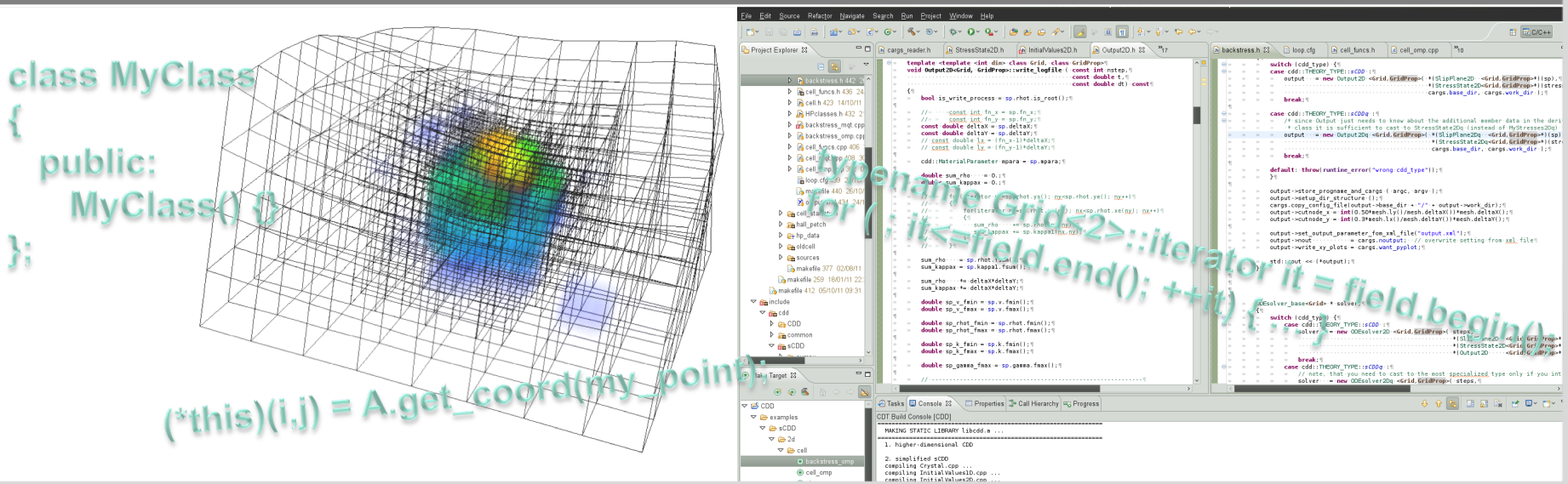
Karlsruher Institut für Technologie

Wissenschaftliches Programmieren für Ingenieure

A. Trenkle, T.Achkar, Dr. M. Stricker Dr. D. Weygand

Übung 5: (~2 Termine)
Pointer und Felder; Klassen; Lineare Gleichungssysteme, Anwendung: das Wärmeleitproblem

Institute for Applied Materials – Computational Materials Science (IAM-CMS)
KIT – The Karlsruhe Institute of Technology – University of the State of Baden-Wuerttemberg, GERMANY



Gliederung der Übung

Info

In den letzten Übungen haben Sie die **Vector-/Matrix-Klassen** der **Eigen3** Bibliothek verwendet.

Sie werden heute:

1. Eigene Vector-/Matrix-Klassen schreiben bzw. erweitern
2. Dynamische Felder anlegen
3. Diese Klassen zum Lösen von linearen Gleichungssystemen verwenden
4. Das 1D-Wärmeleitproblem aus der Vorlesung lösen

0.1 Fehlersuche mit `assert`

Info

`assert` Bibliothek erleichtert die Fehlersuche

- Mit `assert` können Bedingungen überprüft werden, z.B. ob Variablen positiv sind etc.
- Nichterfüllen führt während der Ausführung des Programms zum Abbruch und eine Fehlermeldung wird ausgegeben
- Einbinden erfolgt über `#include <cassert>`
- Mit der Compileranweisung `NDEBUG` wird `assert` ignoriert

```cpp
#include <iostream>
#include <cassert>

using namespace std;

int main(){

    int i;

    for (;;){
        cin >>i;
        assert(i!=0);
        cout<<"i="<<i<<endl;
    }

}
```

```
g++ main.cxx
./a.out
```

```
1
i=1
2
i=2
3
i=3
0
a.out: main.cxx:12: int main(): Assertion `i!=0' failed.
Aborted
```

```
g++ -DNDEBUG main.cxx
./a.out
```

```
1
i=1
2
i=2
3
i=3
4
i=4
0
i=0
1
i=1
```

