Advanced Praktika SoSe 26 Notes

Igor Dimitrov

2026-04-21

Table of contents

Preface 3

I Notes 4

1 Project Setup 5
- 1.1 General repository structure . . . 5
  - 1.1.1 Experimental Drivers . . . 6
- 1.2 HDNUM, CPFloat and GMP . . . 7
- 1.3 Changing Top-level `CMakeLists.txt` . . . 8
  - 1.3.1 How to integrate `src/` later . . . 9
  - 1.3.2 A flexible version . . . 10
  - 1.3.3 Rule of thumb . . . 11

2 Project Summary 12
- 2.1 Abstract . . . 12
  - 2.1.1 Problem . . . 12
  - 2.1.2 Algorithm . . . 13
  - 2.1.3 HDNUM & Own Implementation . . . 13
  - 2.1.4 Mixing Types . . . 14
  - 2.1.5 Methodology . . . 14
  - 2.1.6 What Should be Studied . . . 15
  - 2.1.7 Plan . . . 16

3 Conversions 17
- 3.1 Floating Point Types . . . 17
- 3.2 Type Conversions . . . 19
  - 3.2.1 Implementing Conversions - Naive Approach . . . 19
  - 3.2.2 Structure of `FP` and `CPFloat` Classes . . . 24
  - 3.2.3 Implementing Conversions - Correct Approach . . . 25

4 Report 34
- 4.1 This is some title . . . 34

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Notes

1 Project Setup

1.1 General repository structure

The whole project is managed inside a Quarto book repository. The Quarto project contains notes, theory summaries, the final report, and the implementation code.

The planned directory structure is:

```
project-root/
  index.qmd
  report/
  notes/
  code/
      CMakeLists.txt
      external/
         hdnum/                 # git submodule
             cpfloat/           # cloned inside hdnum
      include/
         mixed_ir.hpp           # algorithm
         hdnum_conversions.hpp  # convert()
         error_metrics.hpp      # forward/backward error helpers, later
      experiments/
         exp_convergence.cc
         exp_condition_sweep.cc
         exp_residual_precision.cc
      examples/
         hdnum_sandbox.cc
      tests/
         test_conversion.cc
         test_mixed_ir_sanity.cc
      scripts/
         run_all_experiments.py
         plot_results.py
      results/
          raw/
          plots/
```

Purpose of the main sub-directories:

```
external/       external dependencies, especially HDNUM
include/        own header-only algorithm and helper code
src/            optional implementation files, if needed later
examples/       small sandbox/demo programs for learning and testing
experiments/    reproducible experimental drivers for report data
tests/          correctness and sanity checks
scripts/        automation and plotting scripts
results/raw/    generated CSV or data files
results/plots/  generated figures for the report
```

1.1.1 Experimental Drivers

The will go in:

```
code/
  experiments/
     exp_convergence.cc
     exp_condition_sweep.cc
     exp_residual_precision.cc
```

They will be written in c++ because they need to instantiate HDNUM types like FP16, bfloat16, FP64, FP128, FP256, run LU, perform refinement, and compute errors.

But the overall experiment workflow will be split:

```
C++ drivers:
    run numerical experiments
    write CSV files

Python / shell scripts:
    run many driver configurations
    collect results
    make plots
```

like:

```
code/
  experiments/
     exp_convergence.cc
     exp_condition_sweep.cc
     exp_residual_precision.cc
  scripts/
     run_all_experiments.py
     plot_results.py
  results/
      raw/
      plots/
```

The cpp experiments can be written so that they can be run like:

```
./build/exp_condition_sweep --uf FP16 --u FP64 --ur FP128 --n 100 --matrix random_orthogonal
./build/exp_condition_sweep --uf bfloat16 --u FP64 --ur FP128 --n 100 --matrix random_orthogo
./build/exp_condition_sweep --uf FP32 --u FP64 --ur FP128 --n 100 --matrix random_orthogonal
```

They a python or shell script can run all of those automatically.

Precision sweeps should vary precision, condition number, and matrix type, and experiments should be scripted so rerunning them is convenient. In practice, that means your experiment system should let you rerun a whole table of experiments with one command, rather than manually recompiling/editing files.

