Scripting for Numerics with C++

Uwe Naumann

STCE, RWTH Aachen University

- Our scripting language SNC++ is a (very small) subset of C++ augmented with support for linear algebra and designed for compatibility with algorithmic differentiation.
- The focus is on illustration of fundamental algorithmic aspects taught as part of various STCE courses.
- The focus is neither on efficiency, nor on quality of software engineering.
- Learning SNC++ will be straightforward, if you are not new to (imperative) programming; it may even serve as motivation to dive deeper into the exciting world of C++.
- The Web[1] will be referred to for further reading. This includes the option to search for appropriate literature that may not be accessible online.

[1] e.g., https://en.cppreference.com for details on C++

Contents

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

Outline

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

Source and I/O

Source file, e.g., source.cpp, generated with a standard text editor:

```cpp
// Hello World example
// naumann@stce.rwth-aachen.de

#include <iostream> // chapter from standard library
using namespace std; // avoid namespace std:: prefix, e.g. in std::cout

int main() {
  cout << "What's your name?" << endl; // output to screen
  string s; // variable declaration
  cin >> s; // read from keyboard
  cout << "Hello " << s << '!' << endl;
  return 0; // ignored
}
```

Source and I/O (Explained)

- Describe (line 1) and "sign" your script (2) (to be omitted in the remaining sample scripts for brevity). Use further comments, wherever appropriate.
- Include relevant chapters of the C++ standard library (4), and avoid the need for specifying the std:: namespace (5).
- The function **int** main() is required to obtain an executable program (7–13). We choose to ignore its integer return value.
- A concatenation of strings is written to the screen (8,11). String constants are enclosed in double quotes (8,11). Single character constants are enclosed in single quotes (11). A special "end of line" marker endl is used for formatting.
- The value of a previously declared variable s of type string (9) is read from the keyboard (10) (sequence of characters terminated by pressing [Enter]).
- Variables need to be declared (9) prior to their first use (10). Declarations yield aliases for corresponding sections in memory. They are valid inside the current scope (code wrapped into closest pair of curly brackets) (7–13).

You are encouraged to use a Linux computer for building and execution. It matches the environment used for the presentation of the course.

The GNU C++ Compiler[2] g++ is used to translate the source file (e.g., source.cpp) into an executable (e.g., source.exe).

Type

```
g++ source.cpp -o source.exe
```

on the command line to generate the executable in the subdirectory where source.cpp is located.

[2] https://gcc.gnu.org

Type

```
./source.exe
```

on the command line to run the executable in the subdirectory where `source.exe` is located.

Sample session

```
:–) ./source.exe
What's your name?
Uwe
Hello Uwe!
:–)
```

The three character string ":–)" denotes the command line prompt, which is likely to look different by default.

Outline

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

All variables need to be declared, that is, a type needs to be assigned to them.

The following built-in[3] numeric types turn out to be sufficient for scripting:

- **int**: integers
- **double**: (double-precision) floating-point numbers

We use double instead of single precision floating-point numbers to allow for better stability of the numerical methods to be discussed in further detail in the various courses offered by STCE.

[3] into C++

Type **int** (int.cpp)

```cpp
#include <iostream>
#include <limits> // numeric properties
using namespace std;

int main() {
  int i; // declaration
  int j=0; // declaration and initialization
  i=1; // assignment
  cin >> j; // user input
  j=max((j+3*i)/2,abs(-i)); // sample arithmetic
  cout << j << endl; // output
  cout << numeric_limits<int>::max() << endl // largest value
       << numeric_limits<int>::min() << endl; // smallest value
  return 0;
}
```

Type **int** (Explained)

- Declared variables hold uncertain values (line 6). They should be initialized with some integer constant for deterministic behavior (7).
- Alternatively, values can be assigned explicitly (8).
- Users can supply values via the keyboard (9) (sequence of digits terminated by [Enter]); values can be written to the screen (11).
- A wide range of integer arithmetic is supported, including $+,-,*,/,\%$. Expressions are evaluated from left to right. The usual rules of operator precedence apply. Alternative orders can be implemented by explicit bracketing (10).
- Access to the numeric properties of integers is provided by the <limits> chapter of the standard library (2), e.g., their range is limited to $[-2147483648, 2147483647]$ (lines 12,13).
- Refer to the Web for further details on integer variables and arithmetic.

Type **int** (Sample Session)

```
:-) ./int.exe
2 // user input
22 // output ...
2147483647
-2147483648
```

Integer division rounds to towards zero, e.g., $\frac{45}{2} = 22$.

