Code Dokumentation - Wissenschaftliches Programmieren für Ingenieure im Wintersemester 2018/2019

Jens Weber - Matrikel Nummer: 1851194

18.12.2018

Inhaltsverzeichnis

Kapitel 1

Aufgabe 1: Rechnen mit komplexen Zahlen

1.1 Verwendung des Codes

Der für diese Aufgabe verwendete Code ist in 1.3 zu finden. Er lässt sich mit dem Makefile kompilieren. Nach dem das kompilieren erfolgreich abgeschlossen ist, startet man das Programm mit:

```
$ ./aufgabe_1
```

Die Berechnungsergebnisse werden im Terminal ausgegeben.

1.2 Ergebnisse der Berechnung

Hier sind die Ergebnisse der Berechnung (**Aufgabenteil A)**), ausgehend von:

$$z_1 = 2 + 7 * i \tag{1.1}$$

$$z_2 = 42 - 9 * i \tag{1.2}$$

$$z_3 = -11 + 19 * i \tag{1.3}$$

, tabellarisch in 1.2 aufgeführt.

Formel	Ergebnis
$z_4 = z_1 * z_2$	$147 + 276 * i$
$z_5 = (z_1 + z_2)$	$44 - 2 * i$
$z_6 = (z_1 + z_2) * 2$	$88 - 4 * i$
$z_7 = (z_2 + z_3) * z_1$	$-8 + 237 * i$
$z_8 = z_1 + 5$	$7 + 7 * i$
$z_9 = -z_1 + z_2$	$40 - 16 * i$

1.2.1 Code in Operatorschreibweise

Entsprechend Aufgabenteil B wird hier der Code zur Berechnung der Ergebnisse in Operatorschreibweise als Quellcode aufgelistet.
Eine ausführbares Programm findet sich im Ordner *aufgabe_1* mit dem Namen *main_complex_operator_schreibweise.cpp*

```
z4.operator=(z1.operator*(z2));
z5.operator=((z1.operator+(z2)));
z6.operator=((z1.operator+(z2)).operator*(2.));
z7.operator=((z2.operator+(z3)).operator*(z1));
z8.operator=(z1.operator+(5.));
z9.operator=(z1.operator-().operator+(z2));
```

1.3 Code

Im Folgenden ist der komplette Code, inklusive Makefile zu Aufgabe 1 zu finden.

complex.h:

```
#ifndef MYCOMPLEXNUM_
#define MYCOMPLEXNUM_

class MyComplex{

public:

  // Konstruktoren
  MyComplex();
  MyComplex(const double xVal, const double yVal);
  MyComplex(const MyComplex & complexNumber);

  // Destruktoren
  ~MyComplex();

  // Operatoren
  const MyComplex operator+(const MyComplex & additionComplex) const;
  const MyComplex operator+(const double additionConstant) const;
  const MyComplex operator-(const MyComplex & subtractionComplex) const;
  const MyComplex operator-(const double subtractionConstant) const;
  const MyComplex operator*(const MyComplex & multiplicationComplex) const;
  const MyComplex operator*(const double multiplicationConstant) const;
  MyComplex & operator-();
            // Fuer Aufgabe 2
  const MyComplex operator^(const int a);
  const MyComplex operator/(MyComplex & complexDivisionNum) const;
  MyComplex & operator=(const MyComplex & assignmentComplex);
```

```
  // Methoden
            // Getter-Methoden
  const double norm() const;
  const double real() const;
  const double imag() const;
            // Setter Methoden
  void setRe(double newRe);
  void setIm(double newIm);

  void printComponents(); // zum debuggen

private:
  // Membervaribalen: z = x+iy
  double x;
  double y;
};
#endif
```