1.2 HDNUM, CPFloat and GMP

HDNUM is included as a git submodule:

```
git submodule add https://parcomp-git.iwr.uni-heidelberg.de/Teaching/hdnum.git code/external/
git submodule update --init --recursive
```

CPFloat is cloned and built inside the HDNUM top-level directory, matching HDNUM's expected layout:

```
cd code/external/hdnum
git clone https://github.com/north-numerical-computing/cpfloat.git
cd cpfloat
make lib
```

GMP is installed system-wide:

```
sudo apt install libgmp-dev
```

Because CPFloat sits inside the HDNUM submodule but is not part of HDNUM itself, untracked files are hidden locally only for that submodule:

```
cd code/external/hdnum
git config status.showUntrackedFiles no
```

and the parent repository ignores untracked content specifically inside the HDNUM submodule:

```
git config submodule.code/external/hdnum.ignore untracked
```

1.3 Changing Top-level `CMakeLists.txt`

Currently the implementations are header-only and are contained in the folder include. Later if source files that not only header are added e.g. to a src/ folder it has to be accounted for:

Right now, this part:

```
add_library(mixed_precision_core INTERFACE)
```

is only an `INTERFACE` target. That means it does not compile any `.cc` files. It only carries usage requirements:

```
include directories
compile definitions
linked libraries
compile features
```

So it works well for a header-only project, but it does not yet build files from `src/`.

But if later you add:

```
src/csv_writer.cc
src/matrix_generators.cc
src/experiment_io.cc
```

then those files will not automatically be compiled unless we update CMake.

1.3.1 How to integrate `src/` later

The usual clean solution is to change from a purely `INTERFACE` core target to a real library target.

For example:

```
add_library(mixed_precision_core
    src/csv_writer.cc
    src/matrix_generators.cc
    src/experiment_io.cc
)
```

Then keep the same include directories and dependencies:

```
target_include_directories(mixed_precision_core
    PUBLIC
        "${CMAKE_CURRENT_SOURCE_DIR}/include"
        "${HDNUM_DIR}"
        "${CPFLOAT_INCLUDE_DIR}"
)

target_compile_definitions(mixed_precision_core
    PUBLIC
        HDNUM_HAS_GMP=1
        HDNUM_HAS_CPFLOAT=1
)

target_link_libraries(mixed_precision_core
    PUBLIC
        "${CPFLOAT_LIBRARY}"
        gmpxx
        gmp
)

target_compile_features(mixed_precision_core
    PUBLIC
        cxx_std_17
)
```

Then all your examples, tests, and experiments can continue doing:

```
target_link_libraries(my_target
    PRIVATE
        mixed_precision_core
        mixed_precision_warnings
)
```

and they will automatically get both:

```
the headers from include/
the compiled implementation files from src/
```

1.3.2 A flexible version

If you do not want to manually list every source file, you can use globbing:

```
file(GLOB project_sources CONFIGURE_DEPENDS
    "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cc"
    "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp"
)

add_library(mixed_precision_core
    ${project_sources}
)
```

But there is one small issue: `add_library()` with an empty source list can be awkward in some CMake versions/workflows.

So for now, while `src/` is empty, I would keep the current `INTERFACE` version.

Later, when you add the first real `.cc` file, change:

```
add_library(mixed_precision_core INTERFACE)
```

to:

```
file(GLOB project_sources CONFIGURE_DEPENDS
    "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cc"
    "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp"
)

add_library(mixed_precision_core
```

```
    ${project_sources}
)
```

and replace `INTERFACE` with `PUBLIC` in the relevant `target_*` commands.

1.3.3 Rule of thumb

Use this now:

```
add_library(mixed_precision_core INTERFACE)
```

while the project is header-only.

Use this later:

```
add_library(mixed_precision_core ${project_sources})
```

once `src/` contains real compiled support code.

2 Project Summary

2.1 Abstract

This *Fortgeschrittenenpraktikum* project investigates mixed precision iterative refinement for solving linear systems

$$Ax = b.$$

The goal is to implement a three-precision LU-based method: the LU factorization and correction solves use low precision u_f, the solution updates use working precision u, and the residuals are computed in higher precision u_r. Starting from a low-precision LU solve, the algorithm repeatedly computes $r_i = b - Ax_i$, solves for a correction d_i, and updates x_i until $|d_i|/|x_i| < u$ or a maximum iteration count is reached.

The focus is empirical accuracy, not runtime, since low precisions such as fp8, fp16, and bfloat16 are simulated via CPFloat. The experiments will study convergence, forward/backward error, condition-number dependence, precision triples (u_f, u, u_r) residual precision, and comparison with a direct working-precision solve.

