Praktikum Bildgebung
Computertomographie

Prof. Dr. Heiko Neeb 7. März 2019

1 Lernziele

Im praktischen Teil zur Computertomographie stehen im Wesentlichen zwei Apekte im Vordergrund, die Sie erlernen bzw. vertiefen sollen:

1. Das Vorgehen und alle relevanten Schritte bei der Rekonstruktion der CT Bilder aus den Rohdaten.
2. Praktische Umsetzung dieser Schritte mit Hilfe einer objektorientierten Progammiersprache (C++).

Die Erfahrung hat gezeigt, dass trotz erfolgreichem Abschluss der Klausur in Bildgebung oftmals große Lücken im Verständnis der Bildrekonstruktion vorhanden sind. Daher sollen Sie sich nun nochmal im Rahmen des Praktikums eingehend hiermit beschäftigen. Um den in der Vorlesung kennengelernten Algorithmus auch praktisch implementieren zu können (schließlich soll am Ende ein Bild rauskommen, das man auch ganz real betrachten kann), benötigen Sie eine Applikation, die diese Aufgabe übernimmt. Hier setzt der zweite, ganz wesentliche Punkt ein: in den allermeisten Jobangeboten, die Ihnen jetzt oder künftig über den Weg laufen, werden solide Programmierkenntnisse von Ihnen gefordert - in der Regel in einer objektorientierten Programmiersprache. Erfahrungsgemäß tun sich auch hier viele von Ihnen schwer.

Was liegt also näher, als die beiden Punkte miteinander zu verknüpfen? Genau das soll das Lernziel in diesem Praktikum sein: Sie sollen erlernen, wie man ein konkretes Problem (hier: die Implementierung der naiven Rückprojektion der CT) in einem kleinen Softwareprojekt in der Programmiersprache C++ löst.

2 Kurze Einführung in die objektorientierte Programmierung in C++

Zu allererst: Es existiert eine Vielzahl von guten Einführungen in die Ideenwelt der objektorientierten Programmierung (OOP), sowohl im Internet als auch ganz klassisch in der Bibliothek. Sie sollten sich, sofern Sie noch nie von diesen Dingen gehört haben, daher auf jeden Fall mit der entsprechenden Literatur ausstatten und basierend darauf die Ideen und Konzepte sowie die praktische Implementierung in C++ studieren. Das vorliegende Kapitel kann dies nicht ersetzen und daher nur einen ganz groben Abriss über die wesentlichen Grundlagen geben. Ein sehr guter und beliebter Startpunkt für die eigene Literaturrecherche und zum Nachlesen/Verstehen der objektorientierten Programmierung stellt hierbei sicherlich das Buch *C++ für Dummies* dar.

Unabhängig von der weiterführenden Literatur soll hier nun kurz ein wesentlicher Punkt der OOP dargestellt werden: Das Prinzip der Kapselung in *Klassen* und der Unterschied zwischen einem *Objekt* und einer *Klasse*. Darüber hinaus bietet die OOP noch viele für das Progammieren wichtige Konzepte wie *Vererbung* oder *Polymorphismus* an, die aber für die konkrete Umsetzung des Problems hier nicht notwendigerweise benötigt werden. Trotzdem sollten Sie die Chance nutzen und sich beim Studium der Literatur auch gleich hiermit beschäftigen. Ihre künftige Stellensuche wird sich dadurch mit Sicherheit nicht erschweren, mit hoher Wahrscheinlichkeit aber wesentlich vereinfachen!

2.1 Erstmal nachdenken - Die Designphase

Wir starten mit einer Problemstellung, die auch für Ihre konkrete Implementierung von Relevanz ist: Sie möchten (müssen...) einen Code schreiben, der ein Sinogram einliest, dieses irgendwie bearbeitet und am Ende ein Bild erzeugt und wieder speichert, etwa so wie schematisch in Abbildung 1 dargestellt.

Abbildung 1: Grundlegende Komponenten und grundlegendes Ablaufschema Ihrer Applikation.

Jetzt könnten Sie z.B. so vorgehen, dass Sie sich hinsetzen, einen Texteditor oder irgendeine C++ Entwicklungsumgebung öffnen und dann einfach von oben nach unten eine Funktion ausimplementieren, die diese Aufgaben nach und nach erfüllt. Das würde sicher funktionieren, aber schon sehr schnell zu einem recht unübersichtlichen und schlecht verständlichen Code führen. Nun, das mag für Sie kein Problem darstellen, aber ein so implementierter Code hat einen ganz entscheidenden Nachteil: Sobald Sie eine Änderung vornehmen müssen (in der Praxis: Weil der Kunde es verlangt oder weil sich externe Komponenten geändert haben oder weil sich gesetzliche Vorgaben ändern.....Sie können die Liste beliebig lang fortsetzen) wird es heikel. Denn in einem solchen monolitischen Code hat eine Änderung an einer Stelle oftmals Auswirkungen an anderer Stelle, die Sie gar nicht bedenken. Vielleicht denken Sie noch dran, wenn Sie den Code erst letzte Woche geschrieben haben. Aber was ist, wenn das schon länger her ist? Dann haben Sie vieles vergessen. Jetzt müssen Sie etwas ändern und plötzlich merken Sie, dass Ihre Änderung noch an vielen anderen Stellen etwas bewirkt hat, sodass Ihre Applikation am Ende nicht mehr das tut was sie tun soll.

