Introduction to Algorithmic Differentiation

Motivation. Essential Calculus. Finite Differences ($F : \boldsymbol{R}^n \to \boldsymbol{R}^m$)

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Contents

Outline

Motivation

$F(x) = 0$ by Newton's Method

A given system of nonlinear equations $F(x) = 0$ is solved iteratively by Newton's method. If the new iterate is not close enough to the root, i.e, $\|F(x + \Delta x)\|_2 > \epsilon$ for some measure of accuracy (e.g, in terms of the $\rightarrow$ Euclidean norm $\|.\|_2$) of the numerical approximation $1 \gg \epsilon > 0$, then it becomes the starting point for the next iteration yielding the recurrence

$$F'(x) \cdot \Delta x = -F(x)$$
$$x = x + \Delta x$$

Convergence is not guaranteed in general. Damping of the magnitude of the next step may help.

$$x = x - \alpha \cdot F'(x)^{-1} \cdot F(x) \quad \text{for } 0 < \alpha \leq 1 \, .$$

The damping parameter α can be determined by line search (e.g, recursive bisection yielding $\alpha = 1,\ 0.5,\ 0.25,\ \ldots$) such that decrease in absolute function value is ensured.

Gradient descent aims to approximate a solution to the unconstrained nonlinear optimization problem $\min_{x \in \mathbb{R}^n} f(x)$ iteratively.

Starting from some initial estimate for $\tilde{x}$ steps into descent directions are taken.

The gradient f' of f wrt. x indicates local increase ($\|f'\|_2 > 0$) or decrease ($\|f'\|_2 < 0$) of the function value.

Aiming for decrease the next step should be in direction of the negative gradient $-f'$. No further local decrease in the function value can be achieved for $\|f'\|_2 = 0$ (necessary optimality condition).

The step size is typically damped in order to ensure continued progress toward the minimum yielding the recurrence

$$x = x - \alpha \cdot f'(x) \quad \text{while } \|f'(x)\|_2 > \epsilon \, .$$

The damping parameter can be determined by line search.

Newton's method can be used to approximate a solution to the unconstrained nonlinear optimization problem $\min_{x \in R^n} f(x)$ iteratively.

Linearization of the first-order optimality condition yields a sequence (for evolving x) of local linear (in $\Delta x \in R^n$) approximations of the first-order optimality condition as

$$f'(x + \Delta x) = f'(x) + f''(x) \cdot \Delta x = 0$$

and hence systems of linear equations

$$f'' \cdot \Delta x = -f'$$

to be solved for Δx and followed by updating

$$x = x + \alpha \cdot \Delta x$$

iteratively for a suitable start value x and damping parameter $R \ni \alpha > 0$ determined by line search.

Outline

The following details have little relevance for the "black-box" perspective taken during this course.

Internal structural and mathematical properties will be exploited in the course "Advanced Algorithmic Differentiation" offered in the summer semester.

We simulate the expected "payoff" $\mathbb{E}((x^* - K)_+) \in \boldsymbol{R}$ with "strike" $K \in \boldsymbol{R}$ and solution $x^* = x(p(t), t = T) \in \boldsymbol{R}$ of the stochastic initial value problem

$$dx = f_1(x(p(t), t), p(t), t))dt + f_2(x(p(t), t), p(t), t)dW$$

at target time ("expiry") $T > 0$ and with Brownian Motion dW.

The initial state $x(p(0), 0) = x^0(p(0))$ is defined implicitly as the solution of the convex unconstrained optimization problem

$$\min_{x^0} f_0(x^0(p(0)), p(0))$$

and approximation of which is obtained using Newton's method.

Forward finite differences in time with time step $0 < \Delta t \ll 1$ applied to $dx = f_1(x(p(t), t), p(t), t))dt + f_2(x(p(t), t), p(t), t)dW$ yield the Euler-Maruyama scheme

$$x^{i+1} = x^i + \Delta t \cdot f_1(x^i, p_i, i \cdot \Delta t) + \sqrt{\Delta t} \cdot f_2(x^i, p_i, i \cdot \Delta t) \cdot dW^i$$

for $i = 0, \ldots, n-1$, target time $T = n \cdot \Delta t$, parameter vector $\mathrm{p} = (p_i) \in \boldsymbol{R}^{n+1}$, and with random numbers dW^i drawn from the standard normal distribution $N(0, 1)$.

The solution $\mathbb{E}((x^* - K)_+)$ is approximated using Monte Carlo simulation over the Euler-Maruyama paths for $K = p_n$. Contributions of individual paths become equal to

$$\max(x^* - p_n, 0) \ .$$

We are interested in first- and second-order sensitivities of $\mathbb{E}((x^* - K)_+)$ wrt. p. The size of the gradient grows as n.

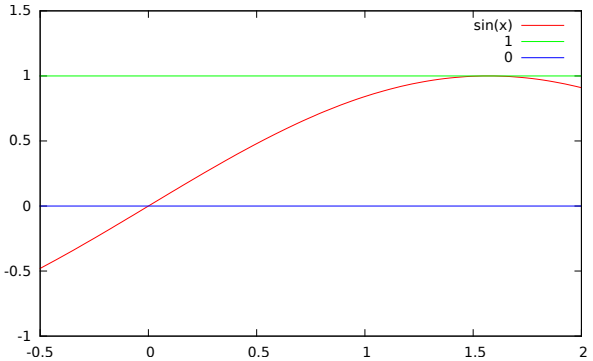
Differentiability?

