Einführung in die Programmierung für Studierende der Physik

JOHANN WOLFGANG GOETHE UNIVERSITÄT
10.05.2022

MATTHIAS HANAUSKE

FRANKFURT INSTITUTE FOR ADVANCED STUDIES
JOHANN WOLFGANG GOETHE UNIVERSITÄT
INSTITUT FÜR THEORETISCHE PHYSIK
ARBEITSGRUPPE RELATIVISTISCHE ASTROPHYSIK
D-60438 FRANKFURT AM MAIN
GERMANY

5. Vorlesung

Plan für die heutige Vorlesung

- Kurze Wiederholung der Vorlesung 4
- C++ Arrays, Zeiger und Referenzen
- Anwendungsbeispiel: Interpolation und Polynomapproximation
- Übungsaufgaben: Übungsblatt Nr.5
- Aus aktuellem Anlass (Donnerstag, der 12.05.2022 ab 15.00 Uhr)
  Liveübertragung der ESO-Pressekonferenz
  Neues vom Schwarzen Loch im Zentrum der Milchstrasse

Vorlesung 4

In dieser Vorlesung werden wir zunächst die *Auswahlanweisungen* der Sprache C++ vorstellen und danach auf die Definition von *C++ Funktionen* eingehen. In dem Anwendungsbeispiel *Nullstellensuche einer Funktion* werden dann die erlernten Konzepte am Beispiel der *Methode der Bisektion*, dem sogenannten *Intervallhalbierungsverfahren* verdeutlicht. Zusätzlich wird am Ende die *Newton-Raphson Methode* der Nullstellenermittlung vorgestellt.

C++ Anweisungen: Auswahlanweisungen mit if und switch

Die Programmiersprache C++ bietet einen konventionellen und flexiblen Satz von Anweisungen. Im Prinzip ist jede Programmzeile, die mit einem Semikolon endet, eine Anweisung. In diesem Unterpunkt werden wir die sogenannten *Auswahlanweisungen* behandeln, wobei wir in der vorigen Vorlesung die *Schleifenanweisungen* vorstellt hatten (siehe C++ Anweisungen: Die for-, while- und do-Schleifen). *Auswahlanweisungen* werden auch als Verzweigung , Tests oder *bedingte Anweisungen* bezeichnet und sind immer dann anzuwenden, wenn das Programm bei einem gewissen Ereignis bzw. unter einer gewissen Bedingung etwas Bestimmtes tun soll. Es teilt somit das Programm in unterschiedliche Anweisungspfade auf. *Auswahlanweisungen* können in Form einer **if**-, (**if-else**)- oder **switch**-Anweisung ausgedrückt werden. C++ Anweisungen: Auswahlanweisungen mit if und switch

Funktionen in C++

Die Definition einer C++ Funktion ist im Grunde nichts Anderes, als eine Code-Block (*Anweisungsblock*) mit einem Funktionsnamen zu verbinden. C++ Funktionen werden außerhalb der main()-Hauptfunktion definiert und vereinfachen somit das Verständnis und die Lesbarkeit des Quelltextes. C++ Funktionen sind ein wichtiges Werkzeug, um den Quelltext eines Programms zu ordnen und wesentliche Algorithmen und zusammenhängende Anweisungsblöcke der main()-Hauptfunktion in einer zusammenhängenden Form auszulagern. Die Definition einer C++ Funktion besteht aus einer *Deklaration* und einem *Anweisungsblock* und sie ist der formalen Definition einer mathematischen Funktion nicht unähnlich: "Eine C++ Funktion ist eine Abbildung von dem Datenraum der *Argumentenliste* in den Datenraum des *Rückgabetyps*. Die dabei benutzte Abbildungsvorschrift findet sich in dem *Anweisungsblock* der Funktion. Der 'Rückgabe Typ' kann hierbei eine der schon besprochenen Datentypen (z.B. int oder double) oder ein Daten-Array (siehe nächste Vorlesung) sein. Die *Argumentenliste* setzt sich aus einer Liste von Datentypen der formalen Argumente (Parameter) der Funktion zusammen, die jeweils mit einem Komma voneinander getrennt sind. In C++ hat das Wort Funktion eine allgemeinere Bedeutung als im Bereich der Mathematik und die in der Mathematik und Physik definierte Funktionen stellen eine echte Teilmenge der C++ Funktionen dar (näheres siehe Funktionen in C++).

Anwendungsbeispiel: Nullstellensuche einer Funktion

Vorlesung 4

Die möglichen integrierten Anweisungsbefehle innerhalb einer Programmiersprache sind die wohl wichtigsten Grundvokabeln, die man als Programmierer kennen muss. In der vorigen Vorlesung hatten wir uns bereits mit den *Schleifenanweisungen* befasst und in dieser Vorlesung werden wir uns mit den *Auswahlanweisungen* beschäftigen. Im Speziellen werden die **if**-Anweisung, die (**if-else**)-Anweisung und die **switch**-Anweisung besprochen. *Auswahlanweisungen* stellen eine Art von Programmverzweigungen dar und sind immer dann anzuwenden, wenn das Programm bei einem gewissen Ereignis etwas Bestimmtes tun soll.
Ein weiteres wichtiges Konzept bei der Erstellung eines C++ Quelltextes ist der Begriff der "Funktion". *C++ Funktionen* sind ein wichtiges Werkzeug, um den Quelltext eines Programms zu ordnen und wesentliche Algorithmen und zusammenhängende Anweisungsblöcke der main()-Hauptfunktion in einer zusammenhängenden Form auszulagern. Man könnte salopp sagen, dass Funktionen kleine Unterprogramme sind, die definierte Teilprobleme lösen. Im Grunde bedeutet die Definition einer Funktion im Programm nichts Anderes, als eine Code-Block (*Anweisungsblock*) mit einem Funktionsnamen zu verbinden. In C++ hat das Wort Funktion eine allgemeinere Bedeutung als im Bereich der Mathematik und die in der Mathematik und Physik definierte Funktionen stellen eine echte Teilmenge der C++ Funktionen dar. Wir werden in dieser Vorlesung lediglich eine erste grundsätzliche Einführung in den Themenbereich der *C++ Funktionen* geben und auf die allgemeinere Verwendung von Funktionen im Laufe der Vorlesung noch genauer eingehen. Gerade in den Unterpunkten, die sich mit der Objekt-orientierten Programmierung befassen, sind Klassen-Funktionen, Konstruktoren und das Überladen von Funktionen ein wichtiges Thema.
Am Ende dieser Vorlesung wenden wir das Erlernte an und besprechen eines der grundlegenden Probleme der numerischen Mathematik: *Die Nullstellensuche einer Funktion*. Wir nehmen dabei an, dass eine Funktion $f(x)$ im Intervall $[a, b] \in \mathbb{R}$ eine Nullstelle hat und berechnen dann diese Nullstelle numerisch, approximativ mittels der *Methode der Bisektion* und dem *Newton-Raphson Algorithmus*. Beide Methoden werden in dem Anwendungsbeispiel:

C++ Anweisungen: Auswahlanweisungen mit if und switch

Die Programmiersprache C++ bietet einen konventionellen und flexiblen Satz von Anweisungen. Im Prinzip ist jede Programmzeile, die mit einem Semikolon endet eine Anweisung. In diesem Unterpunkt werden wir die sogennanten *Auswahlanweisungen* behandeln wobei wir in der vorigen Vorlesung die *Schleifenanweisungen* vorstellt hatten (siehe C++ Anweisungen: Die for-, while- und do-Schleifen). *Auswahlanweisungen* werden auch als Verzweigung , Tests oder *bedingte Anweisungen* bezeichnet und sind immer dann anzuwenden, wenn das Programm bei einem gewissen Ereignis bzw. unter einer gewissen Bedingung etwas bestimmtes tun soll. Es teilt somit das Programm in unterschiedliche Anweisungspfade auf. *Auswahlanweisungen* können in Form einer **if**-, (**if-else**)- oder **switch**-Anweisung ausgedrückt werden.

