures that $e_{\text{trunc}} \approx e_{\text{round}}$.

- I created a modified variant with more accurate estimates

```
Grad(func = f, x = x0, h = "CR")
Grad(func = f, x = x0, h = "CRm")
```

Controlling the truncation-branch slope

Stepleman & Winarsky (1979) and Mathur (2012) propose similar algorithms based on the idea of descending down the right branch of the estimated combined error:

- The slope of the right branch of the combined error is a
- Choose h_0 large enough, set $h_1 = 0.5h_0$, get the truncation error estimate from $f'_{\text{CD}}(x, h_1)$ and $f'_{\text{CD}}(x, h_0)$
- Continue shrinking while the slope of the truncation branch is ≈ 2; stop when it deviates due to the substantial round-off error

```
Grad(f, x = x0, h = "SW")
Grad(f, x = x0, method = "M")
```

Fitting the check function (new & experimental!)

The total error looks (in logarithmic axes) like the letter 'V':

- The left, rounding branch is due to division by $h^d \Rightarrow$ slope $= -d$
- The right, truncation branch is due to the remainder in the Taylor series that is approximately a multiple of $h^a \Rightarrow$ slope $= a$

Fit a check function (✓) with known slopes $-d$ and a and unknown horizontal and vertical shifts to find the approximate minimum of the error.

```
Grad(f, x = x0, h = "K")  # For "kink"
```

Showcasing pnd

Compatibility with `numDeriv`

`numDeriv` remains the most popular R package for non-parallel computation of accurate derivatives without step-size selection.

Simply replace the first lowercase letter with an uppercase one.

`numDeriv`	`pnd`
`grad(f, x)`	`Grad(f, x)`
`jacobian(fvector, x)`	`Jacobian(fvector, x)`
`hessian(fscalar, x)`	`Hessian(fscalar, x)`

User-friendliness and thoroughness of pnd

`pnd`

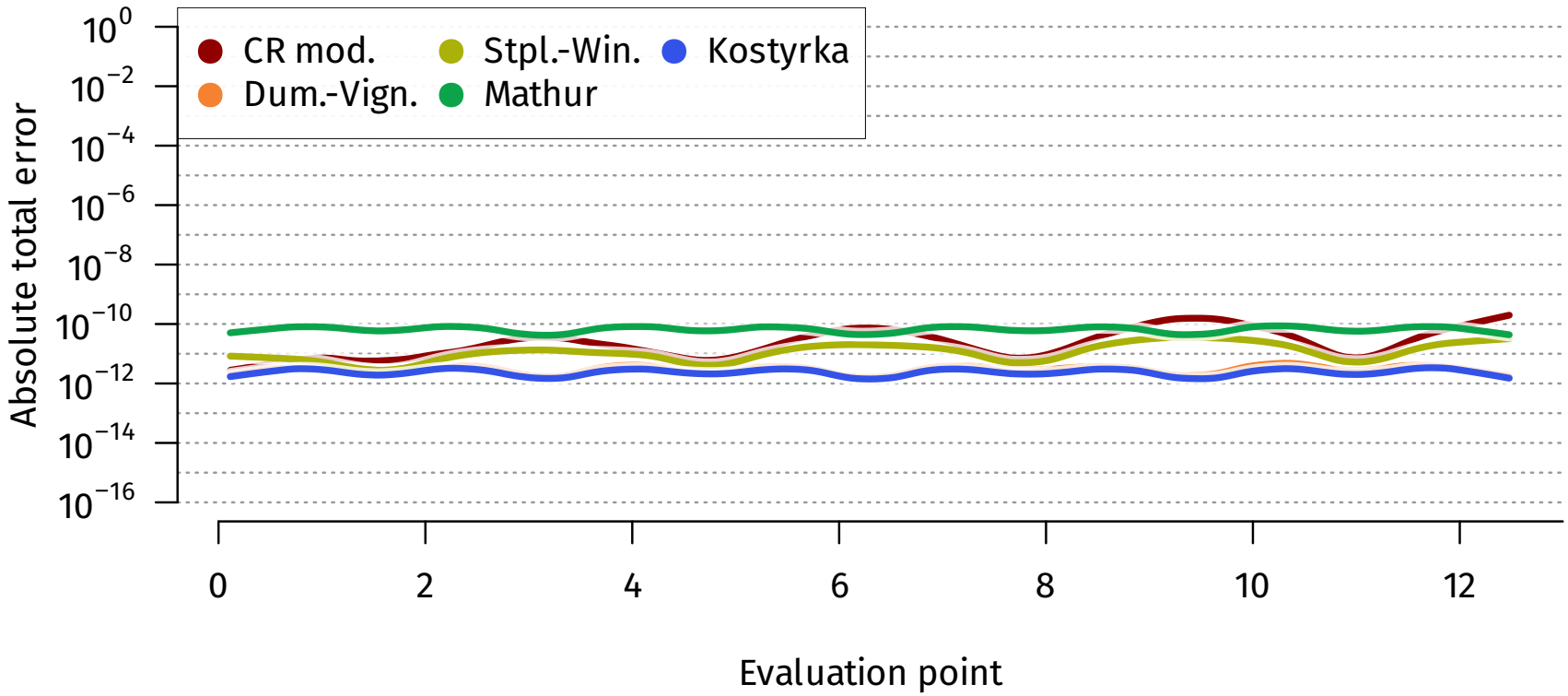
- Catches 74 errors (so far)
- Prints 44 foreseen warnings (so far)
- Supports 5 possible configurations of function properties and capabilities
  - Multi-stage input checks with error handling and possible parallelisation
- Handles arbitrary stencils