complex.cpp:

```
#include <iostream>
#include <cmath>

#include "complex.h"

using namespace std;

// Construktoren
MyComplex::MyComplex(){
  // Standartkonstruktor
};

MyComplex::MyComplex(const double xVal, const double yVal){
  this->x = xVal;
  this->y = yVal;
};

MyComplex::MyComplex(const MyComplex & complexNumber){
  this->x = complexNumber.x;
  this->y = complexNumber.y;
};

// Destruktor
MyComplex::~MyComplex(){
  // Leerer Destruktor!
};


// Operatoren

const MyComplex MyComplex::operator+(const MyComplex & additionComplex) const{
  MyComplex resultAdditionCompl;
        resultAdditionCompl.x = this->x + additionComplex.real();
        resultAdditionCompl.y = this->y + additionComplex.imag();
        return resultAdditionCompl;
};

const MyComplex MyComplex::operator+(const double additionConstant) const{
  MyComplex resultAddConst(*this);
        resultAddConst.x = resultAddConst.x + additionConstant;
        return resultAddConst;
}

const MyComplex MyComplex::operator-(const MyComplex & subtractionComplex) const{
```

```
  MyComplex resultSubtractionCompl;
        resultSubtractionCompl.x = this->x - subtractionComplex.real();
        resultSubtractionCompl.y = this->y - subtractionComplex.imag();
        return resultSubtractionCompl;
};

const MyComplex MyComplex::operator-(const double subtractionConstant) const{
  MyComplex resultSubConst(*this);
        resultSubConst.x = resultSubConst.x + subtractionConstant;
        return resultSubConst;
}

const MyComplex MyComplex::operator*(const MyComplex & multiplicationComplex) const{
        MyComplex resultMultiplCompl;
        resultMultiplCompl.x = this->x*multiplicationComplex.real()
                              -this->y*multiplicationComplex.imag();
        resultMultiplCompl.y = this->x*multiplicationComplex.imag()
                               +multiplicationComplex.real()*this->y;
        return resultMultiplCompl;
};

const MyComplex MyComplex::operator*(const double multiplicationConstant) const{
  MyComplex resultMultiplicationConst(*this);
        resultMultiplicationConst.x = resultMultiplicationConst.x
                                      *multiplicationConstant;
        resultMultiplicationConst.y = resultMultiplicationConst.y
                                      *multiplicationConstant;
        return resultMultiplicationConst;
};

const MyComplex MyComplex::operator/(MyComplex& complexDivisionNum) const{
        MyComplex divisionResult;
        divisionResult.x = (this->x * complexDivisionNum.x + this->y
                           * complexDivisionNum.y) / (complexDivisionNum.x
                               * complexDivisionNum.x + complexDivisionNum.y
                                   * complexDivisionNum.y);
        divisionResult.y = (this->x * complexDivisionNum.x - this->x
                           * complexDivisionNum.y) / (complexDivisionNum.x
                               * complexDivisionNum.x
                                   + complexDivisionNum.y
                                       * complexDivisionNum.y);
        return divisionResult;
};


MyComplex & MyComplex::operator=(const MyComplex & assignmentComplex){
```

```
        this->x = assignmentComplex.real();
        this->y = assignmentComplex.imag();
        return *this;
};

MyComplex & MyComplex::operator-(){
        this->x = -this->x;
        this->y = -this->y;
        return *this;
};

const MyComplex MyComplex::operator^(const int a){
        MyComplex resultPow(*this);
        if (a > 0){
                for (int i = 1; i < a; i++){
                        resultPow = resultPow * *this;
                }
        }
        else if (a == 0){
                resultPow.x = 1;
                resultPow.y = 0;
        }
        else{
                resultPow = resultPow / *this;
        }
        return resultPow;
};


// Methoden
const double MyComplex::norm() const{
        return sqrt(this->x*this->x + this->y*this->y);
};

const double MyComplex::real() const{
  return this->x;
};

const double MyComplex::imag() const{
  return this->y;
};

void MyComplex::setRe(double newRe){
        this->x = newRe;
};
```

```
void MyComplex::setIm(double newIm){
        this->y = newIm;
};

void MyComplex::printComponents(){
        cout << "x: " << this->x << endl;
        cout << "y: " << this->y << endl;
};
```