Hier setzen nun objektorientierte Sprachen an, die Ihnen helfen können, dieses in der Praxis (fast) immer auftretende Problem zu lösen bzw. zumindest abzumildern. Dazu müssen Sie aber erst einmal nachdenken, bevor Sie überhaupt anfangen zu Programmieren. Was ist damit konkret gemeint? Nun, es gilt z.B. Dinge zu identifizieren, die einen gemeinsamen Ursprung oder irgendeine Gemeinsamkeit aufweisen, sodass Sie am Ende eine bessere Struktur Ihres Codes erhalten. Schauen Sie dazu auf unser Beispiel aus Abbildung 1. Hier werden Sie in Ihrem Code sicherlich ein Sinogram laden und ein Bild speichern müssen. Also fragen wir uns doch mal, was Sinogram und Bild evtl. gemeinsam haben, so dass wir evtl. eine weitere Komponente (die wir dann ganz bald als *Klasse* bezeichen werden) mit ins Spiel bringen können und die wir sowohl in unserem Codeteil zum Sinogram als auch im Codeteil zum Bild verwenden können.

Überlegen Sie erst mal selbst, was Sinogram und Bild im Hinblick auf Ihre Darstellung in einer Programmiersprache gemeinsam haben.

Sind Sie drauf gekommen? Genau, beides sind Matrizen. In einem Fall speichern wir in der Matrix die CT Messwerte, die zu unterschiedlichen Lateralpositionen p und unterschiedlichen Winkeln ϕ gehören. Das sind genau unsere beiden Matrixdimensionen. Im anderen Fall sind die Matrixdimensionen die x und y Komponente eines Bildes.

Daher liegt es nahe, in unserem Gedankenkonzept eine weitere Komponente mit ins Spiel zu bringen. Da wir uns sowohl bei der Erzeugung des Bildes als auch beim Lesen des Sinograms mit Matrizen beschäftigen müssen, nennen wir diese neue Komponente also einfach **Matrix**. Da sowohl das Sinogram also auch das Bild durch eine Matrix repräsentiert werden, verdeutlichen wir diese Abhängigkeit in unserem Schema einfach durch einen Pfeil (siehe Abbildung 2) und merken uns, dass ein solcher Pfeil mit einer Raute am Ende bedeuten soll, dass eine Matrix zu einem Sinogram bzw. zu einem Bild gehört. Alternativ formuliert: Eine Matrix ist Teil eines Bildes / eine Matrix ist Teil eines Sinograms.

Aber eine Matrix ist eben tatsächlich nur ein Teil eines Bildes. Machen Sie sich auch das nochmal klar. Überlegen Sie, welche Informationen neben den in einer Matrix gespeicherten Werten Sie typischerweise benötigen, um ein Bild oder ein Sinogram eindeutig zu charakterisieren?

Alleine die Matrixwerte eines Sinograms sind beispielsweise wertlos, denn Sie sagen nichts über die konkrete Winkelstellung und die konkrete Lateralposition z.B. des Elements in der 34. Zeile und 15. Spalte aus. Um dies zu wissen, müssen Sie noch weitere Informationen haben und speichern, beispielsweise die minimale und maximale Winkeleinstellung oder den Abstand zwischen zwei Detektorelementen. Erst dann wird aus den Daten in der Matrix ein sinnvoll interpretierbares Sinogram. Analog beim Bild: Erst wenn Sie das Field-of-View (FoV) kennen, können Sie sinnvolle Aussagen über den physikalischen Abstand zwischen zwei Bildpunkten machen (denken Sie hier z.B. an einen Radiologen, der den maximalen Tumordurchmesser vor und nach der Therapie bestimmen möchte).

Sie sehen also, das sowohl ein Bild als auch ein Sinogram mehr Informationen als nur die Matrixelemente enthalten muss, um zu einem sinnvollen Bild oder einem sinnvollen Sinogram zu werden. Dies ist der Grund wieso wir eben gesagt haben, dass eine Matrix nur Teil eines Bildes/Sinogramms ist.

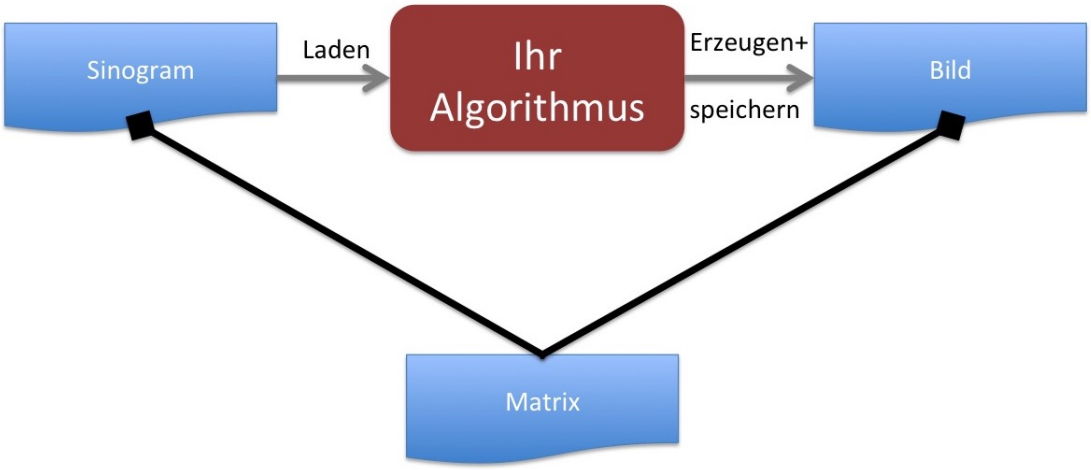

Abbildung 2: Um die gemeinsame Komponente erweitertes Ablaufschema Ihrer Applikation.