Summary:

- implement three-precision vaiant of **iterative refinement for linear systems**
- study it's accuracy

2.1.1 Problem

- A linear system $Ax = b$. **iterative refinement** improves an approximate solution by repeatedly computing the residual and solving a correction equation.
  - *LU* factorization in low precision: u_f
  - **working arithmetic** in 'middle' precision: u
  - residual computation in high precision: u_r

Note

Review the definition of **accuracy** of an algorithm

- precision triples (u_r, u, u_f) with $u_r \succ u \succ u_f$:
  - i.e. u_f low precisions like: `fp8`, `fp16`, `bfloat16`, come from CPFloat:
    * `FP8: CPFloat<4, 4>`
    * `bfloat16: CPFloat<8, 8>`
    * `FP16: CPFloat<11, 5>`
  - u is probably the native `FP64` i.e. `double` precision

Note

Study the `FP8`, `FP16` and `bfloat16` formats and how they relate to template parameters in `CPfloat<>`

2.1.2 Algorithm

Algorithm: Three-precision iterative refinement

```
1. Compute the LU factorization PA = LU in precision uf.
2. Solve LU x0 = P b in precision uf; cast x0 to precision u.
3. While not converged:
   4. Compute ri = b - A xi in precision ur.
   5. Cast ri to precision u.
   6. Solve LU di = P ri in precision uf; cast di to precision u.
   7. Update xi+1 = xi + di in precision u.
```

- convergence condition: $\frac{\|d_i\|}{\|x_i\|} < u$ where u is the unit roundoff of the working precision,
- or stop after a max number of iterations.

Note

why is $\frac{\|d_i\|}{\|x_i\|} < u$ the convergence condition?

2.1.3 HDNUM & Own Implementation

relevant parts:

matrix and vector classes:

- `src/densematrix.hh`: `DenseMatrix<T>` with:
  - `mv`: matrix vector product
  - `mm`: matrix matrix product
  - `transpose()` etc
- `src/vector.hh`: `Vector<T>` with vector arithmetic operations (multiplication with scalar, addition) and norms
- LU decomposition in `src/lr.hh`:
  - `lr_fullpivot(A, p, q)`
  - `lr_partialpivot(A, p)`
  - triangular solves: `solveL(A, x, b)`, `solveR(A, x, b)`
  - `permute_forward` and `permute_backward`

2.1.4 Mixing Types

In a mixed precision algorithm different parts of the algorithm are instantiated / computed with different precisions, i.e. different types:

- LU factorization is done in lower precision u_f: E.g. `DenseMatrix<FP16>`
- Residual computation in higher precision u_r: E.g. `DenseMatrix<FP128>`

Therefore we must be able to convert from one type (class) like `DenseMatrix<FP16` to another type (class) like `DenseMatrix<FP128`. This conversion requires explicit construction or element-wise copying.

Managing type conversions between precisions is they key challenge. It is suggested to do this with a explicit template functions for matrices and vectors:

```
template<class T_out, class T_in>
void convert(Vector<T_out>& out, const Vector<T_in>& in)

template<class T_out, class T_in>
void convert(DenseMatrix<T_out>& out, const Vector<T_in>& in)
```

2.1.5 Methodology

- **reference solutions**: run a standard solver for $A \cdot x = b$ in `FP256` and treat the result x_{ex} as exact.
- **error metrics**:
  - report errors using norms appropriate to your algorithm

- compute **all error measures** in *a precision higher than the working precision of your algorithm*, i.e. u
- otherwise you are measuring the precision of your measurement.

> **Note**
>
> - What does **error** mean in this context? $\|x^* - x_{ex}\|_2$?
> - What is a norm that is **apropriate** to LU, simply $\|\bullet\|_2$?
> - Does Compute error measures in a precision higher than u simply mean that we compute $\|\cdot\|$ in something like `FP128`
>   - what does "otherwise you are measuring the precision of your measurement" mean?

- experiments should be scripted so that re-running them with various precision, condition number and matrix type is convenient.

2.1.6 What Should be Studied

- **Convergence Histories**: for each precision combination (u_f, u, u_r) and several test matrices A_i with different condition numbers $k(A_i)$:
  - plot **forward error** as a function of i
  - plot **backward error** as a function of i

> **Note**
>
> review the definitions of **forward error** and **backward error**

- **Summary Across Condition Numbers**: for a wide range of $k(A)$ for each precision combinations
  - record:
    - the final achieved **forward error**
    - number of iterations
  - plot these as a function of $k(A)$: theory predicts convergence requires roughly $k(A) \cdot u_f < 1$.
- **The Role** u_r: Fix (u_f, u) and vary u_r. When does $u_r > u$ make a measurable difference

> ℹ Note
>
> - Normally $u_r < u$, i.e. the residual computation is done in a precision lower than the working precision.
> - What happens when we increase it, when does it make a difference?

