1 Testat-Übung: Matrix

1.1 Einführung

In dieser Übung werden Sie eine Container-Klasse schreiben, welche 2D-Matrizen representiert und diverse Operationen darauf erlaubt.

Das entsprechende Projekt ist bereits vorbereitet. Ebenfalls finden Sie die Interface-Definition im `src/matrix.h` Header. Sie sollen die entsprechend geforderte Funktionalität in `src/matrix.cpp` implementieren. Natürlich dürfen Sie jederzeit weitere Hilfsfunktionen hinzufügen. Die Signatur der definierten *öffentlichen* Methoden dürfen Sie nicht verändern.

Die dazu passenden Unit-Tests sind bereits implementiert und erlauben ihnen, ihre Implementation zu testen. Zudem existiert auch eine kleine Applikation, welche Sie zum Experimentieren verwenden können. Oder Sie schreiben zusätzliche Unit-Tests. Das verwendete Test-Framework Catch2 haben Sie bereits kennengelernt.

Öffnen Sie das `workspace`-Repo in CLion als Root-Projekt und navigieren Sie zu `uebungen/matrix`. Hier finden sie das vorbereitete Projekt. Aktivieren Sie es indem Sie in `uebungen/CMakeLists.txt` die folgende Zeile aktivieren:

```
1 add_subdirectory(matrix)
```

Die Aufgaben sind Schritt für Schritt abzuarbeiten. Bis auf den mit *optional* markierten Teil sind alle zwingend korrekt zu implementieren. Aktivieren Sie die einzelnen Tests/Signaturen, indem Sie die vorbereiteten `#defines` in `matrix.h` aktivieren.

Als Beispiel-Implementation oder zum Ausprobieren können Sie einen Online-Rechner verwenden wie z.B: matrixcalc.org

1.2 Testat

Geben Sie die geforderten Implementationen bis zum vereinbarten Datum ab. Senden Sie die Dateien `matrix.h` und `matrix.cpp` per Mail an christian.lang@fhnw.ch. Die Lösung muss einerseits alle Unit-Tests, anderseits auch bestimmte Qualitäts-Merkmale erfüllen. Diese Qualitätsmerkmale werden einerseits automatisch durch cpplint andererseits durch den Dozenten geprüft. Sie erhalten dementsprechend Feedback zu ihrer Abgabe.

1.3 Ziele

- Implementieren einer Container-Klasse für 2D-Matrizen.

- Implementieren von Matrix-Operationen.
- Performance-Optimierung mittels Cache-Awareness.
- Performance-Optimierung mittels Algorithmus.

2 Basis-Funktionen der Matrix-Klasse

In diesem ersten Teil sollen Sie die Grundfunktionen der Klasse implementieren. Achten Sie dabei auf die Inline-Dokumentation im Header. Diese ist mittels doxygen geschrieben, welches mit Javadoc vergleichbar ist. Allerdings ist doxygen nicht Teil der Sprachdefinition von C/C++ sondern nur eine der vielen Arten wie Code-Doku für C/C++ erstellt werden kann.

2.1 Konstruktoren und Destruktor

Bei der Implementation der Konstruktoren und des Destruktors ist Memory auf dem Heap zu allozieren und entsprechend wieder frei zu geben. Allozieren Sie das Memory als *eindimensionales* C-Array und initialisieren Sie die anderen Member-Variablen bevorzugt in der Initialisierungs-Liste. Ebenfalls können Sie bei Bedarf in der Initialisierungs-Liste andere Konstruktoren mittels `Matrix(...)` aufrufen. Sie sollten mittels Assertions sicherstellen dass keine ungültigen Matrizen erstellt werden können.

Tipp: in der gesamten Aufgabe sollten Sie nur ein einziges **`new`**-Statement benötigen.

2.2 Hilfs-Methoden

Die erste Hilfsmethode die Sie implementieren sollen ist der `Size()`-Getter für die Grösse der Matrix. Des weiteren sollen Sie eine `Data()`-Methode implementiern, welche Zugriff auf das interne Memory erlaubt. Einmal als veränderbare-Variante und einmal mit **`const`** markiert. Diese beiden Methoden werden vorallem in den Unit-Tests verwendet, könnten aber auch für die folgenden Implementierungen hilfreich sein.