Mit dieser Erkenntnis ausgerüstet können wir nun zurück zu unserer Strukturierung des Codes gehen. Offensichtlich haben wir es hier mit insgesamt drei Komponenten zu tun: Einem Bild, einem Sinogram und einer Matrix, die Teil sowohl des Bildes als auch des Sinograms ist. Wie lässt sich so etwas nun in der Programmiersprache abbilden? Hier kommt der erste wesentliche Vorteil der objektorientierten Sprachen zum tragen. Wir definieren uns für jede der eben erkannten Komponenten eine sogenannte *Klasse*. So soll beispielsweise die Bildklasse alles das enthalten und zusammenführen, was ein Bild ausmacht und was man mit einem Bild machen kann.

Nehmen Sie sich einen Stift und ein Stück Papier zur Hand und schreiben Sie Dinge auf, die ein Bild beschreiben und Dinge, die man mit einem Bild machen kann.

Vielleicht sind Sie ja auf etwas ähnliches gestoßen wie in Abb. 3 dargestellt. Dort sehen Sie im oberen Teil eine Aufstellung von typischen Bildeigenschaften und im unteren Teil von typischen Dingen, die man mit einem Bild anstellen kann. Die Eigenschaften wollen wir ab jetzt als *(Klassen)attribute* und die Funktionen als *(Klassen)methoden* bezeichnen.

Überlegen Sie sich analog zum Bild, welche Klassenattribute und Klassenmethoden für die beiden anderen Klassen (Sinogram und Matrix) relevant sein könnten.

Nun haben wir schon einen wichtigen Teil unserer Progammieraufgabe gelöst und zwar ohne eine einzige Zeile an Code zu schreiben! Fassen wir nochmal zusammen, wie wir bis hierhin vorgegangen sind und welche Schritte wir dabei durchgeführt haben:

1. Wir haben im 1. Schritt die wesentlichen Komponenten unseres Problems identifiziert (hier: Sinogram und Bild)
2. Wir haben im 2. Schritt nach Gemeinsamkeiten der Komponenten gesucht (hier: beides sind Matrizen)

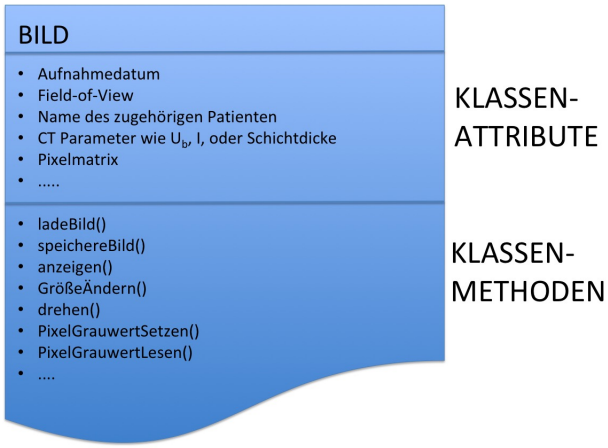

Abbildung 3: Mögliche Attribute und Methoden der Klasse Bild.

3. Im 3. Schritt wurde dann der Zusammenhang zwischen den Komponenten definiert (hier: Sinogram und Bild enthalten beide als Teilinformation eine Matrix).

4. Im 4. Schritt haben wir die Komponenten mit Leben gefüllt: Wir definieren, welche Eigenschaften jede Komponente haben soll und welche Funktionen mit einer solchen Komponente ausführbar sein sollen. Wir nennen jede Komponente eine Klasse.

In unserem Beispiel haben wir also drei relevante Klassen identifiziert:

- Die Klasse *Sinogram*
- Die Klasse *Bild*
- Die Klasse *Matrix*

2.2 Jetzt handeln - Die Implementierung

Bis hierhin ist dies zunächt ein sehr theoretisches Konstrukt, denn rein durch dieses Nachdenken haben wir ja das Problem noch nicht praktisch gelöst. Aber Sie werden sehen, dass wir durch diese Definition der Struktur unsere praktische Progammieraufgabe stark vereinfacht haben. Es geht also jetzt darum, diese eben gefundene Struktur in einer Programmiersprache abzubilden. Hier setzen die objektorientierten Sprachen an, denn sie geben uns genau das dazu nötige Werkzeug. Im Prinzip können Sie das Konzept nun in irgendeine beliebige OO Programmiersprache überführen, aber wir wollen es auf Grund der immensen praktischen Relevanz hier und im weiteren nur für den Fall der C++ Programmierung besprechen.

