Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

A First Course in

Engineering Numerical Software with C++

Uwe Naumann

Informatik 12 (STCE), RWTH Aachen University

Aims / Prerequisites

cppNum v1 : The Scalar Case

- Foundations
  - Algebraic Equations by Newton Method
- Extensions
  - Differential Equations by Explicit Euler Method
  - Convex Objectives by Gradient Descent Method
  - Convex Objectives by Newton Method
  - Differential Equations by Implicit Euler Method

cppNum v2 : The Vector Case

TODO

Aims / Prerequisites

cppNum v1 : The Scalar Case
- Foundations
  - Algebraic Equations by Newton Method
- Extensions
  - Differential Equations by Explicit Euler Method
  - Convex Objectives by Gradient Descent Method
  - Convex Objectives by Newton Method
  - Differential Equations by Implicit Euler Method

cppNum v2 : The Vector Case

TODO

Essential software engineering for numerical methods with C++ including gentle introductions to

- fundamental numerical methods
- selected methods and tools for software engineering
- associated essential C++

Prerequisites

- basic understanding of
  - object-oriented software design
  - object-oriented programming (ideally with C++)
  - numerical methods
- willingness to fill gaps along the way (self-study, Q&A sessions)

All problems are scalar, parameterized, sufficiently often differentiable wrt. both x and p.

The corresponding functions

$$f : \boldsymbol{R} \times \boldsymbol{R} \to \boldsymbol{R} : \quad y = f(x, p)$$

are implemented as differentiable C++ programs.

Derivatives are denoted as

$$f_{x^k} = y_{x^k} \equiv \frac{d^k f}{dx^k}(x, p); \quad f_{p^k} = y_{p^k} \equiv \frac{d^k f}{dp^k}(x, p)$$

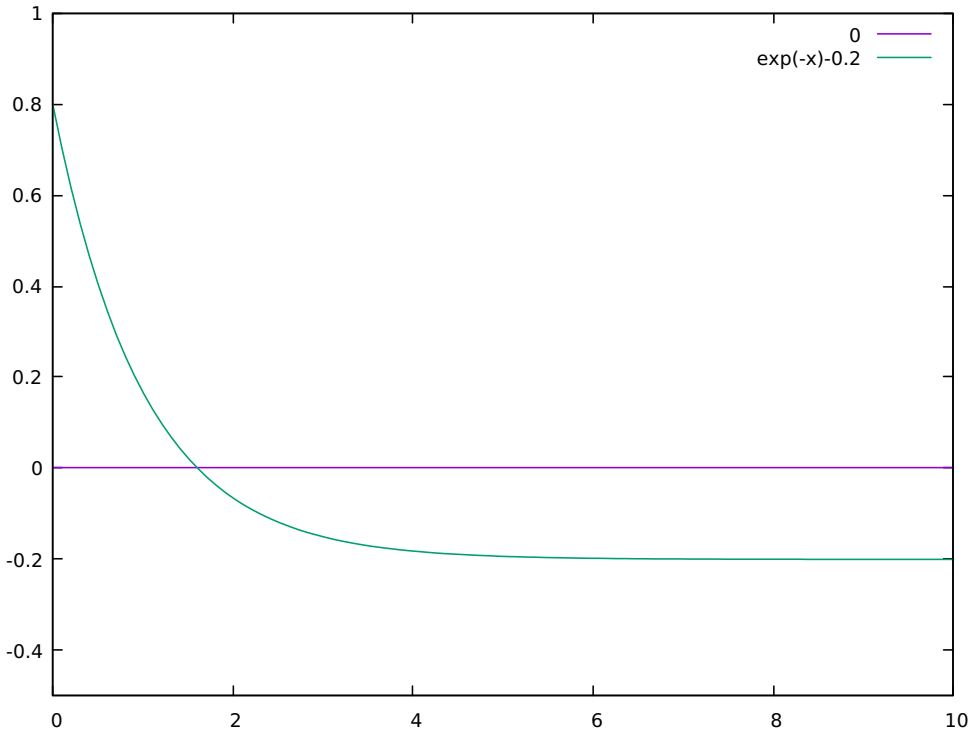
We also write $f_{xx} = f_{x^2}$, $f_{xxx} = f_{x^3}$ and so forth.

We look for roots

$$x = x(p): \quad f(x,p) = 0$$

of user-defined differentiable programs

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



- user requirements
- illustration
- Newton method
- use cases
- essential ad::
- system requirements
- usage incl. essential C++
- design
- implementation incl. essential C++
- build process with `make`
- documentation with `doxygen`
- optional use cases (visualization, parameter sensitivity)

Users of the software want to solve algebraic equations $f(x, p) = 0$ including the ability to

- implement the residual $f(x, p)$
- apply the Newton method for computing x such that $|f(x, p)| < a$ for given accuracy $a > 0$, while ensuring extensibility for adding further methods

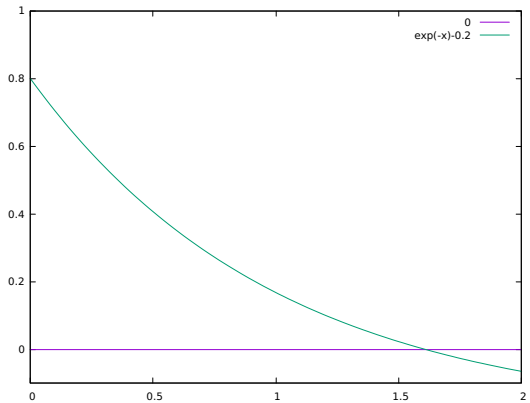
and, optionally, allowing

- visualization of the individual steps performed during the iterative approximation of the solution x with gnuplot[1]
- parameter sensitivity analysis, i.e. computation of x_p using algorithmic differentiation by overloading with ad::

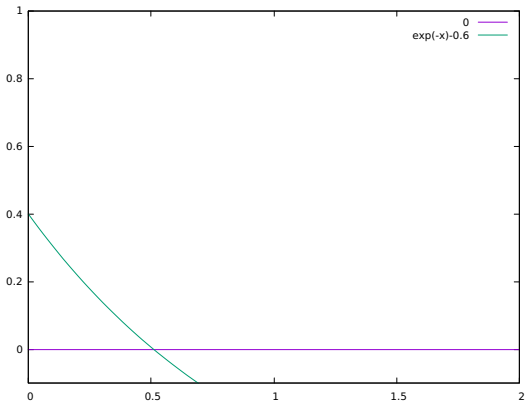
Note: Clear formulation of the user requirements can be difficult in practice.

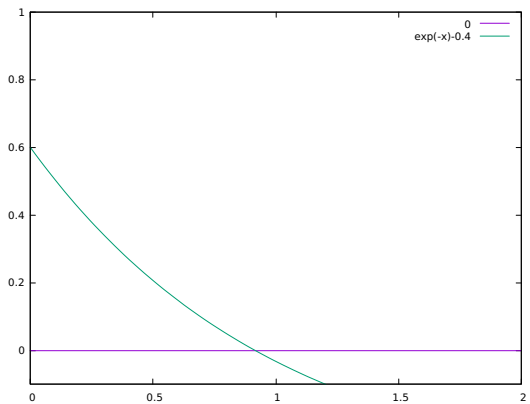
[1] `www.gnuplot.info`

Think of a pandemic [2] with $p \in (0, 1]$ quantifying the percentage of vaccinated people and $x > 0$ being the time to complete extinction of the virus...



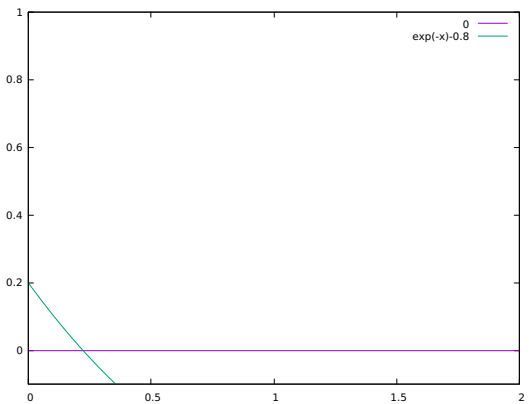
$$f(x, p) = e^{-x} - p = 0$$





$$\Downarrow$$

$$x = -\log(p)$$



[2] Disclaimer: This is not real ...

The Newton method $x^k = x^k(f, x^0, p, a)$ performs k iterations

$$x^{i+1} = x^i - \Delta x^{i+1} = x^i - \frac{f(x^i, p)}{f_x(x^i, p)}, \quad i = 0, \ldots, k-1$$

to approximate a root $x = x(p) : \ f(x, p) = 0$ for given initial estimate x^0 for the solution and p.

The number of iterations $k = k(a)$ is determined by the desired accuracy $a > 0$ ensuring $|f(x^k, p)| < a$.

Conditions for convergence apply.

Many numerical methods for nonlinear problems are based on *linearization*. The Newton method is a prominent example.

The nonlinear function $f(x)$ is replaced locally (at the current x) with a linear (affine; in Δx) approximation derived from the truncated Taylor series expansion of f and "hoping" that

$$f(x + \Delta x) \approx f(x) + f_x(x) \cdot \Delta x \ ,$$

i.e, hoping for a reasonably small remainder.

For $y = f(x, p)$, the Newton iteration follows immediately from

$$f(x^i, p) + f_x(x^i, p) \cdot \Delta x^{i+1} = f(x^i, p) + f_x(x^i, p) \cdot (x^{i+1} - x^i) = 0 \ .$$

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

Newton's method can be regarded as a fixed point iteration

$$x = g(x) = x - \frac{f(x)}{f_x(x)} \, .$$

If at the solution

$$|g_x(x)| < 1 \, ,$$

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

The convergence rate of a fixed-point iteration grows linearly with decreasing values of $|g_x(x)|$.

For $|g_x(x)| = 0$ we get at least quadratic convergence; cubic for $|g_x(x)| = |g_{xx}(x)| = 0$ and so forth.

Newton's method becomes

$$x = g(x) = x - \frac{f(x)}{f_x(x)}$$

yielding

$$g_x(x) = f(x) \cdot \frac{f_{xx}(x)}{(f_x(x))^2} \ .$$

At the solution $f(x) = 0$ implies $g_x(x) = 0$. Assuming a simple root ($f(x) = 0$, $f_x(x) \neq 0$) the second derivative of g becomes equal to

$$g_{xx}(x) = f_x(x) \cdot \frac{f_{xx}(x)}{(f_x(x))^2} + \underbrace{f(x)}_{=0} \cdot (\ldots)$$

implying quadratic convergence within the corresponding neighborhood of the solution if $f_{xx}(x) \neq 0$ as well as convergence after a single iteration for linear f.

For a differentiable program

$$y = f(x)$$

with first derivative $f_x = f_x(x)$ and given $x, \dot{x}$ the tangent (also: forward) mode of algorithmic differentiation computes the directional derivative

$$\dot{y} = f_x \cdot \dot{x}.$$

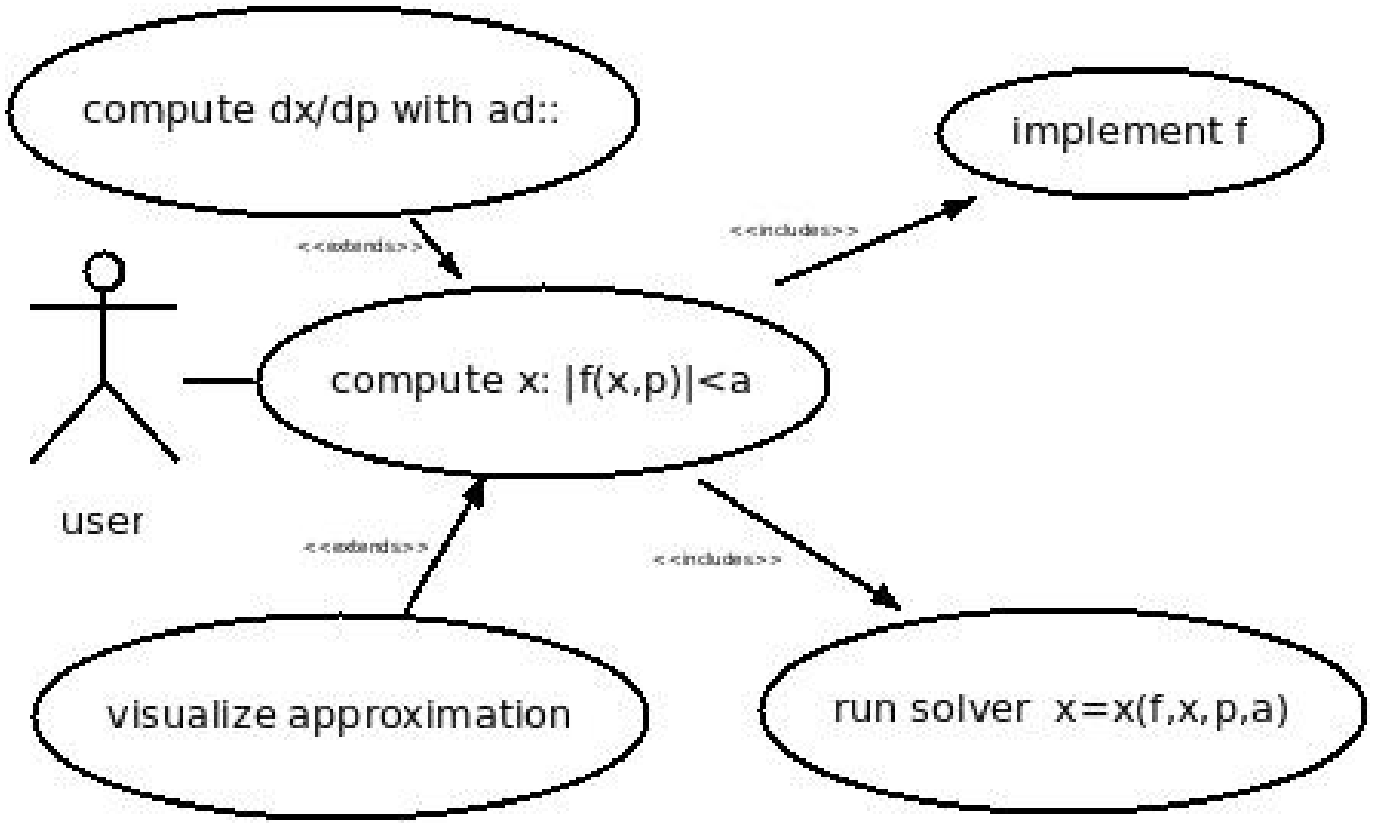
Seeding $\dot{x} = 1$ enables harvesting of $\dot{y} = f_x$.

Example: $y = f(x) = \sin(x) \Rightarrow \dot{y} = f_x \cdot \dot{x} = \cos(x) \cdot \dot{x}$.

The ad:: library implements algorithmic differentiation for arbitrary differentiable C++ programs.

U. Naumann: The Art of Differentiating Computer Programs. An Introduction to Algorithmic DIfferentiation. SIAM 2012.

Static perspective as UML[3] use case diagram.



[3] www.uml.org

Alternatively, a textual representation of the diagram can be used.

1. compute x : $|f(x,p)| < a$

   1.1 ≪includes≫

      1.1.1 implement f

      1.1.2 run solver $x(f,x,p,a)$

   1.2 ≪extended by≫

      1.2.1 compute x_p

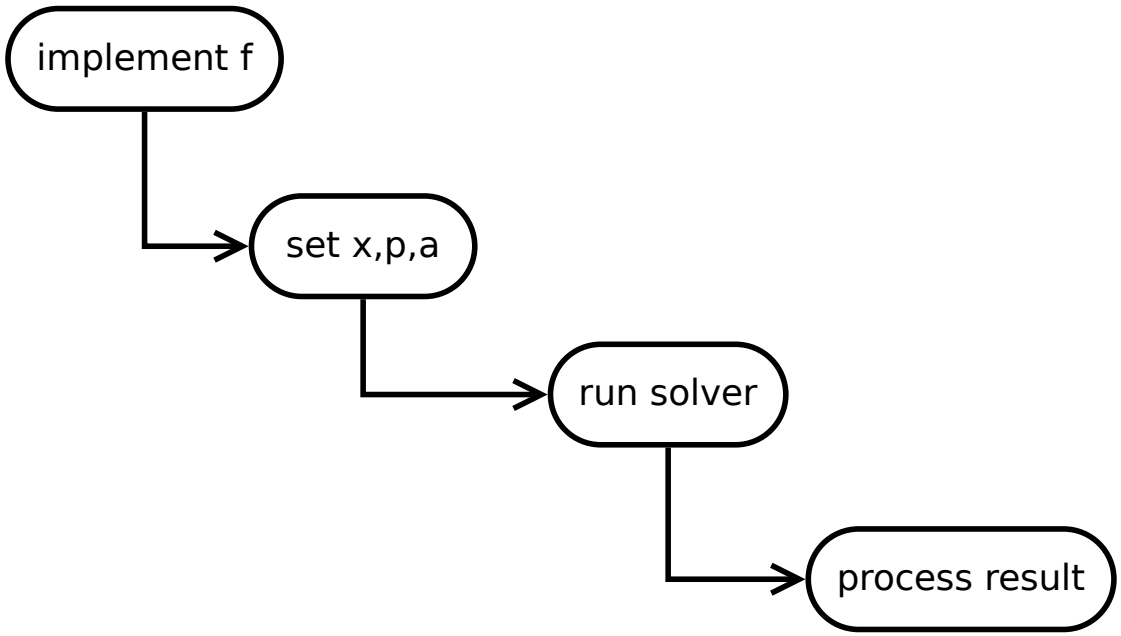
      1.2.2 visualize iteration

Dynamic perspective as three UML activity diagrams corresponding to the main use cases.

- compute $x : |f(x, p) < a|$
- visualize approximation
- compute dx/dp

Prototypes may come in handy during the discussion of use cases with the customer. While (after several interations) the (agreed on) user code should be very close to the final solution pretty much all of the actual functionality (e.g. the solver) may still have to be implemented.

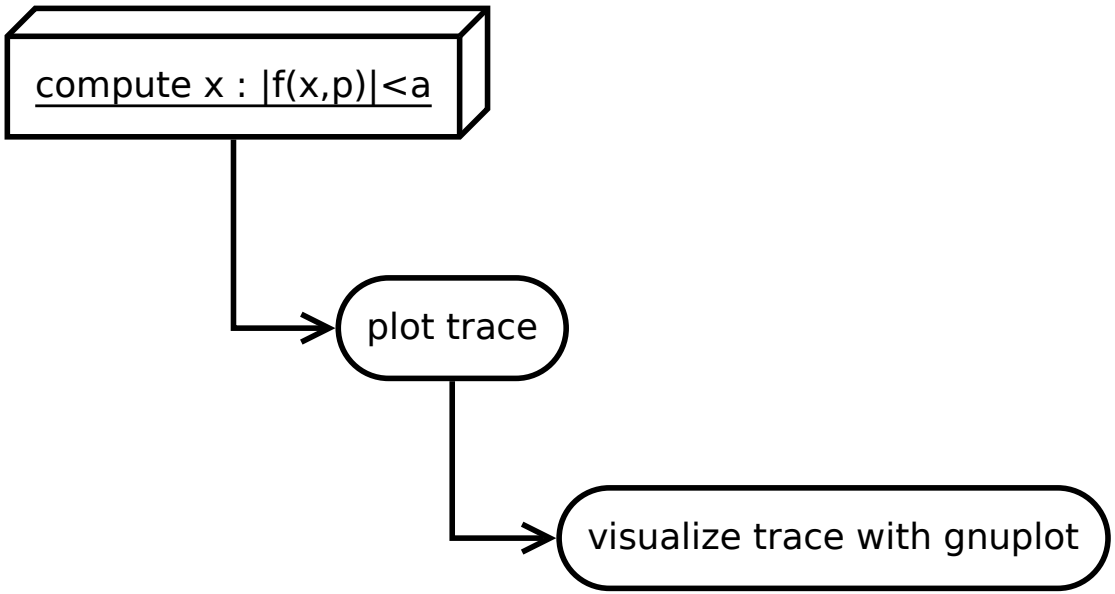
Use Case: Compute $x : |f(x, p) < a|$



Usage (Prototype)

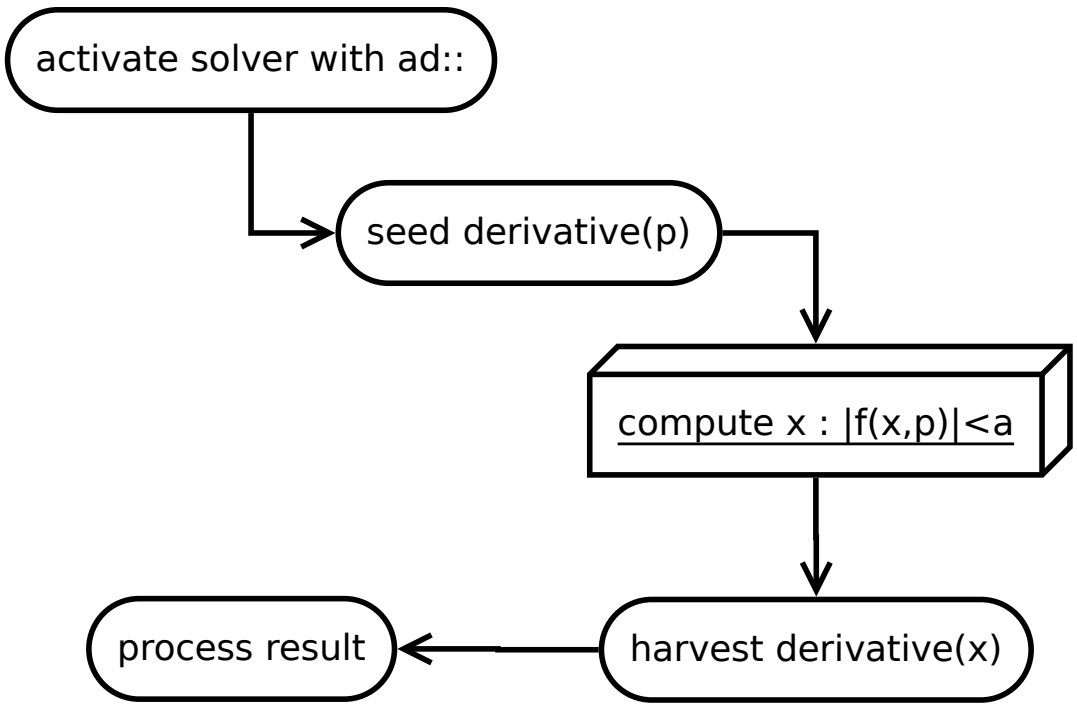
```
1  #include "cppNum/algebraicEquation/equation.hpp"
2  #include <cmath>
3
4  // implement f
5  template<typename T>
6  T ae::equation_t::f(const T &x, const T &p) {
7    using namespace std;
8    return exp(-x)-p;
9  }
10
11 #include "cppNum/algebraicEquation/newton.hpp"
12 #include <iostream>
13
14 int main() {
15   using T=double;
16   // set x,p,a
17   ae::newton_solver_t<T> solver(1e-7); T p=0.5, x=0;
18   // run solver
19   x=solver.run(x,p);
20   // process result
21   std::cout << "f(" << x << ")=" << ae::equation_t::f(x,p) << std::endl;
22   return 0;
23 }
```

Use Case: Visualize approximation



Usage (Prototype)

```
1  #include "cppNum/algebraicEquation/equation.hpp"
2  #include <cmath>
3
4  template<typename T>
5  T ae::equation_t::f(const T &x, const T &p) {
6    using namespace std; // for std::exp
7    return exp(-x)-p;
8  }
9
10 #include "cppNum/algebraicEquation/newton.hpp"
11 #include <iostream>
12
13 int main() {
14   // enable tracing
15   ae::newton_solver_t<> solver(1e-7,true); double p=0.5, x=0;
16   x=solver.run(x,p);
17   // plot trace
18   solver.plot("newton_steps.plt");
19   std::cout << "f(" << x << ")=" << ae::equation_t::f(x,p) << std::endl;
20   return 0;
21 }
```


activate solver with ad::
seed derivative(p)
compute x : |f(x,p)|<a
harvest derivative(x)
process result

Usage (Prototype)


```
1  #include "cppNum/algebraicEquation/equation.hpp"
2  #include <cmath>
3
4  template<typename T>
5  T ae::equation_t::f(const T &x, const T &p) {
6    using namespace std;
7    return exp(-x)-p;
8  }
9
10 #include "cppNum/algebraicEquation/newton.hpp"
11 // use ad::
12 #include "ad.hpp"
13 #include <iostream>
14
15 int main() {
16   // use ad::tangent_t as active type
17   using T=ad::tangent_t<double>;
18   ae::newton_solver_t<T> solver(1e-7); T p=0.5, x=0;
19   // seed derivative(p)
20   ad::derivative(p)=1;
21   // run overloaded solver
22   x=solver.run(x,p);
23   // harvest and process result
24   std::cout << "dx/dp(" << x << ")=" << ad::derivative(x) << std::endl;
25   return 0;
26 }
```

Executable programs require the function **int** main(). It returns an integer (e.g. error code) to the caller (e.g. operating system shell)

```
1 // info@stce.rwth-aachen.de
2 #include <iostream> // from standard library
3
4 int main() {
5   std::cout << "Leave me alone, world!" << std::endl; // output
6   return 0; // success
7 }
```

In-/output requires inclusion of chapter <iostream> of the standard library. Information to be displayed (e.g. a string) is sent to the standard output stream std::cout (the screen) using the heavily overloaded <<-operator. It should be terminated by sending std::endl. Access to functionality from the standard library requires specification of the namespace std.

Comments (//...) improve readability → "Sign" your code in first line.

Typed variables are aliases for memory to hold values of that type. They should be initialized at the time of allocation. The memory is accessible within the associated scopes (13). Aliases for type names can be defined (11).

Explicit specification of namespaces can be avoided within a scope (12,14).

Mathematical functions are defined in chapter <cmath> of the standard library (1,5).

Function templates are instantiated automatically by the compiler for all relevant call scenarios (3–6,14).

Arguments can be passed to functions in various ways, e.g. by constant reference restricting to read-only use inside the function while avoiding a copy (4).

```
1  #include <cmath>
2
3  template<typename T>
4  T f(const T& v) {
5    return std::cos(v);
6  }
7
8  #include <iostream>
9
10 int main() {
11   using T=double;
12   using namespace std;
13   T x=3.14;
14   cout << f(x) << endl;
15   return 0;
16 }
```

Non-static [member] data and [member] functions inside classes are local to all instances (objects). Classes can/should be declared inside namespaces (1–11). In cppNum parameter sensitivity analysis with ad:: requires them to be type-generic (class template; 2–10) e.g. ae::newton_solver_t<T> with variable type T.

Non-static member data (4) enables objects with varying states, e.g. solvers of type ae::newton_solver_t<T> aming for different accuracies.

Use of non-static data inside member functions requires the latter to be non-static (7–9), e.g. ae::newton_solver_t<T>::run).

```
1  namespace ns {
2    template<typename T>
3    class C {
4      int i;
5    public:
6      C(int i) : i(i) {}
7      T f(const T& v) {
8        return i+v;
9      }
10   };
11 }
12
13 #include <iostream>
14
15 int main() {
16   double x=3.14;
17   ns::C<double> c1(42);
18   std::cout << c1.f(x);
19   ns::C<double> c2(24);
20   std::cout << c2.f(x);
21   return 0;
22 }
```

Static member functions (4–7) are shared by all instances of a class.

They can/should be type-generic (4).

Calls do not require instantiation (14).