Die if-Anweisung

Eine **if**-Anweisung hat die folgende Struktur:

if ('Anweisungsbedingung') { 'Block von Anweisungen' }

Die 'Anweisungsbedingung' spezifiziert, in welchem Fall der 'Block von Anweisungen' ausgeführt werden soll. Ist diese Bedingung nicht erfüllt, so überspringt das Programm einfach die **if**-Anweisung.

If_0.cpp

```cpp
#include <iostream>                                          // Ein- und Ausgabebibliothek
using namespace std;                                         // Benutze den Namensraum std

int main(){                                                  // Hauptfunktion
    int zahl;                                                // Deklaration der Integer Variable 'zahl'

    cout << "Geben Sie bitte eine Fibonacci-Zahl im Bereich zwischen 100 und 200 ein: "; // Ausgabe eines Textes
    cin  >> zahl;                                            // Einlesen der Zahl mittels der Tastatur

    if( zahl == 144 ){                                       // if-Anweisung Anfang
        cout << "Richtig!" << endl;                          // Ausgabe falls if-Bedingung erfüllt ist
    }                                                        // if-Anweisung Ende
```

Das nebenstehende Programm gibt ein einfaches Beispiel für die Verwendung einer **if**-Anweisung. Das Programm fordert zunächst den Benutzer auf eine Zahl der Fibonacci-Folge im Bereich zwischen 100 und 200 einzugeben. Da es in diesem Zahlenbereich nur eine Fibonacci-Zahl gibt (144) ist es eindeutig, ob die vom Benutzer eingegebene Zahl richtig ist. Die **if**-Anweisung wird nun dafür benutzt zu entscheiden, ob die eingegebene Zahl richtig ist und dies wird mittels der 'Anweisungsbedingung' (zahl == 144) überprüft. Falls die 'Anweisungsbedingung' erfüllt ist, wird der 'Block von Anweisungen', in

Die if-Anweisung

Die 'Anweisungsbedingung' spezifiziert, in welchem Fall der 'Block von Anweisungen' ausgeführt werden soll. Ist diese Bedingung nicht erfüllt, so überspringt das Programm einfach die **if**-Anweisung.

Eine **if**-Anweisung hat die folgende Struktur:

if ('Anweisungsbedingung') { 'Block von Anweisungen' }

Das untere Programm gibt ein einfaches Beispiel für die Verwendung einer **if**-Anweisung. Das Programm fordert zunächst den Benutzer auf eine Zahl der Fibonacci-Folge im Bereich zwischen 100 und 200 einzugeben. Da es in diesem Zahlenbereich nur eine Fibonacci-Zahl gibt (144) ist es eindeutig, ob die vom Benutzer eingegebene Zahl richtig ist.

If_0.cpp

```cpp
#include <iostream>                                          // Ein- und Ausgabebibliothek
using namespace std;                                         // Benutze den Namensraum std

int main(){                                                  // Hauptfunktion
    int zahl;                                                // Deklaration der Integer Variable 'zahl'

    cout << "Geben Sie bitte eine Fibonacci-Zahl im Bereich zwischen 100 und 200 ein: "; // Ausgabe eines Textes
    cin  >> zahl;                                            // Einlesen der Zahl mittels der Tastatur

    if( zahl == 144 ){                                       // if-Anweisung Anfang
        cout << "Richtig!" << endl;                          // Ausgabe falls if-Bedingung erfüllt ist
    }                                                        // if-Anweisung Ende

    cout << "Vielen Dank für Ihre Eingabe."  << endl;        // Ausgabe eines Textes
}
```

Beispiel: Eingeschränkte Doppelsumme

Ein einfaches mathematisches Beispiel der Verwendung einer **if**-Anweisung ist z.B. die neben abgebildete Doppelsumme. In der zweiten Summe, besteht der Zusatz, dass der Summenindex i niemals den Wert des Summenindex k annehmen darf.

$$\sum_{k=1}^{4} \sum_{\substack{i=1 \\ i \neq k}}^{20} \frac{i^k}{(k-i)}$$

Doppelsumme_If_0.cpp

```cpp
#include <iostream>                                   // Ein- und Ausgabebibliothek
#include <cmath>                                      // Bibliothek für mathematisches (e-Funktion, Betrag, ...)
using namespace std;                                  // Benutze den Namensraum std

int main(){                                           // Hauptfunktion
    int k, i;                                         // Deklaration der Summenindexe k und i
    double Ergebnis = 0;                              // Deklaration und Initialisierung der Ergebnis-Variable

    for(k=1; k<=4; ++k){                              // Schleifen Anfang der 1.Summe
        for(i=1; i<=20; ++i){                          // Schleifen Anfang der 2.Summe
            if( i != k ){                             // if-Anweisung Anfang
                Ergebnis = Ergebnis + pow(i,k)/(k - i);   //Innerer, math. Teil der Doppelsumme
                }                                     // if-Anweisung Ende
        }                                             // Schleifen Ende der 2.Summe
    }                                                 // Schleifen Ende der 1.Summe

    cout << "Das Ergebnis der Doppelsumme beträgt: " << Ergebnis << endl; // Ausgabe des Ergebnisses
}
```

Das links abgebildete C++ Programm zeigt den Quelltext des Programmes zur Berechnung der angegebenen Doppelsumme. Der Anweisungsblock der 2. **for**-Schleife wird nur ausgeführt, falls der Wert des Index i ungleich dem Index k ist (**if**(i != k){ ... }). Das Ergebnis ergibt sich zu "Ergebnis = -64344".

(**if-else**)-Anweisung

Eine (**if-else**)-Anweisung hat die folgende Struktur:

if ('Anweisungsbedingung') { '1. Block von Anweisungen' } **else** { '2. Block von Anweisungen' }

Die 'Anweisungsbedingung' spezifiziert, in welchem Fall der '1. Block von Anweisungen' ausgeführt werden soll. Ist diese Bedingung nicht erfüllt, so führt das Programm den '2. Block von Anweisungen' aus.

If_Else_0.cpp

```cpp
#include <iostream>                                          // Ein- und Ausgabebibliothek
using namespace std;                                         // Benutze den Namensraum std

int main(){                                                  // Hauptfunktion
    int zahl;                                                // Deklaration der Integer Variable 'zahl'

    cout << "Geben Sie bitte eine Fibonacci-Zahl im Bereich zwischen 100 und 500 ein: "; // Ausgabe eines Textes
    cin  >> zahl;                                            // Einlesen der Zahl mittels der Tastatur

    if( zahl == 144 || zahl == 233 || zahl == 377 ){         // if-Anweisung Anfang mit logischem 'oder'
        cout << "Richtig!" << endl;                          // Ausgabe falls if-Bedingung erfüllt ist
    } else{                                                  // if-Anweisung Ende, else-Anweisung Anfang
        cout << "Leider Falsch :-(" << endl;                 // Ausgabe falls if-Bedingung nicht erfüllt ist
    }                                                        // else-Anweisung Ende

    cout << "Vielen Dank für Ihre Eingabe."  << endl;        // Ausgabe eines Textes
}
```

Die **switch**-Anweisung

Formale Struktur einer **switch**-Anweisung

```
switch ('Ganzzahlige Variable') {
   case Ganze Zahl Nr.1 :
      'Block von Anweisungen' ;
      break ;
   case Ganze Zahl Nr.2 :
      'Block von Anweisungen' ;
      break ;
    ....
   default :
      'Block von Anweisungen' ;
      break ;
}
```

Eine **switch**-Anweisung wählt unter einem Satz von Alternativen (**case**-Marken) aus. Der Ausdruck welcher direkt hinter dem **switch**-Befehl in runden Klammern steht, ist die 'Auswahl-Variable' der **switch**-Anweisung und der aktuelle Wert dieser Variable spezifiziert welche der **case**-Marken ausgewählt wird.