2.3 `Set*`-Methoden

Die unterschiedlichen `Set*`-Methoden sind für die Initialisierung des kompletten Speichers auf dem Heap zuständig. Verwenden Sie für die Random-Initialisierung `uniform_real_distribution<` **`double`**> um Werte im Interval $[0, 1)$ zu generieren.

2.4 `Equals`-Methode

Die `Equals`-Methode hält einige Tücken bereit, da wir hier mit Floating-Point-Werten arbeiten. Entsprechend können Unschärfen bei den verfügbaren Zahlen entstehen, sollten die Werte genügend gross werden. Verwenden Sie für den Vergleich von **`double`**-Werten deshalb `Math::AlmostEquals` in `utils/math.h`.

3 Matrixmultiplikation

Die erste Operation welche Sie implementieren sollen ist die Matrix-Multiplikation. Aktivieren Sie dazu das `#define OPERATIONS_MULTIPLY` in `matrix.h`.

Die Multiplikation zweier Matrizen A und B, $R = A \times B$ ist wie folgt definiert: $r_{i,j} = \sum_{k=1}^{v} a_{i,k} b_{k,j}$, wobei A eine $u \times v$, B eine $v \times w$ und R eine $u \times w$ Matrix sind.

Zum Beispiel: $\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \end{pmatrix} \times \begin{pmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \end{pmatrix} = \begin{pmatrix} a_{0,0} \times b_{0,0} + a_{0,1} \times b_{1,0} + a_{0,2} \times b_{2,0} \\ a_{1,0} \times b_{0,0} + a_{1,1} \times b_{1,0} + a_{1,2} \times b_{2,0} \end{pmatrix} = \begin{pmatrix} r_{0,0} \\ r_{1,0} \end{pmatrix}$

Der Rechenaufwand für quadratische Matrizen der Seitenlänge n ist, wie aus der verwendeten Formel ersichtlich, relativ hoch und beträgt $O(n^3)$. Es existieren schnellere Algorithmen, wie beispielsweise der Strassen-Algorithmus, welcher mit etwa $O(n^{2.7})$ Operationen auskommt. Wir verwenden in dieser Aufgabe den langsameren Standardalgorithmus.

Da die nächsten Aufgaben auf dieser Multiplikation zweier Matrizen aufbauen werden, macht es Sinn die rohe Multiplikation in eine Hilfsfunktion auszulagern. Verwenden Sie dazu folgende Signatur:

```
1 static void MultiplyIntern(const double* a,
2                             const double* b,
3                             double* result,
4                             size_t a_vertical,
5                             size_t b_horizontal,
6                             size_t between);
```

wobei `between` die Dimension bezeichnet, welche beide Matrizen gemeinsam haben. In der Implementation von `Multiply` können Sie diese aufrufen und sich um Assertions, Allocations und Initialisierung kümmern. Denken Sie daran, dass hier und auch im Weiteren keinerlei Matrix-Instanzen auf dem Heap alloziert werden sollen.

4 Matrix-Potenzierung

Implementieren Sie nun die Methode `Power(size_t k)`, welche für ein $k > 1$ eine quadratische Matrix M potenziert, d.h. $(k-1)$-mal M mit sich selber multipliziert:

$$M^k = ((\Big(\Big(M \times M\Big) \times M\Big) \times M \times \dots).$$

Vermeiden Sie unnötige Speicherallokationen. Verwenden Sie auch hier `MultiplyIntern`.

5 Optimierung der Matrixmultiplikation

Als nächstes sollen Sie eine Performance-optimierte Version von `Multiply` implementieren. Sie brauchen dazu nicht den Algorithmus zu verändern sondern nur auf unnötige Allocations und Multiplikationen zu verzichten und Caching-Effekte auszunutzen. Stellen Sie sicher, dass der PC/Laptop am Netzteil hängt und sich nicht im Stromspar-Modus befindet. Zudem sollten Sie sicherstellen, dass im Bios alle Virtualisierungs-Technologien eingeschaltet sind wie etwa: VT-x und VT-d (Intel) oder AMD SVM und AMD-V.