2.1.7 Plan

1. Implement hdnum_conversions.hpp.
2. Add conversion tests.
3. Implement a minimal mixed_ir.hpp.
4. First test with all precisions set to FP64.
5. Then test real mixed precision triples.
6. Add systematic experiments and plotting scripts.

3 Conversions

3.1 Floating Point Types

In this project there are three groups of number formats:

1. simulated low precision via CPFloat
2. native C++ hardware types `float` and `double`
3. high / arbitrary precision via GMP

The low precision `fp8`, `fp16`, `bfloat16` are simulated via CPFloat.

An overview of types defined in HDNUM:

HDNUM / project type	Provider	Parameters	Corresponds to in Higham–Mary	Unit roundoff, roughly
`FP8`	CPFloat	`CPFloat<4,4>`	custom 8-bit / "quarter precision"-like format	$2^{-4} \approx 6.25 \cdot 10^{-2}$
`bfloat16`	CPFloat	`CPFloat<8,8>`	Higham–Mary `bfloat16`	$2^{-8} \approx 3.91 \cdot 10^{-3}$
`FP16`	CPFloat	`CPFloat<11,5>`	IEEE half precision, Higham–Mary `fp16`	$2^{-11} \approx 4.88 \cdot 10^{-4}$
`FP32`	native C++ `float`	effectively `(24,8)`	IEEE single precision, Higham–Mary `fp32`	$2^{-24} \approx 5.96 \cdot 10^{-8}$
`FP64`	native C++ `double`	effectively `(53,11)`	IEEE double precision, Higham–Mary `fp64`	$2^{-53} \approx 1.11 \cdot 10^{-16}$
`FP128`	GMP	approx. 128-bit, `FP<64>`	software high precision / "quadruple-like" role, not necessarily IEEE `fp128`	depends on HDNUM wrapper
`FP256`	GMP	approx. 256-bit, `FP<192>`	multiprecision reference precision	depends on chosen GMP precision

HDNUM / project type	Provider	Parameters	Corresponds to in Higham–Mary	Unit roundoff, roughly
`FP512`, `FP1024`	GMP	approx. 512 / 1024-bit	multiprecision, beyond Higham–Mary's standard table	depends on chosen GMP precision

As a reminder the three-precision LU-IR algorithm is:

```
1. Compute the LU factorization PA = LU in precision uf.
2. Solve LU x0 = P b in precision uf; cast x0 to precision u.
3. While not converged:
   4. Compute ri = b - A xi in precision ur.
   5. Cast ri to precision u.
   6. Solve LU di = P ri in precision uf; cast di to precision u.
   7. Update xi+1 = xi + di in precision u.
```

i.e.:

1) Compute LU in u_f
2) Solve the initial system in u_f
3) Compute residuals in u_r
4) Cast residuals to u
5) Solve correction equations using the low-precision LU factors in u_f
6) update in u

For the algorithm the intended mapping is:

Algorithm role	Meaning	Typical project choices
u_f	factorization precision; low precision used for LU	`FP8`, `bfloat16`, `FP16`, sometimes `FP32`
u	working precision; stores A, b, x_i and updates solution	usually `FP32` or `FP64`
u_r	residual precision; computes $r_i = b - Ax_i$ accurately	`FP64`, `FP128`, maybe `FP256` for experiments

A reasonable set of experiments could be:

Experiment	u_f	u	u_r
baseline sanity check	`FP64`	`FP64`	`FP64`
classical mixed precision	`FP32`	`FP64`	`FP64`
half/double refinement	`FP16`	`FP64`	`FP64`
half/double with higher residual	`FP16`	`FP64`	`FP128`
bfloat comparison	`bfloat16`	`FP64`	`FP64`
very low precision stress test	`FP8`	`FP64`	`FP128`

3.2 Type Conversions

Again, the precisions for the algorithm are:

1) u_f: factorization / solve precision (low)
2) u: working precision (medium)
3) u_r: residual precision (high)

LU is computed in u_f, the initial solution is cast to u, the residual is computed in u_r and cast back to u, the correction solve uses the low-precision LU factors, and the correction is cast to u before updating.