NECESSARY!

Is

$$y = f(x) = \begin{cases} \sin(x) & 0 \neq x < \frac{\pi}{2} \\ 1 & x \geq \frac{\pi}{2} \\ 0 & x = 0 \end{cases}$$

everywhere (twice) differentiable?



... once yes.

... not twice at $x = \frac{\pi}{2}$ as

$$\underbrace{\lim_{h \to 0} \frac{\frac{df}{dx}\left(\frac{\pi}{2}\right) - \frac{df}{dx}\left(\frac{\pi}{2} - h\right)}{h}}_{= -\sin\left(\frac{\pi}{2}\right) = -1} \neq \underbrace{\lim_{h \to 0} \frac{\frac{df}{dx}\left(\frac{\pi}{2} + h\right) - \frac{df}{dx}\left(\frac{\pi}{2}\right)}{h}}_{=0} !$$

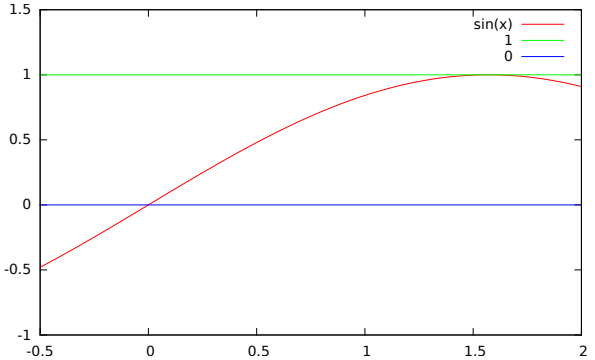
Differentiability!

SUFFICIENT?

Is

$$y = f(x) = \begin{cases} \sin(x) & 0 \neq x < \frac{\pi}{2} \\ 1 & x \geq \frac{\pi}{2} \\ 0 & x = 0 \end{cases}$$

everywhere algorithmically (twice) differentiable?



No, since

$$\frac{df}{dx}(0) = 0 \quad \text{while} \quad \lim_{h \to 0} \frac{f(0) - f(0-h)}{h} = \lim_{h \to 0} \frac{f(0+h) - f(0)}{h} = \cos(0) = 1$$

$$\frac{d^2 f}{dx^2}\left(\frac{\pi}{2}\right) = 0 \quad \text{while} \quad \underbrace{\lim_{h \to 0} \frac{\frac{df}{dx}\left(\frac{\pi}{2}\right) - \frac{df}{dx}\left(\frac{\pi}{2} - h\right)}{h}}_{= -\sin\left(\frac{\pi}{2}\right) = -1} \neq \underbrace{\lim_{h \to 0} \frac{\frac{df}{dx}\left(\frac{\pi}{2} + h\right) - \frac{df}{dx}\left(\frac{\pi}{2}\right)}{h}}_{=0} !$$

Hello AD World

Algorithmic Differentiability?


```
1  template<typename T, typename PT>
2  void paths(size_t ncs, size_t from, size_t to, T& x, const std::vector<T>& p, const
       std::vector<std::vector<PT>>& dW, T& s) {
3    using namespace std;
4    size_t n=p.size()-1;
5    T x0=x;
6    for (size_t j=from;j<to;j++) {
7      for (size_t i=0;i<n;i+=ncs) steps(j,i,i+ncs,x,p,dW);
8      s+=max(x-p[n],0.0); // non-differentiable
9      x=x0;
10   }
11 }
```

- kink at `x-p[n]` $\Rightarrow$ subgradient
- vanishing second derivative (wrong!)
- sigmoid for smoothing and to preserve higher-order sensitivities on local `x` and `p[n]`

As an alternative to the non-differentiable max-payoff we consider the following smoothed (using sigmoid function) version:

```
1  template<typename T, typename PT>
2  void paths(size_t ncs, size_t from, size_t to, T& x, const std::vector<T>& p, const
       std::vector<std::vector<PT>>& dW, T& s) {
3    using namespace std;
4    size_t n=p.size()-2;
5    T x0=x;
6    for (size_t j=from;j<to;j++) {
7      for (size_t i=0;i<n;i+=ncs) steps(j,i,i+ncs,x,p,dW);
8      T sig=1/(1+exp(-(x-p[n])/p[n+1])); s+=(x-p[n])*sig; // differentiable
9      x=x0;
10   }
11 }
```

+ differentiability
- skewed primal value
+/- hyperparameter optimization problem
→ hybrid: exact primal + smoothed derivative

```
1  template<typename T, typename PT>
2  void f(size_t ncp, size_t ncs, const PT &eps, T& x, const std::vector<T>& p, const
       std::vector<std::vector<PT>>& dW) {
3    newton(x,p[0],eps);
4    T s=0;
5    size_t m=dW.size();
6    for (size_t j=0;j<m;j+=ncp) paths(ncs,j,j+ncp,x,p,dW,s);
7    x=s/m;
8  }
```

We are looking for the gradient of the active result `x` with respect to the active arguments `p`. All remaining arguments (and results) are passive.