In cppNum we assume ae::equation_t to be a singleton. The residual ae::equation_t::f of the unique equation to be solved is hence declared as static. Parameter sensitivity analysis with ad:: requires it to be type-generic. Its signature is complete, that is, all data read and/or written by it is passed as an argument / return value.

```
1  namespace ns {
2    class C {
3    public:
4      template<typename T>
5      static T f(const T& v) {
6        return 42+v;
7      }
8    };
9  }
10
11 #include <iostream>
12
13 int main() {
14   std::cout << ns::C::f(3.14);
15   return 0;
16 }
```

Individual classes can/should be declared in separate header files (1), e.g. ae::equation_t in equation.hpp and ae::newton_solver_t<T> in newton.hpp. Infeasible repeated inclusion of a header file is prevented by **#pragma** once.

Maintainability of the source code can benefit from a suitable directory structure.

User-defined code for a given static interface (8 in C.hpp) must be provided as part of the application (4–7 in main.cpp). Failure to do so results in a link time error.

In cppNum the user is required to implement the static member function

T ae::equation_t::f(**const** T &x, **const** T &p).

```
1  // C.hpp
2  #pragma once
3
4  namespace ns {
5    class C {
6    public:
7      template<typename T>
8      static T f(const T&);
9    };
10 }
```

```
1  // main.cpp
2  #include "C.hpp"
3
4  template<typename T>
5  T ns::C::f(const T& v) {
6    return 42+v;
7  }
8
9  #include <iostream>
10
11 int main() {
12   std::cout << ns::C::f(3.14);
13   return 0;
14 }
```

System requirements derive from the agreed on use cases / prototypes.

Functional:

- Newton method $x = x(f, x^0, p, a)$ for the iterative computation of an approximate solution of the algebraic equation $f(x, p) = 0$ with accuracy $a > 0$ for given initial estimate x^0 and parameter p including
  - implementation of $f = f(x, p)$
  - evaluation of f_x
- ability to implement alternative methods
- evaluation of x_p by application of ad:: to the method
- optional record of iterations $(x, f(x, p))$ and output to file for subsequent visualization with gnuplot

Non-functional: implementation in C++, documentation with doxygen[4]

[4] `www.doxygen.nl`

For a differentiable program $y = f(x)$ with first derivative f_x and given $x, \dot{x}$ the tangent mode of algorithmic differentiation computes the directional derivative $\dot{y} = f_x \cdot \dot{x}$.

Seeding $\dot{x} = 1$ enables harvesting of $\dot{y} = f_x$

This functionality is implemented by the ad:: library through overloading of f with the generic type ad::tangent_t<BT> for variable base type BT.

Read/write access to value (ad::value) and derivative (ad::derivative) member data is provided.

```
// info@stce.rwth-aachen.de
#include <cmath>

template<typename T, typename BT>
T f(const T &x, const BT &p) {
  using namespace std;
  return exp(-x)-p;
}

#include "ad.hpp"
#include <iostream>

int main() {
  using BT=double;
  BT p=0.5;
  ad::tangent_t<BT> x=1;
  ad::derivative(x)=1;
  ad::tangent_t<BT> y=f(x,p);
  std::cout << "(f(x,p), df/dx(x,p))=("
    << ad::value(y) << ", "
    << ad::derivative(y) << ')' << std::endl;
  return 0;
}
```

Essentially, C++ software amounts to a collection of classes whose interaction impements the given user requirements.

To indetify class candidates we search for nouns / noun phrases in whatever documentation has been produced so far ...

- Newton solver
- approximation (Newton solver is an approximation method)
- equation (equiv. residual)
- derivative
- application
- record of iterations (Newton solver is an iterative method with optional record)

Associations amongst classes ("Who talks to whom and how?") need to be modelled.

Consolidation of class candidates and formalization of associations amongst them yields a first static perspective on the class model.

- approximation_t ≪is a≫ iteration_t
- ae::solver_t ≪is a≫ approximation_t
- ae::solver_t ≪uses≫ ae::equation_t
- ae::newton_solver_t ≪is a≫ ae::solver_t
- ae::newton_solver_t ≪uses≫ derivative_t
- derivative_t ≪uses≫ ae::equation_t
- derivative_t ≪uses≫ ad::tangent_t
- application ≪owns≫ ae::newton_solver_t

Namespaces collect classes for a specific task (e.g. ae::, ad::). Classes shared by several tasks are placed in the global namespace.

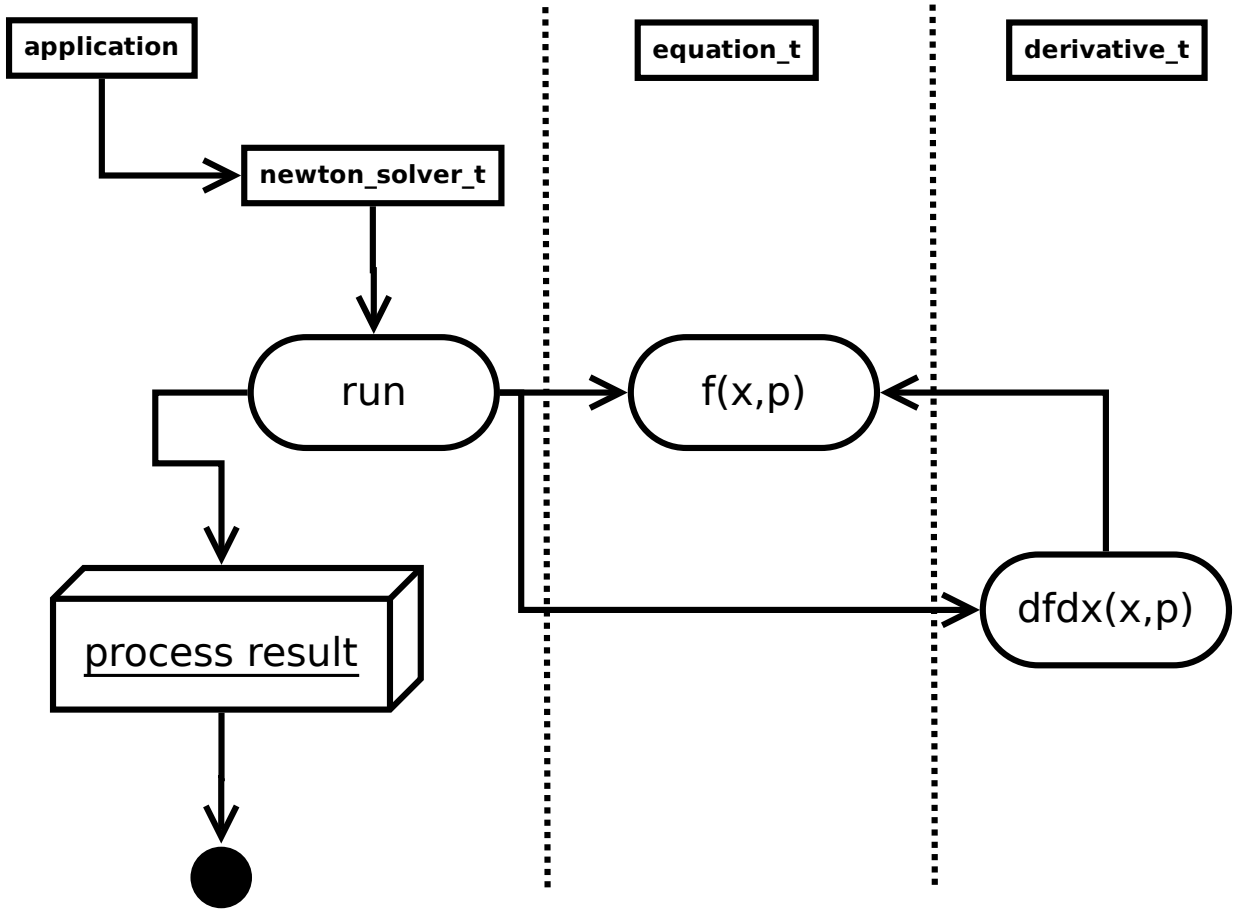
Refinement / completion of the class model requires investigation of the dynamics of all use cases mapped to the reduced class model.

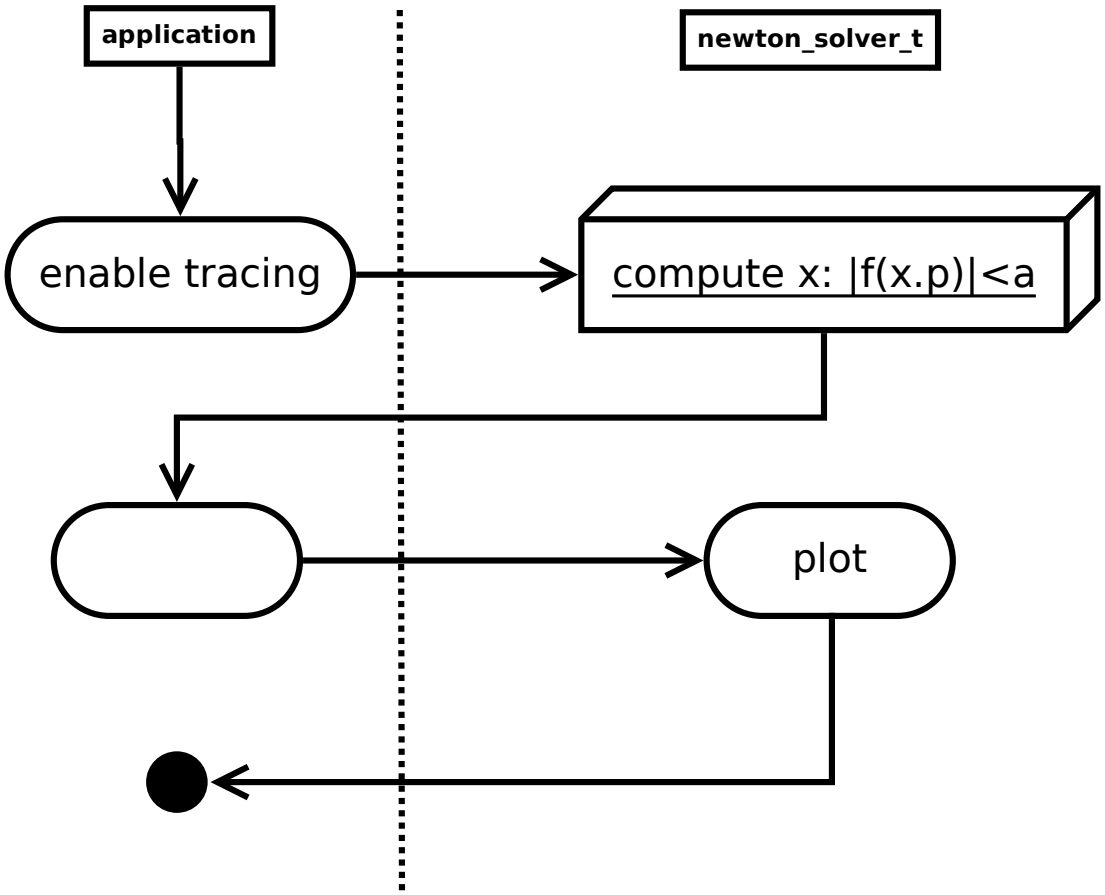
For simplicity we stick with UML activity diagram syntax to visualize the dynamics of the class model with regard to the three use cases

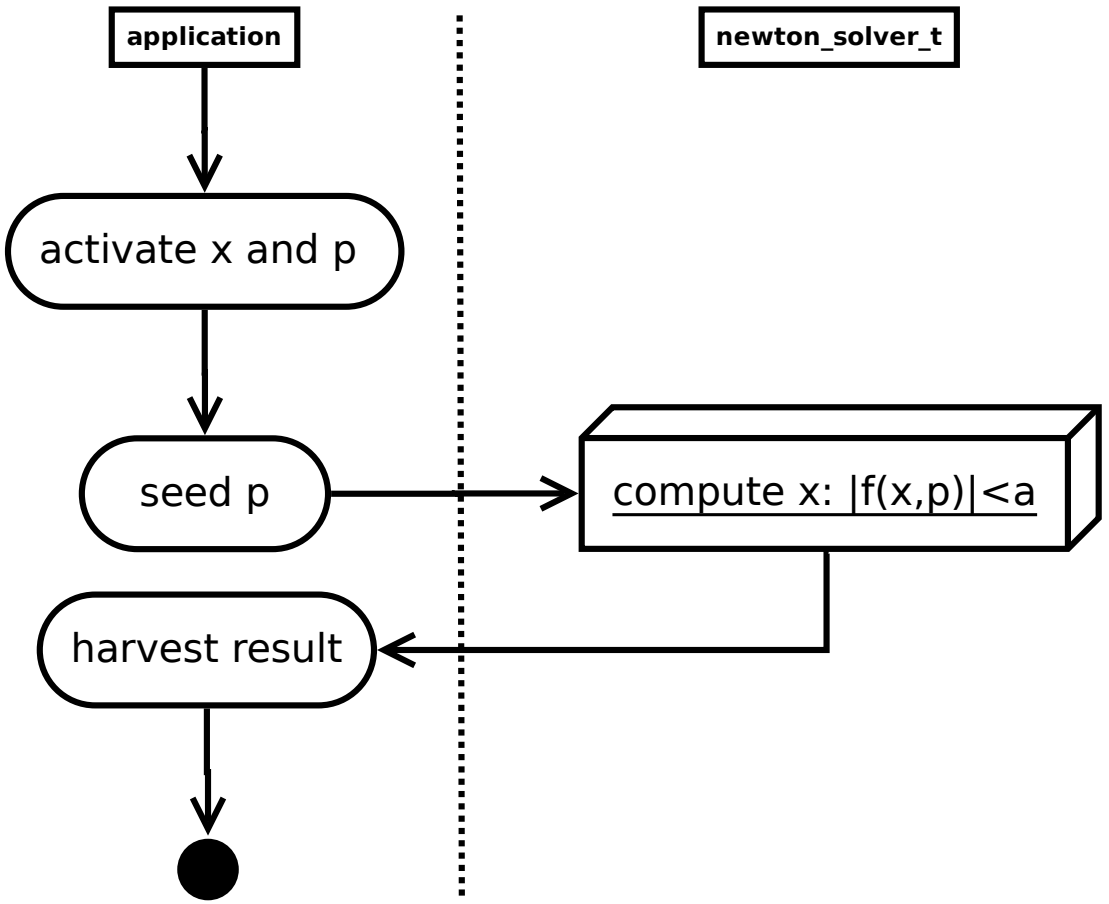
- Compute $x : |f(x, p) < a|$
- Visualize approximation
- Compute dx/dp

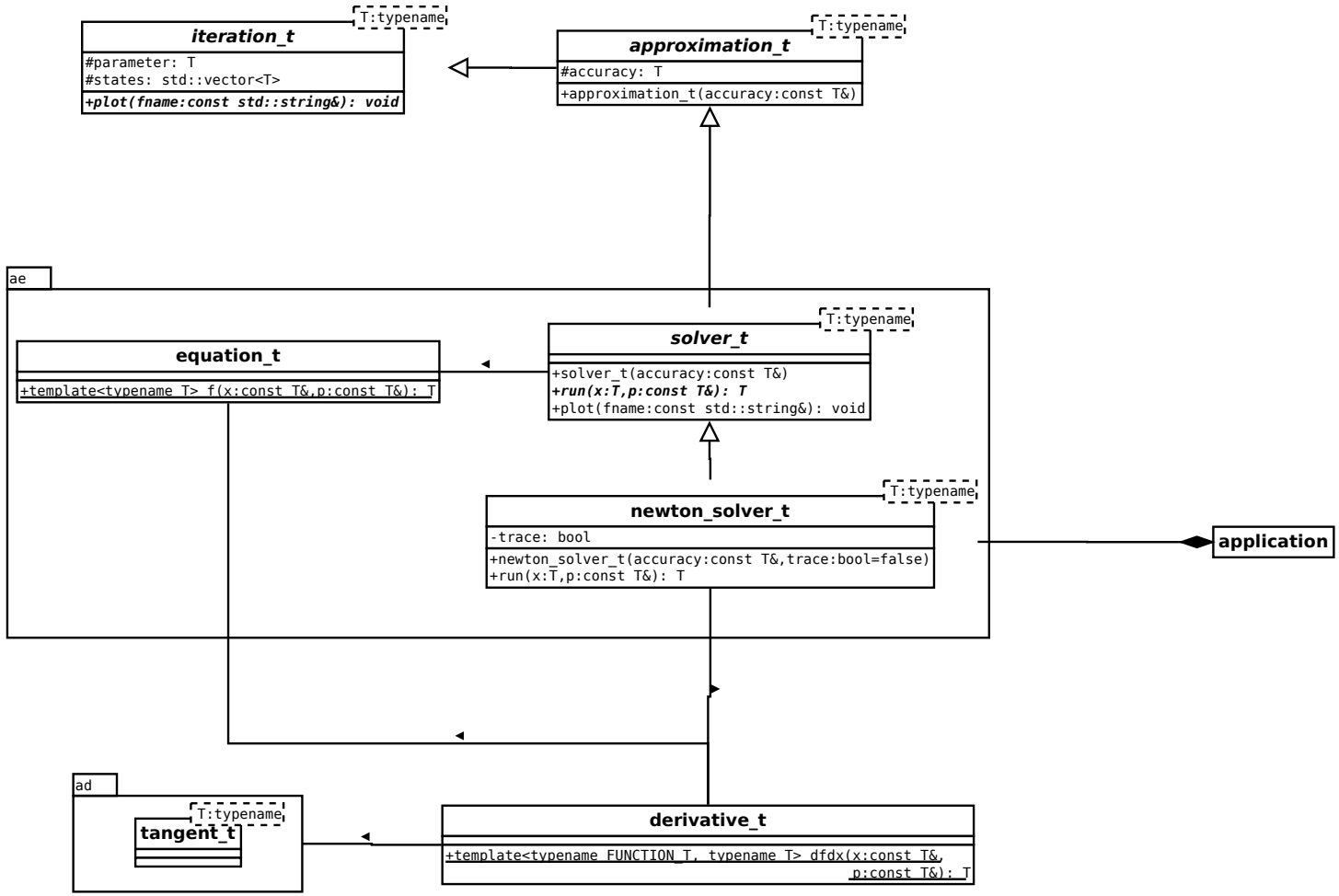
More rigorously, UML sequence / collaboration diagrams should be employed to model the dynamics of use cases at the level of the class model.

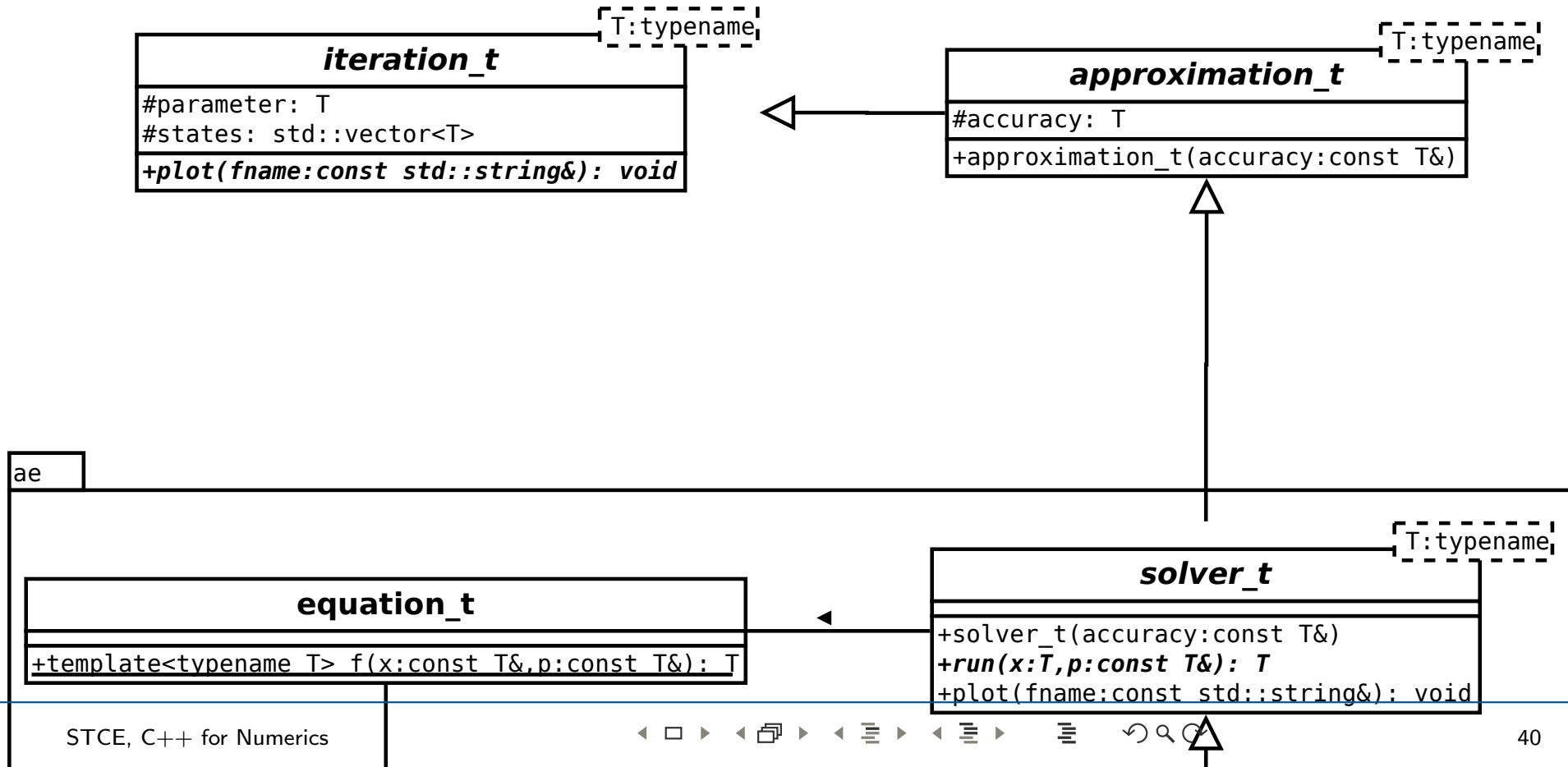
Class Model: Compute $x : |f(x, p) < a|$

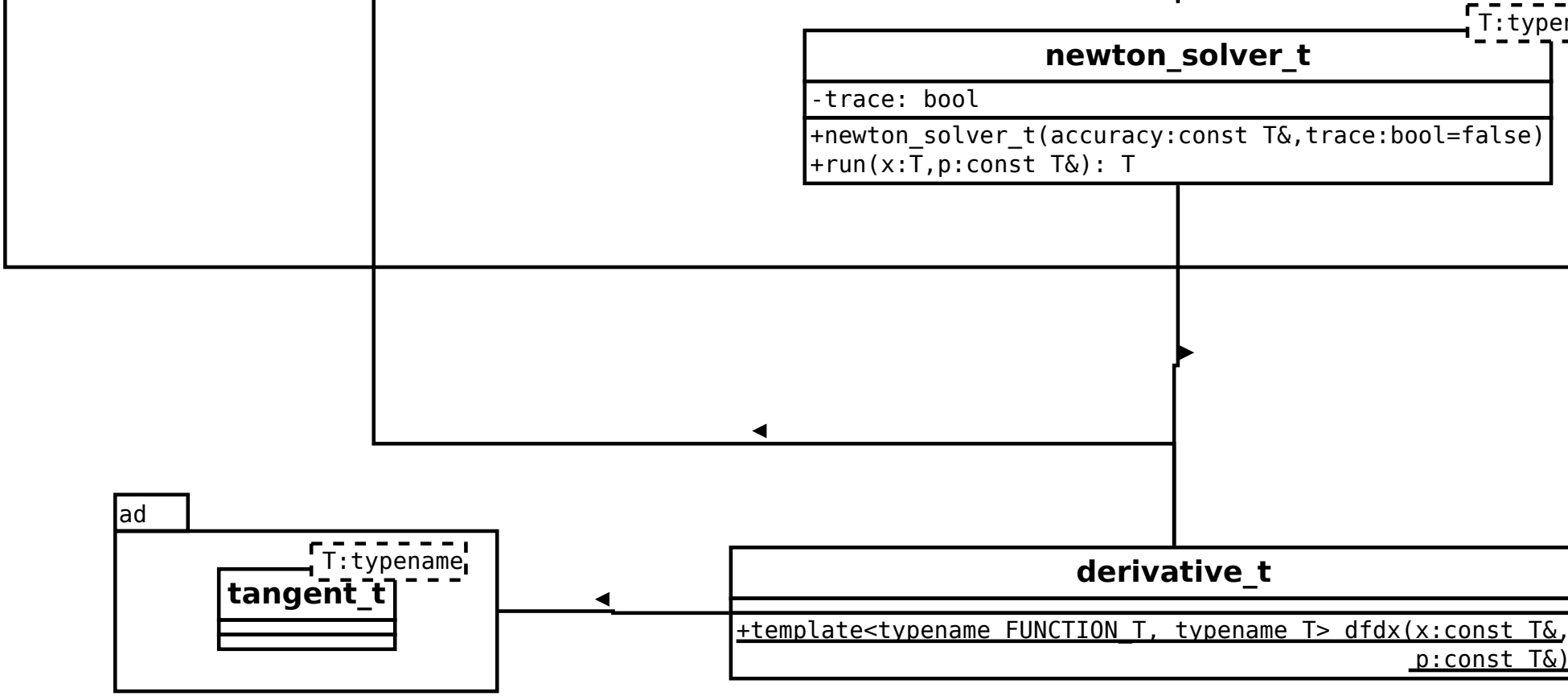



application
newton_solver_t
enable tracing
compute x: |f(x.p)|<a
plot


application
newton_solver_t
activate x and p
seed p
compute x: |f(x,p)|<a
harvest result


iteration_t
T:typename
#parameter: T
#states: std::vector<T>
+plot(fname:const std::string&): void
approximation_t
T:typename
#accuracy: T
+approximation_t(accuracy:const T&)
ae
equation_t
+template<typename T> f(x:const T&,p:const T&): T
solver_t
T:typename
+solver_t(accuracy:const T&)
+run(x:T,p:const T&): T
+plot(fname:const std::string&): void
newton_solver_t
T:typename
-trace: bool
+newton_solver_t(accuracy:const T&,trace:bool=false)
+run(x:T,p:const T&): T
application
ad
tangent_t
T:typename
derivative_t
+template<typename FUNCTION_T, typename T> dfdx(x:const T&,
p:const T&): T


T:typename
iteration_t
#parameter: T
#states: std::vector<T>
+plot(fname:const std::string&): void
T:typename
approximation_t
#accuracy: T
+approximation_t(accuracy:const T&)
ae
T:typename
solver_t
+solver_t(accuracy:const T&)
+run(x:T,p:const T&): T
+plot(fname:const std::string&): void
equation_t
+template<typename T> f(x:const T&,p:const T&): T


T:typen
newton_solver_t
-trace: bool
+newton_solver_t(accuracy:const T&,trace:bool=false)
+run(x:T,p:const T&): T
ad
T:typename
tangent_t
derivative_t
+template<typename FUNCTION_T, typename T> dfdx(x:const T&,
p:const T&)

Implementation: Structure of Source Code


```
cppNum
  algebraicEquation
    equation.hpp
    newton.cpp
    newton.hpp
    solver.cpp
    solver.hpp
  approximation.cpp
  approximation.hpp
  derivative.hpp
  iteration.hpp
apps
  algebraicEquation
    main.cpp // usage
    Makefile // building
    refOutput
      ref.out // testing
    Makefile.inc // build
Doxyfile // documentation
README.md // "first contact"
```

- cppNum library in ./cppNum/ subdirectory
- applications of cppNum library in ./apps/
- build process automated with make
- reference output in ./apps/refOutput/
- documentation of source code with doxygen
- "first contact" through README.md

(by doxygen)

C derivative_t	Host for static differentiation methods
C ae::equation_t	Host for static residual of the algebraic equation
▼ C iteration_t< T >	Abstract base for iterative algorithm enables recording of trace
▼ C approximation_t< T >	
▼ C ae::solver_t< T >	Abstract base for solvers of algebraic equations
C ae::newton_solver_t< T >	Newton solver for algebraic equation

- make test in ./apps/algebraicEquation/
- run ./main.exe
- inspect output

  `f(0.693147)=5.80396e-09`
- modify in main.cpp
  - initial state
  - parameter
  - accuracy

Implementation: iteration.hpp


```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  #include <vector>
5  #include <string>
6
7  /// abstract base for iterative algorithm enables recording of trace
8  template<typename T>
9  class iteration_t {
10   protected:
11     /// value of p
12     T _parameter;
13     /// values of all x visited during iteration
14     std::vector<T> _states;
15     /// tracing mode
16     bool _trace;
17   public:
18     /// tracing mode set at time of construction
19     iteration_t(bool trace);
20     /// specialization requires output of recorded data into file for optional visualization
21     virtual void plot(const std::string&) const=0;
22 };
23
24 #include "iteration.cpp" // implementations of member functions
```

The type std::string (2) implements sequences of characters of variables length.

std::vector<T> (1) implements dynamic arrays of elements of variable type T. Dynamic growth is provided by the append operation push_back (7–9).