main_complex_beispiel.cpp:

```
#include <iostream>
#include <string>

#include "complex.h"

using namespace std;

void output_my_cplx(const MyComplex &c,const std::string txt){
    cout<<txt<<": ("<<c.real()<<", "<<c.imag()<<")"<<endl;
}

int main(){
    MyComplex z1 {2.,7.};
    MyComplex z2 {42.,-9};
    MyComplex z3 {-11.,19.};
    MyComplex z4,z5,z6,z7,z8,z9;

    output_my_cplx(z1,"z1 ");
    output_my_cplx(z2,"z2 ");
    output_my_cplx(z3,"z3 ");

    z4=z1*z2;
    output_my_cplx(z4,"z4=z1*z2 =");
    z5=(z1+z2);
    output_my_cplx(z5,"z5=(z1+z2) =");
    z6=(z1+z2)*2.;
    output_my_cplx(z6,"z6=(z1+z2)*2. = ");
    z7=(z2+z3)*z1;
    output_my_cplx(z7,"z7=(z2+z3)*z1 = ");
    z8=z1+5.;
    output_my_cplx(z8,"z8=z1+5. = ");
    z9=-z1+z2;
    output_my_cplx(z9,"z9=-z1+z2 = ");

    return 0;
}
```

Makefile:

```
# Variablen
PROG = aufgabe_1

FLAGS = -O2

CC = g++

SRC = complex.cpp main_complex_operator_schreibweise.cpp

OBJ = $(SRC:.cpp=.o)

# Targets
all: $(SRC) $(PROG)

$(PROG): $(OBJ)
        $(CC) $(FLAGS) $(OBJ) -o $@


%.o:%.cpp
        $(CC) $(FLAGS) -c $<

clean:
         rm *.o $(PROG)

# deps
complex.o: complex.cpp complex.h
main_complex_beispiel.o: main_complex_beispiel.cpp complex.h
```

Kapitel 2

Aufgabe 2: Untersuchung der Konvergenz komplexer Zahlenfolgen

2.1 Verwendung des Codes

Der Code in Ordner *aufgabe_2* lässt sich mit dem enthaltenen Makefile bauen. Das entstandene Programm lässt sich auf drei Arten starten:

1. Direkte Nutzereingabe der auszuwertenden Parameter
2. Übergabe der gegebenen Start-Dateien mittels des Operators ¡ .
3. Mittels des angefügten Shell-Skripts (*run_all.sh*), welches in 2.3 zu finden ist.

Nachdem die Berechnungen abgeschlossen sind, können die Ergebnisbilder mit dem ebenfalls in 2.3 angefügten Gnuplot-Skript erstellt werden.
Dafür wird Gnuplot aufgerufen:

```
$ gnuplot
```

Und anschließend das Skript zum erstellen der Ergebnisbilder aufgerufen:

```
> load 'plot_result.gp'
```

Die Resultate befinden sich nun im Ordner *aufgabe_2*.

2.2 Ergebnisse der Konvergenzanalyse

Die Ergebnisse der Konvergenzanalyse mit allen Datensätzen ist im Folgenden aufgeführt:

start1A.dat:

Iterationsvorschrift	1
Wertebereich	$(-1.5, -1) - (1.5, 1)$
Unterteilung (Nxmax, Nymax)	$750, 500$
Exponent	2
Nmax	2000
Rc	100
Komplexe Konstante c0	$(-0.75, 0.1)$

