Fundamental Numerical Methods for Model Parameter Estimation

A Gentle Hands-On Introduction

Uwe Naumann

Software and Tools for Computational Engineering,[1]
RWTH Aachen

[1] https://www.stce.rwth-aachen.de/

Outline

- RWTH since 2004
- before that Dresden (Germany), Sophia-Antipolis (France), Hatfield (UK), Argonne (USA)
- Principal Scientist at Numerical Algorithms Group Ltd., Oxford, UK (`www.nag.com`)
- Diplom / Ph.D. in Applied Mathematics (TU Dresden)
- Interests
  - algorithmic differentiation
  - adjoint numerical methods
  - combinatorial problems / algorithms
  - research software engineering
  - program analysis / compiler construction
  - HPC

Research (current PhD projects)

- S. Afghan: Pruning Neural Networks
- Z. Arshad: AD of Linear Algebra in .NET
- A. Fleming: Differentiability of Shocks in CFD Simulations
- G. Kauerauf: Differential Inversion
- N. Kichler: Delocalized AD
- S. Märtens: AD Mission Planning and Control

- external PhD students at FZ Jülich, UHerts, CERN

Teaching (current courses)

- Algorithmic Differentiation (Ba/Ma)
- Numerical Methods and Software with C++ (Ba)
- Programming with C++ (Ba)
- Program Transformation and Compiler Construction Lab (Ba/Ma)
- Parallel Graph Algorithms Lab (Ba/Ma)
- Simulation Software Engineering Lab (Ba/Ma)

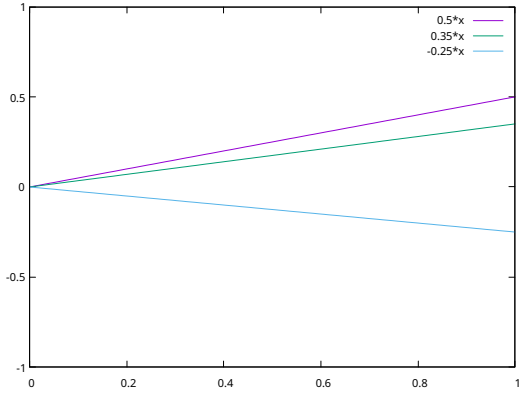
- The parameterized model $y = f(x, p) \equiv p \cdot x$ is used to simulate the states of the journey taken by the two gentlemen.
- The value of the parameter p (slope of the straight line) is unknown. It shall be estimated based on data $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}}$ collected by the ladies through observation of previous journeys taken by the gentlemen. The model shall be calibrated.
- The objective is to minimize the error of the simulation, that is, to determine a value of p such that $p \cdot \mathbf{x} \approx \mathbf{y}$.
- A naïve search method is implemented in SNC++ (Scripting for Numerics with C++). The simple subset of C++ is introduced along the way.
- A first "proper" numerical method is derived mathematically to set the stage for the remainder of the course.

Whiteboard: model and data in x-y coordinate system

$$y = f(p, x) = p \cdot x$$

$$f : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R} : (-\infty, \infty) \times [0, \infty) \to (-\infty, \infty)$$

```
double f(double p, double x)
  {
  assert(x>=0);
  return p*x;
}
```



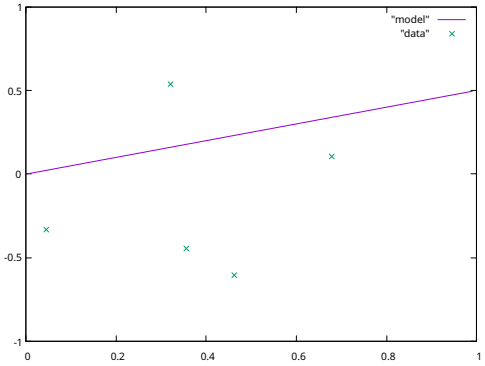
"All models are wrong, some are useful."

(George Box, British Statistician, 1976)

We consider (randomly generated) observations

$$(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}}$$
$$\mathbf{x} = (x_i)_{0,\ldots,n_{\text{obs}}-1}$$
$$\mathbf{y} = (y_i)_{0,\ldots,n_{\text{obs}}-1} ,$$

e.g.,



$p = 0.5$?

```
1  int nobs=5; // e.g. 5 observations
2
3  using VT=Eigen::VectorX<double>;
4
5  VT xobs=VT::Random(nobs); // random entries in [−1,1]
6  xobs=xobs.cwiseProduct(xobs); // random entries in [0,1]
7  VT yobs=VT::Random(nobs); // random entries in [−1,1]
```


- Scripting for Numerics with C++ (SNC++) is a (very small) subset of C++ augmented with support for linear algebra and designed for compatibility with algorithmic differentiation.
- Programming with SNC++ will be straightforward, if you are not new to (imperative) programming.
- An introductory slide deck and various sample scripts are provided for your reference.

→ Interactively: intro/px_1.cpp to reproduce picture on previous slide

We aim to minimize the least-squares error of the simulation relative to the given observation of reality, that is,

$$\min_p \left(\sum_{i=0}^{n_{\text{obs}}-1} (p \cdot x_i - y_i)^2 \right) .$$

→ Whiteboard: error of model wrt. data

The following implementation of this objective in SNC++ will be considered:

```
1  double e(double p, VT xobs, VT yobs) {
2    // conditions
3    assert(xobs.size()==nobs); assert(xobs.size()==yobs.size());
4    double e=0; // variable holding the result
5    int i=0;
6    while (i<xobs.size()) {
7      e=e+pow(f(p,xobs(i))−yobs(i),2); // accumulation of local errors
8      i=i+1;
9    }
10   return e;
11 }
```

→ Interactively: intro/px_2.cpp

The optimal value of the free parameter p can be computed trivially for a single observation. The error becomes equal to zero.

Initial guess

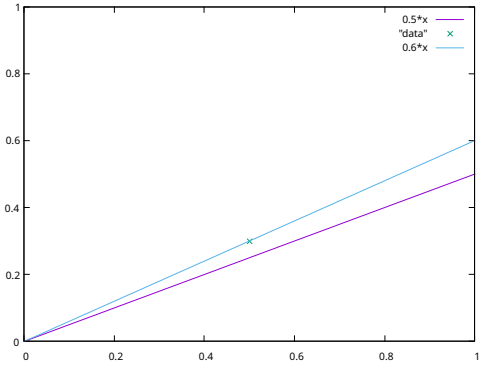
$$p = 0.5$$

Observation

$$(0.5, 0.3)$$

Parameter estimation

$$p = \frac{0.3}{0.5} = 0.6$$



→ Interactively: intro/px_3.cpp

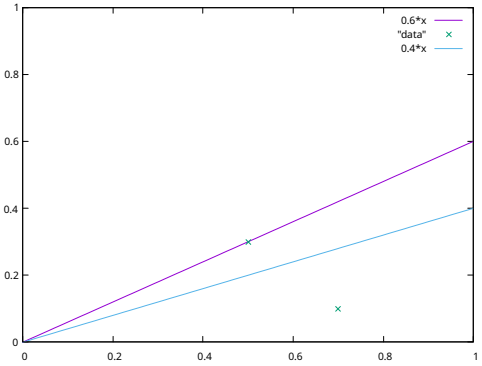
The estimation of p becomes less trivial for two or more observations.

Current estimate

$$p = 0.6$$

Observations

$$\left(\begin{pmatrix}0.5\\0.7\end{pmatrix}\begin{pmatrix}0.3\\0.1\end{pmatrix}\right)$$



Parameter estimation

$$\min_p (p \cdot 0.5 - 0.3)^2 + (p \cdot 0.7 - 0.1)^2$$

Perhaps

$$p \approx 0.4?$$

Naive optimization by grid search in main.cpp:

- Given p ...
- from $p - 1$ to $p + 1$ using steps of size 0.01 ...
- evaluate e and check for potentially lower value.
- Return minimum and minimizer.
- Eigen library in `../../thirdParty/eigen`.
- Building:

```
g++ -I../../thirdParty/eigen main.cpp -o main.exe
```

→ exercises/1/

Exploitation of further mathematical insight yields better optimization methods.
E.g., the model $y = p \cdot x$ is *linear*[2] in p.

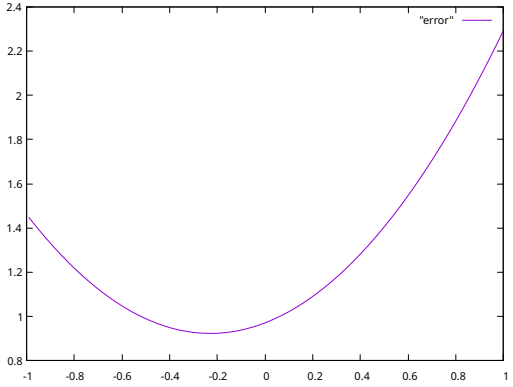
The error

$$e = e(p) = \sum_{i=0}^{n_{\text{obs}}-1} (p \cdot x_i - y_i)^2$$

is *quadratic* in p.

The error is *convex* as it is twice *differentiable* and

$$\frac{d^2 e}{dp^2} = 2 \cdot \sum \cdot x_i^2 > 0 \,.$$



→ Inspection: intro/px_4.cpp for plot

→ Hands on: exercises/1/main_opt.cpp for optimized grid search

[2] ... to be properly defined later

Convexity of the error implies a minimum at the *stationary point*, where

$$\frac{de}{dp} = 0 \,.$$

Consequently,

$$\begin{aligned}\frac{de}{dp} &= \frac{d}{dp}\sum_{i=0}^{n_{\text{obs}}-1}(p \cdot x_i - y_i)^2 = \frac{d}{dp}\sum_{i=0}^{n_{\text{obs}}-1}\left((p \cdot x_i)^2 - p \cdot x_i \cdot y_i + y_i^2\right)\\ &= \sum_{i=0}^{n_{\text{obs}}-1}\frac{d}{dp}\left((p \cdot x_i)^2 - 2 \cdot p \cdot x_i \cdot y_i + y_i^2\right) = \sum_{i=0}^{n_{\text{obs}}-1}\left(2 \cdot p \cdot x_i^2 - 2 \cdot x_i \cdot y_i\right)\\ &= 2 \cdot \left(p \cdot \sum_{i=0}^{n_{\text{obs}}-1} x_i^2 - \sum_{i=0}^{n_{\text{obs}}-1} x_i \cdot y_i\right) = 2 \cdot \left(p \cdot \mathbf{x}^T \cdot \mathbf{x} - \mathbf{x}^T \cdot \mathbf{y}\right) = 0\end{aligned}$$

and, hence,[3]

$$p = \frac{\mathbf{x}^T \cdot \mathbf{y}}{\mathbf{x}^T \cdot \mathbf{x}} \,.$$

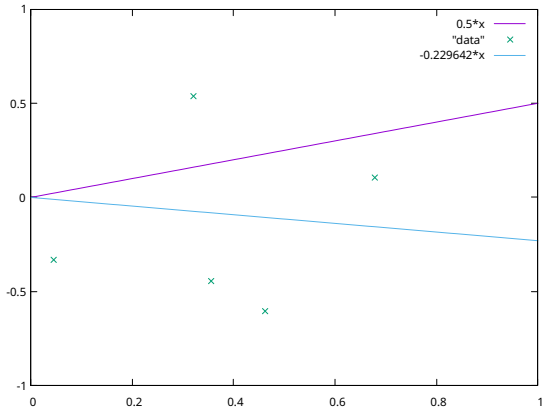
[3] The (row) vector $\mathbf{x}^T$ is the transpose of the (column) vector $\mathbf{x}$.

Let's implement

$$p = \frac{\mathbf{x}^T \cdot \mathbf{y}}{\mathbf{x}^T \cdot \mathbf{x}}$$

for random observations

$$(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}} .$$



→ Inspection and experiments: intro/px_5.cpp

In many relevant cases, an approximation of p is the best we can hope for.

Inaccurate p immediately yield the question about the effect of this error on the quality of predictions produced by the calibrated model, that is, ...

How does an error Δp in the parameter p influence the result?
Does the error Δy in the result remain bounded (cannot become arbitrarily large) relative to the value y of the correct result?

The model $y = f(p, x) = p \cdot x$ is actually evaluated as

$$y + \Delta y = f(p + \Delta p, x) = (p + \Delta p) \cdot x .$$

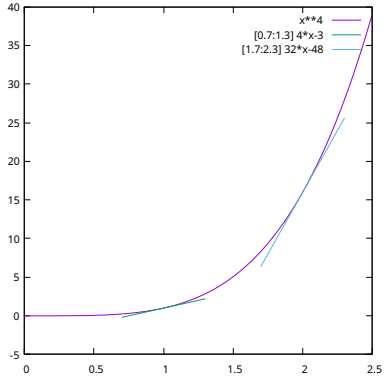
The (absolute) error Δp in the parameter p yields the (absolute) error Δy in the prediction y produced by the model at a given x.

The value of the first derivative[a] of y with respect to (wrt.) p suggests a possible approach to error analysis as

$$\Delta y \approx \frac{df}{dp} \cdot \Delta p$$

in some (possibly tiny) neighborhood of the value of p.

[a] f needs to be *differentiable*.



Note: replace x by p in plot.

From

$$\Delta y \approx \frac{df}{dp} \cdot \Delta p = \frac{d(p \cdot x)}{dp} \cdot \Delta p = x \cdot \Delta p$$

it follows that the absolute error Δy grows as x does.

However, so does $y = p \cdot x$, implying that the relative error

$$\delta y \equiv \frac{\Delta y}{y}$$

in the result remains bounded, that is, it does not exceed a constant multiple ($C \in \mathbb{R}$) of the relative error δp in the parameter. Obviously,

$$\delta y \equiv \frac{\Delta y}{y} \approx \frac{\frac{df}{dp} \cdot \Delta p}{f(x, p)} = \frac{x \cdot \Delta p}{p \cdot x} = \frac{\Delta p}{p} = \delta p$$

yields $C = 1$.

Let

$$y = f(p, x) = x + p \; .$$

The relative error δy in the result can become arbitrarily large, implying that scalar addition (as well as scalar subtraction) is numerically instable.

Obviously,

$$\delta y = \frac{\Delta y}{y} \approx \frac{\frac{df}{dp} \cdot \Delta p}{f(x,p)} = \frac{\Delta p}{x+p}$$

Note that $|\delta y| \to \infty$ for $x \to -p$ and $|\Delta p| > 0$.

Numerical instability of addition and subtraction implies the numerical instability of parameter estimation for more than one observation. However, potential complications are restricted to selected points (e.g. for $x \to -p$ in the above) only.

Scalar division, required, e.g., for the solution

$$p = \phi(x, y) = \frac{y}{x}$$

of the parameter estimation problem for a single observation (x, y), is numerically stable. From

$$\Delta p \approx \frac{d\phi}{dx} \cdot \Delta x = -\frac{y}{x^2} \cdot \Delta x$$

it follows that

$$\delta p = \frac{\Delta p}{p} \approx \frac{-\frac{y}{x^2}}{\frac{y}{x}} \cdot \Delta x = -\frac{1}{x} \cdot \Delta x = -\delta x \; ,$$

implying that the relative error δp grows as $-\delta x$ does. Similarly,

$$\delta p = \frac{\Delta p}{p} \approx \frac{\frac{d\phi}{dy} \cdot \Delta y}{p} = \frac{\frac{1}{x}}{\frac{y}{x}} \cdot \Delta y = \frac{1}{y} \cdot \Delta y = \delta y \; .$$

Contents

Outline

Errors are typically introduced during mathematical modelling as the result of abstraction and simplification; see also George Box (1976): "All models are wrong, some are useful."

There are three major sources of additional numerical errors not to be ignored. The results of numerical simulation and optimization can get terribly wrong due to

- problem condition (to be considered in further detail during the discussion of the vector case)
- numerical approximation (see upcoming error analysis of linearization of x^2 as illustration of $\mathcal{O}$-notation)
- computer arithmetic.

The latter deserves a closer look before we proceed.


```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    double a=1, h=1e-15, b=a+h, c=b-a, one=c/h;
6    cout << one << "=1?" << endl;
7    return 0;
8  }
```

Building followed by running yields

```
1.11022=1?
```

Something is wrong ...

→ Live: Experiments with misc/computer_arithmetic/ca1.cpp

Unfortunately, real numbers $x \in \mathbb{R}$ cannot be represented precisely by today's computers. They are approximated by floating-point numbers with base β, accuracy t and exponent range $[L, U]$ as follows:

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-1}}{\beta^{t-1}} \right) \beta^{\varepsilon} ,$$

where $0 \leq d_i \leq \beta - 1$ for $i = 0, \ldots, t-1$ and $L \leq \varepsilon \leq U$. The sequence of digits $M = d_0 d_1 \ldots d_{t-1}$ over base β is called mantissa. The exponent is denoted as ε.

We consider normalized floating-point number systems, that is, $d_0 = 0 \Leftrightarrow x = 0$ implying $1 \leq M < \beta$. E.g.,

$$\pm 0.0127 = \pm \left(1 + \frac{2}{10} + \frac{7}{10^2} \right) \cdot 10^{-2} = \pm 1.27 \cdot 10^{-2}$$

in a decimal ($\beta = 10$) system with $t \geq 3$ and $-2 \in [L, U]$.

The double precision floating-point data type **double** uses 64 bits:

- 52 bits for the mantissa (53rd bit equal to 1 due to normalization)
- 11 bits for the exponent
- 1 sign bit

yielding 15 significant digits in decimal output format with minimum and maximum absolute values of 2.22507e-308 and 1.79769e+308, respectively.

The signed exponent ε is shifted into the unsigned (positive) $[0, 2^{11} - 1]$ range (biased exponent). Its true value is obtained by subtracting $2^{10} - 1 = 1023$.

The commonly used floating-point number formats (double as well as single precision) are described in the IEEE 754 standard.

Special cases are defined to represent

- ± 0 (ignoring normalization):

$$M = e = 0$$

- $\pm\infty$ (e.g., division of nonzero by zero):

$$e = 2^{11} - 1 \text{ (all bits equal to 1)} \quad \text{and} \quad M = 0$$

- "Not a Number" (NaN, e.g., resulting from $\infty + (-\infty)$):

$$e = 2^{11} - 1 \quad \text{and} \quad M > 0\ .$$

Further conventions apply. Refer to the standard for details.

Consider a 5-bit (binary) normalized floating-point format using

- 2 bits for the mantissa (3rd bit equal to 1 due to normalization)
- 2 bits for the (biased exponent)
- 1 sign bit

with, hence, $\beta = 2$, $t = 2$ and $[L, U] = [-1, 2]$.

The signed exponent ε is shifted into the $[0, 2^2 - 1] = [0, 3]$ (actually, $[1, 2^2 - 2]$ due to special interpretation of $\varepsilon = 2^2 - 1$ as NaN or $\pm\infty$ and because the combination of a vanishing exponent with a nonzero mantissa denotes denormalized numbers) range. Its true value is obtained by subtracting $2^1 - 1 = 1$.

E.g., the following conversion from a binary value to its decimal value holds:

$$\underbrace{0}_{\pm}\ \underbrace{01}_{\varepsilon+1}\ \underbrace{11}_{M-1} = (1 + 2^{-1} + 2^{-2}) \cdot 2^{2^0 - (2^1 - 1)} = 1.75 \cdot 2^0 = 1.75$$

The following sixteen (negative versions by leading 1) special cases are defined:[4]

- Denormalized numbers

$$00000_2 = 0_{10} \quad \text{(zero as special denormalized number)}$$
$$00001_2 = (2^{-2}) \cdot 2^{0-(2^1-1)} = 0.25 \cdot 2^{-1} = 0.125_{10}$$
$$00010_2 = (2^{-1}) \cdot 2^{0-(2^1-1)} = 0.5 \cdot 2^{-1} = 0.25_{10}$$
$$00011_2 = (2^{-1} + 2^{-2}) \cdot 2^{0-(2^1-1)} = 0.75 \cdot 2^{-1} = 0.375_{10}$$

- Infinity: $01100_2 = \infty_{10}$
- NaN: 01101_2 or 01110_2 or 01111_2

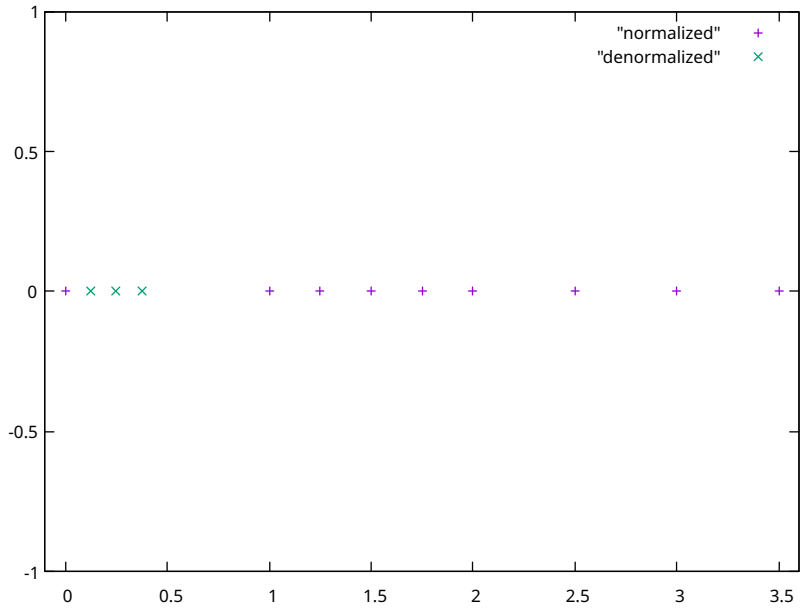
→ Optionally: Experiments with **double** in misc/to_bin

[4] Subscripts mark the base of the number format, here binary and decimal.

Moreover, the following eight absolute values[5] can be represented:

$$00100_2 = 1 \cdot 2^{2^0 - (2^1 - 1)} = 1 \cdot 2^0 = 1_{10}$$
$$00101_2 = (1 + 2^{-2}) \cdot 2^{2^0 - (2^1 - 1)} = 1.25 \cdot 2^0 = 1.25_{10}$$
$$00110_2 = (1 + 2^{-1}) \cdot 2^{2^0 - (2^1 - 1)} = 1.5 \cdot 2^0 = 1.5_{10}$$
$$00111_2 = (1 + 2^{-1} + 2^{-2}) \cdot 2^{2^0 - (2^1 - 1)} = 1.75 \cdot 2^0 = 1.75_{10}$$
$$01000_2 = 1 \cdot 2^{2^1 - (2^1 - 1)} = 1 \cdot 2^1 = 2_{10}$$
$$01001_2 = (1 + 2^{-2}) \cdot 2^{2^1 - (2^1 - 1)} = 1.25 \cdot 2^1 = 2.5_{10}$$
$$01010_2 = (1 + 2^{-1}) \cdot 2^{2^1 - (2^1 - 1)} = 1.5 \cdot 2^1 = 3_{10}$$
$$01011_2 = (1 + 2^{-1} + 2^{-2}) \cdot 2^{2^1 - (2^1 - 1)} = 1.75 \cdot 2^1 = 3.5_{10}$$

[5] covering both positive and negative values and transformed into decimal format



The IEEE 754 standard defines various rounding rules for approximating real values that cannot be represented precisely by the given floating-point number system. The default is

Round to nearest, break ties to even.

"Even" implies a vanishing last entry d_{t-1} of the mantissa for $\beta = 2$. E.g.,

$$
\begin{aligned}
0.99_{10} &\Rightarrow \text{ underflow} \\
1_{10} &= 1_{10} \\
1.2_{10} &\approx 1.25_{10} \\
2.25_{10} &\approx 2_{10} \\
2.26_{10} &\approx 2.5_{10} \\
3.51_{10} &\Rightarrow \text{ overflow}
\end{aligned}
$$

The smallest value $\epsilon > 0$ such that

$$1 + \epsilon \neq 1$$

is called machine epsilon. It plays an important role for the decision about (small) perturbations of floating-point values in various numerical methods, e.g., finite difference approximation of derivatives to be discussed later.

Our 5-bit sample floating-point number system features $\epsilon = 1$ as

$$1 + 1 = 2 \neq 1 \,,$$

while there is no smaller positive number in the system, which is not the case for less simple floating-point number systems such as IEEE 754 **double**.

Let the 5-bit sample floating-point type be implemented as my_float_type in

```
1 #include "my_float_type.h"
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6   my_float_type a=1.75, h=1, b=a+h, c=b-a, one=c/h;
7   cout << one << "=1?" << endl;
8   return 0;
9 }
```

Explain the following output:

```
1.25=1?
```

Under rounding, b=3 and c=1.25 yields one=1.25.

Models
Executive Summary

- Many models used in Computational [Life] Science, Engineering, Economics, Finance, etc. describe local change of the state of the underlying system rather than the value of the latter as the solution of an algebraic equation or even as an explicit function of given arguments.
- Consequently, mathematical modeling often yields differential instead of algebraic equations.
- We consider four ordinary differential equations with known symbolic solutions yielding four corresponding algebraic models. Different aspects are captured.
- Essential mathematical terminology (linearity and beyond, continuity, differentiability, Taylor series expansion) is introduced.

Models
[differential|algebraic]×[implicit|explicit]

We distinguish parameterized (by $p \in \mathbb{R}$)

- explicit algebraic: $y(x,p) = f(x,p)$, e.g., $y = p \cdot x$
- implicit algebraic: $F(y(x,p),x,p) = 0$, e.g., $p \cdot x - y = 0 \Rightarrow y = p \cdot x$
- explicit differential: $\frac{dy}{dx} = g(y(x,p),x,p)$, e.g., $\frac{dy}{dx} = p \Rightarrow y = p \cdot x + c$
- implicit differential: $G\left(\frac{dy}{dx}, y(x,p), x, p\right) = 0$

models (also: equations). All but the last play a role in this course.

Unique solutions for differential models require initial values $y^0(p) = y(x^0,p)$, yielding so-called initial value problems.

We aim to simulate lack of the gentlemen's fitness (deviation from x-axis; none $\Rightarrow$ 100% fit) as a function of their progress made in x-direction.

The explicit algebraic model $y = f(p, x)$ features an unknown parameter $p \in \mathbb{R}$, later to be estimated for given observations by the ladies of the gentlemen's journey.

$y = f(p, x)$ is defined as the solution of the following initial values problems:
TODO: reformulate!

Scenario 1: Constant fitness while walking from $x = 0$ toward $x = 1$

$$\frac{dy}{dx} = g(p, x, y) = p; \qquad y(0) = 0$$

Scenario 2: Fitness improves with distance covered (exposure to fresh air ...)

$$\frac{dy}{dx} = g(p, x, y) = \frac{p}{x+1}; \qquad y(0) = 0$$

Scenario 3: Fitness improves with fitness (self-fulfilling prophecy ...)

$$\frac{dy}{dx} = g(p, x, y) = \frac{p}{y+1}; \qquad y(0) = 0$$

Scenario 4: Fitness improves with fitness while worsening with distance covered

$$\frac{dy}{dx} = g(p, x, y) = \frac{p \cdot x}{y+1}; \qquad y(0) = 0$$

Show correctness of the following symbolic solutions for the four scenarios:

$$\frac{dy}{dx} = p;\ y(0) = 0 \quad \Rightarrow \quad y = p \cdot x$$

$$\frac{dy}{dx} = \frac{p}{x+1};\ y(0) = 0 \quad \Rightarrow \quad y = p \cdot \log(x+1)$$

$$\frac{dy}{dx} = \frac{p}{y+1};\ y(0) = 0 \quad \Rightarrow \quad y = \sqrt{2 \cdot p \cdot x + 1} - 1$$

$$\frac{dy}{dx} = \frac{p \cdot x}{y+1};\ y(0) = 0 \quad \Rightarrow \quad y = \sqrt{1 + p \cdot x^2} - 1\ ,$$

yielding corresponding explicit algebraic models.

E.g.,

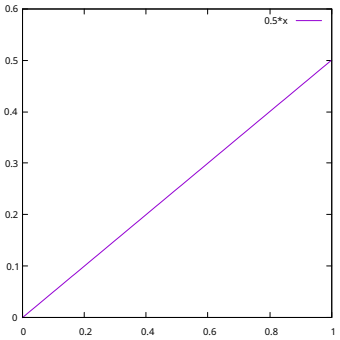
$$y = \sqrt{1 + p \cdot x^2} - 1 + C$$

$$\Downarrow$$

$$y(0) = 0 \quad \Rightarrow \quad \sqrt{1 + p \cdot 0^2} - 1 + C = 0 \quad \Rightarrow \quad C = 0$$

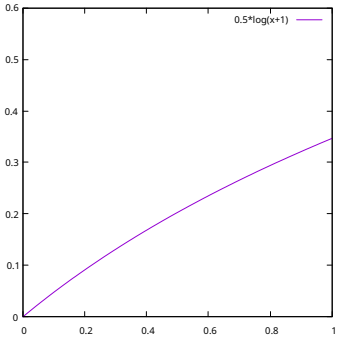
$$\frac{dy}{dx} = \frac{1}{2 \cdot \sqrt{1 + p \cdot x^2}} \cdot 2 \cdot p \cdot x = \frac{p \cdot x}{y+1}$$

Linear in x



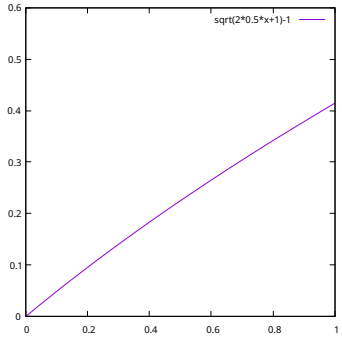
$p \cdot x$

Nonlinear in x



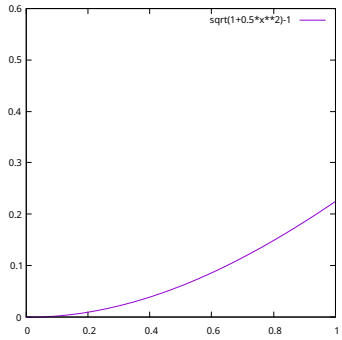
$p \cdot \log(x + 1)$

Nonlinear in x



$\sqrt{2 \cdot p \cdot x + 1} - 1$

Nonlinear in x



$\sqrt{1 + p \cdot x^2} - 1$

The model $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : y = f(p, x)$ is linear in p if

$$f(v + u, x) = f(v, x) + f(u, x) \quad \text{and} \quad f(\alpha \cdot v, x) = \alpha \cdot f(v, x)$$

for all $v, u, \alpha \in \mathbb{R}$.

Models of the form $f(p, x) = a \cdot p + b$ with $a = a(x), b = b(x) \in \mathbb{R}$ are affine in p. Linear functions are affine with $b = 0$.

Roots of affine functions are defined implicitly by linear equations $a \cdot p + b = 0$ implying the explicit equation (solution) $p = -\frac{b}{a}$.

Show that $f(p, x) = p \cdot g(x)$ is linear in p.

Proof:

$$f(v + u, x) = (v + u) \cdot g(x) = v \cdot g(x) + u \cdot g(x) = f(v, x) + f(u, x)$$
$$f(\alpha \cdot v, x) = (\alpha \cdot v) \cdot g(x) = \alpha \cdot (v \cdot g(x)) = \alpha \cdot f(v, x)$$

for all $v, u, \alpha \in \mathbb{R}$.

Wanted:

$$y = f(p, x)$$

for fixed p and given x.

The unknown function $f(p, x)$ is expected to be

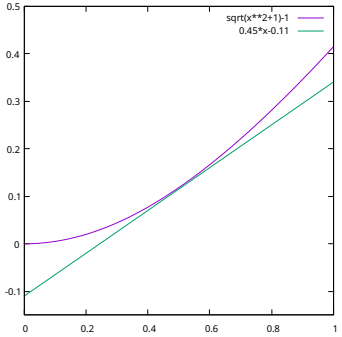
differentiable

and, hence,

continuous

over the domain of interest.

All numerical solution methods to be discussed rely on continuity and differentiability of essentially all parts of the mathematical problem formulation.



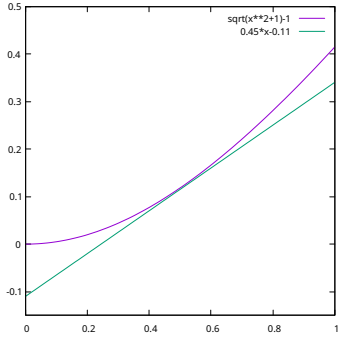
Let p be fixed. The univariate scalar function $f : \mathbb{R} \to \mathbb{R}$, $y = f(x)$, is

right-continuous at $x \in \mathbb{R}$ if

$$\lim_{\Delta x \to 0, \Delta x > 0} f(x + \Delta x) = f(x) \quad .$$

left-continuous at x if

$$\lim_{\Delta x \to 0, \Delta x > 0} f(x - \Delta x) = f(x) \quad .$$



continuous at x if it is both left- and right-continuous at x.