Efficient (random) access to the i-th entry of a vector v can be realized by v[i]. Sequential access to the elements of a vector in growing order of their indexes i=0,...,v.size()−1 can be implemented as a range for loop (10–11). Access to the string entries by constant reference ensures efficiency due to lack of copying while making the intent explicit.

```
1  #include <vector>
2  #include <string>
3  #include <iostream>
4
5  int main() {
6    std::vector<std::string> v;
7    v.push_back("Leave me");
8    v.push_back("alone");
9    v.push_back("world!");
10   for (const std::string& s : v)
11     std::cout << s << ' ';
12   std::cout << std::endl;
13   return 0;
14 }
```

yields the following output:

```
Leave me alone world!
```

Declaration of a pure virtual member function (5) makes a base class (e.g. B) abstract. Abstract classes may not be instantiated. Derived, non-abstract specializations (10) need to implement all pure virtual functions.

Private members (e.g. D::j) are accessible from inside the class definition exclusively (14). Protected members (e.g. B::i) are inherited by specializations and hence accessible there (14). Public members are accessible within the scope of a given instance.

Public derivation of a specialization leaves the access rights of members inherited from the base class unchanged (10).

```
1  class B {
2    protected:
3      int i=42;
4    public:
5      virtual void print() const=0;
6  };
7
8  #include <iostream>
9
10 class D : public B {
11     int j=24;
12   public:
13     void print() const {
14       std::cout << i+j;
15     }
16 };
17
18 int main() {
19   D d;
20   d.print();
21   return 0;
22 }
```

yields the following output: 66

Implementation: approximation.[h,c]pp

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  #include "iteration.hpp"
5
6  /** abstract base for approximation algorithms augments iterative methods
7      with an accuracy to define convergence **/
8  template<typename T>
9  class approximation_t : public iteration_t<T> {
10   protected:
11     /// accuracy defines convergence of the iterative method
12     T _accuracy;
13   public:
14     /// constructor sets desired accuracy of approximation and tracing mode
15     approximation_t(const T& accuracy, bool trace);
16 };
17
18 #include "approximation.cpp" // implementations of member functions
```

```
1  template<typename T>
2  approximation_t<T>::approximation_t(const T& accuracy, bool trace) : iteration_t<T>(trace), _accuracy
       (accuracy) {}
```

In template class hierarchies access to inherited members inside specializations requires specification of the name of the hosting base class (e.g. B<T>::v). Accessibility can be declared for the entire class definition by using a corresponding **using** clause (9).

Definition of members outside of the [template] class declaration improves source code structure by keeping the class declaration as compact as possible (12, 17–20; not applied to constructors due to space restrictions).

Constructors of specializations may need to call constructors of their base class(es) explicitely. Initialization of type-generic members requires availability of appropriate constructors for the instantiation type (11).

```
1  template<typename T>
2  class B {
3    protected:
4      T v; B(const T& u) : v(u) {}
5  };
6
7  template<typename T>
8  class D : public B<T> {
9      using B<T>::v;
10   public:
11     D() : B<T>(T(42)) {}
12     void print() const;
13 };
14
15 #include <iostream>
16
17 template<typename T>
18 void D<T>::print() const {
19   std::cout << v;
20 }
21
22 int main() {
23   D<float> d; d.print(); return 0;
24 }
```

yields the following output: 42

Implementation: derivative.hpp

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  #include "ad.hpp"
5
6  /// host for static differentiation methods
7  struct derivative_t {
8    /// first derivative of f wrt. x
9    template<typename FUNCTION_T, typename T>
10   static T dfdx(const T &x, const T &p) {
11     ad::tangent_t<T> x_ad=x, p_ad=p;
12     ad::derivative(x_ad)=1;
13     ad::tangent_t<T> y_ad=FUNCTION_T::f(x_ad,p_ad);
14     return ad::derivative(y_ad);
15   }
16 };
```

See also: Essential ad::

Classes declared as **struct** feature public members exclusively.

Type-generic static member functions enable seemless [recursive] calls to varying instances through template programming by passing host types as template arguments (11,20). Actual implementations of the host members (e.g. A::f) can be provided *late* (16–17), independent of the library code (1–12).

Template arguments of type-generic [member] functions can be deduced by the compiler if the functions are called with actual arguments of the type in question (20). They have to be provided explicitely otherwise, e.g. A.

```
1  struct A {
2    template<typename T>
3    static void f(const T&);
4  };
5
6  struct B {
7    template<typename F, typename T>
8    static void g(const T&);
9  };
10
11 template<typename F, typename T>
12 void B::g(const T& v) { F::f(v); }
13
14 #include <iostream>
15
16 template<typename T>
17 void A::f(const T& v) { std::cout << v; }
18
19 int main() {
20   B::g<A>(3.14); B::g<A>("Pi");
21   return 0;
22 }
```

yields the following output:

```
3.14Pi
```

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  namespace ae {
5
6    /// host for static residual of the algebraic equation
7    struct equation_t {
8      /// residual
9      template<typename T>
10     static T f(const T& x, const T& p);
11   };
12
13 }
```

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  #include "cppNum/approximation.hpp"
5  #include <string>
6
7  namespace ae {
8    /// abstract base for solvers of algebraic equations
9    template<typename T>
10   class solver_t : public approximation_t<T> {
11     protected:
12       using approximation_t<T>::_states;
13       using approximation_t<T>::_parameter;
14     public:
15       /// constructor sets desired accuracy of approximation and tracing mode
16       solver_t(const T& accuracy, bool trace);
17       /// pure virtual solution method allows internal overwrites of x
18       virtual T run(T x, const T& p)=0;
19       /// record of states visited during iteration is written to file with given name
20       void plot(const std::string& filename) const;
21   };
22 }
23
24 #include "solver.cpp" // implementations of member functions
```

```
1  #include "equation.hpp"
2  #include <fstream>
3
4  namespace ae {
5
6    template<typename T>
7    solver_t<T>::solver_t(const T& accuracy, bool trace) : approximation_t<T>(accuracy,trace) {}
8
9    template<typename T>
10   void solver_t<T>::plot(const std::string& filename) const {
11     std::ofstream ofs(filename);
12     for (const auto& state : _states)
13       ofs << state << " 0\n" << state << ' ' << equation_t::f(state,_parameter) << std::endl;
14     ofs.close();
15   }
16
17 }
```

Vectors of fixed (initial) length can be allocated and initialized (5), e.g. an integer vector of length three with all entries equal to 42.

Iterations over the vector entries can be implemented as type-generic (keyword **auto**) range for loops. Write access requires iteration over non-constant references to the entries (7). Constant references restrict access to read-only (9) which suffices for output to the text file `output.txt`.

A corresponding file output stream is opened (8) and closed after completion of the task (11).

```
1  #include <vector>
2  #include <fstream>
3
4  int main() {
5    std::vector<int> v(3,42);
6    int j=0;
7    for (auto& i:v) i+=++j;
8    std::ofstream ofs("output.txt");
9    for (const auto& i:v)
10     ofs << i << '\n';
11   ofs.close();
12   return 0;
13 }
```

yields output.txt with the following contents:

```
43
44
45
```

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  #include "solver.hpp"
5
6  namespace ae {
7
8    /// Newton solver for algebraic equation
9    template<typename T>
10   class newton_solver_t : public solver_t<T> {
11       using solver_t<T>::_states;
12       using solver_t<T>::_parameter;
13       using solver_t<T>::_accuracy;
14       using solver_t<T>::_trace;
15     public:
16       /// constructon of instance with given accuracy; tracing disabled by default
17       newton_solver_t(const T& accuracy, bool trace=false);
18       /// solution method allows internal overwrites of x
19       T run(T x, const T& p);
20   };
21
22 }
23
24 #include "newton.cpp" // implementations of member functions
```

Implementation: algebraicEquation/newton.cpp


```
1  #include "cppNum/derivative.hpp"
2
3  namespace ae {
4
5    template<typename T>
6    newton_solver_t<T>::newton_solver_t(const T& accuracy, bool trace)
7      : solver_t<T>(accuracy,trace) {}
8
9    template<typename T>
10   T newton_solver_t<T>::run(T x, const T &p) {
11     using namespace std; // enable overloading of std::fabs outside of std::
12     if (_trace) { _states.push_back(x); _parameter=p; }
13     T residual=equation_t::f(x,p);
14     do {
15       x-=residual/derivative_t::dfdx<equation_t>(x,p);
16       if (_trace) _states.push_back(x);
17       residual=equation_t::f(x,p);
18     } while (fabs(residual)>_accuracy);
19     return x;
20   }
21
22 }
```

Trailing default arguments can be specified for constructors and other member functions ().

Intraprocedural control flow constructs in C++ include **if**-branches and **do**–**while** -loops. The latter are traversed at least once.

using namespace std facilitates overloading of mathematical functions from **namespace** std with functions from a different namespace, e.g. ad::.

```
1  #include <cmath>
2
3  void f(double& d, bool b=true) {
4    using namespace std;
5    if (b)
6      do {
7        d=sin(d);
8        if (d<0.5) b=false;
9      } while (b);
10 }
11
12 #include <iostream>
13
14 int main() {
15   double d=1.5;
16   f(d);
17   std::cout << d;
18   return 0;
19 }
```

yields the following output:

```
0.481104
```

First Contact: README.md

```
1  # C++ for Numerics
2
3  * root of algebraic equation by Newton method
4  * contact: info@stce.rwth-aachen.de
5
6  ## Building
7  * edit second line in apps/Makefile.inc: set BASE_DIR to absolute path to this file's directory
8  * make depend
9  * make
10
11 ## Testing
12 * make test
13
14 ## Running
15 * ./main.exe in subdirectories of ./apps
16
17 ## Visualizing Results
18 * gnuplot run.gnuplot wherever run.gnuplot available
19
20 ## Cleaning Up
21 * make clean
```

Building: make

We use GNU make[5] for automation of the build process.

```
1  # edit according to your environment
2  BASE_DIR=$(HOME)/Documents/git/stce/cppNum/v1.1
3
4  CPPC=g++ -Wall -Wextra -pedantic -O3 -march=native
5  INC=-I$(BASE_DIR)
6  THIRDPARTY_INC=-I$(BASE_DIR)/../thirdParty/ad
7
8  main.exe : main.o
9          $(CPPC) $< -o$@
10
11 main.o : main.cpp
12         $(CPPC) -c $(INC) $(THIRDPARTY_INC) $< -o$@
13
14 depend : ...
15
16 test: ...
17
18 clean:
19         rm -f *.o *.exe Makefile.bak *.out
20
21 .PHONY: depend test clean
```

[5] www.gnu.org/software/make/manual/make.html

Selective recompilation may be essential for an efficient build process. It is facilitated by the Linux tool makedepend[6].

```
1 depend :
2         makedepend $(INC) main.cpp
```

appends dependences of main.o, that is,

```
1  main.o: .../cppNum/algebraicEquation/equation.hpp
2  main.o: .../cppNum/algebraicEquation/newton.hpp
3  main.o: .../cppNum/algebraicEquation/solver.hpp
4  main.o: .../cppNum/approximation.hpp
5  main.o: .../cppNum/iteration.hpp
6  main.o: .../cppNum/approximation.cpp
7  main.o: .../cppNum/algebraicEquation/solver.cpp
8  main.o: .../cppNum/algebraicEquation/equation.hpp
9  main.o: .../cppNum/algebraicEquation/newton.cpp
10 main.o: .../cppNum/derivative.hpp
```

to Makefile.

[6] http://manpages.ubuntu.com/manpages/trusty/man1/makedepend.1.html

```
1 ...
2
3 test: main.exe
4         ./$< > main.out
5         diff main.out refOutput/ref.out
6 ...
```

redirects standard output to main.out, e.g.

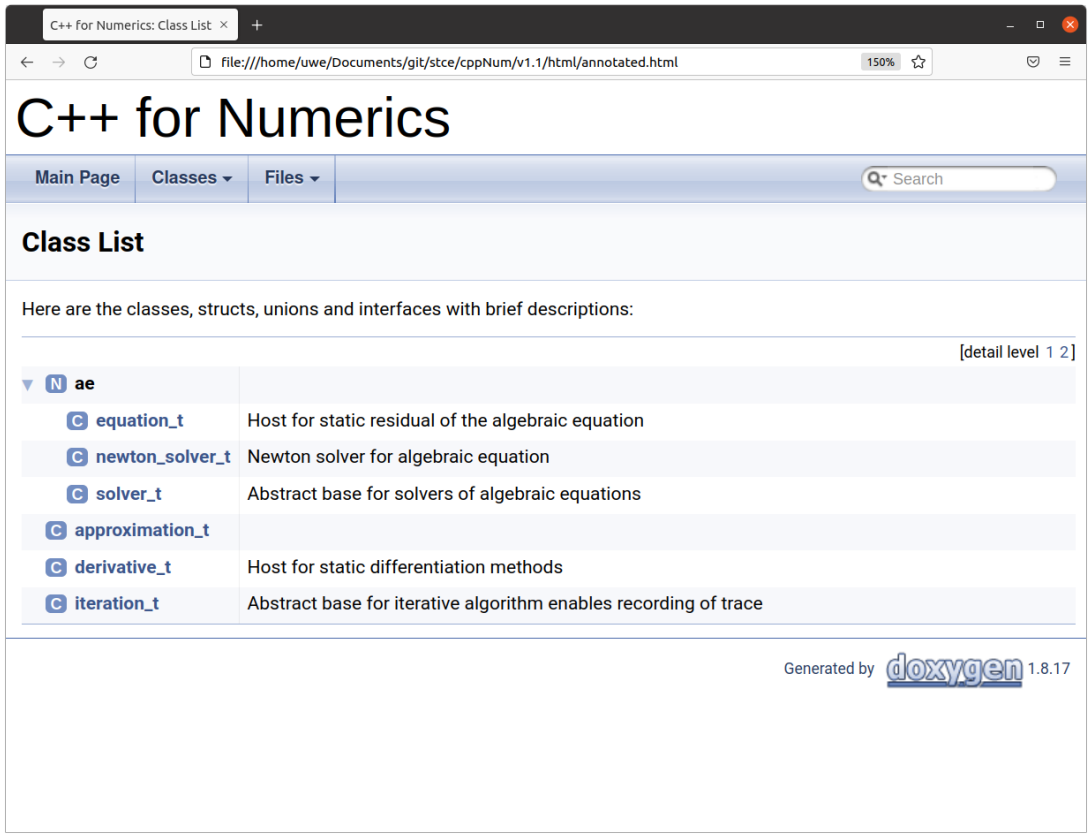
```
1 f(0.693147)=5.80396e-09
```

and compares it with the reference output.

Consider googletest [7] as a less basic test environment.

[7] `github.com/google/googletest`

Doxygen supports thorough documentation of C++ code through automatic conversion into (e.g.) the web browser format HTML.



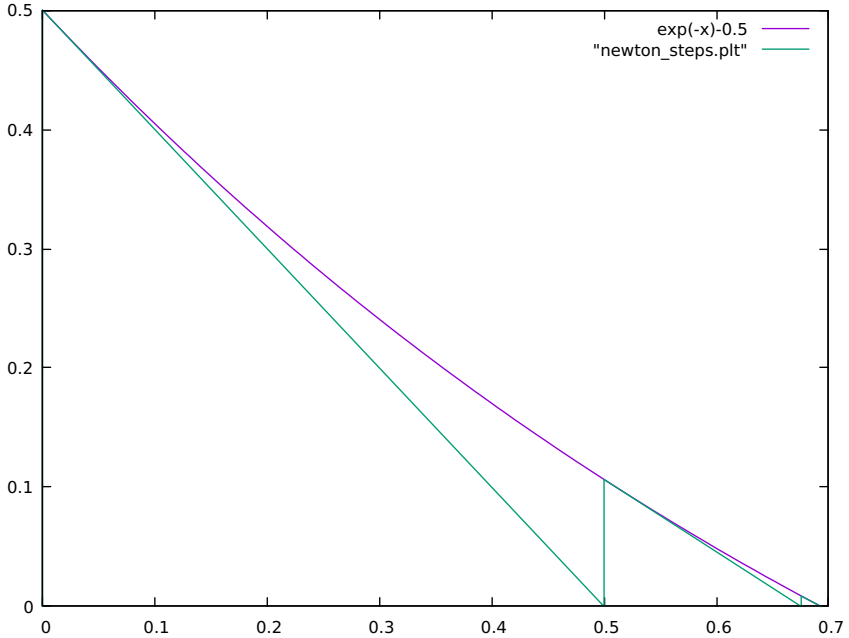
Visualization: User Code


```
1  // info@stce.rwth-aachen.de
2  #include "cppNum/algebraicEquation/equation.hpp"
3  #include <cmath>
4
5  template<typename T>
6  T ae::equation_t::f(const T &x, const T &p) {
7    using namespace std;
8    return exp(-x)-p;
9  }
10
11 #include "cppNum/algebraicEquation/newton.hpp"
12 #include <iostream>
13
14 int main() {
15   using T=double;
16   T p=0.5, x=0;
17   // enable tracing
18   ae::newton_solver_t<T> solver(1e-7,true);
19   // record trace
20   x=solver.run(x,p);
21   // write trace to file
22   solver.plot("newton_steps.plt");
23   std::cout << "f(" << x << ")=" << ae::equation_t::f(x,p) << std::endl;
24   return 0;
25 }
```

```
1 plot exp(-x)-0.5, "newton_steps.plt" with lines
2 pause -1 "Close gnuplot window."
```

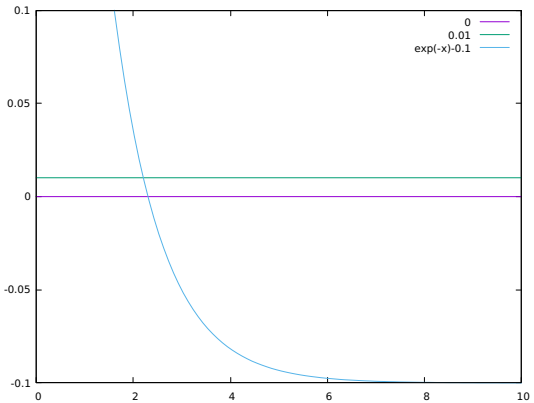
```
:-) gnuplot run.gnuplot
```

```
1  0 0
2  0 0.5
3  0.5 0
4  0.5 0.106531
5  0.675639 0
6  0.675639 0.00883099
7  0.692995 0
8  0.692995 7.61914e-05
9  0.693147 0
10 0.693147 5.80396e-09
```

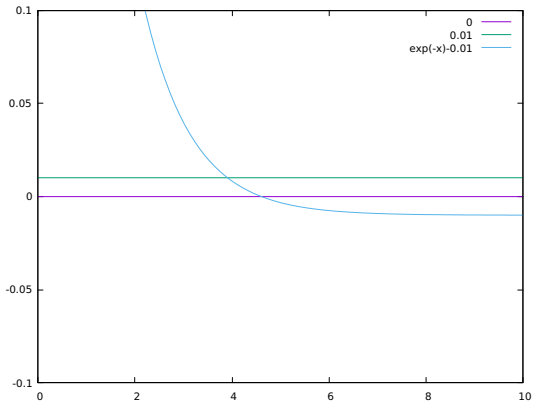


A numerical approximation of the solution needs to be computed in most less trivial cases.

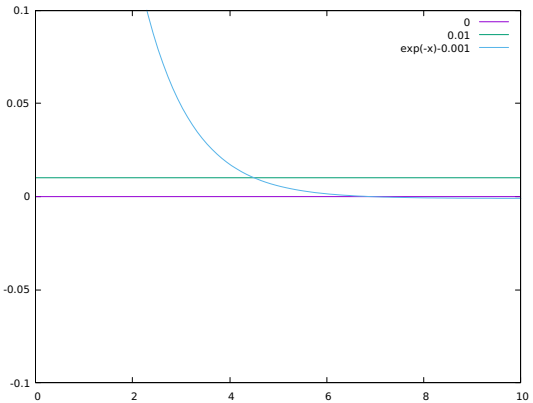
The estimate for the solution becomes more sensitive wrt. errors in the parameter for decreasing values of the parameter.



error ≈ 0.5



error ≈ 1



error ≈ 2

Parameter sensitivity is essential for the interpretation of the numerical solution.

(Forward, backward, centeral) finite differences can be used to approximate derivatives of f at a given point (x, p).

$$f_x(x,p) \approx_1 \frac{f(x+\Delta x,p)-f(x,p)}{\Delta x} \approx_1 \frac{f(x,p)-f(x-\Delta x,p)}{\Delta x}$$
$$\approx_2 \frac{f(x+\Delta x,p)-f(x-\Delta x),p}{2\cdot\Delta x}$$

where $\Delta x = \Delta x(x)$ is picked as a compromise between accuracy and numerical stability, e.g,

$$\Delta x = \begin{cases} \sqrt{\epsilon} & \tilde{x} = 0 \\ \sqrt{\epsilon}\cdot|\tilde{x}| & \tilde{x} \neq 0 \end{cases}$$

with machine epsilon ϵ dependent on the floating-point precision.

Perturbations of p yield corresponding derivatives wrt. p.

Forward and backward/central finite differences exhibit first-/second-order accuracy ($\approx_1/\approx_2$; error scales with $\Delta x^2/\Delta x^3$).

Parameter Sensitivities with Finite Differences


```
1  // info@stce.rwth-aachen.de
2  #include "cppNum/algebraicEquation/equation.hpp"
3  #include <cmath>
4
5  template<typename T>
6  T ae::equation_t::f(const T &x, const T &p) {
7    using namespace std;
8    return exp(-x)-p;
9  }
10
11 #include "cppNum/algebraicEquation/newton.hpp"
12 #include <limits>
13 #include <iostream>
14
15 int main() {
16   using T=double;
17   T p=0.5, x=0;
18   ae::newton_solver_t<T> solver(1e-7);
19   T h=std::sqrt(std::numeric_limits<T>::epsilon());
20   T xm=solver.run(x,p-h); T xp=solver.run(x,p+h);
21   x=solver.run(x,p);
22   std::cout << "dx/dp(" << x << ")=" << (xp-xm)/(h+h) << std::endl;
23   return 0;
24 }
```

Parameter Sensitivities with ad::


```
1  // info@stce.rwth-aachen.de
2  #include "cppNum/algebraicEquation/equation.hpp"
3  #include <cmath>
4
5  template<typename T>
6  T ae::equation_t::f(const T &x, const T &p) {
7    using namespace std;
8    return exp(-x)-p;
9  }
10
11 #include "cppNum/algebraicEquation/newton.hpp"
12 #include "ad.hpp"
13 #include <iostream>
14
15 int main() {
16   using T=ad::tangent_t<double>;
17   T p=0.5, x=0;
18   ae::newton_solver_t<T> solver(1e-7);
19   ad::derivative(p)=1;
20   x=solver.run(x,p);
21   std::cout << "dx/dp(" << x << ")=" << ad::derivative(x) << std::endl;
22   return 0;
23 }
```

- finite differences

```
dx/dp(0.693147)=-2
```

- ad::

```
dx/dp(0.693147)=-2
```

We consider initial value problems for explicit ordinary differential equations yielding solutions $x(p, t)$ depending on a parameter p and time t.

For given $x^0(p) = x(p, 0)$ and target time $t_{\text{end}} > 0$ the solution $x(p, t_{\text{end}})$ is obtained by integration of the explicit ordinary differential equation

$$x_t = g(x, p)$$

with user-defined g.

cppNum v1.2 implements the explicit (also: forward) Euler integration method for the iterative approximation of the solution.

- user requirements
- illustration
- explicit Euler method
- use cases
- system requirements
- usage incl. essential C++
- revised design
- implementation incl. essential C++
- optional use cases (visualization, parameter sensitivity)

Users of the software want to solve initial value problems including the ability to

- implement $g(x, p)$
- apply the explicit Euler integration method for computing $x(g, x^0, p, t_{\text{end}})$ and ensuring extensibility for adding further integration methods

and, optionally, allowing

- visualization the individual steps performed during the iterative integration of the solution $x(p, t_{\text{end}})$
- parameter sensitivity analysis, i.e. computation of x_p and x_{x^0} using algorithmic differentiation by overloading with ad::

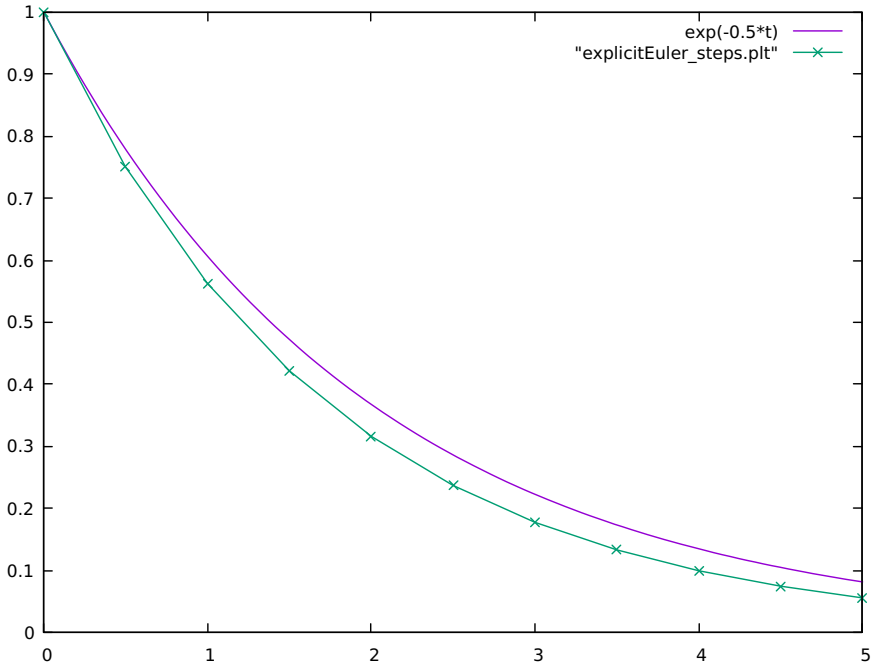
The initial value problem

$$x_t = -p \cdot x, \quad x(p,0) = 1$$

can be solved analytically yielding

$$x = e^{-p \cdot t} .$$

The explicit Euler method approximates this solution. A small time step may be required for sufficient accuracy.



An unreasonably large time step of $0.5s$ is used in the plot to visualize the approximative nature of the method.

The explicit Euler method replaces the time derivative x_t in

$$x_t = g(x(t), p)$$

with a forward finite difference yielding

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

and, hence, the iterative approximation of the solution as

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

for given $x(0) = x^0$ and $\Delta t > 0$.

1. compute $x(t_{\text{end}}) = x(g, x^0, p, t_{\text{end}}) : \; x_t = g(x, p), \; x(0) = x^0$

   ≪includes≫

   1.1 implement g

   1.2 run $x(g, x^0, p, t_{\text{end}})$

   ≪extended by≫

   1.1 compute x_p

   1.2 compute $\frac{dx}{dx^0}$

   1.3 visualize evolution

The dynamics ($\rightarrow$ UML activity diagrams) are similar to v1.1.

Usage (Prototype)