Switch_0.cpp

C++ Beispielprogramm

```
#include <iostream>                                                    // Ein- und Ausgabebibliothek
#include <cmath>                                                       // Bibliothek für mathematisches (e-Funktion, Betrag, ...)
using namespace std;                                                   // Benutze den Namensraum std

int main(){                                                            // Hauptfunktion
    double zahl;                                                       // Deklaration der Double Variable 'zahl'
    int auswahl;                                                       // Deklaration der Integer Variable 'auswahl' für switch

    cout << "Geben Sie bitte eine Gleitkommazahl ein: ";               // Ausgabe eines Textes
    cin  >> zahl;                                                      // Einlesen der Zahl mittels der Tastatur
    cout << "Möchten Sie die Zahl ..." << endl; ;                      // Ausgabe eines Textes
    cout << "... mal 25 nehmen, dann geben Sie 1 ein." << endl;                 // Ausgabe eines Textes
    cout << "... die Zahl hoch 5 nehmen, dann geben Sie 2 ein." << endl;        // Ausgabe eines Textes
    cout << "... den Sinus dieser Zahl berechnen, dann geben Sie 3 ein." << endl; // Ausgabe eines Textes
    cin  >> auswahl;                                                           // Einlesen der Variable 'auswahl'

    switch( auswahl ){                                                         // switch-Anweisung Anfang
        case 1:                                                                // switch-Fall Nr.1
            cout << "Der berechnete Wert beträgt: " << 25*zahl << endl;        // Anweisung Nr.1
            break;                                                             // switch-Fall Nr.1 Ende
        case 2:                                                                // switch-Fall Nr.2
            cout << "Der berechnete Wert beträgt: " << pow(zahl,5) << endl;    // Anweisung Nr.2
            break;                                                             // switch-Fall Nr.2 Ende
        case 3:                                                                // switch-Fall Nr.3
            cout << "Der berechnete Wert beträgt: " << sin(zahl) << endl;      // Anweisung Nr.3
            break;                                                             // switch-Fall Nr.3 Ende
        default:                                                               // switch-default
            cout << "Leider haben Sie die falsche Auswahl getroffen :-(" << endl; // Anweisung default
            break;                                                             // switch-default Ende
    }                                                                          // switch-Anweisung Ende

    cout << "Vielen Dank für Ihre Eingabe."  << endl;                          // Ausgabe eines Textes
}
```

Die 'Auswahl-Variable' sollte eine ganzzahlige Integer-Variable, oder ein char-Zeichen sein. Die einzelnen Werte der **case**-Marken legen die jeweilige Alternative und den auszuführenden 'Block von Anweisungen' fest. Am Ende einer jeden **case**-Marke steht ein **break**-Befehl, der ein Verlassen der **switch**-Auswahl bewirkt. Am Ende einer **switch**-Anweisung sollte zusätzlich eine **default**-Marke stehen, sodass nicht explizit betrachtete, potentiell mögliche Werte der 'Auswahl-Variable' auch behandelt werden können.

C++ Funktionen

Die Definition einer C++ Funktion besteht aus einer *Deklaration* und einem *Anweisungsblock*.

```
'Rückgabe Typ' Funktionsname ('Argumentenliste') { 'Block von Anweisungen' }
```

Die allgemeine Definition einer C++ Funktion ist der formalen Definition einer mathematischen Funktion nicht unähnlich und man könnte sie auch wie folgt definieren: "Eine C++ Funktion ist eine Abbildung von dem Datenraum der Argumentenliste in den Datenraum des Rückgabetyps. Die dabei benutzte Abbildungsvorschrift findet sich in ihrem Anweisungsblock.

- **Der Name der Funktion**:
  Hier sollte der Programmierer einen Namen wählen, der die im *Anweisungsblock* definierten Anweisungen präzise und kurz in einem Wort zusammen fasst.
- **Rückgabe Typ**:
  Welchen Datentyp gibt die Funktion an das Hauptprogramm zurück? Der "Rückgabe Typ" steht vor dem Funktionsnamen.
- **Argumentenliste**:
  Die "Argumentenliste" steht direkt hinter dem Funktionsnamen, im Aufrufoperator "(...)". Sie spezifiziert die Datentypen der Variablen, die die Funktion zur Berechnung ihrer Aufgabe benötigt.

Für die *Deklaration* einer C++ Funktion sind gewisse Angaben erforderlich (siehe nebenstehende Abbildung) und gewisse zusätzliche Spezifikationen möglich (z.B. '**inline**' oder '**virtual**'). Um eine Funktion schließlich zu definieren, muss man zusätzlich noch die zu erledigenden Anweisungen in einem Anweisungsblock zusammenfassen.

Mathematische Funktionen in C++

Definition von mathematischen Funktionen in C++

In C++ hat das Wort Funktion eine allgemeinere Bedeutung als im Bereich der Mathematik und die in der Mathematik und Physik definierte Funktionen stellen eine echte Teilmenge der C++ Funktionen dar. In diesem Teilkapitel möchten wir weiter in den Themenbereich der C++ Funktionen einführen und uns hauptsächlich mit der Definition von mathematischen Funktionen in C++ befassen. Betrachten wir uns z.B. die Funktion $f(x) = x^2$. In der Mathematik definiert man diese Funktion als eine Abbildung von der Menge der reellen Zahlen $\mathbb{R}$ in die Menge der positiv reellen Zahlen $\mathbb{R}^+$ und man schreibt formal:

$$f: \underbrace{\mathbb{R}}_{\text{Definitionsmenge}} \rightarrow \underbrace{\mathbb{R}^+}_{\text{Bildmenge}} \quad \text{mit:} \quad f(x) = x^2 \ , \ x \in \mathbb{R}$$

Wir möchten nun diese mathematische Funktion in C++ formulieren und z.B. ihre Funktionswerte in einem Teilintervall ihrer Definitionsmenge $[a, b] \in \mathbb{R}$ ausgeben lassen. Die einfachste C++ Version der oben definierten mathematischen Funktion lautet:

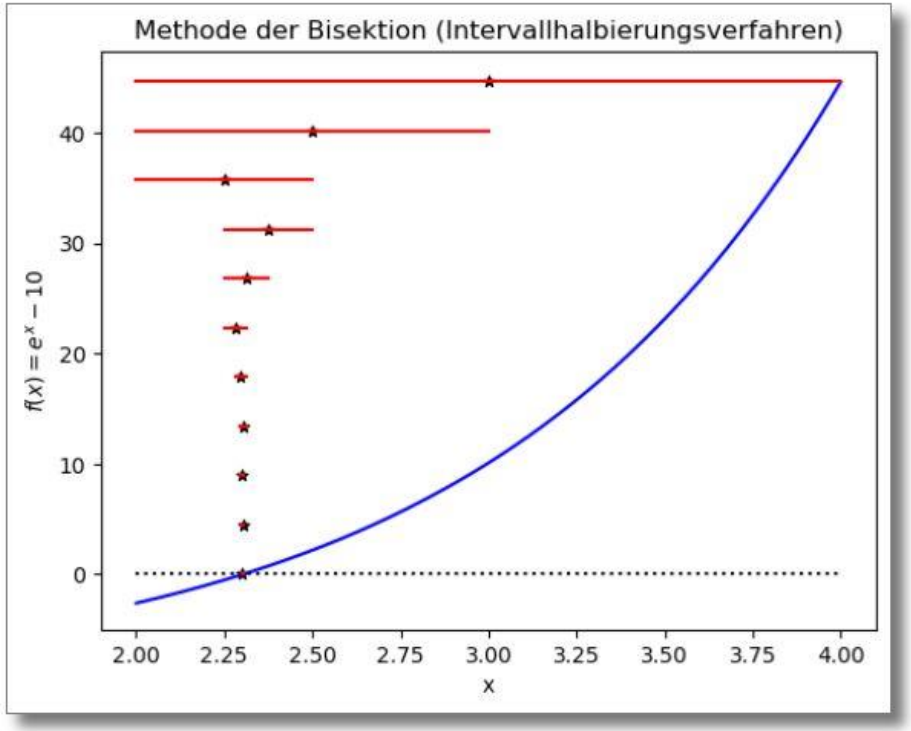
```
double f (double x) { return x*x; }
```

Nullstellensuche einer Funktion

Anwendungsbeispiel: Nullstellensuche einer Funktion

In diesem Unterpunkt möchten wir ein grundlegendes Problem der numerischen Mathematik behandeln, die Nullstellensuche einer Funktion. Wir betrachten im speziellen die Funktion $f(x) = e^x - 10$ und wollen berechnen, bei welchem x-Wert die Funktion Null wird ($f(x) = 0$). Bei der angegebenen Funktion ist das nicht schwierig und man erhält mittels einfacher analytischer Umformungen das Ergebnis $p = \ln(10) \approx 2.302585...$. Eine solche analytische Umformung ist jedoch nicht immer möglich und man kann dann eine Nullstelle auf numerischem Weg mittels unterschiedlicher Methoden berechnen. In diesem Unterpunkt betrachten wir zunächst die *Methode der Bisektion* (das *Intervallhalbierungsverfahren*) und danach die *Newton-Raphson Methode* zur numerischen Ermittlung der Nullstelle.