`numDeriv`

- 19 errors
- Zero foreseen warnings
- Only 3 possible function configurations
  - One-stage input check, only one error check
- Impossible to obtain Jacobians for certain functions (e. g. $f(x) := (\sin x, \cos x)'$)
  - No user controls

Error of step-selection methods for $f(x) := \sin x$

Evaluation grid: $x \in [0.1, 12.5]$, 10 000 points.



The orange line is obstructed by the blue one.

Further work

- Test improvements for the step-size selection algorithms
- Add memoisation to reuse function values for more accurate derivative estimates
- Respond to users' failing examples and fix bugs
  - Unit tests $<$ user feedback and reproducible errors

Practical recommendations

Do not:

- Set step size $h = 0.01$ because it 'feels right' or you interpret a 1-¢ change
- Use forward differences when evaluating f is fast
- Request 24 cores for quick functions (overhead!)
- Skip step-size search when gradients are the object of interest

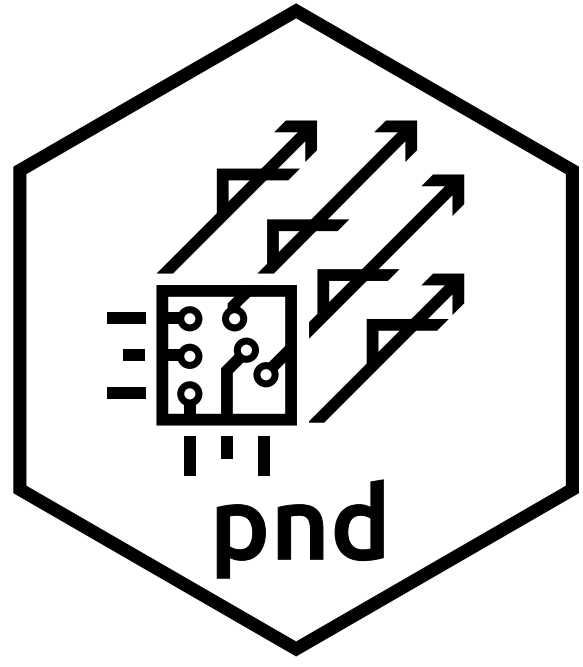
Do:

- Supply function information to skip checks

```
GenD(...,
elementwise = ...,
vectorised = ...,
multivalued = ...)
```

- Use optimal-step search
- Use all CPU cores only if f takes longer than 0.02 s
  - On Windows: create a cluster and pass it to `Grad()` / `Jacobian()`