```
1  #include "cppNum/differentialEquation/equation.hpp"
2
3  // implement g
4  template<typename T>
5  T de::equation_t::g(const T &x, const T &p) {
6    return -p*x;
7  }
8
9  #include "cppNum/differentialEquation/explicitEuler.hpp"
10 #include <iostream>
11
12 int main() {
13   using T=double;
14   // set x,p,t_end(=1), dt=t_end/m (m=100)
15   de::explicitEuler_integrator_t<T> integrator(1,100); T p=0.5, x=1;
16   // run integrator
17   x=integrator.run(x,p);
18   // process result
19   std::cout << "x=" << x << std::endl;
20   return 0;
21 }
```

Functional:

- explicit Euler method $x = x(g, x^0, p, t_{\text{end}}, m)$ for solving the initial value problem $x_t = g(x, p)$, $x(p, 0) = x^0$ for given parameter p and target time t_{end} reached by performing m time steps including
  - implementation of g
- ability to implement alternative integration methods
- evaluation of x_p and x_{x^0} by application of ad:: to the method
- optional record of evolution $(t, x(p, t))$ and output to file for visualization with gnuplot

Non-functional: implementation in C++, documentation with doxygen

- evolution_t ≪is a≫ iteration_t
- de::integrator_t ≪is a≫ evolution_t
- de::integrator_t ≪uses≫ de::equation_t
- de::explicitEuler_integrator_t ≪is a≫ de::integrator_t
- application ≪owns≫ de::explicitEuler_integrator_t

The dynamics (→ e.g. UML activity diagrams) are similar to v1.1.

The full design (→ e.g. UML class diagram) includes member data and functions; see implementation.

Implementation: Structure of Source Code


```
cppNum
  differentialEquation
    equation.hpp
    explicitEuler.cpp
    explicitEuler.hpp
    integrator.cpp
    integrator.hpp
  evolution.cpp
  evolution.hpp
  iteration.hpp
apps
  differentialEquation
    main.cpp
    Makefile
      refOutput
        ref.out
  Makefile.inc
Doxyfile
README.md
```

- cppNum library in ./cppNum/ subdirectory
- applications of cppNum library in ./apps/
- build process automated with make
- reference output in ./apps/refOutput/
- documentation of source code with doxygen
- "first contact" through README.md

(by doxygen)

C de::equation_t	Explicit nonlinear parameterized ordinary differential equation x'=g(x,p)
▼ C iteration_t< T >	
▼ C evolution_t< T >	Abstract base class for integration methods for differential equations
▼ C de::integrator_t< T >	Abstract base class for solvers of initial value problems
C de::explicitEuler_integrator_t< T >	Explicit Euler solver for initial value problems

- make test in ./apps/differentialEquation/
- run ./main.exe
- inspect output

  `x=0.60577`
- modify in main.cpp
  - initial state
  - parameter
  - target time
  - number of time steps

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  #include "iteration.hpp"
5  #include <vector>
6
7  /// abstract base class for integration methods for differential equations
8  template<typename T>
9  class evolution_t : public iteration_t<T> {
10   protected:
11     using iteration_t<T>::_states;
12     /// values of times due to time steps performed
13     std::vector<T> _times;
14     /// tracing mode
15     bool _trace;
16   public:
17     /// tracing mode set at construction
18     evolution_t(bool trace);
19     /// writes record of states visited at recorded times to file with given name
20     void plot(const std::string& filename) const;
21 };
22
23 #include "evolution.cpp"
```

Implementation: evolution.cpp

```
1  #include <fstream>
2  #include <cassert>
3
4  template<typename T>
5  evolution_t<T>::evolution_t(bool trace) : _trace(trace) {}
6
7  template<typename T>
8  void evolution_t<T>::plot(const std::string& filename) const {
9    std::ofstream ofs(filename);
10   assert(_states.size()==_times.size());
11   for (size_t k=0; k<_times.size(); ++k)
12     ofs << _times[k] << ' ' << _states[k] << std::endl;
13   ofs.close();
14 }
```

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  namespace de {
5
6    /// explicit nonlinear parameterized ordinary differential equation x'=g(x,p)
7    struct equation_t {
8      /// right-hand side to be provided by user
9      template <typename T>
10     static T g(const T& x, const T& p);
11   };
12
13 }
```

Implementation: differentialEquation/integrator.hpp


```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "cppNum/evolution.hpp"
4
5  namespace de {
6
7    /// abstract base class for solvers of initial value problems
8    template<typename T>
9    class integrator_t : public evolution_t<T> {
10     protected:
11       using evolution_t<T>::_trace;
12       /// target time
13       T _t_end;
14       /// number of time steps performed during integration
15       int _number_of_steps=1;
16     public:
17       /// constructor sets target time, number of times steps and tracing mode
18       integrator_t(const T& t_end, int number_of_steps, bool trace);
19       /// integrator allowing internal overwrites of x required by all specializations
20       virtual T run(T x, const T& p)=0;
21   };
22
23 }
24
25 #include "integrator.cpp"
```

```
1  namespace de {
2
3    template<typename T>
4    integrator_t<T>::integrator_t(const T& t_end, int number_of_steps, bool trace)
5      : evolution_t<T>(trace), _t_end(t_end), _number_of_steps(number_of_steps) {}
6
7  }
```

Implementation: differentialEquation/explicitEuler.hpp

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "integrator.hpp"
4  #include "equation.hpp"
5
6  namespace de {
7
8    /// Explicit Euler solver for initial value problems
9    template<typename T>
10   class explicitEuler_integrator_t : public integrator_t<T> {
11       using integrator_t<T>::_states; using integrator_t<T>::_times;
12       using integrator_t<T>::_t_end; using integrator_t<T>::_number_of_steps;
13       using integrator_t<T>::_trace;
14     public:
15       /// constructor sets target time, number of time steps, and tracing mode
16       explicitEuler_integrator_t(const T& t_end, int number_of_steps, bool trace=false);
17       /// integrator allows internal overwrites of x
18       T run(T x, const T& p);
19   };
20
21 }
22
23 #include "explicitEuler.cpp"
```

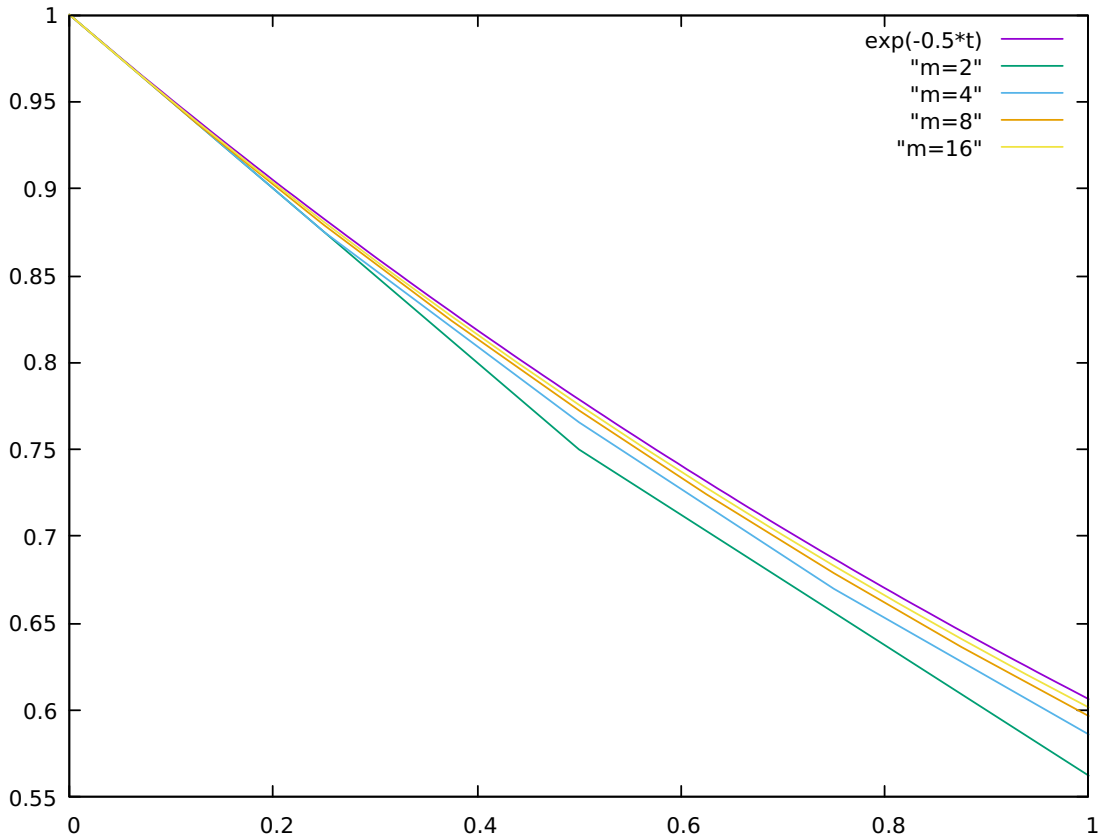
Implementation: differentialEquation/explicitEuler.cpp


```
1  namespace de {
2
3    template<typename T>
4    explicitEuler_integrator_t<T>::explicitEuler_integrator_t(const T& t_end, int number_of_steps, bool
         trace) : integrator_t<T>(t_end,number_of_steps,trace) {}
5
6    template<typename T>
7    T explicitEuler_integrator_t<T>::run(T x, const T& p) {
8      T t=0, dt=_t_end/_number_of_steps;
9      if (_trace) { _states.push_back(x); _times.push_back(t); }
10     for (int i=0;i<_number_of_steps;++i) {
11       t+=dt;
12       x+=dt*equation_t::g(x,p);
13       if (_trace) { _states.push_back(x); _times.push_back(t); }
14     }
15     return x;
16   }
17
18 }
```

```
1  // info@stce.rwth-aachen.de
2  #include "cppNum/differentialEquation/equation.hpp"
3
4  template<typename T>
5  T de::equation_t::g(const T &x, const T &p) {
6    return -p*x;
7  }
8
9  #include "cppNum/differentialEquation/explicitEuler.hpp"
10 #include <iostream>
11
12 int main() {
13   using T=double;
14   T p=0.5, x=1;
15   // enable tracing
16   de::explicitEuler_integrator_t<T> integrator(1,100,true);
17   // record trace
18   x=integrator.run(x,p);
19   // write trace to file
20   integrator.plot("explicitEuler_steps.plt");
21   std::cout << "x=" << x << std::endl;
22   return 0;
23 }
```

```
plot [t=0:5] exp(−0.5*t), "explicitEuler_steps.plt" with linespoints
```

```
0 1
0.01 0.995
0.02 0.990025
0.03 0.985075
0.04 0.98015
0.05 0.975249
...
0.96 0.618039
0.97 0.614949
0.98 0.611874
0.99 0.608815
1 0.60577
```



Parameter Sensitivities


```
1  // info@stce.rwth-aachen.de
2  #include "cppNum/differentialEquation/equation.hpp"
3
4  template<typename T>
5  T de::equation_t::g(const T &x, const T &p) { return -p*x; }
6
7  #include "cppNum/differentialEquation/explicitEuler.hpp"
8  #include "ad.hpp"
9  #include <iostream>
10
11 int main() {
12   using T=double;
13   ad::tangent_t<T> p=0.5, x_in=1; // derivatives initially equal to zero
14   de::explicitEuler_integrator_t<ad::tangent_t<T>> integrator(1,100);
15   ad::derivative(p)=1; // seed p
16   ad::tangent_t<T> x=integrator.run(x_in,p);
17   std::cout << "dx/dp=" << ad::derivative(x) << std::endl; // harvest dx/dp
18   ad::derivative(p)=0; ad::derivative(x_in)=1; // seed x_in
19   x=integrator.run(x_in,p);
20   std::cout << "dx/dx^0=" << ad::derivative(x) << std::endl; // harvest dx/dx_in
21   return 0;
22 }
```

- finite differences

```
dx/dp=-0.608815
dx/dx^0=0.60577
```

- ad::

```
dx/dp=-0.608815
dx/dx^0=0.60577
```

We look for local minimizers

$$x = \text{argmin}(f)$$

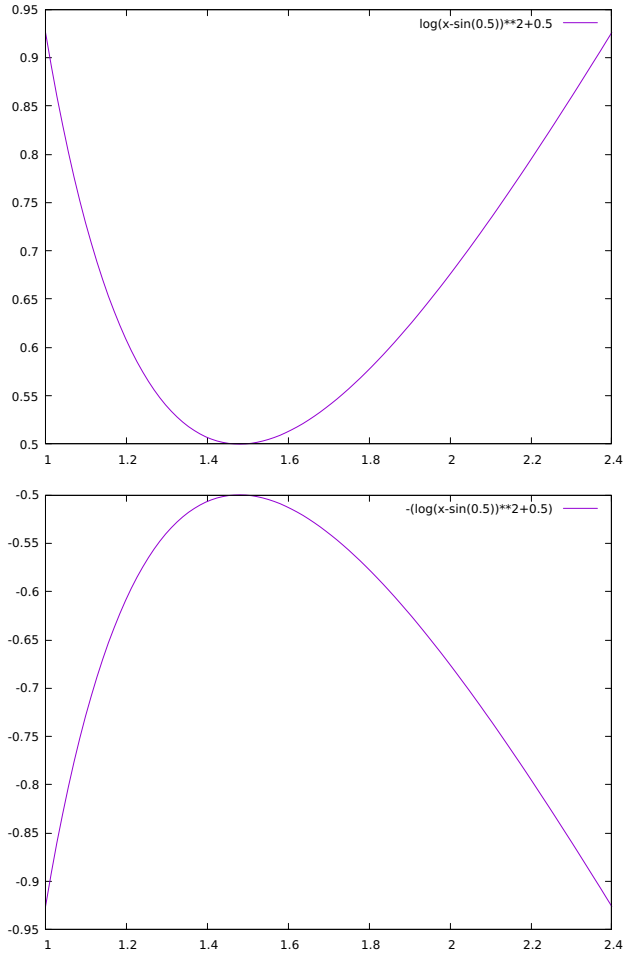
of parameterized convex objectives

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

with user-defined f.

Note

$$\text{argmin}(f) = \text{argmax}(-f)$$



cppNum v1.3 implements the gradient descent method for iterative approximation of the solution.

- user requirements
- illustration
- gradientDescent method
- use cases
- system requirements
- usage incl. essential C++
- design
- implementation incl. essential C++
- optional use cases (visualization, parameter sensitivity)

Users of the software want to minimize convex objectives including the ability to

- implement the objective $f(x, p)$
- apply the gradient descent method for computing a local minimizer x^*
- validate the second-order optimality condition
- add further local optimization methods

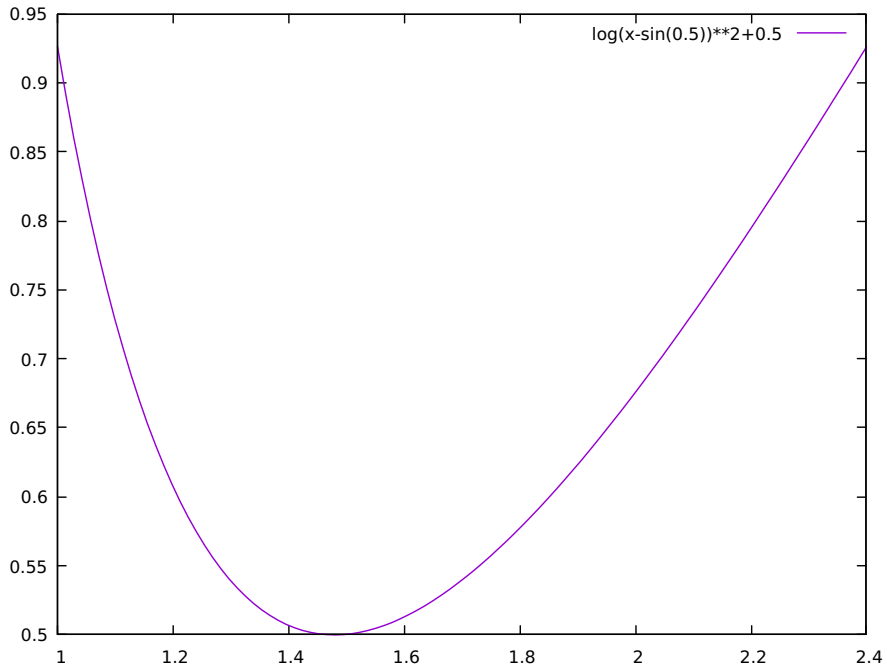
and, optionally, allowing

- visualization the individual steps performed during the iterative approximation of x^*
- first-order parameter sensitivity analysis using algorithmic differentiation by overloading with ad::
- second-order parameter sensitivity analysis with ad::

The convex objective

$$f(x, p) = \log(x - \sin(p))^2 + p$$

has its unique minimizer at $x = 1 + \sin(p)$.



The first-order optimality condition

$$f_x(x, p) = 0$$

implies

$$\begin{aligned} & 2 \cdot \log(x - \sin(p)) \cdot \frac{1}{x - \sin(p)} = 0 \\ \Rightarrow \quad & \log(x - \sin(p)) = 0 \\ \Rightarrow \quad & x - \sin(p) = 1 \\ \Rightarrow \quad & x = 1 + \sin(p) \end{aligned}$$

The second-order optimality condition

$$f_{xx}(x, p) > 0$$

holds implying convexity of the objective at $x = 1 + \sin(p)$. (Exercise: Verify!)

The gradient descent method iterates as

$$x = x - \alpha \cdot f_x(x, p)$$

while

$$|f_x(x, p)| > a$$

for accuracy $a > 0$. For a local minimum we require

$$f_{xx}(x, p) > 0$$

The optimal step length $\alpha > 0$ can be determined by line search. Heuristically we use bisection of α starting from $\alpha = 1$.

1. compute $x = \text{argmin}(f)$ with the gradient descent method

   ≪includes≫

   1.1 implement objective f

   1.2 run gradient descent method $x(f, x^0, p, a)$

   1.3 check second-order optimality condition $f_{xx}(x, p) > 0$

   - ≪extended by≫

     1.1 compute x_p

     1.2 compute x_{pp}

     1.3 validate vanishing x_{x^0}

     1.4 visualize iteration

The dynamics ($\rightarrow$ UML activity diagrams) are similar to v1.1.

Usage (Prototype)


```
1  // implement f
2  #include "apps/objectives/f1.hpp"
3  #include "cppNum/convexObjective/gradientDescent.hpp"
4  #include "cppNum/derivative.hpp"
5  #include <iostream>
6
7  int main() {
8    using T=double;
9    // set x,p,a
10   T p=0.5, x=1; co::gradientDescent_minimizer_t<T> minimizer(1e-7);
11   // run minimizer
12   x=minimizer.run(x,p);
13   // process result
14   std::cout << "x=" << x << ", f(x)=" << co::objective_t::f(x,p)
15   // validate optimality conditions
16     << ", dfdx=" << derivative_t::dfdx<co::objective_t>(x,p)
17     << ", ddfdxx=" << derivative_t::ddfdxx<co::objective_t>(x,p) << std::endl;
18   return 0;
19 }
```

Functional:

- gradient descent method $x = x(f, x^0, p, a)$ for minimizing the convex objective $f(x, p)$ for given parameter p, initial estimate x^0 for the minimizer and accuracy a including
  - implementation of f
- ability to implement alternative minimization methods
- evaluation of x_p and x_{x^0} by application of ad:: to the method
- optional record of iterative approximation $(x, f(x, p))$ and output to file for visualization with gnuplot

Non-functional: implementation in C++, documentation with doxygen

- co::minimizer_t ≪is a≫ approximation_t
- co::minimizer_t ≪uses≫ objective_t
- co::gradientDescent_minimizer_t ≪is a≫ minimizer_t
- co::gradientDescent_minimizer_t ≪uses≫ derivative_t
- application ≪owns≫ co::gradientDescent_minimizer_t
- application ≪uses≫ co::objective_t
- application ≪uses≫ derivative_t

The dynamics (→ e.g. UML activity diagrams) are similar to v1.1.

The full design (→ UML class diagram) includes member data and functions; see implementation.

For a twice differentiable program

$$y = f(x)$$

with second derivative $f_{xx} = f_{xx}(x)$ and given $x, \dot{x}, \tilde{x}$ the second-order tangent mode of algorithmic differentiation computes the second directional derivative

$$\dot{\tilde{y}} = \dot{x} \cdot f_{xx} \cdot \tilde{x}.$$

Seeding $\dot{x} = 1$ and $\tilde{x} = 1$ enables harvesting of $\dot{\tilde{y}} = f_{xx}$.

Example: $y = f(x) = \sin(x) \Rightarrow \dot{\tilde{y}} = \dot{x} \cdot f_{xx} \cdot \dot{x} = \dot{x} \cdot (-\sin(x)) \cdot \tilde{x}$.

The ad:: library implements second-order tangents as first-order tangents of first-order tangents by nesting of tangent types.

Essential ad::


```
1  // info@stce.rwth-aachen.de
2  #include <cmath>
3
4  template<typename T, typename PT>
5  T f(const T &x, const PT &p) {
6    using namespace std;
7    return exp(-x)-p;
8  }
9
10 #include "ad.hpp"
11 #include <iostream>
12
13 int main() {
14   using PT=double; using BT=ad::tangent_t<PT>;
15   PT p=0.5; ad::tangent_t<BT> x=1;
16   ad::derivative(ad::value(x))=1; // \tilde{x}
17   ad::value(ad::derivative(x))=1; // \dot{x}
18   ad::tangent_t<BT> y=f(x,p);
19   std::cout << "(f(x,p), df/dx(x,p), ddf/dxx(x,p))=(" << ad::value(ad::value(y)) << ", "
20     << ad::derivative(ad::value(y)) << ", " << ad::derivative(ad::derivative(y)) << ')' << std::endl;
21   return 0;
22 }
```

```
(f(x,p), df/dx(x,p), ddf/dxx(x,p))=(-0.132121, -0.367879, 0.367879)
```

Implementation: Structure of Source Code

```
cppNum
  convexObjective
    gradientDescent.cpp
    gradientDescent.hpp
    minimizer.cpp
    minimizer.hpp
    objective.hpp
  approximation.hpp
  derivative.hpp
  iteration.hpp
apps
  convexObjective
    main.cpp
    Makefile
      refOutput
        ref.out
  objectives
    f1.hpp
  Makefile.inc
Doxyfile
README.md
```

- cppNum library in ./cppNum/ subdirectory
- applications of cppNum library in ./apps/
- objectives in ./apps/objectives/
- build process automated with make
- reference output in ./apps/refOutput/
- documentation of source code with doxygen
- "first contact" through README.md

(by doxygen)

C derivative_t	Derivatives of f
▼ C iteration_t< T >	
▼ C approximation_t< T >	
▼ C co::minimizer_t< T >	Abstract base class for minimization methods
C co::gradientDescent_minimizer_t< T >	Gradient descent minimizer
C co::objective_t	Convex objective f(x,p)

- make test in ./apps/convexObjective/
- run ./main.exe
- inspect output

  ```
  x=1.47943, f(x)=0.5, dfdx=1.37668e-14, ddfdxx=2
  ```

- modify in main.cpp
  - initial state
  - parameter
  - accuracy

Implementation: derivative.hpp


```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "ad.hpp"
4
5  /// derivatives of f
6  struct derivative_t {
7    /// first derivative of f wrt. x
8    template<typename FUNCTION_T, typename T=double>
9    static T dfdx(const T& x, const T &p) {
10     ad::tangent_t<T> x_ad=x, p_ad=p;
11     ad::derivative(x_ad)=1;
12     ad::tangent_t<T> y_ad=FUNCTION_T::f(x_ad,p_ad);
13     return ad::derivative(y_ad);
14   }
15   /// second derivative of f wrt. x
16   template<typename FUNCTION_T, typename T=double>
17   static T ddfdxx(const T& x, const T &p) {
18     ad::tangent_t<T> x_ad=x, p_ad=p;
19     ad::derivative(x_ad)=1;
20     // second derivative computed as first derivative of first derivative
21     ad::tangent_t<T> dydx_ad=derivative_t::dfdx<FUNCTION_T,ad::tangent_t<T>>(x_ad,p_ad);
22     return ad::derivative(dydx_ad);
23   }
24
25 };
```

Implementation: convexObjective/objective.hpp


```
1  // info@stce.rwth-aachen.de
2  #pragma once
3
4  namespace co {
5
6    /// convex objective f(x,p)
7    struct objective_t {
8      /// user-defined implementation of f is required
9      template <typename T>
10     static T f(const T& x, const T& p);
11   };
12
13 }
```

Implementation: convexObjective/minimizer.hpp

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "cppNum/approximation.hpp"
4  #include <string>
5
6  namespace co {
7
8    /// abstract base class for minimization methods
9    template<typename T>
10   class minimizer_t : public approximation_t <T> {
11     protected:
12       using approximation_t<T>::_states;
13       using approximation_t<T>::_parameter;
14       /// tracing mode
15       bool _trace;
16     public:
17       /// constructor sets accuracy and tracing mode
18       minimizer_t(const T& accuracy, bool trace);
19       /// writes record of iterations performed by minimizer to file with given name
20       void plot(const std::string& filename) const;
21   };
22
23 }
24
25 #include "minimizer.cpp"
```

```
1  #include "objective.hpp"
2  #include <fstream>
3
4  namespace co {
5
6    template<typename T>
7    minimizer_t<T>::minimizer_t(const T& accuracy, bool trace) : approximation_t<T>(accuracy), _trace(
       trace) {}
8
9    template<typename T>
10   void minimizer_t<T>::plot(const std::string& filename) const {
11     std::ofstream ofs(filename);
12     for (const auto& state : _states)
13       ofs << state << ' ' << objective_t::f(state,_parameter) << std::endl;
14     ofs.close();
15   }
16
17 }
```

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "minimizer.hpp"
4
5  namespace co {
6
7    /// gradient descent minimizer
8    template<typename T>
9    class gradientDescent_minimizer_t : public minimizer_t<T> {
10       using minimizer_t<T>::_states;
11       using minimizer_t<T>::_parameter;
12       using minimizer_t<T>::_accuracy;
13       using minimizer_t<T>::_trace;
14     public:
15       /// constructor sets accuracy and tracing mode
16       gradientDescent_minimizer_t(const T& accuracy, bool trace=false);
17       /// runs the method for given x and p while avoiding side-effects due to internal overwrites of x
18       T run(T x, const T& p);
19   };
20
21 }
22
23 #include "gradientDescent.cpp"
```