Der Algorithmus der Bisektion (Intervallhalbierungsverfahren)



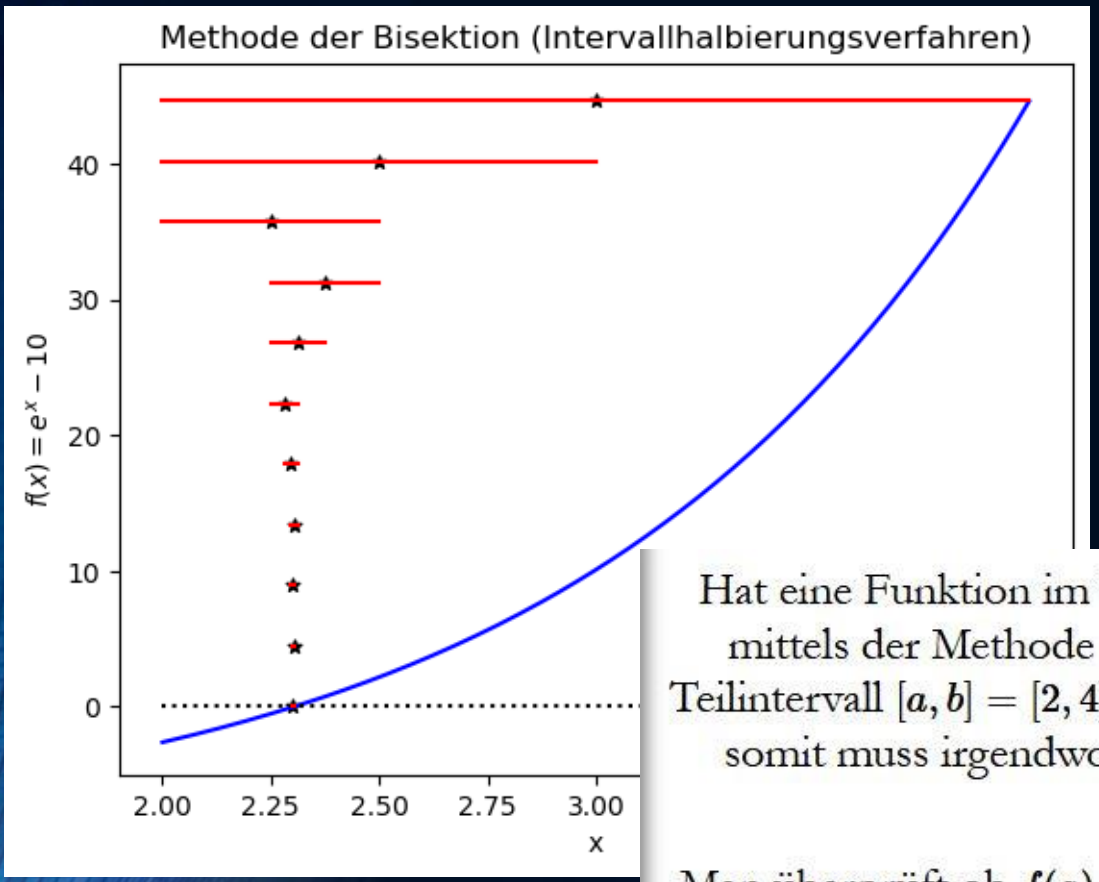
Hat eine Funktion im Teilintervall $[a, b] \in \mathbb{R}$ eine Nullstelle, so kann man diese Nullstelle numerisch mittels der Methode der Bisektion ermitteln. Betrachten wir z.B. die Funktion $f(x) = e^x - 10$ im Teilintervall $[a, b] = [2, 4]$. Es gilt $f(2) = e^2 - 10 \approx -2.61094 < 0$ und $f(4) = e^4 - 10 \approx 44.59815 > 0$ und somit muss irgendwo zwischen 2 und 4 eine Nullstelle existent sein. Man geht nun wie folgt vor:

Man überprüft ob $f(a) \cdot f(a + \frac{(b-a)}{2})$ größer oder kleiner Null ist. Ist der Wert größer Null, so ist man sicher, dass sich die Nullstelle nicht im Teilintervall $[a, a + \frac{(b-a)}{2}]$ befindet und kann das neue Teilintervall auf $[a + \frac{(b-a)}{2}, b]$ setzen. Ist der Wert hingegen kleiner Null, so befindet sich die Nullstelle im Teilintervall $[a, a + \frac{(b-a)}{2}]$, welches man dann als neues Teilintervall verwendet. Der spezielle Fall $f(a) \cdot f(a + \frac{(b-a)}{2}) = 0$ bedeutet, dass die Nullstelle gerade bei $p_0 = p = a + \frac{(b-a)}{2}$ ist und man kann den Algorithmus des Intervallhalbierungsverfahrens abrechen.

Die nebenstehende Abbildung veranschaulicht die Methode der Bisektion an unserem Beispiel. Die blaue Kurve zeigt die Funktion $f(x) = e^x - 10$ im Teilintervall $x \in [2, 4]$. Die roten, horizontalen Linien kennzeichnen die jeweiligen Teilintervalle, wobei die oberste Linie das ursprüngliche Teilintervall $[2, 4]$ kennzeichnet. Die Mitte des Intervalls ist mit einem schwarzen Stern markiert und dieser Wert stellt auch gleichzeitig den jeweiligen approximierten Wert p_i der Nullstelle dar. Beim Anfangsintervall $[a, b] = [2, 4]$ erhält man $p_0 = a + \frac{(b-a)}{2} = 2 + \frac{(4-2)}{2} = 3$ und da $f(a) \cdot f(a + \frac{(b-a)}{2}) = f(2) \cdot f(3) < 0$ ist, befindet sich die wirkliche Nullstelle im Teilintervall $[2, 3]$, welches wir bei dann bei der nächsten Iteration als neues Teilintervall $[a, b]$ definieren. Wie wir in der nebenstehenden Abbildung sehen, tastet sich das Intervallhalbierungsverfahren bei fortlaufender Iteration immer näher an den wirklichen Wert der Nullstelle heran.

Das entsprechende C++ Programm des betrachteten Beispiels der Methode der Bisektion (siehe Nullstelle_Bisektion_0.cpp) ist in dem folgenden Frame abgebildet:

Nullstellensuche einer Funktion



Der Algorithmus der Bisektion

Intervallhalbierungsverfahren

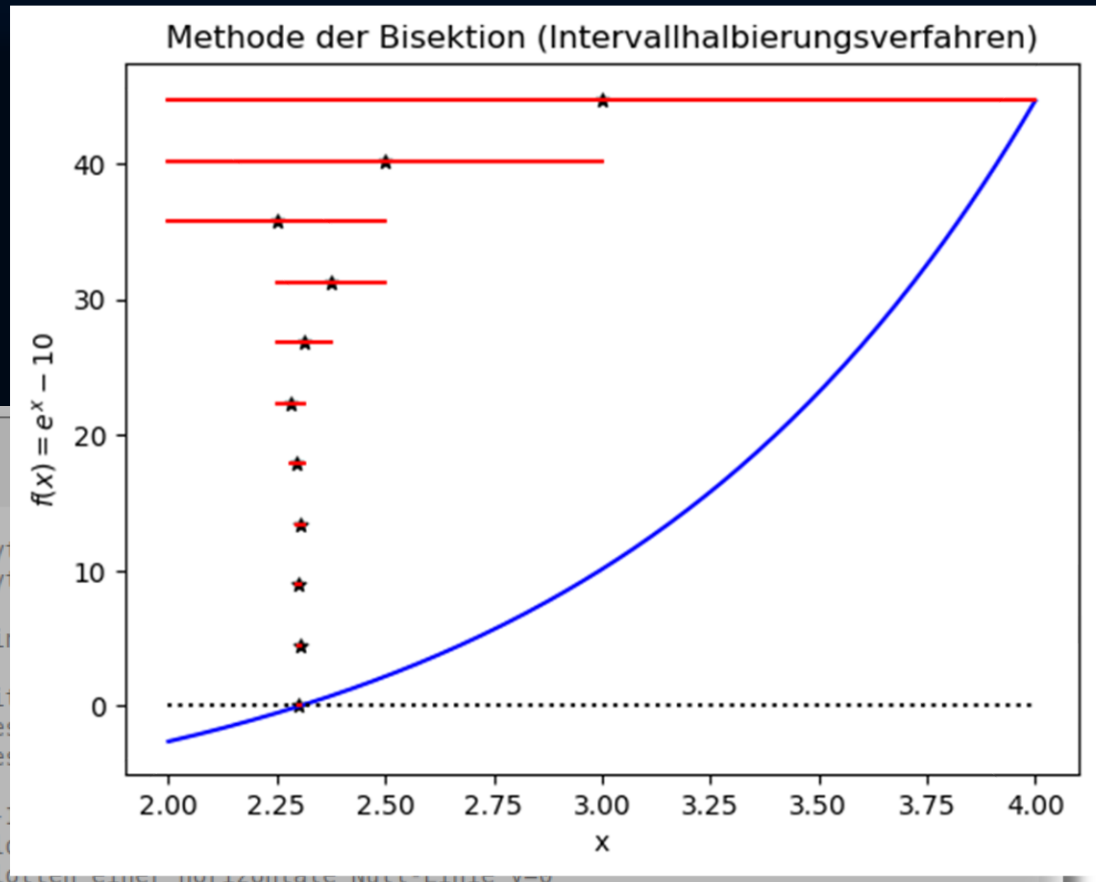
In diesem Anwendungsbeispiel möchten wir ein grundlegendes Problem der numerischen Mathematik behandeln, die Nullstellensuche einer Funktion. Wir betrachten eine Funktion f(x) und wollen berechnen, bei welchem x-Wert die Funktion Null wird (f(x)=o).

Hat eine Funktion im Teilintervall $[a, b] \in \mathbb{R}$ eine Nullstelle, so kann man diese Nullstelle numerisch mittels der Methode der Bisektion ermitteln. Betrachten wir z.B. die Funktion $f(x) = e^x - 10$ im Teilintervall $[a, b] = [2, 4]$. Es gilt $f(2) = e^2 - 10 \approx -2.61094 < 0$ und $f(4) = e^4 - 10 \approx 44.59815 > 0$ und somit muss irgendwo zwischen 2 und 4 eine Nullstelle existent sein. Man geht nun wie folgt vor:

Man überprüft ob $f(a) \cdot f(a + \frac{(b-a)}{2})$ größer oder kleiner Null ist. Ist der Wert größer Null, so ist man sicher, dass sich die Nullstelle nicht im Teilintervall $[a, a + \frac{(b-a)}{2}]$ befindet und kann das neue Teilintervall auf $[a + \frac{(b-a)}{2}, b]$ setzen. Ist der Wert hingegen kleiner Null, so befindet sich die Nullstelle im Teilintervall $[a, a + \frac{(b-a)}{2}]$, welches man dann als neues Teilintervall verwendet. Der spezielle Fall $f(a) \cdot f(a + \frac{(b-a)}{2}) = 0$ bedeutet, dass die Nullstelle gerade bei $p_0 = p = a + \frac{(b-a)}{2}$ ist und man kann den Algorithmus des Intervallhalbierungsverfahrens abrechen.

Visualisierung mit einem Python Skript

Um die jeweiligen Teilintervalle als diskrete horizontale Linien darstellen zu können, definieren wir die Liste 'y_interval', welche gerade 'N+1' äquidistante Werte im Bereich [0,f(4)] beinhaltet. Die einzelnen Resultate der Bisektions-Iteration werden schließlich mittels einer Python **for**-Schleife realisiert und die einzelnen Teilintervalle als horizontale rote Linien und der approximierte Nullstellenwert p_i als Stern mittels der Funktion 'plt.scatter(...)' geplottet.



PythonPlot_Bisektion_0.py

```python
# Python Programm zum Plotten der berechneten Daten von (Nullstelle_Bisektion_0.cpp)
import matplotlib.pyplot as plt
import numpy as np

data = np.genfromtxt("./Nullstelle_Bisektion_0.dat")

plt.title(r'Methode der Bisektion (Intervallhalbierungsverfahren)')
plt.ylabel(r'$f(x)=e^x - 10$')
plt.xlabel('x')

x_interval=np.linspace(data[0,4], data[0,5], 200)
plt.plot(x_interval,np.exp(x_interval)-10, color="blue")
plt.plot([data[0,4],data[0,5]],[0,0], color="black", linestyle=":")

y_interval=np.linspace(np.exp(x_interval[-1])-10, 0, int(data[-1,0])+1)

for i in data[:,0]:
    index=int(i)
    plt.plot([data[index,4],data[index,5]],[y_interval[index],y_interval[index]], color="red")
    plt.scatter(data[index,1],y_interval[index], marker='*', color="black", s=25)

plt.savefig("Nullstelle_Bisektion_0.png", bbox_inches="tight")
plt.show()
```

Die Newton-Raphson Methode der Nullstellensuche

Die *Newton-Raphson Methode* der Nullstellenermittlung ist ein sehr effektives Verfahren zur Ermittlung einer Nullstelle. Im Gegensatz zur Methode der Bisektion hat dieses Verfahren jedoch den Nachteil, dass man neben der Funktion $f(x)$ zusätzlich auch noch die erste Ableitung der Funktion ($f'(x) = \frac{df(x)}{dx}$) im Teilintervall $[a, b]$ benötigt. Die *Newton-Raphson Methode* basiert auf dem Prinzip der Taylorreihenentwicklung der Funktion, wobei man nur die ersten beiden Terme der Reihenentwicklung berücksichtigt und alle nicht-linearen Terme vernachlässigt. Man nimmt hierbei an, dass die Funktion im Teilintervall $[a, b]$ eine Nullstelle bei $x = p \in [a, b]$ hat ($f(p) = 0$) und rät im 0-ten Schritt des Algorithmus einen x-Wert, der nahe dieser Nullstelle liegen soll. Wir bezeichnen diesen geratenen Wert im Folgenden mit p_0 und entwickeln unsere Funktion um diesen Wert in eine Taylorreihe

$$f(x) = f(p_0) + (x - p_0) \cdot f'(p_0) + \frac{(x - p_0)^2}{2} \cdot f''(p_0) + \dots \quad .$$

Betrachtetet man nun die taylorentwickelte Funktion bei $x = p$ und nimmt an, dass der Wert von p_0 nahe der wirklichen Nullstelle ist, so erhält man:

$$0 = f(p) = f(p_0) + (p - p_0) \cdot f'(p_0) \underbrace{+ \frac{(p - p_0)^2}{2} \cdot f''(p_0) + \dots}_{\approx 0} \approx f(p_0) + (p - p_0) \cdot f'(p_0)$$

$$\Longrightarrow p \approx p_0 - \frac{f(p_0)}{f'(p_0)}$$

In der *Newton-Raphson Methode* benutzt man nun diesen neu approximierten Wert der Nullstelle und nimmt diesen für eine neue Iteration. Der Algorithmus benutzt somit die folgende Gleichung um die n-te Approximation der Nullstelle p_n mithilfe der (n-1)-ten Approximation zu berechnen:

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \quad \text{mit: } n \geq 1$$

Übungsblatt Nr. 4

Aufgabe 1 *(7.5 Punkte)*

Im Intervall $x \in [0.25, 2.5]$ hat die Funktion $g(x) = x^3$ mit der Funktion $h(x) = \ln(x) + 5$ einen Schnittpunkt. Berechnen Sie diesen Schnittpunkt auf nummerischem Wege mittels der Methode der Bisektion und stellen sie die ersten 11 Schritte des Intervallhalbierungsverfahrens grafisch mittels eines Python Skriptes dar. Bemerkung: Der Literaturwert des x-Wertes des Schnittpunktes beträgt $x = 1.7729093296693715...$