Continuity is a necessary condition for differentiability.

Is $f(x) = \sin(x)$ continuous over its entire domain?

Is $f(x) = |x|$ continuous at $x = 0$?

Is

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x & \text{otherwise} \end{cases}$$

continuous at $x = 1$? What about $x = 0$?

Is

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

left-continuous at $x = 0$? Is it continuous at $x = 0$?

$f(x)$ is right-differentiable at $x \in \mathbb{R}$ if the limit

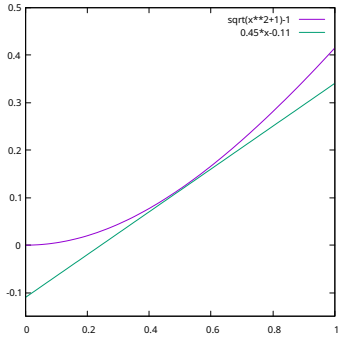
$$\lambda^+ = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

exists (is finite). f is left-differentiable at x if

$$\lambda^- = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x}$$

exists (is finite). f is differentiable at x if it is both left- and right-differentiable with first derivative

$$\lambda^+ = \lambda^- = f' \equiv \frac{df}{dx} \in \mathbb{R} \, .$$



The first derivative is equal to the slope of the tangent (also: tangent of angle spanned with x-axis.

It is continuously differentiable at x if f' is continuous.

All tangents of

$$f(x) = \begin{cases} 2 \cdot x & x < 0 \\ 2 \cdot x + 1 & x >= 0 \end{cases}$$

have slope equal to two. Still, f is not differentiable at the origin ($x = 0$) due to missing left-differentiability.

Note that

$$\lim_{\Delta x \to 0} \left| \frac{f(x + \Delta x) - f(x)}{\Delta x} \right| = \infty$$

if f is not right-differentiable.

Similarly,

$$\lim_{\Delta x \to 0} \left| \frac{f(x) - f(x - \Delta x)}{\Delta x} \right| = \infty$$

if f is not left-differentiable.

Is $f(x) = \sin(x)$ differentiable over its entire domain?

Is $f(x) = |x|$ differentiable at $x = 0$? Is it differentiable at $x = 1$?

Is

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x & \text{otherwise} \end{cases}$$

differentiable at $x = 1$? What about $x = 0$?

Is

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

differentiable at $x = -10^{-10}$? What about $x = 10^{-10}$?

The first derivative of the k-th derivative of a k times continuously differentiable function

$$f : \mathbb{R} \to \mathbb{R}$$

is continuous making f potentially $k + 1$ times differentiable for $k = 1, 2, \ldots$.

$$f' \equiv \frac{df}{dx}$$
$$f'' \equiv \frac{d^2 f}{dx^2} = \frac{df'}{dx}$$
$$f''' \equiv \frac{d^3 f}{dx^3} = \frac{df''}{dx}$$
$$\vdots$$

e.g.,

$$f' = \frac{d \sin(x)}{dx} = \cos(x)$$
$$f'' = \frac{d \cos(x)}{dx} = -\sin(x)$$
$$f''' = \frac{d(-\sin(x))}{dx} = -\cos(x)$$
$$\vdots$$

Disclaimer: All functions considered during this course are assumed to be sufficiently often continuously differentiable. Exceptions are commented on explicitly.

Let $f : \mathbb{R} \to \mathbb{R}$ be sufficiently often differentiable.

f is constant if its derivatives vanish identically for all $x \in \mathbb{R}$, e.g, $f(x) = 42$ is constant over $\mathbb{R}$.

f is (at most) affine if its second and higher derivatives vanish identically for all $x \in \mathbb{R}$, e.g, $f(x) = 42 \cdot x - 24$ is affine over $\mathbb{R}$ while $f(x) = 42 \cdot x$ is linear.

f is (at most) quadratic if its third and higher derivatives vanish identically for all $x \in \mathbb{R}$, e.g, $f(x) = 42 \cdot x^2 - 24 \cdot x + 1$ is quadratic over $\mathbb{R}$.

f is (at most) cubic if its fourth and higher derivatives vanish identically for all $x \in \mathbb{R}$, e.g, $f(x) = 42 \cdot x^3 - 24$ is cubic over $\mathbb{R}$.

etc.

Implicitly defined models $y(x)$ are unknown in the sense of no explicit algebraic solution $y = f(x)$ being available.

If the model is continuously differentiable up to some order in a neighborhood of the point x of interest, then Taylor's theorem provides a way to approximate the model as a weighted sum of its derivatives of lower orders.

Most numerical approximation algorithms rely on this observation.

E.g., an unknown function $y(x)$ could be approximated locally by its tangent, which is also known as linearization. The "less linear" $y(x)$ actually is, the "less accurate" this approach becomes when moving away from the given x. Similar remarks apply to quadratic, cubic, etc. approximations by (truncated) Taylor expansion of order two, three, etc..

Let $f : \mathbb{R} \to \mathbb{R}$ be n-times continuously differentiable.

Given the value of $f(x)$ at some point $x \in \mathbb{R}$ the function value $f(x + \Delta x)$, $\Delta x \in \mathbb{R}$, at a neighboring point can be approximated by a Taylor expansion as

$$f(x + \Delta x) \approx_{\mathcal{O}(|\Delta x|^n)} f(x) + \sum_{k=1}^{n-1} \frac{1}{k!} \cdot \frac{d^k f}{dx^k}(x) \cdot \Delta x^k \ .$$

Throughout this course we assume convergence of the Taylor expansion for $k \to \infty$ to the true value of $f(x + \Delta x)$ within all subdomains of interest, which is not the case for arbitrary functions.

For $n = 4$ we get

$$f(x + \Delta x) = f(x) + f' \cdot \Delta x + \frac{1}{2} \cdot f'' \cdot \Delta x^2 + \frac{1}{6} \cdot f''' \cdot \Delta x^3 + \mathcal{O}(|\Delta x|^4) \ .$$

Given two functions $f(x)$ and $g(x)$ the notation

$$f = \mathcal{O}(g)$$

implies that f grows up to a constant factor as g, that is,

$$\exists C > 0 \in \mathbb{R} : |f(x)| \leq C \cdot |g(x)|$$

for all x within the shared domains of f and g.

E.g, $f(x) = \mathcal{O}(x^2)$ implies that $f(x)$ does not grow faster than $C \cdot x^2$ for some constant $C > 0$.

Although,

$$f(x) = \mathcal{O}(x) \quad \Rightarrow \quad f(x) = \mathcal{O}(x^2) \quad \Rightarrow \quad f(x) = \mathcal{O}(x^3) \quad \ldots$$

we are interested in the lowest upper bound (*supremum*).

The $\mathcal{O}(\cdot)$ notation is used in the the following to estimate computational costs of numerical methods as well as errors induced by them.

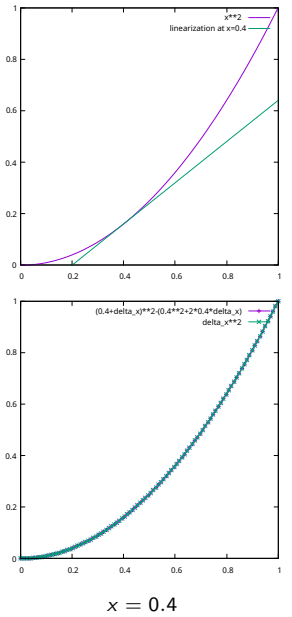
E.g., an approximation of the function $y = f(x) = x^2$ at a given point x by its linearization

$$\begin{aligned} \bar{y} = \bar{f}(\Delta x) &\equiv f(x) + f' \cdot \Delta x \\ &= x^2 + 2 \cdot x \cdot \Delta x \end{aligned}$$

induces an error of order Δx^2 as

$$\begin{aligned} & y(x + \Delta x) - \bar{y} \\ &= (x + \Delta x)^2 - (x^2 + 2 \cdot x \cdot \Delta x) \\ &= \Delta x^2 = \mathcal{O}(\Delta x^2) \end{aligned}$$

with $C = 1$.



$x = 0.4$

Got It?

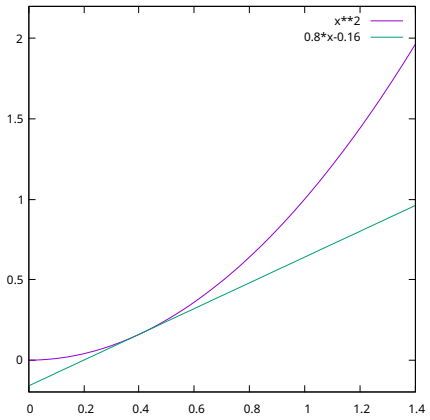
Use `gnuplot` to visualize the error Δx^2.

Note that $x + \Delta x = 1.4$ at $x = 0.4$ for $\Delta x = 1$. Moreover,

$$0.8 \cdot (x + \Delta x) - 0.16 = 0.96$$

and

$$(x + \Delta x)^2 = 1.96 \, .$$



Hands On! (Exercise 2)

Write an SNC++ script to validate the decreasing absolute error of a truncated Taylor expansion of increasing order $k = 1, 2, \ldots$ for $y = \sin(x)$.

Repeat for $y = e^x$.

Notes:

- Declare and initialize x and Δx, e.g., $x = 1$ and $\Delta x = 0.1$.
- Print difference between $f(x + \Delta x)$ and truncated Taylor expansion for increasing orders, e.g., $f(x + \Delta x) - (f(x) + f' \cdot \Delta x + \frac{1}{2} \cdot f'' \cdot \Delta x^2)$ for the error of a second-order truncated Taylor expansion.
- Explore behavior for other values of x and Δx.
- Explore behavior for other functions.

→ exercises/2/

Outline

Simulation
Executive Summary

- We consider explicit differential models with symbolic solutions yielding explicit algebraic models. The latter can be used for validation of approximate solutions obtained by the numerical methods to be introduced.
- Essential mathematical terminology is introduced along the way.
- Forward finite difference approximation of the derivative term in the differential models yields the iterative explicit Euler method.
- Backward finite differences yield the iterative implicit Euler method. An [implicit] algebraic equation needs to be solved in each iteration. Bisection and Newton(-Raphson) methods are employed.
- Central finite differences are used for parameter sensitivity analysis.

Outline

Explicit Euler Method
Executive Summary

- The explicit Euler method is applicable to all four scenarios.
- Local linearization (first-order truncated Taylor expansion with step size Δx) yields a numerical approximation of $\frac{dy}{dx}$ by a forward finite difference.
- A sequence of explicit algebraic equations is evaluated to integrate the differential model numerically from $x = 0$ to $x = 1$.
- The term "numerical integration" refers to the approximation of the value of the model $y(x)$ at some point x based on the given initial value $y(0)$ and by chaining a (large) number of local approximations of $y(x)$, e.g., linearizations.
- An error of order $\geq \mathcal{O}(\Delta x^2)$ is induced.

Approximate First Derivative
Forward Finite Difference

Truncation of the Taylor expansion of

$$y : \mathbb{R} \to \mathbb{R} : y = y(x)$$

in direction $0 < \Delta x \ll 1$ after the first-order term yields

$$y(x + \Delta x) = y(x) + \frac{dy}{dx} \cdot \Delta x + \mathcal{O}(\Delta x^2) \qquad \text{(linearization)}$$

and, hence, a first-order accurate approximation of the first derivative at x as

$$y' \equiv \frac{dy}{dx} = \frac{y(x + \Delta x) - y(x)}{\Delta x} + \mathcal{O}(\Delta x) \; .$$

The error of this forward finite difference approximation is of the order Δx.

→ Whiteboard: graphical illustration

Explicit Euler Method
Derivation

The explicit Euler method (also referred to as explicit Euler integration) replaces the derivative in the ordinary differential equation (ODE)

$$\frac{dy}{dx} = g(p, x, y(p, x))$$

with a forward finite difference in direction $0 < \Delta x \ll 1$ yielding

$$\frac{y(p, x + \Delta x) - y(p, x)}{\Delta x} = g(p, x, y(p, x))$$

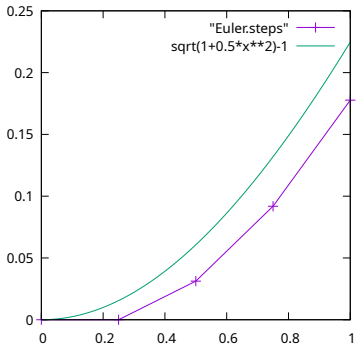
and, hence, the iterative approximation of the solution as

$$y(p, x + \Delta x) = y(p, x) + \Delta x \cdot g(p, x, y(p, x))$$

for given $y(p, 0) = y^0(p)$. In the given Scenarios 1–4, $y^0 = 0$ is independent of p.

The explicit Euler method evaluates the following sequence of explicit algebraic equations

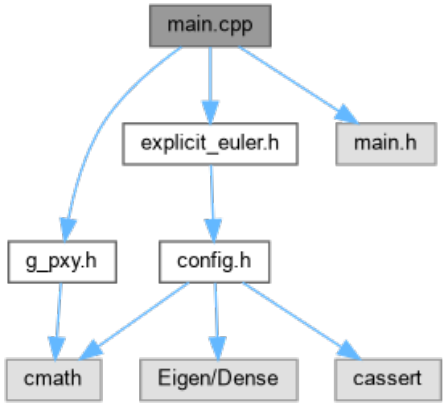
$$y^{i+1} = y^i + \frac{x^m}{m} \cdot g(p, x^i, y^i) \,,$$
$$i = 0, \ldots, m-1 \,,$$



Scenario 4

for given initial value $y^0(p)$ at $x^0 = 0$ and number of integration steps $m > 0$.


```
1  int m=100; // number of integration steps
2
3  /// explicit Euler simulation of differential model (note: doxygen comment)
4  template<typename T>
5  T f(T p, T x) {
6    // condition
7    assert(x>0);
8    // integration step size
9    T delta_x=x/m;
10   // initial position fixed to zero
11   x=0;
12   // initial state fixed to zero
13   T y=0;
14   int i=0;
15   while (i<m) {
16     // explicit Euler step
17     y=y+delta_x*g(p,x,y);
18     x=x+delta_x;
19     i=i+1;
20   }
21   return y;
22 }
```

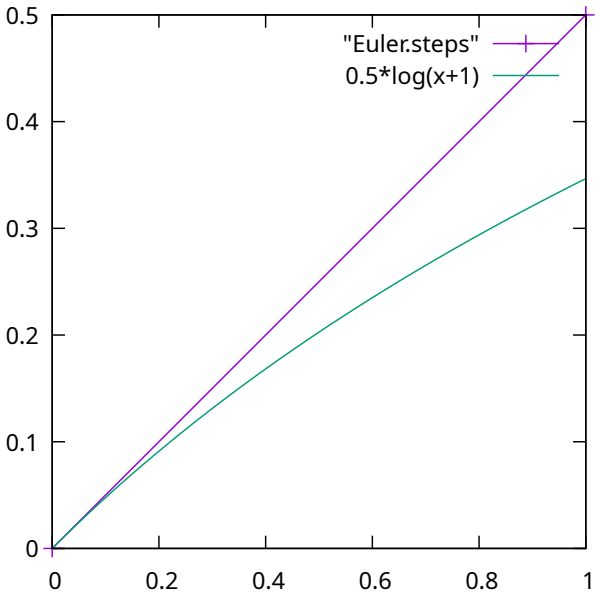
→ 1D/simulation/differential/explicit_euler/

An absolute error

$$\begin{aligned} & y(p, x^i + \Delta x) - y^{i+1} \\ & = \underbrace{\underbrace{y(p,x^i)}_{=y^i} + \Delta x \cdot \underbrace{y'(p,x^i)}_{=g(p,x^i,y^i)} + \frac{1}{2} \cdot \Delta x^2 \cdot y''(p,x^i) + \mathcal{O}(\Delta x^3)}_{\text{Taylor expansion}} \\ & \quad - \underbrace{\left(y^i + \Delta x \cdot g(p,x^i,y^i)\right)}_{\text{explicit Euler step}} \\ & = \frac{1}{2} \cdot \Delta x^2 \cdot y''(p,x^i) + \mathcal{O}(\Delta x^3) \end{aligned}$$

of order Δx^2 is induced by each explicit Euler step due to local linearization and assuming convergence of the Taylor expansion.

This error is potentially accumulated over subsequent steps.



An absolute error Δy^i on input to an explicit Euler step

$$y^{i+1} = y^i + \Delta x \cdot g(p, x^i, y^i) \equiv \tilde{g}(p, x^i, y^i, \Delta x)$$

yields

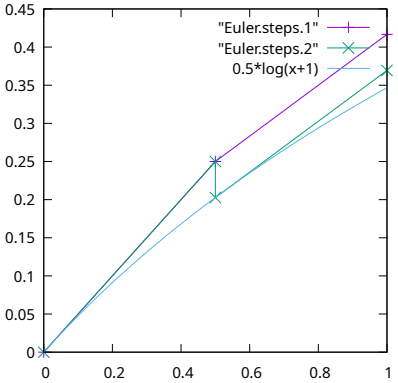
$$y^{i+1} + \Delta y^{i+1} = y^i + \Delta y^i + \Delta x \cdot g(p, x^i, y^i + \Delta y^i)$$

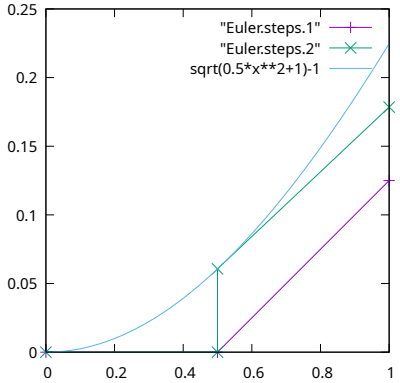
where the resulting error

$$\Delta y^{i+1} \approx_1 \frac{d\tilde{g}}{dy}(p, x^i, y^i, \Delta x) \cdot \Delta y^i = \Delta y^i + \Delta x \cdot \frac{dg}{dy}(p, x^i, y^i) \cdot \Delta y^i \ .$$

If $\frac{dg}{dy} = 0$, then Δy^i is not propagated (see Scenarios 1 and 2). Still an error is induced due to local linearization.

First two explicit Euler steps for Scenarios 2 (left) and 4 (right).





Both scenarios yield errors due to local linearization (Euler.step.2). The error from the first step is propagated[6] in Scenario 4 but not in Scenario 2 (Euler.steps.1).

[6] The purple and green lines of the second step are not parallel.

Plot the solution of explicit Euler integration of

$$\frac{dy}{dx} = g(p, x, y)$$

from $x = 0$ to $x = 1$ for $y(0) = 0$. Use gnuplot for visualization.

Plot the evolution of the algebraic solution over the same interval for comparison.

- Make a copy of the code required for explicit Euler integration of all four sample scenarios, including config.h, explicit_euler.h, f_p*.h, g_p*.h, main.h, p/main.cpp, px/main.cpp, py/main.cpp, pxy/main.cpp.
- Build, e.g.,

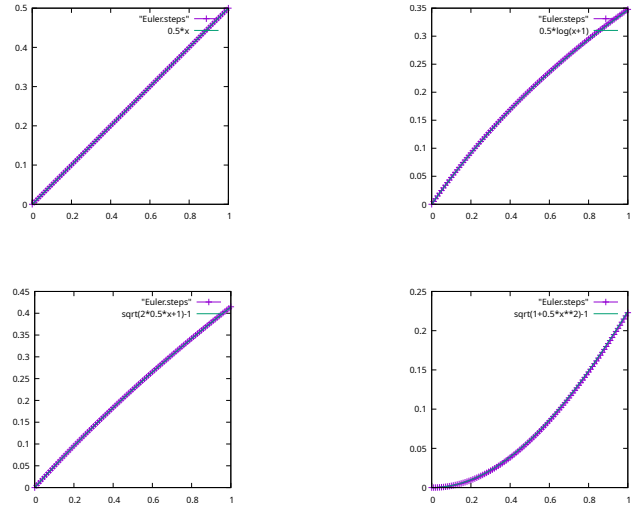
```
cd pxy
g++ -O3 -I$(EIGEN_DIR) -I.. main.cpp -o main.exe
```

  and run `./main.exe` to see, e.g., `y=0.222964`.
- Augment the function `T f(T p, T x)` in explicit_euler.h with statements for writing all points (x, y) computed by explicit Euler method into the text file Euler.steps. Rebuild and run to generate Euler.steps.
- Run gnuplot. Plot both the contents of Euler.steps and the graph of the corresponding algebraic model, e.g.,

```
plot "Euler.steps" with linespoints, sqrt(1+0.5*x**2)-1
```

→ exercises/3/

You should see the following.[7]



[7] assuming $m = 100$ and use of "set size square" in gnuplot

Outline

Implicit Euler Method

Executive Summary

- The implicit Euler method is applicable to all four scenarios.
- Local textcolorrwth-bluelinearization (first-order truncated Taylor expansion with step size $-\Delta x$) yields a numerical approximation of $\frac{dy}{dx}$ by a backward finite difference.
- A sequence of [implicit] algebraic equations is solved to integrate the differential model from $x = 0$ to $x = 1$.
- The corresponding roots are computed by the bisection or the Newton[-Raphson] methods.
- An error of order $\geq \mathcal{O}(\Delta x^2)$ is induced.
- The implicit Euler method can remain stable (with bounded error) for larger step sizes than the explicit Euler method.

Approximate First Derivative

Backward Finite Difference

Truncation of the Taylor expansion of

$$y : \mathbb{R} \to \mathbb{R} : y = y(x)$$

in direction $-1 \ll -\Delta x < 0$ after the first-order term yields

$$y(x - \Delta x) = y(x) - \frac{dy}{dx} \cdot \Delta x + \mathcal{O}(\Delta x^2)$$

and, hence, a first-order accurate approximation of the Jacobian at x as

$$y' \equiv \frac{dy}{dx} = \frac{y(x) - y(x - \Delta x)}{\Delta x} + \mathcal{O}(\Delta x) \; .$$

The error of this backward finite difference approximation is of the order Δx.

→ Whiteboard: graphical illustration

Implicit Euler Method

Derivation

The implicit Euler method replaces the derivative in the ODE with a backward finite difference yielding

$$\frac{y(p, x) - y(p, x - \Delta x)}{\Delta x} = g(p, x, y(p, x))$$

and, hence, $y(p, x)$ as the solution of the algebraic equation

$$y(p, x) - y(p, x - \Delta x) - \Delta x \cdot g(p, x, y(p, x)) = 0 \; .$$

The solution is approximated iteratively for given $y(p, 0) = y^0(p)$ and $\Delta x > 0$. In the given Scenarios 1–4, $y^0 = 0$ is independent of p.

The error of order Δx^2 of the implicit Euler method can remain bounded for larger steps sizes than the explicit Euler method. This improved stability comes at a higher computational cost.

Implicit Euler Method

Algorithm

The implicit Euler method iteratively solves the following nonlinear equation (also: implicit Euler equation):

$$r(y^i) \equiv y^i - y^{i-1} - \frac{x^m}{m} \cdot g(p, x^i, y^i) = 0 \; , \quad i = 0, \ldots, m - 1 \; .$$

For given initial value $y^0(p)$ at $x^0 = 0$ and number of integration steps $m > 0$, the method approximates a root of the residual defined by the left-hand side.

The residual is implemented as follows:

```
1 /// residual of implicit Euler equation
2 template<typename T>
3 T r(T p, T x, T y, T y_prev, T delta_x) {
4   // conditions
5   assert(x>=0);
6   assert(delta_x>0);
7   // residual
8   return y-y_prev-delta_x*g(p,x,y);
9 }
```


Implicit Euler Method

Implementation

```
1  int m=100; // number of integration steps
2
3  /// implicit Euler simulation of differential model
4  template<typename T>
5  T f(T p, T x) {
6    // condition
7    assert(x>=0);
8    // integration step size
9    T delta_x=x/m;
10   // initial position fixed to zero
11   x=0;
12   // initial state fixed to zero
13   T y=0;
14   int i=0;
15   while (i<m) {
16     // implicit Euler step
17     y=bisection(p,x,y,delta_x); // root of implicit Euler equation
18     x=x+delta_x;
19     i=i+1;
20   }
21   return y;
22 }
```

We consider two methods for solving the algebraic implicit Euler equation, namely, the

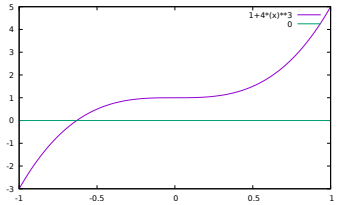
- bisection

and

- Newton

methods.

Let $r = r(y)$ be continuous within a neighborhood $y + \Delta y$ of y taking values $r(y)$ and $r(y + \Delta y)$ at the endpoints of the interval.



Then f takes all values between $r(y)$ and $r(y + \Delta y)$ over the same interval.

If $r(y)$ and $r(y + \Delta y)$ have different signs, then f has a root within the interval bounded by y and $y + \Delta y$, i.e,

The simplest possible root finding algorithm follows from iterative / recursive bisection of the interval $[y, y + \Delta y]$.

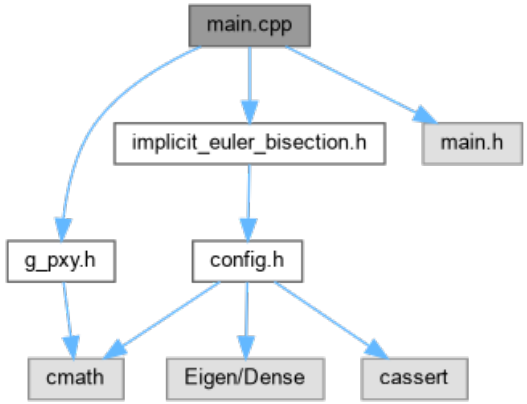
Unfortunately, the bisection algorithm converges only linearly (too slow) due to the (worst case) error halved in each step. Neither does it generalize to higher dimensions. Hence, we are going to investigate superior alternatives.


```
1  /// implicit Euler system solved by bisection
2  template<typename T>
3  T bisection(T p, T x, T y, T delta_x) {
4    // conditions
5    assert(x>=0);
6    assert(delta_x>0);
7    // search interval (smarter search required in general)
8    T y_prev=y, y_up=y+2;
9    y=y-2;
10   // bisection to decrease width of search interval
11   while (fabs(y-y_up)>eps) {
12     // midpoint of search interval
13     T y_mid=(y_up+y)/2;
14     // residual at lower bound
15     T r_=r(p,x,y,y_prev,delta_x),
16     // residual at midpoint
17     r_mid=r(p,x,y_mid,y_prev,delta_x);
18     // collapse search interval if root found
19     if (fabs(r_mid)<eps) {
20       y=y_up=y_mid;
21     }
```



```
22     // continue bisection if root not found
23     else if ((r_<=0&&r_mid<=0)||(r_>=0&&r_mid>=0)) {
24       // search in right half of sign switches there
25       y=y_mid;
26     }
27     else {
28       // search in left half otherwise
29       y_up=y_mid;
30     }
31   }
32   return y;
33 }
```

→ Whiteboard: graphical illustration



→ 1D/simulation/differential/implicit_euler/bisection/

The solution of linear equations (to find their roots) amounts to simple scalar division. The solution of nonlinear equations can be challenging.

Many numerical methods for nonlinear problems, such as $r(y) = 0$, rely on local replacement of the target function with an affine (in Δy) approximation derived from the truncated Taylor expansion and "hoping" that

$$r(y + \Delta y) \approx r(y) + r' \cdot \Delta y \ ,$$

with a reasonably small error.

$r' = r'(y)$ for Scenarios 3 and 4 where $g = g(y)$.

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

Further requirements must be satisfied.

Consider the nonlinear equation $r(y) = 0$ at some (starting) point y.
Building on the assumption that $r(y + \Delta y) \approx r(y) + r' \cdot \Delta y$ the root finding problem for r can be replaced locally (at the current y) by the root finding problem for the linearization

$$\bar{r}(\Delta y) = r(y) + r' \cdot \Delta y \ .$$

The right-hand side is a straight line intersecting the r-axis in $(\Delta y = 0, \bar{r}(\Delta y) = r(y))$. Solution of

$$\bar{r}(\Delta y) = r(y) + r' \cdot \Delta y = 0$$

for Δy yields

$$\Delta y = -\frac{r(y)}{r'}$$

implying $r(y + \Delta y) \approx 0$.

→ Whiteboard: graphical illustration

If the new iterate is not close enough to the root, i.e., $|r(y + \Delta y)| > \epsilon$ for some measure of accuracy of the numerical approximation $\epsilon > 0$, then it becomes the starting point for the next iteration yielding the recurrence

$$y^{i+1} = y^i - \frac{r(y^i)}{r'(y^i)} \quad \text{for } i = 0, \ldots .$$

Convergence of the Newton method is not guaranteed in general. See below for a discussion of local contractiveness of the fixpoint iteration as a sufficient (not necessary) condition for convergence.

Newton Method
Implementation I

```
1  /// residual of implicit Euler equation
2  template<typename T>
3  T r(T p, T x, T y, T y_prev, T delta_x) {
4    // conditions
5    assert(x>=0);
6    assert(delta_x>0);
7    // residual
8    return y-y_prev-delta_x*g(p,x,y);
9  }
10
11 /// derivative of residual implicit Euler system
12 template<typename T>
13 T dr_dy(T p, T x, T y, T delta_x) {
14   // conditions
15   assert(x>=0);
16   assert(delta_x>0);
17   // derivative of residual
18   return 1-delta_x*dg_dy(p,x,y);
19 }
20
21
```


Newton Method
Implementation II

```
22 /// implicit Euler system solved by Newton method
23 template<typename T>
24 T newton(T p, T x, T y, T delta_x) {
25   // conditions
26   assert(x>=0);
27   assert(delta_x>0);
28   // state computed by previous implicit Euler step
29   T y_prev=y;
30   do {
31     // Newton step
32     y=y-r(p,x,y,y_prev,delta_x)/dr_dy(p,x,y,delta_x);
33   } while (fabs(r(p,x,y,y_prev,delta_x))>eps);
34   return y;
35 }
```


Newton Method
Symbolic Differentiation

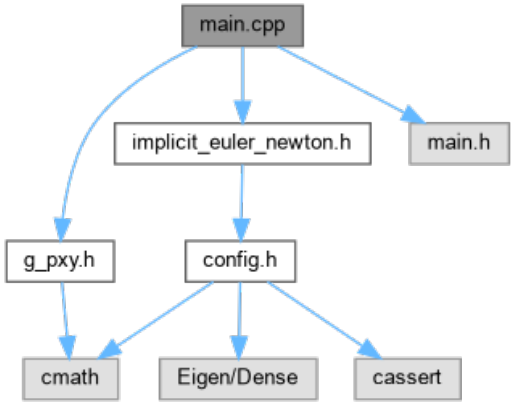
$$r'(y) = 1 - \Delta x \cdot g'(p,x,y)$$

yields

$$\begin{aligned}
g(p,x,y) &= p & \Rightarrow \quad r'(y) &= 1 \\
g(p,x,y) &= \frac{p}{x+1} & \Rightarrow \quad r'(y) &= 1 \\
g(p,x,y) &= \frac{p}{y+1} & \Rightarrow \quad r'(y) &= 1 + \Delta x \cdot \frac{p}{(y+1)^2} \\
g(p,x,y) &= \frac{p \cdot x}{y+1} & \Rightarrow \quad r'(y) &= 1 + \Delta x \cdot \frac{p \cdot x}{(y+1)^2}
\end{aligned}$$

Implicit Euler Method using Newton Method
Implementation

```
1  int m=100; // number of integration steps
2
3  /// implicit Euler simulation of differential model
4  template<typename T>
5  T f(T p, T x) {
6    // condition
7    assert(x>=0);
8    // integration step size
9    T delta_x=x/m;
10   // initial position fixed to zero
11   x=0;
12   // initial state fixed to zero
13   T y=0;
14   int i=0;
15   while (i<m) {
16     // implicit Euler step
17     y=newton(p,x,y,delta_x); // root of implicit Euler equation
18     x=x+delta_x;
19     i=i+1;
20   }
21   return y;
22 }
```



→ 1D/simulation/differential/implicit_euler/newton/

The Newton method can be regarded as a fixed point iteration

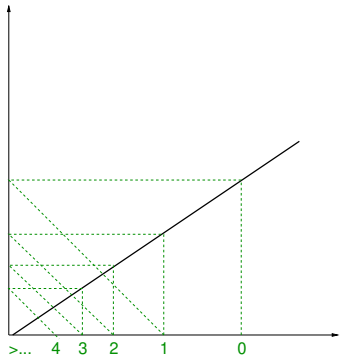
$$y := \check{r}(y) = y - \frac{r(y)}{r'(y)} .$$

If at the solution

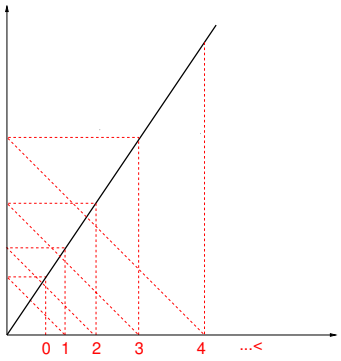
$$|\check{r}'| \equiv \left|\frac{d\check{r}}{dy}\right| = \left|1 - \frac{r'(y)}{r'(y)} + \frac{r(y)\cdot r''(y)}{r'(y)^2}\right| = \left|\frac{r(y)\cdot r''(y)}{r'(y)^2}\right| < 1 ,$$

then there exists a neighborhood containing values of y for which the fixed-point iteration converges to this solution.

