The *Compensated* Library Documentation

Quick links

Introduction

Why *Compensated*?

The idea behind *Compensated* is to provided a flexible and easy to use library for compensanted Kahan and Kahan–Neumaier addition without introducing any external dependencies. The code is generic and can be used both with standard and with user-defined types.

Why is Kahan–Neumaier addition needed?

Compensated addition is needed whenever a large number of floating point values are accumulated or when very small values are added to very large values. Typical applications include numerical integration.

Consider this code:

```cpp
const double big   = 1.0e30;
const double small = 1.0e-30;

double result = big + small - big - small; // Should be zero
std::cout << "Result: " << result << std::endl;
```

which prints `Result: -1e-30` on x86-64 with 64-bit doubles, even though, at least mathematically, we would expect the output to be `Result: 0.0`.

The reason for this loss of precision is that the [IEEE-754] standard doubles contain 53 mantissa bits, which is not enough to represent the number `big + small` exactly. After the extra bits are killed off by rounding, the temporary value `big + small` ends up being equal to just `big`. Subsequent subtraction of `big` brings the value back to zero, and then we finally subtract `small`, so the result is actually equal to `-small`. Obviously, errors of this kind can accumulate further if we perform more additions.

The Kahan–Neumaier running compensation algorithm [N74] prevents this from happening. In fact, no matter how many values are added, the numerical error from the entire summation is equal to the error from just a single addition.

Of course, there is a price to pay for this superior precision: the performance is a few times slower, but this is often more acceptable than the uncontrolled accumulation of errors that could otherwise occur.

Simple example of basic usage

Before we get into the details, let's look at some code using *Compensated*:

```cpp
#include <iostream>
#include "compensated/compensated.h"

const double big   = 1.0e30;
const double small = 1.0e-30;

int main()
{
    compensated::value test{big};
    test += small;
    test -= big;
    test -= small;

    double result = test; // Convert back to double
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

In addition to the operators += and −= used in the example, the `compensated::value` class provides operators: +, −, equality comparisons, and many more features. You can convert between floating point types and `compensated::value` objects.

One thing to keep in mind is that the `compensated::value` constructor is declared as **`explicit`**, so the initialization must mention the class name explicitly, as in the first line of `main()` above.

The file 'examples.cpp'

For people who prefer reading code over reading documentations, there is a very simple demo program included in the repository as `example/example.cpp`.

Requirements

What is required for using *Compensated*?

- A compiler supporting C++20. This may require setting the flag `-std=c++20`.
- You **must not** compile *Compensated* with "fastmath" or "unsafe math optimizations" enabled. These features break the numerical accuracy brought about by *Compensated*.

Dependencies needed to run tests

The *Compensated* library comes with unit tests which use the GoogleTest framework. If you wish to build and run the tests, you need to have GoogleTest installed. However, this is not needed for simply using the *Compensated* library in your project.

What is *not* needed

The *Compensated* library does not use exceptions. Therefore, you can compile it with **exceptions** and **rtti** disabled.

Also, as *Compensated* consists of headers only, there are no link-time dependencies.

With cmake

Since *Compensated* is a header-only library, you can install it without building anything. Nevertheless, the repository comes configured for use with `cmake`, which can build the **tests** and the **example program.**

In order to download and configure these builds, you will typically run:

```
git clone https://github.com/S-Rafael/compensated
cd compensated
mkdir build
cd build
cmake ..
```

If you wish, you can then build the tests and the example app by running `make`.

To install the *Compensated* header to your include path, type

```
sudo make install
```

(on a sudo-enabled system).

If you wish to build and run tests, run:

```
make check
```

Manual install

Alternatively, you may ignore the `cmake` system entirely and install *Compensated* by copying the following files to wherever they need to be:

```
compensated/
├── compensated.h
└── LICENSE
```

Provided that you comply with the license terms, you may include the *Compensated* library directly into the source tree of your project.

If you distribute your program with the source code, under an open-source license, you can distribute *Compensated* with it. We kindly ask you to keep *Compensated* in a separate directory, together with its `LICENSE`.

Regardless of whether your program is open-source or closed-source, we ask you to mention the use of *Compensated* in your program's copyright notice and credits.

Glossary of terms

- **raw value type** – refers to the type of the underlying value on which we want to perform compensated addition and subtraction.
- **admissible type** – refers to a raw value type which is admissible for Kahan/Kahan–Neumaier summation. Admissible raw value types satisfy the constraint `kahanizable`.

  In order for a raw value type T to be admissible, it must satisfy the following conditions:

  1. T must be no-throw copy assignable.
  2. T must be assignable from the literal `0`, so that the following code compiles:

     ```
     T variable = 0;
     ```

     For example, there has to be a constructor for T which takes a single argument of a numerical type. This condition is satisfied by standard types such as `float`, `double`, `long double`, `std::complex<float>`, `std::complex<double>`, etc.
  3. There must be addition and subtraction operators defined for type T, which both return something convertible to T. In other words, the following code must compile:

     ```
     T a, b, c, d;
     c = a + b;
     d = a - b;
     ```

     In practice, this means that your custom classes need to define suitable `operator+` and `operator-` members in order for them to work with *Compensated*.
- **real type** – refers to an admissible raw value type T which "represents a real number", i.e., it satisfies the following conditions:
  1. T is three-way comparable.
  2. Either T has a public member function `T::abs(void)` or it can be fed into an overload of `std::abs()`. The return type of one of these functions must be three-way comparable.
- **complex type** – refers to a type T which "represents a complex number", i.e., it satisfies the following conditions:
  1. T has public member functions `T::real(void)` and `T::imag(void)` which both return a **real type** for which there is an overload of `std::abs()`.
  2. T has a constructor which accepts two arguments of the types returned by `T::real(void)` and `T::imag(void)`. It is expected, but not enforced, that these constructor arguments represent the real and imaginary parts of a complex number, so that we always have:

     ```
     z == T(z.real(), z.imag())
     ```
- **general type** – refers to an admissible raw value type which is neither a real type nor a complex type.

Basic idea of *Compensated*

The header file `compensated.h` defines a class template `compensated::value<T>` which wraps around a raw value of type `T` and provides compensated addition and subtraction for these values.

In order to instantiate the template for a raw value type `T`, the type `T` must satisfy the constraint `compensated::kahanizable<T>` which says that `T` is an admissible type.

You can then operate on the `compensated::value<T>` objects by using addition and subtraction operators, both between such objects as well as between them and raw values.

Kahan's vs Neumaier's addition

The type of compensated addition available for a raw value type `T` depends on whether `T` is real, complex, or general:

- For **real** and **complex** types, the fully compensated Kahan–Neumaier addition algorithm is implemented. In the case of complex types, the running compensations are calculated independently for the real and imaginary parts. This means that maximum possible precision is retained regardless of the order of addition and the magnitudes of the real and imaginary parts.
- For **general** types, only the plain Kahan algorithm is available. This algorithm works well for accumulating a large number of small values or (equivalently) for adding a small value to a large one. Nonetheless, Kahan's algorithm is sensitive to the order of addition:

$$\begin{aligned} \text{large} + \text{small} &= \text{precise result} \\ \text{small} + \text{large} &= \text{potential loss of precision} \end{aligned}$$

If you wish to have the full Kahan–Neumaier algorithm for your custom type, consider using `compensated::value<T>` objects as your data members and choose the type `T` to be either real or complex, for example:

```cpp
class point_2D {
    compensated::value<double> x, y;
public:
    inline point_2D operator+ (const point_2D& other) const {
        point_2D result;
        result.x = x + other.x;
        result.y = y + other.y;
        return result;
    }
};
```

Library reference

Namespace compensated

The *Compensated* library introduces the namespace `compensated` which contains all identifiers introduced by the library. Whenever we refer to a type, a global identifier or a concept introduced by *Compensated*, it is understood that it belongs to the namespace `compensated`.

The class `compensated::value`

Class declaration:

```
template<kahanizable T> class value {...};
```

The template parameter T describes the raw value type for the class `value`. The type T must satisfy the constraint `kahanizable`, which says that T is an admissible value type.

Objects of class `compensated::value` are trivially copiable and movable.

Note

From now on, the symbol T will always refer to the admissible raw value type which was used as the template parameter in the respective instantiation of the `compensated::value` template class.

Default constructor

```
constexpr value( ) = default;
```

This trivial constructor default-initializes the private data members which are of the raw value type. Since there are no constructor arguments, the raw value type cannot be deduced. Thus, you must pass a template argument when using default-initialization, for example:

```
compensated::value<double> my_variable;
```

Warning

If the raw value type is not default-constructible, then you are not allowed to use the default constructor of `compensated::value`. Use the one-argument constructor instead.

Constructor from raw value type

```
explicit constexpr value(const T& initial_value);
```

`initial_value` – the raw value with which the object will be initialized.

After this type of initialization, the resulting `compensated::value` object will represent the `initial_value`. Since this is the only one-argument constructor, the template parameter (the type T) can be deduced, for example:

```
using CC = std::complex<double>;
CC z(42.0, -1.0);
compensated::value cz{z};
```

Assignment from raw value type

```
void value::operator= (T raw);
```

Assigns the raw value `raw` to the `compensated::value` object. The previous state of the object is reset, and after the assignment the object represents the new value `raw`.

Conversion to raw value type

```
constexpr operator value::T() const;
```

Returns a raw value which best represents the mathematical quantity currently stored in the `compensated::value` object. Note that this conversion can be performed implicitly.

Warning

Conversion to the raw value type may cause a loss of precision, so it should be avoided, except at the very end of summation, when we want to get the result out. In order to get an estimate of the error from the conversion, use value::error().

value::error()