1.1 Dynamische Felder

Info

Siehe auch Vorlesung ab S.109, **besonders S.112-116**

- **Erzeugen** eines eindimensionale Felds:
  - `int *a;`
  - `a=new int[10];`
- **Löschen** eines eindimensionalen Felds:
  `delete[] a;`

- **Zugriff**:
```
a[2] = 5; //Schreiben
int i = a[1]; //Lesen
```

- **Erzeugen** eines zweidimensionalen Felds (3x5):
```
int **b;
b = new int*[3];
b[0]=new int[3*5];
for (int i=1; i<3 ; i++)
     b[i] = b[0]+i*5;
```
- **Löschen** eines zweidimensionalen Felds:
```
delete[] b[0];
delete[] b;
```

- **Zugriff**:
```
b[0][4] = 3; //Schreiben
int i;
i = b[2][3]; //Lesen
```

2. Klassen: Aufbau

Info

Siehe auch **Vorlesung ab S.179**

- In Klassen werden (eigene) Datentypen definiert
- Klassen beinhalten:
  - **Membervariablen**: Variablen, die notwendig sind, um die Aufgaben der Klasse umsetzen zu können
  - **Memberfunktionen**: Schnittstelle zur Verwendnung der Klasse, wenn als public deklariert
  - **Operatoren**: klassenspezifisches Verhalten für „normale“ Operatoren wie +,-,*,=,(),...
  - **Konstruktoren**: deklarieren Membervariablen bei Erzeugung eines Objekts der Klasse
  - **Destruktor**: Löscht Membervariablen nach Ende des Gültigkeitsbereichs des Objekts

- Zugriffsrechte in Klassen:
  - **`public`**: für alle sichtbar, Schnittstelle der Klasse zur „Außenwelt“
  - **`private`**: nur für für Objekte der gleichen Klasse sichtbar, Variablen sollten immer private deklariert werden
  - **`protected`**: auch Objekte aus abgeleiteten Klassen haben Zugriff, hier nicht verwendet

Vector-/Matrix-Klassen

Karlsruher Institut für Technologie

Todo | Machen Sie sich mit der Struktur einer einfachen Klasse vertraut

- Laden Sie von ILIAS '**vec_mat.tgz**' herunter und entpacken sie es in einem Ordner /home/…/UB05/
- Erstellen Sie daraus in **Eclipse** ein neues **Makefile Projekt**
- In diesem Projekt werden die beiden Klassen **Vector** und **Matrix** implementiert
- Die Klasse **Vector** ist bereits ausreichend implementiert, öffnen Sie vector_func.h und vector_func.cpp und beantworten Sie folgende Fragen:
  - Aus welchen Membervariablen besteht die Klasse Vector?
  - Wie werden die Vectoreinträge gespeichert?
  - Welche Arten von Konstruktoren gibt es?
  - Warum ist der Default-Konstruktor `Vector();` auf `private` gesetzt? Wie kann man das noch und besser in c++11 realisieren?
  - Wie werden die Membervariablen im Konstruktor erzeugt?
  - Wie wird intern auf die Membervariablen zugegriffen?
  - Welche Aufgabe hat der Destruktor?

  - Welche Operatoren werden für Vector überladen?
  - Wie kann von außen auf die Membervariablen zugegriffen werden (lesen/schreiben)?

Vector-Klasse: Konstruktor und Destruktor

Info

vector_funcs.h

```
class Vector { //Beginn der Vector Klasse

public:
   //Constructor
    Vector ( const std::size_t dim );
    //Copy-Constructor
    Vector( const Vector & c );
    //Destructor
    ~Vector ();
              ...

private:


    //Default Constructor: Hier PRIVATE
um Ausführen des Default Constructor zu
unterbinden
    Vector ();


    double * dataPtr;  // pointer for
data
    std::size_t      N;          // number
of entries in the vector
```

vector_funcs.cpp

```
#include "vector_funcs.h"

using namespace std;
//Constructor
Vector::Vector ( const std::size_t dim ){
  this->N = dim;
  this->dataPtr = new double[dim];
};

//Copy-Constructor
Vector::Vector( const Vector & c ){
  this->N = c.N;
  this->dataPtr = new double[ this->N ];
  //kopiere komponentenweise den Vektor c
  for (std::size_t nx=0; nx<c.N; ++nx)
    this->dataPtr[ nx ] = c.dataPtr[ nx ];
};

//Destructor
Vector::~Vector (){
  delete [] this->dataPtr;
};
```