Rationale: Keep it simple. Capture representative features of numerical simulations.

Let

$$f_0(x, p) = e^x - p \cdot x$$

and

$$f_1(x, p, t) = p \cdot \sin(x \cdot t) \text{ and } f_2(x, p, t) = p \cdot \cos(x \cdot t) \text{ for } t \in [0, 1] .$$

The given `*.exe` expect six command-line arguments: number of paths (`m`), number of time steps per path (`n-2`), number of paths per bundle (`ncp`), number of time steps per bundle (`ncs`), upper bound on absolute value f_0' (`eps`). "width" of sigmoidal smoothing (last element in `p`).

Demo:

- `g_cfd.exe` yields approximation of gradient by central finite differences
- `g_t.exe` yields exact (machine accuracy) gradient by tangent AD
- `g_a.exe` yields exact gradient by adjoint AD

Outline

- terminology
  - continuity
  - differentiability
  - derivatives
  - linearity
  - convexity / concavity
- Taylor series
- linearization
- chain rule (on dag)

Let $\boldsymbol{R}^n$ be the domain of the multivariate scalar function $f : \boldsymbol{R}^n \to \boldsymbol{R}$. The function f is continuous at a point $\tilde{x} \in \boldsymbol{R}^n$ if

$$\lim_{x \to \tilde{x}} f(x) = f(\tilde{x}) \quad .$$

A multivariate vector function

$$F = \begin{pmatrix} f_1 \\ \vdots \\ f_m \end{pmatrix} : \boldsymbol{R}^n \to \boldsymbol{R}^m$$

is continuous if and only if all its component functions f_i, $i = 1, \ldots, m$ are continuous.

Let $\boldsymbol{R}^n$ be the domain of the multivariate scalar function $f : \boldsymbol{R}^n \to \boldsymbol{R}$. The function f is differentiable at point $\tilde{x} \in \boldsymbol{R}^n$ if there is a vector $f' \in \boldsymbol{R}^n$ such that

$$f(\tilde{x} + \Delta x) = f(\tilde{x}) + f' \cdot \Delta x + r$$

with asymptotically vanishing remainder $r = r(\tilde{x}, \Delta x) \in \boldsymbol{R}$, such that

$$\lim_{\Delta x \to 0} \frac{r}{\|\Delta x\|_2} = 0 \,, \quad \text{where } \|v\|_2 \equiv \sqrt{v^T \cdot v} = \sqrt{\sum_{i=0}^{n-1} v_i^2}$$

denotes the Euclidean norm of the vector $v \in \boldsymbol{R}^n$.

$$f^{[1]} = f' = f'(x) \equiv \frac{df}{dx}(x) : \boldsymbol{R}^n \to \boldsymbol{R}^n$$

is called the gradient of f.

Let $f : \boldsymbol{R}^n \to \boldsymbol{R}$ be differentiable at $\tilde{\mathrm{x}} \in \boldsymbol{R}^n$. Then

$$f'(\tilde{\mathrm{x}}) = \begin{pmatrix} \frac{dy}{dx_0} \\ \vdots \\ \frac{dy}{dx_{n-1}} \end{pmatrix}$$

where $y = f(\mathrm{x})$ and

$$\frac{dy}{dx_i} = \frac{dy}{dx_i}(\tilde{\mathrm{x}}) = \lim_{\Delta x \to \pm 0} \frac{f(\tilde{\mathrm{x}} + \Delta x \cdot \mathrm{e}_i) - f(\tilde{\mathrm{x}})}{\Delta x} < \infty$$

with the *i*-th Cartesian basis vector in $\boldsymbol{R}^n$ denoted by e_i.

Let $\boldsymbol{R}^n$ be the domain of the multivariate vector function $F : \boldsymbol{R}^n \to \boldsymbol{R}^m$. The function F is differentiable at point $\tilde{x} \in \boldsymbol{R}^n$ if there is a matrix $F' \in \boldsymbol{R}^{m \times n}$ such that

$$F(\tilde{x} + \Delta x) = F(\tilde{x}) + F' \cdot \Delta x + r$$

with asymptotically vanishing remainder $r = r(\tilde{x}, \Delta x) \in \boldsymbol{R}^m$, such that

$$\lim_{\Delta x \to 0} \frac{\|r\|_2}{\|\Delta x\|_2} = 0 \quad .$$

The matrix

$$F^{[1]} = F' = F'(x) \equiv \frac{dF}{dx}(x) : \boldsymbol{R}^n \to \boldsymbol{R}^{m \times n}$$

is called the Jacobian of F.