Implementation: convexObjective/gradientDescent.cpp I

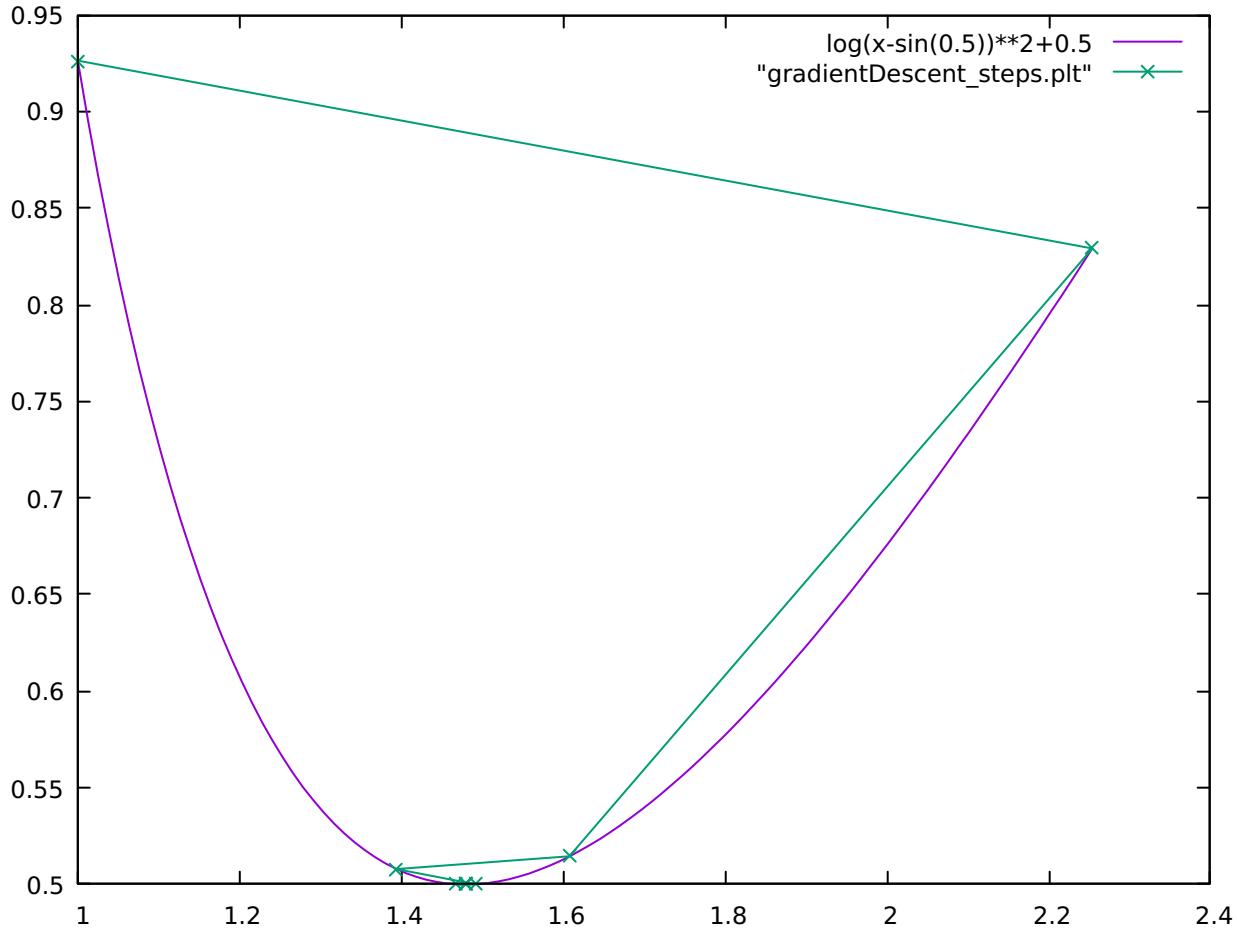
```
1  #include "objective.hpp"
2  #include "cppNum/derivative.hpp"
3
4  namespace co {
5
6    template<typename T>
7    gradientDescent_minimizer_t<T>::gradientDescent_minimizer_t(const T& accuracy, bool trace) :
       minimizer_t<T>(accuracy,trace) {}
8
9    template<typename T>
10   T gradientDescent_minimizer_t<T>::run(T x, const T& p) {
11     using namespace std; // enables potential overloading of fabs outside of std::
12     T y=objective_t::f(x,p), y_prev;
13     if (_trace) { // record trace
14       _states.push_back(x);
15       _parameter=p;
16     }
17     T dydx=derivative_t::dfdx<objective_t,T>(x,p);
18     do { // at least one descent step to establish dependence on x (and p)
19       y_prev=y;
20       double alpha=2.;
21       while (y_prev<=y&&alpha>_accuracy) { // line search
22         T x_trial=x;
23         alpha/=2.; // half-splitting
24         x_trial-=alpha*dydx;
```

```
25          y=objective_t::f(x_trial,p);
26        }
27        x-=alpha*dydx;
28        if (_trace) _states.push_back(x); // record trace
29        dydx=derivative_t::dfdx<objective_t,T>(x,p);
30      } while (fabs(dydx)>_accuracy);
31      return x;
32    }
33
34  }
```

```
1  // info@stce.rwth-aachen.de
2  #include "apps/objectives/f1.hpp"
3  #include "cppNum/convexObjective/gradientDescent.hpp"
4  #include "cppNum/derivative.hpp"
5  #include <iostream>
6
7  int main() {
8    using T=double;
9    T p=0.5, x=1;
10   // enable tracing
11   co::gradientDescent_minimizer_t<T> minimizer(1e-7,true);
12   // record trace
13   x=minimizer.run(x,p);
14   // write trace to file
15   minimizer.plot("gradientDescent_steps.plt");
16   std::cout << "x=" << x << "\nf(x)=" << co::objective_t::f(x,p)
17     << "\ndfdx=" << derivative_t::dfdx<co::objective_t>(x,p)
18     << "\nddfdxx=" << derivative_t::ddfdxx<co::objective_t>(x,p) << std::endl;
19   return 0;
20 }
```

```
plot log(x-sin(0.5))**2+0.5, "gradientDescent_steps.plt" with linespoints
```

```
1 0.926177
2.25404 0.828999
1.60761 0.514547
1.3938 0.508013
1.4917 0.500149
1.4676 0.500142
1.47964 0.5
1.47921 0.5
1.47943 0.5
1.47943 0.5
```



Parameter Sensitivities with ad::


```
1  // info@stce.rwth-aachen.de
2  #include "apps/objectives/f1.hpp"
3  #include "cppNum/convexObjective/gradientDescent.hpp"
4  #include "ad.hpp"
5  #include <iostream>
6
7  int main() {
8    using T=ad::tangent_t<ad::tangent_t<double>>;
9    T p=0.5, x0=1; // derivatives initially equal to zero
10   co::gradientDescent_minimizer_t<T> solver(1e-7);
11   ad::derivative(ad::value(p))=1; // \tilde{p}
12   ad::value(ad::derivative(p))=1; // \dot{p}
13   T x=solver.run(x0,p);
14   std::cout << "f(" << x << ")=" << co::objective_t::f(x,p) << std::endl;
15   std::cout << "dx/dp(" << x << ")=" << ad::derivative(x) << std::endl;
16   std::cout << "ddx/dpp(" << x << ")=" << ad::derivative(ad::derivative(x)) << std::endl;
17   ad::derivative(p)=0; // ad::value(ad::derivative(p))=ad::derivative(ad::derivative(p))=0
18   ad::derivative(x0)=1; // ad::value(ad::derivative(x0))=1; ad::derivative(ad::derivative(x0))=0
19   x=solver.run(x0,p);
20   std::cout << "dx/dx0(" << x << ")="
21     << ad::derivative(x) // ad::value(ad::derivative(x))
22     << std::endl;
23   return 0;
24 }
```

- finite differences

```
f(1.47943)=0.5
dx/dp(1.47943)=0.877583
ddx/dpp(1.47943)=-0.479426
dx/dx0(1.47943)=-2.57244e-12
```

- ad::

```
f(1.47943)=0.5
dx/dp(1.47943)=0.877583
ddx/dpp(1.47943)=-0.479426
dx/dx0(1.47943)=-1.88229e-12
```

We look for local minimizers

$$x = \text{argmin}(f)$$

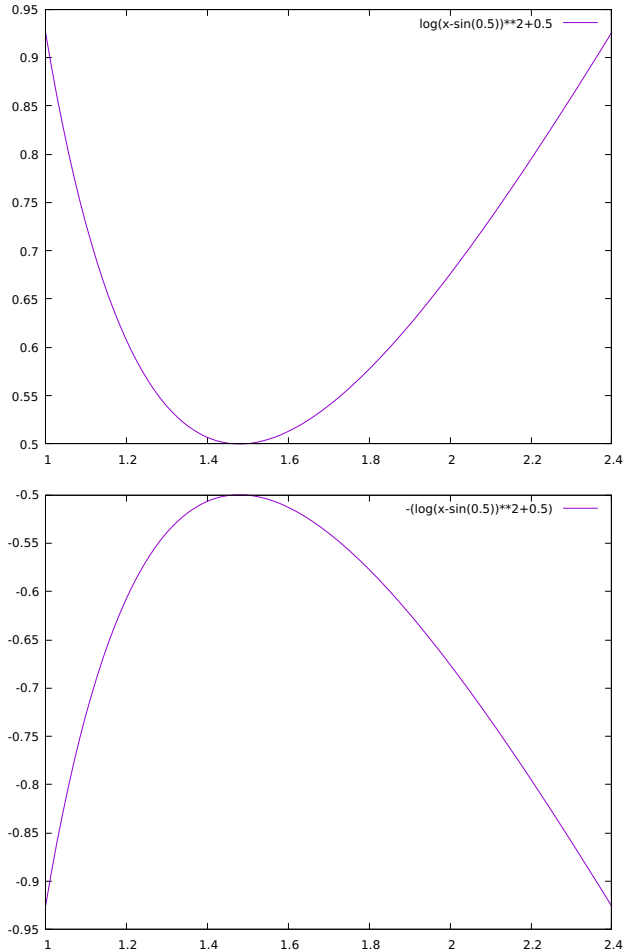
of parameterized convex objectives

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

with user-defined f.

Note

$$\text{argmin}(f) = \text{argmax}(-f)$$



cppNum v1.4 implements the Newton method for iterative approximation of the solution.

- user requirements
- illustration
- Newton method
- use cases
- system requirements
- usage incl. essential C++
- design
- implementation incl. essential C++
- optional use cases (visualization, parameter sensitivity)

Users of the software want to minimize convex objectives including the ability to

- implement the objective $f(x, p)$
- apply the Newton method for computing a local minimizer x^*
- validate the second-order optimality condition
- add further local optimization methods

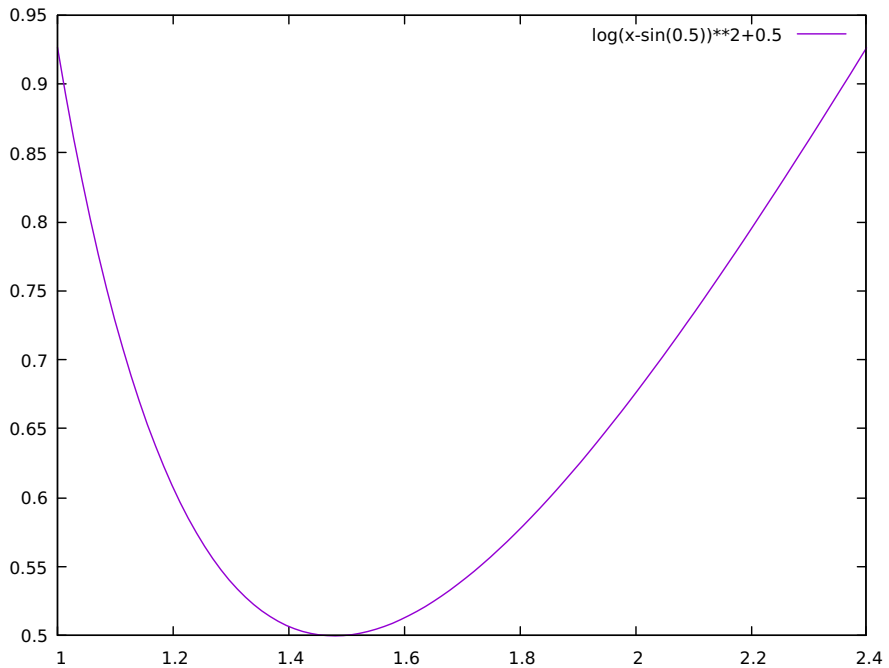
and, optionally, allowing

- visualization the individual steps performed during the iterative approximation of x^*
- first- and second-order parameter sensitivity analysis with ad::

The convex objective

$$f(x, p) = \log(x - \sin(p))^2 + p$$

has its unique minimizer at $x = 1 + \sin(p)$.



The Newton method iterates as

$$x = x - \frac{f(x,p)}{f_x(x,p)}$$

while

$$|f_x(x,p)| > a$$

for given accuracy $a > 0$. For a local minimum we require

$$f_{xx}(x,p) > 0$$

Conditions for convergence apply. Line search may help to meet them.

Newton's method for $f_x(x) = 0$ can be regarded as a fixed point iteration

$$x = g(x) = x - \frac{f_x(x)}{f_{xx}(x)} \, .$$

If at the solution

$$|g_x(x)| < 1 \, ,$$

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

The convergence rate of a fixed-point iteration grows linearly with decreasing values of $|g_x(x)|$.

For $|g_x(x)| = 0$ we get at least quadratic convergence; cubic for $|g_x(x)| = |g_{xx}(x)| = 0$ and so forth.

Newton's method becomes

$$x = g(x) = x - \frac{f_x(x)}{f_{xx}(x)}$$

yielding

$$g_x(x) = f_x(x) \cdot \frac{f_{xxx}(x)}{(f_{xx}(x))^2} \ .$$

At the solution $f_x(x) = 0$ implies $g_x(x) = 0$. Assuming a simple root ($f_x(x) = 0$, $f_{xx}(x) \neq 0$) the second derivative of g becomes equal to

$$g_{xx}(x) = f_{xx}(x) \cdot \frac{f_{xxx}(x)}{(f_{xx}(x))^2} + \underbrace{f_x(x)}_{=0} \cdot (\ldots)$$

implying quadratic convergence within the corresponding neighborhood of the solution if $f_{xxx}(x) \neq 0$ as well as convergence after a single iteration for quadratic f (linear f_x).

1. compute $x = \text{argmin}(f(x, p))$ with Newton method

   ≪includes≫

   1.1 implement objective f

   1.2 run Newton method $x(f, x^0, p, a)$

   1.3 check second-order optimality condition $f_{xx}(x, p) > 0$

   ▶ ≪extended by≫

   1.1 compute x_p

   1.2 compute x_{pp}

   1.3 validate vanishing x_{x^0}

   1.4 visualize iteration

The dynamics ($\rightarrow$ UML activity diagrams) are similar to v1.1.

Usage (Prototype)

```
1  // implement f
2  #include "apps/objectives/f1.hpp"
3  #include "cppNum/convexObjective/newton.hpp"
4  #include "cppNum/derivative.hpp"
5  #include <iostream>
6
7  int main() {
8    using T=double;
9    // set x,p,a
10   T p=0.5, x=1; co::newton_minimizer_t<T> minimizer(1e-7);
11   // run minimizer
12   x=minimizer.run(x,p);
13   // process results
14   std::cout << "x=" << x << "\nf(x)=" << co::objective_t::f(x,p)
15   // validate optimality conditions
16     << "\ndfdx=" << derivative_t::dfdx<co::objective_t>(x,p)
17     << "\nddfdxx=" << derivative_t::ddfdxx<co::objective_t>(x,p) << std::endl;
18   return 0;
19 }
```

Usage (Modified Algebraic Equation Solver)

```
1  // info@stce.rwth-aachen.de
2  #include "cppNum/algebraicEquation/equation.hpp"
3  #include <cmath>
4
5  template<typename T>
6  T ae::equation_t::f(const T &x, const T &p) {
7    using namespace std;
8    return exp(-x)-p;
9  }
10
11 #include "cppNum/algebraicEquation/newton.hpp"
12 #include <iostream>
13
14 int main() {
15   using T=double;
16   T p=0.5, x=0;
17   // default template argument ae::equation_t can be omitted
18   ae::newton_solver_t<T,ae::equation_t> solver(1e-7);
19   x=solver.run(x,p);
20   std::cout << "f(" << x << ")=" << ae::equation_t::f(x,p) << std::endl;
21   return 0;
22 }
```

Functional:

- Newton method $x = x(f, x^0, p, a)$ for minimizing the convex objective $f(x, p)$ for given parameter p, initial estimate x^0 and accuracy a including
  - implementation of f
- ability to implement alternative methods
- evaluation of x_p, x_{x^0} and x_{pp} by application of ad:: to the method
- optional record of iterative approximation $(x, f(x, p))$ and output to file for visualization with gnuplot

Non-functional: implementation in C++, documentation with doxygen

- co::objective_t
- co::minimizer_t ≪is a≫ approximation_t
- co::newton_minimizer_t ≪is a≫ co::minimizer_t
- application ≪owns≫ co::newton_minimizer_t

The dynamics (→ e.g. UML activity diagrams) are similar to v1.1.

The full design (→ UML class diagram) includes member data and functions; see implementation.

Important Detail

The Newton method for local optimization solves the algebraic system $f_x = 0$ to establish first-order optimality.

We make the algebraic solver accept residuals associated with variable types, that is, not just ae::equation_t::f but also co::newton_minimizer_t<T>::f implemented as

```
template<typename T>
 template<typename ae_T>
 ae_T newton_minimizer_t<T>::f(const ae_T &x, const ae_T& p) {
   return derivative_t::dfdx<objective_t,ae_T>(x,p);
 }
```

This functionality is enabled by making the equation type a template parameter:

```
template<typename T, typename EQUATION_T> ae::solver_t;
```

Implementation: Structure of Source Code

```
cppNum
  algebraicEquation // abridged
    newton.cpp newton.hpp
  convexObjective
    minimizer.cpp minimizer.hpp
    newton.cpp newton.hpp
    objective.hpp
  approximation.hpp
  derivative.hpp
  iteration.cpp iteration.hpp
apps
  convexObjective
    main.cpp
    Makefile
      refOutput
        ref.out
  objectives
    f1.hpp
  Makefile.inc
Doxyfile
README.md
```

- cppNum library in ./cppNum/ subdirectory
- system-generic algebraic equation solver in ./cppNum/algebraicEquation/
- applications of cppNum library in ./apps/
- objectives in ./apps/objectives/
- build process automated with make
- reference output in ./apps/refOutput/
- documentation of source code with doxygen
- "first contact" through README.md

(by doxygen)

C derivative_t	
C ae::equation_t	
▼ C iteration_t< T >	
▼ C approximation_t< T >	
C ae::solver_t< T, EQUATION_T >	
▼ C co::minimizer_t< T >	Base class for minimization methods
C co::gradientDescent_minimizer_t< T >	Gradient descent minimizer
C co::newton_minimizer_t< T >	Newton method
▼ C ae::solver_t< T, equation_t >	
C ae::newton_solver_t< T, EQUATION_T >	
C co::objective_t	Convex objective f(x,p)

- make test in ./apps/convexObjective/
- run ./main.exe
- inspect output

  ```
  x=1.47943, f(x)=0.5, dfdx=-3.90799e-14, ddfdxx=2
  ```

- modify in main.cpp
  - initial state
  - parameter
  - accuracy
- make test in ./apps/algebraicEquation/

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "minimizer.hpp"
4  #include "objective.hpp"
5
6  namespace co {
7    /// Newton method
8    template<typename T>
9    class newton_minimizer_t : public minimizer_t<T> {
10       using minimizer_t<T>::_states; using minimizer_t<T>::_parameter;
11       using minimizer_t<T>::_accuracy; using minimizer_t<T>::_trace;
12     public:
13       /// constructor sets accuracy and tracing mode
14       newton_minimizer_t(const T& accuracy, bool trace=false);
15       /// runs the method avoiding side effects due to internal overwrites of x
16       T run(T x, const T& p);
17       /// residual of first-order optimality condition (vanishing first derivative)
18       template <typename ae_T> static ae_T f(const ae_T&, const ae_T&);
19   };
20 }
21 #include "newton.cpp"
```

Implementation: convexObjective/newton.cpp

```
1  #include "cppNum/derivative.hpp"
2  #include "cppNum/algebraicEquation/newton.hpp"
3  namespace co {
4    template<typename T>
5    newton_minimizer_t<T>::newton_minimizer_t(const T& accuracy, bool trace) : minimizer_t<T>(
         accuracy,trace) {}
6
7    template<typename T>
8    template<typename ae_T>
9    ae_T newton_minimizer_t<T>::f(const ae_T &x, const ae_T& p) {
10     return derivative_t::dfdx<objective_t,ae_T>(x,p);
11   }
12
13   template<typename T>
14   T newton_minimizer_t<T>::run(T x, const T &p) {
15     ae::newton_solver_t<T,newton_minimizer_t<T>> ae_solver(_accuracy,_trace);
16     x=ae_solver.run(x,p);
17     if (_trace) { _states=ae_solver.get_states(); _parameter=p; }
18     return x;
19   }
20 }
```

```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "cppNum/approximation.hpp"
4  #include "equation.hpp"
5  #include <string>
6
7  namespace ae {
8    // abstract base class for solvers of generic algebraic equations
9    template<typename T, typename EQUATION_T=equation_t>
10   class solver_t : public approximation_t<T> {
11     protected:
12       using approximation_t<T>::_states; using approximation_t<T>::_parameter;
13     public:
14       solver_t(const T& accuracy, bool trace);
15       virtual T run(T x, const T& p)=0;
16       const std::vector<T>& get_states() const;
17       void plot(const std::string& filename) const;
18   };
19 }
20
21 #include "solver.cpp"
```

```
1  #include <fstream>
2
3  namespace ae {
4
5    template<typename T, typename EQUATION_T>
6    solver_t<T,EQUATION_T>::solver_t(const T& accuracy, bool trace)
7      : approximation_t<T>(accuracy, trace) {}
8
9    template<typename T, typename EQUATION_T>
10   const std::vector<T>& solver_t<T,EQUATION_T>::get_states() const { return _states; }
11
12   template<typename T, typename EQUATION_T>
13   void solver_t<T,EQUATION_T>::plot(const std::string& filename) const {
14     std::ofstream ofs(filename);
15     for (const auto& state : _states)
16       ofs << state << " 0\n" << state << ' '
17           << EQUATION_T::f(state,_parameter) << std::endl;
18     ofs.close();
19   }
20
21 }
```

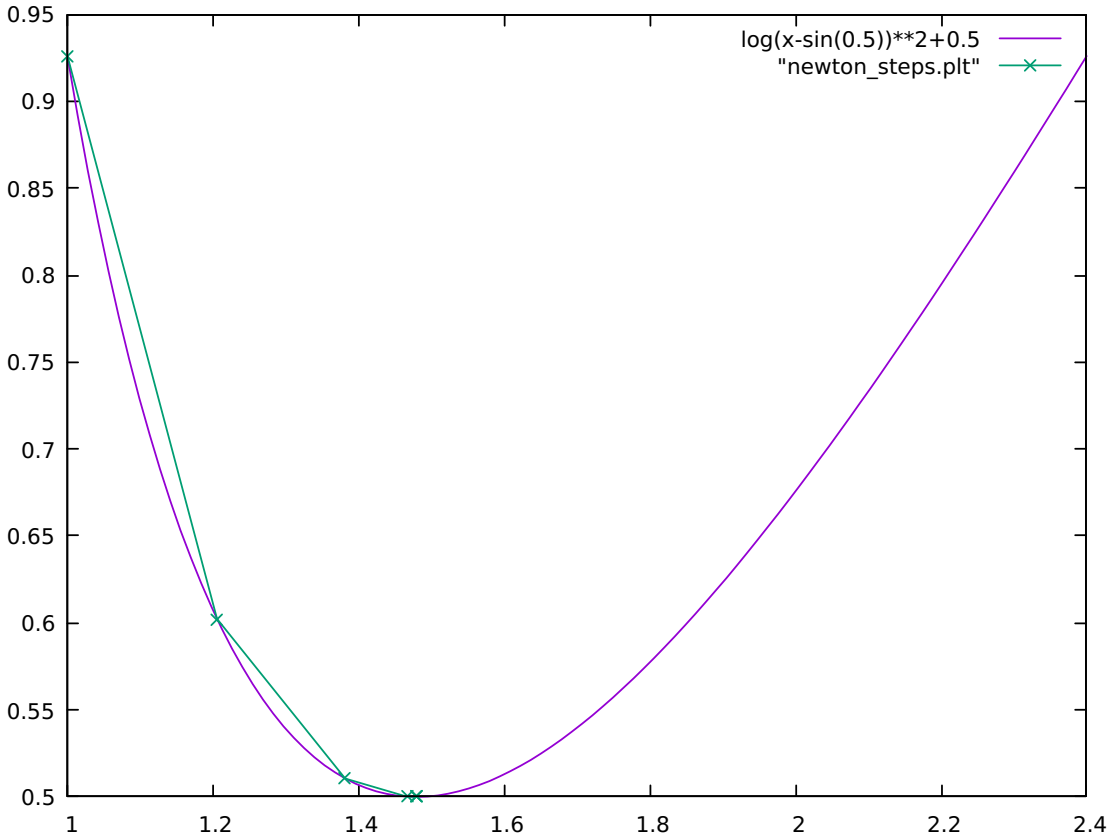
```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "solver.hpp"
4  #include "equation.hpp"
5
6  namespace ae {
7    template<typename T, typename EQUATION_T=equation_t>
8    class newton_solver_t : public solver_t<T, EQUATION_T> {
9        using solver_t<T, EQUATION_T>::_states;
10       using solver_t<T, EQUATION_T>::_parameter;
11       using solver_t<T, EQUATION_T>::_accuracy;
12       using solver_t<T, EQUATION_T>::_trace;
13     public:
14       newton_solver_t(const T& accuracy, bool trace=false);
15       T run(T x, const T& p);
16   };
17 }
18 #include "newton.cpp"
```

```
1  #include "cppNum/derivative.hpp"
2
3  namespace ae {
4    template<typename T, typename EQUATION_T>
5    newton_solver_t<T,EQUATION_T>::newton_solver_t(const T& accuracy, bool trace)
6      : solver_t<T, EQUATION_T>(accuracy,trace) {}
7
8    template<typename T, typename EQUATION_T>
9    T newton_solver_t<T,EQUATION_T>::run(T x, const T &p) {
10     using namespace std;
11     if (_trace) { _states.push_back(x); _parameter=p; }
12     T residual=EQUATION_T::f(x,p);
13     do {
14       x-=residual/derivative_t::dfdx<EQUATION_T,T>(x,p);
15       if (_trace) _states.push_back(x);
16       residual=EQUATION_T::f(x,p);
17     } while (fabs(residual)>_accuracy);
18     return x;
19   }
20 }
```

```
1  // info@stce.rwth-aachen.de
2  #include "apps/objectives/f1.hpp"
3  #include "cppNum/convexObjective/newton.hpp"
4  #include "cppNum/derivative.hpp"
5  #include <iostream>
6
7  int main() {
8    using T=double;
9    T p=0.5, x=1;
10   // enable tracing
11   co::newton_minimizer_t<T> minimizer(1e-7,true);
12   // record trace
13   x=minimizer.run(x,p);
14   // write trace to file
15   minimizer.plot("newton_steps.plt");
16   std::cout << "x=" << x << ", f(x)=" << co::objective_t::f(x,p)
17     << ", dfdx=" << derivative_t::dfdx<co::objective_t>(x,p)
18     << ", ddfdxx=" << derivative_t::ddfdxx<co::objective_t>(x,p) << std::endl;
19   return 0;
20 }
```