So the practically necessary conversions are:

Conversion		
$u \to u_f$		create low-precision copies of A, b, or residuals for LU / triangular solves.
$u_f \to u$		bring x_0 and correction d_i back to working precision.
$u \to u_r$		compute $r_i = b - Ax_i$ in high precision.
$u_r \to u$		The residual is cast back before solving the correction equation.
$u_f \leftrightarrow u_r$	not strictly needed	You normally convert through the working representation, or independently convert from the original A, b, x.

3.2.1 Implementing Conversions - Naive Approach

HDNum implements wrappers `FP` and `CPFloat` for the high-precision GMP and the low precision CPFloat types in the source files `highprec_gmp.hh` and `lowprec_cpfloat.hh`.

We are asked not to modify HDNum source files, instead implement explicit conversion functions:

```cpp
template<class T_out, class T_in>
void convert(Vector<T_out>& out, const Vector<T_in>& in);

template<class T_out, class T_in>
void convert(DenseMatrix<T_out>& out, const DenseMatrix<T_in>& in);
```

The approach is to assign to `out` element-wise in loops, where the values constructed from the corresponding elements of `in`. A naive approach is to do:

```cpp
#pragma once

#include <cstddef>
#include "hdnum.hh"

namespace mpir {

// Vector conversion: resizes output if needed.
template<class T_out, class T_in>
void convert(hdnum::Vector<T_out>& out,
             const hdnum::Vector<T_in>& in)
{
    if (out.size() != in.size()) {
        out.resize(in.size());
    }

    for (std::size_t i = 0; i < in.size(); ++i) {
        out[i] = T_out(in[i]);
    }
}

// Convenience wrapper: allocates a new vector.
template<class T_out, class T_in>
hdnum::Vector<T_out> convert_vector(const hdnum::Vector<T_in>& in)
{
    hdnum::Vector<T_out> out(in.size());
    convert(out, in);
    return out;
}


// Matrix conversion: output matrix must already have correct shape.
template<class T_out, class T_in>
```

```
void convert(hdnum::DenseMatrix<T_out>& out,
             const hdnum::DenseMatrix<T_in>& in)
{
    if (out.rowsize() != in.rowsize() || out.colsize() != in.colsize()) {
        throw std::invalid_argument(
            "mpir::convert(DenseMatrix): output matrix has wrong size"
        );
    }

    for (std::size_t i = 0; i < in.rowsize(); ++i) {
        for (std::size_t j = 0; j < in.colsize(); ++j) {
            out[i][j] = T_out(in[i][j]);
        }
    }
}

// Convenience wrapper: allocates a new matrix.
template<class T_out, class T_in>
hdnum::DenseMatrix<T_out> convert_matrix(const hdnum::DenseMatrix<T_in>& in)
{
    hdnum::DenseMatrix<T_out> out(in.rowsize(), in.colsize());
    convert(out, in);
    return out;
}
```

Where the type conversion is done via the assignment:

```
out[i] = T_out(in[i])
```

for each element.

But this doesn't work for all pairs of `T_in, T_out` because not all the necessary constructors or conversion operators are implemented in the classes `CPFloat` and `FP`

Which Conversions Work which Don't ?

`FP` provides the constructors:

- `FP()`
- `FP(const double &)`
- `FP(const float &)`
- `FP(const int &)`

- FP(const mpf_class &)
- FP<mm>(const FP<mm>&)

So the conversion:

- double => GMP

works

CPFloat provides the constructors:

- CPFloat()
- CPFloat(const double &)
- CPFloat<mm, ee>(const double &)

So:

- double => CPFloat

works

CPFloat also provides the conversion operator operator double() const. Therefore

- CPFloat => double

works as well.

Because CPFloat can be converted to double and GMP can be constructed from a double:

- CPFloat => GMP

works too, by the implicit conversion of CPFloat to a double

But since the FP class doesn't provide a double() operator the conversion

- GMP => double

doesn't work.

Also CPFloat doesn't implemenet a constructor that accepts a GMP type, (or also because FP doesn't provide a double() operator) the conversion:

- GMP => CPFloat

doesn't work.