Let $F : \boldsymbol{R}^n \to \boldsymbol{R}^n$ be differentiable at $\tilde{x} \in \boldsymbol{R}^n$. Then

$$F'(\tilde{x}) = \begin{pmatrix} \frac{dy_0}{dx_0} & \cdots & \frac{dy_0}{dx_{n-1}} \\ \vdots & & \vdots \\ \frac{dy_{n-1}}{dx_0} & \cdots & \frac{dy_{n-1}}{dx_{n-1}} \end{pmatrix}$$

where $y_j = F_j(x)$ and

$$\frac{dy_j}{dx_i} = \frac{dy_j}{dx_i}(\tilde{x}) = \lim_{\Delta x \to \pm 0} \frac{F_j(\tilde{x} + \Delta x \cdot e^i) - F_j(\tilde{x})}{\Delta x} < \infty \, .$$

Let $f : \boldsymbol{R}^n \to \boldsymbol{R}$ be differentiable at $\tilde{x} \in \boldsymbol{R}^n$. It is twice differentiable at $\tilde{x}$ if $f' : \boldsymbol{R}^n \to \boldsymbol{R}^n$ is differentiable at $\tilde{x}$.

The matrix

$$f^{[2]}(\tilde{x}) = f''(\tilde{x}) \equiv \begin{pmatrix} \frac{d^2 y}{dx_0^2} & \cdots & \frac{d^2 y}{dx_0 dx_{n-1}} \\ \vdots & & \vdots \\ \frac{d^2 y}{dx_{n-1} dx_0} & \cdots & \frac{d^2 y}{dx_{n-1}^2} \end{pmatrix}$$

is called the Hessian matrix of f at point $\tilde{x}$.

If f' is continuous [at some point, within some subdomain], then f is called continuously differentiable [at this point, within this subdomain].

If f is twice continuously differentiable at $\tilde{x}$, then its Hessian is symmetric, i.e, $f''(\tilde{x}) = f''(\tilde{x})^T$.

Hessians of multivariate vector functions $F : \boldsymbol{R}^n \to \boldsymbol{R}^m$ are 3-tensors. So are third derivatives of f, and so forth.

Wake Up!

Derive gradient and Hessian for $y = \sin(x_0)/x_1$.

A function $F : \boldsymbol{R}^n \to \boldsymbol{R}^m$ is linear if

- $F(\mathrm{a} + \mathrm{b}) = F(\mathrm{a}) + F(\mathrm{b})$
- $F(\alpha \cdot \mathrm{a}) = \alpha \cdot F(\mathrm{a})$

for all $\mathrm{a}, \mathrm{b} \in \boldsymbol{R}^n$ and $\alpha \in \boldsymbol{R}$.

Example: $F(\mathrm{x}) = M \cdot \mathrm{x}$ with $M \in \boldsymbol{R}^{m \times n}$ is linear.

$$F(\mathrm{a} + \mathrm{b}) = M \cdot (\mathrm{a} + \mathrm{b}) = M \cdot \mathrm{a} + M \cdot \mathrm{b} = F(\mathrm{a}) + F(\mathrm{b})$$
$$F(\alpha \cdot \mathrm{a}) = M \cdot \alpha \cdot \mathrm{a} = \alpha \cdot M \cdot \mathrm{a} = \alpha \cdot F(\mathrm{a})$$

Functions $F(\mathrm{x}) = M \cdot \mathrm{x} + \mathrm{v}$ with $\mathrm{v} \in \boldsymbol{R}^m$ are called affine.

Affine functions define linear systems $m = n$ as well as linear least-squares problems $m \neq n$.

A function $f : \boldsymbol{R}^n \to \boldsymbol{R}$ is convex if and only if its Hessian f'' is positive semi-definite for all $x \in \boldsymbol{R}^n$, i.e, $\forall 0 \neq v \in \boldsymbol{R}^n$

$$v^T \cdot f''(x) \cdot v \geq 0 \, .$$

One can show that f is strictly convex over $\boldsymbol{R}^n$ if f'' is positive definite for all $x \in \boldsymbol{R}^n$, i.e,

$$v^T \cdot f''(x) \cdot v > 0 \, .$$

The other direction does not hold in general.

Similarly, concavity is defined in terms of negative (semi-)definiteness of the Hessian.

The concepts can be generalized for multivariate vector functions $F : \boldsymbol{R}^n \to \boldsymbol{R}^m$.

Wake Up!

Is the identity matrix positive definite?

- $f : \boldsymbol{R}^n \to \boldsymbol{R}$

$$f(x + \Delta x) = f(x) + f'(x)^T \cdot \Delta x + \frac{1}{2} \cdot \Delta x^T \cdot f''(x) \cdot \Delta x + O(\|\Delta x\|_2^3)$$

- $F : \boldsymbol{R}^n \to \boldsymbol{R}^m$

$$F(x + \Delta x) = F(x) + F'(x) \cdot \Delta x + O(\|\Delta x\|_2^2)$$

Higher-order terms are omitted to avoid tensor notation.

Similar to the scalar case, many numerical methods for nonlinear problems in nD are built on local replacement of the target function with a linear (affine) approximation derived from the truncated Taylor series expansion and "hoping" that

$$F(x + \Delta x) \approx F(x) + F'(x) \cdot \Delta x$$

i.e, relying on the assumption that the remainder is reasonably small within the subdomain of interest.

The solution of a sequence of linear problems is expected to yield an iterative approximation of the solution to the nonlinear problem.