Thank you for your attention and feedback!



github.com/Fifis/pnd

andrei.kostyrka@gmail.com

Function and its derivative accuracy comparison

- The vast majority of function evaluations on a computer are lossy due to finite memory, even linear transformations
  - Each operation typically adds a $\approx 10^{-16}$ relative error (at least)
- Numerical derivatives are **much less accurate** than function values
  - **...by a factor of $\approx$100 000 *in the best case!***
  - Many software packages settle for a $\times$10 000 000 accuracy degradation
  - ...which is worse $\approx$100 times than it could have been

Non-existent literature / software

- Most modern articles focus on ultra-high-dimensional numerical gradients with much fewer evaluations
  - Only one (!) paper (Mathur 2012, Ph. D. thesis) with a comprehensive treatment of the classical case useful for low-dimensional models
- Existing algorithms (Curtis & Reid 1974, Dumontet & Vignes 1977, Stepleman & Winarsky 1979) lack open-source implementations
  - Popular software packages implement very rough rules and do not refer to any optimality results in the literature
- Most implementations of higher-order and cross-derivatives are through repeated differencing
  - Slower and less accurate than a one-time weighted sum

Partial solutions

- R packages `numDeriv` and `optimParallel`
  - `numDeriv`: the most full-featured arsenal in terms of accuracy, but slow; `optimParallel`: speed gains but no focus on accuracy
- Python's `numdifftools`
  - Discusses Richardson extrapolation; no error analysis
- MATLAB's `Optimisation Toolbox`xt
  - Focuses on parallel evaluation, not accuracy
- Stata's `deriv`
  - Implements a step-size search to obtain 8 accurate digits

Higher-order accuracy of m^{th}-order derivatives

Stencil: strictly increasing sequence of real numbers: $b_1 < \ldots < b_n$. (Preferably symmetric around 0 for the best accuracy.) Example: $b = (-2, -1, 1, 2)$.

Derivatives of any order m with error $O(h^a)$ may be approximated as weighted sums of f evaluated on the **evaluation grid** for that stencil: $x + b_1 h, \ldots, x + b_n h$.

With enough points ($n > m$), one can find such weights $\{w_i\}_{i=1}^n$ that yield the a^{th}-order-accurate approximation of $f^{(m)}$, where $a \leq n - m$:

$$\frac{\mathrm{d}^m f}{\mathrm{d}x^m}(x) = h^{-m} \sum_{i=1}^{n} w_i f(x + b_i h) + O(h^a)$$

Gradient of a function

Gradient: column vector of partial derivatives of a differentiable scalar function.

$$\nabla f(x) := \begin{pmatrix} \frac{\partial f}{\partial x^{(1)}}(x) \\ \vdots \\ \frac{\partial f}{\partial x^{(d)}}(x) \end{pmatrix}$$

- Vector input x + scalar output f = vector ∇
- At any point x, the gradient – the d-dimensional slope – is the **direction and rate of the steepest growth** of f

'A source of anxiety for non-mathematics students.'

J. Nash, 'Nonlinear Parameter Optimization' (2014).

Jacobian of a function

Jacobian: Matrix of gradients for a vector-valued function f.

If $\dim x = d$, $\dim f = k$,

$$\nabla f(x) := \begin{pmatrix} \frac{\partial f}{\partial x^{(1)}}(x) & \cdots & \frac{\partial f}{\partial x^{(d)}}(x) \end{pmatrix}_{k \times d} = \begin{pmatrix} \nabla^T f^{(1)}(x) \\ \vdots \\ \nabla^T f^{(k)}(x) \end{pmatrix}_{k \times d}$$

- Vector input x + vector output f = matrix ∇
- In constrained problems, most solvers (e. g. NLopt) for $\min_x f(x)$ s. t. $g(x) = 0$ require an explicit $\nabla g(x)$

Including incorrectly computed derivatives (mostly gradients or Jacobian matrices) <...> explains almost all the 'failures' of optimisation codes I see. *(Idem.)*