Aufgabe 2 *(7.5 Punkte)*

Erstellen Sie ein Plot-Programm für die folgenden Funktionen

$$f_1(x) = e^x - 10$$

$$f_2(x) = x^2 - 10\,x + 8$$

$$f_3(x) = x^3 - 8\,x^2 + 2\,x + 3$$

$$f_4(x) = 10\,e^{x/5} \cdot \sin(3\,x)$$

Erstellen Sie dafür ein C++ Programm, welches zunächst mittels einer Benutzereingabe (1,2,3 oder 4) abfragt, welche Funktion geplottet werden soll und dann die (x-y)-Wertetabelle (200 Punkte) im Intervall $x \in [0, 10]$ im Terminal ausgibt. Benutzen Sie bitte bei der Plot-Auswahl eine **switch**-Anweisung, wobei bei einer falschen Eingabe einfach die Funktion $f(x) = 0$ geplottet werden soll. Die Datentabelle leiten Sie dann in eine Textdatei um und lesen diese Daten danach in ein Python Skript ein, mit welchem Sie dann die Datenpunkte visualisieren und als Bild ausgeben lassen. Bemerkung: Gerne können Sie auch die Ausführung des C++ und Python Programms zusammengefügt in einem Shell Skript (z.B. A4_2.sh) ausführen. (Man hätte natürlich das gesamte Plot-Programm auch ausschließlich mittels eines Python Skriptes schreiben können. Bitte trennen Sie jedoch die Berechnung der (x-y)-Wertetabelle und die Visualisierung dieser Daten wie oben angegeben).

Aufgabe 3 *(5 Punkte)*

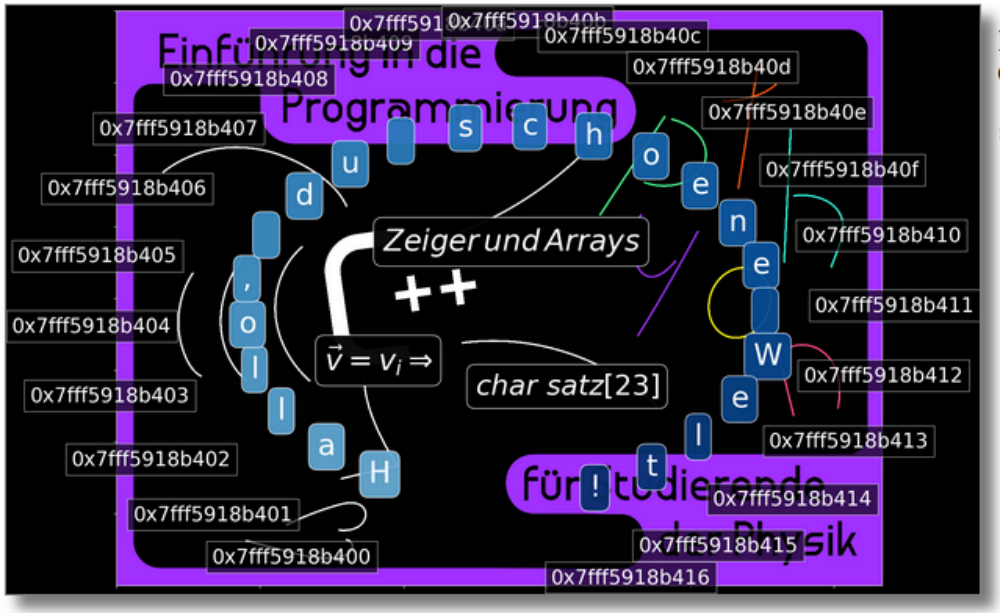
Berechnen Sie die folgenden Ausdrücke mittels eines C++ Programms und lassen Sie sich die berechneten Werte (falls diese Gleitkommazahlen sind) auf 15 Stellen genau ausgeben.

$$\prod_{k=1}^{11} k^2 \quad , \quad \sum_{k=0}^{20} \sum_{\substack{i=0 \\ i \neq k}}^{10} \frac{k + i^3}{(k-i)^2} \quad , \quad \text{Bestimmen Sie N} \quad : \quad \sum_{k=0}^{N} \sum_{\substack{i=0 \\ i \neq k}}^{250} \left(k + i^2\right) = 672013120$$

Vorlesung 5

In dieser Vorlesung werden wir zunächst auf das *C++ Array- und Zeiger Konzept* eingehen und sowohl die eindimensionalen Zahlenarrays, als auch die eindimensionalen Zeichenarrays an Beispielen verdeutlichen. Am Ende des ersten Teils gehen wir auf die Problematik der Übergabe eines Arrays an eine C++ Funktion ein. Als ein Anwendungsbeispiel der eindimensionalen Arrays betrachten wir dann die *Lagrange Polynome Methode*, bei der ein Polynom mittels einer gegebenen Menge Stützstellen berechnet wird. Es wird ein C++ Programm vorgestellt, welches mittels des Lagrange Polynom Algorithmus die xy-Wertetabelle des Polynoms erzeugt und diese Daten werden dann mittels eines Python Jupyter Notebooks visualisiert. Zusätzlich wird in dem Notebook auch die analytische Form der Polynome unter Verwendung der Computer-Algebra-Bibliothek "sympy" berechnet und dieses Notebook zeigt somit ein weiteres wichtiges Anwendungsfeld von Python Jupyter Notebooks auf, das dem Verständnis und der Herleitung der zu berechnenden Gleichungen dient.

C++ Arrays, Zeiger und Referenzen



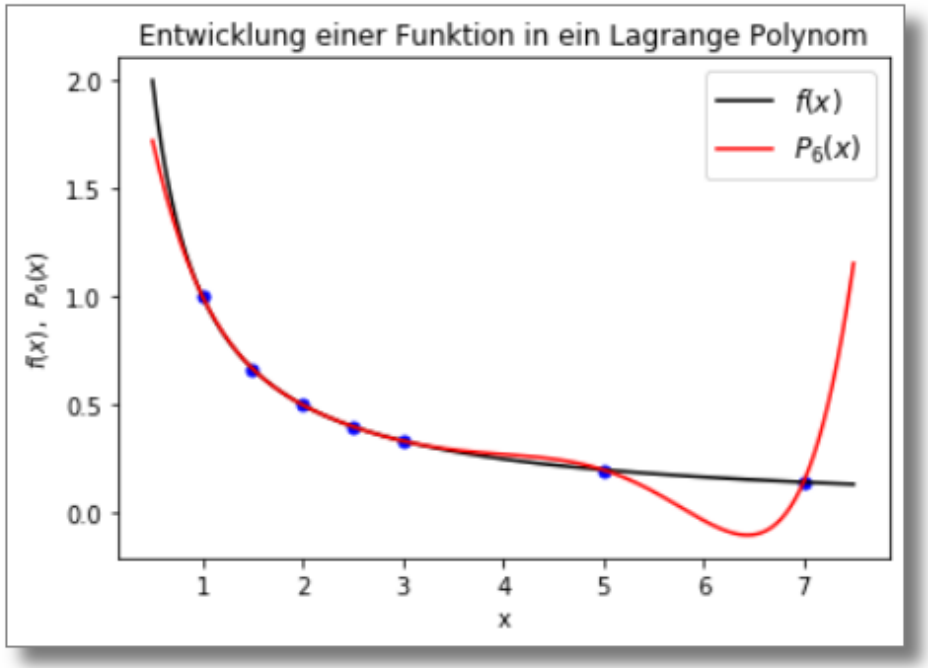
Alle die bis zu diesem Abschnitt besprochenen Datentypen, verbinden den Namen des deklarierten Typs mit einem einzelnen Wert, d.h. mit einer skalar wertigen Größe. Um die in der Physik und Mathematik gebräuchlichen Größen wie Vektoren und Matrizen in einem Computerprogramm adäquat abbilden zu können, gibt es die sogenannten Datenarrays. Ein C++ Array ist eine geordnete Menge von Elementen des gleichen Typs T. In diesem Unterpunkt werden die wesentlichen Grundlagen zu eindimensionalen *C++ Arrays* vorgestellt. Da die interne Array-Implementierung der Programmiersprache C++ eng mit dem *C++ Zeiger Konzept* verknüpft ist, werden wir zunächst die Begriffe *Zeiger*, *Adresse* und *Referenz* an mehreren Beispielen vorstellen, um danach die eindimensionalen Arrays am Beispiel des Zeichen-Arrays des "HelloWorld" - Programmes zu verdeutlichen (siehe nebenstehende Abbildung). Das C++-Zeigerkonzept ist der grundlegende integrierte Sprachmechanismus, um den Zugriff auf den Hauptspeicher des Computers direkt zu ermöglichen. Am Ende dieses Unterpunktes zeigen wir, wie man Arrays an Funktionen übergibt. Möchte man ein Array an eine Funktion als Argument übergeben, sollte man dies nicht über die einzelnen Werte des Arrays machen, da man dann die Array Einträge nicht verändern kann. Stattdessen