Wrong values are computed if integer values exceed the range of type **int**, e.g.,

```
:-) ./int.exe
2147483647 // user input
-1073741823 // output ... <-- ???
2147483647
-2147483648
```

Refer to the Web for details.

Type **double** (double.cpp)

```
#include <iostream>
#include <cmath> // arithmetic
#include <limits> // numeric properties
using namespace std;

int main() {
  double x; // declaration
  double y=1.01; // declaration and initialization
  x=1.1e-2; // assignment of 0.011
  cin >> y; // user input
  y=sin(pow(fabs(-x),y)); // sample arithmetic
  cout << y << endl; // output
  cout << numeric_limits<double>::max() << endl // largest double value
       << numeric_limits<double>::min() << endl // smallest double value
       << numeric_limits<double>::epsilon() << endl; // machine epsilon
  return 0;
}
```

Type **double** (Explained)

- Non-integer numerical values are stored in floating-point format.
- Real values are represented by a grid of discrete floating-point numbers yielding various unpleasant numerical effects due to *rounding* and *cancellation*.
- The range of **double** is limited to $\pm[2.22507 \cdot 10^{-308}, 1.79769 \cdot 10^{308}]$.
- **double** constants can be written in decimal (line 8) or scientific notation (9).
- The precision of **double** yields 15 significant digits in decimal notation.
- Implicit conversion of float-point values to **int** (also: *narrowing*) uses rounding.
- Arithmetic operators include $+,-,*,/$. Arithmetic functions include sin,cos,exp,log,fabs,fmax.
- Refer to the Web for further details on floating-point arithmetic[4] arithmetic operators and <cmath>.

[4] e.g., https://ieeexplore.ieee.org/document/8766229

Type **double** (Sample Session)

```
:-) ./double.exe
0.42 // user input
0.149881 // output ...
1.79769e+308
2.22507e-308
2.22045e-16

:-) ./double.exe
42.0 // user input
5.47637e-83 // output ...
1.79769e+308
2.22507e-308
2.22045e-16
```

Note output in decimal vs. scientific format.

Outline

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

```
#include <iostream>
#include <cassert> // assertion
#include <cmath>
using namespace std;

int main() {
  double x;
  cin >> x;
  assert(x>0); // x required to be greater than zero
  cout << sqrt(x) << endl;
  return 0;
}
```

- The standard library chapter <cassert> (line 2) provides an assertion which fails if the condition provided as its argument evaluates to **false**.
- Conditions are formulated by comparing numerical values using relational (>,<,==,!=,>=,<=) operators, e.g., (9)
- Conditions are joined by logical (!,&&,|| corresponding to negation, AND, inclusive OR, respectively) operators.
- Refer to the Web for further details on forming conditions using relational and logical operators.
- Sample session:

```
:-) ./assert.exe
1
1
:-) ./assert.exe
-1
assert.exe: assert.cpp:9: int main(): Assertion 'x>0' failed.
Aborted (core dumped)
```

Outline

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

We consider

- branches
  - **if** ... [**else** ...]
- values of conditions
  - **bool**
- loops
  - **while** ... **do** ...
  - **do** ... **while** ...

if-Branch (if_1.cpp)

Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
  double x,y;
  cin >> x; // user input
  if (x==0) { // test for possible error due to x equal to zero
    cout << "Error: log(0) not defined." << endl; // report error ...
    return 0; // ... and leave the script
  }
  if (x<0) { // deal with infeasible negative argument ...
      x=-x; // ... by negating it
  }
  y=log(x); // evaluate the natural logarithm
  cout << y << endl;
  return 0;
}
```

if-Branch (Explained)

- The code in curly brackets (e.g. `x=-x;` in line 13) is executed if the condition (e.g. `x<0`, (12)) evaluates to **true**.
- Potentially very complex conditions can be formulated using relational and logical operators.
- Error handling should be implemented, e.g. for potentially incorrect user inputs (8–11). Premature termination of the script may have to be the consequence (10).
- Alternative branches (**else**) as well as nested branches can be implemented as illustrated by the upcoming variants.

if-Branch (Sample Session)

```
:-) ./if_1.exe
1 // user input
0 // output

:-) ./if_1.exe
-2.71 // user input
0.996949 // output

:-) ./if_1.exe
0 // user input
Error: log(0) not defined. // output
```

if-Branch Variant (if_2.cpp)

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
  double x,y;
  cin >> x;
  if (x==0) {
    cout << "Error: log(0) not defined." << endl;
    return 0;
  }
  if (x<0) { // self-explanatory
    y=log(-x);
  } else {
    y=log(x);
  }
  cout << y << endl;
  return 0;
}
```