```
T value::error(void) const;
```

Returns an element of the raw value type which best represents the numerical error from conversion to the raw value type if such conversion were to be made now.

When the raw value type T is **real** or **complex**, then the magnitude of the returned value is a faithful representation of the magnitude of the actual error from conversion at the present time.

When the raw value type T is **general**, the estimate is merely *best effort* and may not be accurate.

value::imag()

```
constexpr auto value::imag(void) const
requires is_complex<T>;
```

When the raw value type T is complex, this function returns the imaginary part of the underlying raw value. The return type is the same as the return type of `T::imag()`.

value::real()

```
constexpr auto value::real(void) const
requires is_complex<T>;
```

When the raw value type T is complex, this function returns the real part of the underlying raw value. The return type is the same as the return type of `T::real()`.

Algebraic operators for `compensated::value`

This section describes mathematical operators involving the class `compensated::value` and providing the Kahan/Kahan–Neumaier compensated addition and subtraction.

Unary minus

```
constexpr value value::operator- (void) const
```

Returns a `compensated::value` object which mathematically represents the *negative* of the given `value`.

Example:

```
float test = 42.0;
compensated::value v{test};
auto w = -v; // w represents -42.0
```

Addition of raw value

```
value<T> value::operator+ (const T& raw) const;
void value::operator+= (const T& raw);
template<kahanizable T> value<T> operator+(T raw, value<T> obj);
```

These three operators perform, respectively: compensated right addition, compensated addition-in-place, and compensated left addition, of a raw value to an object of class `compensated::value`.

Example:

```
compensated::value<double> obj{0.0}, other{0.0};
other = obj + 42.0;
other += 3.14;
obj = 77.7 + other;
```

Addition of two `compensated::value` objects

```
value<T> value::operator+ (const value<T>& other) const;
void operator+= (const value<T>& other);
```

These operators perform addition and addition-in-place between two objects of class `compensated::value`. These additions use the compensated algorithms as well.

Example:

```
compensated::value<double> a, b, c;
a = 42.0;
b = 69.0;
c = a + b;
b += a;
```

Subtraction of raw value

```
value<T> value::operator- (const T& to_subtract) const;
void value::operator-= (const T& to_subtract);
template<kahanizable T>
    value<T> operator-(T from_what, value<T> subtract_what);
```

These three operators perform, respectively: compensated right subtraction, compensated subtraction-in-place, and compensated left subtraction, of a raw value with an object of class `compensated::value`.

Example:

```
compensated::value<double> obj{0.0}, other{0.0};
other = obj - 42.0;
other -= 3.14;
obj = 77.7 - other;
```

Subtraction of two compensated::value objects

```
value<T> value::operator- (const value<T>& other) const;
void value::operator-= (const value<T>& other);
```

These operators perform subtraction and subtraction-in-place between two objects of class compensated::value. Like in the case of additions, the running compensation algorithms are used for these subtractions.

value::accumulate()

```
template<typename It>
requires is_iterator_to<It, T>
void value::accumulate(It begin, It end);
```

Accumulates a collection of raw values by iterating between the provided iterators. The iterators must satisfy the constraint is_iterator_to<It, T> which says that the values returned by “dereferencing” them are of type T. These values are added in-place to the present object.

The example below demonstrates the usage of value::accumulate() with iterators to an std::vector.

```
std::vector<double> v = {1.0, 2.0, 3.0, 4.0};
compensated::value<double> a{0.0};
a.accumulate(v.begin(), v.end());
std::cout << "a: " << a << std::endl; // Prints "a: 10"
```

Note

In the code snippet above, we passed the object a of type compensated::value<double> to the output operator std::ostream::operator<<. This trigers the implicit conversion to the raw value type double and calls std::ostream::operator<<(double).

Equality comparison operators

```
constexpr bool operator== (const T& value) const
    requires std::equality_comparable<T>;
constexpr bool operator== (const value<T>& other) const
    requires std::equality_comparable<T>;
template<kahanizable T>
    requires std::equality_comparable<T>
value<T> operator==(T raw, value<T> kn);
```

These three equality comparisons will be enabled whenever the raw value type `T` is equality-comparable. Note that these operators are non-trivial: they attempt to tell whether the values being compared represent the same mathematical quantity.

Note

Inequality operators `!=` are also available to the library user, even though they are not expressly declared in the source. Instead, *Compensated* relies on the overload resolution mechanisms specified by the C++20 standard, which mandate the consideration of rewritten candidates. In practice, the compiler calls the operators `==` discussed above and then flips the resulting `bool`.

License and author information

License terms

Acknowledgements

During the work on the *Compensated* library, Rafał Siejakowski was supported by grant #2018/12483-0 from the São Paulo Research Foundation (FAPESP). All opinions, assumptions and conclusions or recommendations expressed in this material are the responsibility of the author and do not necessarily reflect the point of view of the FAPESP.

Bibliography

Bibliographic references

IEEE
Standard for Floating-Point Arithmetic, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 84 pp., 2019

Neumaier, A.
Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen
Zeitschrift für Angewandte Mathematik und Mechanik. 54(1): 39–51, 1974