Der Clou ist Folgender: C++ (und alle anderen OOP) bietet Ihnen direkt die Möglichkeit, die auf dem Papier gefundene Klassenstruktur in nutzbaren Code zu überführen! Dazu gibt es in C++ ein neues Primitiv, dass *class* genannt wird. Jetzt müssen wir uns nur anschauen, wie dies konkret geschieht. Wir machen dies am konkreten Beispiel der Matrix-Klasse. Erzeugen Sie sich zunächst eine Header Datei (die Sie ja schon aus C kennen), die z.B. *matrix.h* heißt und öffnen diese im Editor (Ihrer Entwicklungsumgebung). Die Headerdatei enthält im Fall von C typischerweise die Deklaration von Funktionen oder von statischen Variablen. Das C++ eine direkte Erweiterung von C ist, können Sie dies auch in diesem Fall so verwenden. Allerdings wird die Header-Datei jetzt im

```
 * matrix.h

#ifndef MATRIX_H_
#define MATRIX_H_

#include <vector>
#include <string>

using namespace std;

class matrix {

    vector<double> m_Mat;
    int m_nRow;
    int m_nCol;

public:

    // Constructor and Destructor
    matrix(int nrow=0, int ncol=0);
    virtual ~matrix();

    // Assginment operator: m(i,j) gives access to the element in the i-th row and j-th column
    double& operator() (int i, int j) { return m_Mat[(i-1)*m_nCol+j-1]; }

    // Load a matrix from a space-separated text (ASCII) file with filename "fName"
    bool loadFromFile(string fName);

    // Store the current contents of the matrix in a space-separated text (ASCII) file with filename "fName"
    bool storeMatrixInTextFile( string fName );

    // Print current matrix content on the screen
    void print(void);

    // Return the number of matrix columns
    int getNCols(void) { return m_nCol; };

    // Return the number of matrix rows
    int getNRows(void) { return m_nRow; };

    // Resize matrix to nrow x ncol and set elements to zero
    void reset(int nrow, int ncol);

    // Set the matrix element at location ("row","col") to "value"
    void set(int row, int col, double value);
};

#endif /* MATRIX_H_ */
```

Abbildung 4: Eine mögliche Deklaration der Klasse Matrix in C++.

Wesentlichen die Deklaration der zugehörigen Klasse enthalten. Hierunter ist gemeint, dass Sie dort Ihre theoretisch gefundene Struktur (siehe Abbildung 3) in nutzbaren Code übertragen. Sie sagen in der Deklaration dem Compiler also nur, welche Attribute und welche Methoden (Funktionen) Ihre Klasse ausmachen sollen. Dies könnte im Fall der Matrixklasse z.B. dann so aussehen wie in Abb. 4 dargestellt.
Sie sehen, dass dort sowohl Attribute als auch Funktionen definiert sind. Die Klassenattribute einer Matrix sind die Anzahl an Spalten (Variable m_nCol), die Anzahl an Spalten (Variable m_nCol) sowie ein double-Vektor, der die eigentlichen Matrixelemente enthält (Variable m_Mat). Darüber hinaus enthält die Deklaration der Klasse matrix eine Reihe von Methoden, u.a. zum Laden einer Matrix aus einer Datei (*loadFromFile*), zum Resetten der Matrix auf eine neue Größe von $nrow \times ncol$ (*reset*) oder zum Zugriff auf das Element der i-ten Zeile und j-ten Spalte mit Hilfe des Operators *operator()(int i, int j)*.

Damit die Methoden von außen (also vom Programmteil, das die Klasse Matrix nutzt) zugänglich

```
 * main.cpp

#include <iostream>
#include "matrix.h"

using namespace std;


int main() {

    // Hier deklarieren wir 2 Variablen (Objekte) von unserem neuen Typ "matrix".
    matrix alteMatrix;
    matrix neueMatrix;

    // Ausserdem benötigen wir noch ein paar andere Variable, z.B. die
    // Dateipfade der einzulesenden und der zu schreibenden Matrix
    string alteMatrixPfad = "/Users/neeb/Summary/Teaching/SS2019/CT Praktikum/TestMatrix.txt";
    string neueMatrixPfad = "/Users/neeb/Summary/Teaching/SS2019/CT Praktikum/NeueMatrix.txt";

    // Nun können wir die alte Matrix einaden. Dazu nutzen wir die Methode
    // loadFromFile() der Matrix Klasse. Die Matrixwerte sollen dem Objekt "alteMatrix"
    // zugeordnet werden. Dies erreichen wir mit folgenden Befehl:
    alteMatrix.loadFromFile( alteMatrixPfad );

    // Jetzt können wir die Größe der Matrix ausgeben
    cout << "Anzahl an Spalten = " <<alteMatrix.getNRows() <<" Anzahl an Zeilen = " << alteMatrix.getNCols() <<endl;

    // Nun soll die 10x10 große Untermatrix am linken oberen Ende kopiert werden. Dazu müssen wir dem Objekt
    // "neueMatrix" sagen, dass seine Größe nun 10x10 sein soll.
    neueMatrix.reset(10,10);

    // Jetzt können wir Punkteweise kopieren. Dazu müssen wir die Methode set() des Objekts neueMatrix aufrufen und
    // dort an der Stelle (row,col) den Wert einfüllen, der in der altenMatrix an genau dieser Stelle steht.
    for(int row=1; row<=10; row++) {
        for(int col=1; col<=10; col++) {
            neueMatrix.set(row,col, alteMatrix(row,col));
        }
    }

    // Schließlich können wir die so neu erzeugte Matrix wieder als Textfile speichern
    neueMatrix.storeMatrixInTextFile(neueMatrixPfad);
}
```

Abbildung 5: Eine mögliche Implementierung des Codes zur Matrixmanipulation basierend auf der Verwendung der Klasse matrix.

sind, ist ihnen das Schlüsselwort *public* vorangestellt. Das bedeutet aber auch, dass auf alle nicht public Teile nicht von außen zugegriffen werden kann. In dem Beispiel der matrix Klasse sind das genau die drei Klassenattribute (m_nRow, m_nCol, m_Mat).