The convergence rate of a fixed-point iteration grows linearly with decreasing values of $|\check{r}'|$. For $\check{r}' = 0$ we get at least quadratic convergence; cubic for $\check{r}' = \check{r}'' = 0$ and so forth. For linear problems (constant $r' \Rightarrow r'' = 0$), the Newton method requires a single iteration to converge.



convergence for $\left|\frac{d\check{r}}{dy}\right| \leq 1$



divergence for $\left|\frac{d\check{r}}{dy}\right| > 1$

Implement root finding for the function

$$y = \sin(x)$$

by the the bisection and Newton methods. Write corresponding SNC++ scripts.

For the Newton method, keep track of local contractiveness by reporting the value of

$$\left|\frac{d\check{r}}{dx}\right|$$

in every iteration.

Experiment with different starting points and initial search intervals.

- Copy and paste the bisection algorithm from the sample code and modify the signature to match that of `T f(T x)`.
- Write the first, `T df_dx(T x)`, and second, `T ddf_dx_dx(T x)`, derivatives of the function `T f(T x)` that implements $y = f(x) = \sin(x)$.
- Implement the Newton method, `T newton(T x)`, with convergence defined as $|f(x)|$ falling below some $0 < \epsilon \ll 1$, e.g., $\epsilon = 10^{-7}$.
- In each Newton iteration, print the value of

$$\left|\frac{d\check{f}}{dx}\right| = \left|\frac{f(x) \cdot f''(x)}{f'(x)^2}\right| .$$

→ exercises/4/

Simulation yields

$$y[= y(p, x)] = y(p) .$$

Parameter sensitivity analysis aims to quantify the impact of (tiny) errors in p on the result of the simulation.

The first derivative

$$\frac{dy}{dp} \in \mathbb{R}$$

is computed for this purpose.

Large absolute values of $\frac{dy}{dp}$ indicate high sensitivity of the solution with respect to (to be abbreviated in the following as wrt.) likely errors in the free parameter p.

The central finite difference

$$\tilde{y}'(\Delta p) = \frac{y(p + \Delta p) - y(p - \Delta p)}{2 \cdot \Delta p} \tag{1}$$

yields a second-order accurate approximation of the first derivative, that is,

$$y' = \tilde{y}'(\Delta p) + \mathcal{O}(\Delta p^2) .$$

→ Whiteboard: graphical illustration

The error of central finite difference approximation is of the order Δp^2. Hence, central finite differences are more accurate that forward or backward finite differences as $\Delta p^2 \ll \Delta p$ for $0 < \Delta p \ll 1$.

Subtraction of the second-order Taylor expansion in direction $-\Delta p < 0$

$$y(p - \Delta p) = y(p) - y' \cdot \Delta p + \frac{1}{2} \cdot y'' \cdot \Delta p^2 - \mathcal{O}(\Delta p^3)$$

from the second-order Taylor expansion in direction $\Delta p > 0$

$$y(p + \Delta p) = y(p) + y' \cdot \Delta p + \frac{1}{2} \cdot y'' \cdot \Delta p^2 + \mathcal{O}(\Delta p^3)$$

yields

$$y(p + \Delta p) - y(p - \Delta p) = 2 \cdot y' \cdot \Delta p + \mathcal{O}(\Delta p^3)$$

and, hence, Equation (1).

Hands On! (Exercise 5)

Implement the computation of the parameter sensitivity

$$\frac{dy}{dp} \in \mathbb{R}$$

for $y(1)$ approximated by implicit (with Newton method for root finding) as well as explicit Euler integration of

$$\frac{dy}{dx} = g(p, x, y), \; y(0) = 0,$$

for $g(p, x, y)$ as in Scenario 4. Use central finite differences.

Compare with the parameter sensitivity of the algebraic model.

Check what happens, if bisection is used for root finding as part of the implicit Euler method.

Exercise 5

Notes

- Add

```
1 double delta_p=1e-7;
2 double y_l=f(p-delta_p,x);
3 double y_r=f(p+delta_p,x);
4 cout << "dy/dp=" << (y_r-y_l)/(2*delta_p) << endl;
```

  to main.h.
- Optional: Isolate the code required for the simulation of Scenario 4.
- You should see the following results:
  - algebraic model: $\frac{dy}{dp} = 0.408248$
  - differential model with explicit Euler method: $\frac{dy}{dp} = 0.405693$
  - differential model with implicit Euler method using Newton method for root finding: $\frac{dy}{dp} = 0.403989$

  If the implicit Euler method is combined with bisection for root finding, then the wrong result $\frac{dy}{dp} = 0.596046$ is due to nondifferentiability.

→ exercises/5/*

Outline

Optimization

Executive Summary

- The parameter estimation (also: calibration) problem can be considered as an unconstrained nonlinear optimization problem.
- General-purpose nonlinear optimization methods such as the gradient descent and Newton methods can be employed for its solutions. Conditions apply.
- Linear (in the parameter to be estimated) models (Scenario 1 and Scenario 2) enable the use of more efficient linear regression algorithms such as the normal equation and Householder methods.
- Linearization and linear regression can be combined for the calibration of nonlinear models (Scenario 3 and Scenario 4). Conditions apply.

We consider continuous models given by numerical solutions of the four initial value problems described by Scenarios 1–4, that is,

$$\frac{dy}{dx} = p;\ y(0) = 0 \quad \Rightarrow \quad y = p \cdot x$$

$$\frac{dy}{dx} = \frac{p}{x+1};\ y(0) = 0 \quad \Rightarrow \quad y = p \cdot \log(x+1)$$

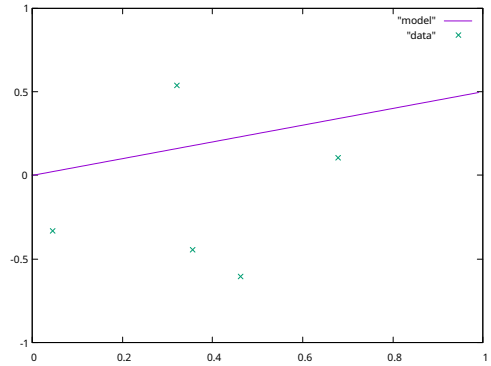
$$\frac{dy}{dx} = \frac{p}{y+1};\ y(0) = 0 \quad \Rightarrow \quad y = \sqrt{2 \cdot p \cdot x + 1} - 1$$

$$\frac{dy}{dx} = \frac{p \cdot x}{y+1};\ y(0) = 0 \quad \Rightarrow \quad y = \sqrt{1 + p \cdot x^2} - 1$$

Euler integration methods are employed. Symbolic solutions can be derived for validation.

We consider randomly generated data

$$(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}}$$
$$\mathbf{x} = (x_i)_{i=0,\ldots,n_{\text{obs}}-1}$$
$$\mathbf{y} = (y_i)_{i=0,\ldots,n_{\text{obs}}-1}\ .$$

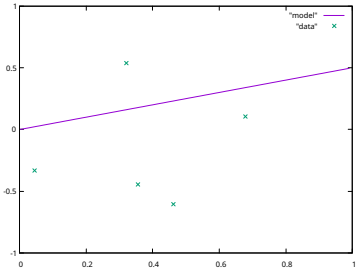


The residual vector

$$\mathbf{r} = \mathbf{r}(p, \mathbf{x}, \mathbf{y}) \equiv (f(p, x_i) - y_i)_{i=0,\ldots,n_{\text{obs}}-1}$$

yields a scalar least squares error as

$$e = e(p, \mathbf{x}, \mathbf{y}) \equiv \|\mathbf{r}\|_2^2 = \mathbf{r}^T \cdot \mathbf{r} = \sum_{i=0}^{n_{\text{obs}}-1} r_i^2 = \sum_{i=0}^{n_{\text{obs}}-1} (f(p, x_i) - y_i)^2$$



We aim to minimize the least-squares error of the model predictions wrt. the given data, that is, $\min_p e$, where $e = e(p, \mathbf{x}, \mathbf{y})$ is implemented as

```
1  /// least squares error of model predictions relative to observations
2  template<typename T>
3  T e(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs);
6    assert(xobs.size()==yobs.size());
7    T e=0;
8    int i=0;
9    while (i<nobs) {
10     e=e+pow(f(p,xobs(i))−yobs(i),2);
11     i=i+1;
12   }
13   return e;
14 }
```

The objective

$$e = e(p, \mathbf{x}, \mathbf{y}) = \sum_{i=0}^{n_{\text{obs}}-1} (f(p, x_i) - y_i)^2$$

depends on the model parameter $p \in \mathbb{R}$ and on given observations $\mathbf{x} \times \mathbf{y} \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}}$.

Most parameter estimation methods to be discussed require derivatives of the objective wrt. the model parameter. Program variables used to hold the values of e and p are referred to as active. In the same context the program variables for $\mathbf{x}$ and $\mathbf{y}$ are passive.

Data error analysis (to be discussed later) will be based on derivatives of the optimal p wrt. $\mathbf{x}$ and $\mathbf{y}$ making all three program variables active despite the input instance of p being passive.

The objective

$$e = \sum_{i=0}^{n_{\text{obs}}-1} (f(p, x_i) - y_i)^2$$

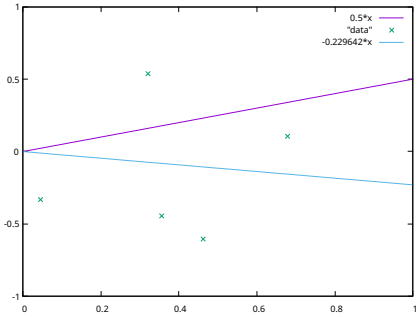
has a (unique[a]) stationary point at

$$e' \equiv \frac{de}{dp} = 0$$

(necessary (first-order) optimality condition). This stationary point is a (local) minimum as

$$e'' \equiv \frac{d^2e}{dp^2} > 0$$

(second-order optimality condition).



e.g., $f(p, x_i) = p \cdot x_i$ for given $\mathbf{x}, \mathbf{y}$.

[a] ... if e is *strictly convex* or *strictly concave*

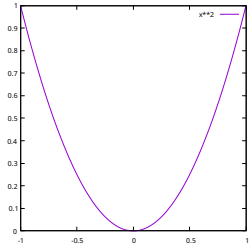
Let $f : \mathbb{R} \to \mathbb{R}$ be continuous over $[a, b] \subset \mathbb{R}$. Then f is [strictly] convex if

$$\forall x_0, x_1 \in [a, b] : f\left(\frac{x_0 + x_1}{2}\right) \ [<] \ \leq \frac{f(x_0) + f(x_1)}{2}$$

(points of all secants above the graph of f)

Let $f : \mathbb{R} \to \mathbb{R}$ be twice differentiable over $[a, b] \subset \mathbb{R}$. Then f is [strictly] convex if $f'' \ [>] \ \geq 0$ for all $x \in [a, b]$.

Examples: $f(x) = x^2$ and $f(x) = e^x$ are strictly convex over $\mathbb{R}$; $f(x) = 42 \cdot x$ is (not strictly) convex over $\mathbb{R}$.



Let $f : \mathbb{R} \to \mathbb{R}$ be continuous over $[a, b] \subset \mathbb{R}$.

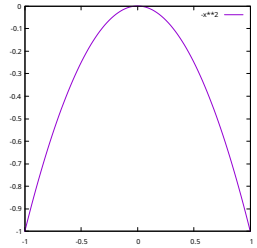
f is [strictly] concave if

$$\forall x_0, x_1 \in [a, b] : f\left(\frac{x_0 + x_1}{2}\right) \ [>] \ \geq \frac{f(x_0) + f(x_1)}{2}$$

(points of all secants below the graph of f)

Let $f : \mathbb{R} \to \mathbb{R}$ be twice differentiable over $[a, b] \subset \mathbb{R}$. Then f is [strictly] concave if $f'' \ [<] \ \leq 0$ for all $x \in [a, b]$.

Examples: $f(x) = -x^2$ and $f(x) = -e^x$ are strictly concave over $\mathbb{R}$; $f(x) = 42 \cdot x$ is (not strictly) concave over $\mathbb{R}$.



Got It?

- Over which of its subdomains is $\sin(x)$ strictly convex / concave?
- Over which of its subdomains is $\cos(x)$ strictly convex / concave?
- Is $|x|$ strictly convex?
- For which values of $p \geq 0$ is the following function convex and differentiable at $x = 0$?

$$f(x) = \begin{cases} -x & x < -p \\ x & x > p \\ p & -p \leq x \leq p \end{cases}.$$

  Is it also strictly convex?
- For which values of $p \geq 0$ is the following function strictly concave and differentiable over it entire domain?

$$f(x) = \begin{cases} -p & -p \leq x \leq p \\ -x^2 & (x < -p) \wedge (x > p) \end{cases}.$$

Outline

Bisection Method
Derivation

Apart from grid search (see Exercise 1), the computation of a (in this course "the" due to uniqueness) root

$$e'(p) = \frac{de}{dp}(p) = 0 \in \mathbb{R}$$

of the first derivative of the objective e wrt. the parameter p, to satisfy the necessary first-order optimality criterion, turns out to be simplest method.

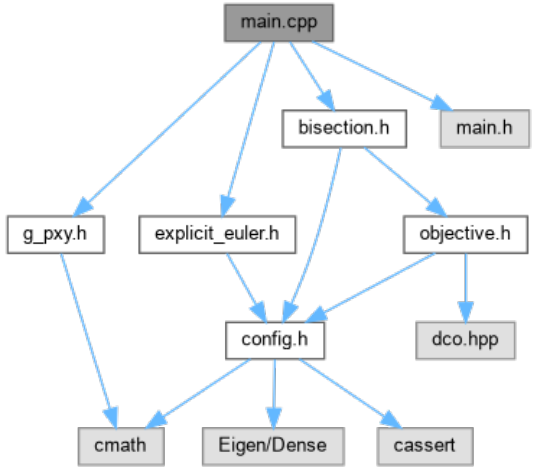
Knowing $\underline{p} \in \mathbb{R}$ and $\overline{p} \in \mathbb{R}$, such that $e'(\underline{p})$ and $e'(\overline{p})$ exhibit different signs, a (the) root can be approximated iteratively using bisection of the interval $[\underline{p}, \overline{p}]$ as described in the context of the implicit Euler method.

Convergence of this method is slow. It does not generalize to the vector case.

Bisection Method
Implementation I

```
1  /// Stationary point of objective wrt. model parameter by bisection
2  template<typename T>
3  T fit(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs);
6    assert(xobs.size()==yobs.size());
7    // search interval (smarter search required in general)
8    T p_up=p+10;
9    p=p-10;
10   // iterative bisection to decrease width of search interval
11   while (fabs(p-p_up)>eps) {
12     // midpoint of search interval
13     T p_mid=(p_up+p)/2;
14     // derivative of objective at lower bound
15     T dedp=de_dp(p,xobs,yobs);
16     // derivative of objective at midpoint
17     T dedp_mid=de_dp(p_mid,xobs,yobs);
18     // collapse search interval if stationary point found
19     if (fabs(dedp_mid)<eps) {
20       p=p_up=p_mid;
21     }
```

```
22      // continue bisection if stationary point not found
23      else if ((dedp<=0&&dedp_mid<=0)||(dedp>=0&&dedp_mid>=0)) {
24        // search in right half of sign switches there
25        p=p_mid;
26      } else {
27        // search in left half otherwise
28        p_up=p_mid;
29      }
30    }
31    return p;
```

→ 1D/optimization/bisection/*

- "Going downhill" appears to be a reasonable heuristic for minimization.
- The negative first derivative (gradient) points downhill.
- The main question is about "how far" to follow it in order to ensure a decrease in the objective. Line search is employed.
- The accuracy of the first derivative can be crucial for the performance of gradient descent. Exact differentiation methods are required instead of numerical approximation by finite differences.
- Algorithmic differentiation (AD) turns out to be the preferred method. The AD library dco/c++ enables exact and efficient numerical differentiation of differentiable programs.

The negative first derivative (also: gradient in vector case) points into the direction of steepest descent, which turns out to be a reasonable search direction in the context of continuous minimization.

Starting from some initial estimate for the stationary point p, the gradient descent method iteratively takes steps in direction of the negative gradient,

$$p := p - e' \quad \text{while } |e'| > \epsilon \, .$$

In the scalar case the choice is between stepping toward $-\infty$ or ∞. The first derivative e' indicates local increase ($e' > 0$) or decrease ($e' < 0$) of the the function value.

Aiming for decrease the next step should be toward $-\infty$ if $e' > 0$ or toward ∞ if $e' < 0$. No further local decrease in the function value can be achieved for $e' = 0$ (necessary optimality condition).

Validation of a local minimum at requires $e'' > 0$. Similarly, a local maximum is found if $e'' < 0$. $e'' = 0$ indicates a degenerate (locally not unique) stationary point.

The gradient descent method amounts to a fixpoint iteration as

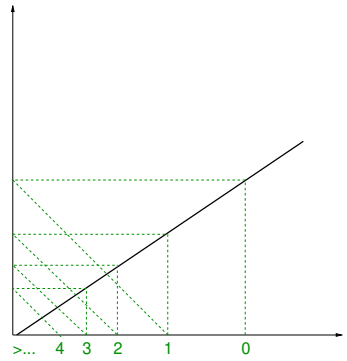
$$p = \check{e}(p) = p - e' \, .$$

The above converges locally, that is, there is a neighborhood of the current iterate p, such that a gradient descent step yields a decrease in $|e'| = |e'(p)|$, if $\check{e}$ is locally contractive, that is,

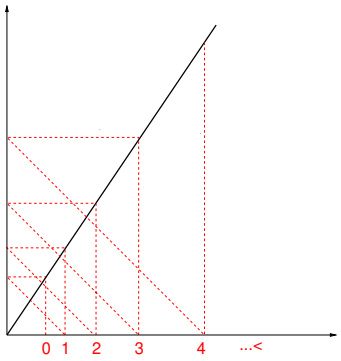
$$\left| \frac{d\check{e}}{dp} \right| = |1 - e''| < 1$$

implying

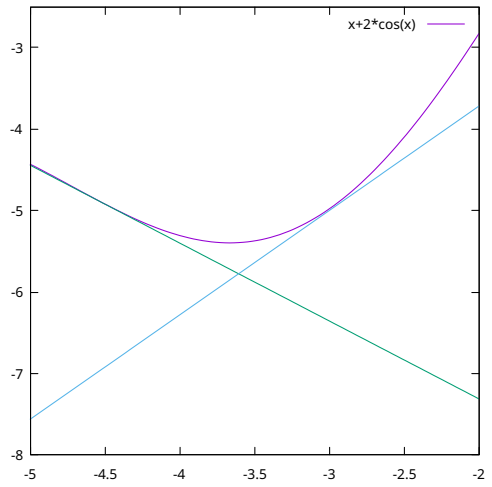
$$0 < e'' < 2 \, .$$



convergence for $\left| \frac{d\check{e}}{dp} \right| \leq 1$



divergence for $\left| \frac{d\check{e}}{dp} \right| > 1$



Note potential for "ping-pong" (step length too long) or "near-stalling" (step length too short).

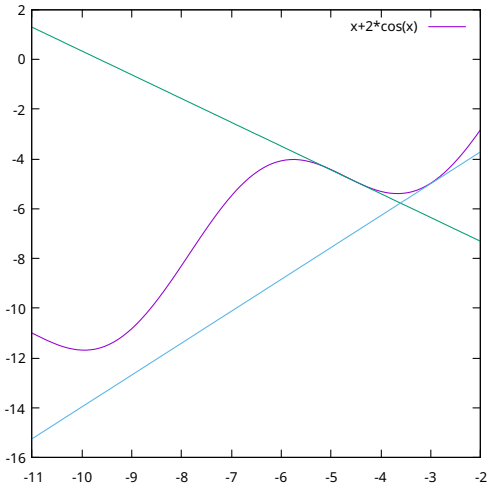
Illustrate the "ping-pong" effect of gradient descent applied to $\min_x f(x)$ for $f(x) \equiv x^2$.

Note: $f'' = 2$ violates the local contractiveness criterion.

Illustrate the "near-stalling" effect of gradient descent applied to $\min_x f(x)$ for $f(x) \equiv x^8$.

Note: $f' = 8 \cdot x^7$ results in immediate convergence for x still rather far from zero, e.g., $x = 10^{-3}$.

→ misc/gradient_descent_convergence/*



Note potential for "random" jumps across local maxima.

It remains to be decided how large the step into the current descent direction should be. The step size is typically damped in order to ensure continued progress toward $\min_{p \in \mathbb{R}} e(p, \mathbf{x}, \mathbf{y})$ yielding the recurrence

$$p := p - \alpha \cdot e' \quad \text{while } |e'| > \epsilon \ .$$

The damping parameter[8] α is often determined by line search, e.g., by iterative bisection starting from $\alpha = 1$. More rigorously, optimal α can be computed that maximize the decrease, which is rarely done due to higher computational cost.

The gradient descent method with line search is globally convergent (in infinite precision arithmetic). Convergence is linear (slow).

[8] also known as *learning rate* in machine learning


```
1  /// stationary point of objective wrt. model parameter by the gradient descent method
2  template<typename T>
3  T fit(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs);
6    assert(xobs.size()==yobs.size());
7    // derivative of objective wrt. model parameter
8    T dedp=de_dp(p,xobs,yobs);
9    // search for stationary point of objective wrt. model parameter
10   do {
11     // simple line search
12     T alpha=1;
13     T e_=e(p,xobs,yobs);
14     while (e(p-alpha*dedp,xobs,yobs)>=e_) {
15       alpha=alpha/2;
16     }
17     // gradient descent step
18     p=p-alpha*dedp;
19     dedp=de_dp(p,xobs,yobs);
20   } while (fabs(dedp)>eps); // convergence criterion
21   return p;
22 }
```

Accurate derivatives are highly desirable. Symbolic differentiation can be tedious and error-prone. For higher derivatives, the size of the symbolic derivative expression grows exponentially with the order due to the chain,

$$b = f(g(a)) \quad \Rightarrow \quad b' = f'(g(a)) \cdot g'(a) \,,$$

and product,

$$(a \cdot b)' = a' \cdot b + a \cdot b' \,,$$

rules of differentiation. Common subexpressions appear. E.g., $f(x) = \sin(\cos(x))$ yields

$$\begin{aligned} f'(x) &= \cos(\cos(x)) \cdot (-\sin(x)) \\ f''(x) &= -\sin(\cos(x)) \cdot (-\sin(x)) \cdot (-\sin(x)) + \cos(\cos(x)) \cdot (-\cos(x)) \\ f'''(x) &= \ldots \end{aligned}$$

AD automates the differentiation of differentiable programs. It avoids exponential computational complexity of higher derivatives.

First-order tangents

$$e^{(1)} : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : \quad e^{(1)} = e^{(1)}(p, p^{(1)})$$

of [9]

$$e : \mathbb{R} \to \mathbb{R} : \quad e = e(p)$$

are scaled first derivatives of e, i.e,

$$e^{(1)} = e^{(1)}(p, p^{(1)}) \equiv e' \cdot p^{(1)} \,,$$

where $p^{(1)} = 1$ yields

$$e' \equiv \frac{de}{dp} \in \mathbb{R} \,.$$

Superscripts $*^{(1)}$ are used to denote first-order tangents of functions and (program) variables.

[9] Dependence of e on observations **x** and **y** is omitted to simplify the notation.

The dco/c++ library[10] is a commercial AD solution by *n*AG Ltd., Oxford, UK. It has been developed jointly with STCE since 2006.

All active program variables must have their types changed into the custom first-order tangent type dco::gt1s<**double**>::type featuring value and tangent components.

Both value and tangent are initialized to zero at the time of declaration.

Read and write accesses to the value of a scalar program variable v is provided by dco::value(v). Read and write accesses to the corresponding tangent is provided by dco::derivative(v).

Second and higher derivatives are supported.

[10] derivative code by overloading in C++

In dco/c++ all relevant floating-point operations (e.g., sin and pow) are *overloaded* for variables of tangent type dco::gt1s<**double**>::type to compute both values v and tangents $v^{(1)}$.
E.g., let

$$y = f(x) = \sin(x^2)$$

be implemented as the differentiable program

```
1 double f(double x) {
2   return sin(pow(x,2));
3 }
```

This program is evaluated at runtime as a sequence of *elemental* operations as

$$v = x^2; \; y = \sin(v) \,.$$

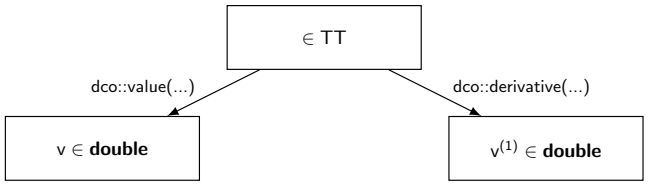
Tangent AD augments this decomposed program with the evaluation of tangents as

$$\begin{pmatrix} v \\ v^{(1)} \end{pmatrix} = \begin{pmatrix} x^2 \\ 2 \cdot x \cdot x^{(1)} \end{pmatrix}$$
$$\begin{pmatrix} y \\ y^{(1)} \end{pmatrix} = \begin{pmatrix} \sin(v) \\ \cos(v) \cdot v^{(1)} \end{pmatrix} \,.$$

Setting $x^{(1)} = 1$ yields

$$y^{(1)} = \frac{df}{dx} = \cos(x^2) \cdot 2 \cdot x.$$

$$e = e(p)$$
$$e^{(1)} = e' \cdot p^{(1)}$$



$$\text{p\_t} = (p, p^{(1)})$$
$$\text{e\_t} = (e, e^{(1)})$$

```
using TT=dco::gt1s<double>::type;
TT p_t,e_t;
```



```
#include "dco.hpp"

/// first derivative of objective wrt. model parameter by tangent AD
template<typename T>
T de_dp(T p, VT<T> xobs, VT<T> yobs) {
  // conditions
  assert(xobs.size()==nobs);
  assert(xobs.size()==yobs.size());
  // tangent type
  using TT=dco::gt1s<T>::type;
  // activation
  TT p_t=p, e_t;
  // seeding derivative of model parameter
  dco::derivative(p_t)=1;
  // overloaded computation of objective (explicit template instantiation required due to passive xobs and
      yobs)
  e_t=e<TT>(p_t,xobs,yobs);
  // harvesting derivative of objective wrt. model parameter
  return dco::derivative(e_t);
}
```


Let

- the source be given as main.cpp.
- the dco/c++ library be installed in thirdParty/dcocpp.
- the Eigen library be installed in thirdParty/eigen.

To build type

```
export NAG_KUSARI_FILE=../../../../thirdParty/dcocpp/nag_key.txt

g++ -I../../../../thirdParty/eigen -I../../../../thirdParty/dcocpp/include \
  -DDCO_DISABLE_AUTO_WARNING -DDCO_DISABLE_AVX2_WARNING \
  -DDCO_EXT_EIGEN_IGNORE_VERSION -DDCO_CHUNK_TAPE \
  -DDCO_NO_INTERMEDIATES -L../../../../thirdParty/dcocpp/lib \
  -ldcoc -I.. main.cpp -o main.exe
```

Refer to the dco/c++ user guide for details.

The second-order optimality criterion for a local minimum requires the second derivative of e, which can be approximated as the derivative of [a finite difference approximation of] e'.

$$e''(x) = e''(p, \Delta p) \approx \frac{e'(p + \Delta p, \Delta p) - e'(p - \Delta p, \Delta p)}{2 \cdot \Delta p}$$

$$= \frac{e(p + 2 \cdot \Delta p) - 2 \cdot e(p) + e(p - 2 \cdot \Delta p)}{4 \cdot \Delta p^2}$$

The first expression yields natural approaches to implementing second-order finite differences as first-order finite differences of implementations of the first derivative.

The second expression suggests potential numerical issues due to squaring of an ideally small Δp.

A numerically more stable validation of the second-order optimality criterion can be approximated as a [central] finite difference of the given tangent objective as follows:

$$\tilde{e}''(\Delta p) = \frac{e^{(1)}(p + \Delta p, 1) - e^{(1)}(p - \Delta p, 1)}{2 \cdot \Delta p}$$

One would like to see

$$\tilde{e}''(\Delta p) > 0 \, .$$


```
1  /// derivative of derivative of objective wrt. model parameter wrt. model parameter by finite differences
2  template<typename T>
3  T dde_dp_dp_cfd(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs);
6    assert(xobs.size()==yobs.size());
7    // perturbing approx. midpoint of mantissa
8    T delta_p=eps;
9    if (p!=0) {
10     delta_p=delta_p*fabs(p);
11   }
12   // central finite difference of first derivative
13   return (de_dp(p+delta_p,xobs,yobs)-de_dp(p-delta_p,xobs,yobs))/(delta_p+delta_p);
14 }
```


First-order tangents

$$e'^{(1)} : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : \quad e'^{(1)} = e'^{(1)}(p, p^{(1)})$$

of[11]

$$e' : \mathbb{R} \to \mathbb{R} : \quad e' = e'(p)$$

yield second derivatives of

$$e : \mathbb{R} \to \mathbb{R} : \quad e = e(p)$$

as

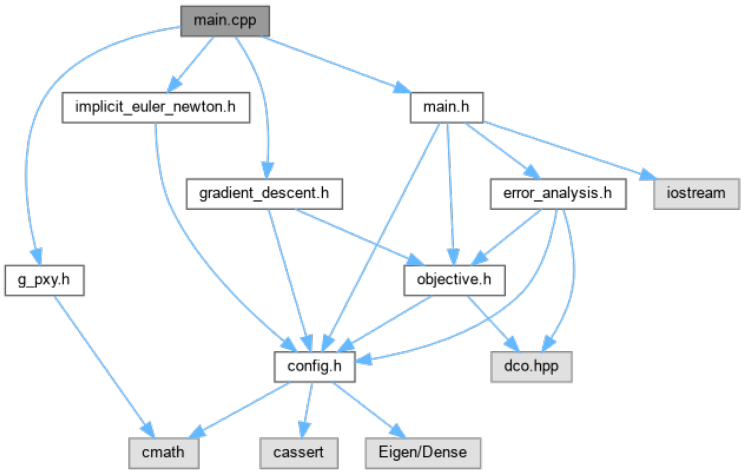
$$e'^{(1)} = e'^{(1)}(p, p^{(1)}) \equiv e'' \cdot p^{(1)} \, ,$$

where $p^{(1)} = 1$ yields

$$e'' \equiv \frac{d^2 e}{dp^2} \in \mathbb{R} \, .$$

[11] Dependence of e' on $\mathbf{x}$ is omitted to simplify the notation.


```
1  /// derivative of derivative of objective wrt. model parameter wrt. model parameter by tangent AD
2  template<typename T>
3  T dde_dp_dp(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs);
6    assert(xobs.size()==yobs.size());
7    // tangent type
8    using TT=dco::gt1s<T>::type;
9    // activation
10   TT p_t=p, dedp_t;
11   // seeding derivative of model parameter
12   dco::derivative(p_t)=1;
13   // overloaded computation of derivative of objective wrt. model parameter
14   dedp_t=de_dp<TT>(p_t,xobs,yobs);
15   // harvesting derivative of derivative of objective wrt. model parameter wrt. model parameter
16   return dco::derivative(dedp_t);
17 }
```



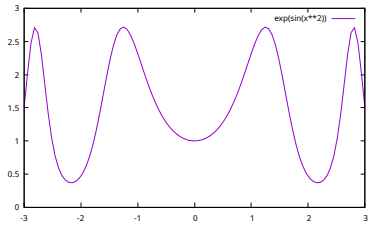
→ 1D/optimization/gradient_descent/*

Refer to upcoming discussion of error analysis for details on error_analysis.h.