Newton's method for the solution of systems of nonlinear equations as a sequence of solutions of linear problems is THE example, as

$$F(x) + F'(x) \cdot \Delta x = 0 \quad \text{at} \quad \Delta x = -F'(x)^{-1} \cdot F(x) = 0 \,.$$

Linearization of scalar problems yield linear equations $a \cdot x = b$. Small changes in $b \in \boldsymbol{R}$ imply small (same order) changes in $x \in \boldsymbol{R}$.

Linearization of multivariate vector functions yields systems of linear equations $A \cdot \mathrm{x} = \mathrm{b}$. Small changes in $\mathrm{b} \in \boldsymbol{R}^n$ can yield large changes in $\mathrm{x} \in \boldsymbol{R}^n$ due to poor conditioning of $A \in \boldsymbol{R}^{n \times n}$. For example,

$$\begin{matrix} x + y = 2 \\ x + 1.001 \cdot y = 2 \end{matrix} \quad \Rightarrow \quad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

while

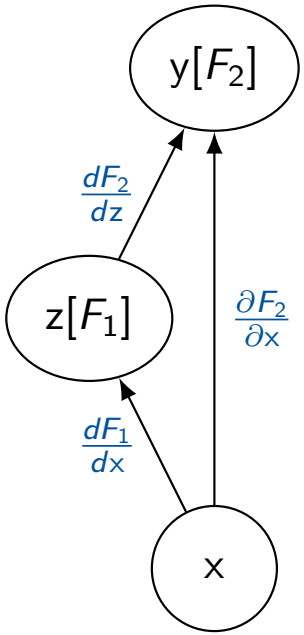
$$\begin{matrix} x + y = 2 \\ x + 1.001 \cdot y = 2.001 \end{matrix} \quad \Rightarrow \quad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

due to near singularity of A ($\mathrm{cond}(A) \approx 4004$).

Let $y = F(x) : \boldsymbol{R}^n \to \boldsymbol{R}^m$ be such that

$$y = F(x) = F_2(F_1(x), x) = F_2(z, x)$$

with (continuously) differentiable $F_1 : \boldsymbol{R}^n \to \boldsymbol{R}^p$ and $F_2 : \boldsymbol{R}^p \times \boldsymbol{R}^n \to \boldsymbol{R}^m$.



Then F is continuously differentiable over $\boldsymbol{R}^n$ and

$$\frac{dF}{dx}(\tilde{x}) = \frac{dF_2}{dx}(\tilde{z}, \tilde{x}) = \frac{dF_2}{dz}(\tilde{z}, \tilde{x}) \cdot \frac{dF_1}{dx}(\tilde{x}) + \frac{\partial F_2}{\partial x}(\tilde{z}, \tilde{x})$$

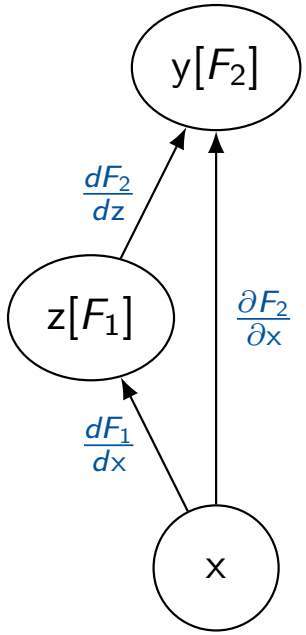
for all $\tilde{x} \in \boldsymbol{R}^n$ and $\tilde{z} = F_1(\tilde{x})$.

Notation: $\frac{\partial F_2}{\partial x}$ partial derivative; $\frac{dF_2}{dx}$ total derivative

A composite function $y = F(x)$ such as

$$z = F_1(x)$$
$$y = F_2(z, x)$$

induces a directed acyclic graph (DAG) $G = (V, E)$ with vertices in V representing variables (e.g, x, z and y) and with local (partial) derivatives associated with the edges in E.



$$F'(x) \equiv \frac{dy}{dx} = \sum_{\text{path} \in \text{DAG}} \prod_{(i,j) \in \text{path}} \frac{\partial v_j}{\partial v_i} = \frac{\partial y}{\partial x} + \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} + \frac{dy}{dz} \cdot \frac{dz}{dx}$$

Outline

The directional derivative of F in direction $x^{(1)} \in R^n$ (product of the Jacobian of F with $x^{(1)}$) can be approximated at $\tilde{x} \in \boldsymbol{R}^n$ by finite difference quotients as

$$F'(\tilde{x}) \cdot x^{(1)} \approx_1 \left(\frac{F(\tilde{x} + \Delta x_i \cdot x^{(1)}) - F(\tilde{x})}{\Delta x_i} \right)_{i=0}^{n-1} \quad \text{(forward)}$$

$$\approx_1 \left(\frac{F(\tilde{x}) - F(\tilde{x} - \Delta x_i \cdot x^{(1)})}{\Delta x_i} \right)_{i=0}^{n-1} \quad \text{(backward)}$$

$$\approx_2 \left(\frac{F(\tilde{x} + \Delta x_i \cdot x^{(1)}) - F(\tilde{x} - \Delta x_i \cdot x^{(1)})}{2 \cdot \Delta x_i} \right)_{i=0}^{n-1} \quad \text{(central)}$$

where $\Delta x = \Delta x(\tilde{x}) \in \boldsymbol{R}^n$ is a vector of suitable perturbations.

