klassische Templates und C++20 Concepts

Ab C++20 können `Concept`-Namen verwendet werden, um einen Template-Typparameter einzuschränken, denn manche Algorithmen funktionieren nicht mit jedem Datentyp. Dafür gibt es 3 Varianten, die man auch kombinieren kann:

1) Direkt in der Template-Klausel **statt `typename`**
2) Eine `requires` Klausel direkt nach der Template-Klausel
3) Eine `requires` Klausel direkt vor dem Code-Block

```
template<std::integral T> auto f(T x)                    // Variante 1
{…}

template<typename T> requires std::integral<T>           // Variante 2
auto f(T x) {…}

template<typename T>                                     // Variante 3
T auto f(T x)  requires std::integral<T>
{…}
```

Version 1 ist die kürzeste Methode, die anderen 2 sind allerdings etwas flexibler. Wir wollen als Beispiel das Template für den Newtonalgorithmus `newton(f, x)` so einschränken, dass für `x` nur Gleitkommazahlen und für `f` nur auf `x` anwendbare Funktionen verwendet werden können. Diese Einschränkungen sind seit C++20 in der Datei `<concepts>` vordefiniert und heißen `std::invocable` und `std::floating_point`. Variante 1 ist hier nur schwer möglich, deshalb verwende ich eine Mischform:

```
template<typename Function, std::floating_point T>
     requires std::invocable<Function, T>
auto newton(Function f, T x) {…}
```

Die Verwendung von ausdrucksstarken Typplatzhalter (wie `Function`) verbessert die Lesbarkeit des Codes, ist aber für C++ ohne Bedeutung, es wird beim Aufruf dennoch jeder Argumenttyp akzeptiert. Nur durch Concepts lässt sich die Akzeptanz einschränken. So lassen sich unerlaubte Template-Aufrufe schon in der TAD-Phase erkennen. Das Template wird dann ignoriert (SFINAE!) und es wird keine Fehlermeldung ausgegeben. Findet C++ allerdings keine passende Alternative, kommt eine (einzige und verständlichere) Fehlermeldung wie: `Constraints not satisfied`