Hands On! (Exercise 6)

Compute local minima of

$$y = e^{\sin(p \cdot x^2)} .$$



$p = 1$

Implement the required first derivatives

1. symbolically
2. numerically (central finite differences)
3. algorithmically (dco/++).

Use second-order tangent AD to check the second-order optimality criterion.

Experiment with different starting points and values of the parameter p. Record the number of iterations performed.

Implement the bisection method and compare the results.

Exercise 6
Notes

- Implement the gradient descent method (copy'n'paste from sample code).
- Implement the first, `T df_dx(T p, T x)`, and second, `T ddf_dx_dx(T p, T x)`, derivatives of

```
1 template<typename T>
2 T f(T p, T x) {
3   return exp(sin(p*pow(x,2)));
4 }
```

- Build and run to see

```
+++++++++++++++
x=27.4301
y=0.367879
dfdx=1.64092e-08
ddfdxx=1107.19
```

  for $p = x = 1$. The number of iterations is equal to the number of +'s printed.
- Play with p and x.

→ exercises/6/

Data Error Analysis
Motivation

For given data $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}}$ and model $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : y = f(p, x)$, parameter estimation computes $p \in \mathbb{R}$ such that the least squares error of model predictions wrt. the data is minimized, that is,

$$\phi(p, \mathbf{v}) \equiv \text{argmin}_p e = \text{argmin}_p \left(\sum_{i=0}^{n_{\text{obs}}-1} (f(p, x_i) - y_i)^2 \right) ,$$

where $\mathbf{v} = \binom{\mathbf{x}}{\mathbf{y}} \in \mathbb{R}^{2 \cdot n_{\text{obs}}}$.

Data error analysis aims to quantify the sensitivity of the optimal p wrt. highly likely (infinitesimal) errors in the data, that is, the gradient

$$\phi' \equiv \left(\frac{d\phi}{d\mathbf{v}} \right)^T \in \mathbb{R}^{2 \cdot n_{\text{obs}}}$$

needs to be computed. Large sensitivity wrt. a data point implies (at least) the need for more precise observation techniques.

Let $\mathbb{R}^{2 \cdot n_{\text{obs}}}$ be the domain of the multivariate scalar function $\phi : \mathbb{R}^{2 \cdot n_{\text{obs}}} \to \mathbb{R}$.
The function ϕ is continuous at a point $\tilde{\mathbf{v}} \in \mathbb{R}^{2 \cdot n_{\text{obs}}}$ if

$$\lim_{\mathbf{v} \to \tilde{\mathbf{v}}} \phi(\mathbf{v}) = \phi(\tilde{\mathbf{v}}) \quad .$$

The function ϕ is differentiable at $\tilde{\mathbf{v}}$ if there is a vector $\phi' \in \mathbb{R}^{2 \cdot n_{\text{obs}}}$ such that

$$\phi(\tilde{\mathbf{v}} + \Delta\mathbf{v}) = \phi(\tilde{\mathbf{v}}) + \phi'^{T} \cdot \Delta\mathbf{v} + r$$

with asymptotically vanishing remainder $r = r(\tilde{\mathbf{v}}, \Delta\mathbf{v}) \in \mathbb{R}$, such that

$$\lim_{\Delta\mathbf{v} \to 0} \frac{r}{\|\Delta\mathbf{v}\|_2} = 0 \; .$$

ϕ' is called the gradient of ϕ.

The central finite difference

$$\phi'_i \approx_{\mathcal{O}(\|\Delta\mathbf{v}\|_2^2)} \frac{\phi(p, \mathbf{v} + \Delta v_i \cdot \mathbf{e}_i) - \phi(p, \mathbf{v} - \Delta v_i \cdot \mathbf{e}_i)}{2 \cdot \Delta v_i} \; , \quad i = 0, \ldots 2 \cdot n_{\text{obs}} - 1$$

yields a second-order accurate approximation of the gradient

$$\phi' = (\phi'_i)_{i=0,\ldots 2 \cdot n_{\text{obs}} - 1}$$

The runtime of approximating ϕ' is equal to $2 \cdot n_{\text{obs}} \cdot \mathcal{O}(\text{Cost}(\phi))$.


```
1  /// error analysis by central finite differences
2  template<typename T>
3  VT<T> dp_dobs_cfd(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs);
6    assert(xobs.size()==yobs.size());
7    // return value
8    VT<T> dpdobs(2*nobs);
9    int i=0, j=0;
10   while (i<nobs) {
11     T xobs_i_save=xobs(i), yobs_i_save=yobs(i), pp, pm;
12     // perturbing approx. midpoint of mantissa
13     T delta=eps;
14     if (xobs(i)!=0) {
15       delta=delta*fabs(xobs(i));
16     }
17     // central finite difference quotient
18     xobs(i)=xobs(i)+delta;
19     pp=fit(p,xobs,yobs);
20     xobs(i)=xobs_i_save-delta;
21     pm=fit(p,xobs,yobs);
```



```
22     xobs(i)=xobs_i_save;
23     dpdobs(j)=(pp-pm)/(delta+delta); j=j+1;
24     // perturbing approx. midpoint of mantissa
25     delta=eps;
26     if (yobs(i)!=0) {
27       delta=delta*fabs(yobs(i));
28     }
29     // central finite difference quotient
30     yobs(i)=yobs(i)+delta;
31     pp=fit(p,xobs,yobs);
32     yobs(i)=yobs_i_save-delta;
33     pm=fit(p,xobs,yobs);
34     yobs(i)=yobs_i_save;
35     dpdobs(j)=(pp-pm)/(delta+delta); j=j+1;
36     i=i+1;
37   }
38   return dpdobs;
39 }
```

Tangent AD

$$\phi'_i = \phi^{(1)}(p, \underbrace{0}_{=p^{(1)}}, \mathbf{v}, \underbrace{\mathbf{e}_i}_{=\mathbf{v}^{(1)}}), \quad i = 0, \ldots 2 \cdot n_{\text{obs}} - 1$$

yields the gradient

$$\phi' = (\phi'_i)_{i=0,\ldots 2\cdot n_{\text{obs}}-1}$$

with machine accuracy.

Note that the program variable p is a passive input yielding $p^{(1)} = 0$. An implementation of $\phi^{(1)}$ may use the same program variable to return the optimal p, making it an active output.

The runtime of computing ϕ' is equal to $2 \cdot n_{\text{obs}} \cdot \mathcal{O}(\text{Cost}(\phi))$.

Let

$$y = f(\mathbf{x}) = \sin(x_0 \cdot x_1)$$

be implemented as the differentiable program

```
1 double f(VT<double> x) {
2   return sin(x(0)*x(1));
3 }
```

decomposing into

$$v = x_0 \cdot x_1$$
$$y = \sin(v) .$$

Tangent AD yields

$$\begin{pmatrix} v \\ v^{(1)} \end{pmatrix} = \begin{pmatrix} x_0 \cdot x_1 \\ x_0^{(1)} \cdot x_1 + x_0 \cdot x_1^{(1)} \end{pmatrix}$$
$$\begin{pmatrix} y \\ y^{(1)} \end{pmatrix} = \begin{pmatrix} \sin(v) \\ \cos(v) \cdot v^{(1)} \end{pmatrix} .$$

Setting $x_0^{(1)} = 1$ and $x_1^{(1)} = 0$ yields

$$y^{(1)} = \frac{df}{dx_0} = \cos(x_0 \cdot x_1) \cdot x_1.$$

Similarly, $x_0^{(1)} = 0$ and $x_1^{(1)} = 1$ yields

$$y^{(1)} = \frac{df}{dx_1} = \cos(x_0 \cdot x_1) \cdot x_0.$$


```
1  /// error analysis by tangent AD
2  template<typename T>
3  VT<T> dp_dobs_tad(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs); assert(xobs.size()==yobs.size());
6    // tangent type
7    using TT=dco::gt1s<T>::type;
8    // activation
9    TT p_t; VT<TT> xobs_t(nobs), yobs_t(nobs);
10   int i=0;
11   while (i<nobs) {
12     xobs_t(i)=xobs(i); yobs_t(i)=yobs(i); i=i+1;
13   }
14   // return value
15   VT<T> dpdobs(2*nobs);
16   int j=0; i=0;
17   while (i<nobs) {
18     // tangent in standard basis direction of current point of observation
19     p_t=p; // reset model parameter to its initial value
20     // seed
21     dco::derivative(xobs_t(i))=1;
```



```
22     // overloaded parameter fitting
23     p_t=fit(p_t,xobs_t,yobs_t); // all arguments of same type
24     // harvest
25     dpdobs(j)=dco::derivative(p_t); j=j+1;
26     // enable correct next seed
27     dco::derivative(xobs_t(i))=0;
28     // tangent in standard basis direction of current observed value
29     p_t=p; // reset model parameter to its initial value
30     // seed
31     dco::derivative(yobs_t(i))=1;
32     // overloaded parameter fitting
33     p_t=fit(p_t,xobs_t,yobs_t);
34     // harvest
35     dpdobs(j)=dco::derivative(p_t); j=j+1;
36    // enable correct next seed
37     dco::derivative(yobs_t(i))=0;
38     i=i+1;
39   }
40   return dpdobs;
41 }
```

Adjoint AD

$$\phi' = p_{(1)} := \phi_{(1)}(p, \underbrace{1}_{=p_{(1)}}, \mathbf{v}, \underbrace{0}_{=\mathbf{v}_{(1)}})$$

yields the gradient $\phi' \equiv \frac{dp}{d\mathbf{v}} \in \mathbb{R}^{2 \cdot n_{\text{obs}}}$ with machine accuracy. Subscripts $*_{(1)}$ are used to denote first-order adjoints.

We use ":=" to denote imperative assignment as opposed to equality ("=") since the program variable $p_{(1)}$ appears both on the left- and right-hand sides.

Note that the adjoint $\mathbf{v}_{(1)}$ of $\mathbf{v}$ is an output of $\phi_{(1)}$. It is not an output of ϕ yielding $\mathbf{v}_{(1)} = 0$ as input to $\phi_{(1)}$.

The runtime of computing ϕ' is equal to $1 \cdot \mathcal{O}(\text{Cost}(\phi))$. For large values of n_{obs} adjoint AD promises to outperform both finite differences and tangent AD.

The types of all active program variables must be changed into the custom first-order adjoint type AT=dco::ga1s<**double**>::type featuring value and, conceptually,[12] adjoint components.

The active inputs must be registered with a tape recording the evaluation of the target function.

The resulting data structure (variable name tape) of type dco::smart_tape_ptr_t<dco::mode<AT>> is generated during an the execution of the overloaded target function.

Interpretation of the tape by calling tape->interpret_adjoint() yields adjoints of the active inputs for given adjoints of the active outputs.

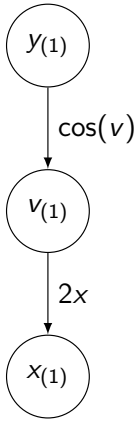
Both value and adjoint are initialized to zero at the time of declaration. Read and write access to the value of an active program variable v is provided by dco::value(v). Its adjoint can be accessed by dco::derivative(v).

Second and higher derivatives are supported.

[12] Details of the implementation are beyond the scope of this course.

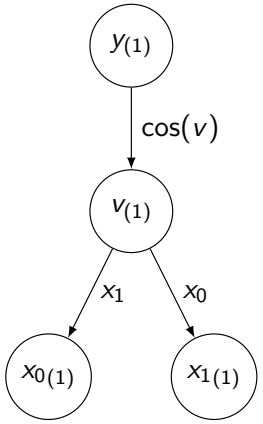
A sequence of elemental adjoints is evaluated. Data flow reversal is implemented by interpretation of the tape.

$v = x^2$
$y = \sin(v)$



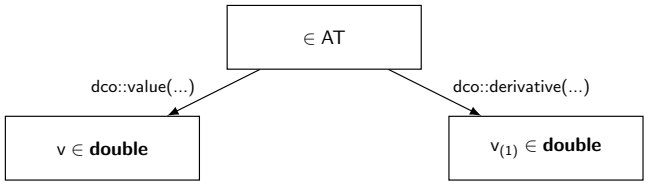
$v_{(1)} = \cos(v) \cdot y_{(1)}$
$x_{(1)} = 2x \cdot v_{(1)}$

$v = x_0 \cdot x_1$
$y = \sin(v)$



$v_{(1)} = \cos(v) \cdot y_{(1)}$
$x_{1(1)} = x_0 \cdot v_{(1)}$
$x_{0(1)} = x_1 \cdot v_{(1)}$

$$e = e(p)$$
$$p_{(1)} = e' \cdot e_{(1)}$$



p_a= $(p, p_{(1)})$
e_a= $(e, e_{(1)})$

```
1  using AT=dco::ga1s<double>::type;
2  AT p_a,e_a;
```



```
1  /// error analysis by adjoint AD
2  template<typename T>
3  VT<T> dp_dobs_aad(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs);
6    assert(xobs.size()==yobs.size());
7    // adjoint type
8    using AT=dco::ga1s<T>::type;
9    // tape
10   dco::smart_tape_ptr_t<dco::mode<AT>> tape;
11   // activation and registration
12   AT p_a=p;
13   VT<AT> xobs_a(nobs), yobs_a(nobs);
14   int i=0;
15   while (i<nobs) {
16     xobs_a(i)=xobs(i);
17     tape->register_variable(xobs_a(i));
18     yobs_a(i)=yobs(i);
19     tape->register_variable(yobs_a(i));
20     i=i+1;
21   }
```



```
22   // overloaded parameter fitting
23   p_a=fit(p_a,xobs_a,yobs_a);
24   // seed
25   dco::derivative(p_a)=1;
26   // interpretation of tape
27   tape->interpret_adjoint();
28   // harvest
29   VT<T> dpdobs(2*nobs);
30   int j=0;
31   i=0;
32   while (i<nobs) {
33     dpdobs(j)=dco::derivative(xobs_a(i)); j=j+1;
34     dpdobs(j)=dco::derivative(yobs_a(i)); j=j+1;
35     i=i+1;
36   }
37   return dpdobs;
38 }
```


Implement the computation of the gradient $f' \in \mathbb{R}^n$ of the differentiable program

$$y = f(x) = \sin\left(\sum_{i=0}^{n-1} x_i^2\right)$$

in the tangent and adjoint modes of AD with dco/c++.

Verify equality of the numerical results.

Compare run times and memory requirements for increasing n.

Hands On! (Exercise 7)

Modify the sample implementation of the gradient descent method applied to Scenario 4 such that e' is computed by adjoint AD with dco/c++ as

$$\begin{aligned} e_{(1)} &: \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R} \to \mathbb{R} \\ &: p_{(1)} := e_{(1)}(p, p_{(1)}, \mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}, \mathbf{y}_{(1)}, \epsilon_{(1)}) \end{aligned}$$

of

$$e : \mathbb{R} \times \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}} \to \mathbb{R} : \quad \epsilon = e(p, \mathbf{x}, \mathbf{y}) \; .$$

Note that the adjoint $\epsilon_{(1)}$ of the exclusive output ϵ is appended to the argument list of $e_{(1)}$.

Implement

$$p_{(1)} = e_{(1)}(p, 0, \mathbf{x}, 0, \mathbf{y}, 0, 1) = \frac{de}{dp} \in \mathbb{R} \; .$$

Exercise 7

Notes

- Copy all relevant files, that is,

  config.h g_pxy.h implicit_euler_newton.h main.cpp
  objective.h gradient_descent.h main.h

  into separate directory. Remove data error analysis from main.h.
- Modify the function T de_dp(T p, VT<T> xobs, VT<T> yobs) for use with dco/c++ in adjoint mode.
- Build and run to see results matching those obtained by tangent AD.
- Note that there is no runtime benefit compared to tangent AD. Actually, a slowdown can be expected for computing the first derivative of a univariate scalar function due to data flow reversal.

→ exercises/7/

Data Error Analysis
Symbolic Differentiation

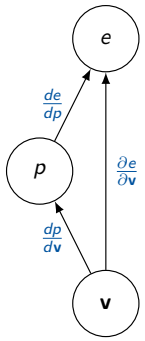
The first-order optimality condition

$$\frac{de}{dp}(p, \mathbf{v}) = 0$$

for the parameter estimation problem

$$\min_p e(p, \mathbf{v})$$

defines $p = p(\mathbf{v})$ as an implicit function, yielding the (directed acyclic) data dependence graph on the right.



We distinguish between partial, e.g., $\frac{\partial e}{\partial \mathbf{v}}$, and total, e.g., $\frac{dp}{d\mathbf{v}}$, derivatives. Total derivatives capture all dependences wrt. the active argument (e.g., there are no alternative paths from $\mathbf{v}$ to p apart from the edge labelled $\frac{dp}{d\mathbf{v}}$). Partial derivatives do not (e.g., there is an alternative path from $\mathbf{v}$ to e in addition to the edge labelled $\frac{\partial e}{\partial \mathbf{v}}$).

Data Error Analysis
Symbolic Differentiation

Let $p = p(\mathbf{v}) = \min_p e(p(\mathbf{v}), \mathbf{v})$ for the twice continuously differentiable objective $e : \mathbb{R} \times \mathbb{R}^{n_{\text{obs}}} \to \mathbb{R}$. Differentiation of the first-order optimality condition

$$\frac{de}{dp}(p, \mathbf{v}) = 0$$

at the solution $p \in \mathbb{R}$ wrt. $\mathbf{v}$ yields

$$\underbrace{\frac{d^2 e}{dp d\mathbf{v}}}_{\in \mathbb{R}^{1 \times n_{\text{obs}}}} = \underbrace{\frac{\partial \frac{de}{dp}}{\partial \mathbf{v}}}_{\in \mathbb{R}^{1 \times n_{\text{obs}}}} + \underbrace{\frac{d^2 e}{dp^2}}_{\in \mathbb{R}} \cdot \underbrace{\frac{dp}{d\mathbf{v}}}_{\in \mathbb{R}^{1 \times n_{\text{obs}}}} = 0$$

and, hence,

$$\frac{dp}{d\mathbf{v}} = -\frac{\frac{\partial \frac{de}{dp}}{\partial \mathbf{v}}}{\frac{d^2 e}{dp^2}} = -\frac{\frac{\partial \frac{de}{d\mathbf{v}}}{\partial p}}{\frac{d^2 e}{dp^2}}$$

implying ...

. . . the tangent

$$p^{(1)} \equiv \frac{dp}{d\mathbf{v}} \cdot \mathbf{v}^{(1)} = -\frac{\frac{\partial \frac{de}{d\mathbf{v}}}{\partial p}}{\frac{d^2 e}{dp^2}} \cdot \mathbf{v}^{(1)}$$

and the adjoint

$$\mathbf{v}_{(1)} \equiv p_{(1)} \cdot \frac{dp}{d\mathbf{v}} = -p_{(1)} \cdot \frac{\frac{\partial \frac{de}{d\mathbf{v}}}{\partial p}}{\frac{d^2 e}{dp^2}}$$

Set $p_{(1)} = 1$ for

$$\frac{dp}{d\mathbf{v}} == -\frac{\frac{\partial \frac{de}{d\mathbf{v}}}{\partial p}}{\frac{d^2 e}{dp^2}}$$

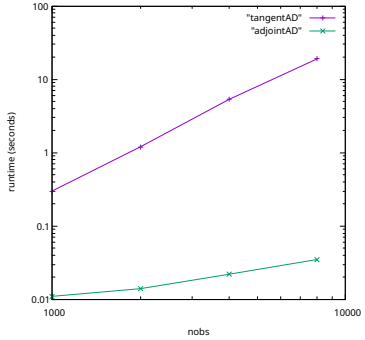

```
1 /// error analysis by symbolic differentiation
2 template<typename T>
3 VT<T> dp_dobs_sym(T p, VT<T> xobs, VT<T> yobs) {
4   // conditions
5   assert(xobs.size()==nobs);
6   assert(xobs.size()==yobs.size());
7   // result of differentiation of first-order optimality condition
8   return -dde_dobs_dp(p,xobs,yobs)/dde_dp_dp(p,xobs,yobs);
9 }
```

where dde_dxobs_dp(p,xobs,yobs) is preferably computed by tangent AD applied to the gradient of the objective wrt. the observations.

The latter should be computed by adjoint AD as typically $n_{\text{obs}} \gg 1$.

Building on Exercise 7, compare the runtimes of data error analysis by tangent vs. adjoint AD for growing values of n_{obs}. You should observe the behavior depicted on the right.

The relative (wrt. an evaluation of e) runtime of e' by tangent AD grows with n_{obs}. The relative of e' by adjoint AD is independent of n_{obs}.



Collect similar data for the symbolic differentiation and [central] finite differences methods and add them to the plot.

Replace gradient descent by the bisection method with initial search interval $[-1.5 : 0.5]$. Compare the results.

- Copy the code from Exercise 7.
- Measure the runtimes of the different data error analysis methods (tangent AD, adjoint AD, central finite differences, symbolic differentiation) for $n_{\text{obs}} = 1000, 2000, 4000, 8000$. Use
  `time ./main.exe`
- Store the runtimes $t = t(n_{\text{obs}})$ as a sequence of pairs (n_{obs}, t) in four different text files (`tangentAD`, `adjointAD`, `cfd`, `symdiff`).
- Plot the results.
  - Run gnuplot.
  - Type `plot "tangentAD" with linespoints` (append the remaining three files separated by commas.

- Linearity of the model in its parameter yields linear regression problems (e.g., Scenario 1 and Scenario 2).
- Exploitation of their special properties yields better performing numerical optimization methods, including normal equation and Householder methods.
- Parameter estimation amounts to a quadratic minimization problem. Linear regression methods compute the unique solution very efficiently.
- Both the normal equation and Householder method for linear regression generalize to the vector case ($\mathbf{p} \in \mathbb{R}^{n_p}$, $n_p > 1$). The numerical stability of the former may become unsatisfactory due to poor *conditioning*[13].

[13] See material on vector case for more.

Let the linear[14] (in p) model

$$f : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : y = f(p,x) = {}^{15} \quad f'(x) \cdot p \equiv \frac{df}{dp}(x) \cdot p$$

be implemented as

```
1 template<typename T>
2 T f(T p, T x);
```

computing the solution of the initial value problem

$$\frac{dy}{dx} = g(p,x)\,, \quad y(p,0) = 0\,.$$

A linear (in p) right-hand side $g(p,x) = g'(x) \cdot p \equiv \frac{dg}{dp}(x) \cdot p$ is implied.

[14] more generally, affine
[15] Equality follows immediately from linearity in p.

The above holds for both Scenario 1 and Scenario 2, yielding

$$\frac{dy}{dx} = p;\ y(0) = 0 \quad \Rightarrow \quad g'(x) = 1,\ f'(x) = x$$
$$\frac{dy}{dx} = \frac{p}{x+1};\ y(0) = 0 \quad \Rightarrow \quad g'(x) = \frac{1}{x+1},\ f'(x) = \log(x+1)\,.$$

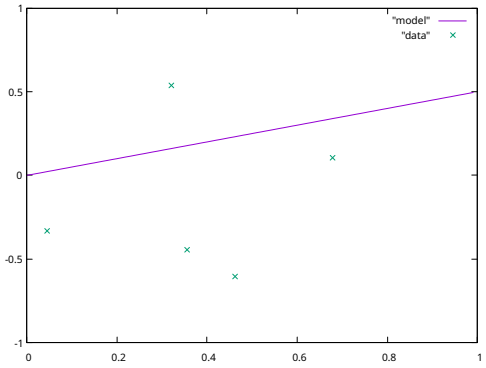
For simulation we combine implicit Euler integration of the given initial value problems with Newton's method for root finding.

Combination with bisection results in erroneous behavior due to non-differentiability (see Exercise 9).

Implementation of a similar setup for explicit Euler integration follows naturally.

We consider randomly generated data

$$(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}}$$
$$\mathbf{x} = (x_i)_{0,\ldots,n_{\text{obs}}-1}$$
$$\mathbf{y} = (y_i)_{0,\ldots,n_{\text{obs}}-1} .$$



The residual vector

$$\mathbf{r} = \mathbf{r}(p, \mathbf{x}, \mathbf{y}) \equiv (f(p, x_i) - y_i)_{i=0,\ldots,n_{\text{obs}}-1} \in \mathbb{R}^{n_{\text{obs}}}$$

yields a scalar error as its squared Euclidean norm, that is,

$$e = \|\mathbf{r}\|_2^2 \equiv \sum_{i=0}^{n_{\text{obs}}-1} r_i^2 = \sum_{i=0}^{n_{\text{obs}}-1} (f(p, x_i) - y_i)^2 .$$

The first derivative of the residual wrt. p

$$\mathbf{r}' \equiv \frac{d\mathbf{r}}{dp}$$

is required for linear as well as for nonlinear regression (to be discussed later). It can be computed accurately by (tangent) AD.

The magnitude of a vector $\mathbf{v} \in \mathbb{R}^\mu$ is measured by its norms defined as

$$\|\mathbf{v}\|_k = \left(\sum_{i=0}^{\mu-1} |v_i|^k \right)^{\frac{1}{k}} , \text{ e.g.,}$$

1-norm: $$\|\mathbf{v}\|_1 = \sum_{i=0}^{\mu-1} |v_i|$$

2-norm: $$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=0}^{\mu-1} |v_i|^2} = \sqrt{\sum_{i=0}^{\mu-1} v_i^2} = \sqrt{\mathbf{v}^T \cdot \mathbf{v}}$$

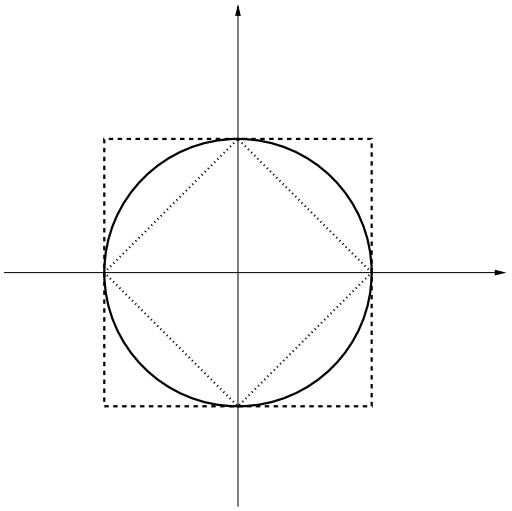
Squared 2-norm: $$\|\mathbf{v}\|_2^2 = \mathbf{v}^T \cdot \mathbf{v}$$

$\infty -$ norm: $$\|\mathbf{v}\|_\infty = \max_{0 \le i \le \mu-1} |v_i|$$

Vector-induced matrix norms can be defined such that the norms of single-row matrices become equivalent to the corresponding vector norm, e.g., $\|A\|_1 = \max_{j=0,\ldots,\nu-1} \sum_{i=0}^{\mu-1} |a_{i,j}|$ for $A \in \mathbb{R}^{\mu \times \nu}$.

Visualize the set of all vectors $\mathbf{v} \in \mathbb{R}^2$ with

1. $\|\mathbf{v}\|_1 = 1$
2. $\|\mathbf{v}\|_2 = 1$
3. $\|\mathbf{v}\|_\infty = 1$.



```
/// model
template<typename T>
T f(T p, T x);

/// residual of model predictions relative to observations
template<typename T>
VT<T> r(T p, VT<T> xobs, VT<T> yobs) {
  // conditions
  assert(xobs.size()==nobs);
  assert(xobs.size()==yobs.size());
  // return value
  VT<T> r_(nobs);
  int i=0;
  while (i<nobs) {
    // model error relative to observations
    r_(i)=f(p,xobs(i))-yobs(i);
    i=i+1;
  }
  return r_;
}
```


For the given Scenarios 1–4, the residual

$$\mathbf{r} : \mathbb{R} \times \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}} \to \mathbb{R}^{n_{\text{obs}}}$$

is differentiable wrt. p (also wrt. $\mathbf{x}$ and $\mathbf{y}$).

Its scalar components $r_i, i = 0, \ldots, n_{\text{obs}}$ can be differentiated individually yielding

$$\mathbf{r}' \equiv \frac{d\mathbf{r}}{dp} = \left(\frac{dr_i}{dp}\right) = \left(\frac{dr_i}{dp} \cdot p^{(1)}\right) = \left(\frac{dr_i}{dp}\right) \cdot p^{(1)} = \frac{d\mathbf{r}}{dp} \cdot p^{(1)}$$

for $p^{(1)} = 1$ by tangent AD.


```
/// derivative of residual wrt. model parameter
template<typename T>
VT<T> dr_dp(T p, VT<T> xobs) {
  // conditions
  assert(xobs.size()==nobs);
  // return value
  VT<T> drdp(nobs);
  // tangent type
  using TT=dco::gt1s<T>::type;
  int i=0;
  while (i<nobs) {
    // activation
    TT p_t=p, y_t;
    // seeding derivative of model parameter
    dco::derivative(p_t)=1;
    // overloaded computation of model value at current point of observation
    y_t=f<TT>(p_t,xobs(i));
    // harvesting derivative of residual wrt. model parameter at current point of observation
    drdp(i)=dco::derivative(y_t);
    i=i+1;
  }
  return drdp;
}
```


We aim to minimize the error of the model predictions wrt. the given data, that is,

$$\min_p e = \min_p \left(\mathbf{r}^T \cdot \mathbf{r}\right) = \min_p \|\mathbf{r}\|_2^2 = \min_p \left(\sum_{i=0}^{n_{\text{obs}}-1} \left(f(p, x_i) - y_i\right)^2\right) .$$

Note that, in the linear case,

$$\mathbf{r} = \left(\underbrace{\frac{df}{dp}}_{\equiv f'}(x_i) \cdot p - y_i\right)_{i=0,\ldots,n_{\text{obs}}-1}$$

yielding

$$\mathbf{r}' = (r_i')_{i=0,\ldots,n_{\text{obs}}-1} = (f'(x_i))_{i=0,\ldots,n_{\text{obs}}-1} .$$

Recall that the error

$$e = \sum_{i=0}^{n_{\text{obs}}-1} (f'(x_i) \cdot p - y_i)^2$$

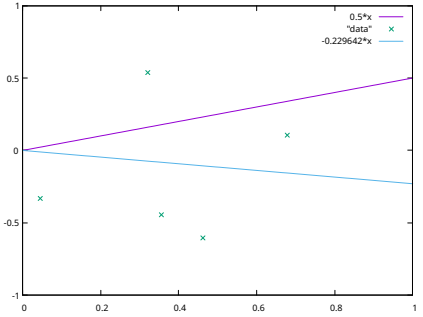
has a (unique) stationary point at

$$\frac{de}{dp} = 0$$

(first-order optimality condition). This stationary point is a (unique) minimum as

$$\frac{d^2e}{dp^2} > 0$$

(second-order optimality condition).



- The normal equation method for linear regression follows immediately from the first-order optimality criterion for the parameter of the objective.

  It is applicable to Scenario 1 and Scenario 2.

- Exploitation of the special structure of linear regression problems yields a simple explicit equation for the optimal parameter. Solutions for arbitrary instances of linear regression problems can be derived easily.

- In the simplest case $\mathbf{x} \cdot p \approx \mathbf{y}$,

$$p = \frac{\mathbf{x}^T \cdot \mathbf{y}}{\mathbf{x}^T \cdot \mathbf{x}} .$$

From

$$\frac{de}{dp} = \frac{d}{dp} \sum_{i=0}^{n_{\text{obs}}-1} (f'(x_i) \cdot p - y_i)^2 = 2 \cdot \sum_{i=0}^{n_{\text{obs}}-1} (f'(x_i) \cdot p - y_i) \cdot f'(x_i)$$
$$= 2 \cdot \sum_{i=0}^{n_{\text{obs}}-1} p \cdot r_i \cdot r_i - r_i \cdot y_i = 2 \cdot \left(p \cdot \mathbf{r'}^T \cdot \mathbf{r'} - \mathbf{r'}^T \cdot \mathbf{y} \right) = 0 ,$$

with

$$\frac{d^2e}{dp^2} = 2 \cdot \mathbf{r'}^T \cdot \mathbf{r'} = 2 \cdot \sum_{i=0}^{n_{\text{obs}}-1} r_i'^2 > 0 ,$$

follows the normal equation

$$p = \frac{\mathbf{r'}^T \cdot \mathbf{y}}{\mathbf{r'}^T \cdot \mathbf{r'}}$$

as the solution of the linear regression problem $p \cdot \mathbf{r'} \approx \mathbf{y}$.