```
plot [1:2.4] log(x-sin(0.5))**2+0.5, "newton_steps.plt" with linespoints
```

```
1 0.926177
1.20561 0.602366
1.38164 0.51059
1.46582 0.500188
1.47915 0.5
1.47943 0.5
1.47943 0.5
```



Parameter Sensitivities with ad::

```
1  // info@stce.rwth-aachen.de
2  #include "apps/objectives/f1.hpp"
3  #include "cppNum/convexObjective/newton.hpp"
4  #include "ad.hpp"
5  #include <iostream>
6
7  int main() {
8    using T=ad::tangent_t<ad::tangent_t<double>>;
9    T p=0.5, x0=1; // derivatives initially equal to zero
10   co::newton_minimizer_t<T> solver(1e-7);
11   ad::derivative(ad::value(p))=1; // \tilde{p}
12   ad::value(ad::derivative(p))=1; // \dot{p}
13   T x=solver.run(x0,p);
14   std::cout << "f(" << x << ")=" << co::objective_t::f(x,p) << std::endl;
15   std::cout << "dx/dp(" << x << ")=" << ad::derivative(x) << std::endl;
16   std::cout << "ddx/dpp(" << x << ")=" << ad::derivative(ad::derivative(x)) << std::endl;
17   ad::derivative(p)=0; // ad::value(ad::derivative(p))=ad::derivative(ad::derivative(p))=0
18   ad::derivative(x0)=1; // ad::value(ad::derivative(x0))=1; ad::derivative(ad::derivative(x0))=0
19   x=solver.run(x0,p);
20   std::cout << "dx/dx0(" << x << ")="
21     << ad::derivative(x) // ad::value(ad::derivative(x))
22     << std::endl;
23   return 0;
24 }
```

- finite differences

```
f(1.47943)=0.5
dx/dp(1.47943)=0.877583
ddx/dpp(1.47943)=-0.479426
dx/dx0(1.47943)=2.57244e-12
```

- ad::

```
f(1.47943)=0.5
dx/dp(1.47943)=0.877583
ddx/dpp(1.47943)=-0.479426
dx/dx0(1.47943)=2.12893e-12
```

We consider initial value problems for explicit ordinary differential equations yielding solutions $x(p, t)$ depending on a parameter p and time t.

For given $x^0(p) = x(p, 0)$ and target time $t_{\text{end}} > 0$ the solution $x(p, t_{\text{end}})$ is obtained by integration of the explicit ordinary differential equation

$$x_t = g(x, p)$$

with user-defined g.

cppNum v1.5 implements the implicit (also: backward) Euler integration method for the iterative approximation of the solution.

- user requirements
- illustration
- implicit Euler method
- use cases
- system requirements
- usage incl. essential C++
- design
- implementation incl. essential C++
- optional use cases (visualization, parameter sensitivity)

Users of the software want to solve initial value problems including the ability to

- implement $g(x, p)$
- apply the implicit Euler method $x(g, x^0, p, t_{\text{end}})$ for computing $x(p, t_{\text{end}})$

and, optionally, allowing

- visualization the individual steps performed during the iterative integration of the solution $x(p, t_{\text{end}})$
- parameter sensitivity analysis with ad::

The implicit Euler method replaces the time derivative x_t in

$$x_t = g(x(t), p)$$

with a backward finite difference yielding

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

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

$$x(t) - x(t - \Delta t) - \Delta t \cdot g(x(t), p) = 0 \ ,$$

e.g. using the Newton method with user-supplied accuracy $a \in \boldsymbol{R}$..

The solution is approximated iteratively for given $x(0) = x^0$ and $\Delta t > 0$. The algebraic equation is solved for each time step.

1. compute $x(t_{\text{end}}) = x(g, x^0, p, t_{\text{end}}) : \ x_t = g(x, p), \ x(0) = x^0$ with implicit Euler method

   ≪includes≫

   1.1 implement g

   1.2 run implicit Euler method $x(g, x^0, p, t_{\text{end}})$

   ≪extended by≫

   1.1 compute x_p

   1.2 compute x_{x^0}

   1.3 visualize evolution

The dynamics ($\rightarrow$ UML activity diagrams) are similar to v1.1.

Usage (Prototype)

```
1  // implement g
2  #include "apps/equations/g1.hpp"
3  #include "cppNum/differentialEquation/implicitEuler.hpp"
4  #include <iostream>
5
6  int main() {
7    using T=double;
8    // set x,p,t_end(=1), dt=t_end/m (m=100), accuracy=1e-8
9    de::implicitEuler_integrator_t<T> integrator(1,100,1e-8);
10   T p=0.5, x=1;
11   // run integrator
12   x=integrator.run(x,p);
13   // process result
14   std::cout << "x=" << x << std::endl;
15   return 0;
16 }
```

Usage (Modified Algebraic Equation Solver)


```
1  // info@stce.rwth-aachen.de
2  #include "cppNum/algebraicEquation/equation.hpp"
3  #include <vector>
4  #include <cmath>
5
6  template<typename T, typename DATA_T>
7  T ae::equation_t::f(const T &x, const T &p, const DATA_T* const data) {
8    using namespace std;
9    return data->i*exp(-x)-p;
10 }
11
12 #include "cppNum/algebraicEquation/newton.hpp"
13 #include <iostream>
14
15 int main() {
16   using T=double;
17   T x=0, p=0.5;
18   struct data_t { int i=1; } data;
19   // default template argument ae::equation_t can be omitted
20   ae::newton_solver_t<T,data_t,ae::equation_t> solver(1e-7,&data);
21   x=solver.run(x,p);
22   std::cout << "f(" << x << ")=" << ae::equation_t::f(x,p,&data) << std::endl;
23   return 0;
24 }
```

Functional:

- implicit Euler method $x = x(g, x^0, p, t_{\text{end}}, m, a)$ for solving the initial value problem $x' = g(x, p, t)$, $x(p, 0) = x^0$ for given parameter p and target time t_{end} reached by performing m time steps including
  - specification of accuracy a of Newton solver
  - implementation of g
- ability to implement alternative methods
- evaluation of x_p and x_{x^0} by application of ad:: to the method
- optional record of evolution $(t, x(p, t))$ and output to file for visualization with gnuplot

Non-functional: implementation in C++, documentation with doxygen

- evolution_t ≪is a≫ iteration_t
- de::integrator_t ≪is a≫ evolution_t
- de::integrator_t ≪uses≫ de::equation_t
- de::implicitEuler_integrator_t ≪is a≫ de::integrator_t
- application ≪owns≫ de::implicitEuler_integrator_t

The dynamics (→ e.g. UML activity diagrams) are similar to v1.1.

The full design includes member data and functions; see implementation.

Important Detail

The implicit Euler method solves the algebraic equation

$$f(x^i, p) \equiv x^i - x^{i-1} - \Delta t \cdot g(x^i, p) = 0$$

at each time step.

The two arguments x^{i-1} and Δt are required in addition to x^i and p.

We make the algebraic equations and corresponding solvers accept pointers to arbitrary auxiliary data of variable type DATA_T, e.g.

```
template<typename T, typename DATA_T, typename EQUATION_T> solver_t;
```

Implementation: Structure of Source Code


```
cppNum
  algebraicEquation // abridged
    newton.cpp newton.hpp
  differentialEquation
    equation.hpp
    implicitEuler.cpp implicitEuler.hpp
    integrator.hpp
  approximation.hpp
  derivative.hpp
  evolution.hpp
  iteration.hpp
apps
  differentialEquation
    main.cpp
    Makefile
      refOutput
        ref.out
  equations
    g1.hpp
  Makefile.inc
Doxyfile
README.md
```

- cppNum library in ./cppNum/ subdirectory
- system-generic algebraic equation solver in ./cppNum/algebraicEquation/
- applications of cppNum library in ./apps/
- differential equations in ./apps/objectives/
- build process automated with make
- reference output in ./apps/refOutput/
- documentation of source code with doxygen
- "first contact" through README.md

(by doxygen)

C derivative_t	
C de::equation_t	
C ae::equation_t	
▼ C iteration_t< T >	
▼ C approximation_t< T >	
C ae::solver_t< T, DATA_T, EQUATION_T >	Abstract base for solvers of algebraic equations
▼ C ae::solver_t< T, void, equation_t >	
C ae::newton_solver_t< T, DATA_T, EQUATION_T >	Newton method for solving algebraic equation f(x,p,aux_data)=0
▼ C evolution_t< T >	
▼ C de::integrator_t< T >	
C de::implicitEuler_integrator_t< T >	Implicit Euler method for solving the initial value problem x'=g(x,p); x(0)=x0

- make test in ./apps/differentialEquation/
- run ./main.exe
- inspect output

  `x=0.607287`
- modify in main.cpp
  - initial state
  - parameter
  - target time
  - number of time steps
  - accuracy of Newton solver
- make test in ./apps/algebraicEquation/

Implementation: differentialEquation/implicitEuler.hpp


```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "integrator.hpp"
4
5  namespace de {
6    /// implicit Euler method for solving the initial value problem x'=g(x,p); x(0)=x0
7    template<typename T>
8    class implicitEuler_integrator_t : public integrator_t<T> {
9        using integrator_t<T>::_states; using integrator_t<T>::_times;
10       using integrator_t<T>::_t_end; using integrator_t<T>::_number_of_steps;
11       float _accuracy;
12       bool _trace;
13     public:
14       /// constructor sets target time, number of time steps and tracing mode
15       implicitEuler_integrator_t(const T& t_end, int number_of_steps, float accuracy=1e-7, bool trace=
             false);
16       /// implicit Euler integrator
17       T run(T x, const T& p);
18       /// algebraic equation solved in each time step requires additional data
19       template <typename ae_T, typename DATA_T>
20       static ae_T f(const ae_T&, const ae_T&, const DATA_T* const);
21   };
22 }
23 #include "implicitEuler.cpp"
```

Implementation: differentialEquation/implicitEuler.cpp I

```
1  #include "cppNum/algebraicEquation/newton.hpp"
2  #include "cppNum/algebraicEquation/equation.hpp"
3
4  #include <iostream>
5  #include <cmath>
6  #include <limits>
7
8  namespace de {
9
10   template<typename T>
11   implicitEuler_integrator_t<T>::implicitEuler_integrator_t(const T& t_end, int number_of_steps, float
         accuracy, bool trace) : integrator_t<T>(t_end,number_of_steps), _accuracy(accuracy), _trace(trace)
         {}
12
13   template<typename T>
14   template<typename ae_T, typename DATA_T>
15   ae_T implicitEuler_integrator_t<T>::f(const ae_T &x, const ae_T& p, const DATA_T* const data_p) {
16     return x-data_p->x_prev-data_p->dt*equation_t::g(x,p);
17   }
18
19   template<typename T>
20   T implicitEuler_integrator_t<T>::run(T x, const T &p) {
21     T t=0, dt=_t_end/_number_of_steps;
22     if (_trace) { _states.push_back(x); _times.push_back(t); }
23     struct data_t { T x_prev,dt; } data;
```

```
24      ae::newton_solver_t<T,data_t,implicitEuler_integrator_t<T>> ae_solver(_accuracy,&data);
25      data.dt=dt;
26      do {
27        t+=dt;
28        data.x_prev=x;
29        x=ae_solver.run(x,p);
30        if (_trace) { _states.push_back(x); _times.push_back(t); }
31      } while (t<_t_end);
32      return x;
33    }
34
35  }
```

Implementation: algebraicEquation/solver.hpp


```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "cppNum/approximation.hpp"
4  #include <string>
5
6  namespace ae {
7    /// abstract base for solvers of algebraic equations
8    template<typename T, typename DATA_T, typename EQUATION_T>
9    class solver_t : public approximation_t<T> {
10     protected:
11       using approximation_t<T>::_parameter; using approximation_t<T>::_states;
12       /// tracing on/off
13       bool _trace;
14       /// pointer to auxiliary data
15       const DATA_T* const _data_p;
16     public:
17       /// accuracy, tracing mode and pointer to auxiliary data set at time of construction
18       solver_t(const T& accuracy, bool trace, const DATA_T* const data_p);
19       /// solution method allowing side-effect-free overwrites of x required by all specializations
20       virtual T run(T x, const T& p)=0;
21       /// read-only access to traced states
22       const std::vector<T>& get_states() const;
23       /// writes trace to file
24       void plot(const std::string&) const;
25   };
26 }
27 #include "solver.cpp"
```

```
1  #include <fstream>
2
3  namespace ae {
4
5    template<typename T, typename DATA_T, typename EQUATION_T>
6    solver_t<T,DATA_T,EQUATION_T>::solver_t(const T& accuracy, bool trace, const DATA_T* const
         data_p)
7      : approximation_t<T>(accuracy), _trace(trace), _data_p(data_p) {}
8
9    template<typename T, typename DATA_T, typename EQUATION_T>
10   void solver_t<T,DATA_T,EQUATION_T>::plot(const std::string& fname) const {
11     std::ofstream ofs(fname);
12     for (const auto& state : _states)
13       ofs << state << " 0\n" << state << ' '
14           << EQUATION_T::f(state,_parameter,_data_p) << std::endl;
15     ofs.close();
16   }
17
18 }
```

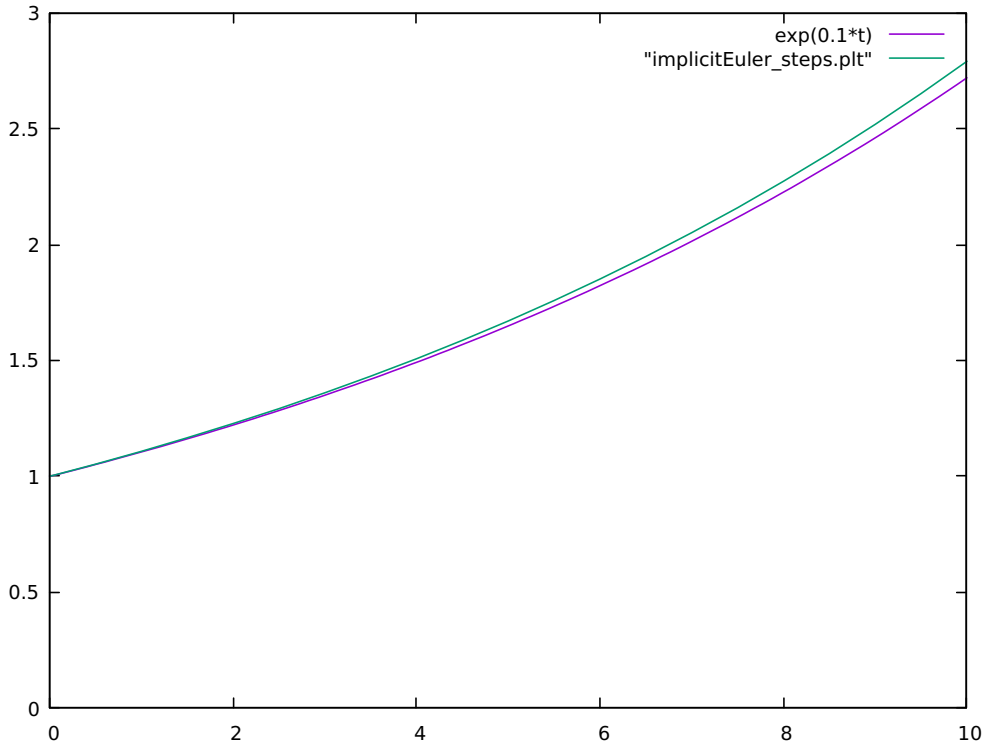
```
1  // info@stce.rwth-aachen.de
2  #pragma once
3  #include "solver.hpp"
4  #include "equation.hpp"
5
6  namespace ae {
7
8    /// Newton method for solving algebraic equation f(x,p,aux_data)=0
9    template<typename T, typename DATA_T=void, typename EQUATION_T=equation_t>
10   class newton_solver_t : public solver_t<T,DATA_T,EQUATION_T> {
11       using solver_t<T,DATA_T,EQUATION_T>::_accuracy;
12       using solver_t<T,DATA_T,EQUATION_T>::_parameter;
13       using solver_t<T,DATA_T,EQUATION_T>::_states;
14       using solver_t<T,DATA_T,EQUATION_T>::_trace;
15       using solver_t<T,DATA_T,EQUATION_T>::_data_p;
16     public:
17       /// accuracy, auxiliary data and tracing mode set at time of construction
18       newton_solver_t(const T& accuracy, const DATA_T* const data_p=nullptr, bool trace=false);
19       /// Newton method allowing side-effect-free overwrites of x
20       T run(T x, const T& p);
21   };
22
23 }
24 #include "newton.cpp"
```

```
1  #include "cppNum/derivative.hpp"
2
3  namespace ae {
4
5    template<typename T, typename DATA_T, typename EQUATION_T>
6    newton_solver_t<T,DATA_T,EQUATION_T>::newton_solver_t(const T& accuracy, const DATA_T*
       const data_p, bool trace) : solver_t<T,DATA_T,EQUATION_T>(accuracy,trace,data_p) {}
7
8    template<typename T, typename DATA_T, typename EQUATION_T>
9    T newton_solver_t<T,DATA_T,EQUATION_T>::run(T x, const T &p) {
10     using namespace std;
11     T residual=EQUATION_T::f(x,p,_data_p);
12     if (_trace) { _parameter=p; _states.push_back(x); }
13     do {
14       x-=residual/derivative_t::dfdx<EQUATION_T,T,DATA_T>(x,p,_data_p);
15       residual=EQUATION_T::f(x,p,_data_p);
16       if (_trace) _states.push_back(x);
17     } while (fabs(residual)>_accuracy);
18     return x;
19   }
20
21 }
```

```
1  // info@stce.rwth-aachen.de
2  #include "apps/equations/g1.hpp"
3  #include "cppNum/differentialEquation/implicitEuler.hpp"
4  #include <vector>
5  #include <iostream>
6
7  int main() {
8    using T=double;
9    T p=0.5, x=1, t_target=1; int number_of_timesteps=100;
10   // enable tracing
11   bool trace=true;
12   de::implicitEuler_integrator_t<T> integrator(t_target,number_of_timesteps,trace);
13   // record trace
14   x=integrator.run(x,p);
15   // write trace to file
16   integrator.plot("implicitEuler_steps.plt");
17   std::cout << "x=" << x << std::endl;
18   return 0;
19 }
```

```
plot [t=0:10][0:3] exp(0.1*t), "implicitEuler_steps.plt" with lines
```

```
0 1
0.5 1.05263
1 1.10803
1.5 1.16635
2 1.22774
2.5 1.29236
3 1.36037
3.5 1.43197
...
9 2.51753
9.5 2.65003
10 2.78951
```



Parameter Sensitivities with ad::

```
1  // info@stce.rwth-aachen.de
2  #include "apps/equations/g1.hpp"
3  #include "cppNum/differentialEquation/implicitEuler.hpp"
4  #include "ad.hpp"
5  #include <iostream>
6
7  int main() {
8    using T=ad::tangent_t<double>;
9    T p=0.5, x0=1, t_target=1; // derivatives initially equal to zero
10   int number_of_timesteps=100;
11   de::implicitEuler_integrator_t<T> integrator(t_target,number_of_timesteps);
12   ad::derivative(p)=1; // seed p
13   T x=integrator.run(x0,p);
14   std::cout << "dx/dp=" << ad::derivative(x) << std::endl; // harvest dx/dp
15   ad::derivative(p)=0; ad::derivative(x0)=1; // seed x0
16   x=integrator.run(x0,p);
17   std::cout << "dx/dx^0=" << ad::derivative(x) << std::endl; // harvest dx/dx0
18   return 0;
19 }
```

- finite differences

```
dx/dp=-0.604265
dx/dx^0=0.607287
```

- ad::

```
dx/dp=-0.604265
dx/dx^0=0.607287
```

All problems are parameterized, sufficiently often differentiable wrt. both x and p.

The corresponding functions

$$F : \boldsymbol{R}^{nx} \times \boldsymbol{R}^{np} \to \boldsymbol{R}^{m} : \quad y = F(x, p)$$

are implemented as differentiable C++ programs.

Derivatives are denoted as

$$F_{x^k} = y_{x^k} \equiv \frac{d^k F}{dx^k}(x, p); \quad F_{p^k} = y_{p^k} \equiv \frac{d^k F}{dp^k}(x, p) \; .$$

We also write $F_{xx} = F_{x^2}$, $F_{xxx} = F_{x^3}$ and so forth.

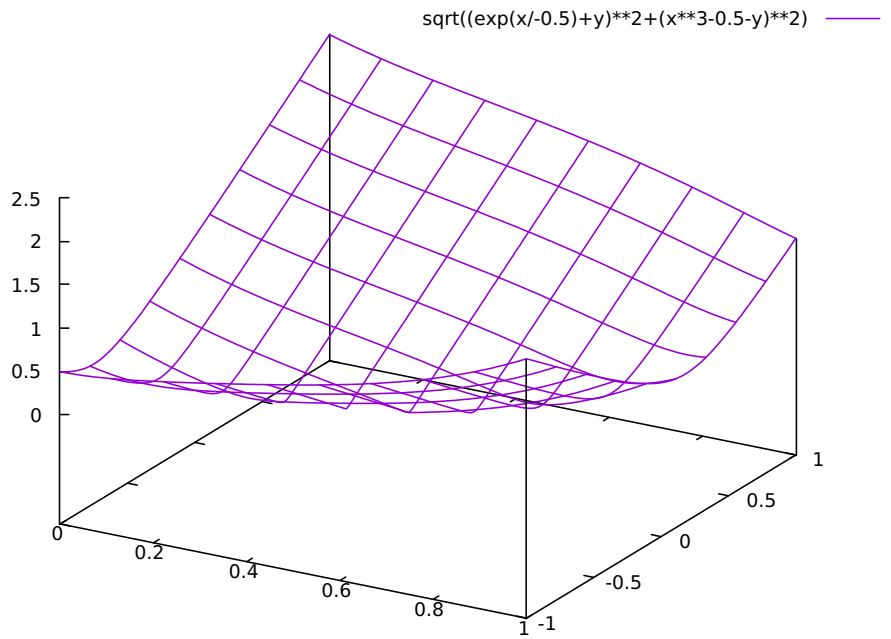
We look for roots

$$x = x(p): \quad F(x,p) = 0$$

of user-defined differentiable programs

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

with $F : \boldsymbol{R}^{nx} \times \boldsymbol{R}^{np} \to \boldsymbol{R}^{nx}$.



Example:

$$e^{\frac{x_0}{-0.5}} + x_1 = 0$$
$$x_0^3 - 0.5 - x_1 = 0\ .$$

- Newton method
- usage
- demo
- discussion of the source code
- case study
  - visualization
  - parameter sensitivity

The Newton method $x^k = x^k(F, x^0, p, a)$ performs k iterations

$$x^{i+1} = x^i - \Delta x^{i+1} = x^i - F_x(x^i, p)^{-1} \cdot F(x^i, p), \quad i = 0, \ldots, k-1$$

to approximate a root $x = x(p) : \ F(x, p) = 0$ for given initial estimate x^0 for the solution and p.

The number of iterations $k = k(a)$ is determined by the desired accuracy $a > 0$ ensuring $\|F(x^k, p)\| < a$.

$\|.\|$ denotes a suitable vector norm, e.g. the L_2-norm.

Conditions for convergence apply.

```
1  #include "intersection.hpp"
2  #include "cppNum/algebraicSystem/newton.hpp"
3  #include <iostream>
4
5  int main() {
6    using T=double;
7    using namespace std;
8    const int np=2, nx=2;
9    la::vector_t<T> p(np), x(nx);
10   p(0)=1; p(1)=-0.5; x(0)=1; x(1)=1;
11   as::newton_solver_t<T> solver(1e-7);
12   x=solver.run(x,p);
13   cout << "x=" << x.transpose() << endl;
14   return 0;
15 }
```

- make test in ./apps/intersection
- run ./main.exe
- inspect output

  ```
  x= 0.553031 -0.330859
  ```
- modify in main.cpp
  - initial state
  - parameters
  - accuracy

Class Hierarchy (by doxygen):

- derivative_t
- iteration_t< T >
  - approximation_t< T >
    - as::solver_t< T >
      - as::newton_solver_t< T, LINEAR_SOLVER_T >
- la::llt_solver_t< T >
- la::lu_solver_t< T >
- as::system_t

- Wrappers for Eigen[8] vectors and matrices are provided in linearAlgebra.hpp implying access to the corresponding rich set of functionalities, see Eigen documentation for details. Direct solvers for systems of linear equations based on LU and LL^T factorizations are provided.
- Fundamentals for tracing of states visited during the iteration are provided in iteration.hpp.
- Approximation methods are defined as iterations to be terminated after reaching a given accuracy.
- Functions for computing first derivatives of F wrt. both x and p can be found in derivatives.hpp.
- The signature of the residual F to be supplied by the user is fixed in algebraicSystem/system.hpp.
- An abstract base for solution methods is defined in algebraicSystem/solver.hpp.
- The Newton method is implemented in algebraicSystem/newton.hpp.

[8] `eigen.tuxfamily.org`

We look for the point of intersection of the two functions

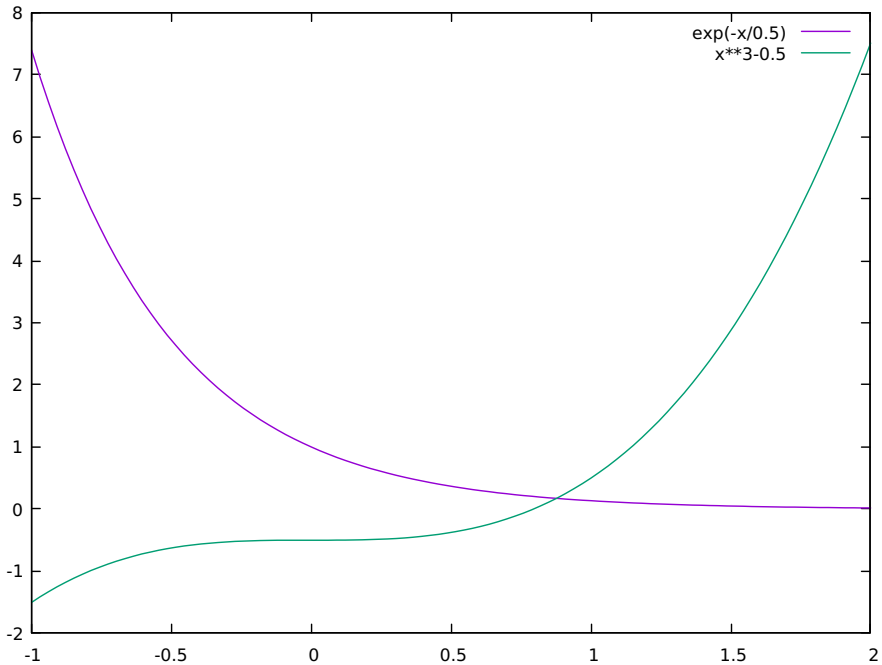
$$x_1 = -e^{\frac{x_0}{p_0}} \quad \text{and} \quad x_1 = p_1 \cdot x_0^3 \ ,$$

that is, x such that

$$x_1 = -e^{\frac{x_0}{p_0}} = p_1 \cdot x_0^3 \ .$$

This problem amounts to the solution of the system of algebraic equations

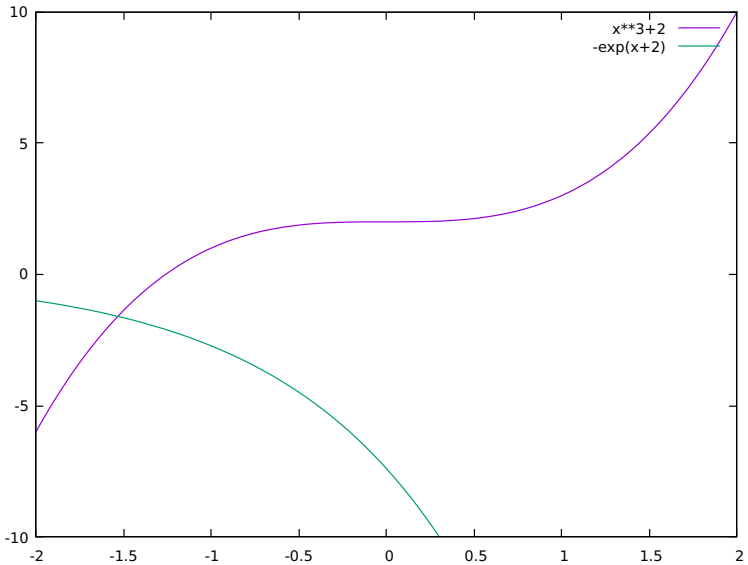
$$p_0 \cdot e^{\frac{x_0}{p_1}} + x_1 = 0$$
$$p_0 \cdot x_0^3 + p_1 - x_1 = 0 \ .$$