Zugriff:
- mit `this->` können Membervariablen angesprochen werden
- this ist ein Pointer auf das aufrufende Objekt

Konstruktor:
- Größe des Vektors `N` festgelegt
- 1D Feld der Größe `dim` wird mit `new` angelegt

Copy-Konstruktor:
- Größe des Vektors `N` wird kopiert
- 1D Feld der Größe `N` wird angelegt
- Werte des Vektors `c` werden kopiert

Destructor:
- alle mit `new` angelegten Objekte müssen explizit mit `delete` gelöscht werden!!!

Membervariablen:
- `dataPtr` zeigt auf das 1D-Feld
- `N` ist Größe des Vektors

Vector-Klasse: Operatoren und Memberfunktionen

Info

vector_funcs.h

```
//Return size of Vector
const std::size_t size() const;
```

Memberfunktionen: Zugriff auf Vector von außen

```
//Ausgabe aller Einträge
void print(const std::string title="", const int w=8) const;
 //const int w=8 default Wert für Abstand. Übergabe bei Aufruf optional
//Copy Operator
Vector & operator= ( const Vector & c );
//Initialisierung mit Skalar
Vector & operator=(const double value);
//Index Operator Reading
const double  operator() ( std::size_t n ) const;
//Index Operator Writing
double & operator() ( std::size_t n );
//Vektor-Addition
const Vector operator+ (const Vector & c2) const;
//Vektor-Subtraktion
const Vector operator- (const Vector & c2) const;
//Skalarprodukt
const double operator* (const Vector & c2) const;
//Multiplikation mit Skalar
const Vector operator* ( const double & lambda) const;
```

Operatoren:
Hier sind eine Reihe von gängigen Operatoren für die Klasse Vector definiert

- Kopieren
- ...
- Lesen/Schreiben
- Vektor-Addition/Subtraktion
- Skalarprodukt
- Multiplikation mit Skalar c

Matrix-Klasse

Info

matrix_funcs.h: Aufbau und Schnittstellen

```
class Matrix {
 public:
  //Constructor TODO: Selber machen
  Matrix (std::size_t const rows, std::size_t const cols);
  //Copy-Constructor TODO: Selber machen
  Matrix (const Matrix & m);
  //Destructor: TODO: Selber machen
  ~Matrix ();
  //Copy Operator
  Matrix & operator= ( const Matrix & m );
  //Fill with same Value
  Matrix & operator= ( const double val );
  //Index Operator Reading
  const double   operator() ( const std::size_t nx, const std::size_t ny ) const;
  //Index Operator Writing TODO: Selber machen
  double & operator() ( const std::size_t nx, const std::size_t ny );
  //Matrix-Vector Produkt
  const Vector operator* ( const Vector & b ) const;
  //Matrix Addition TODO: Selber machen
  const Matrix operator+ ( const Matrix & A) const;
  const std::size_t rows() const;
  const std::size_t cols() const;
  //Ausgabe Matrix
  void print(const int w=8) const; //const int w=8 default Wert für Abstand. Übergabe bei Aufruf optional

 private:
  //Default Constructor: Hier PRIVATE um des Ausführen Default Constructor zu unterbinden
  Matrix ();
  double ** dataPtr;
  int _rows, _cols;
};
```

Konstruktoren & Destruktor:

- Müssen selbst implementiert werden (siehe nächste Folie)
- Regeln für dynamische Variablendeklaration beachten

Memberfunktionen & Operatoren:

- Methode zum Verändern einzelner Matrixeinträge muss implementiert werden
- Addition zweier Matrizen muss ebenfalls implementiert werden

Membervariablen:

- `dataPtr`: Pointer für das 2D-Feld
- `_rows, _cols` ist Größe der Matrix

Matrix-Klasse

Todo

Ergänzen Sie die Matrix-Klasse

- In der Headerdatei **matrix_funcs.h** ist die Matrix-Klasse und ihr Schnittstellen definiert. Ergänzen Sie in **matrix_funcs.cpp** fehlenden Implementierungen:

1. Konstruktor: Legen Sie ein 2D Feld **mit dynamischer Variablendeklaration** an
2. Copy-Konstruktor: Kopieren der Werte eine vorhandenen Matrix
3. Destruktor: Beachten Sie, dass jedes **`new`** im Konstruktor ein eigenes **`delete`** im Destruktor erfordert

4. Operator `( )` für die Matrixklasse: wodurch unterscheidet sich der Operator, der nur Lesen bzw. auch Schreiben erlaubt? Sind beide notwendig?