Forward and backward finite differences exhibit first-order accuracy ($\approx_1$) while central finite differences turn out to be second-order accurate ($\approx_2$).

Individual columns of the Jacobian of F can be approximated at $\tilde{x} \in \boldsymbol{R}^n$ as

$$F'(\tilde{x}) \approx_1 \left(\frac{F(\tilde{x} + \Delta x_i \cdot e_i) - F(\tilde{x})}{\Delta x_i} \right)_{i=0}^{n-1} \quad \text{(forward)}$$

$$\approx_1 \left(\frac{F(\tilde{x}) - F(\tilde{x} - \Delta x_i \cdot e_i)}{\Delta x_i} \right)_{i=0}^{n-1} \quad \text{(backward)}$$

$$\approx_2 \left(\frac{F(\tilde{x} + \Delta x_i \cdot e_i) - F(\tilde{x} - \Delta x_i \cdot e_i)}{2 \cdot \Delta x_i} \right)_{i=0}^{n-1} \quad \text{(central)}$$

where e_i denotes the i-th Cartesian basis vector in $\boldsymbol{R}^n$.

Forward and backward finite difference quotients follow immediately from the Taylor series expansion

$$F(x \pm \Delta x) = F(x) \pm F'(x) \cdot \Delta x + O(\|\Delta x\|_2^2)$$

truncated after the first-order term. The error decreases linearly (following division by Δx) with $0 < \|\Delta x\|_2 \ll 1$ yielding first-order accuracy.

The central finite difference quotient is obtained by truncation of the difference

$$F(x + \Delta x) - F(x - \Delta x)$$

of both Taylor series expansions after the first-order term. The error turns out to decrease quadratically with $0 < \|\Delta x\|_2 \ll 1$ yielding second-order accuracy.

Finite Differences

Implementation


```
1  #include "f.hpp" // primal source
2
3  #include <vector>
4
5
6
7  template<typename T, typename PT> // central finite difference gradient driver
8  void gradient(size_t ncp, size_t ncs, const PT &eps, T& x, std::vector<T>& p,
     const std::vector<std::vector<PT>> &dW, std::vector<T>& dxdp) {
9
10
11
12
13
14
15
16
17   }
18
19 }
20
21 #include "g_main.hpp" // main
```

Finite Differences

Implementation


```
1  #include "f.hpp" // primal source
2
3  #include <vector>
4
5
6
7  template<typename T, typename PT> // central finite difference gradient driver
8  void gradient(size_t ncp, size_t ncs, const PT &eps, T& x, std::vector<T>& p,
     const std::vector<std::vector<PT>> &dW, std::vector<T>& dxdp) {
9    size_t n=p.size(); // size of gradient
10
11
12
13
14
15
16
17
18
19 }
20
21 #include "g_main.hpp" // main
```

Finite Differences

Implementation


```
1  #include "f.hpp" // primal source
2
3  #include <vector>
4
5
6
7  template<typename T, typename PT> // central finite difference gradient driver
8  void gradient(size_t ncp, size_t ncs, const PT &eps, T& x, std::vector<T>& p,
     const std::vector<std::vector<PT>> &dW, std::vector<T>& dxdp) {
9    size_t n=p.size(); // size of gradient
10   for (size_t i=0;i<n;i++) { // perturb active arguments individually
11
12
13
14
15
16
17   }
18
19 }
20
21 #include "g_main.hpp" // main
```

Finite Differences

Implementation


```
1  #include "f.hpp" // primal source
2
3  #include <vector>
4  #include <cmath>
5  #include <limits>
6
7  template<typename T, typename PT> // central finite difference gradient driver
8  void gradient(size_t ncp, size_t ncs, const PT &eps, T& x, std::vector<T>& p,
     const std::vector<std::vector<PT>> &dW, std::vector<T>& dxdp) {
9    size_t n=p.size(); // size of gradient
10   for (size_t i=0;i<n;i++) { // perturb active arguments individually
11     T h= (p[i]==0) // custom perturbation
12         ? sqrt(std::numeric_limits<T>::epsilon())
13         : sqrt(std::numeric_limits<T>::epsilon())*fabs(p[i]);
14
15
16
17   }
18
19 }
20
21 #include "g_main.hpp" // main
```

Finite Differences

Implementation


```
1  #include "f.hpp" // primal source
2
3  #include <vector>
4  #include <cmath>
5  #include <limits>
6
7  template<typename T, typename PT> // central finite difference gradient driver
8  void gradient(size_t ncp, size_t ncs, const PT &eps, T& x, std::vector<T>& p,
     const std::vector<std::vector<PT>> &dW, std::vector<T>& dxdp) {
9    size_t n=p.size(); // size of gradient
10   for (size_t i=0;i<n;i++) { // perturb active arguments individually
11     T h= (p[i]==0) // custom perturbation
12         ? sqrt(std::numeric_limits<T>::epsilon())
13         : sqrt(std::numeric_limits<T>::epsilon())*fabs(p[i]);
14     T x_p=x; p[i]+=h; f(ncp,ncs,eps,x_p,p,dW); // "right"
15
16
17   }
18
19 }
20
21 #include "g_main.hpp" // main
```