```
1  #include "cppNum/algebraicSystem/system.hpp"
2
3  template<typename T>
4  la::vector_t<T> as::system_t::F(const la::vector_t<T> &x, const la::vector_t<T> &p) {
5     la::vector_t<T> r(2);
6     r(0)=p(0)*exp(x(0)/p(1))+x(1);
7     r(1)=p(0)*pow(x(0),3)+p(1)-x(1);
8     return r;
9  }
```

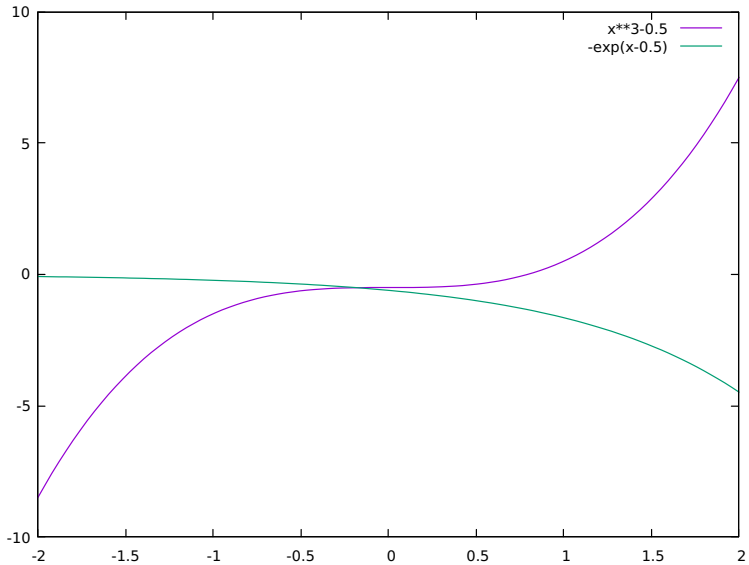
Problem Condition: $a = 10-1, \ldots, 10^{-10}$

$p_0 = 1,\ p_1 = 2$




```
F( -1.36143 -0.505998)=0.000256057  -0.0174154
F( -1.35848 -0.507001)= 5.51568e-07 -3.55558e-05
F( -1.35848 -0.507001)= 5.51568e-07 -3.55558e-05
F( -1.35848 -0.507001)= 5.51568e-07 -3.55558e-05
F( -1.35847 -0.507003)= 2.31637e-12 -1.48961e-10
F( -1.35847 -0.507003)= 2.31637e-12 -1.48961e-10
F( -1.35847 -0.507003)= 2.31637e-12 -1.48961e-10
F( -1.35847 -0.507003)= 2.31637e-12 -1.48961e-10
F( -1.35847 -0.507003)= 2.31637e-12 -1.48961e-10
F( -1.35847 -0.507003)=           0 -1.11022e-16
```

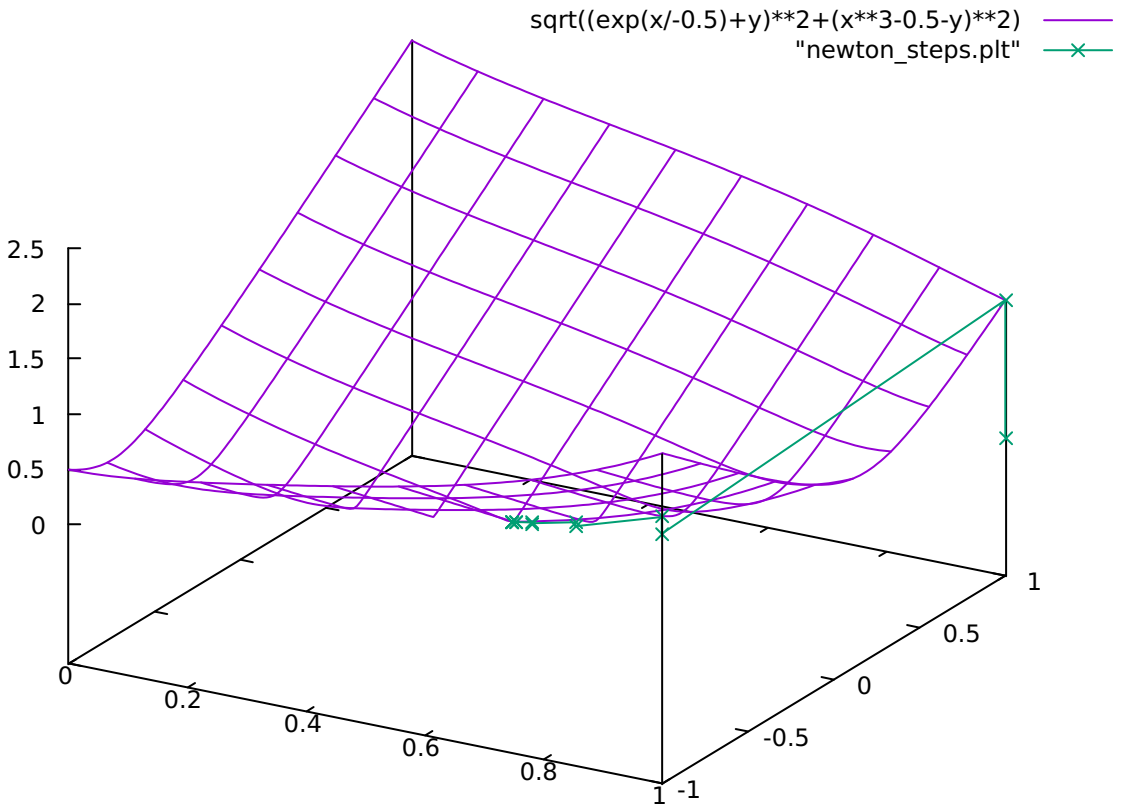
$p_0 = 1,\ p_1 = -0.5$




```
F( 0.641964 -0.269581)=0.00736642  0.0341455
F( 0.581137 -0.310639)=0.00213507 0.00690062
F( 0.557825 -0.327357)=0.000345277 0.000934726
F( 0.553223 -0.330719)=1.39268e-05 3.53535e-05
F( 0.553031 -0.330859)=2.43811e-08 6.11593e-08
F( 0.553031 -0.330859)=2.43811e-08 6.11593e-08
F( 0.553031 -0.330859)=2.43811e-08 6.11593e-08
F( 0.55303 -0.33086)=7.40519e-14 1.85463e-13
F( 0.55303 -0.33086)=7.40519e-14 1.85463e-13
F( 0.55303 -0.33086)=7.40519e-14 1.85463e-13
```

```
1  #include "apps/intersection/intersection.hpp"
2  #include "cppNum/algebraicSystem/newton.hpp"
3
4  int main() {
5    using T=double;
6    const int np=2, nx=2;
7    la::vector_t<T> p(np), x(nx);
8    p(0)=1; p(1)=-0.5; x(0)=1; x(1)=1;
9    as::newton_solver_t<T> solver(1e-7,true);
10   x=solver.run(x,p);
11   solver.plot("newton_steps.plt",0,1);
12   return 0;
13 }
```

:-) `gnuplot run.gnuplot` yields



Let $F(x(p), p) = 0$ with $F : \boldsymbol{R}^{nx} \times \boldsymbol{R}^{np} \to \boldsymbol{R}^{nx}$ be continuously differentiable wrt. both x and p.

From

$$\frac{dF}{dp} = \frac{\partial F}{\partial p} + \frac{dF}{dx} \cdot \frac{dx}{dp} = 0$$

follows ($\to$ implicit function theorem)

$$\underbrace{\frac{dx}{dp}}_{\in \boldsymbol{R}^{nx \times np}} = - \underbrace{\frac{dF}{dx}^{-1}}_{\in \boldsymbol{R}^{nx \times nx}} \cdot \underbrace{\frac{\partial F}{\partial p}}_{\in \boldsymbol{R}^{nx \times np}} .$$

Parameter Sensitivity I

```
1  #include "apps/intersection/intersection.hpp"
2  #include "cppNum/algebraicSystem/newton.hpp"
3  #include <iostream>
4
5  int main() {
6    using BT=double;
7    using T=ad::tangent_t<BT>;
8    using namespace std;
9    as::newton_solver_t<T> solver(1e-7);
10   const int np=2, nx=2;
11   la::vector_t<T> p(np), x(nx);
12   p(0)=1; p(1)=-0.5; x(0)=1; x(1)=1;
13
14   // algorithmic differentiation
15   cout << "x_p^T (ad) =" << endl;
16   for (int i=0;i<np;i++) {
17     ad::derivative(p(i))=1;
18     cout << ad::derivative(solver.run(x,p)(0)) << ' '
19          << ad::derivative(solver.run(x,p)(1)) << endl;
20     ad::derivative(p(i))=0;
21   }
```

```
22
23    // symbolic differentiation (implicit function theorem)
24    x=solver.run(x,p);
25    cout << "x_p^T (sd) =" << endl
26          << derivative_t::dFdx<as::system_t>(x,p).lu().solve(-derivative_t::dFdp<as::
                system_t>(x,p)).transpose() << endl;
27    return 0;
28  }
```

We consider initial value problems for explicit ordinary differential equations yielding solutions $x(p,t) \in \boldsymbol{R}^{nx}$ depending on a parameter $p \in \boldsymbol{R}^{np}$ and time $t \in \boldsymbol{R}$.

For given $x^0(p) = x(p,0)$ and target time $t_{\text{end}} > 0$ the solution $x(p, t_{\text{end}})$ is obtained by integration of the explicit ordinary differential equation

$$x_t = G(x,p)$$

with user-defined $G : \boldsymbol{R}^{nx} \times \boldsymbol{R}^{np} \to \boldsymbol{R}^{nx}$.

cppNum v2.2 implements the explicit (also: forward) Euler integration method for the iterative approximation of the solution.

- explicit Euler method
- usage
- demo
- discussion of the source code
- case study
  - visualization
  - parameter sensitivity

The explicit Euler method replaces the time derivative x_t in

$$x_t = G(x(t), p)$$

with a forward finite difference yielding

$$\frac{x(t + \Delta t) - x(t)}{\Delta t} = G(x(t), p)$$

and, hence, the iterative approximation of the solution as

$$x(t + \Delta t) = x(t) + \Delta t \cdot G(x(t), p)$$

for given $x(0) = x^0$ and $\Delta t > 0$.

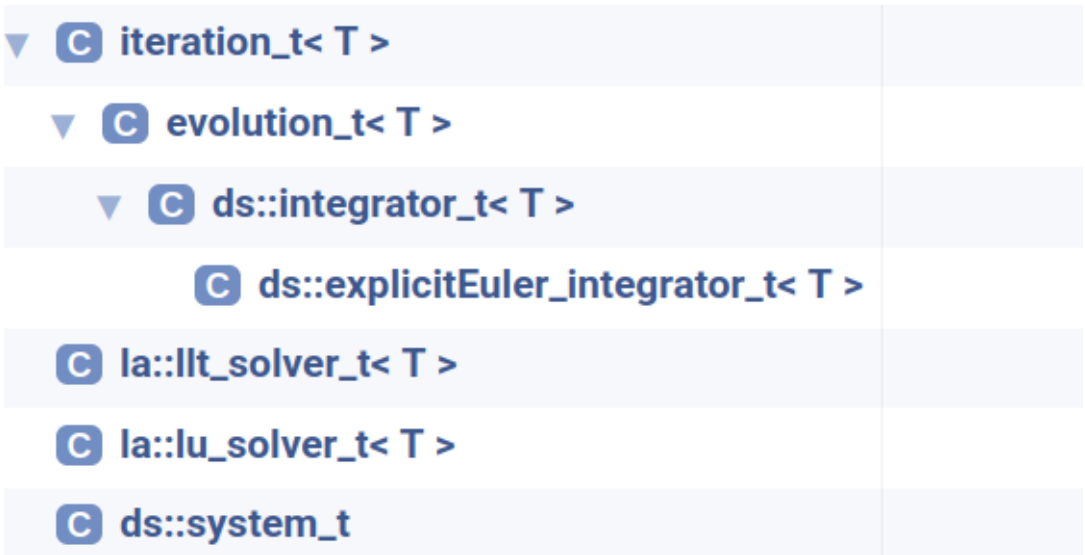
```
1  #include "lotkaVolterra.hpp"
2  #include "cppNum/differentialSystem/explicitEuler.hpp"
3  #include <iostream>
4
5  int main() {
6    using T=double;
7    using namespace std;
8    const int np=4, nx=2;
9    la::vector_t<T> p(np), x(nx);
10   p(0)=1.1; p(1)=p(2)=0.4; p(3)=0.1; x(0)=x(1)=10;
11   ds::explicitEuler_integrator_t<T> integrator(100,10000);
12   x=integrator.run(x,p);
13   cout << "x=" << x.transpose() << endl;
14   return 0;
15 }
```

- make test in ./apps/lotkaVolterra/
- run ./main.exe
- inspect output

  ```
  x=0.0491593   0.453338
  ```

- modify in main.cpp
  - initial state
  - parameters
  - target time

Class Hierarchy (by doxygen):



- Eigen wrappers are supplied in linearAlgebra.hpp.
- Fundamentals for tracing of states visited during the iteration are provided in iteration.hpp.
- Fundamentals for tracing of time steps performed by the evolution and for output of the trace for 2D subspaces in gnuplot format are provided in evolution.hpp.
- The signature of the residual G to be supplied by the user is fixed in differentialSystem/system.hpp.
- An abstract base for integration methods is defined in differentialSystem/integrator.hpp.
- The explicit Euler integration method is implemented in differentialSystem/explicitEuler.hpp.

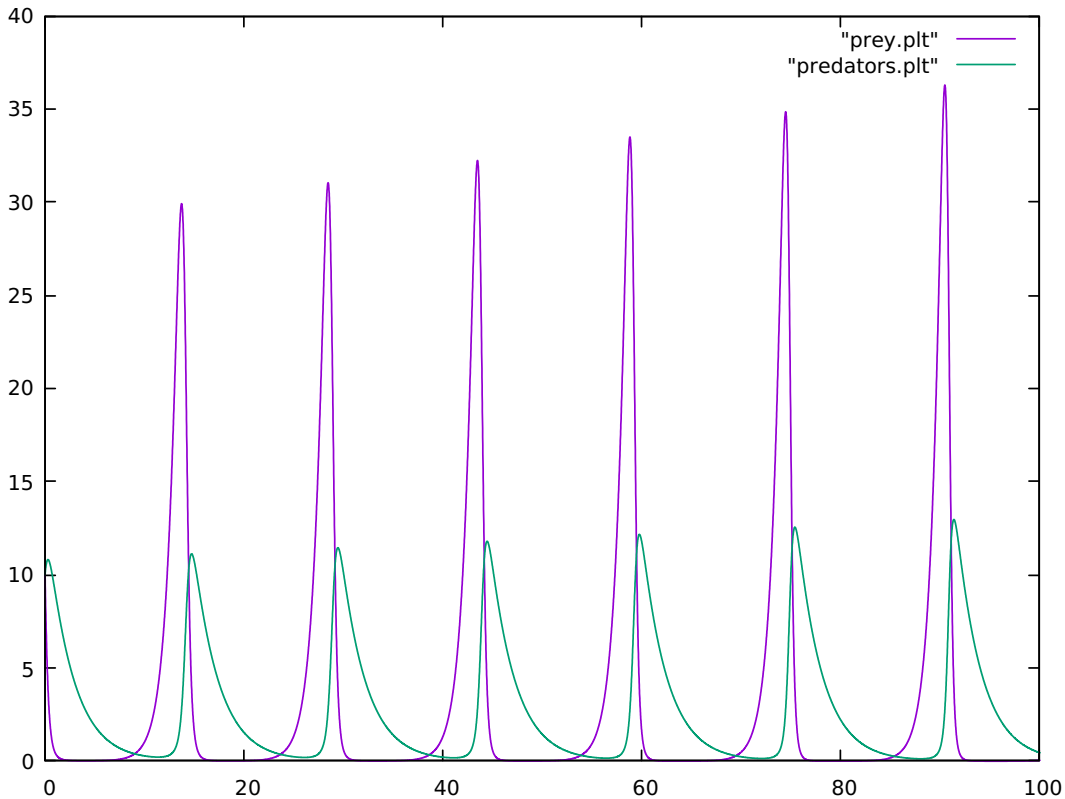
The Lotka–Volterra equations describe the dynamics of biological systems in which two species (prey and predator) interact:

$$x_{0\,t} = p_0 \cdot x_0 - p_1 \cdot x_0 \cdot x_1$$
$$x_{1\,t} = -p_2 \cdot x_1 + p_3 \cdot x_0 \cdot x_1$$

where x_0 is the number of prey, x_1 is the number of predators and $p \in \boldsymbol{R}^4$ describes the interaction (predators eat prey for as long as sufficient prey available)

```
1  #include "apps/lotkaVolterra/lotkaVolterra.hpp"
2  #include "cppNum/differentialSystem/explicitEuler.hpp"
3
4  int main() {
5    using T=double;
6    const int np=4, nx=2;
7    la::vector_t<T> p(np), x(nx);
8    p(0)=1.1; p(1)=p(2)=0.4; p(3)=0.1; x(0)=x(1)=10;
9    ds::explicitEuler_integrator_t<T> integrator(100,10000,true);
10   x=integrator.run(x,p);
11   integrator.plot("prey.plt",0);
12   integrator.plot("predators.plt",1);
13   return 0;
14 }
```

:-) `gnuplot run.gnuplot` yields



Parameter Senstivity

```
1  #include "ad.hpp"
2
3  int main() {
4    using BT=double; using T=ad::tangent_t<BT>; using namespace std;
5    const int np=4, nx=2; la::vector_t<T> p(np), xs(nx), x(nx);
6    p(0)=1.1; p(1)=p(2)=0.4; p(3)=0.1; xs(0)=xs(1)=10;
7    ds::explicitEuler_integrator_t<T> integrator(100,10000);
8    for (int i=0;i<np;i++) {
9      ad::derivative(p(i))=1;
10     x=integrator.run(xs,p);
11     cout << ad::derivative(x(0)) << ' ' << ad::derivative(x(1)) << endl;
12     ad::derivative(p(i))=0;
13   }
14   for (int i=0;i<nx;i++) {
15     ad::derivative(xs(i))=1;
16     x=integrator.run(xs,p);
17     cout << ad::derivative(x(0)) << ' ' << ad::derivative(x(1)) << endl;
18     ad::derivative(xs(i))=0;
19   }
20   return 0;
21 }
```

We look for local minimizers

$$x = \text{argmin}(f)$$

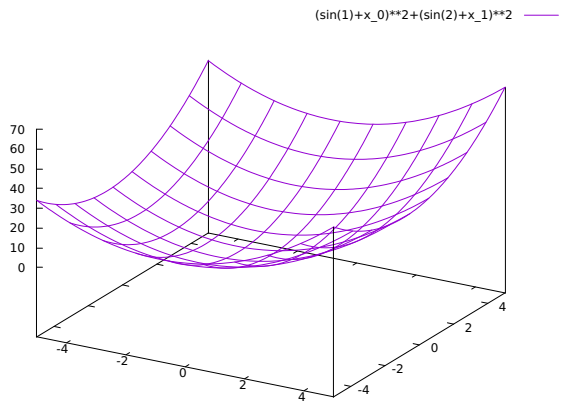
of parameterized, locally convex objectives

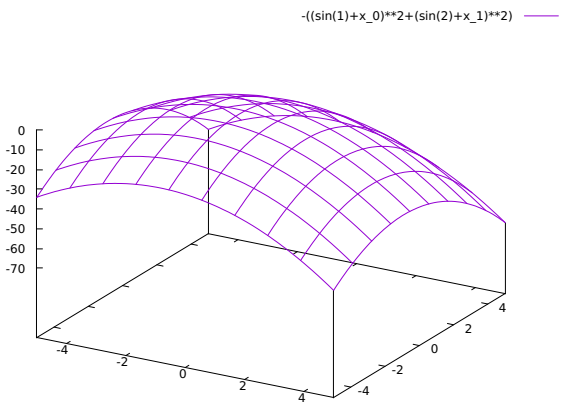
$$y = f(x, p)$$

with user-defined f.

Note

$$\text{argmin}(f) = \text{argmax}(-f)$$





cppNum v2.3 implements the gradient descent method for iterative approximation of the solution.

- gradient descent method
- usage
- demo
- discussion of the source code
- case study
  - visualization
  - parameter sensitivity

The gradient descent method iterates as

$$x = x - \alpha \cdot f_x(x, p)$$

while

$$\|f_x(x, p)\| > a$$

for accuracy $a > 0$. For a local minimum we require $f_{xx}(x, p)$ to be positive definite. Cholesky factorization truns out to be successful in this case.

The optimal step length $\alpha > 0$ can be determined by line search. Heuristically we use bisection of α starting from $\alpha = 1$.

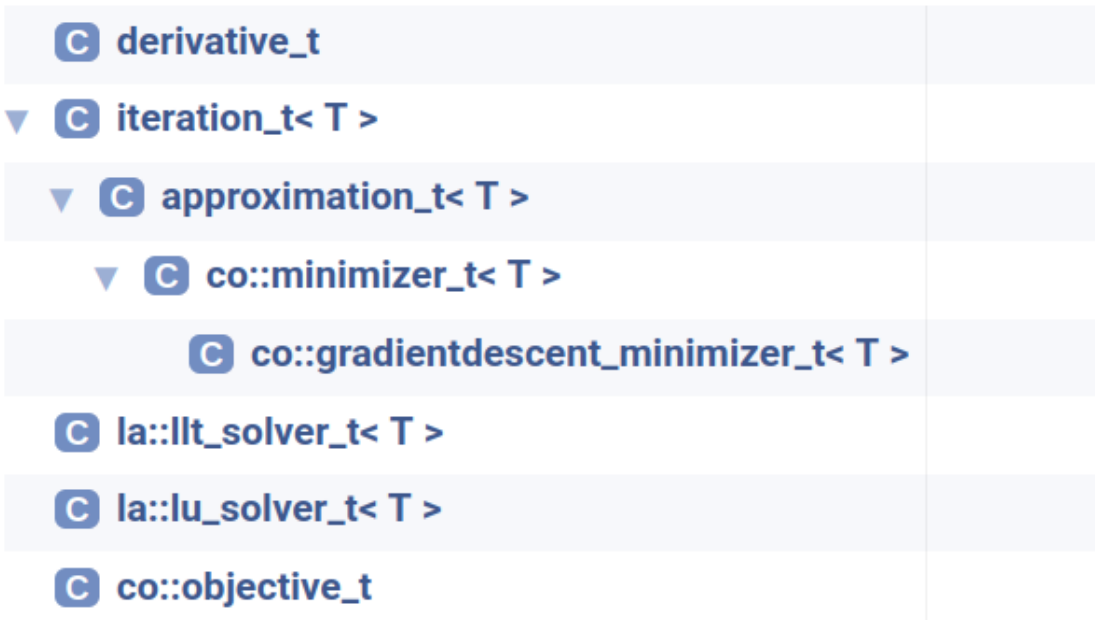
```
1  #include "apps/sphere/sphere.hpp"
2  #include "cppNum/convexObjective/gradientDescent.hpp"
3  #include "cppNum/linearAlgebra.hpp"
4  #include "cppNum/derivative.hpp"
5  #include <iostream>
6
7  int main(int argc, char* argv[]) {
8    using T=double; using namespace std;
9    assert(argc==2);
10   int nx=std::stoi(argv[1]), np=nx;
11   la::vector_t<T> p=la::vector_t<T>::Random(np);
12   la::vector_t<T> x=la::vector_t<T>::Random(nx);
13   co::gradientdescent_minimizer_t<T> minimizer(1e-7);
14   x=minimizer.run(x,p);
15   cout << "x=" << x.transpose() << "\nf(x)=" << co::objective_t::f(x,p) << endl
16     << "||dfdx||=" << derivative_t::dfdx<co::objective_t>(x,p).norm() << endl
17     << "spd(ddfdxx)=" << !(derivative_t::ddfdxx<co::objective_t>(x,p).llt().info()) <<
         endl;
18   return 0;
19 }
```

- make test in ./apps/sphere
- run ./main.exe 2
- inspect output

```
x=-0.629085   0.209668
f(x)=5.0367e-14
||dfdx||=9.48128e-08
spd(ddfdxx)=1
```

- run ./main.exe n for $n > 2$
- modify in main.cpp
  - initial state
  - parameters
  - accuracy

Class Hierarchy (by doxygen):

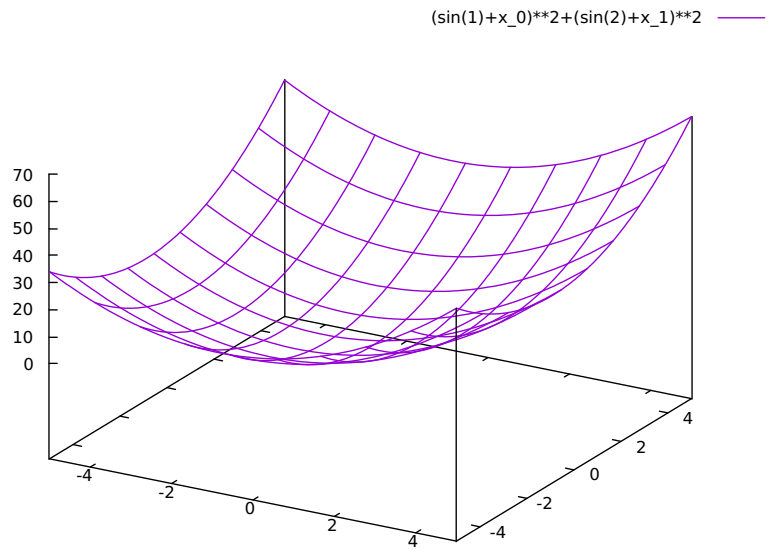


- Eigen wrappers are supplied in linearAlgebra.hpp.
- Fundamentals for tracing of states visited during the iteration and for output of the trace for 2D and 3D subspaces in gnuplot format are provided in iteration.hpp.
- Approximation methods are defined as iterations to be terminated after reaching a given accuracy.
- The signature of the objective f to be supplied by the user is fixed in convexObjective/objective.hpp.
- Functions for computing the required first and second derivatives of f wrt. both x and p can be found in derivatives.hpp.
- An abstract base for local minimization methods is defined in convexObjective/minimizer.hpp.
- The gradient descent method is implemented in convexObjective/gradientDescent.hpp.

The Sphere function a test function for optimization methods. We consider a slightly modified version:

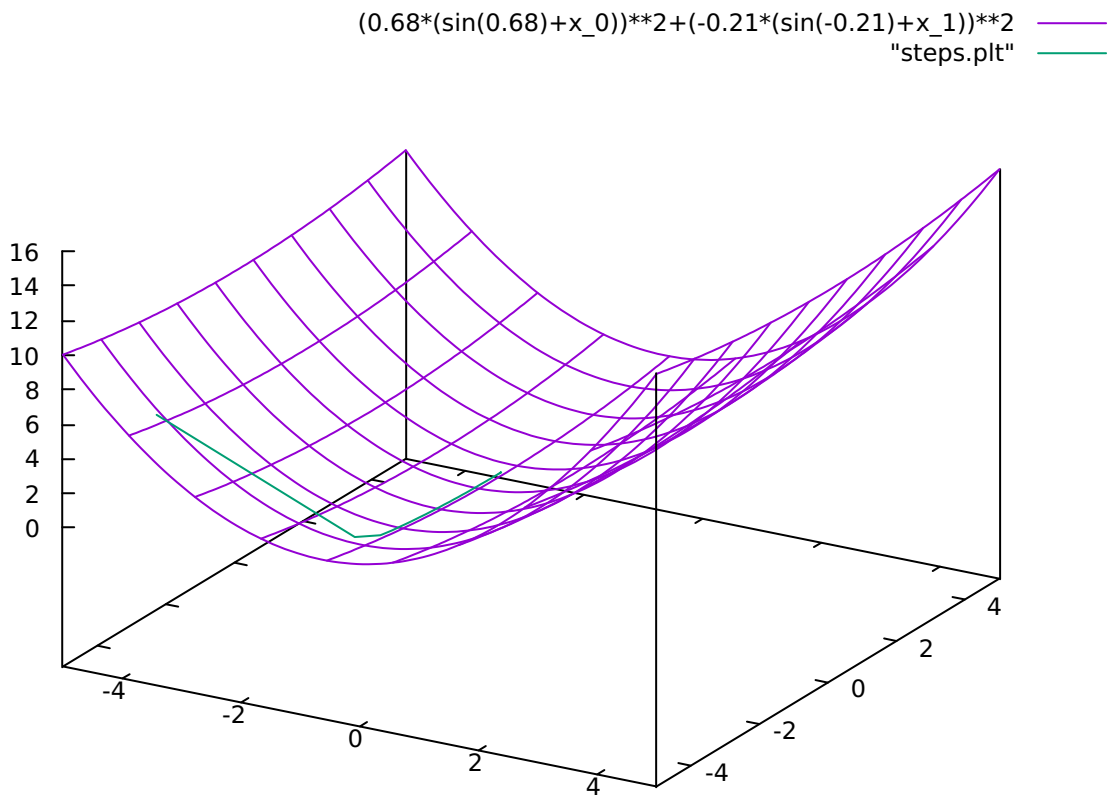
$$f(x) = \sum_{i=0}^{n-1} (p_i \cdot (\sin(p_i) + x_i))^2 \ ,$$

where $p_i \neq 0$. It has exactly one (global) minimum at $x = -\sin(p)$.