Vorlesung 5

Die zugrundeliegenden Gleichungen der Physik sind oft in Form von vektoriellen und matrixwertigen Gleichungen formuliert. Vektoren und Matrizen sind mathematische Konstrukte, die eine Vielzahl von Elementen (Zahlen) in einer gebundenen Form zusammenfassen. Wie deklariert man ein solches zusammengesetztes Konstrukt in einem C++ Programm? An diesem Punkt der Vorlesung wollen wir noch nicht auf das C++ Container Konzept eingehen, welches unter anderem den wichtigen Container vector der Standardbibliothek bereitstellt, sondern wir betrachten uns das schon in alten C-Programmen oft verwendete, integrierte C++ Array Konzept. Ein C++ Array (nicht zu verwechseln mit dem Container array der Standardbibliothek) stellt ein zusammengesetztes Konstrukt von Elementen eines Datentyps dar. Ein Vektor $\vec{v} = (1, 3, 7)$ bestehend aus drei ganzzahligen Elementen kann z.B. mittels des int-Arrays "int v[3] = {1,3,4};" definiert werden. Der Zugriff auf die einzelnen Werte des Arrays erfolgt hierbei durch Angabe seines Indexes, wobei die Elemente von 0 bis 2 indiziert werden (z.B. $v_1 = 3 \rightarrow$ v[1]). Wie wird nun ein solches zusammenhängendes Konstrukt von Zahlen im Hauptspeicher geordnet gespeichert? Da die interne Array-Implementierung der Programmiersprache C++ eng mit dem *C++ Zeiger Konzept* verknüpft ist, werden wir in dieser Vorlesung zunächst die Begriffe *Zeiger*, *Adresse* und *Referenz* an mehreren Beispielen vorstellen und die eindimensionalen Arrays am Beispiel des Zeichen-Arrays des "HelloWorld" - Programmes zu verdeutlichen (siehe obere Abbildung im linken Frame). Das C++-Zeigerkonzept ist der grundlegende integrierte Sprachmechanismus, um den Zugriff auf den Hauptspeicher des Computers direkt zu ermöglichen. Zusätzlich wird darauf eingegangen, wie man Arrays an C++ Funktionen übergibt.

Das erlernte *C++ Array Konzept* wird dann an dem Beispiel der *Methode der Lagrange Polynome* verdeutlicht. Die Lagrange Polynome Methode ist ein wichtiges Teilgebiet der numerischen Mathematik und es befasst sich mit der Problematik ein Polynom vom Grade n ($P_n(x)$) mittels einer gegebenen Menge von $(n+1)$ Stützstellenpunkten zu bestimmen. Das Verfahren wird bei einer Vielzahl von numerischen Algorithmen verwendet, bei denen eine vorgegebene Funktion $f(x)$ durch ein Polynom $P_n(x)$, mittels der Angabe der $(n+1)$ x-Werte der Stützstellenpunkten (

Anwendungsbeispiel: Interpolation und Polynomapproximation



Die Interpolation ist ein Teilgebiet der numerischen Mathematik und es befasst sich mit der Problematik, eine analytische Funktion mittels einer gegebenen Menge von Werten zu bestimmen. In diesem Unterpunkt werden wir die Interpolation einer Funktion $f(x)$ mittels der *Methode der Lagrange Polynome* vorstellen, bei der man mehrere Stützstellenpunkte $\vec{x} = (x_0, x_1, \ldots, x_n)$ für die Konstruktion des approximierenden Polynoms verwendet. Die nebenstehende Abbildung stellt z.B. das Lagrange Polynom $P_6(x)$ dar, welches die Funktion $f(x) = 1/x$ mittels der sieben Stützstellen $\vec{x} = (1, 1.5, 2, 2.5, 3, 5, 7)$ approximiert. Nachdem die Theorie der *Lagrange Polynome Methode* kurz vorgestellt wurde, wird ein C++ Programm entworfen, welches den Kern-Algorithmus des Verfahrens implementiert und dabei das Array-Konzept benutzt. Es wird unter anderem z.B. der Vektor $\vec{x}$ der x-Werte der Stützstellenpunkte als ein double-Array "double points[]" deklariert und direkt mit den Stützstellenwerten initialisiert. Die berechneten Lagrangepolynome werden danach mittels eines Jupyter Notebook visualisiert und mit der Funktion $f(x)$ verglichen. Zusätzlich wird in dem Notebook auch die analytische Form der Polynome unter Verwendung der Computer-Algebra-Bibliothek "sympy" berechnet (näheres siehe Anwendungsbeispiel: Interpolation und Polynomapproximation).

Weitere Links

- Folien der 5.Vorlesung
- Übungsblatt Nr.5

C++ Arrays, Zeiger und Referenzen

In diesem Unterpunkt werden die wesentlichen Grundlagen zu eindimensionalen *C++ Arrays* vorgestellt. Da die interne Array-Implementierung der Programmiersprache C++ eng mit dem *C++ Zeiger Konzept* verknüpft ist, werden wir zunächst die Begriffe *Zeiger*, *Adresse* und *Referenz* an mehreren Beispielen vorstellen, um danach die eindimensionalen Arrays am Beispiel des Zeichen-Arrays des "HelloWorld" - Programmes zu illustrieren. Der Bezeichner (Variablenname) des Arrays kann hierbei als Zeiger auf sein erstes Element verstanden werden. Bei der Deklaration eines Arrays muss sein Name, Datentyp und die Anzahl der Elemente spezifiziert werden (in einem *C++ Array* dürfen keine Elemente unterschiedlichen Typs auftreten). Möchte man ein Array an eine Funktion als Argument übergeben, sollte man dies nicht über die einzelnen Werte des Arrays machen, da man dann die Array-Einträge nicht verändern kann. Stattdessen übergibt man ein Array als Zeiger auf sein erstes Element mit einem zusätzlichen Vermerk zu seiner Dimension. Mehrdimensionale *C++ Arrays* sind im Prinzip als ein Array von Arrays dargestellt und beschreiben den Übergang zu eines Matrix-ähnlichen integrierten Datentyp. Diese werden jedoch erst in der nächsten Vorlesung behandelt.

Zeiger, Adressen und Referenzen

In diesem Teilkapitel werden wir die grundlegenden integrierten Sprachmechanismen kennenlernen, die für den Zugriff auf den Hauptspeicher des Computers vorgesehen sind. Mittels des Variablennamens können wir in einem C++ Programm auf den in der Variable abgelegten Wert zugreifen. Diese abstrakte Variablenebene eines C++ Quelltextes besitzt eine physikalische Identität im Hauptspeicher des Computers, wo der Wert der Variable in Form von Nullen und Einsen in mehreren Bits gespeichert ist (siehe Unterpunkt Datentypen und Variablen und Computerarithmetik mit Python). Dies bedeutet, dass es zu jeder deklarierten Variable eines bestimmten Datentyps eine Adresse im Hauptspeicher gibt und wir werden nun beschreiben, wie man eine solche Adresse/Referenz ermittelt und die dafür benötigten C++ Sprachkonstrukte des *Zeigers* und der *Referenz* vorstellen.