Hessian of a function

Hessian: Square matrix of second-order partial derivatives of a twice-differentiable scalar function.

$$\nabla^2 f(x) := \left\{ \frac{\partial^2 f}{\partial x^{(i)} \partial x^{(j)}} \right\}_{i,j=1}^{d} = \begin{pmatrix} \frac{\partial^2 f}{\partial x^{(1)} \partial x^{(1)}} & \cdots & \frac{\partial^2 f}{\partial x^{(1)} \partial x^{(d)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x^{(d)} \partial x^{(1)}} & \cdots & \frac{\partial^2 f}{\partial x^{(d)} \partial x^{(d)}} \end{pmatrix} (x)$$

The Hessian is the transpose Jacobian of the gradient:

$$\nabla^2 f(x) = \nabla^T [\nabla f(x)]$$

- Vector input x + scalar output f = matrix ∇^2
- If ∇f is differentiable, ∇_f^2 is symmetric

Numerical Hessians via central differences

Let $h_i := (0 \ \ldots \ 0 \underbrace{h}_{i^{\text{th}} \text{ position}} 0 \ \ldots \ 0)'$ and $x_{+-} := x + h_i - h_j$.

4 evaluations of f are required to approximate $\nabla^2_{ij} f$ via CD:

$$\nabla^2_{ij} f(x) := \left[\nabla^T \left(\nabla f(x)\right)\right]_{ij} := \nabla^2_{ij,\text{CD}} f(x) + O(h^2) =$$
$$= \frac{f(x_{++}) - f(x_{-+}) - f(x_{+-}) + f(x_{--})}{4h^2} + O(h^2)$$

- The 4-term sum is as **fast** as the 4-term $\frac{\nabla_i f(x+h_j) - \nabla_i f(x-h_j)}{2h_j}$, but guaranteed to be **symmetric**: $\hat{\nabla}^2_{ij,\text{CD}} = \hat{\nabla}^2_{ji,\text{CD}}$
  - Symmetric repeated differences require 8 terms
- Accuracy implications are being investigated

Total error function properties

On the log-log scale,

- The slope of the left branch is the differentiation order m (times -1)
  - The rounding error of the difference is divided by h^m
- The slope of the right branch is the accuracy order a
  - The truncation error is approximately $f''' \cdots / a!$ times h^a

Optimal step tips and tricks

Rules of thumb to help one save time and obtain more useful quantities once they have determined $h^*_{\text{CD},2}$"

- Since $h^{**}_{\text{CD},2} \propto \epsilon^{1/4}_{\text{mach}}$, $h^*_{\text{CD},2}/h^{**}_{\text{CD},4} \propto \epsilon^{1/12}_{\text{mach}}$.
  Multiply $h^*_{\text{CD},2}$ by $\approx$20 for a reasonable step size for **second derivatives** (f'')
  - Logic: higher derivation order $\Rightarrow$ division by h^2 instead of h $\Rightarrow$ higher rounding error $\Rightarrow$ increasing h^* to reduce it
- Similarly, $h^*_{\text{CD},4} = \propto \epsilon^{1/5}_{\text{mach}}$, $h^*_{\text{CD},2}/h^*_{\text{CD},4} \propto \epsilon^{2/15}_{\text{mach}}$.
  Multiply $h^*_{\text{CD},2}$ by $\approx$100 for a reasonable step size for **4$^{\text{th}}$-order-accurate first derivatives** (f' but better)
  - Logic: higher approximation order $\Rightarrow$ more points $\Rightarrow$ smaller truncation error at $h^*_{\text{CD},2}$ $\Rightarrow$ increasing h^* to reduce the rounding error