Type bool (bool.cpp)

```
#include <iostream>
#include <limits> // numeric properties
using namespace std;

int main() {
  bool b; // declaration
  bool c=0; // declaration and initialization with 0=false
  b=1; // assignment of 1=true
  cin >> c; // user input
  b=!((b&&c)||c); // sample arithmetic
  cout << b << endl; // output
  cout << numeric_limits<bool>::max() << endl // largest value
       << numeric_limits<bool>::min() << endl; // smallest value
  return 0;
}
```

Type **bool** (Explained)

- Values of conditions, e.g. for defining the flow of control, can be stored in Boolean variables, e.g. `b` (line 6).
- Boolean variables should be initialized with a Boolean constant, e.g. `0` (equivalently, **false**) (7).
- Boolean constant `1` (equivalently, **true**) can be assigned to Boolean variables (8).
- Values of Boolean variables can be provided via the keyboard (9).
- Boolean arithmetic uses negation (!), logical AND (&&) and logical (inclusive) OR (||) (10).
- The properties of Boolean variables can be investigated using the <limits> chapter of the standard library (2,12,13).
- Refer to the Web for further details on Boolean variables and arithmetic.

Type **bool** (Sample Session)

```
:–) ./bool.exe
1 // user input
0 // output ...
1
0

:–) ./bool.exe
0 // user input
1 // output ...
1
0
```

if-Branch Variant (if_3.cpp)

Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
  double x,y;
  cin >> x;
  bool error=false; // Boolean variable
  if (x==0) {
    cout << "Error: log(0) undefined" << endl;
    error=true; // x equal to zero amounts to an error
  } else if (x<0) {
    y=log(-x);
  } else {
    y=log(x);
  }
  if (!error) { // print result unless there was an error
    cout << y << endl;
  }
  return 0;
}
```

while-do-Loop (while_do.cpp)

```
#include <cmath>
#include <iostream>
using namespace std;

int main() {
  double x, eps=5e-1;
  cin >> x;
  while (fabs(x)>eps) { // [re]iteration criterion
    x=sin(x); // loop ...
    cout << x << endl; // ... body
  }
  return 0;
}
```

while-do-Loop (Explained)

- The loop body (lines 9–10) is reexecuted while the condition (fabs(x)>eps) evaluates to **true**.
- Zero or more loop iterations are possible. The condition is checked before the first iteration.
- Termination may not be guaranteed. The sample loop terminates due to contractiveness of the sine function.
- Nesting of loops and combinations with branch constructs are supported.

while-do-Loop (Sample Session)

```
:—) ./while_do.exe
1 // user input
0.841471 // output ...
0.745624
0.67843
0.627572
0.587181
0.554016
0.526107
0.502171
0.481329

:—) ./while_do.exe
0.5 // user input
// no output
```

do-while-Loop (do_while.cpp)


```cpp
#include <cmath>
#include <iostream>
using namespace std;

int main() {
  double x, eps=5e-1;
  cin >> x;
  do { // at least one iteration guaranteed
    x=sin(x); // loop ...
    cout << x << endl; // ... body
  } while (fabs(x)>eps); // reiteration criterion
  return 0;
}
```

do-while-Loop (Explained)

- The loop body (lines 9–10) is reexecuted while the condition (fabs(x)>eps) evaluates to **true**.
- One or more loop iterations are possible. The condition is checked after the first iteration.
- Termination may not be guaranteed. The sample loop terminates due to contractiveness of the sine function.
- Nesting of loops and combinations with branch constructs are supported.

do-while-Loop (Sample Session)

```
:−) ./do_while.exe
1 // user input
0.841471 // output ...
0.745624
0.67843
0.627572
0.587181
0.554016
0.526107
0.502171
0.481329

:−) ./do_while.exe
0.5 // user input
0.479426 // output
```

Outline

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

We discuss

- definition
- execution
- arguments
  - passed by value
  - passed by reference
  - return values
- templates
- headers

Subroutines (subroutines_1.cpp)

```
#include <cmath>
#include <iostream>
using namespace std;

void do_something(double x, double eps) { // subroutine (no value returned)
  do {
    x=sin(x);
    cout << x << endl;
  } while (fabs(x)>eps);
}

int main() {
  double x;
  cin >> x;
  do_something(x,0.5); // subroutine call
  return 0;
}
```

- Subroutines have a name (e.g. do_something), a (return) type (e.g. **void** for missing return value), and a list of typed arguments (e.g. **double** x, **double** eps) (line 5).
- Subroutines are called by their name with a possibly empty list of actual arguments (e.g. x,0.5) (15).
- By default, arguments are passed "by value". All computation inside the subroutine is performed on local copies of the actual arguments (e.g. local x, which is not the same x as the one declared in main, and eps).
- Optionally, additional local variables can be declared and used inside subroutines.
- The given subroutine has no effect on its caller. It performs a task and prints information to the screen.

Subroutines Variant: **return** (subroutines_2.cpp)