Die eigentliche Implementierung dieser Funktionen ist dabei in einer getrennten Datei enthalten, die nach der allgemein angewandten Konvention den Namen *matrix.cpp* trägt. Diese ist aber für Sie als Nutzer der Klasse weniger interessant. Alles, was Sie als Nutzer wissen müssen, ist die Header Datei mit der Deklaration der Klasse. Daher werden wir uns nun damit beschäftigen, wie Sie eine solche Klasse in einem eigenen Programm nutzen können.

Stellen Sie sich vor, Sie möchten eine Matrix aus einer Datei laden und die Elemente der zehn ersten Zeilen und Spalten in eine neue Matrix kopieren. Dann soll die neue Matrix in einer neuen Datei gespeichert werden. Ausserdem sollen die Anzahl an Zeilen und Spalten der ersten Matrix auf dem Bildschirm ausgegeben werden. Eine mögliche Implementierung hierzu ist in Abbildung 5 gezeigt.

Sie sehen, dass Sie Ihrem *main()* Programm mitteilen müssen, wo Ihre Klasse matrix deklariert ist.

Dies geschieht durch den Befehl *include "matrix.h"*. Darüber hinaus erkennen Sie an diesem kleinen Beispiel direkt den Umgang mit Ihrer neu definierten Klasse. Diese definiert nämlich einen neuen Datentyp analog zu den vorgegebenen Typen wie *int*, *double* oder *string*! Das bedeutet aber, dass Sie mit matrix nun genauso umgehen wie Sie es z.B. von *int* gewohnt sind. Sie können sich nun beliebig viele Variablen des Typs matrix definieren. In unserem Fall werden zwei Variablen angelegt (*alteMatrix* und *neueMatrix*), die dann im weiteren Programm verwendet werden können. Dieses Anlegen von Variablen hat nun in der objektorierentieren Terminologie eine andere Bezeichnung. Dort spricht man davon, dass **zwei Objekte der Klasse matrix erzeugt werden**. Aber wie immer im Leben: Man kann ein und dieselbe Sache auf unterschiedliche Weise ausdrücken, ohne dass sich der Inhalt dadurch ändert.

Es gibt allerdings trotzdem einen entscheidenden Unterschied zwischen einer Variable, die z.B. als *int* deklariert wurde und einer Variable, die zum Typ matrix gehörend deklariert wurde: Während die *int* Variable tatsächlich nur einen einzelnen Wert speichert, kann die Variable vom Typ matrix viel mehr: Sie haben nun Zugriff auf alle Methoden, die Ihnen die Klasse Matrix zur Verfügung stellt. Davon wird im weiteren Programmverlauf an mehreren Stellen Gebrauch gemacht. So wird beispielsweise mit dem Befehl *alteMatrix.loadFromFile(alteMatrixPfad);* das Objekt *alteMatrix* mit den Werten gefüllt, die in der Datei gespeichert sind, die unter dem Pfad *alteMatrixPfad* abgelegt ist. Analog sorgt der Befehl *neueMatrix.reset(10,10)* dafür, dass das Objekt *neueMatrix* (das die neuen zu kopierenden Matrixelemente enthalten soll) auf eine Größe von 10×10 erweitert wird.

Zusammenfassend sehen Sie also folgendes allgemein gültige Vorgehen bei der Benutzung von Klassen in Ihrem eigenen Code:

1. Machen Sie Ihrem Code die zu nutzenden Klassendeklarartion durch das Einladen der zugehörigen Header Datei mit dem Befehl *include* bekannt.

2. Definieren Sie sich soviele Objekte vom Typ Ihrer Klasse, wie Sie es für die Bearbeitung der Aufgabe benötigen.

3. Wenn Ihr Objekt (Ihre Variable) den Namen *meineVariable* trägt, so können Sie durch *meineVariable.MethodenName();* die Klassenmethode mit dem Namen *MethodenName()* aufrufen.
   Achtung: Wenn Sie die Variable als Pointer deklarieren sollten, so müssen Sie anstelle des Punktes einen Pfeil zum Aufruf nutzen, konkret also *meineVariable→MethodenName();*.

3 Aufgabenstellungen

Der grundlegende Zusammenhang und Ablauf der unterschiedlichen Teilaufgaben ist in Abbildung 6 zusammengestellt. Im Vorgerdrund steht dabei die Implementierung einer kleinen C++ Applikation. Allerdings müssen Sie auch Matlab verwenden, um z.B. die in Ihrer Applikation erzeugten Bilder einzuladen und entsprechend dem in der Vorlesung besprochenen Vorgehen zu filtern (mit dem richtigen Hochpassfilter). Außerdem nutzen Sie sinnvollerweise Matlab, um selbst Sinogramme zu erzeugen, die einzelnen oder mehreren Punktobjekten entsprechen. Die Details hierzu werden im Folgenden dargestellt.

3.1 Implementierung der naiven Rückprojektion in C++

3.1.1 Grundlegender Ablauf

Die wesentlichen Komponenten haben Sie schon im vorherigen Kapitel kennengelernt (siehe Abbildung 2). Ihre Aufgabe besteht nun darin, in einer *main()* Funktion den grundlegenden Ablauf der naiven Rückprojektion zu implementieren, den Sie in der Vorlesung kennengelernt haben. Dazu stehen Ihnen Klassen zur Verfügung, die Sie zur Repräsentation eines Sinograms und eines Bildes nutzen können (siehe unten).