Derive the normal equation methods for

$$\frac{dy}{dx} = p;\ y(0) = 0 \quad \Rightarrow \quad f(p, x) = p \cdot x$$

$$\frac{dy}{dx} = \frac{p}{x+1};\ y(0) = 0 \quad \Rightarrow \quad f(p, x) = p \cdot \log(x+1) .$$

Normal Equation Method

$g(p,x) = p$

$$\frac{dy}{dx} = g(p,x) = p \;, y(0) = 0 \qquad (\Rightarrow\; g'(x) = 1)$$

$$\Downarrow$$

$$f(p,x) = p \cdot x \qquad (\Rightarrow\; f'(x) = x)$$

$$\Downarrow$$

$$\mathbf{r}' = (f'(x_i)) = \mathbf{x}$$

$$\Downarrow$$

$$p = \frac{\mathbf{x}^T \cdot \mathbf{y}}{\mathbf{x}^T \cdot \mathbf{x}}$$

Normal Equation Method

$g(p,x) = \frac{p}{x+1}$

$$\frac{dy}{dx} = g(p,x) = \frac{p}{x+1} \;, y(0) = 0 \qquad \left(\Rightarrow\; g'(x) = \frac{1}{x+1}\right)$$

$$\Downarrow$$

$$f(p,x) = p \cdot \log(x+1) \qquad (\Rightarrow\; f'(x) = \log(x+1))$$

$$\Downarrow$$

$$\mathbf{r}' = (f'(x_i)) = (\log(x_i+1))$$

$$\Downarrow$$

$$p = \frac{\mathbf{r}'^T \cdot \mathbf{y}}{\mathbf{r}'^T \cdot \mathbf{r}'}$$

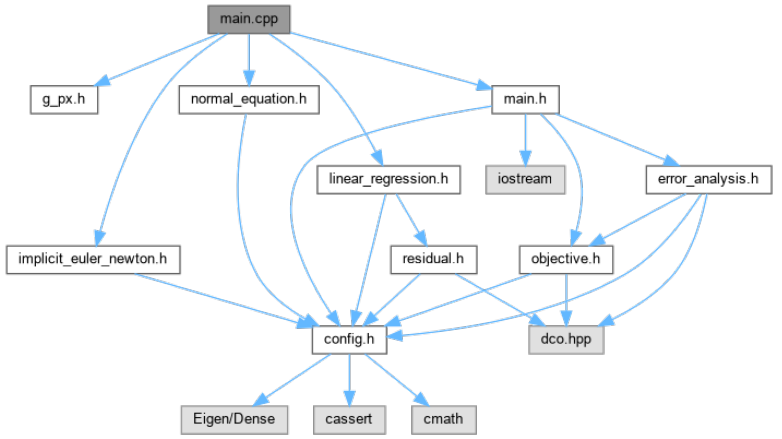
Normal Equation Method

Implementation

```
1  /// linear regression by normal equation method
2  template<typename T>
3  T linear_regression(VT<T> drdp, VT<T> yobs) {
4    // conditions
5    assert(drdp.size()==nobs);
6    assert(drdp.size()==yobs.size());
7    // normal equation
8    return drdp.dot(yobs)/drdp.dot(drdp);
9  }
10
11 /// stationary point of error wrt. model parameter by linear regression
12 template<typename T>
13 T fit(T p, VT<T> xobs, VT<T> yobs) {
14   // conditions
15   assert(xobs.size()==nobs);
16   assert(xobs.size()==yobs.size());
17   // run linear regression method
18   return linear_regression(dr_dp(p,xobs),yobs);
19 }
```


Normal Equation Method

Inspection of Source Code and Experiments



→ 1D/optimization/normal_equation/p[x]

Hands On! (Exercise 9)

1. Fit $y = p \cdot x + x$ to randomly generated data $(\mathbf{x}, \mathbf{y})$. Visualize the results.

2. Modify the code from the lecture such that explicit Euler integration is used instead of implicit Euler integration.

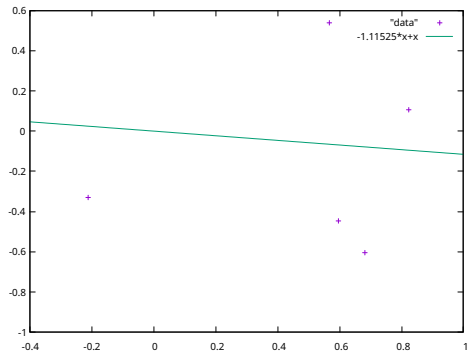
3. Modify the code from the lecture such that implicit Euler integration is combined with bisection instead of Newton's method. Approximate $\mathbf{r}'$ using central finite differences to achieve convergence. Run experiments with varying perturbations of p.

   Note that differentiability of all parts of the optimization method is highly desirable ⇒ differentiable programming.

Exercise 9

Notes on 1.

- Implement the normal equation as **void** fit(T &p, VT<T> x, VT<T> y).
- Use it to solve the linear regression problem for randomly generated data. Note that $y - x = p \cdot x$.
- Write a function **void** plot(VT<T> x, VT<T> y) to generate a text file data containing the random data.
- Use gnuplot to visualize the data as well as the algebraic model for the estimated parameter p.



→ exercises/9/1/

Exercise 9

Notes on 2.

- Copy all relevant header files from the sample code, that is,

  config.h linear_regression.h objective.h main.h
  normal_equation.h explicit_euler.h residual.h

  and, e.g., g_px.h into a separate directory.
- main.cpp becomes

```
1 #include "g_px.h"
2 #include "explicit_euler.h"
3 #include "normal_equation.h"
4 #include "linear_regression.h"
5 #include "main.h"
```

- Build and run to get

  e=1.29347
  p=−0.298436
  e=0.919193
  dedp=2.77556e−16
  ddedpp=1.1742

→ exercises/9/2/

Exercise 9

Notes on 3.

- Copy all relevant header files from the sample code, that is,

  config.h implicit_euler_bisection.h normal_equation.h
  objectve.h linear_regression.h residual.h main.h

  and, e.g., g_px.h into a separate directory.
- main.cpp becomes

```
1 #include "g_px.h"
2 #include "implicit_euler_bisection.h"
3 #include "normal_equation.h"
4 #include "linear_regression.h"
5 #include "main.h"
```

- Replace the call to dr_dp with a call to dr_dp_cfd for approximating $\mathbf{r}'$ with central finite differences.
- E.g., $\Delta p = 0.01$ makes the method converge to $p = -0.298394$ which matches the value obtained by implicit Euler with Newton root finding ($p = -0.298436$) relatively well.

→ exercises/9/3/

Again, we aim to find $p \in \mathbb{R}$ such that

$$\mathbf{r}' \cdot p \approx \mathbf{y} \,,$$

where $\mathbf{r}' = \mathbf{r}'(\mathbf{x}) \equiv \frac{d\mathbf{r}}{dp}(\mathbf{x})$ and $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}}$.

So far, this objective has been reached by minimizing the error

$$\sum_{i=0}^{n_{\text{obs}}-1} (f(p, x_i) - y_i)^2$$

yielding the gradient descent and normal equation methods.

A closer look at the geometry behind $\mathbf{r}' \cdot p \approx \mathbf{y}$ yields a better (in the vector case, numerically more stable than the normal equation) method.

- The Householder equation method follows from a geometric approach to the linear regression problem.
- A simple, yet elegant, algorithm can be derived.
- It is applicable to Scenario 1 and Scenario 2.
- The Householder method for linear regression generalizes to the vector case ($\mathbf{p} \in \mathbb{R}^{n_p}$, $n_p > 1$), where it promises better numerical stability than the normal equation method.

Operations on vectors are characterized by their effect on norms and relative positions, i.e. the angle spanned by two vectors in $\mathbb{R}^{n_{\text{obs}}}$. The law of cosines

$$\|\mathbf{z}\|^2 = \|\mathbf{r}' - \mathbf{y}\|^2 = \|\mathbf{r}'\|^2 + \|\mathbf{y}\|^2 - 2 \cdot \|\mathbf{r}'\| \cdot \|\mathbf{y}\| \cdot \cos(\theta)$$

follows from

$$\|\mathbf{z}\|^2 = (\|\mathbf{y}\| - d)^2 + h^2$$

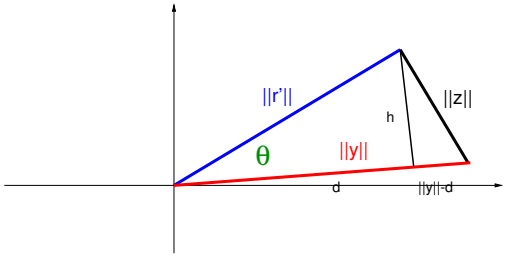
and

$$\|\mathbf{r}'\|^2 = d^2 + h^2$$

implying

$$\|\mathbf{z}\|^2 = \|\mathbf{r}'\|^2 + \|\mathbf{y}\|^2 - 2 \cdot \|\mathbf{y}\| \cdot d$$

and, hence, the law as $d = \|\mathbf{r}'\| \cdot \cos(\theta)$.



$\mathbf{r} = \mathbf{y} + \mathbf{z}$

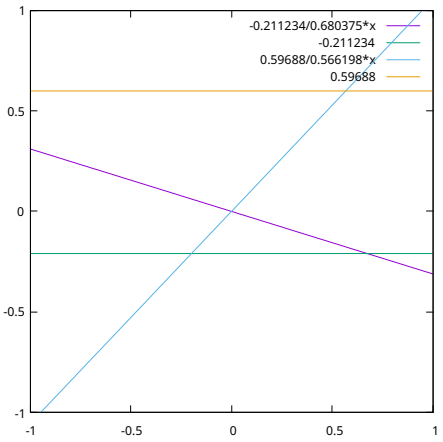
A similar argument holds for h outside of the triangle spanned by $\mathbf{r}', \mathbf{y}$ and $\mathbf{z}$.

The angle θ spanned by two vectors $\mathbf{r}', \mathbf{y} \in \mathbb{R}^{n_{\text{obs}}}$ is characterized by

$$\cos(\theta) = \frac{\mathbf{r}'^T \cdot \mathbf{y}}{\|\mathbf{r}'\| \cdot \|\mathbf{y}\|} \,.$$

The above follows from the law of cosines with

$$\begin{aligned}
\|\mathbf{r}' - \mathbf{y}\|^2 = (\mathbf{r}' - \mathbf{y})^T \cdot (\mathbf{r}' - \mathbf{y}) &= \sum_{i=0}^{n_{\text{obs}}-1} f'(x_i)^2 + y_i^2 - 2 \cdot f'(x_i) \cdot y_i \\
&= \sum_{i=0}^{n_{\text{obs}}-1} f'(x_i)^2 + \sum_{i=0}^{n_{\text{obs}}-1} y_i^2 - 2 \cdot \sum_{i=0}^{n_{\text{obs}}-1} f'(x_i) \cdot y_i \\
&= \|\mathbf{r}'\|^2 + \|\mathbf{y}\|^2 - 2 \cdot \mathbf{r}'^T \cdot \mathbf{y} \\
&= \|\mathbf{r}'\|^2 + \|\mathbf{y}\|^2 - 2 \cdot \|\mathbf{r}'\| \cdot \|\mathbf{y}\| \cdot \cos(\theta) \,.
\end{aligned}$$



$$\begin{aligned}\text{degrees} &= \frac{180^o}{\pi} \cdot \text{radians} \\ &= \frac{180^o}{4 \cdot \frac{\pi}{4}} \cdot \text{radians} \\ &= \frac{180^o}{4 \cdot \arctan(1)} \cdot \text{radians} \\ &= \frac{45^o}{\arctan(1)} \cdot \text{radians}\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}\theta &= \arccos\left(\frac{\mathbf{r'}^T \cdot \mathbf{y}}{\|\mathbf{r'}\| \cdot \|\mathbf{y}\|}\right) \quad \text{(radians)} \\ &= \frac{45^o}{\arctan(1)} \cdot \arccos\left(\frac{\mathbf{r'}^T \cdot \mathbf{y}}{\|\mathbf{r'}\| \cdot \|\mathbf{y}\|}\right) \\ &= 63.7589^o\end{aligned}$$

The following program computes the angle (in degrees) between two random vectors in $\mathbb{R}^2$ and it generates a gnuplot command for visualization.

```
1  using VT=Eigen::VectorX<double>;
2
3  // slopes of linear functions and lengths at points
4  // of intersection with v(1) and w(1), respectively
5  void plot(VT v, VT w) {
6    cout << "set size square; plot [-1:1] [-1:1] "
7         << v(1) << '/' << v(0) << "*x, " << v(1) << ", "
8         << w(1) << '/' << w(0) << "*x, " << w(1) << endl;
9  }
10
11 int main() {
12   int n=2;
13   // two random vectors in R^2
14   VT v=VT::Random(n), w=VT::Random(n);
15   plot(v,w);
16   // radians to degrees
17   cout << (45/atan(1.))*acos(v.dot(w)/(v.norm()*w.norm())) << endl;
18   return 0;
19 }
```

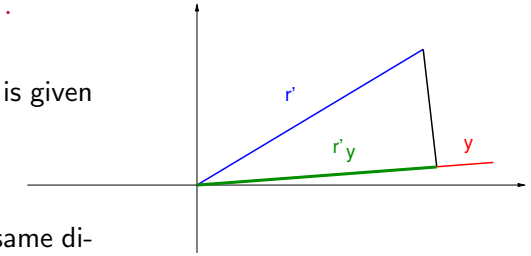

From the law of cosines follows immediately, that the scalar projection of $\mathbf{r}' \in \mathbb{R}^{n_{\text{obs}}}$ onto $\mathbf{y} \in \mathbb{R}^{n_{\text{obs}}}$ is given by

$$\|\mathbf{r}'_\mathbf{y}\| = \frac{\mathbf{r'}^T \cdot \mathbf{y}}{\|\mathbf{y}\|} = \cos(\theta) \cdot \|\mathbf{r}'\| \, .$$

The corresponding vector projection is given by

$$\mathbf{r}'_\mathbf{y} = \|\mathbf{r}'_\mathbf{y}\| \cdot \frac{\mathbf{y}}{\|\mathbf{y}\|}$$

(vector of length $\|\mathbf{r}'_\mathbf{y}\|$ pointing into same direction as $\mathbf{y}$).

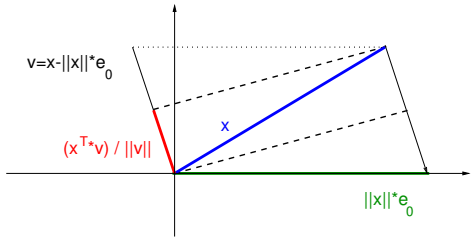


The scalar projection of $\mathbf{r}' \in \mathbb{R}^{n_{\text{obs}}}$ onto the i-th standard basis vector $\mathbf{e}_i \in \mathbb{R}^{n_{\text{obs}}}$ is given by $\|\mathbf{r}'_{\mathbf{e}_i}\| = f'(x_i)$.

The corresponding vector projection is given by $\mathbf{r}'_{\mathbf{e}_i} = f'(x_i) \cdot \mathbf{e}_i$.

To transform $\mathbf{r}' \in \mathbb{R}^{n_{\text{obs}}}$ into $\|\mathbf{r}'\| \cdot \mathbf{e}_0$, twice the vector projection of $\mathbf{r}'$ onto $\mathbf{v} = \mathbf{r}' - \|\mathbf{r}'\| \cdot \mathbf{e}_0$ needs to be subtracted from $\mathbf{r}'$.

$$\begin{aligned}\mathbf{r}'_\mathbf{v} &= \frac{\mathbf{r'}^T \cdot \mathbf{v}}{\|\mathbf{v}\|} \cdot \frac{\mathbf{v}}{\|\mathbf{v}\|} = \frac{\mathbf{r'}^T \cdot \mathbf{v} \cdot \mathbf{v}}{\|\mathbf{v}\|^2} \\ &= \frac{\mathbf{r'}^T \cdot \mathbf{v} \cdot \mathbf{v}}{\|\mathbf{v}\|^2} = \frac{\mathbf{v} \cdot \mathbf{r'}^T \cdot \mathbf{v}}{\|\mathbf{v}\|^2} \\ &= \frac{\mathbf{v} \cdot \mathbf{v}^T \cdot \mathbf{r}'}{\|\mathbf{v}\|^2} \, .\end{aligned}$$



It follows

$$\|\mathbf{r}'\| \cdot \mathbf{e}_0 = \mathbf{r}' - 2 \cdot \mathbf{r}'_\mathbf{v} = \mathbf{r}' - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T \cdot \mathbf{r}'}{\|\mathbf{v}\|^2} \, .$$

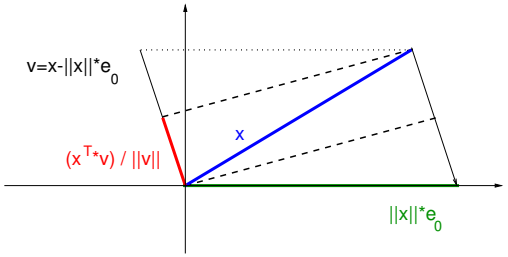
```
x^T=2 1
||x||=2.23607

v^T=(x-||x||*e_0)^T=-0.236068 1

v^T*((x^T*v)/||v||)=-0.121278 0.513743

H=
 0.894427 0.447214
 0.447214 -0.894427

x^T=(H*x)^T= 2.23607 3.33067e-16
```



→ misc/householder_1D/householder_reflection_verbose.cpp

From the above it follows that

$$\|\mathbf{r}'\| \cdot \mathbf{e}_0 = \mathbf{r}' - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T \cdot \mathbf{r}'}{\|\mathbf{v}\|^2} = \mathbf{r}' - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}} \cdot \mathbf{r}' = \left(I_m - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}} \right) \cdot \mathbf{r}' \ .$$

Note that a unit vector (length equal to one) in the direction of a given vector $\mathbf{v} \in \mathbb{R}^{n_{\text{obs}}}$ is obtained by dividing $\mathbf{v}$ by its norm, i.e,

$$\mathbf{e}_\mathbf{v} = \frac{\mathbf{v}}{\|\mathbf{v}\|} \ .$$

The Householder matrix

$$H = I - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}}$$

is symmetric ($H = H^T$) and orthogonal ($H^{-1} = H^T$); more on this later.

The product of an orthogonal matrix $H \in \mathbb{R}^{n_{\text{obs}} \times n_{\text{obs}}}$ with two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{n_{\text{obs}}}$ preserves their lengths as

$$(H \cdot \mathbf{v})^T \cdot (H \cdot \mathbf{v}) = \mathbf{v}^T \cdot (H^T \cdot H) \cdot \mathbf{v} = \mathbf{v}^T \cdot (H^{-1} \cdot H) \cdot \mathbf{v} = \mathbf{v}^T \cdot \mathbf{v}$$

as well as the angle θ spanned by them as

$$\begin{aligned}
\frac{(H \cdot \mathbf{u})^T \cdot (H \cdot \mathbf{v})}{\|(H \cdot \mathbf{u}\|_2 \cdot \|(H \cdot \mathbf{v}\|_2} &= \frac{(H \cdot \mathbf{u})^T \cdot (H \cdot \mathbf{v})}{(H \cdot \mathbf{u})^T \cdot (H \cdot \mathbf{u}) \cdot (H \cdot \mathbf{v})^T \cdot H \cdot \mathbf{v}} \\
&= \frac{\mathbf{u}^T \cdot (H^T \cdot H) \cdot \mathbf{v})}{(\mathbf{u}^T \cdot (H^T \cdot H) \cdot \mathbf{u}) \cdot (\mathbf{v}^T \cdot (H^T \cdot H) \cdot \mathbf{v})} \\
&= \frac{\mathbf{u}^T \cdot \mathbf{v}}{\mathbf{u}^T \cdot \mathbf{u} \cdot \mathbf{v}^T \cdot \mathbf{v}} \\
&= \frac{\mathbf{u}^T \cdot \mathbf{v}}{\|\mathbf{u}\|_2 \cdot \|\mathbf{v}\|_2} = \cos\theta \ .
\end{aligned}$$

The following program computes the Householder reflection of a random vector onto the first standard basis direction.

```
template<typename T>
using VT=Eigen::VectorX<T>;

template<typename T>
VT<T> Householder(VT<T> x) {
  VT<T> v=x; v(0)=v(0)-x(0)/fabs(x(0))*x.norm();
  x=x-2*v*(v.dot(x)/v.dot(v));
  return x;
}
```

Explicit computation of the Householder matrix can and should be avoided.

→ misc/householder_1D/householder_reflection.cpp

Householder reflection can be used to solve the scalar linear regression problem. It transforms $\mathbf{r}' \cdot p \approx \mathbf{y}$ into

$$\|\mathbf{r}'\|_2 \cdot \mathbf{e}_0 \cdot p = H \cdot \mathbf{r}' \cdot p \approx H \cdot \mathbf{y}$$

(reflection of $\mathbf{r}'$ onto the first standard basis direction ($\mathbf{e}_0$) implies orthogonality to remaining standard basis directions) using the orthogonal matrix $H \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}}}$.

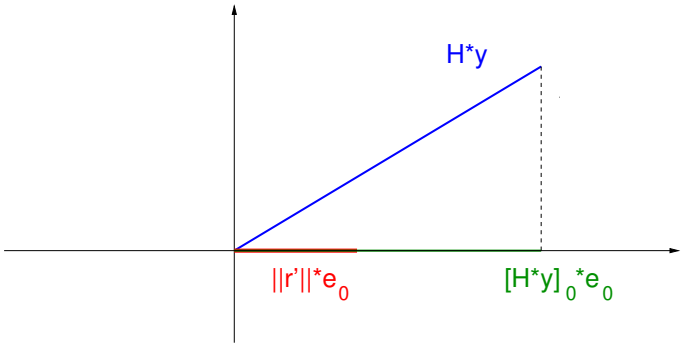
Applicability to linear regression follows from

$$\|\mathbf{r}'\|_2 \cdot \mathbf{e}_0 \cdot p \approx H \cdot \mathbf{y} \quad \Rightarrow \quad p = \frac{[H \cdot \mathbf{y}]_0}{\|\mathbf{r}'\|_2}$$

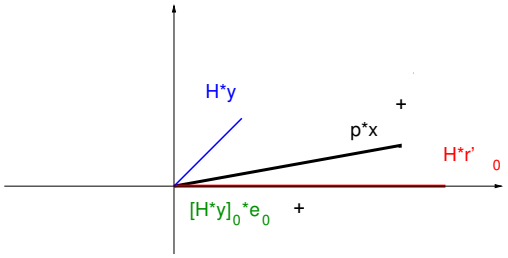
where $[H \cdot \mathbf{y}]_i$ denotes the i-th entry of the vector defined by the bracketed expression. For simplicity, we typically write $[\mathbf{v}]_i \equiv v_i$.

The minimal distance between $H \cdot \mathbf{y}$ and the, e.g., first standard basis direction is equal to the geometric projection of $H \cdot \mathbf{y}$ onto that direction:

$$\|\mathbf{r}'\|_2 \cdot \mathbf{e}_0 \cdot p \approx H \cdot \mathbf{y} \quad \Rightarrow \quad p = \frac{[H \cdot \mathbf{y}]_0}{\|\mathbf{r}'\|_2}$$




```
1  r'^T=2 1
2
3  y^T= 0.680375 -0.211234
4
5  H=
6   0.894427 0.447214
7   0.447214 -0.894427
8
9  (H*r')^T= 2.23607 3.33067e-16
10
11 (H*y)^T= 0.51408 0.493207
12
13 p=0.229903
```



→ misc/householder_1D/householder_regression_verbose.cpp

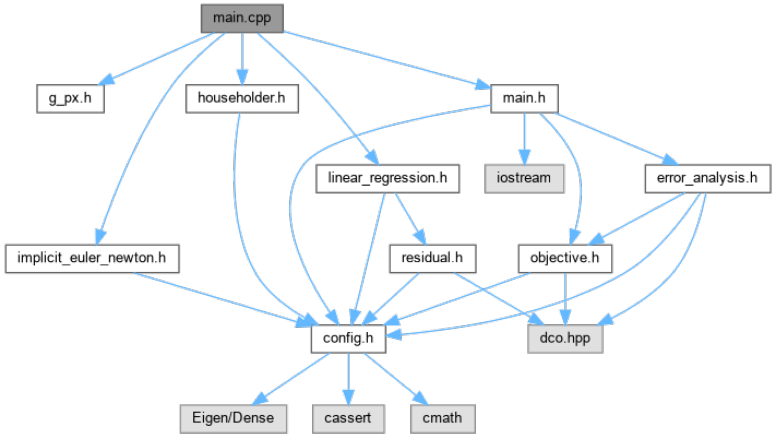
For $\mathbf{v} = \mathbf{r}' - \|\mathbf{r}'\| \cdot \mathbf{e}_0$,

$$p = \frac{[H \cdot \mathbf{y}]_0}{\|\mathbf{r}'\|_2} = \frac{\left[\left(I - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}}\right) \cdot \mathbf{y}\right]_0}{\|\mathbf{r}'\|_2} = \frac{\left[\mathbf{y} - 2 \cdot \frac{\mathbf{v} \cdot (\mathbf{v}^T \cdot \mathbf{y})}{\mathbf{v}^T \cdot \mathbf{v}}\right]_0}{\|\mathbf{r}'\|_2},$$

yielding

```
1  /// linear regression by Householder reflection
2  template<typename T>
3  T linear_regression(VT<T> drdp, VT<T> yobs) {
4    // conditions
5    assert(drdp.size()==nobs);
6    assert(drdp.size()==yobs.size());
7    // Householder reflection
8    VT<T> v=drdp;
9    v(0)=v(0)-drdp(0)/fabs(drdp(0))*drdp.norm();
10   VT<T> n=yobs-2*v/(v.dot(v))*(v.dot(yobs));
11   return n(0)/drdp.norm();
12 }
```

Again, explicit computation of the Householder matrix is avoided.



→ 1D/optimization/householder/p[x]

Hands On! (Exercise 10)

1. Fit $y = p \cdot x - x$ to randomly generated data $(\mathbf{x}, \mathbf{y})$. Visualize the results.

2. Modify the code from the lecture such that explicit Euler integration is used instead of implicit Euler integration.

Note: Replace the normal equation method by the Householder method in Exercise 9 (1. and 2.).

→ exercises/10/

Outline

- "Finding a stationary point of a local quadratic approximation of the objective" appears to be another reasonable heuristic for iterative minimization.
- The minimum of this local approximation can be computed exactly.
- Convergence of the Newton method is quadratic (fast) in a neighborhood of the solution. It is not guaranteed.
- Good starting values (close to the solution) may be essential. They can be computed by some other *preprocessing* method, e.g., by a few gradient descent steps.
- First and second derivatives of the objective are required. Their accuracy can be crucial for the performance of the Newton method. AD becomes the differentiation method of choice.

We aim to minimize the least squares error of the model predictions wrt. to the given data, that is,

$$\min_p e = \min_p \left(\sum_{i=0}^{n_{\text{obs}}-1} (f(p, x_i) - y_i)^2 \right) .$$

Similar to gradient descent, the Newton method is applicable to general unconstrained nonlinear optimization problems.

It can be derived from a second-order truncated Taylor expansion of $e = e(p, \mathbf{x}, \mathbf{y})$ in direction Δp.

Similarly, a first-order truncated Taylor expansion of the first-order optimality condition $e' = 0$ in direction Δp can be used.

The truncated Taylor expansion

$$e(p + \Delta p) \approx_{O(\Delta p^n)} e(p) + \sum_{k=1}^{n-1} \frac{1}{k!} \cdot \frac{d^k e}{dp^k} \cdot \Delta p^k .$$

yields a minimization problem for a quadratic approximation (truncation after Δp^2-term) of the objective at the current p as

$$\min_{\Delta p} \left(e(p) + e' \cdot \Delta p + \frac{1}{2} \cdot e'' \cdot \Delta p^2 \right) .$$

The correponding first-order optimality condition implies the Newton step Δp as the root of the first derivative of the local objective, that is,

$$e' + \cdot e'' \cdot \Delta p = 0 \quad \Rightarrow \quad \Delta p = -\frac{e'}{e''}$$

A Taylor expansion of the first-order optimality condition

$$e' = 0$$

yields

$$e'(p + \Delta p) \approx_{O(\Delta p^n)} e'(p) + \sum_{k=1}^{n-1} \frac{1}{k!} \cdot \frac{d^k e'}{dp^k} \cdot \Delta p^k$$

and, hence, a linear approximation (truncation after Δp-term; also: linearization) as

$$e' + e'' \cdot \Delta p = 0 \quad \Rightarrow \quad \Delta p = -\frac{e'}{e''} .$$

Validation of a local minimum at requires $e'' > 0$. Similarly, a local maximum is found if $e'' < 0$. $e'' = 0$ indicates a degenerate stationary point.

The Newton method can be written as the fixpoint iteration

$$p = \check{e}(p) = p - \frac{e'(p)}{e''(p)}$$

which converges locally if $\check{e}$ is locally contractive, that is,

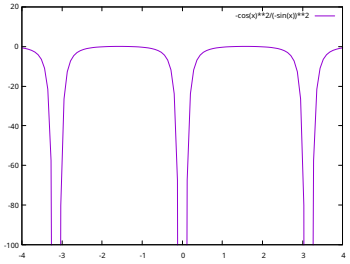
$$\left| \frac{d\check{e}}{dp} \right| = \left| 1 - \frac{e''(p)}{e''(p)} + \frac{e'(p) \cdot e'''(p)}{e''(p)^2} \right| = \left| \frac{e'(p) \cdot e'''(p)}{e''(p)^2} \right| \leq 1 .$$

Convergence of the Newton method does not necessarily require local convergence in each iteration. Locally divergent steps may lead into areas of contraction.

Consider the computation of a local minimum of $e(p) = \sin(p)$ using the Newton method.

Obviously, $e'(p) = \cos(p)$, $e''(p) = -\sin(p)$, and $e'''(p) = -\cos(p)$, yielding

$$\frac{e'(p) \cdot e'''(p)}{e''(p)^2} = \frac{\cos(p) \cdot (-\cos(p)}{(-\sin(p))^2}$$



Start values in the neighborhoods of $k \cdot \pi, k = ..., -1, 0, 1, ...$ result in local divergence, e.g., starting from $x = 0.1$ the local maximum at $x = 3 \cdot \pi + \frac{\pi}{2} \approx 11$ is found.

First-order tangents

$$e''^{(1)} : \mathbb{R} \times \mathbb{R} \to \mathbb{R} : \quad e''^{(1)} = e''^{(1)}(p, p^{(1)})$$

of[16]

$$e'' : \mathbb{R} \to \mathbb{R} : \quad e'' = e''(p)$$

yield third derivatives of

$$e : \mathbb{R} \to \mathbb{R} : \quad e = e(p)$$

as

$$e''^{(1)} = e''^{(1)}(p, p^{(1)}) \equiv e''' \cdot p^{(1)} ,$$

where $p^{(1)} = 1$ yields

$$e''' \equiv \frac{d^3 e}{dp^3} \in \mathbb{R} .$$

[16] Dependence of e'' on $\mathbf{x}$ is omitted to simplify the notation.

Alternatively, third-order finite differences could be employed, However ...

We use central finite differences to approximate derivatives of

$$f(x) = e^x$$

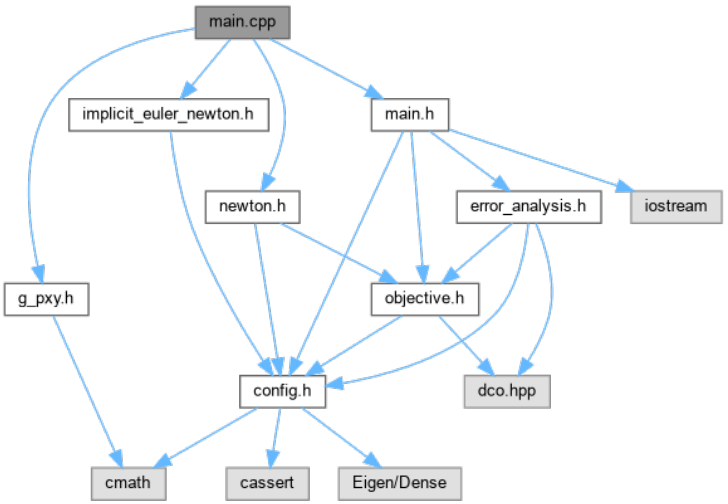
at $x = 1$ up to order eight. The following results are obtained.

$$\tilde{f}^{[1]}(1) = 2.71828 \quad \tilde{f}^{[2]}(1) = 2.7183$$
$$\tilde{f}^{[3]}(1) = 2.71822 \quad \tilde{f}^{[4]}(1) = 2.72783$$
$$\tilde{f}^{[5]}(1) = 2.77225 \quad \tilde{f}^{[6]}(1) = 2.82998$$
$$\tilde{f}^{[7]}(1) = 3.12907 \quad \tilde{f}^{[8]}(1) = 3.12907$$

In practice, even second derivatives can become hard to approximate well.