Finite Differences

Implementation


```
1  #include "f.hpp" // primal source
2
3  #include <vector>
4  #include <cmath>
5  #include <limits>
6
7  template<typename T, typename PT> // central finite difference gradient driver
8  void gradient(size_t ncp, size_t ncs, const PT &eps, T& x, std::vector<T>& p,
     const std::vector<std::vector<PT>> &dW, std::vector<T>& dxdp) {
9    size_t n=p.size(); // size of gradient
10   for (size_t i=0;i<n;i++) { // perturb active arguments individually
11     T h= (p[i]==0) // custom perturbation
12         ? sqrt(std::numeric_limits<T>::epsilon())
13         : sqrt(std::numeric_limits<T>::epsilon())*fabs(p[i]);
14     T x_p=x; p[i]+=h; f(ncp,ncs,eps,x_p,p,dW); // "right"
15     T x_m=x; p[i]-=2*h; f(ncp,ncs,eps,x_m,p,dW); p[i]+=h; // "left"
16
17   }
18
19 }
20
21 #include "g_main.hpp" // main
```

Finite Differences

Implementation


```
1  #include "f.hpp" // primal source
2
3  #include <vector>
4  #include <cmath>
5  #include <limits>
6
7  template<typename T, typename PT> // central finite difference gradient driver
8  void gradient(size_t ncp, size_t ncs, const PT &eps, T& x, std::vector<T>& p,
     const std::vector<std::vector<PT>> &dW, std::vector<T>& dxdp) {
9    size_t n=p.size(); // size of gradient
10   for (size_t i=0;i<n;i++) { // perturb active arguments individually
11     T h= (p[i]==0) // custom perturbation
12         ? sqrt(std::numeric_limits<T>::epsilon())
13         : sqrt(std::numeric_limits<T>::epsilon())*fabs(p[i]);
14     T x_p=x; p[i]+=h; f(ncp,ncs,eps,x_p,p,dW); // "right"
15     T x_m=x; p[i]-=2*h; f(ncp,ncs,eps,x_m,p,dW); p[i]+=h; // "left"
16     dxdp[i]=(x_p-x_m)/(2*h); // finite difference quotient
17   }
18
19 }
20
21 #include "g_main.hpp" // main
```

Finite Differences

Implementation


```
1  #include "f.hpp" // primal source
2
3  #include <vector>
4  #include <cmath>
5  #include <limits>
6
7  template<typename T, typename PT> // central finite difference gradient driver
8  void gradient(size_t ncp, size_t ncs, const PT &eps, T& x, std::vector<T>& p,
     const std::vector<std::vector<PT>> &dW, std::vector<T>& dxdp) {
9    size_t n=p.size(); // size of gradient
10   for (size_t i=0;i<n;i++) { // perturb active arguments individually
11     T h= (p[i]==0) // custom perturbation
12         ? sqrt(std::numeric_limits<T>::epsilon())
13         : sqrt(std::numeric_limits<T>::epsilon())*fabs(p[i]);
14     T x_p=x; p[i]+=h; f(ncp,ncs,eps,x_p,p,dW); // "right"
15     T x_m=x; p[i]-=2*h; f(ncp,ncs,eps,x_m,p,dW); p[i]+=h; // "left"
16     dxdp[i]=(x_p-x_m)/(2*h); // finite difference quotient
17   }
18   f(ncp,ncs,eps,x,p,dW); // unperturbed function value
19 }
20
21 #include "g_main.hpp" // main
```

The Hessian

$$f'' = f''(x) \equiv \frac{d^2 f}{dx^2}(x) = \left(\frac{d^2 f}{dx_i dx_j}(x) \right) \in \boldsymbol{R}^{n \times n}$$

of a twice continuously differentiable multivariate scalar function $f : \boldsymbol{R}^n \to \boldsymbol{R}$ can be approximated at a given point $\tilde{x} \in \boldsymbol{R}^n$ as a (central) finite difference approximation of the Jacobian of a (central) finite difference approximation of the gradient

$$f' = f'(x) \equiv \frac{df}{dx}(x) = \left(\frac{df}{dx_i}(x) \right) \in \boldsymbol{R}^n$$

of f :

$$\frac{d^2 f}{dx_i dx_j}(\tilde{x}) \approx \frac{\frac{df}{dx_i}(\tilde{x} + e_j \cdot \Delta x_j) - \frac{df}{dx_i}(\tilde{x} - e_j \cdot \Delta x_j)}{2 \cdot \Delta x_j} .$$

e_j denotes the j-th Cartesian basis vector in $\boldsymbol{R}^n$.