```
#include <cmath>
#include <iostream>
using namespace std;

double f(double x, double eps) { // function (with return value)
  double y=x; // result to be computed
  do {
    y=sin(y);
  } while (fabs(y)>eps);
  return y; // result returned
}

int main() {
  double x;
  cin >> x;
  double y=f(x,0.5); // function call
  cout << y << endl;
  return 0;
}
```

- The subroutine is declared to return a value of type **double** to the caller, that is, to `main` (line 5).
- A local variable `y` is declared to hold the value to be returned (6). Equivalently, this computation could have been performed on the local variable `x`, similar to subroutines_1.cpp.
- A copy of `y` is returned (10) as the local variable runs out of scope when leaving the subroutine.
- The value of this copy is assigned to a variable declared in the caller (16).

Subroutines Variant: Reference (subroutines_3.cpp)

```
#include <cmath>
#include <iostream>
using namespace std;

void f(double& x, double eps) { // x passed by reference; eps passed by value
   do {
      x=sin(x);
   } while (fabs(x)>eps);
}

int main() {
   double x;
   cin >> x;
   f(x,0.5);
   cout << x << endl;
   return 0;
}
```

- Arguments to be passed by reference are marked by the prefix & (e.g., x in line 5). Note that whitespaces are ignored by the compiler, that is, you can decide whether to attach the & to the type (as done here) or to the variable name.
- Passing arguments by reference represents an alternative to returning values to the caller as all modification are performed on the actual argument.
- Constants need to be passed to subroutines by value (e.g., 0.5 (14) initializing the local variable eps inside f). Only variables can be passed by reference (e.g., x (14)).

```cpp
#include <cmath>
#include <iostream>
using namespace std;

template<typename T> // variable argument type T
int f(T& x) { // reference to actual argument of type T
  int i=0;
  while (fabs(x)>1) {
    x=x/2; i=i+1;
  }
  return i;
}

int main() { // compiler generates two versions of f ...
  double x=-42.0;
  cout << f(x) << ' '; cout << x << endl; // ... T=double
  int i=42;
  cout << f(i) << ' '; cout << i << endl; // ... T=int
  return 0;
}
```

- The compiler can help to generate instances of subroutines for varying types of arguments (and local variables). Templates for such subroutines need to be provided (lines 5–14).
- Here, the argument x has variables type. It is passed by reference.
- Actual instances of the subroutine are computed by the compiler based on calling scenarios. Here, instances for both T=**int** (18) and T=**double** (20) are generated.
- The arithmetic performed by the subroutine needs to be valid for all types used for instantiation.
- Templates will play a prominent role in the context of algorithmic differentiation; see various courses offered by STCE.

The following output is generated.

```
6 -0.65625
5 1
```

The T=**double** instance of the template subroutine performs six loop iterations to reduce the absolute value of x to at most one. It computes x=−21.0,−10.5,−5.25,−2.625,−1.3125,−0.65625.

Due to integer arithmetic, the T=**int** instance of the template subroutine performs only five loop iterations to reduce the absolute value of x to at most one. It computes x=21,10,5,2,1.

Subroutines Variant: Header (subroutines_5.cpp)

```
#include "subroutines_5.h" // contents of this file replaces this line

#include <iostream>
using namespace std;

int main() {
  double x=42.0;
  cout << f(x) << ' '; cout << x << endl;
  int i=-42;
  cout << f(i) << ' '; cout << i << endl;
  return 0;
}
```

Subroutines Variant (subroutines_5.h)

```
#include<cmath> // required by instances of the template for f
using namespace std;

template<typename T> // template for instances of f
int f(T& x) {
  int i=0;
  while (fabs(x)>1) {
    x=x/2; i=i+1;
  }
  return i;
}
```

subroutines_5.[h,cpp] (Explained)

- Subroutines as well as corresponding templates can be “outsourced” into header files (e.g., subroutines_5.h). Readability and maintainability may thus be improved.
- Header files need to be included into scripts that call subroutines (or instances of corresponding templates) contained therein, e.g., line 1 in subroutines_5.cpp. The **#include** command needs to precede any uses of contents of the included header file.
- Chapters of the C++ standard library should be included where used, e.g., <cmath> in subroutines_5.h and <iostream> in subroutines_5.cpp. Possible duplication is taken care of by the compiler.
- The following output is generated.