→ misc/higher_order_df/cfd.cpp


```
1  /// stationary point of error wrt. model parameter by Newton method
2  template<typename T>
3  T fit(T p, VT<T> xobs, VT<T> yobs) {
4    // conditions
5    assert(xobs.size()==nobs);
6    assert(xobs.size()==yobs.size());
7    // derivative of error wrt. model parameter
8    T dedp=de_dp(p,xobs,yobs);
9    do {
10     // Newton step requires second derivative of error wrt. model parameter
11     p=p-dedp/dde_dp_dp(p,xobs,yobs);
12     dedp=de_dp(p,xobs,yobs);
13   } while (fabs(dedp)>eps);
14   return p;
15 }
```



→ 1D/optimization/newton/*

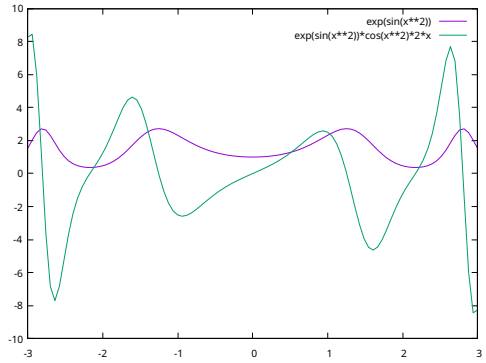
Hands On! (Exercise 11)

Use the Newton method to compute local minima of

$$y = e^{\sin(p \cdot x^2)} .$$

Implement the required first and second derivatives

1. symbolically
2. numerically (central finite differences)
3. algorithmically (dco/c++).



Use AD to check for local convergence of the fixed-point iteration (third derivative required).

Experiment with different starting points and values of the parameter p. Record the number of iterations performed and compare with gradient descent.

Exercise 11
Notes

- Implement the Newton method (copy'n'paste from sample code).
- Implement the first, `T df_dx(T p, T x)`, second, `T ddf_dx_dx(T p, T x)`, and third, `T dddf_dx_dx_dx(T p, T x)`, derivatives of

```
1 template<typename T>
2 T f(T p, T x) { return exp(sin(p*pow(x,2))); }
```

- Build and run to see

```
-++++
x=2.1708
y=0.367879
dfdx=3.23341e-14
ddfdxx=6.93436
```

  for $p = x = 1$. The number of iterations is equal to the sum of the numbers of +'s and −'s printed. The former indicate local contractiveness while the latter do not.
- Play with p and x. Compare with the gradient descent method.

→ exercises/11/

Outline

- Linearization of the residual yields an iterative method for nonlinear regression problems (Scenario 3 and Scenario 4).
- Convergence is not guaranteed. Conditions apply.
- The iterative updates are computed as solutions to linear regression problems. Both normal equation and Householder methods can be applied.
- Solutions can differ from those computed by, e.g., gradient descent or Newton methods.

Reformulation of the first-order optimality condition

$$e' \equiv \frac{de}{dp} = \frac{d\|\mathbf{r}\|_2^2}{dp} = 0$$

in terms of a linearization in direction Δp of the residual $\mathbf{r} = \mathbf{r}(p, \mathbf{x}, \mathbf{y})$ as

$$\frac{d\|\mathbf{r} + \frac{d\mathbf{r}}{dp} \cdot \Delta p\|_2^2}{d\Delta p} = \frac{d\|\mathbf{r} + \mathbf{r}' \cdot \Delta p\|_2^2}{d\Delta p} = 0$$

yields an iterative optimization scheme for p as

$$p := p + \Delta p \; .$$

The first derivative $\mathbf{r}' \in \mathbb{R}^{n_{\text{obs}}}$ of the residual $\mathbf{r} \in \mathbb{R}^{n_{\text{obs}}}$ with respect to p is required. It can be computed symbolically as well as by finite difference approximation or by AD.


```
1  // linear regression method (normal equation or Householder)
2  template<typename T>
3  T linear_regression(VT<T> drdp, VT<T> r);
4
5  /// stationary point of error wrt. model parameter by nonlinear regression
6  template<typename T>
7  T fit(T p, VT<T> xobs, VT<T> yobs) {
8    // conditions
9    assert(xobs.size()==nobs);
10   assert(xobs.size()==yobs.size());
11   T delta_p;
12   do {
13     // linearization
14     delta_p=linear_regression(dr_dp(p,xobs),r(p,xobs,yobs));
15     // iterative update
16     p=p-delta_p;
17   } while (fabs(delta_p)>eps);
18   return p;
19 }
```


The linear regression problem

$$\mathbf{r}' \cdot \Delta p \approx -\mathbf{r} \; , \quad \mathbf{r}, \mathbf{r}' \in \mathbb{R}^{n_{\text{obs}}}, \; \Delta p \in \mathbb{R}$$

can be solved by both the normal equation and Householder methods.

As before, local convergence of the fixed-point iteration

$$p = \check{e}(p) \equiv p + \Delta p$$

requires

$$\|\check{e}'(p)\| \leq 1 \; .$$

The linear regression problem

$$\mathbf{r}' \cdot \Delta p \approx -\mathbf{r}$$

yields the normal equation

$$\mathbf{r}'^T \cdot \mathbf{r}' \cdot \Delta p = -\mathbf{r}'^T \cdot \mathbf{r}$$

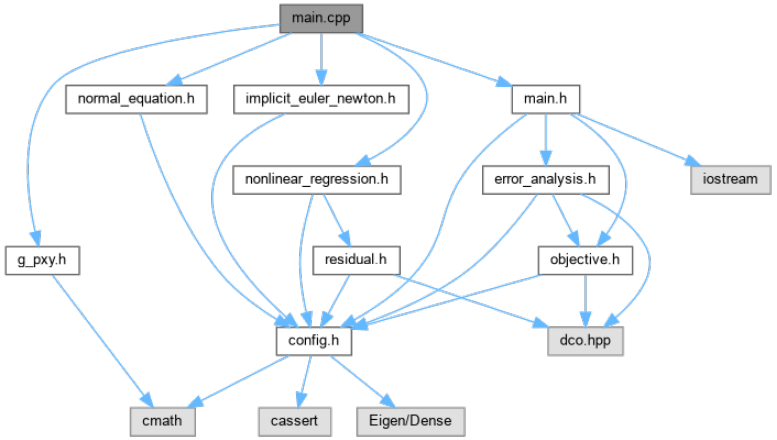
and hence the solution

$$\Delta p := -\frac{\mathbf{r}'^T \cdot \mathbf{r}}{\mathbf{r}'^T \cdot \mathbf{r}'} .$$

$$\begin{aligned} 0 = \frac{d\|\mathbf{r}' \cdot \Delta p - \mathbf{r}\|_2^2}{d\Delta p} &= \frac{d\left[\sum_{i=0}^{n_{\text{obs}}-1}(r_i' \cdot \Delta p - r_i)^2\right]}{d\Delta p} \\ &= 2 \cdot \sum_{i=0}^{n_{\text{obs}}-1} r_i' \cdot (r_i' \cdot \Delta p - r_i) = \sum_{i=0}^{n_{\text{obs}}-1} r_i'^2 \cdot \Delta p - r_i' \cdot r_i \\ &= \Delta p \cdot \sum_{i=0}^{n_{\text{obs}}-1} r_i'^2 - \sum_{i=0}^{n_{\text{obs}}-1} r_i' \cdot r_i = \Delta p \cdot \mathbf{r}'^T \cdot \mathbf{r}' - \mathbf{r}'^T \cdot \mathbf{r} \end{aligned}$$

implies

$$\Delta p = \frac{\mathbf{r}'^T \cdot \mathbf{r}}{\mathbf{r}'^T \cdot \mathbf{r}'} .$$



→ 1D/optimization/normal_equation/p[x]y

Solution of the linear regression problem

$$\mathbf{r}' \cdot \Delta p \approx -\mathbf{r}$$

by the Householder method transforms the vector

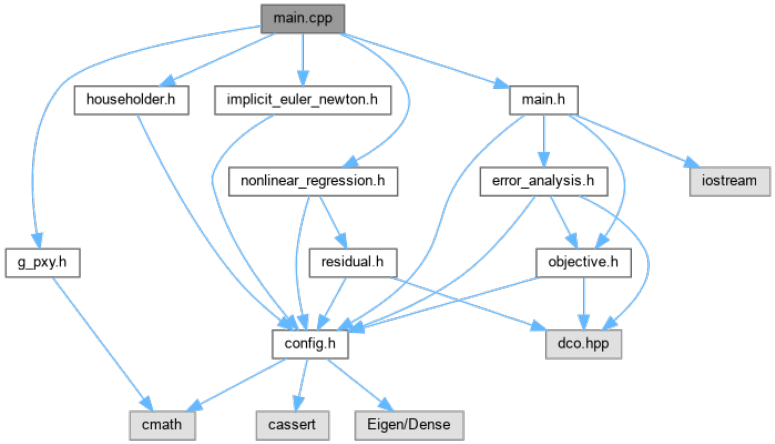
$$\mathbf{r}' \in \mathbb{R}^{n_{\text{obs}}} \quad \text{into} \quad H \cdot \mathbf{r}' = \|\mathbf{r}'\|_2 \cdot \mathbf{e}_0$$

followed by the solution of

$$\|\mathbf{r}'\|_2 \cdot \mathbf{e}_0 \cdot \Delta p \approx -H \cdot \mathbf{r}$$

yielding

$$\Delta p = -\frac{[H \cdot \mathbf{r}]_0}{\|\mathbf{r}'\|_2} .$$



→ 1D/optimization/householder/p[x]y

Hands On! (Exercise 12)

Fit

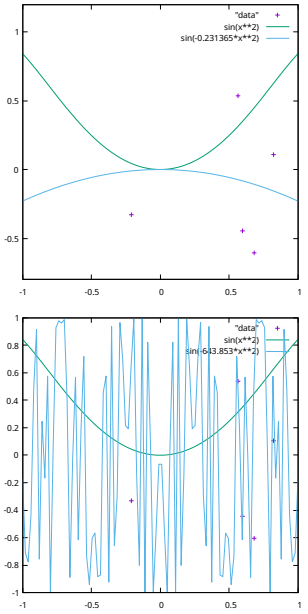
$$y = \sin(p \cdot x^2) \, .$$

to randomly generated data $(\mathbf{x}, \mathbf{y})$ using the

- nonlinear normal equation
- nonlinear Householder
- gradient descent (all: top plot)
- Newton (bottom plot)

methods. Perform data error analysis using central finite differences, tangent and adjoint AD, and symbolic differentiation.

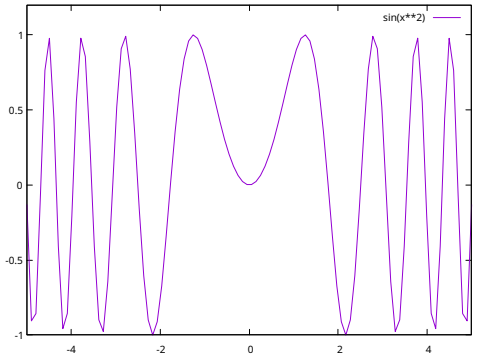
→ exercises/12/



(Exercise 12)
Notes

Copy/paste from the sample code is your friend!

Blame the first Newton step ...



Outline

- Both the state $\mathbf{y} \in \mathbb{R}^n$ of the system to be modelled and its parameter $\mathbf{p} \in \mathbb{R}^{n_p}$ become vectors.
- Many aspects carry over naturally from the scalar case.
- The main difference lies in linear equations becoming systems of linear equations. The former were solved trivially by scalar division. The latter require inversion of system matrices, which can yield a number of challenges due to poor conditioning.
- This course assumes that this is not the case. Algorithmic aspects of the previously introduced numerical methods will be generalized for the vector case.

- Two models defined as symbolic solutions $\mathbf{y} \in \mathbb{R}^n$ of initial value problems parameterized by $\mathbf{p} \in R^{n_p}$ are introduced.
- They turn out to be linear and nonlinear in $\mathbf{p}$, respectively. Linear as well as nonlinear regression and general-purpose nonlinear optimization methods shall be discussed.
- Both explicit and implicit Euler methods are employed for numerical approximation.
- The latter yields a system of nonlinear equations to be solved by the Newton method for vector root finding.
- All derivatives involved are computed by AD.

Outline

The model

$$f : \mathbb{R}^{n_p} \times \mathbb{R} \to \mathbb{R}^n : \quad \mathbf{y} = f(\mathbf{p}, x)$$

is defined as the solution of the initial value problem

$$\frac{d\mathbf{y}}{dx} = g(\mathbf{p}, x, \mathbf{y}); \ \mathbf{y}(\mathbf{p}, 0) = 0 \ .$$

x is often interpreted as time. Hence, it remains scalar.

Two scenarios are constructed,[17] yielding a linear (in $\mathbf{p}$; Scenario 5) and a nonlinear (Scenario 6) model, respectively.

[17] Not representing any real-world application, they are built for illustration only.

The model $f : \mathbb{R}^{n_p} \times \mathbb{R} \to \mathbb{R}^n : \mathbf{y} = f(\mathbf{p}, x)$ is linear in $\mathbf{p}$ if

$$f(\mathbf{v} + \mathbf{u}, x) = f(\mathbf{v}, x) + f(\mathbf{u}, x) \quad \text{and} \quad f(\alpha \cdot \mathbf{v}, x) = \alpha \cdot f(\mathbf{v}, x)$$

for all $\mathbf{v}, \mathbf{u} \in \mathbb{R}^{n_p}$ and $\alpha \in \mathbb{R}$.

Models of the form $f(\mathbf{p}, x) = A \cdot \mathbf{p} + \mathbf{b}$ with $A = A(x) \in \mathbb{R}^{n \times n_p}$ and $\mathbf{b} = \mathbf{b}(x) \in \mathbb{R}^n$ are affine. Linear functions are affine with $\mathbf{b} = 0$; see below.

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

Roots of affine functions are defined implicitly as solutions of systems of linear equations. Conditions apply.

Show that $f(\mathbf{p}, x) = A(x) \cdot \mathbf{p}$ with $A = A(x) \in \mathbb{R}^{n \times n_p}$ is linear in $\mathbf{p} \in \mathbb{R}^{n_p}$.

Proof:

$$f(\mathbf{v} + \mathbf{u}) = A \cdot (\mathbf{v} + \mathbf{u}) = A \cdot \mathbf{v} + A \cdot \mathbf{u} = f(\mathbf{v}) + f(\mathbf{u})$$
$$f(\alpha \cdot \mathbf{v}) = A \cdot \alpha \cdot \mathbf{a} = \alpha \cdot A \cdot \mathbf{v} = \alpha \cdot f(\mathbf{v})$$

for all $\mathbf{v}, \mathbf{u} \in \mathbb{R}^{n_p}$ and $\alpha \in \mathbb{R}$.

We consider the solution of systems

$$A \cdot \mathbf{p} + \mathbf{b} = 0$$

of n linear equations. The following statements are equivalent:

- A is regular (also: invertible).
- A solution to $A \cdot \mathbf{p} + \mathbf{b} = 0$ exists for any $\mathbf{b}$.
- A solution to $A \cdot \mathbf{p} + \mathbf{b} = 0$ is unique, if it exists.
- $\forall \mathbf{p} : A \cdot \mathbf{p} = 0 \Rightarrow \mathbf{p} = 0$
- the columns (rows) of A are linearly independent
- the inverse A^{-1} of A exists and $A^{-1} \cdot A = A \cdot A^{-1} = I_n$, where $I_n \in \mathbb{R}^{n \times n}$ denotes the identity in $\mathbb{R}^n$, i.e, $\forall \mathbf{v} \in \mathbb{R}^n : \; I_n \cdot \mathbf{v} = \mathbf{v}$.
- $\det(A) \neq 0$ (nonzero determinant of A)

Scalar models yield linear equations $a \cdot p + b = 0$. They are easily solved and numerically stable. Small errors in $b \in \mathbb{R}$ (as well as in $a \in \mathbb{R}$) imply small (same order) errors in $p \in \mathbb{R}$.

Vector models yield systems of linear equations $A \cdot \mathbf{p} + \mathbf{b} = 0$. Small changes in $\mathbf{b} \in \mathbb{R}^{n_p}$ can yield large changes in $\mathbf{p} \in \mathbb{R}^{n_p}$ due to poor conditioning of $A \in \mathbb{R}^{n_p \times n_p}$. E.g.,

$$\begin{matrix} p_0 + p_1 = 2 \\ p_0 + 1.001 \cdot p_1 = 2 \end{matrix} \quad \Rightarrow \quad \mathbf{p} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

while

$$\begin{matrix} p_0 + p_1 = 2 \\ p_0 + 1.001 \cdot p_1 = 2.001 \end{matrix} \quad \Rightarrow \quad \mathbf{p} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} .$$

All systems of linear equations to be solved in the following are well-conditioned, allowing us to focus on algorithmic instead of numerical issues.

The model

$$f : \mathbb{R}^{n_p} \times \mathbb{R} \to \mathbb{R}^n : \quad \mathbf{y} = f(\mathbf{p}, x)$$

is defined as the solution of the initial value problem

$$\frac{d\mathbf{y}}{dx} = g(\mathbf{p}, x);\ \mathbf{y}(\mathbf{p}, 0) = 0\ ,$$

where

$$g(\mathbf{p}, x) = \frac{\mathbf{p}_1}{x+1} + \mathbf{p}_2$$

for $n_p = 2 \cdot n$ and

$$\mathbf{p} = \begin{pmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \end{pmatrix}$$

with $\mathbf{p}_i \in \mathbb{R}^n$, $i = 1, 2$.

Independence of linear (in $\mathbf{p}$) g on $\mathbf{y}$ yields a linear (in $\mathbf{p}$) model (similar to Scenarios 1 and 2).


```
1  /// linear (in p) right-hand side of the explicit ordinary differential equation independent of y
2  template<typename T>
3  VT<T> g(VT<T> p, T x, VT<T> /*y*/) {
4    // conditions
5    assert(p.size()==np); assert(np==2*n);
6    // result
7    VT<T> dydx(n);
8    // evaluation of result
9    int i=0;
10   while (i<n) {
11     dydx(i)=p(i)/(x+1)+p(n+i);
12     i=i+1;
13   }
14   return dydx;
15 }
```


The model

$$f : \mathbb{R}^{n_p} \times \mathbb{R} \to \mathbb{R}^n : \quad \mathbf{y} = f(\mathbf{p}, x)$$

is defined as the solution of the initial value problem

$$\frac{d\mathbf{y}}{dx} = g(\mathbf{p}, x, \mathbf{y});\ \mathbf{y}(\mathbf{p}, 0) = 0\ ,$$

where $n_p = 2 \cdot n$ and

$$\left[\frac{d\mathbf{y}}{dx}\right]_i = \begin{cases} \sum_{j=i}^{i+1} \frac{p_j \cdot x}{y_j+1} + p_{n+i} & i = 0 \\ \sum_{j=i-1}^{i} \frac{p_j \cdot x}{y_j+1} + p_{n+i} & i = n-1 \\ \sum_{j=i-1}^{i+1} \frac{p_j \cdot x}{y_j+1} + p_{n+i} & \text{otherwise}\ . \end{cases}$$

Dependence of g on $\mathbf{y}$ yields a nonlinear (in $\mathbf{p}$) model despite of g being linear in $\mathbf{p}$ (similar to Scenarios 3 and 4).


```
1  /// dependence of right-hand side on y yields nonlinear (in p) model
2  template<typename T>
3  VT<T> g(VT<T> p, T x, VT<T> y) {
4    // conditions
5    assert(p.size()==np); assert(y.size()==n); assert(np==2*n);
6    // result
7    VT<T> dydx(n);
8    // evaluation of result
9    int i=0;
10   dydx(i)=p(i)*x/(y(i)+1)+p(i+1)*x/(y(i+1)+1)+p(n+i);
11   i=i+1;
12   while (i<n-1) {
13     dydx(i)=p(i-1)*x/(y(i-1)+1)+p(i)*x/(y(i)+1)+p(i+1)*x/(y(i+1)+1)+p(n+i);
14     i=i+1;
15   }
16   dydx(i)=p(i-1)*x/(y(i-1)+1)+p(i)*x/(y(i)+1)+p(n+i);
17   return dydx;
18 }
```

Wanted:

$$\mathbf{y} = f(\mathbf{p}, x) \in \mathbb{R}^n$$

for fixed $\mathbf{p} \in \mathbb{R}^{n_p}$ and given $x \in \mathbb{R}$.

The unknown function $f(\mathbf{p}, x)$ is expected to be

differentiable

and, hence,

continuous

over the domain of interest.

Let x be fixed. The multivariate vector function $f : \mathbb{R}^{n_p} \to \mathbb{R}^n$, $\mathbf{y} = f(\mathbf{p})$ is continuous at a point $\mathbf{p} \in \mathbb{R}^{n_p}$ if

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

All univariate scalar (sub-)functions

$$y_j = f_j(p_i)$$

become continuous for $j = 0, \ldots, n-1$, $i = 0, \ldots, n_p - 1$ and fixed p_k for $k \neq i$.

The multivariate vector function $f : \mathbb{R}^{n_p} \to \mathbb{R}^n$, $\mathbf{y} = f(\mathbf{p})$ is differentiable at point $\mathbf{p} \in \mathbb{R}^{n_p}$ if there is a matrix $\frac{df}{d\mathbf{p}} \in \mathbb{R}^{n \times n_p}$ such that

$$f(\mathbf{p} + \Delta\mathbf{p}) = f(\mathbf{p}) + \frac{df}{d\mathbf{p}} \cdot \Delta\mathbf{p} + \mathbf{r}$$

with asymptotically vanishing remainder $\mathbf{r} = \mathbf{r}(\mathbf{p}, \Delta\mathbf{p}) \in \mathbb{R}^n$, such that

$$\lim_{\Delta\mathbf{p} \to 0} \frac{\|\mathbf{r}\|}{\|\Delta\mathbf{p}\|} = 0 \ ,$$

where $\|.\|$ denotes some vector norm, e.g., $\|.\| = \|.\|_2^2$ (squared Euclidean norm).

The matrix $\frac{df}{d\mathbf{p}}$ is the Jacobian of f.

The Jacobian of a multivariate scalar function ($n = 1$) is a row vector in $\mathbb{R}^{1 \times n_p}$. Its transpose is the gradient $f' \in \mathbb{R}^{n_p}$ of f.

The first derivative of the gradient wrt. $\mathbf{p}$ is the Hessian $\frac{d^2 f}{d\mathbf{p}^2} \in \mathbb{R}^{n_p \times n_p}$ (the second derivative) of f.

If the derivative of f' is continuous, then f is twice continuously differentiable and its Hessian is symmetric, i.e.,

$$\frac{d^2 f}{d\mathbf{p}^2}^T = \frac{d^2 f}{d\mathbf{p}^2} \ .$$

The Jacobian $f' \in \mathbb{R}^{n \times n_p}$ is a linear operator mapping an argument vector onto a result vector. Two linear functions are induced. We distiguish tangent

$$f' \cdot \mathbf{p}^{(1)} = \mathbf{y}^{(1)} \in \mathbb{R}^n$$

and adjoint

$$\mathbf{y}_{(1)} \cdot f' = \mathbf{p}_{(1)} \in \mathbb{R}^{1 \times n_p} .$$

As the Jacobian of the gradient of f, the Hessian f'' induces similar mappings.

More generally, every matrix implies corresponding tangent and adjoint functions.

In the scalar case, tangent and adjoint become one and the same due to commutativity of scalar multiplication. Validity of this statement in the context of AD is limited due to data flow reversal in adjoint mode.

Taylor expansions for the vector case follow naturally.

If $f : \mathbb{R}^{n_p} \to \mathbb{R}$, then

$$f(\mathbf{p} + \Delta\mathbf{p}) = f(\mathbf{p}) + \frac{df}{d\mathbf{p}} \cdot \Delta\mathbf{p} + \frac{1}{2} \cdot \Delta\mathbf{p}^T \cdot \frac{d^2 f}{d\mathbf{p}^2} \cdot \Delta\mathbf{p} + \mathcal{O}(\|\Delta\mathbf{p}\|_2^3)$$

The Jacobian $\frac{df}{d\mathbf{p}} \in \mathbb{R}^{n_p}$ maps $\Delta\mathbf{p} \in \mathbb{R}^{n_p}$ to a scalar as the result of a scalar (also: inner) vector product. Similarly, $\frac{df}{dp} \in \mathbb{R}$ and $\Delta p \in \mathbb{R}$ yielded $\frac{df}{dp} \cdot \Delta p \in R$ in the scalar case.

The Hessian $\frac{d^2 f}{d\mathbf{p}^2} \in \mathbb{R}^{n_p \times n_p}$ maps $\Delta\mathbf{p}^T \in \mathbb{R}^{1 \times n_p}$ to $\Delta\mathbf{p}^T \cdot \frac{d^2 f}{d\mathbf{p}^2} \in \mathbb{R}^{1 \times n_p}$ (adjoint) followed by the inner vector product with $\Delta\mathbf{p} \in \mathbb{R}^{n_p}$ (tangent yielding tangent of adjoint). Similarly, $\Delta p \cdot \frac{d^2 f}{dp^2} \cdot \Delta p = \frac{d^2 f}{dp^2} \cdot \Delta p^2$ in the scalar case.

Associativity of matrix multiplication yields an analogous formulation as adjoint of tangent.

The Hessian of the multivariate vector function $f : \mathbb{R}^{n_p} \to \mathbb{R}^n$ is a three-tensor

$$\frac{d^3 f}{d\mathbf{p}^3} \in \mathbb{R}^{n \times n_p \times n_p} .$$

While the fundamental logic of the argument remains unaltered, the more involved notation yields additional complication ("index war").

Hence, we state the first-order Taylor expansion only.

$$f(\mathbf{p} + \Delta\mathbf{p}) = f(\mathbf{p}) + \frac{df}{d\mathbf{p}} \cdot \Delta\mathbf{p} + \mathcal{O}(\|\Delta\mathbf{p}\|_2^2)$$

The Jacobian $\frac{df}{d\mathbf{p}} \in \mathbb{R}^{n \times n_p}$ maps $\Delta\mathbf{p} \in \mathbb{R}^{n_p}$ onto a vector in $\mathbb{R}^n$ in consistency with both $f(\mathbf{p} + \Delta\mathbf{p}) \in \mathbb{R}^n$ and $f(\mathbf{p}) \in \mathbb{R}^n$.

Outline

- The explicit Euler method is applicable to both scenarios.
- Local linearization (first-order truncated Taylor expansion with step size Δx) yields a numerical approximation of $\frac{d\mathbf{y}}{dx}$ by a forward finite difference.
- A sequence of systems of explicit algebraic equations is evaluated to integrate the differential model from $x = 0$ to $x = 1$.
- An error of order $\geq \mathcal{O}(\Delta x^2)$ is induced.

The explicit Euler method replaces the derivative in

$$\frac{d\mathbf{y}}{dx} = g(\mathbf{p}, x, \mathbf{y}(\mathbf{p}, x))$$

with a forward finite difference in direction $0 < \Delta x \ll 1$ yielding

$$\frac{\mathbf{y}(\mathbf{p}, x + \Delta x) - \mathbf{y}(\mathbf{p}, x)}{\Delta x} = g(\mathbf{p}, x, \mathbf{y}(\mathbf{p}, x))$$

and, hence, the iterative approximation of the solution as

$$\mathbf{y}(\mathbf{p}, x + \Delta x) = \mathbf{y}(\mathbf{p}, x) + \Delta x \cdot g(\mathbf{p}, x, \mathbf{y}(\mathbf{p}, x))$$

for given $\mathbf{y}(\mathbf{p}, 0) = \mathbf{y}^0(\mathbf{p})$. In the given Scenarios 5 and 6, $\mathbf{y}^0 = 0$ is independent of $\mathbf{p}$.

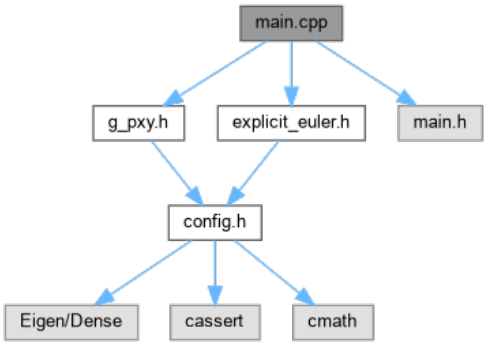
The explicit Euler method evaluates the following sequence of explicit algebraic equations

$$\mathbf{y}^{i+1} = \mathbf{y}^i + \frac{x^m}{m} \cdot g(\mathbf{p}, x^i, \mathbf{y}^i) \,, \quad i = 0, \ldots, m-1$$

for given initial value $\mathbf{y}^0$ at x^0 and number of integration steps $m > 0$.

Note the obvious similarity with scalar case.


```
1  int m=100; // number of integration steps
2
3  /// explicit Euler simulation of differential model
4  template<typename T>
5  VT<T> f(VT<T> p, T x) {
6    // condition
7    assert(x>0);
8    // integration step size
9    T delta_x=x/m;
10   // initial position fixed to zero
11   x=0;
12   // initial state fixed to zero
13   VT<T> y=VT<T>::Zero(n);
14   int i=0;
15   while (i<m) {
16     // explicit Euler step
17     y=y+delta_x*g(p,x,y);
18     x=x+delta_x;
19     i=i+1;
20   }
21   return y;
22 }
```



→ nD/simulation/differential/explicit_euler/

For $f : \mathbb{R}^{n_p} \to \mathbb{R}^n$, truncation of the Taylor expansion at $\mathbf{p} = (p_i) \in \mathbb{R}^{n_p}$ in direction

$$\Delta\mathbf{p} = (\Delta p_i) \in \mathbb{R}^{n_p} : 0 < \Delta p_i \ll 1,\ i = 0, \ldots, n_p - 1$$

after the first-order term yields

$$f(\mathbf{p} + \Delta\mathbf{p}) = f(\mathbf{p}) + f' \cdot \Delta\mathbf{p} + \mathcal{O}(\|\Delta\mathbf{p}\|_2^2) \qquad \text{(linearization)}$$

and, hence, a second-order accurate approximation of the Jacobian-vector product $f' \cdot \Delta\mathbf{p}$ at $\mathbf{p}$ as

$$f' \cdot \Delta\mathbf{p} = f(\mathbf{p} + \Delta\mathbf{p}) - f(\mathbf{p}) + \mathcal{O}(\|\Delta\mathbf{p}\|_2^2)\ .$$

Analogously, truncation of the Taylor expansion at $\mathbf{p} = (p_i) \in \mathbb{R}^{n_p}$ in direction

$$-\Delta\mathbf{p} = (-\Delta p_i) \in \mathbb{R}^{n_p} : 0 < \Delta p_i \ll 1,\ i = 0, \ldots, n_p - 1$$

after the first-order term yields

$$f(\mathbf{p} - \Delta\mathbf{p}) = f(\mathbf{p}) - f' \cdot \Delta\mathbf{p} + \mathcal{O}(\|\Delta\mathbf{p}\|_2^2)$$

and, hence, a second-order accurate approximation of the Jacobian-vector product $f' \cdot \Delta\mathbf{p}$ at $\mathbf{p}$ as

$$f' \cdot \Delta\mathbf{p} = f(\mathbf{p}) - f(\mathbf{p} - \Delta\mathbf{p}) - \mathcal{O}(\|\Delta\mathbf{p}\|_2^2)\ .$$

Setting $\Delta\mathbf{p} = \Delta p_i \cdot \mathbf{e}_i$ for $i = 0, \ldots, n_p - 1$ yields a first-order accurate approximation of the i-th column of the Jacobian at $\mathbf{p}$ as

$$f' \cdot \mathbf{e}_i = \frac{f(\mathbf{p} + \Delta p_i \cdot \mathbf{e}_i) - f(\mathbf{p})}{\Delta p_i} + \mathcal{O}(\Delta p_i)$$

and, hence, a method for computing a forward finite difference approximation of the whole Jacobian.