We summarize this with the following diagram and table:

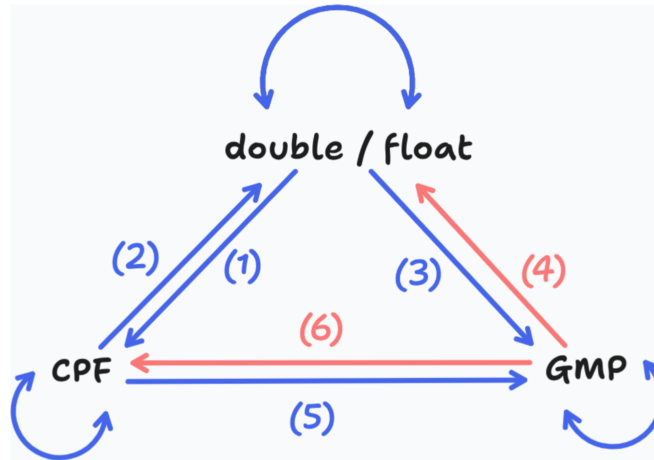

Figure 3.1: conversion triangle

Table 3.5: conversion table

Nr	Conversion	Works ?	Why ?
(1)	`double` ⇒ `CPFloat`	Yes	`CPFloat(const double &)`
(2)	`CPFloat` ⇒ `double`	Yes	`CPFloat::double()`
(3)	`double` ⇒ GMP	Yes	`FP(const double &)`
(4)	GMP ⇒ `double`	**No**	No `FP::double()`
(5)	`CPFloat` ⇒ GMP	Yes	`CPFloat` is converted to `double` implicitly
(6)	GMP ⇒ `CPFloat`	**No**	No `FP(const CPFloat &)` or no `FP::double()`

So if the conversion operator `FP::double()` would be implemented both **(4)** and **(6)** would work. (6) would work too because GMP type would be implicitly converted to a double and a `CPFloat` would be constructed. But more explicitly a constructor `FP(const CPFloat &)` can be implemented too.

But since we are initially not allowed to modify HDNUM source code we take a different approach.

3.2.2 Structure of FP and CPFloat Classes

FP
-mpf_class number
+FP() +FP(const double &) +FP(const mpf_class &) +FP(...) +getNumber() : mpf_class

CPFloat
-double number
+CPFloat() +CPFloat(const double &) +CPFLoat(...) +operator double() +getNumber() : double

- `FP`:
  - `FP` has a single private member variable `number` that is an GMP type, i.e. `mpf_class`.
  - it can be accessed with the public getter `const getNumber() : mpf_class`
- `CPFloat`:
  - `CPFloat` has a single private member variable `number` of type `double`.
  - it can be accessed with the public getter `const getNumber() : double`

> **ℹ Note**
>
> There's a mistake in HDNUM `highprec_gmp.hh` source file. The member variable `number` is documented as:
>
> **Listing 3.1** `highprec_gmp.hh`
>
> ```
> private:
>     // stores a double to represent the low precision number
>     mpf_class number;
> ```
>
> The comment `// stores a double to represent the low precision number` is wrong, and is probably copied over by mistake from `lowprec_cpfloat.hh`:

Listing 3.2 `lowprec_cpfloat.hh`

```
private:
    // stores a double to represent the low precision number
    double number;
```

Instead it should be something like:

```
// stores an mpf_class to represent the high precision GMP number
```

With the above interface, given a GMP number, the double representation of the number can be obtained with:

Listing 3.3 get_d() method

```
hdnum::FP128 fp128_number(1.2345);
auto d = fp128_number.getNumber().get_d(); //d is double
```

Where we use the `.get_d()` method of `mpf_class` to get the `double` representation of the GMP number.

3.2.3 Implementing Conversions - Correct Approach

The main issue is that

```
out[i] = T_out(in[i]);
```

does not work uniformly for all type pairs. We replace this direct construction step with:

```
out[i] = scalar_convert<T_out>(in[i]);
```

This selects the appropriate conversion path at compile time, depending on the input and output types using standard library type traits and `if constexpr`

As summarized in Figure 3.1 and Table 3.5, the only conversions that do not work by direct construction are:

1. `FP` ⇒ `double`,
2. `FP` ⇒ `CPFloat`

All other relevant cases can be handled by direct construction of the form:

```
T_out(x) // where x has type T_in
```

In C++, we can check at compile time whether an object of type `T_out` can be constructed from an argument type `const T_in&` using the standard library type trait:

```
std::is_constructible_v<T_out, const T_in&>
```

This expression evaluates to `true` if such a construction is valid, and to `false` otherwise.