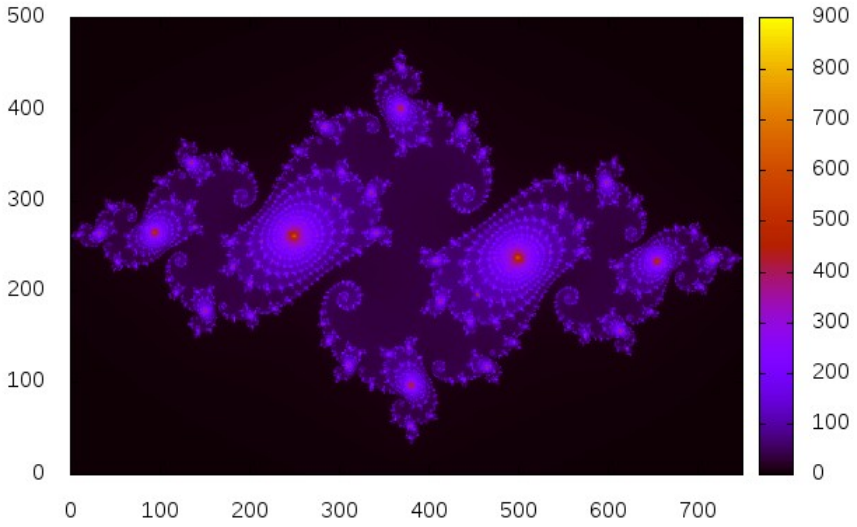

Abbildung 2.1: Ergebnis des Datensatz 1A

start1B.dat:

Iterationsvorschrift	1
Wertebereich	$(-1.5, -1) - (1.5, 1)$
Unterteilung (Nxmax, Nymax)	$750, 500$
Exponent	2
Nmax	2000
Rc	100
Komplexe Konstante c0	$(-0.75, 0.55)$

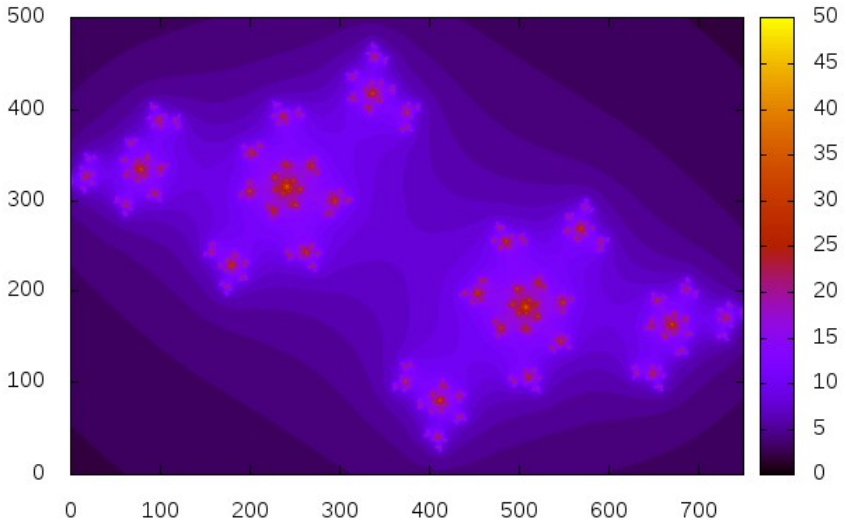

Abbildung 2.2: Ergebnis des Datensatz 1B

start2A.dat:

Iterationsvorschrift	2
Wertebereich	$(-2,-1)-(1,1)$
Unterteilung (Nxmax, Nymax)	$750, 500$
Exponent	2
Nmax	200
Rc	2
Komplexe Konstante c0	$(0,0)$

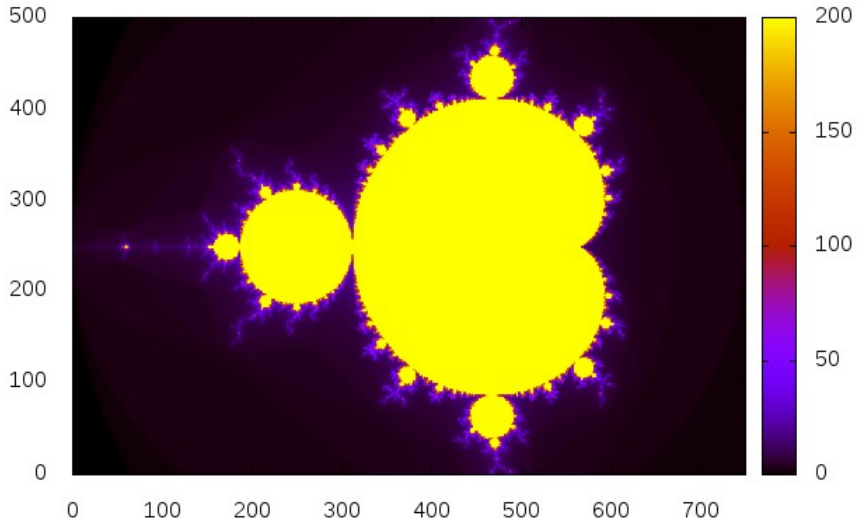

Abbildung 2.3: Ergebnis des Datensatz 2A

start2B.dat:

Iterationsvorschrift	2
Wertebereich	$(-1.5, 1) - (0, 0)$
Unterteilung (Nxmax, Nymax)	$750, 500$
Exponent	2
Nmax	200
Rc	2
Komplexe Konstante c0	$(0, 0)$

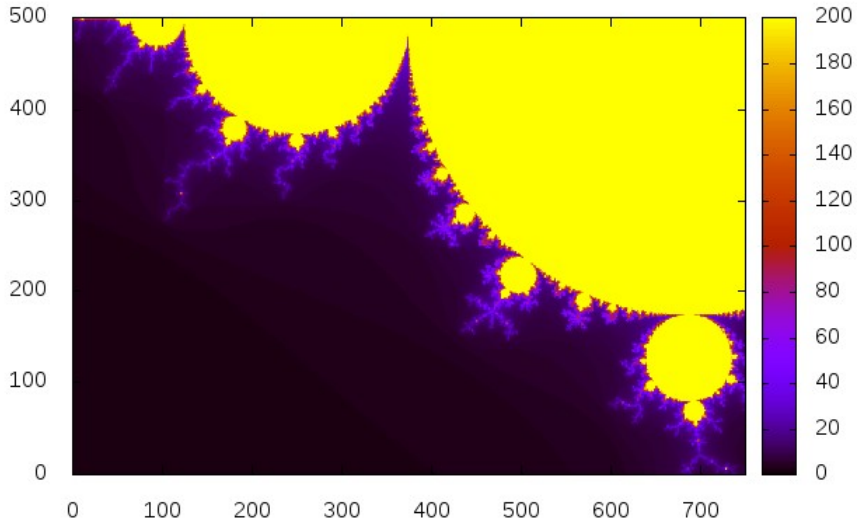

Abbildung 2.4: Ergebnis des Datensatz 2B

start3A.dat:

Iterationsvorschrift	3
Wertebereich	$(-1.5, -1) - (1.5, 1)$
Unterteilung (Nxmax, Nymax)	$750, 500$
Exponent	4
Nmax	2000
Rc	200
Komplexe Konstante c0	$(0, 0)$

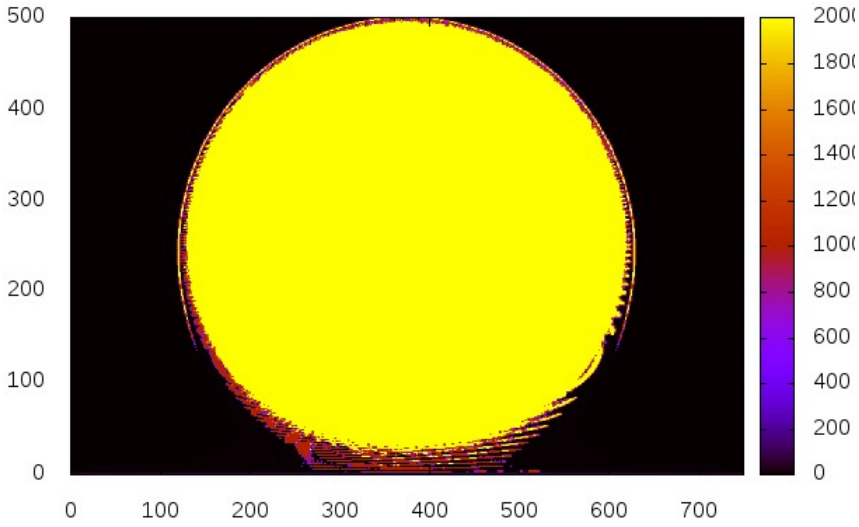

Abbildung 2.5: Ergebnis des Datensatz 3A

2.3 Code

Hier wird der Code für Aufgabe 2, inklusiver des Makefiles und der Shell, bzw Gnuplot Skripte aufgeführt:

complex.h:

```
#ifndef MYCOMPLEXNUM_
#define MYCOMPLEXNUM_
```

```
class MyComplex{

public:

  // Konstruktoren
  MyComplex();
  MyComplex(const double xVal, const double yVal);
  MyComplex(const MyComplex & complexNumber);

  // Destruktoren
  ~MyComplex();

  // Operatoren
  const MyComplex operator+(const MyComplex & additionComplex) const;
  const MyComplex operator+(const double additionConstant) const;
  const MyComplex operator-(const MyComplex & subtractionComplex) const;
  const MyComplex operator-(const double subtractionConstant) const;
  const MyComplex operator*(const MyComplex & multiplicationComplex) const;
  const MyComplex operator*(const double multiplicationConstant) const;
  MyComplex & operator-();
            // Fuer Aufgabe 2
  const MyComplex operator^(const int a);
  const MyComplex operator/(MyComplex & complexDivisionNum) const;
  MyComplex & operator=(const MyComplex & assignmentComplex);


  // Methoden
            // Getter-Methoden
  const double norm() const;
  const double real() const;
  const double imag() const;
            // Setter-Methoden
  void setRe(double newRe);
  void setIm(double newIm);

  void printComponents(); // zum debuggen

private:
  // Membervaribalen: z = x+iy
  double x;
  double y;
};
#endif
```

complex.cpp:

```
#include <iostream>
#include <cmath>

#include "complex.h"

using namespace std;

// Construktoren
MyComplex::MyComplex(){
  // Standartkonstruktor
};

MyComplex::MyComplex(const double xVal, const double yVal){
  this->x = xVal;
  this->y = yVal;
};

MyComplex::MyComplex(const MyComplex & complexNumber){
  this->x = complexNumber.x;
  this->y = complexNumber.y;
};

// Destruktor
MyComplex::~MyComplex(){
};


// Operatoren

const MyComplex MyComplex::operator+(const MyComplex & additionComplex) const{
  MyComplex resultAdditionCompl;
        resultAdditionCompl.x = this->x + additionComplex.real();
        resultAdditionCompl.y = this->y + additionComplex.imag();
        return resultAdditionCompl;
};

const MyComplex MyComplex::operator+(const double additionConstant) const{
  MyComplex resultAddConst(*this);
        resultAddConst.x = resultAddConst.x + additionConstant;
        return resultAddConst;
}

const MyComplex MyComplex::operator-(const MyComplex& subtractionComplex) const{
  MyComplex resultSubtractionCompl;
```

```
        resultSubtractionCompl.x = this->x - subtractionComplex.real();
        resultSubtractionCompl.y = this->y - subtractionComplex.imag();
        return resultSubtractionCompl;
};

const MyComplex MyComplex::operator-(const double subtractionConstant) const{
  MyComplex resultSubConst(*this);
        resultSubConst.x = resultSubConst.x + subtractionConstant;
        return resultSubConst;
}

const MyComplex MyComplex::operator*(const MyComplex& multiplicationComplex) const{
        MyComplex resultMultiplCompl;
        resultMultiplCompl.x = this->x*multiplicationComplex.real()
                          -this->y*multiplicationComplex.imag();
        resultMultiplCompl.y = this->x*multiplicationComplex.imag()
                          +multiplicationComplex.real()*this->y;
        return resultMultiplCompl;
};

const MyComplex MyComplex::operator*(const double multiplicationConstant) const{
  MyComplex resultMultiplicationConst(*this);
        resultMultiplicationConst.x = resultMultiplicationConst.x
                                *multiplicationConstant;
        resultMultiplicationConst.y = resultMultiplicationConst.y
                                *multiplicationConstant;
        return resultMultiplicationConst;
};

const MyComplex MyComplex::operator/(MyComplex & complexDivisionNum) const{
        MyComplex divisionResult;
        divisionResult.x = (this->x * complexDivisionNum.x + this->y
                        * complexDivisionNum.y) / (complexDivisionNum.x
                          * complexDivisionNum.x + complexDivisionNum.y
                            * complexDivisionNum.y);
        divisionResult.y = (this->x * complexDivisionNum.x - this->x
                        * complexDivisionNum.y) / (complexDivisionNum.x
                          * complexDivisionNum.x + complexDivisionNum.y
                            * complexDivisionNum.y);
        return divisionResult;
};


MyComplex & MyComplex::operator=(const MyComplex & assignmentComplex){
        this->x = assignmentComplex.real();
        this->y = assignmentComplex.imag();
```

```
        return *this;
};

MyComplex & MyComplex::operator-(){
        this->x = -this->x;
        this->y = -this->y;
        return *this;
};

const MyComplex MyComplex::operator^(const int a){
        MyComplex resultPow(*this);
        if (a > 0){
                for (int i = 1; i < a; i++){
                        resultPow = resultPow * *this;
                }
        }
        else if (a == 0){
                resultPow.x = 1;
                resultPow.y = 0;
        }
        else{
                resultPow = resultPow / *this;
        }
        return resultPow;
};


// Methoden
const double MyComplex::norm() const{
        return sqrt(this->x*this->x + this->y*this->y);
};

const double MyComplex::real() const{
  return this->x;
};

const double MyComplex::imag() const{
  return this->y;
};

void MyComplex::setRe(double newRe){
        this->x = newRe;
};

void MyComplex::setIm(double newIm){
        this->y = newIm;
```

```
};

void MyComplex::printComponents(){
        cout << "x: " << this->x << endl;
        cout << "y: " << this->y << endl;
};
```