Optimal step troubleshooting

- If the function is quasi-quadratic, $f''' \approx 0$, $f'''' \approx 0, \ldots$, then, the step-size search might be unreliable
  - Happens at the optima of likelihood functions in large samples
  - Solution: use the fixed step $\sqrt[3]{\epsilon_{\text{mach}}} \max\{|x|, 1\}$ **after checking diagnostic messages**
  - Typical error: step size too large after dividing by f''', solution at the search range boundary, or solution greater than $|x|\ldots$
- If the function is noisy / approximate, multiply $h^*_{\text{CD},2}$ by 10 per 3 wrong digits of f
  - If $f(x)$ has numerical root search, optimisation, integration, differentiation, etc., $|f(x) - \hat{f}(x)|/|f(x)| \geq 0$ by more than ϵ_{mach}
  - In general, replace ϵ_{mach} in the total-error formula with the maximum expected relative error $\Rightarrow$ h becomes larger with more wrong decimal digits

Paradigms for step-size search

1. Theoretical (plug-in expressions)
2. Empirical (finding the minimum of the total error)

pnd, provides multiple algorithms **(currently under active feature implementation and testing)**.

Analogy: Silverman's rule-of-thumb bandwidth vs. data-driven cross-validated bandwidth in non-parametric econometrics.

Overhead magnitude

- Requesting 2 cores for a parallel job: ≈0.01 s
  - 0.3–0.4 s on Windows due to its inability to fork effectively!
- Extra per-core time with pre-scheduling: ≈0.005 s
  - Plus extra time losses for communication between cores
- If one evaluation of f takes <0.01 s, compare the gains: reduction of the number of tasks vs. overhead per core
- If one evaluation of f takes 0.005–0.010 s, compare the gains: reduction of the number of tasks vs. overhead per core

Time per f	0.002	0.005	0.01	0.02	0.05	0.1	$>$ 0.2
Use cores	1	2–3	4	8	12	16	$\geq$ 24

Long gradients ⇒ always parallelise! And **always benchmark**!

Overhead of pnd

How faster is calculating $\frac{f(x+h)-f(x-h)}{2h}$ by hand than running dozens of checks for user inputs?

Each call of `Grad()` adds 0.5 ms of overhead due to the infrastructure; it increases with dim x. *(To be improved!)*

Compare the overhead of computing $\nabla f'_{\mathrm{CD},2}$ for $f(x) := \sum_{i=1}^{\dim x} x^2 + 4 \sin x + 1.1^x$ in seconds:

dim X	1	10	100
Overhead	0.0005–0.0010	0.0008–0.0010	0.0038–0.0041

Is it acceptable in your practical application?

Example: overhead in light functions

If there are no memory-heavy operations (cloning pages, passing data to child processes), the run time is roughly proportional to the number of cores.

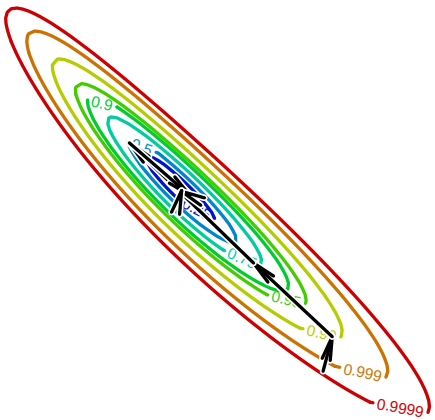
```
f(x) <- {Sys.sleep(s); sin(x)}
```

Times for the Stepleman–Winarsky algorithm to terminate in 7 evaluations / 3 iterations. Ideally, 3 iterations = 3 parallel calls = thrice the time of one call.

s	0.001	0.01	0.1	1
1 core	0.008	0.072	0.702	7.003
2 cores	0.038	0.091	0.456	4.061
3 cores	0.043	0.092	0.368	3.071

Example: slow functions

Smoothed empirical likelihood with missing endogenous variables (Cosma, Kostyrka, Tripathi, 2025). Maximising SEL + computing ∇^2-based std. errors via BFGS on 4 cores.




```
g4 <- function(x) Grad(SEL, x = x,
                       acc.order = 4, cores = 4)
optim(par = c(1, 1), SEL, gr = g4, method = "BFGS")
```

Method	Ord.	Time, s	$\|\nabla \text{SEL}\|$	Evals	Iters
built-in	2	21+3.8	$3.6 \cdot 10^{-4}$	46	10
pnd	2	13+1.5	$2.1 \cdot 10^{-7}$	37	10
pnd	4	16+2.9	$3.3 \cdot 10^{-8}$	32	10