Note that for $n_p = n = 1$, there is a single standard basis "vector" $\mathbf{e}_0 \in \mathbb{R} : \mathbf{e}_0 = 1$ yielding the scalar forward finite difference

$$f' = \frac{f(p + \Delta p) - f(p)}{\Delta p} + \mathcal{O}(\Delta p)\ ,$$

where $p \equiv p_0$.

Setting $\Delta\mathbf{p} = \Delta p_i \cdot \mathbf{e}_i$ for $i = 0, \ldots, n_p - 1$ yields a first-order accurate approximation of the i-th column of the Jacobian at $\mathbf{p}$ as

$$f' \cdot \mathbf{e}_i = \frac{f(\mathbf{p}) - f(\mathbf{p} - \Delta p_i \cdot \mathbf{e}_i)}{\Delta p_i} - \mathcal{O}(\Delta p_i)$$

and, hence, a method for computing a backward finite difference approximation of the whole Jacobian.

The case $n_p = n = 1$ yields the scalar backward finite difference

$$f' = \frac{f(p) - f(p - \Delta p)}{\Delta p} - \mathcal{O}(\Delta p) \,,$$

where $p \equiv p_0$.

Similar to the scalar case, addition of

$$f' \cdot \mathbf{e}_i = \frac{f(\mathbf{p} + \Delta p_i \cdot \mathbf{e}_i) - f(\mathbf{p})}{\Delta p_i} + \mathcal{O}(\Delta p_i)$$

and

$$f' \cdot \mathbf{e}_i = \frac{f(\mathbf{p}) - f(\mathbf{p} - \Delta p_i \cdot \mathbf{e}_i)}{\Delta p_i} - \mathcal{O}(\Delta p_i)$$

yields a second-order accurate approximation of the i-th column of the Jacobian at $\mathbf{p}$ as

$$f' \cdot \mathbf{e}_i = \frac{f(\mathbf{p} + \Delta p_i \cdot \mathbf{e}_i) - f(\mathbf{p} - \Delta p_i \cdot \mathbf{e}_i)}{2 \cdot \Delta p_i} + \mathcal{O}(\Delta p_i^2)$$

and, hence, a method for computing a central finite difference approximation of the whole Jacobian.

The scalar central finite difference $f' = \frac{f(p+\Delta p) - f(p-\Delta p)}{2 \cdot \Delta p} - \mathcal{O}(\Delta p)$ follows immediately.

- The implicit Euler method is applicable to both scenarios.
- Local linearization (first-order truncated Taylor expansion with step size $-\Delta x$) yields a numerical approximation of $\frac{d\mathbf{y}}{dx}$ by a backward finite difference.
- A sequence of systems of implicit algebraic equations is solved to integrate the differential model from $x = 0$ to $x = 1$.
- The corresponding roots are computed by the Newton method.
- An error of order $\geq \mathcal{O}(\Delta x^2)$ is induced.
- The implicit Euler method can remain stable (with bounded error) for larger step sizes than the explicit Euler method.

The implicit Euler method replaces the derivative in the ODE with a backward finite difference yielding

$$\frac{\mathbf{y}(\mathbf{p}, x) - \mathbf{y}(\mathbf{p}, x - \Delta x)}{\Delta x} = g(\mathbf{p}, x, \mathbf{y}(\mathbf{p}, x))$$

and, hence, $\mathbf{y}(\mathbf{p}, x)$ as the solution of the algebraic equation

$$\mathbf{y}(\mathbf{p}, x) - \mathbf{y}(\mathbf{p}, x - \Delta x) - \Delta x \cdot g(\mathbf{p}, x, \mathbf{y}(\mathbf{p}, x)) = 0 \,.$$

The solution is approximated iteratively for given $\mathbf{y}(\mathbf{p}, 0) = \mathbf{y}^0(\mathbf{p})$ and $\Delta x > 0$. In the given Scenarios 5 and 6, $\mathbf{y}^0 = 0$ is independent of $\mathbf{p}$.

The error of order Δx^2 of the implicit Euler method can remain bounded for larger steps sizes than the explicit Euler method. This improved stability comes at a higher computational cost due to multivariate root finding.

Implicit Euler Method

Algorithm

The implicit Euler method iteratively solves the following system of nonlinear equations (also: implicit Euler equation):

$$r(\mathbf{y}^i) \equiv \mathbf{y}^i - \mathbf{y}^{i-1} - \frac{x^m}{m} \cdot g(\mathbf{p}, x^i, \mathbf{y}^i) = 0 \ , \quad i = 0, \ldots, m-1 \ .$$

For given initial value $\mathbf{y}^0(\mathbf{p})$ at $x^0 = 0$ and number of integration steps $m > 0$ the method approximates a root of the residual defined by the left-hand side.

The residual is implemented as follows:

```
1  /// residual of implicit Euler equation
2  template<typename T>
3  VT<T> r(VT<T> p, T x, VT<T> y, VT<T> y_prev, T delta_x) {
4    // conditions
5    assert(x>=0); assert(delta_x>0); assert(p.size()==n_p);
6    assert(y.size()==n); assert(y_prev.size()==n);
7    // residual
8    return y-y_prev-delta_x*g(p,x,y);
9  }
```


Implicit Euler Method

Implementation

```
1  /// implicit Euler simulation of differential model
2  template<typename T>
3  VT<T> f(VT<T> p, T x) {
4    // conditions
5    assert(x>=0); assert(p.size()==np);
6    // integration step size
7    T delta_x=x/m;
8    // initial position fixed to zero
9    x=0;
10   // initial state fixed to zero
11   VT<T> y=VT<T>::Zero(n);
12   int i=0;
13   while (i<m) {
14     // implicit Euler step
15     y=newton(p,x,y,delta_x); // root of implicit Euler equation
16     x=x+delta_x;
17     i=i+1;
18   }
19   return y;
20 }
```


Root Finding

Linearization

Many numerical methods for nonlinear problems, such as

$$r(\mathbf{y}) = 0 \ ,$$

rely on local replacement of the target function with an affine (in $\Delta\mathbf{y}$) approximation derived from the truncated Taylor expansion and "hoping" that

$$r(\mathbf{y} + \Delta\mathbf{y}) \approx r(\mathbf{y}) + \frac{dr}{d\mathbf{y}} \cdot \Delta\mathbf{y} \ ,$$

with a reasonably small error. Note that $\frac{dr}{d\mathbf{y}} = \frac{dr}{d\mathbf{y}}(\mathbf{y}) \in \mathbb{R}^{n \times n}$ for Scenario 6, where $g = g(\mathbf{y})$.

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

Root Finding

Newton Method

Building on the assumption that $r(\mathbf{y} + \Delta\mathbf{y}) \approx r(\mathbf{y}) + \frac{dr}{d\mathbf{y}} \cdot \Delta\mathbf{y}$, the root finding problem for r can be replaced locally (at the current $\mathbf{y}$) by the root finding problem for its linearization

$$\bar{r}(\Delta\mathbf{y}) = r(\mathbf{y}) + \frac{dr}{d\mathbf{y}} \cdot \Delta\mathbf{y} \ .$$

Solution of the system of linear equations

$$r(\mathbf{y}) + \frac{dr}{d\mathbf{y}} \cdot \Delta\mathbf{y} = 0$$

for $\Delta\mathbf{y}$ yields

$$\Delta\mathbf{y} = -\frac{dr}{d\mathbf{y}}^{-1} \cdot r(\mathbf{y}) \ ,$$

implying $r(\mathbf{y} + \Delta\mathbf{y}) \approx 0$. The Jacobian $\frac{dr}{d\mathbf{y}}$ is required to be invertible. It can be computed by tangent AD.

If the new iterate is not close enough to the root, i.e., $|r(\mathbf{y} + \Delta\mathbf{y})| > \epsilon$ for some measure of accuracy of the numerical approximation $\epsilon > 0$, then it becomes the starting point for the next iteration yielding the recurrence

$$\mathbf{y}^{i+1} = \mathbf{y}^i - \underbrace{\frac{dr}{d\mathbf{y}}(\mathbf{y}^i)^{-1} \cdot r(\mathbf{y}^i)}_{=\Delta\mathbf{y}^i} \quad \text{for } i = 0, \ldots .$$

In the sample code, the Newton step $\Delta\mathbf{y}^i$ is computed as the solution of the system of linear equations (also: Newton system)

$$\frac{dr}{d\mathbf{y}}(\mathbf{y}^i) \cdot \Delta\mathbf{y}^i = r(\mathbf{y}^i)$$

with a direct linear solver from the Eigen library (e.g., *LU*-factorization).

Convergence of the Newton method is not guaranteed in general.

Local contractiveness in all iterations is a sufficient (not necessary) condition for the convergence of the Newton method, which can be regarded as a fixed point iteration

$$\mathbf{y} := \check{f}(\mathbf{y}) = \mathbf{y} - \frac{dr}{d\mathbf{y}}^{-1} .$$

An argument similar to the scalar case yields linear growth of the convergence rate with decreasing values of $\|\frac{\check{f}}{d\mathbf{y}}\|$.

For vanishing first derivative of $\check{f}$ wrt. $\mathbf{y}$ we get at least quadratic convergence; cubic for vanshing first and second derivatives and so forth.

For linear problems (constant $\frac{dr}{d\mathbf{y}} \Rightarrow$ vanishing $\frac{d^2r}{d\mathbf{y}^2}$), the Newton method requires a single iteration to converge.


```
1  /// Newton method for computing root of implicit Euler system
2  template<typename T>
3  VT<T> newton(VT<T> p, T x, VT<T> y, T delta_x) {
4    // conditions
5    assert(x>=0);
6    assert(delta_x>0);
7    assert(p.size()==np);
8    assert(y.size()==n);
9    // state computed by previous implicit Euler step
10   VT<T> y_prev=y;
11   // residual of implicit Euler system
12   VT<T> r_=r(p,x,y,y_prev,delta_x);
13   do {
14     // Jacobian of residual by tangent AD
15     MT<T> drdy=dr_dy(p,x,y,y_prev,delta_x);
16     // Newton step solves system of linear equations
17     y=y+drdy.lu().solve(-r_);
18     r_=r(p,x,y,y_prev,delta_x);
19   } while (r_.norm()>eps); // convergence criterion
20   return y;
21 }
```


The Newton system

$$\frac{dr}{d\mathbf{y}}(\mathbf{y}^i) \cdot \Delta\mathbf{y}^i = r(\mathbf{y}^i)$$

needs to be solved in each iteration of the Newton method.

Both direct and *indirect (iterative)* linear solvers can be used. The latter yield so-called *inexact Newton methods*. Our sample software uses direct solvers provided by the Eigen library that are based on *LU*- (also known as *LR*-) LL^T-, LDL^T-, and *QR*-factorizations of the regular system matrices.

Pivoting is often employed to control numerical conditioning. The following brief introduction to direct solvers excludes pivoting.

Diagonal system matrices yield linear systems the solution of which amounts to the solution of mutually independent linear equations.

All diagonal elements need to be nonzero to ensure regularity.

Inversion becomes particularly simple due to

$$\begin{pmatrix} d_{0,0} & & & \\ & \ddots & & \\ & & d_{i,i} & \\ & & & \ddots \\ & & & & d_{n-1,n-1} \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{d_{0,0}} & & & \\ & \ddots & & \\ & & \frac{1}{d_{i,i}} & \\ & & & \ddots \\ & & & & \frac{1}{d_{n-1,n-1}} \end{pmatrix} .$$

Lower / upper triangular system matrices yield linear systems the solution of which amounts to simple substitution.

1. Lower triangular system by forward substitution, e.g.

$$\begin{pmatrix} 1 & 0 \\ -\frac{1}{3} & 1 \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \Rightarrow \quad \mathbf{y} = \begin{pmatrix} 1 \\ \frac{4}{3} \end{pmatrix} .$$

2. Upper triangular system by backward substitution, e.g.

$$\begin{pmatrix} 3 & 1 \\ 0 & \frac{7}{3} \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{4}{3} \end{pmatrix} \quad \Rightarrow \quad \mathbf{y} = \begin{pmatrix} \frac{1}{7} \\ \frac{4}{7} \end{pmatrix} .$$

For orthogonal $A \in \mathbb{R}^{n \times n}$ the solution of $A \cdot \mathbf{x} = \mathbf{b}$ simplifies as

$$\mathbf{x} = A^{-1} \cdot \mathbf{b} = A^T \cdot \mathbf{b} ,$$

e.g.,

$$\begin{pmatrix} \frac{4}{5} & \frac{3}{5} \\ \frac{3}{5} & -\frac{4}{5} \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \end{pmatrix} \quad \Rightarrow \quad \mathbf{y} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}$$

Householder matrices fall into this category. They form the basis of QR-factorization, yielding orthogonal $Q \in \mathbb{R}^{n \times n}$ and upper triangular $R \in \mathbb{R}^{n \times n}$

Direct methods for the solution of linear systems aim to represent A as a product of diagonal, triangular and/or orthogonal matrices, e.g,

$A = L \cdot R$ with lower triangular $L \in \mathbb{R}^{n \times n}$ and upper triangular $R \in \mathbb{R}^{n \times n}$

$$\Rightarrow \quad L \cdot \mathbf{v} = \mathbf{b} ; \quad R \cdot \mathbf{x} = \mathbf{v}$$

$A = L \cdot L^T$ with lower triangular $L \in \mathbb{R}^{n \times n}$

$$\Rightarrow \quad L \cdot \mathbf{v} = \mathbf{b} ; \quad L^T \cdot \mathbf{x} = \mathbf{v}$$

$A = L \cdot D \cdot L^T$ with diagonal $D \in \mathbb{R}^{n \times n}$ and lower triangular $L \in \mathbb{R}^{n \times n}$

$$\Rightarrow \quad L \cdot \mathbf{v} = \mathbf{b} ; \quad D \cdot L^T \cdot \mathbf{x} = \mathbf{v}$$

$A = Q \cdot R$ with orthogonal $Q \in \mathbb{R}^{n \times n}$ and upper triangular $R \in \mathbb{R}^{n \times n}$

$$\Rightarrow \quad \mathbf{v} = Q^T \cdot \mathbf{b} ; \quad R \cdot \mathbf{x} = \mathbf{v}$$

From

$$\begin{pmatrix} \alpha & \mathbf{b}^T \\ \mathbf{a} & A_* \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \mathbf{a}_1 & L_* \end{pmatrix} \cdot \begin{pmatrix} \alpha & \mathbf{b}^T \\ 0 & R_* \end{pmatrix}$$

with $\alpha \in \mathbb{R}$, $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{n-1}$ and lower / upper triangular matrices $L_*, R_* \in \mathbb{R}^{(n-1)\times(n-1)}$ it follows that

$$\begin{aligned} \mathbf{a} &= \alpha \cdot \mathbf{a}_1 \\ A_* &= \mathbf{a}_1 \cdot \mathbf{b}^T + L_* \cdot R_* \end{aligned}$$

and hence

$$\begin{aligned} \mathbf{a}_1 &= \frac{\mathbf{a}}{\alpha} \\ L_* \cdot R_* &= A_* - \mathbf{a}_1 \cdot \mathbf{b}^T \ . \end{aligned}$$

From

$$\begin{pmatrix} \alpha & \mathbf{a}^T \\ \mathbf{a} & A_* \end{pmatrix} = \begin{pmatrix} \rho & 0 \\ \mathbf{b} & L_* \end{pmatrix} \begin{pmatrix} \rho & \mathbf{b}^T \\ 0 & L_*^T \end{pmatrix}$$

it follows that

$$\alpha = \rho^2; \quad \mathbf{a}^T = \rho \cdot \mathbf{b}^T; \quad A_* = \mathbf{b} \cdot \mathbf{b}^T + L_* \cdot L_*^T$$

and hence

$$\begin{aligned} \rho &= \sqrt{\alpha} \\ \mathbf{b} &= \frac{\mathbf{a}}{\rho} \\ L_* \cdot L_*^T &= A_* - \mathbf{b} \cdot \mathbf{b}^T \ . \end{aligned}$$

A is required to be symmetric positive definite.

$A = L \cdot D \cdot L^T$

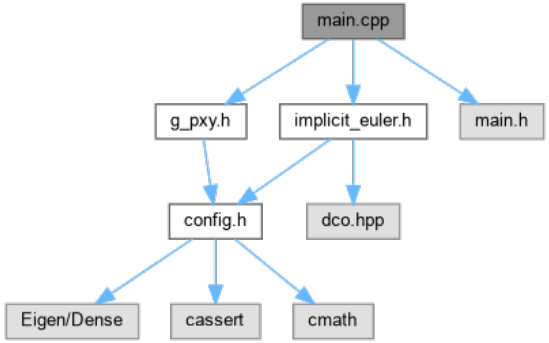
A LDL^T-factorization eliminates the need for computing square roots in LL^T-factorization. It can thus be applied to indefinite systems. Note that

$$A = L \cdot D \cdot L^T = L \cdot \sqrt{D} \cdot \sqrt{D} \cdot L^T = L \cdot \sqrt{D} \cdot \sqrt{D}^T \cdot L^T = L \cdot \sqrt{D} \cdot (L \cdot \sqrt{D})^T \ ,$$

where $\sqrt{D}$ denotes the diagonal matrix with entries equal to the square roots of the corresponding diagonal entries of D.

$A = Q \cdot R$

A QR-factorization of the system matrix can be computed, e.g., by the Householder method. The solution of systems of linear equations (square system matrices) represents a special case of the linear regression problems to discussed in more detail further below.



→ nD/simulation/differential/implicit_euler/

Outline

Optimization
Executive Summary

- The parameter estimation (also: calibration) problem can be considered as an unconstrained nonlinear optimization problem.
- General-purpose nonlinear optimization methods such as the gradient descent and Newton methods can be employed for its solutions. Conditions apply.
- Linear (in the parameter to be estimated) models (Scenario 5) enable the use of more efficient linear regression algorithms such as the normal equation and Householder methods.
- Linearization and linear regression can be combined for the calibration of nonlinear models (Scenario 6). Conditions apply.

Optimization
Models

We consider the models given by numerical solutions of the two initial value problems described by Scenarios 5 and 6, that is,

Scenario 5:

$$\frac{d\mathbf{y}}{dx} = \frac{\mathbf{p}_1}{x+1} + \mathbf{p}_2;\ \mathbf{y}(\mathbf{p},0) = 0\ ,$$

for $n_p = 2 \cdot n$ and $\mathbf{p} = \binom{\mathbf{p}_1}{\mathbf{p}_2}$ with $\mathbf{p}_i \in \mathbb{R}^n$, $i = 1, 2$.

Scenario 6:

$$\frac{d\mathbf{y}}{dx} = g(\mathbf{p}, x, \mathbf{y});\ \mathbf{y}(\mathbf{p},0) = 0\ , \quad \text{where } n_p = 2 \cdot n \text{ and}$$

$$\left[\frac{d\mathbf{y}}{dx}\right]_i = \begin{cases} \sum_{j=i}^{i+1} \frac{p_j \cdot x}{y_j+1} + p_{n+i} & i = 0 \\ \sum_{j=i-1}^{i} \frac{p_j \cdot x}{y_j+1} + p_{n+i} & i = n-1 \\ \sum_{j=i-1}^{i+1} \frac{p_j \cdot x}{y_j+1} + p_{n+i} & \text{otherwise}\ . \end{cases}$$

Optimization
Data

We consider randomly generated data

$$(\mathbf{x}, Y) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}} \times n}$$
$$\mathbf{x} = (x_i)_{i=0,\ldots,n_{\text{obs}}-1}$$
$$Y = (\mathbf{y}_i)_{0,\ldots,n_{\text{obs}}-1} = (y_{i,j})_{i=0,\ldots,n_{\text{obs}}-1,\ j=0,\ldots,n-1}$$

e.g.,

```
1  int n=3; // e.g. 3-dimensional state space
2  int nobs=5; // e.g. 5 observations
3
4  template<typename T>
5  using VT=Eigen::VectorX<T>;
6
7  template<typename T>
8  using MT=Eigen::MatrixX<T>;
9
10 VT<double> xobs=VT<double>::Random(nobs); // random entries in [-1,1]
11 xobs=xobs.cwiseProduct(xobs); // random entries in [0,1]
12 MT<T> yobs=MT<T>::Random(nobs,n); // random entries in [-1,1]
```

The residual vector

$$\mathbf{r} = \mathbf{r}(\mathbf{p}, \mathbf{x}, Y) \equiv \Big([f(\mathbf{p}, x_i)]_j - y_{i,j} \Big)_{i=0,\dots,n_{\text{obs}}-1, j=0,\dots,n-1} \in \mathbb{R}^{n_{\text{obs}} \cdot n}$$

yields a scalar least squares error as

$$e = e(\mathbf{p}, \mathbf{x}, Y) \equiv \|\mathbf{r}\|_2^2 = \mathbf{r}^T \cdot \mathbf{r} = \sum_{i=0}^{n_{\text{obs}} \cdot n - 1} r_i^2 = \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} \Big([f(p, x_i)]_j - y_{i,j} \Big)^2 .$$

We aim to minimize the least squares error of the model predictions wrt. the given data, that is, $\min_{\mathbf{p}} e$ where $e = e(\mathbf{p}, \mathbf{x}, Y)$ is implemented as follows:

```
1  /// least squares error of model predictions relative to observations
2  template<typename T>
3  T e(VT<T> p, VT<T> xobs, MT<T> yobs) {
4    // conditions
5    assert(p.size()==np);
6    assert(xobs.size()==nobs);
7    assert(xobs.size()==yobs.size());
8    // least squares error
9    T e=0;
10   int i=0;
11   while (i<xobs.size()) {
12     e=e+(f(p,xobs(i))-yobs.row(i).transpose()).squaredNorm();
13     i=i+1;
14   }
15   return e;
16 }
```


The objective

$$e = \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} (f(p, x_i)_j - y_{i,j})^2$$

features a (unique[18]) stationary point at $\mathbf{p} \in \mathbb{R}^{n_p}$, where

$$e' \equiv \left(\frac{de}{d\mathbf{p}} \right)^T = 0$$

This stationary point is a local minimum if the Hessian $e'' \equiv \frac{d^2 e}{d\mathbf{p}^2} \in \mathbb{R}^{n_p \times n_p}$ is symmetric positive definite, that is, $\forall 0 \neq \mathbf{v} \in \mathbb{R}^{n_p}$:

$$\mathbf{v}^T \cdot e'' \cdot \mathbf{v} > 0 .$$

[18] ... if e is *strictly convex*, i.e. its Hessian is globally symmetric postive definite

- "Going downhill" appears to be a reasonable heuristic for minimization.
- The negative gradient points downhill. It represents the direction of steepest descent.
- The main question is about "how far" to follow it in order to ensure a decrease in the objective. Line search is employed.
- The accuracy of the first derivative can be crucial for the performance of gradient descent. Exact differentiation methods are required instead of numerical approximation by finite differences.
- AD remains the preferred method.

The negative gradient points into the direction of steepest descent.

Starting from some initial estimate for the stationary point $\mathbf{p} \in \mathbb{R}^{n_p}$, the gradient descent method iteratively takes steps in direction of the negative gradient,

$$\mathbf{p} := \mathbf{p} - e' \quad \text{while } \|e'\| > \epsilon \ .$$

Less progress is made as $\|e'\| \to 0$.

No further local decrease in the function value can be achieved at a stationary point, where $\|e'\| = 0$.

Validation of a local minimum requires for the Hessian $e'' \in \mathbb{R}^{n_p \times n_p}$ to be positive definite.

The gradient descent method amounts to a fixpoint iteration as

$$\mathbf{p} = \check{e}(\mathbf{p}) = \mathbf{p} - e' \ .$$

The above converges locally, that is, there is a neighborhood of the current iterate $\mathbf{p}$ such that a gradient descent step yields a decrease in $\|e'(\mathbf{p})\|$, if $\check{e}$ is locally contractive, that is, if

$$\left\| \frac{d\check{e}}{d\mathbf{p}} \right\| = \|I_n - e''\| < 1 \ .$$

It remains to be decided how large the step into the current descent direction should be. The step size is typically damped in order to ensure continued progress toward $\min_{\mathbf{p} \in \mathbb{R}^{n_p}} e(\mathbf{p}, \mathbf{x}, Y)$ yielding the recurrence

$$\mathbf{p} := \mathbf{p} - \alpha \cdot e' \quad \text{while } \|e'\| > \epsilon \ .$$

The damping parameter α is often determined by the line search, e.g., by iterative bisection starting from $\alpha = 1$. More rigorously, optimal α can be computed that maximize the decrease, which is rarely done due to higher computational cost.

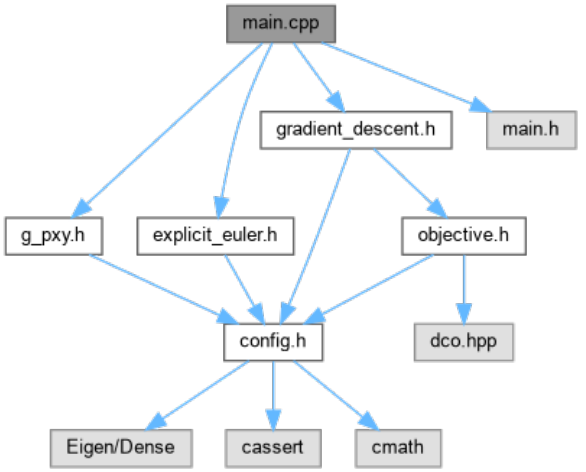
The gradient descent method with line search is globally convergent (in infinite precision arithmetic). Convergence is linear (slow).


```
1  /// stationary point of error wrt. model parameteris by the gradient descent method
2  template<typename T>
3  VT<T> fit(VT<T> p, VT<T> xobs, MT<T> yobs) {
4    // conditions
5    assert(p.size()==np); assert(xobs.size()==nobs);
6    assert(yobs.rows()==nobs); assert(yobs.cols()==n);
7    // derivative (gradient) of error wrt. model parameters
8    VT<T> dedp=de_dp(p,xobs,yobs);
9    int i=0;
10   do {
11     // simple line search
12     T alpha=1; T e_=e(p,xobs,yobs);
13     VT<T> p_trial=p-alpha*dedp;
14     while ((e(p_trial,xobs,yobs)>=e_)&&(alpha>eps)) { // simple line search
15       alpha=alpha/2; p_trial=p-alpha*dedp;
16     }
17     // result of last gradient descent step
18     p=p_trial; dedp=de_dp(p,xobs,yobs);
19     i=i+1;
20   } while (dedp.norm()>eps&&i<pow(10,6)); // (guaranteed) convergence criterion
21   return p;
22 }
```

- e' should be computed by adjoint AD applied to e.
- e'' should be computed by tangent AD applied to e'.
- If e'' is symmetric positive definite (spd), then Cholesky factorization yields a factorization into $L \cdot L^T$ with lower triangular $L \in \mathbb{R}^{n_p \times n_p}$.
- A simple spd test to ensure a local minimum can be implemented with the Eigen library as follows:

```
cout << "spd(dde_dp_dp)="
     << (dde_dp_dp(p,xobs,yobs).llt().info()!=Eigen::NumericalIssue)
     << endl;
```

  Output "1" indicates success, while "0" does not.



→ nD/optimization/gradient_descent/

As in the scalar case, data error analysis can be performed by computing

$$\frac{d\mathbf{p}}{d\mathbf{v}} \in \mathbb{R}^{n_p \times (n+1)\cdot n_{\text{obs}}} ,$$

where

$$\mathbf{v} = \begin{pmatrix} \mathbf{x} \\ (\mathbf{y}_i)_{i=0,\ldots,n_{\text{obs}}-1} \end{pmatrix} \in \mathbb{R}^{(n+1)\cdot n_{\text{obs}}} ,$$

using

- finite differences
- tangent AD
- adjoint AD
- symbolic differentiation.

Large sensitivity of $\mathbf{p}$ wrt. a data point implies (at least) the need for more precise observation techniques.

For given data

$$(\mathbf{x}, Y) \in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}} \times n} , \quad Y = (\mathbf{y}_i^T)_{i=0,\ldots,n_{\text{obs}}-1}$$

and model

$$f : \mathbb{R}^{n_p} \times \mathbb{R} \to \mathbb{R}^n : \mathbf{y} = f(\mathbf{p}, x)$$

parameter estimation computes $\mathbf{p} \in \mathbb{R}^{n_p}$ such that the least squares error of model predictions wrt. the data is minimized, that is,

$$\mathbf{p} := \phi(\mathbf{p}, \mathbf{v}) \equiv \text{argmin}_{\mathbf{p}} e = \text{argmin}_{\mathbf{p}} \left(\sum_{i=0}^{n_{\text{obs}}-1} \| f(\mathbf{p}, x_i) - \mathbf{y}_i \|_2^2 \right) .$$

Let $\mathbf{p} = \mathbf{p}(\mathbf{v}) = \operatorname{argmin}_{\mathbf{p}} e(\mathbf{p}, \mathbf{v})$ for the twice continuously differentiable objective $e : \mathbb{R}^{n_p} \times \mathbb{R}^{(n+1)\cdot n_{\text{obs}}} \to \mathbb{R}$. Differentiation of the first-order optimality condition

$$\frac{de}{d\mathbf{p}}(\mathbf{p}, \mathbf{v}) = 0$$

at the solution $\mathbf{p} \in \mathbb{R}^{n_p}$ wrt. $\mathbf{v}$ yields

$$\underbrace{\frac{d^2 e}{d\mathbf{p} d\mathbf{v}}}_{\in \mathbb{R}^{n_p \times n_{\text{obs}}}} = \underbrace{\frac{\partial \frac{de}{d\mathbf{p}}}{\partial \mathbf{v}}}_{\in \mathbb{R}^{n_p \times n_{\text{obs}}}} + \underbrace{\frac{d^2 e}{d\mathbf{p}^2}}_{\in \mathbb{R}^{n_p \times n_p}} \cdot \underbrace{\frac{d\mathbf{p}}{d\mathbf{v}}}_{\in \mathbb{R}^{n_p \times n_{\text{obs}}}} = 0$$

and, hence, $\frac{d\mathbf{p}}{d\mathbf{v}}$ as the solution of the following n_p systems of n_{obs} simultaneous linear equations:

$$\frac{d^2 e}{d\mathbf{p}^2} \cdot \frac{d\mathbf{p}}{d\mathbf{v}} = -\frac{\partial \frac{de}{d\mathbf{p}}}{\partial \mathbf{v}} = -\left(\frac{d \frac{\partial e}{\partial \mathbf{v}}}{d\mathbf{p}} \right)^T$$

Naïve AD of ϕ can be avoided.

- Linearity of the model in its parameters yields linear regression problems (e.g., Scenario 5).
- Exploitation of their special structure yields better performing numerical optimization methods, including normal equation and Householder methods for linear regression.
- Parameter estimation amounts to a quadratic minimization problem. Linear regression methods compute the unique solution very efficiently.
- The numerical stability or the normal equation method may become unsatisfactory. Hence, the numerically more stable Householder method will be also be discussed.

We consider the model given by the numerical solution of the initial value problem described by Scenario 5, that is,

$$\frac{d\mathbf{y}}{dx} = \frac{\mathbf{p}_1}{x+1} + \mathbf{p}_2; \ \mathbf{y}(\mathbf{p}, 0) = 0 \ ,$$

for $n_p = 2 \cdot n$ and $\mathbf{p} = \binom{\mathbf{p}_1}{\mathbf{p}_2}$ with $\mathbf{p}_i \in \mathbb{R}^n$, $i = 1, 2$.

The model is linear in $\mathbf{p}$ as $g(\mathbf{p}, x)$ is linear in $\mathbf{p}$ and does not depend on the state $\mathbf{y}$.