Die Basis für Ihren zu implementierenden Algorithmus ist das im File *Sinogram_InVivo.txt* gespeicherte Sinogram. Dieses wurde mit einem CT System aufgenommen, dessen Detektorarray eine Gesamtlänge von $75cm$ hat. Hiermit soll nun ein Bild der Matrixgröße 350×400 Pixel erzeugt werden. Das Field-of-View soll dabei in x-Richtung $450mm$ und in $y-$Richung $400mm$ betragen. Das erzeugte Bild soll anschließend von Ihrem Code als Textfile abgespeichert werden. Das Textfile können Sie dann in Matlab z.B. mit dem Befehl *dlmread()* importieren und die eingeladenen Matrix mit *imagesc()* als Bild darstellen.

Machen Sie sich daher zunächst nochmal klar, wie Sie die Einträge im Sinogram (p, ϕ) mit Hilfe der naiven Rückprojektion zu Bilddaten überführen können. Schreiben Sie die Schritte in einer Form auf ein Blatt Papier, die ihnen die Übertragung in einen ausführbaren Code erleichtert.

Tipp: Da Sie hier gleich an zwei Baustellen (Umsetzung eines Algorithmus + vermutlich neue Elemente bei der Programmierung) arbeiten müssen, empfielt es sich, dass Sie den Algorithmus zunächst in Matlab austesten bis er komplett funktioniert. Danach können Sie das dann viel leichter in einen C++ Code übertragen. Dies ist auch ein in der Paxis oftmals gewähltes Vorgehen!

Wenn Sie den Algorithmus in C++ implementiert haben, sodass er genau das tut was er soll, können Sie die erzeugte Bildmatrix in Matlab einlesen. Dazu könnte der Befehl *dlmread()* hilfreich sein. Stellen Sie dann das Bild graphisch mit *imagesc()* dar, sodass Sie sich den beiden folgenden Fragen widmen können:

- Was fällt Ihnen an Ihrem Bild auf?
- Was könnte dort dargestellt sein (welches Organ oder welche Körperstruktur)?

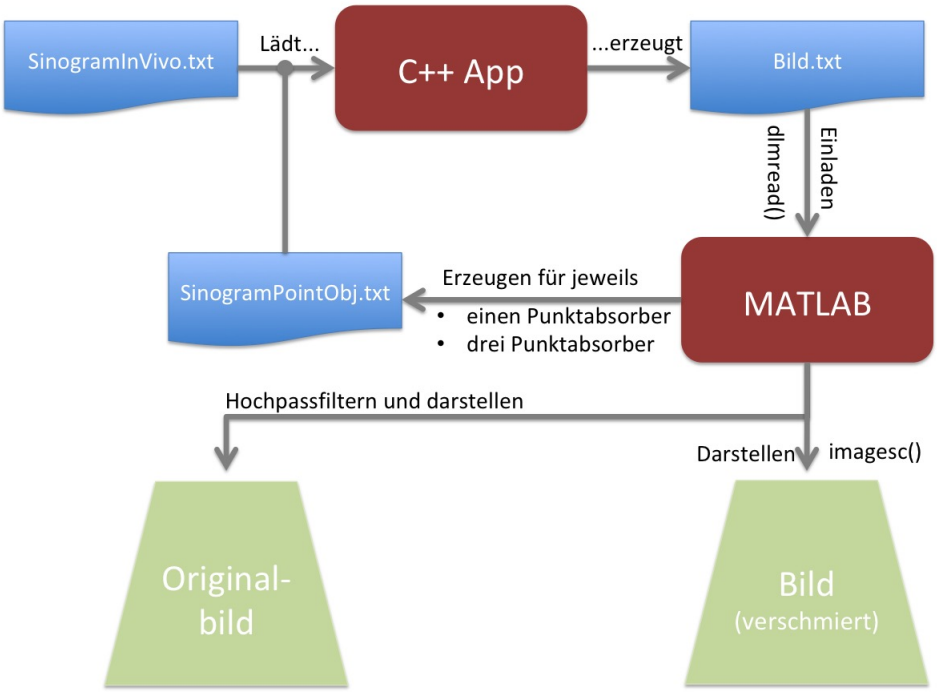

Abbildung 6: Übersicht über die im CT Praktikum zu bearbeitenden Aufgaben.

3.1.2 C++ Klassen und deren Methoden

Die Basis für Ihre Implementierung in C++ ist eine kleine Klassenbibliothek, in der die Klassen *Sinogram*, *Bild* und *matrix* implementiert sind. Für Sie als Nutzer/in der Bibliothek ist dabei aber die Klasse *matrix* nicht relevant. Diese wird ausschließlich von den beiden Klassen *Bild* und *Sinogram* verwendet, wie auch in Abbildung 2 dargestellt. In der objektorientierten Notation bezeichnet man dieses Vorgehen als *Kapselung*. Daher müssen Sie sich hier ausschließlich mit den Klassen *Bild* und *Sinogram* beschäftigen. Wie Sie oben gelernt haben, ist für Sie als Klassennutzer lediglich die Headerdatei (.h) von Interesse, aber i.d.R. nicht die Implementierung der Klassenmethoden, die in der .cpp Datei enthalten ist.

Schauen Sie sich also die zu den Klassen *Bild* und *Sinogram* gehörenden Headerdateien an und machen Sie sich bei jeder einzelnen Klassenmethode anhand der Beschreibung klar, wofür diese eingesetzt wird und welche Aufgabe diese Methode übernimmt.