main_convergenz.cpp:

```
# Variablen
PROG = aufgabe_1

FLAGS = -O2

CC = g++

SRC = complex.cpp main_complex_operator_schreibweise.cpp

OBJ = $(SRC:.cpp=.o)

# Targets
all: $(SRC) $(PROG)

$(PROG): $(OBJ)
        $(CC) $(FLAGS) $(OBJ) -o $@


%.o:%.cpp
        $(CC) $(FLAGS) -c $<

clean:
         rm *.o $(PROG)

# deps
complex.o: complex.cpp complex.h
main_complex_beispiel.o: main_complex_beispiel.cpp complex.h
```

Makefile:

```
# Variablen
PROG = aufgabe_2

FLAGS = -O2

CC = g++

SRC = complex.cpp main_konvergenz.cpp

OBJ = $(SRC:.cpp=.o)

# Targets
all: $(SRC) $(PROG)


$(PROG): $(OBJ)
        $(CC) $(FLAGS) $(OBJ) -o $@


%.o:%.cpp
        $(CC) $(FLAGS) -c $<

clean:
         rm *.o $(PROG)

# deps
complex.o: complex.cpp complex.h
main_konvergenz: main_konvergenz.cpp complex.h
```

Shell Skript:

```
#!/bin/bash

# loesche alle alten Dateien
rm ergebnis*

# Berechnung aller 5 Datensaetze
for i in $(seq 1 3);
do
    NAMECURRENTFILEA=start$i"A.dat"
    NAMECURRENTFILEB=start$i"B.dat"
    ./aufgabe_2 < $NAMECURRENTFILEA
    echo "finished run "$i"A"

    if [ -e $NAMECURRENTFILEB ]
    then
      ./aufgabe_2 < "start"$i"B.dat"
      echo "finished run "$i"B"
    else
      echo "File start"$i"B.dat does not exists"
    fi
done

echo "finished all runs"
```

plot_result.gp:

```
set pm3d map

do for [ii=1:3:1] {
  file=sprintf('ergebnis%dA.dat',ii)

  resultName=sprintf('ergebnis%dA.jpg',ii)
  set term jpeg
  set output resultName

  set xrange[0:750]
  set yrange[0:500]

  splot file u ($1):($2):($3)

  if (ii < 3) {
    file=sprintf('ergebnis%dB.dat',ii)

    resultName=sprintf('ergebnis%dB.jpg',ii)
    set term jpeg
    set output resultName

    splot file u ($1):($2):($3)
  }
}
```