We consider randomly generated data

$$\begin{aligned}(\mathbf{x}, Y) &\in \mathbb{R}^{n_{\text{obs}}} \times \mathbb{R}^{n_{\text{obs}} \times n} \\ \mathbf{x} &= (x_i)_{i=0,\ldots,n_{\text{obs}}-1} \\ Y &= (y_{i,j})_{i=0,\ldots,n_{\text{obs}}-1,\ j=0,\ldots,n-1} \ .\end{aligned}$$

We aim to minimize the least squares error of the model predictions wrt. the given data, that is, $\min_{\mathbf{p}} e$, where

$$e = e(\mathbf{p}, \mathbf{x}, Y) \equiv \|\mathbf{r}\|_2^2 = \mathbf{r}^T \cdot \mathbf{r} = \sum_{k=0}^{n_{\text{obs}} \cdot n - 1} r_k^2 = \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} \left([f(p, x_i)]_j - y_{i,j} \right)^2$$

with residual vector

$$\mathbf{r} = \mathbf{r}(\mathbf{p}, \mathbf{x}, \mathbf{y}) \equiv \left([f(\mathbf{p}, x_i)]_j - y_{i \cdot n + j} \right)_{i=0,\ldots,n_{\text{obs}}-1, j=0,\ldots,n-1} \in \mathbb{R}^{n_{\text{obs}} \cdot n} \ ,$$

where $\mathbf{y} = (y_k) \in \mathbb{R}^{n_{\text{obs}} \cdot n}$ denotes the row-wise serialization of $Y \in \mathbb{R}^{n_{\text{obs}} \times n}$.
For linear regression,

$$\mathbf{r} = \mathbf{r}(\mathbf{p}, \mathbf{x}, \mathbf{y}) \equiv \left(\left[\frac{df}{d\mathbf{p}}(x_i) \cdot \mathbf{p} \right]_j - y_{i \cdot n + j} \right)_{i=0,\ldots,n_{\text{obs}}-1, j=0,\ldots,n-1} \in \mathbb{R}^{n_{\text{obs}} \cdot n} \ .$$

The error

$$e = \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} ([f(\mathbf{p}, x_i)]_j - y_{i,j})^2$$

features a (unique[19]) stationary point at $\mathbf{p} \in \mathbb{R}^{n_p}$, where

$$e' \equiv \left(\frac{de}{d\mathbf{p}}\right)^T = 0$$

This stationary point is a local minimum if the Hessian $e'' \equiv \frac{d^2 e}{d\mathbf{p}^2} \in \mathbb{R}^{n_p \times n_p}$ is symmetric positive definite, that is, $\forall 0 \neq \mathbf{v} \in \mathbb{R}^{n_p}$

$$\mathbf{v}^T \cdot e'' \cdot \mathbf{v} > 0 \,.$$

[19] ... if e is *strictly convex*, i.e. its Hessian is globally symmetric postive definite

From

$$\begin{aligned}
\mathbb{R}^{1\times n_p} \ni \frac{de}{d\mathbf{p}} &= \frac{d}{d\mathbf{p}} \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} \left([f(\mathbf{p}, x_i)]_j - y_{i,j}\right)^2 \\
&= \frac{d}{d\mathbf{p}} \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} \left(\left[\frac{df}{d\mathbf{p}}(x_i)\cdot\mathbf{p}\right]_j - y_{i,j}\right)^2 && \text{model linear in } \mathbf{p} \\
&= \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} \frac{d}{d\mathbf{p}} \left(\left[\frac{df}{d\mathbf{p}}(x_i)\cdot\mathbf{p}\right]_j - y_{i,j}\right)^2 && \text{element-wise differentiation} \\
&= \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} 2\cdot\left(\left[\frac{df}{d\mathbf{p}}(x_i)\cdot\mathbf{p}\right]_j - y_{i,j}\right)\cdot\frac{d}{d\mathbf{p}}\left(\left[\frac{df}{d\mathbf{p}}(x_i)\cdot\mathbf{p}\right]_j - y_{i,j}\right) \\
&= 2\cdot\sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} \left(\left[\frac{df}{d\mathbf{p}}(x_i)\cdot\mathbf{p}\right]_j - y_{i,j}\right)\cdot\left[\frac{df}{d\mathbf{p}}(x_i)\right]_j && \frac{df^2}{d\mathbf{p}^2} = 0
\end{aligned}$$

$$\begin{aligned}
&= 2\cdot\sum_{i=0}^{n_{\text{obs}}-1}\left(\mathbf{p}^T\cdot\frac{df}{d\mathbf{p}}(x_i)^T - \mathbf{y}_i^T\right)\cdot\frac{df}{d\mathbf{p}}(x_i) && \text{rows of } Y = (\mathbf{y}_i) \\
&= \mathbf{p}^T\cdot\frac{d\mathbf{r}}{d\mathbf{p}}^T\cdot\frac{d\mathbf{r}}{d\mathbf{p}} - \mathbf{y}^T\cdot\frac{d\mathbf{r}}{d\mathbf{p}} = 0 && \text{row-wise serialization } \mathbf{y} = (\mathbf{y}_i^T) \text{ of } Y\,,
\end{aligned}$$

it follows that

$$e' = \frac{de}{d\mathbf{p}}^T = \frac{d\mathbf{r}}{d\mathbf{p}}^T\cdot\frac{d\mathbf{r}}{d\mathbf{p}}\cdot\mathbf{p} - \frac{d\mathbf{r}}{d\mathbf{p}}^T\cdot\mathbf{y} = 0\,,$$

and, hence, the normal equation

$$\left(\frac{d\mathbf{r}}{d\mathbf{p}}^T\cdot\frac{d\mathbf{r}}{d\mathbf{p}}\right)\cdot\mathbf{p} = \frac{d\mathbf{r}}{d\mathbf{p}}^T\cdot\mathbf{y}$$

yielding the solution of the linear regression problem $\frac{d\mathbf{r}}{d\mathbf{p}}\cdot\mathbf{p} \approx \mathbf{y}$.

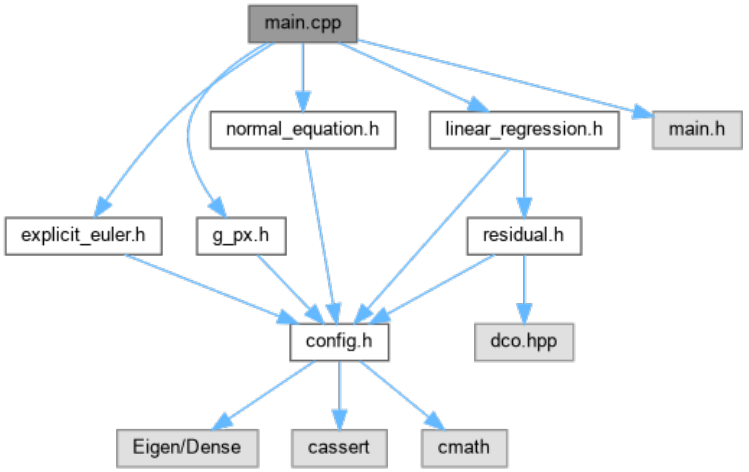
If $n_{\text{obs}} = n_p = n = 2$, then

$$\left(\frac{df}{d\mathbf{p}}(x_0)^T \quad \frac{df}{d\mathbf{p}}(x_1)^T\right)\cdot\begin{pmatrix}\frac{df}{d\mathbf{p}}(x_0)\\ \frac{df}{d\mathbf{p}}(x_1)\end{pmatrix}\cdot\mathbf{p} = \left(\frac{df}{d\mathbf{p}}(x_0)^T \quad \frac{df}{d\mathbf{p}}(x_1)^T\right)\cdot\mathbf{y} = 0 \in \mathbb{R}^{n_{\text{obs}}\cdot n}$$

corresponding to the linear regression problem

$$\begin{pmatrix}\left[\frac{df}{dp_0}(x_0)\right]_0 & \left[\frac{df}{dp_1}(x_0)\right]_0\\ \left[\frac{df}{dp_0}(x_0)\right]_1 & \left[\frac{df}{dp_1}(x_0)\right]_1\\ \left[\frac{df}{dp_0}(x_1)\right]_0 & \left[\frac{df}{dp_1}(x_1)\right]_0\\ \left[\frac{df}{dp_0}(x_1)\right]_1 & \left[\frac{df}{dp_1}(x_1)\right]_1\end{pmatrix}\cdot\begin{pmatrix}p_0\\ p_1\end{pmatrix} \approx \begin{pmatrix}y_{0,0}\\ y_{0,1}\\ y_{1,0}\\ y_{1,1}\end{pmatrix}\,.$$

```
1  /// linear regression by normal equation method
2  template<typename T>
3  VT<T> linear_regression(MT<T> drdp, VT<T> yobs) {
4    // conditions
5    assert(yobs.size()==nobs*n);
6    assert(drdp.rows()==nobs*n);
7    assert(drdp.cols()==np);
8    // normal equation
9    return (drdp.transpose()*drdp).llt().solve(drdp.transpose()*yobs);
10 }
```

→ nD/optimization/normal_equation/px

Recall that Householder reflection transforms the scalar linear regression problem $\mathbf{r}' \cdot p \approx \mathbf{y}$ into

$$H \cdot \mathbf{r}' \cdot p = \|\mathbf{r}'\|_2 \cdot \mathbf{e}_0 \cdot p = \begin{pmatrix} \|\mathbf{r}'\|_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \cdot p \approx H \cdot \mathbf{y}$$

using the orthogonal Householder matrix $H \in \mathbb{R}^{n_{\text{obs}} \times n_{\text{obs}}}$ $(H = H^T = H^{-1})$. The transformed linear regression problem is feasible due to preservation of vector lengths and of relative positions of $\mathbf{r}' \cdot p$ and $\mathbf{y}$ under multiplication with H. The resulting linear equation

$$\|\mathbf{r}'\|_2 \cdot p = [H \cdot \mathbf{y}]_0$$

has the unique solution

$$p = \frac{[H \cdot \mathbf{y}]_0}{\|\mathbf{r}'\|_2} \,.$$

Obviously, $H^T = H$ as

$$H^T = \left(I_m - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}}\right)^T = I_m - 2 \cdot \frac{\left(\mathbf{v} \cdot \mathbf{v}^T\right)^T}{\mathbf{v}^T \cdot \mathbf{v}} = I_m - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}} = H$$

implying symmetry of H. Orthogonality of H follows from

$$\begin{aligned} H^T \cdot H = H \cdot H &= \left(I_m - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}}\right) \cdot \left(I_m - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}}\right) \\ &= I_m - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}} - 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}} + 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}} \cdot 2 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}} \\ &= I_m - 4 \cdot \frac{\mathbf{v} \cdot \mathbf{v}^T}{\mathbf{v}^T \cdot \mathbf{v}} + 4 \cdot \frac{\mathbf{v} \cdot \left(\mathbf{v}^T \cdot \mathbf{v}\right) \cdot \mathbf{v}^T}{\left(\mathbf{v}^T \cdot \mathbf{v}\right) \cdot \mathbf{v}^T \cdot \mathbf{v}} \\ &= I_m = H^{-1} \cdot H \,. \end{aligned}$$

To solve the linear regression problem

$$\frac{d\mathbf{r}}{d\mathbf{p}} \cdot \mathbf{p} \approx \mathbf{y}$$

with $\mathbf{y} \in \mathbb{R}^{n_{\text{obs}} \cdot n}$ denoting the row-wise serialization of $Y \in \mathbb{R}^{n_{\text{obs}} \times n}$, a sequence of n_p Householder reflections H_i, $i = 0, \ldots, n_p - 1$, is applied to the subdiagonal elements (including the diagonal element) of the columns of

$$\frac{d\mathbf{r}}{d\mathbf{p}} \in \mathbb{R}^{n_{\text{obs}} \cdot n \times n_p}$$

to get a *QR*-factorization as

$$\frac{d\mathbf{r}}{d\mathbf{p}} = Q \cdot R = H_{n_p-1} \cdot \ldots \cdot H_0 \cdot R \ .$$

The j-th column of $\frac{d\mathbf{r}}{d\mathbf{p}}$ contains $n_{\text{obs}} - j$ subdiagonal elements (including the diagonal element). The corresponding Householder matrix

$$\check{H}_j \in \mathbb{R}^{(n_{\text{obs}}-j)\times(n_{\text{obs}}-j)}$$

yields

$$H_j = \begin{pmatrix} I_j & 0 \\ 0 & \check{H}_j \end{pmatrix} \in \mathbb{R}^{n_{\text{obs}} \times n_{\text{obs}}} \ ,$$

where the leading principal $(j \times j)$-submatrix is equal to the identity I_j in $\mathbb{R}^j$. This structure ensures that H_j does not modify the j first rows of

$$H_{j-1} \cdot \ldots \cdot H_0 \cdot \frac{d\mathbf{r}}{d\mathbf{p}}$$

nor the j first entries of

$$H_{j-1} \cdot \ldots \cdot H_0 \cdot \mathbf{y} \ .$$

QR-factorization of $\frac{d\mathbf{r}}{d\mathbf{p}} \in \mathbb{R}^{n_{\text{obs}} \cdot n \times n_p}$ yields

$$Q \cdot R \cdot \mathbf{p} \approx \mathbf{y}$$

implying

$$R \cdot \mathbf{p} \approx Q^T \cdot \mathbf{y}$$

as

$$\begin{aligned} Q^T &= (H_{n_p-1} \cdot \ldots \cdot H_0)^T = H_0^T \cdot \ldots \cdot H_{n_p-1}^T \\ &= H_0^{-1} \cdot \ldots \cdot H_{n_p-1}^{-1} = (H_{n_p-1} \cdot \ldots \cdot H_0)^{-1} = Q^{-1} \ , \end{aligned}$$

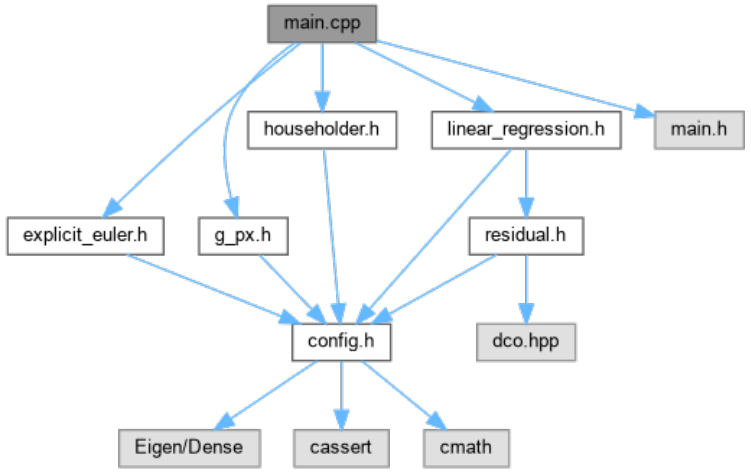
and, hence,

$$(R \cdot \mathbf{p})_{0,\ldots,n_p-1} = (R)_{0,\ldots,n_p-1} \cdot \mathbf{p} = (Q^T \cdot \mathbf{y})_{0,\ldots,n_p-1}$$

as the solution of the system of linear equations with an upper triangular system matrix $R \in \mathbb{R}^{n_{\text{obs}} \times n_p}$. Q is orthogonal. It is typically not symmetric.


```
1  /// linear regression by Householder reflection
2  template<typename T>
3  VT<T> linear_regression(MT<T> drdp, VT<T> yobs) {
4    // conditions
5    assert(yobs.size()==nobs*n); assert(drdp.rows()==nobs*n); assert(drdp.cols()==np);
6    // factorization
7    MT<T> R=drdp, Q_transpose=MT<T>::Identity(nobs*n,nobs*n);
8    VT<T> p(np);
9    int i,j;
10   j=0;
11   while (j<np) {
12     VT<T> x=R.col(j);
13     // ensures identity as leading principal (j x j)-submatrix of H
14     i=0;
15     while (i<j) {
16       x(i)=0;
17       i=i+1;
18     }
19   // computation of j-th Householder matrix
20     VT<T> v=x;
21     v(j)=v(j)-x(j)/fabs(x(j))*x.norm();
```

```
22      VT<T> c=2*v/(v.dot(v));
23      MT<T> H=Eigen::MatrixX<T>::Identity(nobs*n,nobs*n)-2*v*v.transpose()/v.dot(v);
24      // update of Q^T
25      Q_transpose=Q_transpose-c*(v.transpose()*Q_transpose);
26      // update of R
27      R=R-c*(v.transpose()*R);
28      j=j+1;
29    }
30    // solution (backward substitution)
31    VT<T> r=Q_transpose*yobs;
32    i=np-1;
33    while (i>=0) {
34      j=np-1;
35      while (j>i) {
36        r(i)=r(i)-R(i,j)*p(j); j=j-1;
37      }
38      p(i)=r(i)/R(i,i); i=i-1;
39    }
40    return p;
41  }
```

→ nD/optimization/householder/px

- "Finding a stationary point of a local quadratic approximation of the objective" appears to be another reasonable heuristic for iterative minimization.
- The minimum of this local approximation can be computed exactly.
- Convergence of the Newton method is quadratic (fast) in a neighborhood of the solution. It is not guaranteed.
- Good starting values (close to the solution) may be essential. They can be computed by some other *preprocessing* method, e.g., by a few gradient descent steps.
- First and second derivatives of the objective are required. Their accuracy can be crucial for the performance of the Newton method. AD remaines the differentiation method of choice.

As before, we aim to minimize the least squares error of the model predictions wrt. the given data, that is,

$$\min_{\mathbf{p}} e = \min_{\mathbf{p}} \left(\sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} (f(p, x_i)_j - y_{i,j})^2 \right) .$$

Similar to gradient descent, the Newton method is applicable to general unconstrained nonlinear optimization problems.

It can be derived from truncated Taylor expansions of $e = e(\mathbf{p}, \mathbf{x}, Y)$ in direction $\Delta\mathbf{p}$.

A quadratic approximation of the objective $e \in \mathbb{R}$ at $\mathbf{p} \in \mathbb{R}^{n_p}$, that is, truncation of the Taylor expansion

$$e(\mathbf{p}) + (e')^T \cdot \Delta\mathbf{p} + \frac{1}{2} \cdot \Delta\mathbf{p}^T \cdot e'' \cdot \Delta\mathbf{p} + \mathcal{O}\left(\|\Delta\mathbf{p}\|^3\right)$$

after $\Delta\mathbf{p}^2$-term, yields

$$\min_{\Delta\mathbf{p}\in\mathbb{R}^{n_p}} \tilde{e}(\Delta\mathbf{p}) = \min_{\Delta\mathbf{p}} e(\mathbf{p}) + (e')^T \cdot \Delta\mathbf{p} + \frac{1}{2} \cdot \Delta\mathbf{p}^T \cdot e'' \cdot \Delta\mathbf{p} \; .$$

Differentiation yields

$$\frac{d}{d\Delta\mathbf{p}} e(\mathbf{p}) + (e')^T \cdot \Delta\mathbf{p} + \frac{1}{2} \cdot \Delta\mathbf{p}^T \cdot e'' \cdot \Delta\mathbf{p} = (e')^T + \Delta\mathbf{p}^T \cdot e'' \in \mathbb{R}^{1\times n_p}$$

implying

$$\tilde{e}' \equiv \left(\frac{d}{d\Delta\mathbf{p}} \tilde{e}(\Delta\mathbf{p})\right)^T = e' + e'' \cdot \Delta\mathbf{p} \; .$$

The error e is assumed to be twice continuously differentiable wrt. $\mathbf{p}$ implying symmetry of the Hessian

$$e'' \equiv \left(\frac{d^2 e}{d\mathbf{p}^2}\right)^T = \frac{d^2 e}{d\mathbf{p}^2} \in \mathbb{R}^{n_p \times n_p}$$

From the first-order optimality condition $\tilde{e}' = 0$ it follows that

$$e' + e'' \cdot \Delta\mathbf{p} = 0 \quad \Rightarrow \Delta\mathbf{p} = -(e'')^{-1} \cdot e' \; .$$

The Newton step $\Delta\mathbf{p} \in \mathbb{R}^{n_p}$ is computed as the solution of the (symmetric, ideally positive definite) system of linear equations (Newton system)

$$e'' \cdot \Delta\mathbf{p} = -e'$$

followed by the update $\mathbf{p} = \mathbf{p} + \Delta\mathbf{p}$.

A solution to

$$\min_{\mathbf{p}} e(\mathbf{p})$$

satisfies the first-order optimality condition

$$e' = 0 \; .$$

Linearization, that is truncation of the Taylor expansion

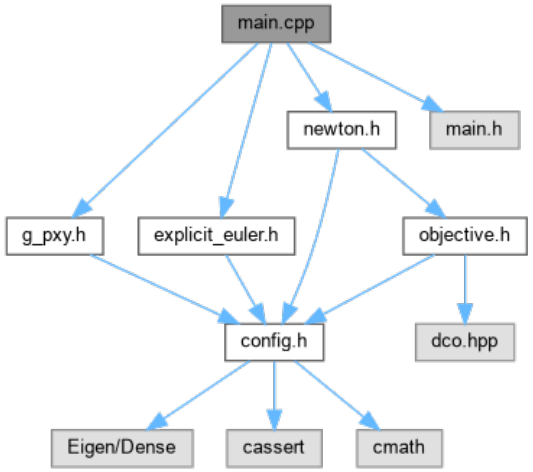
$$e(\mathbf{p}) + (e')^T \cdot \Delta\mathbf{p} + \mathcal{O}\left(\|\Delta\mathbf{p}\|^2\right)$$

after $\Delta\mathbf{p}$-term yields

$$e' + e'' \cdot \Delta\mathbf{p} = 0 \quad \Rightarrow \Delta\mathbf{p} = -(e'')^{-1} \cdot e' \; .$$

and, again, the Newton step $\Delta\mathbf{p} \in \mathbb{R}^{n_p}$ as the solution of the system of linear equations $e'' \cdot \Delta\mathbf{p} = -e'$. Validation of a local minimum at requires the Hessian e'' to be positive definite at the solution.


```
1  /// stationary point of error wrt. model parameter by Newton method
2  template<typename T>
3  VT<T> fit(VT<T> p, VT<T> xobs, MT<T> yobs) {
4    // conditions
5    assert(p.size()==np); assert(xobs.size()==nobs);
6    assert(yobs.rows()==nobs); assert(yobs.cols()==n);
7    // gradient of error wrt. model parameter
8    VT<T> dedp=de_dp(p,xobs,yobs);
9    int i=0;
10   do {
11     // Newton step requires Hessian of error wrt. model parameter
12     MT<T> ddedpp=dde_dp_dp(p,xobs,yobs);
13     // Newton step solved by LDLT factorization to account for
14     // potential indefiniteness of the Hessian
15     p=p+ddedpp.ldlt().solve(-dedp);
16     dedp=de_dp(p,xobs,yobs);
17     i=i+1;
18   } while ((dedp.norm()>eps)&&(i<1000)); // guaranteed to terminate
19   return p;
20 }
```



→ nD/optimization/newton/pxy

- Linearization of the residual yields an iterative method for nonlinear regression problems (Scenario 6).
- Convergence is not guaranteed. Conditions apply.
- The iterative updates are computed as solutions to linear regression problems. Both normal equation and Householder methods can be applied.
- Solutions can differ from those computed by, e.g., gradient descent or Newton methods.

Model:

$$f(x, \mathbf{p}) : \mathbb{R} \times \mathbb{R}^{n_p} \to \mathbb{R}^n$$

Data:

$$(\mathbf{x}, Y) \in \mathbb{R}^m \times \mathbb{R}^{n_{\text{obs}} \times n}$$

$$\mathbf{x} = (x_i)_{i=0,\ldots,n_{\text{obs}}-1}$$

$$Y = (\mathbf{y}_i)_{i=0,\ldots,n_{\text{obs}}-1} = (y_{i,j})_{i=0,\ldots,n_{\text{obs}}-1;\ j=0,\ldots,n-1}$$

Residual:

$$\mathbb{R}^{n_{\text{obs}} \cdot n} \ni \mathbf{r}(\mathbf{x}, \mathbf{p}, Y) = (r_i(x_i, \mathbf{p}, \mathbf{y}_i))_{i=0,\ldots,n_{\text{obs}}-1} \equiv (f(x_i, \mathbf{p}) - \mathbf{y}_i)_{i=0,\ldots,n_{\text{obs}}-1} \approx 0$$

Objective:

$$e = e(\mathbf{x}, \mathbf{p}, Y) = \|\mathbf{r}(\mathbf{x}, \mathbf{p}, Y)\|_2^2 = \sum_{i=0}^{n_{\text{obs}}-1} \|r_i(x_i, \mathbf{p}, \mathbf{y}_i)\|_2^2 = \sum_{i=0}^{n_{\text{obs}}-1} \sum_{j=0}^{n-1} (f(x_i, \mathbf{p}) - y_{i,j})^2 \ .$$

Formulation of the first-order optimality condition

$$\frac{de}{d\mathbf{p}} = \frac{d\|\mathbf{r}\|_2^2}{d\mathbf{p}} = 0$$

in terms of a linearization (linearity in $\Delta\mathbf{p}$) of the residual $\mathbf{r} = \mathbf{r}(\mathbf{x}, \mathbf{p}, Y)$ as

$$\frac{d\|\mathbf{r} + \frac{d\mathbf{r}}{d\mathbf{p}} \cdot \Delta\mathbf{p}\|_2^2}{d\Delta\mathbf{p}} = 0$$

yields a [damped] iterative optimization scheme for $\mathbf{p}$ as

$$\mathbf{p} := \mathbf{p} + [\alpha \cdot] \Delta\mathbf{p}$$

for $0 < \alpha \leq 1$. The Jacobian $\frac{d\mathbf{r}}{d\mathbf{p}} \in \mathbb{R}^{n_{\text{obs}} \cdot n \times n_p}$ is required.

The linear regression problem

$$\frac{d\mathbf{r}}{d\mathbf{p}} \cdot \Delta\mathbf{p} \approx -\mathbf{r}\,, \quad \mathbf{r} \in \mathbb{R}^{n_{\text{obs}} \cdot n},\ \frac{d\mathbf{r}}{d\mathbf{p}} \in \mathbb{R}^{n_{\text{obs}} \cdot n \times n_p},\ \Delta\mathbf{p} \in \mathbb{R}^{n_p}$$

can be solved using, e.g., the normal equations and Householder methods.

Convergence of the fixpoint iteration $\mathbf{p} = G(\mathbf{p}) = \mathbf{p} + \Delta\mathbf{p}$ requires

$$\|G'(\mathbf{p})\| < 1$$

at the solution $\mathbf{p}^*$ implying existence of a neighborhood of $\mathbf{p}^*$ containing values of $\mathbf{p}$ for which the fixed-point iteration converges to this solution.

The linear regression problem

$$\frac{d\mathbf{r}}{d\mathbf{p}} \cdot \Delta\mathbf{p} \approx -\mathbf{r}$$

yields the normal equation

$$\underbrace{\left(\frac{d\mathbf{r}}{d\mathbf{p}}\right)^T \cdot \frac{d\mathbf{r}}{d\mathbf{p}}}_{A^T \cdot A \in \mathbb{R}^{n_p \times n_p}} \cdot \Delta\mathbf{p} = -\underbrace{\left(\frac{d\mathbf{r}}{d\mathbf{p}}\right)^T}_{\in \mathbb{R}^{n_p \times n_{\text{obs}} \cdot n}} \cdot F$$

which can be solved by LL^T (LDL^T) factorization of the symmetric matrix $A^T \cdot A$.

This method is also known as the Gauss-Newton method.

$A \equiv \frac{d\mathbf{r}}{d\mathbf{p}} = (\mathbf{a}_i)_{i=0}^{n_{\text{obs}} \cdot n - 1} \in \mathbb{R}^{n_{\text{obs}} \cdot n \times n_p}$ and $\mathbf{b} \equiv -\mathbf{r} = (b_i)_{i=0}^{n_{\text{obs}} \cdot n - 1} \in \mathbb{R}^{n_{\text{obs}} \cdot n}$ yield

$$\begin{aligned}
\mathbb{R}^{n_p} \ni 0 &= \left(\frac{d\ \|A \cdot \Delta\mathbf{p} - \mathbf{b}\|_2^2}{d\Delta\mathbf{p}}\right)^T = \left(\frac{d\ \sum_{i=0}^{n_{\text{obs}} \cdot n - 1} (\mathbf{a}_i \cdot \Delta\mathbf{p} - b_i)^2}{d\Delta\mathbf{p}}\right)^T \\
&= 2 \cdot \left(\sum_{i=0}^{m \cdot n - 1} (\mathbf{a}_i \cdot \Delta\mathbf{p} - b_i) \cdot \mathbf{a}_i\right)^T = \sum_{i=0}^{n_{\text{obs}} \cdot n - 1} \mathbf{a}_i^T \cdot (\mathbf{a}_i \cdot \Delta\mathbf{p} - b_i) \\
&= \sum_{i=0}^{n_{\text{obs}} \cdot n - 1} \mathbf{a}_i^T \cdot \mathbf{a}_i \cdot \Delta\mathbf{p} - \sum_{i=0}^{n_{\text{obs}} \cdot n - 1} \mathbf{a}_i^T \cdot b_i = A^T \cdot A \cdot \Delta\mathbf{p} - A^T \cdot \mathbf{b}
\end{aligned}$$

implying

$$\left(\frac{d\mathbf{r}}{d\mathbf{p}}\right)^T \cdot \frac{d\mathbf{r}}{d\mathbf{p}} \cdot \Delta\mathbf{p} = -\left(\frac{d\mathbf{r}}{d\mathbf{p}}\right)^T \cdot \mathbf{r}\,.$$

Regularization aims for improved numerical stability through reducing singularity of the system matrix and yielding, e.g., the Levenberg-Marquardt method

$$\left(\left(\frac{d\mathbf{r}}{d\mathbf{p}}\right)^T \cdot \frac{d\mathbf{r}}{d\mathbf{p}} - \lambda \cdot I_{n_p}\right) \cdot \Delta\mathbf{p} = -\left(\frac{d\mathbf{r}}{d\mathbf{p}}\right)^T \cdot F$$

Different strategies for initialization and evolution of $\lambda \in \mathbb{R}$ exist.

Solution of the linear regression problem

$$\frac{d\mathbf{r}}{d\mathbf{p}} \cdot \Delta\mathbf{p} \approx -\mathbf{r}$$

by Householder reflection factorizes the matrix

$$\frac{d\mathbf{r}}{d\mathbf{p}} \in \mathbb{R}^{n_{\text{obs}} \times n_p} \quad \text{into} \quad \frac{d\mathbf{r}}{d\mathbf{p}} = Q \cdot R$$

with orthogonal $Q \in \mathbb{R}^{n_{\text{obs}} \times n_{\text{obs}}}$ and upper triangular $R \in \mathbb{R}^{n_{\text{obs}} \times n_p}$ followed by the solution of

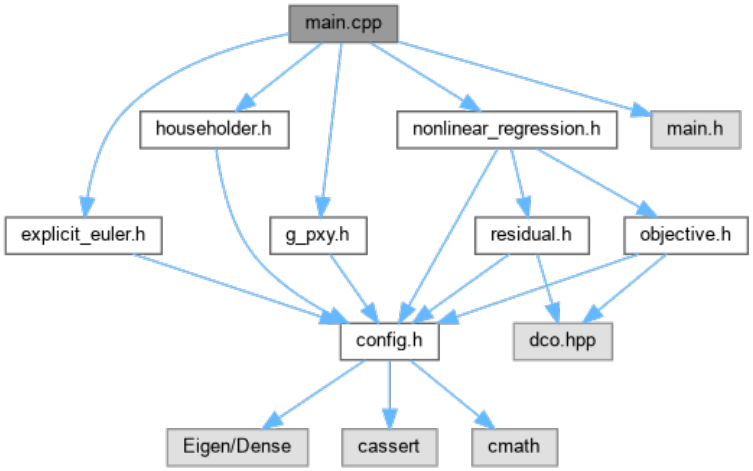
$$R \cdot \Delta p \approx -Q^T \cdot \mathbf{r}$$

yielding

$$(R \cdot \Delta\mathbf{p})_{0,\ldots,n_p-1} = (R)_{0,\ldots,n_p-1} \cdot \Delta\mathbf{p} = (Q^T \cdot \mathbf{r})_{0,\ldots,n_p-1}$$

as the solution of an upper triangular linear system (backward substitution).


```
1  // stationary point of error wrt. model parameter by nonlinear regression
2  template<typename T>
3  VT<T> fit(VT<T> p, VT<T> xobs, MT<T> yobs) {
4    // conditions
5    assert(p.size()==np); assert(xobs.size()==nobs);
6    assert(yobs.rows()==nobs); assert(yobs.cols()==n);
7    T e_=e(p,xobs,yobs);
8    do {
9      VT<T> r_=r(p,xobs,yobs); MT<T> drdp=dr_dp(p,xobs,yobs);
10     // linearization
11     VT<T> delta_p=linear_regression(drdp,r_);
12     // simple line search
13     T e_prev=e_, alpha=2;
14     do {
15       VT<T> p_trial=p;
16       alpha=alpha/2; p_trial=p_trial-alpha*delta_p;
17       e_=e(p,xobs,yobs);
18     } while (e_prev<e_);
19     // iterative update
20     p=p-alpha*delta_p;
21   } while (de_dp(p,xobs,yobs).norm()>eps);
22   return p;
23 }
```

→ nD/[normal_equation | householder]/pxy