Sollte Ihnen bei einigen Methoden die Bedeutung nicht klar geworden sein, so können Sie sich aber auch trotzdem im zugehörigen .cpp File die Implementierung der Methode anschauen. Damit sollte dann eindeutig klar werden, welche Aufgabe diese Methode übernimmt.

Konkret müssen Sie nun ein Sourcefile erzeugen, in dem die *main()* Funktion implementiert ist. Wie Sie aus Ihrer Programmiervorlesung wissen, ist dies der Startpunkt bei der späteren Ausführung Ihrer Applikation. Wir gehen für das weitere Vorgehen davon aus, dass die *main()*-Funktion in einem File mit dem Namen *NaiveBackprojection.cpp* implementiert ist, wobei Sie in der Wahl des Filenamens natürlich komplett frei sein. Als Beispiel wie Sie Ihre beiden Klassen *Sinogram* und

Bild in Ihrer *main()*-Funktion nutzen können, kann sicherlich auch der Code in Abb. 5 dienlich sein.

Noch ein kleiner praktischer Hinweis: Sie können Ihren Sourcecode natürlich in jedem beliebigen Texteditor editieren. Es empfielt sich aber sehr, hier direkt eine C++ Entwicklungsumgebung (Integrated Development Environment, IDE) einzusetzen, da diese Ihnen viele komfortable Optionen zur Erstellung, Editierung, Kompilierung, Ausführung und Test des Codes bietet. Es existieren viele verschiedene, auch frei zugängliche, Entwicklungsumgebungen, die Sie für die C++ Entwicklung einsetzen können, beispielsweise die plattformunabhängigen Tools *Eclipse* und *Netbeans* oder das *Microsoft Visual Studio*, das aber nur auf Windows Rechnern läuft. In den IDEs können Sie direkt die zur Verfügung gestellten Klassen mit in ihr Projekt einbinden, sodass diese automatisch vom Compiler mit übersetzt werden. In diesem Fall brauchen Sie sich um das folgende Kapitel 3.1.3 nicht zu kümmern, da die Kompilierung von Ihrer IDE übernommen wird.

3.1.3 Compilierung am Beispiel des gnu-Compilers

Sofern Sie sich entschieden haben sollten den Code doch nicht in einer integrierten Entwicklungsumgebung zu enwickeln, müssen Sie die Kompilierung von Hand vornehmen. Dazu müssen zwei Vorraussetzungen erfüllt sein:

1. Sie müssen einen C++ Compiler auf Ihrem System installiert haben.
2. Sie müssen ein Commandline Terminal öffen.

Stellen Sie also zunächst sicher, dass der häufig eingesetze und frei verfügbare gnu C++ Compiler auf Ihrem System installiert ist. Wenn Sie Linux nutzen, ist dieser normalerweise bereits Teil Ihres Systems. Bei macOS und Windows müssen Sie diesen i.d.R. erst noch installieren (https://gcc.gnu.org).

Im Folgenden gehen wir nun davon aus, dass sich alle benötigten Source und Headerfiles[1] im gleichen Verzeichnis befinden, z.B. unter *\users\meinComputer\NaiveRückprojektion*.

Welchseln Sie nun im Commandline-Terminal zunächst in dieses Verzeichnis:

```
cd \users\meinComputer\NaiveRückprojektion
```

Nun können Sie auch direkt schon Ihren Code kompilieren. Dazu rufen Sie den gnu-Compiler mit folgendem Befehl auf:

```
g++ -std=c++11 -o "NaiveBackprojection" Image.cpp NaiveBackprojection.cpp Sinogram.cpp matrix.cpp
```

Wenn alles funktioniert hat, wurde eine ausführbare Applikation mit dem Namen NaiveBackprojection erzeugt. Diese können Sie nun einfach mit dem Befehl

```
./NaiveBackprojection
```

in der Kommandozeile starten.

3.2 Bestimmung der PSF

Wenn Ihr C++ Code funktioniert und sich ein sinnvolles Bild für das Sinogram *Sinogram_InVivo.txt* ergibt, können Sie sich nun an die Bestimmung der Point-Spread-Function begeben. Dazu müssen Sie nun ein Sinogram erstellen, das einen Punktabsorber im Isozentrum des Scanners repräsentiert.

[1] Das sind also konkret die zur Verfügung gestellten Files *matrix.h*, *matrix.cpp*, *sinogram.h*, *sinogram.cpp*, *bild.h*, *bild.cpp* sowie das von Ihnen entwickelte File *NaiveBackprojection.cpp*

Dies führen Sie am besten in Matlab durch. Erzeugen Sie sich eine passende Matrix, füllen diese mit den zum Punktabsorber gehörenden Werten und speichern diese Matrix dann als Textdokument ab (dazu könnten Sie z.B. den Befehl *dlmwrite()* nutzen).
Nun müssen Sie nur noch Ihren C++ Code mit diesem neuen Sinogram durchlaufen lassen, das Bild in Matlab einladen und darstellen. Dies ergibt die gesuchte PSF.

Um darüber hinaus noch zu untersuchen wie das Bild in Anwesenheit von mehreren Punktabsorbern aussieht, erzeugen Sie ein weiteres Sinogram. Dieses soll nun insgesamt drei Punktabsorber mit folgenden Eigenschaften repräsentieren:

- Ein Punktabsorber mit $\mu = 1cm^{-1}$ am Ort $\vec{r_1} = (-2, 3)cm$.
- Ein Punktabsorber mit $\mu = 5cm^{-1}$ am Ort $\vec{r_2} = (0, 5)cm$.
- Ein Punktabsorber mit $\mu = 0.5cm^{-1}$ am Ort $\vec{r_3} = (1, 10)cm$.