```
1  #include "apps/sphere/sphere.hpp"
2  #include "cppNum/convexObjective/gradientDescent.hpp"
3  #include "cppNum/linearAlgebra.hpp"
4
5  int main(int argc, char* argv[]) {
6    using T=double;
7    assert(argc==2);
8    int np=std::stoi(argv[1]), nx=np;
9    la::vector_t<T> p=la::vector_t<T>::Random(np);
10   la::vector_t<T> x=-4*la::vector_t<T>::Ones(nx);
11   co::gradientdescent_minimizer_t<T> minimizer(1e-7,true);
12   x=minimizer.run(x,p);
13   minimizer.plot("steps.plt",0,1);
14   return 0;
15 }
```

:-) `gnuplot run.gnuplot` yields



Let $x^* = x^*(p) = \min_x f(x(p), p)$ for twice continuously differentiable $f : \boldsymbol{R}^{nx} \times \boldsymbol{R}^{np} \to \boldsymbol{R}$ wrt. both x and p yielding

$$f_x(x^*) = 0 \quad \text{and} \quad f_{xx}(x^*) > 0 \,.$$

From

$$f_{xp}(x^*) = \frac{\partial f_x}{\partial p}(x^*) + f_{xx}(x^*) \cdot x_p(x^*) = 0$$

follows

$$\underbrace{x_p}_{\in \boldsymbol{R}^{nx \times np}} = - \underbrace{f_{xx}^{-1}}_{\in \boldsymbol{R}^{nx \times nx}} \cdot \underbrace{\frac{\partial f_x}{\partial p}}_{\in \boldsymbol{R}^{nx \times np}} \,.$$

```
1  #include "apps/sphere/sphere.hpp"
2  #include "cppNum/convexObjective/gradientDescent.hpp"
3  #include "cppNum/linearAlgebra.hpp"
4  #include "cppNum/derivative.hpp"
5  #include <iostream>
6
7  int main(int argc, char* argv[]) {
8    using BT=double;
9    using namespace std;
10   using T=ad::tangent_t<BT>;
11   assert(argc==2); int np=stoi(argv[1]), nx=np;
12   la::vector_t<BT> p=la::vector_t<BT>::Random(np);
13 { // algorithmic differentiation
14   la::vector_t<T> p_t(np);
15   for (int i=0;i<np;i++) p_t(i)=p(i);
16   co::gradientdescent_minimizer_t<T> minimizer(1e-7);
17   cout << "x_p (ad) =" << endl;
18   for (int i=0;i<np;i++) {
19     la::vector_t<T> x_t=-4*la::vector_t<T>::Ones(nx);
20     ad::derivative(p_t(i))=1;
21     x_t=minimizer.run(x_t,p_t);
```

```
22      for (int j=0;j<nx;j++) cout << ad::derivative(x_t(j)) << ' ';
23      cout << endl;
24      ad::derivative(p_t(i))=0;
25    }
26  }
27  { // symbolic differentiation (implicit function theorem)
28    la::vector_t<BT> x=-4*la::vector_t<BT>::Ones(nx);
29    co::gradientdescent_minimizer_t<BT> minimizer(1e-7);
30    x=minimizer.run(x,p);
31    cout << "x_p (sd) =" << endl;
32    cout << derivative_t::ddfdxx<co::objective_t>(x,p).lu()
33                .solve(-derivative_t::ddfdxp<co::objective_t>(x,p)) << endl;
34  }
35    return 0;
36  }
```

We look for local minimizers

$$x = \text{argmin}(f)$$

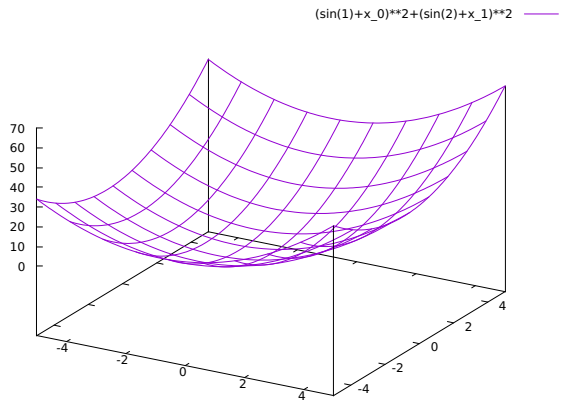
of parameterized, locally convex objectives

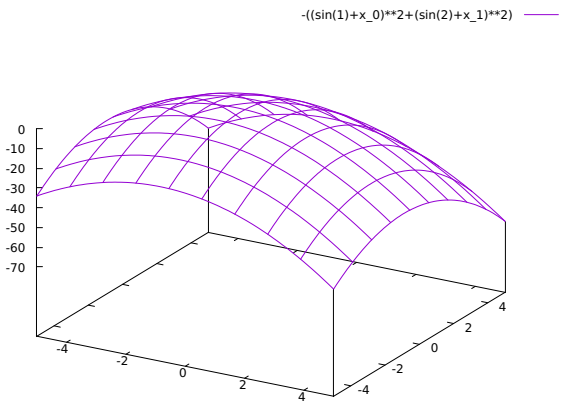
$$y = f(x, p)$$

with user-defined f.

Note

$$\text{argmin}(f) = \text{argmax}(-f)$$





cppNum v2.4 implements the Newton method for iterative approximation of the solution.

- Newton method
- usage
- demo
- discussion of the source code
- case study
  - visualization
  - parameter sensitivity

The Newton method iterates as

$$x = x - f_{xx}(x, p)^{-1} \cdot f_x(x, p)$$

while

$$\|f_x(x, p)\| > a$$

for given accuracy $a > 0$.

The Newton step Δx is computed in each iteration as the solution of the system of linear equations

$$f_{xx}(x, p) \cdot \Delta x = -f_x(x, p)$$

followed by incrementing x.

For a local minimum we require $f_{xx}(x, p)$ to be positive definite. Cholesky factorization turns out to be successful in this case.

Conditions for convergence apply. Line search may help to meet them.

```
1  #include "apps/sphere/sphere.hpp"
2  #include "cppNum/convexObjective/newton.hpp"
3  #include "cppNum/linearAlgebra.hpp"
4  #include "cppNum/derivative.hpp"
5  #include <iostream>
6
7  int main(int argc, char* argv[]) {
8    using T=double;
9    using namespace std;
10   assert(argc==2);
11   int np=std::stoi(argv[1]), nx=np;
12   la::vector_t<T> p=la::vector_t<T>::Random(np);
13   la::vector_t<T> x=la::vector_t<T>::Random(nx);
14   co::newton_minimizer_t<T> minimizer(1e-7);
15   x=minimizer.run(x,p);
16   cout << "x=" << x.transpose() << "\nf(x)=" << co::objective_t::f(x,p) << endl
17     << "||dfdx||=" << derivative_t::dfdx<co::objective_t>(x,p).norm() << endl
18     << "spd(ddfdxx)=" << !(derivative_t::ddfdxx<co::objective_t>(x,p).llt().info()) <<
         endl;
19   return 0;
20 }
```

- make test in ./apps/sphere
- run ./main.exe 2
- inspect output

```
x=-0.629085   0.209667
f(x)=5.84331e-33
||dfdx||=1.02906e-16
spd(ddfdxx)=1
```

- run ./main.exe n for $n > 2$
- modify in main.cpp
  - initial state
  - parameters
  - accuracy

Class Hierarchy (by doxygen):

- derivative_t
- iteration_t< T >
  - approximation_t< T >
    - as::solver_t< T, SYSTEM_T >
      - co::minimizer_t< T >
        - co::newton_minimizer_t< T, LINEAR_SOLVER_T >
      - as::solver_t< T, system_t >
        - as::newton_solver_t< T, SYSTEM_T, LINEAR_SOLVER_T >
- la::llt_solver_t< T >
- la::lu_solver_t< T >
- co::objective_t
- as::system_t

- The functionality provided in linearAlgebra.hpp, iteration.hpp, approximation.hpp, and derivative.hpp is similar to v2.3.
- The signature of the objective f to be supplied by the user is fixed in convexObjective/objective.hpp.
- An abstract base for local minimization methods is defined in convexObjective/minimizer.hpp.
- The Newton method is implemented in convexObjective/newton.hpp. It calls the Newton method for solving algebraic systems, which needs to be extended to handle variable system types similar to v1.4 yielding

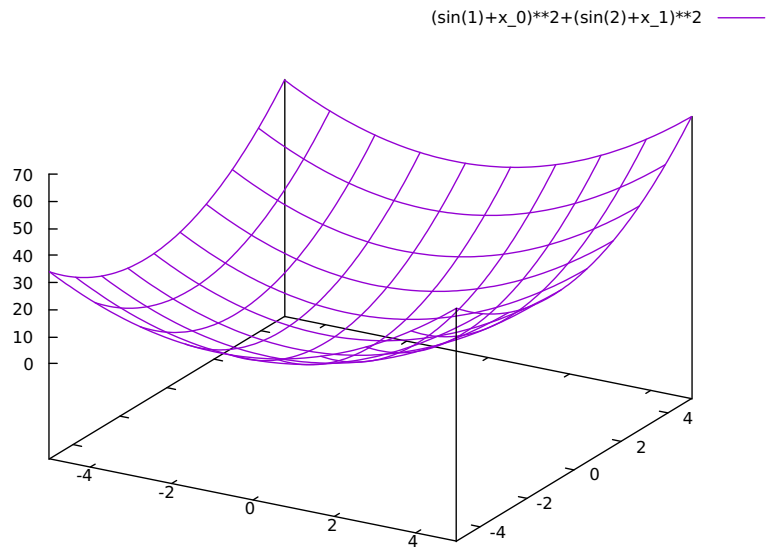
```
1 template<typename T,
2          typename SYSTEM_T=system_t,
3          typename LINEAR_SOLVER_T=la::lu_solver_t<T>
4         >
5   class newton_solver_t : public solver_t<T,SYSTEM_T> { ... }
```

in algebraicSystem/newton.hpp.

The Sphere function a test function for optimization methods. We consider a slightly modified version:

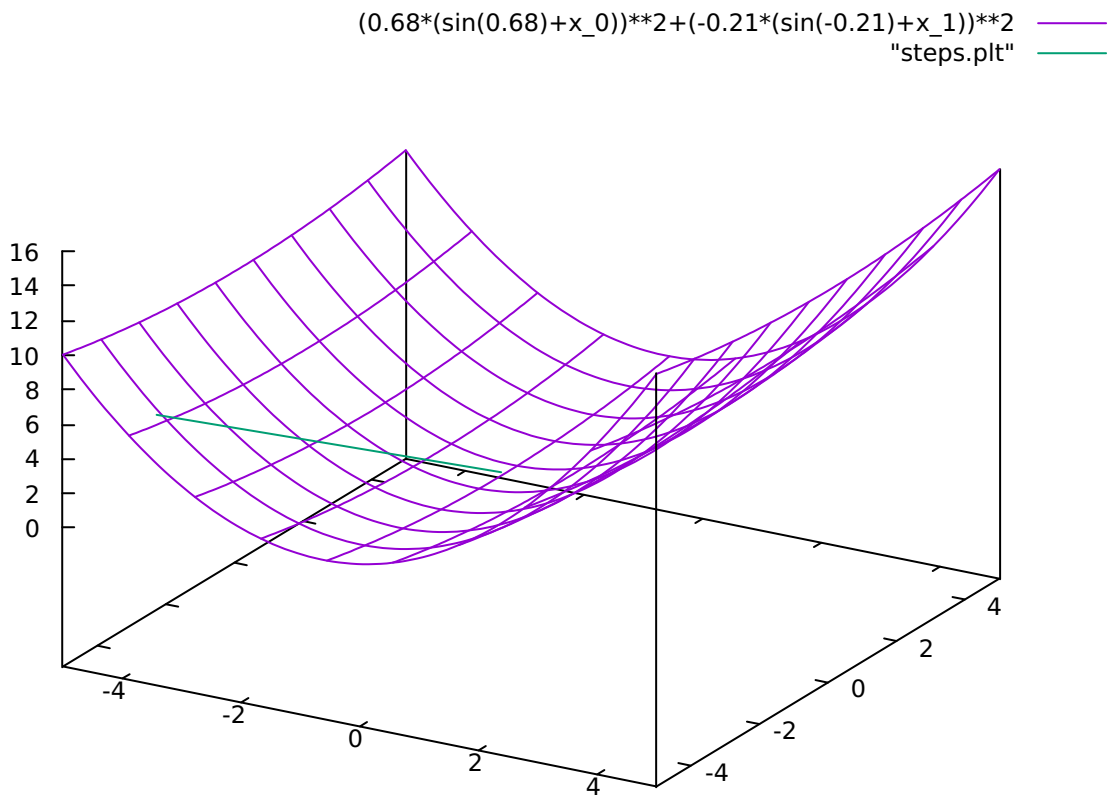
$$f(x) = \sum_{i=0}^{n-1} (p_i \cdot (\sin(p_i) + x_i))^2 \ ,$$

where $p_i \neq 0$. It has exactly one (global) minimum at $x = -\sin(p)$.



```
1  #include "apps/sphere/sphere.hpp"
2  #include "cppNum/convexObjective/newton.hpp"
3  #include "cppNum/linearAlgebra.hpp"
4
5  int main(int argc, char* argv[]) {
6    using T=double;
7    assert(argc==2);
8    int np=std::stoi(argv[1]), nx=np;
9    la::vector_t<T> p=la::vector_t<T>::Random(np);
10   la::vector_t<T> x=-4*la::vector_t<T>::Ones(nx);
11   co::newton_minimizer_t<T> minimizer(1e-7,true);
12   x=minimizer.run(x,p);
13   minimizer.plot("steps.plt",0,1);
14   return 0;
15 }
```

:-) `gnuplot run.gnuplot` yields



Parameter Sensitivity I

```
1  #include "apps/sphere/sphere.hpp"
2  #include "cppNum/convexObjective/newton.hpp"
3  #include "cppNum/linearAlgebra.hpp"
4  #include "cppNum/derivative.hpp"
5  #include <iostream>
6
7  int main(int argc, char* argv[]) {
8    using BT=double;
9    using namespace std;
10   using T=ad::tangent_t<BT>;
11   assert(argc==2); int np=stoi(argv[1]), nx=np;
12   la::vector_t<BT> p=la::vector_t<BT>::Random(np);
13 { // algorithmic differentiation
14   la::vector_t<T> p_t(np);
15   for (int i=0;i<np;i++) p_t(i)=p(i);
16   co::newton_minimizer_t<T> minimizer(1e-7);
17   cout << "x_p (ad) =" << endl;
18   for (int i=0;i<np;i++) {
19     la::vector_t<T> x_t=-4*la::vector_t<T>::Ones(nx);
20     ad::derivative(p_t(i))=1;
21     x_t=minimizer.run(x_t,p_t);
```

```
22        for (int j=0;j<nx;j++) cout << ad::derivative(x_t(j)) << ' ';
23        cout << endl;
24        ad::derivative(p_t(i))=0;
25      }
26    }
27    { // symbolic differentiation (implicit function theorem)
28      la::vector_t<BT> x=-4*la::vector_t<BT>::Ones(nx);
29      co::newton_minimizer_t<BT> minimizer(1e-7);
30      x=minimizer.run(x,p);
31      cout << "x_p (sd) =" << endl;
32      cout << derivative_t::ddfdxx<co::objective_t>(x,p).lu()
33                  .solve(-derivative_t::ddfdxp<co::objective_t>(x,p)) << endl;
34    }
35      return 0;
36    }
```

We consider initial value problems for explicit ordinary differential equations yielding solutions $x(p,t) \in \boldsymbol{R}^{nx}$ depending on a parameter $p \in \boldsymbol{R}^{np}$ and time $t \in \boldsymbol{R}$.

For given $x^0(p) = x(p,0)$ and target time $t_{\text{end}} > 0$ the solution $x(p, t_{\text{end}})$ is obtained by integration of the explicit ordinary differential equation

$$x_t = G(x,p)$$

with user-defined $G : \boldsymbol{R}^{nx} \times \boldsymbol{R}^{np} \to \boldsymbol{R}^{nx}$.

cppNum v2.2 implements the implicit (also: backward) Euler integration method for the iterative approximation of the solution.

- implicit Euler method
- usage
- demo
- discussion of the source code
- case study
  - visualization
  - parameter sensitivity

The implicit Euler method replaces the time derivative x_t in

$$x_t = G(x(t), p)$$

with a backward finite difference yielding

$$\frac{x(t) - x(t - \Delta t)}{\Delta t} = G(x(t), p)$$

and, hence, $x(t)$ as the solution of the algebraic system

$$x(t) - x(t - \Delta t) - \Delta t \cdot G(x(t), p) = 0 \ ,$$

e.g. using the Newton method with user-supplied accuracy $a \in \boldsymbol{R}$..

The solution is approximated iteratively for given $x(0) = x^0$ and $\Delta t > 0$. The algebraic system is solved for each time step.

```
1  #include "lotkaVolterra.hpp"
2  #include "cppNum/differentialSystem/implicitEuler.hpp"
3  #include <iostream>
4
5  int main() {
6    using T=double; using namespace std;
7    const int np=4, nx=2;
8    la::vector_t<T> p(np), x(nx);
9    p(0)=1.1; p(1)=p(2)=0.4; p(3)=0.1; x(0)=x(1)=10;
10   ds::implicitEuler_integrator_t<T> integrator(100,10000,1e-10);
11   x=integrator.run(x,p);
12   cout << "x=" << x.transpose() << endl;
13   return 0;
14 }
```

- make test in ./apps/lotkaVolterra
- run ./main.exe
- inspect output

  ```
  x=0.0984205      1.7817
  ```

- modify in main.cpp
  - initial state
  - parameters
  - target time
- compare with explicit Euler method

Class Hierarchy (by doxygen):

- derivative_t
- iteration_t< T >
  - approximation_t< T >
    - as::solver_t< T, SYSTEM_T, DATA_T >
      - as::solver_t< T, system_t, void >
        - as::newton_solver_t< T, SYSTEM_T, DATA_T, LINEAR_SOLVER_T >
  - evolution_t< T >
    - ds::integrator_t< T >
      - ds::implicitEuler_integrator_t< T >
- la::llt_solver_t< T >
- la::lu_solver_t< T >
- ds::system_t
- as::system_t

- The functionality provided in linearAlgebra.hpp, iteration.hpp, approximation.hpp, evolution.hpp and derivative.hpp is similar to v2.1 and v2.2.
- The signature of the residual G to be supplied by the user is fixed in differentialSystem/system.hpp. An abstract base for integration methods is defined in differentialSystem/integrator.hpp.
- The implicit Euler integration method is implemented in differentialSystem/implicitEuler.hpp. It calls the Newton method for solving algebraic systems, which needs to be extended to accept additional data as an argument similar to v1.5 yielding

```
1 template<typename T,
2          typename SYSTEM_T=system_t,
3          typename DATA_T=void,
4          typename LINEAR_SOLVER_T=la::lu_solver_t<T>
5         >
6   class newton_solver_t : public solver_t<T,SYSTEM_T,DATA_T> { ... }
```

in algebraicSystem/newton.hpp.

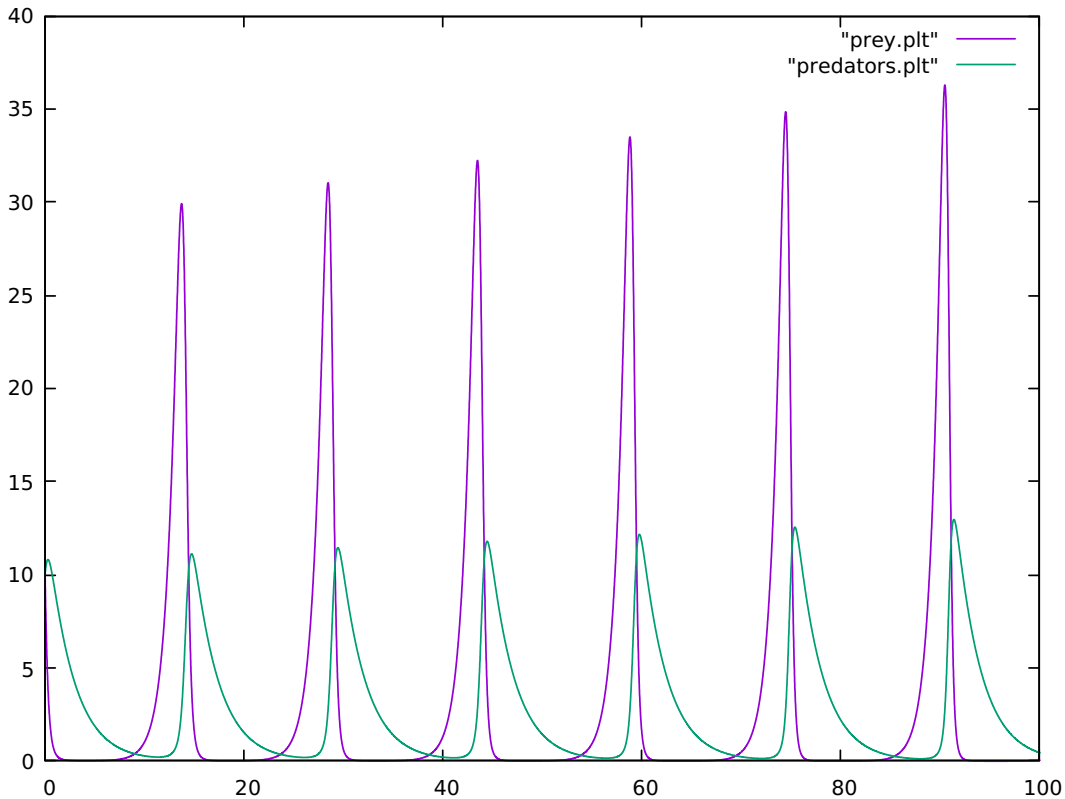
The Lotka–Volterra equations describe the dynamics of biological systems in which two species (prey and predator) interact:

$$x_{0\,t} = p_0 \cdot x_0 - p_1 \cdot x_0 \cdot x_1$$
$$x_{1\,t} = -p_2 \cdot x_1 + p_3 \cdot x_0 \cdot x_1$$

where x_0 is the number of prey, x_1 is the number of predators and $p \in \boldsymbol{R}^4$ describes the interaction (predators eat prey for as long as sufficient prey available)

```
1  #include "apps/lotkaVolterra/lotkaVolterra.hpp"
2  #include "cppNum/differentialSystem/implicitEuler.hpp"
3
4  int main() {
5    using T=double;
6    const int np=4, nx=2;
7    la::vector_t<T> p(np), x(nx);
8    p(0)=1.1; p(1)=p(2)=0.4; p(3)=0.1; x(0)=x(1)=10;
9    ds::implicitEuler_integrator_t<T> integrator(100,10000,true);
10   x=integrator.run(x,p);
11   integrator.plot("prey.plt",0);
12   integrator.plot("predators.plt",1);
13   return 0;
14 }
```

:-) `gnuplot run.gnuplot` yields



Parameter Sensitivity

```
1  #include "ad.hpp"
2
3  int main() {
4    using BT=double; using T=ad::tangent_t<BT>; using namespace std;
5    const int np=4, nx=2; la::vector_t<T> p(np), xs(nx), x(nx);
6    p(0)=1.1; p(1)=p(2)=0.4; p(3)=0.1; xs(0)=xs(1)=10;
7    ds::implicitEuler_integrator_t<T> integrator(100,10000);
8    for (int i=0;i<np;i++) {
9      ad::derivative(p(i))=1;
10     x=integrator.run(xs,p);
11     cout << ad::derivative(x(0)) << ' ' << ad::derivative(x(1)) << endl;
12     ad::derivative(p(i))=0;
13   }
14   for (int i=0;i<nx;i++) {
15     ad::derivative(xs(i))=1;
16     x=integrator.run(xs,p);
17     cout << ad::derivative(x(0)) << ' ' << ad::derivative(x(1)) << endl;
18     ad::derivative(xs(i))=0;
19   }
20   return 0;
21 }
```

TODO

- install and run Linux virtual machine
- download cppNum library from RWTHmoodle page of the course
- build and run sample applications
- study slides and cppNum source code
- refer to external sources to fill potential gaps

Aims / Prerequisites

cppNum v1 : The Scalar Case

- Foundations
  - Algebraic Equations by Newton Method
- Extensions
  - Differential Equations by Explicit Euler Method
  - Convex Objectives by Gradient Descent Method
  - Convex Objectives by Newton Method
  - Differential Equations by Implicit Euler Method

cppNum v2 : The Vector Case

TODO