```
6 0.65625
5 -1
```

Subroutines Variant: Recursion (subroutines_6.cpp)

```
#include <cmath>
#include <iostream>
using namespace std;

template<typename T>
int f(T& x, int i) {
  if (fabs(x)>1) {
    x=x/2;
    i=f(x,i)+1; // recursion
  }
  return i;
}

int main() {
  double x=-42.0;
  cout << f(x,0) << ' '; cout << x << endl;
  int i=42;
  cout << f(i,0) << ' '; cout << i << endl;
  return 0;
}
```

subroutines_6.cpp (Explained)

- Recursion (calls of subroutines from inside themselves, e.g., line 9) is supported.
- Some algorithms are best stated (and implemented) recursively.
- The logic of this algorithm is best understood by augmenting its implementation with statements for writing values of the variables x and i to the screen.
- The same output as for subroutines_4.cpp is generated.

Outline

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

- The structure of source code the can be improved by moving subprograms to separate include files.
- Include files start with **#pragma** once (include/script.h: line 1) to avoid accidental repeated inclusion (and, hence, errors due to multiple definitions of the same subprogram).
- Functions called by several scripts should be implemented only once (include/script.h: lines 7–11). Multiple copies of the same source code can thus be avoided.
- Usage of functions defined in include files requires inclusion of the latter (script.cpp: line 1).
- Include files can be stored in separate directories. The compiler needs to be informed about the relative (with respect to the directory of the source script) location of these directories via its –I option.

```
#pragma once // prevent multiple definitions due to repeated inclusion

#include <cassert>
#include <cmath>
using namespace std;

template <typename T>
T f(T x) {
  assert(x>0);
  return log(x);
}
```

```cpp
#include "script.h" // definition of f

#include <iostream>
using namespace std;

int main() {
  double x;
  cin >> x;
  cout << f(x) << endl;
  return 0;
}
```

Build as

```
g++ -I./include script.cpp -oscript.exe
```

Outline

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

- Dynamically sized vector and matrix types are provided by Eigen[5]., a C++ template library for linear algebra.
- Vectors and matrices (and tensors, e.g., vectors of matrices or matrices of matrices) of entries of diverse types, e.g., **double**, **int**, **bool**, can be implemented.
- Basic linear algebra operations as well as direct solvers for linear systems will be considered.
- Conceptually, the entire range of functionalities provided by Eigen is available for SNC++.

[5] https://eigen.tuxfamily.org/

Vectors (vector.cpp)

```cpp
#include <Eigen/Dense> // Eigen library
#include <cmath>
#include <iostream>
using namespace std;

int main() {
  int n=3;
  Eigen::VectorX<double> v(n); // allocation
  int i=0;
  do {
    v(i)=cos(i); // assignment of values to vector entries
    i=i+1;
  } while (i<n);
  i=0;
  do {
    cout << v(i) << endl; // read access to vector entries
    i=i+1;
  } while (i<v.size()); // vector size
  return 0;
}
```

- The Eigen (template) library needs to be included (line 1).
- A vector v of n elements (here, n=3) of variable type (here, of type **double**) is allocated (8).
- Access to individual entries requires specification of the index in parentheses (11,16).
- Various characteristics of a vector can be queried with the help of member functions, e.g. its size (18). A member function f without arguments is invoked on a variable v as v.f(). Their implementation is beyond the scope of SNC++.
- Sample session:

```
:-) ./vector.exe
1
0.540302
-0.416147
```

By default, vectors are (printed as) column vectors.

Let the Eigen library be installed in the subdirectory `eigen-3.4.0` of the directory containing all sample scripts.

Type

```
g++ -Ieigen-3.4.0 vector.cpp -o vector.exe
```

on the command line to generate the executable vector.exe.

The `-Ieigen-3.4.0` option tells the compiler where to find the Eigen library, more specifically, Eigen/Dense.

Vector Constants (vector_constants.cpp)

```
#include <Eigen/Dense>
#include <iostream>
using namespace std;

int main() {
  int n=3;
  using VT=Eigen::VectorX<double>; // alias for type name
  VT v=VT::Zero(n); // all zeros
  cout << v.transpose() << endl;
  v=VT::Ones(n); // all ones
  cout << v.transpose() << endl;
  v=VT::Unit(n,1); // (second) Cartesian basis vector
  cout << v.transpose() << endl;
  v=VT::Random(n); // all (pseudo-)random
  cout << v.transpose() << endl;
  return 0;
}
```

Vector Constants (Explained)

- Aliases for (vector) types can be introduced with the help of the **using** clause (line 7). The resulting more compact notation can be convenient in case of repeated use (8,10,12,14).
- Vectors can be initialized with vector constants, e.g., the zero vector (8).
- Similarly, vector constants can be assigned to vectors, e.g., a vector of all ones (10), a Cartesian basis vector (here, the second) (12) or a vector of pseudo-random numbers (14).
- Vectors can be treated (e.g., printed) as row vectors by transposing them using the member functiontranspose (9,11,13,15).
- Sample session:

```
:-) ./vector_constants.exe
0 0 0
1 1 1
0 1 0
 0.680375 -0.211234 0.566198
```

Matrices (matrix.cpp) I

```
#include <Eigen/Dense>
#include <iostream>
using namespace std;

int main() {
  int m=2, n=4;
  Eigen::MatrixX<double> M(m,n); // allocation
  int i,j;

  i=0;
  do { // loop over rows
    j=0;
    do { // loop over columns
      M(i,j)=i+j; // assignment if values to matrix entries
      j=j+1;
    } while (j<n);
    i=i+1;
  } while (i<m);

  i=0;
  do {
```

Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

```
      j=0;
      do {
        cout << M(i,j) << ' '; // read access to matrix entries
        j=j+1;
      } while (j<M.cols()); // number of columns
      cout << endl;
      i=i+1;
    } while (i<M.rows()); // number of rows

    return 0;
}
```

- A matrix M with m rows (here, m=2) and n columns ((here, n=4) of entries of variable type (here, of type **double**) is allocated (line 7).
- Access to individual entries requires specification of the row index followed by the column index in parentheses (14,24).
- Various characteristics of a matrix can be queried, e.g. the number of its colums (26) or rows (29).
- Sample session:

```
:-) ./matrix.exe
0 1 2 3
1 2 3 4
```

By default, matrices are treated as (e.g., printed) as row major. They can be transposed (M.transpose()) to obtain column major ordering.

Matrix Constants (matrix_constants.cpp)

```
#include <Eigen/Dense>
#include <iostream>
using namespace std;

template<typename T>
using MT=Eigen::MatrixX<T>;

int main() {
  int n=2, m=4;
  using T=double;
  MT<T> M=MT<T>::Zero(n,m); // all zeros
  cout << M << endl << endl;
  M=MT<T>::Ones(n,m); // all ones
  cout << M << endl << endl;
  M=MT<T>::Identity(n,n); // identity
  cout << M << endl << endl;
  M=MT<T>::Random(n,m); // all (pseudo-)random
  cout << M << endl;
  return 0;
}
```

- Aliases for (matrix) types over variable element types can be introduced with the help of the (global) type-generic **using** clause (lines 5,6). The actual type needs to be specified when using the alias (11,13,15,17). Fixed-type (global) using clauses are also supported.
- Constants similar to those available for vectors can be used to initialize matrices (11), or they can be assigned to matrices (13,15,17).
- Matrix sizes are adapted dynamically (e.g. from line 13 to line 15 to line 17).
- Sample session:

```
:-) ./matrix_constants.exe
0 0 0 0
0 0 0 0

1 1 1 1
1 1 1 1

1 0
0 1

 0.680375 0.566198 0.823295 -0.329554
-0.211234 0.59688 -0.604897 0.536459
```

Inner Vector Product (wTv.cpp)

```
#include <Eigen/Dense>
#include <iostream>
using namespace std;

int main() {
  int n=3;
  using VT=Eigen::VectorX<double>;
  VT v=VT::Random(n);
  cout << v.transpose() << endl; // single line output
  double vTv=v.dot(v);
  cout << vTv << "==" << v.squaredNorm() << endl; // equal
  v=VT::Unit(n,0);
  VT w=VT::Unit(n,n-1);
  cout << w.dot(v) << endl; // orthogonal
  return 0;
}
```

- The inner (dot) product of two vectors $u, v \in \boldsymbol{R}^n$ is defined as $u^T \cdot v = \sum_{i=0}^{n-1} u_i \cdot v_i \in \boldsymbol{R}$. Eigen provides the corresponding functionality in form of the member function dot (lines 10,14). As a member function of the first operand it takes the second operand as an argument.
- Note that $v^T \cdot v = \|v\|_2^2 = \sum_{i=0}^{n-1} v_i^2$, where the squared Euclidean norm of a vector is implemented in Eigen via the member function squaredNorm (11).
- The inner product of orthogonal vectors (e.g., distinct Cartesian basis vectors) is equal to zero squaredNorm function (12–14).
- Sample session:

```
:-) ./wTv.exe
 0.680375 -0.211234 0.566198
0.828111==0.828111
0
```

Matrix-Vector Product (Mv.cpp)

```cpp
#include <Eigen/Dense>
using VT=Eigen::VectorX<double>;
using MT=Eigen::MatrixX<double>;