Die entsprechenden Tests messen die Ausführungsdauer beider Varianten, vergleichen die Ergebnisse und testen die schnelle Version auf die erlaubte Ausführungszeit. Erstellen Sie hier eine neue Variante der internen Multiplikation: `MultiplyInternFast`. Darin sollen Sie die drei verschachtelten Loops so anpassen, dass der Cache optimal ausgenutzt werden kann. Zudem sollten Sie Teile der Index-Berechnungen möglichst in die äusseren Loops verlagern, um die teuren Multiplikationen nicht unnötig mehrfach zu machen.

Des weiteren ist ind er Signatur von `MultiplyInternFast` bereits eine Optimierung ersichtlich. Das Keyword `__restrict` ist ein Compiler-spezifischer Hinweis für den Compiler, dass die drei Pointer auf unabhängiges Memory zeigen. Dies erlaubt dem Compiler ein allfälliges Lesen vor dem Schreiben einer Speicherzellen, auf welche der Pointer zeigt, wegzuoptimieren. Solche Compiler-spezifische Mittel sollten üblicherweise vermieden werden und erst als letztes Mittel eingesetzt werden. Hier dient es als Beispiel für eine Low-Level-Optimierung, welche bei optimaler Cache-Ausnutzung nicht mehr viel Beschleunigung ausmacht.

5.1 Auszug: Compiler-Modes

Damit Sie überhaupt schnelle Ergebnisse erzielen können, müssen Sie den Code im `Release`-Mode kompilieren. Nur so optimiert der Compiler unnötige Indirektionen etc. weg und versucht optimierte Assembler-Befehle zu verwenden. Damit Sie trotzdem immer noch Debug-Informationen haben, sollten Sie noch besser im Mode `RelWithDebInfo` arbeiten. Aber Achtung: In diesem Modus sind zwar immer

noch Debug-Infos im Binary enthalten, allerdings wird sich der Debugger beim Stepen je nachdem seltsam verhalten, da er direkt über wegoptimierte Code-Teile springt.

5.2 Auszug: Caching

Die meisten Algorithmen sind Memory-bound. Das heisst, nicht die Operationen, welche auf den Daten ausgeführt werden sind aufwendig und brauchen sehr viel Zeit, sondern das Lesen der Daten aus dem Hauptspeicher (RAM) in den Cache (Layer 3, 2, 1) dauert lange. Deshalb sollte meistens zuerst versucht werden die «Lokalität» der Daten zu verbessern. Das heisst, es wird zeitnah auf Daten gearbeitet, welche im Speicher nahe beieinander liegen. Dies verbessert die Ausnutzung des Caches.

Wenn Sie z.B. ein zweidimensionales Bild verarbeiten, sollten Sie ihre Loops so organisieren, dass sie entsprechend dem Memory-Layout durch die Pixel iterieren. Ist ihr Bild also Zeile für Zeile im Speicher abgelegt, sollte ihr Verarbeitungs-Loop etwa so aussehen:

```
1 for (int y = 0; y < AnzahlZeilen; ++y) {
2   for (int x = 0; x < AnzahlSpalten; ++x) {
3     int pixel_value = image[x + y * AnzahlSpalten];
4   }
5 }
```

6 Optimierung der Matrix-Potenzierung *(optional)*

Als letztes sollen Sie die Matrix-Potenzierung auf Geschwindigkeit trimmen. Einerseits können Sie die bereits schnelle Variante `MultiplyInternFast` wiederverwenden, andererseits werden Sie hier nicht um algorithmische Optimierungen herum kommen. Das Internet und Wikipedia sind ein guter Anhaltspunkt um die theoretischen Möglichkeiten erläutert oder auch Implementierungs-Vorschläge zu bekommen. Eine Variante wäre die Exponentielle Quadrierung. Zudem ist wichtig, dass Sie keine unnötigen Memory-Allozierungen machen, sondern alloziertes Memory möglichst gut wiederverwenden.