Available algorithms

1. Plug-in
2. Curtis–Reid (1974) and its modification (2025)
3. Dumontet–Vignes (1977)
4. Stepleman–Winarsky (1979)
5. Mathur (2012)
6. Kostyrka (2025)

Improvements for the CR algorithm

1. Estimate the correct truncation error order with 4 parallel evaluations and use the theoretically correct target ratio
   - Instead of 'truncation error = rounding error', use the optimal 'truncation error = rounding error **halved**' rule
2. Obtain $f'_{\mathrm{CD},4}$ with algorithmically chosen $h^*_{\mathrm{CD},2}$ times 120
   - $\approx$ 3 times more accurate than theoretical

Improvements to the AutoDX algorithm

Developed by Ravishankar Mathur (2012, Ph .D. thesis).

- The finite differences may be evaluated on the entire grid on a multi-core machine
- The user may plot the behaviour of the approximated total error as an added bonus

Comparison of median run times

Grid: 9000 exponentially spaced points between 10^{-3} and 10^{6} (exception: 3000 points in $[10^{-2} \ldots 10^{1}]$ for $\exp x$).

Unit: millisecond per step size per grid point + derivative estimation.

Func.	$h^*_{\mathrm{CD},2}$	$\lvert x \rvert \sqrt{\epsilon_{\mathrm{mach}}}$	CR	CRm2	CRm4	DV	SW	M
$\sin x$	<0.01	<0.01	0.18	0.16	0.20	0.46	0.33	1.70
$\exp x$	<0.01	0.02	0.15	0.15	0.15	0.26	0.18	1.72
$\log x$	<0.01	0.01	0.15	0.11	0.15	0.17	0.27	2.09
$\sqrt{x}$	<0.01	<0.01	0.16	0.11	0.15	0.16	0.14	2.13
$\tan^{-1} x$	<0.01	<0.01	0.14	0.11	0.17	0.19	0.42	1.69

Comparison of median absolute errors

Error: $|f'(x) - f'_{\mathrm{CD},2}|$ for 9000 exponentially spaced points between 10^{-3} and 10^{6} (exception: 3000 points in $[10^{-2} \ldots 10^{1}]$ for $\exp x$).

Short exponential notation: `5.6e-9` $= 5.6 \cdot 10^{-9}$.

Func.	$h^*_{\mathrm{CD},2}$	$\lvert x\rvert\sqrt{\epsilon_{\mathrm{mach}}}$	CR	CRm2	**CRm4**	DV	*SW*	M
$\sin x$	5.7e-11	2.6e-09	1.2e-09	1.2e-10	**2.3e-11**	1.1e-09	*3.0e-11*	5.1e-10
$\exp x$	1.5e-11	2.6e-08	2.2e-10	5.7e-11	**1.3e-11**	3.7e-09	*1.4e-11*	2.7e-09
$\log x$	1.3e-12	0.0e+00	5.6e-12	1.7e-12	**1.6e-13**	1.3e-11	*5.3e-13*	1.0e-10
$\sqrt{x}$	2.1e-12	2.7e-10	9.3e-12	2.4e-12	**2.4e-13**	3.7e-11	*8.2e-13*	1.5e-10
$\tan^{-1} x$	6.8e-13	5.9e-11	3.5e-13	2.2e-13	**2.7e-14**	7.8e-13	*1.6e-13*	9.6e-12

Logic behind the best methods

- **Curtis–Reid (1974) + my modification #2:** use 4 available intermediate points and function values from truncation and rounding error estimation to obtain a 4^{th}-order-accurate estimate (unlike 2)
- *Stepleman–Winarsky:* the truncation error should be quartered if the step size is halved $\Rightarrow$ start at a step size larger than the best guess and halve it until the decrease is substantially different from 2 due to rounding errors
  - I added a safety step for checking finiteness and extra warnings for edge cases
- Mathur: SW-like evaluation for many points simultaneously + diagnostic plots available