#include <iostream>
using namespace std;

int main() {
  int m,n;
  cout << "m="; cin >> m; // number of rows
  cout << "n="; cin >> n; // number of columns
  VT x=VT::Random(n);
  MT A=MT::Random(m,n);
  VT y=A*x; // matrix-vector product
  cout << y.transpose() << endl;
  return 0;
}
```

- The product of a matrix $A(A_{j,i}) \in \boldsymbol{R}^{m \times n}$ with a vector $x = (x_i) \in \boldsymbol{R}^n$ is a vector $y = A \cdot x \in \boldsymbol{R}^m$ defined as

  $$y = (y_j) \equiv \left(\sum_{i=0}^{n-1} A_{j,i} \cdot x_i\right)_{j=0,\ldots,m-1} .$$

  The operator $*$ is *overloaded* [6] accordingly (line 1r43).

- A random matrix of user-defined size $m \times n$ (13) is multiplied with a random vector of matching size n (12) to yield the result of size m.

- The result is transposed prior to being printed (15).

[6] Overloading of functions and operators is supported by C++. It will be used prominently in the context of algorithmic differentiation taught as part of several courses offered by STCE.

We run three sample sessions with varying problem sizes for illustration.

```
:-) ./Mv.exe
m=2 // user input ...
n=3
 0.83762 0.378115 // output

:-) ./Mv.exe
m=1 // user input ...
n=3
-0.110297 // output

:-) ./Mv.exe
m=3 // user input ...
n=1
-0.143719 0.385228 0.406103 // output
```

You are encouraged to validate the results with pen and paper ...

Matrix-Matrix Product (MM.cpp)

```
#include <Eigen/Dense>
#include <iostream>
using namespace std;

int main() {
  int m,n,p;
  cout << "m="; cin >> m;
  cout << "n="; cin >> n; // size of shared dimension
  cout << "p="; cin >> p;
  using MT=Eigen::MatrixX<double>;
  MT A=MT::Random(m,n);
  MT B=MT::Random(n,p);
  MT C=A*B; // matrix-matrix product
  cout << C << endl;
  return 0;
}
```

- The product of two matrices $A \in \boldsymbol{R}^{m \times n}$ and $B \in \boldsymbol{R}^{n \times p}$ is a matrix $C = A \cdot B \in \boldsymbol{R}^{m \times p}$ defined as

$$C = (C_{k,i}) \equiv \left(\sum_{j=0}^{n-1} A_{k,j} \cdot B_{j,i} \right)_{i=0,\dots,p-1}^{k=0,\dots,m-1} .$$

  The operator $*$ is overloaded accordingly (line 13).

- Two random matrices with a shared dimension are multiplied.

- The size of the shared dimension is equal to the number of columns of the first operand, which must be equal the number of rows of the second operand.

- We run three sample sessions with varying problem sizes for illustration. Again, you are encouraged to validate the results with pen and paper ...

Matrix-Matrix Product (Sample Sessions)

```
:-) ./MM.exe
m=2 // user input
k=3
n=4
-0.286392 0.260042 0.575817 1.07782 // output ...
 0.658662 -0.20569 -0.473983 -0.276707

:-) ./MM.exe
m=1 // user input
k=10
n=1
0.504171 // output

:-) ./MM.exe
m=3 // user input
k=1
n=3
 0.406103 0.56015 -0.411557 // output ...
-0.126081 -0.173908 0.127775
 0.337953 0.466148 -0.342492
```

Solution of Linear Systems (LS.cpp)

```
#include <Eigen/Dense>
#include <iostream>
using namespace std;

int main() {
  int n;
  cout << "n="; cin >> n;
  using MT=Eigen::MatrixX<double>;
  using VT=Eigen::VectorX<double>;
  MT A=MT::Random(n,n); // (invertible) system matrix
  VT b=VT::Random(n); // right-hand side
  VT x=A.lu().solve(b); // direct solution by LU factorization
  cout << x.transpose() << endl << endl;
  MT X=A.lu().solve(MT::Identity(n,n)); // inversion
  cout << X << endl << "=" << endl << A.inverse() << endl;
  return 0;
}
```

- Direct solvers for systems of linear equations

$$A \cdot x = b,\ A \in \boldsymbol{R}^{n \times n},\ b \in \boldsymbol{R}^n$$

  determine $x \in \boldsymbol{R}^n$, such that $x = A^{-1} \cdot b$, where A^{-1} denotes the inverse of A.
- A needs to be invertible. All pseudo-random square matrices generated by Eigen satisfy this requirement (line 10).
- Different factorizations of A are supported by Eigen, including LU (line 12), LL^T, and QR.
- The solution of the linear system is computed by the function solve, implementing, e.g., forward and backward substitution (12).
- Inversion of A amounts to solving n simultaneous linear equations $A \cdot X = I$ with $X, I \in \boldsymbol{R}^{n \times n}$ and where I denotes the identity matrix (13).
- Eigen provides the member function inverse for the same task (14).