It can be verified that this indeed evaluates to `false` only for the conversion pairs:

- `T_out == double`, `T_in == FP`
- `T_out == CPFloat`, `T_in == FP`

Then, the initial part of `scalar_cast` can be written as:

```
template<class T_out, class T_in>
T_out scalar_cast(const T_in& x)
{
    if constexpr (std::is_constructible_v<T_out, const T_in&>) {
        return T_out(x); // we use the direct construction
    }
    else { //remaining cases: FP => double || FP => CPFloat
        ...
    }
}
```

Logically, we know that the only possible cases in the `else` branch are:

- `T_out == double`, `T_in == FP`
- `T_out == CPFloat`, `T_in == FP`

In both cases `T_in` is an `FP` type. as we've seen in Listing 3.3 the `double` representation of an `FP` number `x` can be extracted with:

```
x.getNumber().get_d()
```

Since both `CPFloat` and `double` can be constructed from a `double`, we could simply write:

```
template<class T_out, class T_in>
T_out scalar_cast(const T_in& x)
{
    if constexpr (std::is_constructible_v<T_out, const T_in&>) {
        return T_out(x); // we use the direct construction
    }
    else { //remaining cases: FP => double || FP => CPFloat
        return T_out(x.getNumber().get_d());
    }
}
```

However, this version is not robust enough because the `else` branch relies on our prior reasoning that only the two exceptional cases can reach it. The code itself does not express this assumption. If `scalar_cast` is later instantiated with another unsupported conversion pair, the compiler will still try to compile

```
x.getNumber().get_d()
```

even if `T_in` is not an `FP` type. This should lead to implementation-dependent and potentially confusing error message.

Therefore, it is better to make the remaining cases explicit using another `if constexpr` condition:

```
template<class T_out, class T_in>
T_out scalar_cast(const T_in& x)
{
    if constexpr (std::is_constructible_v<T_out, const T_in&>) {
        return T_out(x); // direct construction works
    }
    else if constexpr (
        is_hdnum_fp_v<T_in> &&
        std::is_constructible_v<T_out, double>
    ) {
        return T_out(x.getNumber().get_d());
    }
    else {
        static_assert(always_false<T_out, T_in>::value,
                      "unsupported scalar conversion");
    }
}
```

Here, the helper trait `is_hdnum_fp_v<T>` detects whether a type `T` is one of HDNUM's GMP-based high precision numbe types `hdnum::FP<m>`. This is a compile-time Boolean value, and analogous to the standard library traits such as `std::is_constructible_v`.

The implementation, given below, follows a common template metaprogramming partial specialization pattern:

```
// Detect HDNUM's GMP wrapper type hdnum::FP<m>
template<class T>
struct is_hdnum_fp_impl : std::false_type {};

template<int m>
struct is_hdnum_fp_impl<hdnum::FP<m>> : std::true_type {};

template<class T>
inline constexpr bool is_hdnum_fp_v =
    is_hdnum_fp_impl<
        std::remove_cv_t<std::remove_reference_t<T>>
    >::value;
```

1) The first part

   ```
   template<class T>
   struct is_hdnum_fp_impl : std::false_type {};
   ```

   is the default case and says:

   > for an arbitrary type `T`, it is false

   So:

   ```
   is_hdnum_fp_impl<double>::value       // false
   is_hdnum_fp_impl<float>::value        // false
   is_hdnum_fp_impl<hdnum::FP16>::value  // false, because CPFloat, not GMP FP
   ```

2) The second part

   ```
   template<int m>
   struct is_hdnum_fp_impl<hdnum::FP<m>> : std::true_type {};
   ```

   is the specialization case, and says:

   > if the type has the form `hdnum::FP<m>`, it is true.

   So:

```
is_hdnum_fp_impl<hdnum::FP128>::value  // true
is_hdnum_fp_impl<hdnum::FP256>::value  // true
```

3) The last part

```
template<class T>
inline constexpr bool is_hdnum_fp_v =
 is_hdnum_fp_impl<
     std::remove_cv_t<std::remove_reference_t<T>>
 >::value;
```

defines a compile-time boolean variable template. For example:

```
is_hdnum_fp_v<double>          // false
is_hdnum_fp_v<hdnum::FP128>   // true
```

It is analogous to standard library helpers such as:

```
std::is_arithmetic_v<T>
std::is_constructible_v<T, Args...>
std::is_convertible_v<From, To>
```

and allows to write

```
is_hdnum_fp_v<T>
```

instead of

```
is_hdnum_fp_impl<T>::value
```

The keyword `inline` is needed for variable that are defined in header files.