Die komplette Länge des Detektorarrays soll dabei wie oben $75cm$ betragen. Gehen Sie die gleichen Schritte wie bei der Bestimmung der PSF durch und bewerten Sie, ob das dabei erhaltene Bild sinnvoll aussieht.

3.3 Hochpassfilterung der Daten

Nun nutzen Sie wieder das mit dem originalen Sinogram *Sinogram_InVivo.txt* erzeugte Bild, das in Matlab eingelesen wird. Wir haben in der Vorlesung ein Verfahren besprochen wie Sie dieses Bild durch einen passenden Hochpassfilter "besser" machen können. Führen Sie die dazu nötigen Schritte nun in Matlab durch und vergleichen Sie das Bild vor der entsprechenden Hochpassfilterung mit den Bild, das Sie danach erhalten haben. Was fällt Ihnen auf?

Hinweis: Für die Hochpassfilterung könnte der Matlab-Befehl *fft2()* nützlich sein. Wenn Ihnen die dabei erhaltene Frequenzverteilung seltsam vorkommt, mag auch ein Blick auf die Funktion *fftshift()* helfen.

3.4 Zusammenfassung

Als Zusammenfassung nochmal das bevorzugte Vorgehen bei der Lösung der Aufgaben:

1. Überlegen Sie sich auf einem Blatt Papier wie Sie grundsätzlich vorgehen, um den Algorithmus der naiven Rückprojektion in ein ausführbares Programm zu überführen.
2. Testen Sie Ihre Ideen zunächst ausschließlich in Matlab aus und gehen Sie solange wieder zurück zu Punkt (1) bis Ihr Code das tut, was er soll.
3. Nun können Sie sich mit der Umsetzung in C++ beschäftigen. Schauen Sie sich dazu die Header-Dateien der Klassen *Sinogram* und *Bild* genau an. Diese bieten Ihnen bereits vorimplementierte Funktionen, die Sie nutzen sollten.
4. Entscheiden Sie sich idealerweise für die Nutzung einer IDE. Erzeugen Sie ein Sourcefile, das die *main()* Funktion enthält.
5. Implementieren Sie die *main()*-Funktion, sodass diese genau das ausführt, was auch Ihr Matlab-Code macht.
6. Führen Sie Ihren compilierten C++ Code aus und laden Sie das erzeugte Bild in Matlab ein.
7. Nutzen Sie Matlab, um zwei neue Sinogramme zu erzeugen. Erzeugen Sie mit Ihrem C++ Code für beide Sinogramme die zugehörigen Bilder und betrachten/bewerten Sie diese in Matlab.

8. Laden Sie wieder Ihr erstes Bild, das basierend auf (*Sinogram_InVivo.txt*) erzeugt wurde, in Matlab ein.

9. Filtern Sie diese Bild mit dem in der Vorlesung besprochenen Hochpassfilter und vergleichen gefiltertes und nicht gefiltertes Bild.

4 Organisatorisches

Sie können das CT Praktikum zu jedem beliebigen Zeitpunkt beginnen. Idealerweise machen Sie dies bereits parallel zur Vorlesung. Dies wird Ihrem Verständnis des Stoffes extrem zu Gute kommen und die Prüfungsvorbereitung stark erleichtern. Abgeben bzw. von mir bewerten lassen können Sie Ihr Ergebnis aber erst, wenn Sie auch die Klausur bestanden haben, analog zum MRT Versuch. Es gilt auch weiterhin, dass zur erfolgreichen Absolvierung sowohl beide MRT Versuche, beide Ultraschallversuche als auch der CT Versuch bestanden sein müssen.

Bitte bearbeiten Sie die Aufgaben immer in einer **Zweiergruppe**. Wie Sie die Gruppen bilden, bleibt Ihnen überlassen. Teilen Sie aber bitte die Gruppeneinteilung Frau Lafontaine im Rahmen der Anmeldung/Vorbesprechung des Bildgebungspraktikums am Anfang des Semesters mit.

Nachdem Sie die Aufgaben bearbeitet haben, schicken Sie mir bitte den Code (**nur Source-Code, keine ausführbaren Applikationen o.ä!**) per eMail zu. Sie müssen kein großes Protokoll schreiben, sollten aber ebenfall ein kurzes Dokument als PDF beifügen, in dem Sie beschreiben, wie die Bilder aussehen, die Sie bei den verschiedenen Teilaufgaben erhalten haben.

Nach Durchsicht Ihres Codes und des beigefügten Dokuments vereinbaren wir dann individuelle Termine. Hier werde ich mit Ihnen gemeinsam den Code durchgehen um zu schauen, ob Sie diesen auch selbst erstellt haben. Dabei werde ich mir u.a. von Ihnen einige Passagen erklären lassen. Sollte sich dabei herausstellen, dass der Code oder Teile davon von anderen Gruppen übernommen wurde, müssen Sie den Praktikumsversuch im nächsten Semester noch einmal wiederholen.

Wenn bei Ihnen zwischendurch bei der Bearbeitung der Aufgaben Fragen aufkommen sollten, können Sie gerne einen Gesprächstermin per eMail vereinbaren.