```
:-) ./LS.exe
n=3 // user input
   0.60876 -0.231282 0.51038 // output ...

-0.198521 2.22739 2.8357
   1.00605 -0.555135 -1.41603
 -1.62213 3.59308 3.28973
=
-0.198521 2.22739 2.8357
   1.00605 -0.555135 -1.41603
 -1.62213 3.59308 3.28973
```

As before, you are encouraged to validate the results with pen and paper ...

Outline

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization

File I/O (fstream.cpp)

```
#include <fstream>
using namespace std;

template<typename T>
void in(T& x, string filename) { // read from file
  ifstream ifs(filename); ifs >> x; // input file stream
}

template<typename T>
void out(T x, string filename) { // write to file
  ofstream ofs(filename); ofs << x; // output file stream
}

int main() {
  int i; double x;
  in(i,"int.in"); out(i,"int.out");
  in(x,"double.in"); out(x,"double_1.out");
  in(i,"double.in"); out(i,"double_2.out");
  return 0;
}
```

File I/O (Explained)

- Data can be read from / written to text files via file streams defined in <fstream> (line 1).
- Declaration of an input file stream of type ifstream requires specification of the name (of type string) of the file to be read from (6).
- Output file streams of type ofstream are declared analogously (12).
- Their usage is similar to cin and cout (6,12).
- The sample session illustrates the effects of file i/o for varying (potentially incompatible) data types.

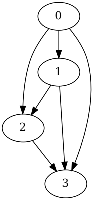
- Let the text file int.in contain the string 42 while double.in contains the string 4.2.
- The text file int.out is an exact copy of int.in (line 16).
- Similarly, double_1.out is an exact copy of double_1.in (17).
- The text file double_2.out contains the string 4 due to narrowing (rounding of 4.2 according to the data type of i, which is equal to **int**) (18).
- Similar effects can be observed for cin and cout.

Drawing Graphs (graphviz.cpp)


```cpp
#include <fstream>
using namespace std;

int main() {
  ofstream ofs("g.dot"); // output file
  ofs << "digraph {" << endl; // a directed graph ...
  int n=4,i,j;
  i=0;
  do {
    j=i+1;
    if (j<n) {
      do {
        ofs << i << " -> " << j << endl; // ... is a set of edges
        j=j+1;
      } while (j<n);
    }
    i=i+1;
  } while (i<n);
  ofs << "}" << endl; // ... end of directed graph
  return 0;
}
```

- Input files for graphviz can be generated, e.g. a directed acyclic complete graph with four vertices.
- :-) `dot -Tpdf g.dot -o g.pdf` generates the following graph in `g.pdf`:




```
digraph {
0 -> 1
0 -> 2
0 -> 3
1 -> 2
1 -> 3
2 -> 3
}
```

The contents of the corresponding file g.dot is shown on the right.

- See www.graphviz.org for detailed information on graphviz and dot.

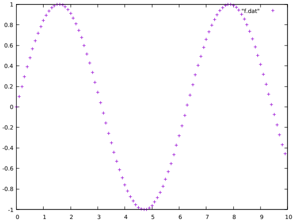
Plotting Data (gnuplot.cpp)

```cpp
#include<cmath>
#include<fstream>
using namespace std;

template<typename T>
T f(T x) { return sin(x); } // function to be drawn

int main() {
  double xmin=0, xmax=10, x; // subdomain
  int nsamples=100; // sampling density
  ofstream ofs("f.dat"); // output file
  x=xmin;
  do {
    ofs << x << ' ' << f(x) << endl; // point on the graph
    x=x+(xmax-xmin)/nsamples; // next argument
  } while (x<=xmax);
  return 0;
}
```

- Input files for gnuplot can be generated, e.g. by sampling a given function into a data file f.dat.
- Run `gnuplot` to open the gnuplot shell.
- Type `plot "f.dat"` to get




```
0 0
0.1 0.0998334
0.2 0.198669
...
9.8 −0.366479
9.9 −0.457536
10 −0.544021
```

The three first and three last lines of the corresponding file f.dat are shown on the right.

- See www.gnuplot.info for detailed information on gnuplot.

Summary

Source Code. Building and Running

Numeric Types and Arithmetic

Defensive Scripting

Flow of Control

Subroutines

Distributed Source

Linear Algebra

File I/O and Visualization