The use of:

```
std::remove_reference_t<T>
std::remove_cv_t<T>
```

makes the trait insensitive to references and `const` / `volatile` qualifiers. For example this way all of the following are treated as the same:

```
hdnum::FP<64>
const hdnum::FP<64>
hdnum::FP<64>&
const hdnum::FP<64>&
```

Without removing references and cv-qualifiers, the specialization would not match types such as `hdnum::FP<m>&`

The final branch of `scalar_cast`

```
else{
    static_assert(always_false<T_out, T_in>::value,
        "unsupported scalar conversion");
}
```

This branch is reached only if neither of the previous compile-time conditions applies:

```
std::is_constructible_v<T_out, const T_in&>
```

is `false`, so direct construction does not work, and

```
is_hdnum_fp_v<T_in> && std::is_constructible_v<T_out, double>
```

is also `false`, so the special `FP => double` / `FP => CPFloat` conversion path doesn't apply either.

Therefore, if the compiler reaches this final 'else' branch, the requested scalar conversion is unsupported. In that case, we want compilation to fail with a clear error message.

The helper trait is defined as:

```
template<class...>
struct always_false : std::false_type {};
```

This defines a type trait that is always `false`, regardless of the template arguments. For example:

```
always_false<int>::value                  // false
always_false<double, int>::value          // false
always_false<hdnum::FP<64>, double>::value // false
```

The `class...` syntax denotes a template parameter pack and means that `always_false` can accept any number of template type arguments. In our case we use two arguments:

```
always_false<T_out, T_in>::value
```

It is important that `always_false<T_out, T_in>::value` depends on template parameters `T_out` and `T_in`, as it lets us create a compile-time `false` value that is evaluated when the templates are instantiated and not when the function is parsed. Omitting and writing simply `static_assert(false, ...)` would cause the assertion to be checked when the function template is parsed, and not only when the template is instantiated and the final branch is actually selected for the concrete instantiation.

`scalar_cast` Final Version

```cpp
#pragma once

#include <cstddef>
#include <stdexcept>
#include <type_traits>

#include "hdnum.hh"

namespace mpir {

// Helper for dependent static_assert(false)
template<class...>
struct always_false : std::false_type {};


// Detect HDNUM's GMP wrapper type hdnum::FP<m>
template<class T>
struct is_hdnum_fp_impl : std::false_type {};

template<int m>
struct is_hdnum_fp_impl<hdnum::FP<m>> : std::true_type {};

template<class T>
inline constexpr bool is_hdnum_fp_v =
    is_hdnum_fp_impl<
        std::remove_cv_t<std::remove_reference_t<T>>
    >::value;


// Scalar conversion
template<class T_out, class T_in>
T_out scalar_cast(const T_in& x)
{
    if constexpr (std::is_constructible_v<T_out, const T_in&>) {
        return T_out(x);
    }
    else if constexpr (
        is_hdnum_fp_v<T_in> &&
        std::is_constructible_v<T_out, double>
    ) {
```

```
        return T_out(x.getNumber().get_d());
    }
    else {
        static_assert(always_false<T_out, T_in>::value,
                      "mpir::scalar_cast: unsupported scalar conversion");
    }
}


// Vector conversion
template<class T_out, class T_in>
void convert(hdnum::Vector<T_out>& out,
             const hdnum::Vector<T_in>& in)
{
    if (out.size() != in.size()) {
        out.resize(in.size());
    }

    for (std::size_t i = 0; i < in.size(); ++i) {
        out[i] = scalar_cast<T_out>(in[i]);
    }
}




// DenseMatrix conversion
template<class T_out, class T_in>
void convert(hdnum::DenseMatrix<T_out>& out,
             const hdnum::DenseMatrix<T_in>& in)
{
    if (out.rowsize() != in.rowsize() || out.colsize() != in.colsize()) {
        throw std::invalid_argument(
            "mpir::convert(DenseMatrix): output matrix has wrong size"
        );
    }

    for (std::size_t i = 0; i < in.rowsize(); ++i) {
        for (std::size_t j = 0; j < in.colsize(); ++j) {
            out[i][j] = scalar_cast<T_out>(in[i][j]);
        }
    }
```

```
}


// convenience function
template<class T_out, class T_in>
hdnum::DenseMatrix<T_out> convert_matrix(const hdnum::DenseMatrix<T_in>& in)
{
    hdnum::DenseMatrix<T_out> out(in.rowsize(), in.colsize());
    convert(out, in);
    return out;
}

// convenience function
template<class T_out, class T_in>
hdnum::Vector<T_out> convert_vector(const hdnum::Vector<T_in>& in)
{
    hdnum::Vector<T_out> out(in.size());
    convert(out, in);
    return out;
}

} // namespace mpir
```

4 Report

4.1 This is some title