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4
5
6
7
8
9
10
11
12
13
14
15
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4    std::vector<T> dxdp_p(n), dxdp_m(n); // perturbed gradients
5
6
7
8
9
10
11
12
13
14
15
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4    std::vector<T> dxdp_p(n), dxdp_m(n); // perturbed gradients
5    for (size_t i=0;i<n;i++) { // perturb active arguments individually
6
7
8
9
10
11
12
13
14
15   }
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4    std::vector<T> dxdp_p(n), dxdp_m(n); // perturbed gradients
5    for (size_t i=0;i<n;i++) { // perturb active arguments individually
6      T h= (p[i]==0) // same perturbation for gradient
7          ? sqrt(sqrt(std::numeric_limits<T>::epsilon()))
8          : sqrt(sqrt(std::numeric_limits<T>::epsilon()))*fabs(p[i]);
9
10
11
12
13
14
15   }
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4    std::vector<T> dxdp_p(n), dxdp_m(n); // perturbed gradients
5    for (size_t i=0;i<n;i++) { // perturb active arguments individually
6      T h= (p[i]==0) // same perturbation for gradient
7          ? sqrt(sqrt(std::numeric_limits<T>::epsilon()))
8          : sqrt(sqrt(std::numeric_limits<T>::epsilon()))*fabs(p[i]);
9      T x_p=x; p[i]+=h; gradient(ncp,ncs,eps,x_p,p,dW,dxdp_p); // "right"
10
11
12
13
14
15   }
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4    std::vector<T> dxdp_p(n), dxdp_m(n); // perturbed gradients
5    for (size_t i=0;i<n;i++) { // perturb active arguments individually
6      T h= (p[i]==0) // same perturbation for gradient
7          ? sqrt(sqrt(std::numeric_limits<T>::epsilon()))
8          : sqrt(sqrt(std::numeric_limits<T>::epsilon()))*fabs(p[i]);
9      T x_p=x; p[i]+=h; gradient(ncp,ncs,eps,x_p,p,dW,dxdp_p); // "right"
10     T x_m=x; p[i]-=2*h; gradient(ncp,ncs,eps,x_m,p,dW,dxdp_m); // "left"
11     p[i]+=h; // unperturb
12
13
14
15   }
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4    std::vector<T> dxdp_p(n), dxdp_m(n); // perturbed gradients
5    for (size_t i=0;i<n;i++) { // perturb active arguments individually
6      T h= (p[i]==0) // same perturbation for gradient
7          ? sqrt(sqrt(std::numeric_limits<T>::epsilon()))
8          : sqrt(sqrt(std::numeric_limits<T>::epsilon()))*fabs(p[i]);
9      T x_p=x; p[i]+=h; gradient(ncp,ncs,eps,x_p,p,dW,dxdp_p); // "right"
10     T x_m=x; p[i]-=2*h; gradient(ncp,ncs,eps,x_m,p,dW,dxdp_m); // "left"
11     p[i]+=h; // unperturb
12     dxdp[i]=(x_p-x_m)/(2*h); // unperturbed gradient
13
14
15   }
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4    std::vector<T> dxdp_p(n), dxdp_m(n); // perturbed gradients
5    for (size_t i=0;i<n;i++) { // perturb active arguments individually
6      T h= (p[i]==0) // same perturbation for gradient
7          ? sqrt(sqrt(std::numeric_limits<T>::epsilon()))
8          : sqrt(sqrt(std::numeric_limits<T>::epsilon()))*fabs(p[i]);
9      T x_p=x; p[i]+=h; gradient(ncp,ncs,eps,x_p,p,dW,dxdp_p); // "right"
10     T x_m=x; p[i]-=2*h; gradient(ncp,ncs,eps,x_m,p,dW,dxdp_m); // "left"
11     p[i]+=h; // unperturb
12     dxdp[i]=(x_p-x_m)/(2*h); // unperturbed gradient
13     for (size_t j=0;j<n;j++) // finite difference quotients
14       ddxdpp[j][i]=(dxdp_p[j]-dxdp_m[j])/(2*h);
15   }
16
17 }
18
19 #include "h_main.hpp" // main
```

Second-Order Finite Differences

Implementation


```
1  template<typename T, typename PT> // central finite difference Hessian driver
2  void hessian(size_t ncp, size_t ncs, const PT &eps, T &x, std::vector<T> &p, const
       std::vector<std::vector<PT>> &dW, std::vector<T> &dxdp, std::vector<std::
     vector<T>>& ddxdpp) {
3    size_t n=p.size(); // size of gradient / Hessian
4    std::vector<T> dxdp_p(n), dxdp_m(n); // perturbed gradients
5    for (size_t i=0;i<n;i++) { // perturb active arguments individually
6      T h= (p[i]==0) // same perturbation for gradient
7          ? sqrt(sqrt(std::numeric_limits<T>::epsilon()))
8          : sqrt(sqrt(std::numeric_limits<T>::epsilon()))*fabs(p[i]);
9      T x_p=x; p[i]+=h; gradient(ncp,ncs,eps,x_p,p,dW,dxdp_p); // "right"
10     T x_m=x; p[i]-=2*h; gradient(ncp,ncs,eps,x_m,p,dW,dxdp_m); // "left"
11     p[i]+=h; // unperturb
12     dxdp[i]=(x_p-x_m)/(2*h); // unperturbed gradient
13     for (size_t j=0;j<n;j++) // finite difference quotients
14       ddxdpp[j][i]=(dxdp_p[j]-dxdp_m[j])/(2*h);
15   }
16   f(ncp,ncs,eps,x,p,dW); // unperturbed function value
17 }
18
19 #include "h_main.hpp" // main
```

Outline

Summary