5. Implementieren Sie den Operator + für eine Matrix-Matrix Addition

6. **Testen Sie die Funktionalitäten** der implementierten Methoden und Operatoren beider Klassen in **vec_mat_test.cpp** und überlegen Sie sich weitere Testfälle.

3. Lösen von linearen Gleichungssystemen

Info

Siehe auch Vorlesung

- Aus der Vorlesung kennen Sie Lösungsverfahren für lineare Gleichungssysteme (LGS)
- In dieser Übung werden der direkte **LU-Solver** und der iterative **CG-Solver** verwendet
- Für das Lösen von Gleichungssystemen benötigen Sie Vektoren und Matrizen, dafür haben Sie die beiden Klassen **Vector** und **Matrix** implementiert
- Für den GC-Solver benötigen Sie **symmetrische Matrizen**

Todo

Machen Sie sich mit den Solvern vertraut

- Laden Sie **`solver.tgz`** herunter, entpacken Sie es und legen in Eclipse ein neues Makefile Projekt an
- Kopieren Sie Ihre Lösungen für **`vector_funcs.h/.cpp`** und **`matrix_funcs.h/.cpp`** ebenfalls in den Ordner **`solver`**
- Testen Sie die Solver in **`main_programm_solver.cpp`**
- Mit der Methode **`write_vector_to_file`** können Sie die Lösungsvektoren in eine Datei schreiben

Anwendung: Das Wärmeleitproblem

Info

Siehe auch Vorlesung S. 150f

- Das 1D-Wärmeleitproblem aus der Vorlesung können Sie mit Hilfe der gegebenen Solver und der Vector-/Matrix-Klassen lösen
- Diskretisieren Sie das Problem mit Hilfe der **Finite Differenzen**
- Beachten Sie, dass Sie bei manchen iterativen Verfahren mit **symmetrischen Matrizen** arbeiten müssen; wenn Sie das LGS aufstellen, wird die Matrix zunächst **unsymmetrisch** sein
- Der Stab hat die Länge `L`
- Die Randbedingungen sind: T_0 (Temperatur am Anfang des Stabs), T_L (Temperatur am Ende des Stabs), T_e (Umgebungstemperatur)
- Der Wärmeleitkoeffizient ist `k`

Anwendung: Das Wärmeleitproblem

Todo

Lösen Sie das Wärmeleitproblem

- Erweitern Sie **`main_programm_solver.cpp`** so, dass Sie das Wärmeleitproblem lösen können.
- Als Rand-/Umgebungsbedingungen wählen Sie die Temperaturen T_0, T_L und T_e.
- Diskretisieren Sie das Problem mit M Stützstellen: T_i die Temperatur ab Stützstelle i .
- Das Gleichungssystem hat die Form $A_{ij}T_j = b_i$
- Durch **Anpassung der rechten Seite** (b_i) der Gleichung können Sie eine symmetrische Matrix erhalten:
  - Da $\mathrm{T}(\mathrm{x}=0)=\mathrm{T}_0$ und $\mathrm{T}(\mathrm{x}=L)=\mathrm{T}_\mathrm{M}=\mathrm{T}_\mathrm{L}$ können Sie die Matrixeinträge A_{i0} und A_{iM} auf der rechten Seite b_i berücksichtigen.
  - **Stellen Sie das das LGS zunächst für wenige Stützstellen „von Hand“ auf Papier auf und überlegen Sie sich die notwendigen Anpassungen der Matrix und rechten Seite der Gleichung!!**
- Lösen Sie das Gleichungssystem mit den mitgelieferten Solvern und lassen Sie sich die Lösungen in Dateien ausgeben.
- Variieren Sie die Parameter und plotten die Temperaturverläufe mit gnuplot.