Betrachten wir z.B. eine Variable, die den Wert einer Gleitkommazahl in doppelter Maschinengenauigkeit speichern soll - eine sogenannte double-Variable mit dem Namen "zahl". Für den Wert dieser Variable wurde beim Deklarationsprozess ein Platz von 8 Bytes im Hauptspeicher reserviert. Die Adresse im Hauptspeicher, wo der Wert der Variable binär abgelegt ist, kann man mittels des

Zeiger, Adressen und Referenzen

Betrachten wir z.B. eine Variable, die den Wert einer Gleitkommazahl in doppelter Maschinengenauigkeit speichern soll - eine sogenannte double-Variable mit dem Namen "zahl". Für den Wert dieser Variable wurde beim Deklarationsprozess ein Platz von 8 Bytes im Hauptspeicher reserviert. Die Adresse im Hauptspeicher, wo der Wert der Variable binär abgelegt ist, kann man mittels des Referenzoperators (&zahl: ein der Variable vorgestelltes &) ermitteln. In der Sprache C++, sind speziell für den Zweck des Hauptspeicherzugriffs, zwei eigene Datentypen definiert, mittels deren man die Adresse der Variable eines bestimmten Datentyps T speichern kann. Der Datentyp "Zeiger auf T" wird mit einem nachgestellten Sternsymbol gekennzeichnet (T*) und eine Variable dieses neuen Datentyps kann die Adresse eines Typs T speichern. Man sollte eine Zeiger-Variable des Typs T* am besten sofort beim Deklarationsprozess mit einer Adresse initialisieren, da es sonst geschehen kann, dass der Zeiger auf ein nicht existentes Objekt im Hauptspeicher zeigt. Möchte man dennoch einen nicht-intitialisierten Zeiger verwenden, so wird es angeraten diesen mit dem Nullzeiger "nullptr" zu initialisieren (z.B. int* a = nullptr;).

Die unten abgebildete Box zeigt die Verwendung von Zeigern, Adressen und Referenzen am Beispiel des Datentyps double.

Zeiger, Adresse und Referenz am Beispiel des Datentyps T → double

Deklaration einer double-Variable und Initialisierung mit einer Gleitkommazahl:

```
double zahl = 2.47654673;
```

Definition des Zeigers auf die Adresse der double-Variable:

```
double* zeiger_zahl = &zahl;
```

Definition der Referenz der double-Variable

```
double& ref_zahl = zahl;
```

Dereferenzierung des Zeigers (vorgestelltes Sternzeichen) liefert den Wert der double-Variable:

```
double stern_zeiger_zahl = *zeiger_zahl;
```

Zeiger, Adressen und Referenzen

Ein weiterer neuer Datentyp ist die "Referenz eines Typs T" und diese wird bei der Deklaration des Typennamens mit einem nachgestellten &-Symbol gekennzeichnet (T&). Eine Referenz ist im Prinzip gleichbedeutend mit der Adresse des Objektes, wobei im Unterschied zum Zeigerkonstrukt eine automatische Umwandlung (Dereferenzierung) der Adresse in den Wert der Variable geschieht. Eine Referenz ist demnach eine Art von Zeiger, der bei jeder Verwendung im Programm dereferenziert wird. Bei einer Zeigervariable Z des Typs T erhält man den Wert mittels eines vorgestellten Sternsymbols "*" (*Z: Dereferenzieren eines Zeigers, Inhaltsoperator "*" angewandt auf den Zeiger Z), wobei bei einer Referenz das Dereferenzieren automatisch geschieht. Eine Referenz bezieht sich immer auf das Objekt, mit dem sie initialisiert wurde und es gibt keine Null-Referenzen im Gegensatz zum Null-Zeiger.

Die unten abgebildete Box zeigt die Verwendung von Zeigern, Adressen und Referenzen am Beispiel des Datentyps double.

Zeiger, Adresse und Referenz am Beispiel des Datentyps T → double

Deklaration einer double-Variable und Initialisierung mit einer Gleitkommazahl:

```
double zahl = 2.47654673;
```

Definition des Zeigers auf die Adresse der double-Variable:

```
double* zeiger_zahl = &zahl;
```

Definition der Referenz der double-Variable

```
double& ref_zahl = zahl;
```

Dereferenzierung des Zeigers (vorgestelltes Sternzeichen) liefert den Wert der double-Variable:

```
double stern_zeiger_zahl = *zeiger_zahl;
```

Zeiger, Adressen und Referenzen

am Beispiel von char, int und double

Das folgende C++ Programm illustriert die Verwendung der besprochenen Größen am Beispiel der Datentypen T → char, int und double

Zeiger_1.cpp

```cpp
#include <iostream>                                // Ein- und Ausgabebibliothek

int main(){                                        // Hauptfunktion
    char zeichen = 'H';                            // Definition einer Variable vom Datentyp char (zeichen) und Initialisierung auf den Buchstaben 'H'
    char* zeiger_zeichen = &zeichen;               // Definition des Zeigers auf die Adresse der char Variable zeichen (&zeichen)
    char& ref_zeichen = zeichen;                   // Definition der Referenz(Adresse) der char Variable zeichen
    char stern_zeiger_zeichen = *zeiger_zeichen; // char Variable zum Überprüfen des Dereferenzierungsoperators angewandt auf den Zeiger (*zeiger_zeichen)

    int ganze_zahl = 234767;                           // Definition einer Variable vom Datentyp int (ganze Zahl) und Initialisierung
    int* zeiger_ganze_zahl = &ganze_zahl;              // Definition des Zeigers auf die Adresse der int Variable ganze_zahl (&ganze_zahl)
    int& ref_ganze_zahl = ganze_zahl;                  // Definition der Referenz(Adresse) der int Variable ganze_zahl
    int stern_zeiger_ganze_zahl = *zeiger_ganze_zahl; // int Variable zum Überprüfen des Dereferenzierungsoperators angewandt auf den Zeiger (*zeiger_ganze_zahl)

    double zahl = 2.47654673;                 // Definition einer Variable vom Datentyp double (zahl) und Initialisierung
    double* zeiger_zahl = &zahl;              // Definition des Zeigers auf die Adresse der double Variable zahl (&zahl)
    double& ref_zahl = zahl;                  // Definition der Referenz(Adresse) der double Variable zahl
    double stern_zeiger_zahl = *zeiger_zahl; // double Variable zum Überprüfen des Dereferenzierungsoperators angewandt auf den Zeiger (*zeiger_zahl)

    printf("Wert der Variable 'zeichen' ist: %c \n", zeichen);                          // Ausgabe des Wertes der Variable
    printf("Wert der Variable 'zeiger_zeichen' ist: %p \n", zeiger_zeichen);            // Ausgabe des Zeigers auf die Adresse der Variable
    printf("Wert der Variable 'stern_zeiger_zeichen' ist: %c \n", stern_zeiger_zeichen); // Ausgabe der Dereferenzierung des Zeigers
    printf("Wert der Variable 'ref_zeichen' ist: %c \n", ref_zeichen);                  // Ausgabe der Referenz (konstanter Zeiger, der direkt dereferenziert wird)
    printf("------------------------------\n");
    printf("Wert der Variable 'ganze_zahl' ist: %i \n", ganze_zahl);                    // Ausgabe des Wertes der Variable
    printf("Wert der Variable 'zeiger_ganze_zahl' ist: %p \n", zeiger_ganze_zahl);      // Ausgabe des Zeigers auf die Adresse der Variable
    printf("Wert der Variable 'stern_zeiger_ganze_zahl' ist: %i \n", stern_zeiger_ganze_zahl); // ....
    printf("Wert der Variable 'ref_ganze_zahl' ist: %i \n", ref_ganze_zahl);
    printf("------------------------------\n");
    printf("Wert der Variable 'zahl' ist: %f \n", zahl);
    printf("Wert der Variable 'zeiger_zahl' ist: %p \n", zeiger_zahl);
    printf("Wert der Variable 'stern_zeiger_zahl' ist: %f \n", stern_zeiger_zahl);
    printf("Wert der Variable 'ref_zahl' ist: %f \n", ref_zahl);
}
```

Zeiger, Adressen und Referenzen

am Beispiel von char, int und double

Das folgende C++ Programm illustriert die V

Zeiger_1.cpp

```
#include <iostream>                                  // Ein- u

int main(){                                          // Haupt
    char zeichen = 'H';                              // Defini
    char* zeiger_zeichen = &zeichen;                 // Defini
    char& ref_zeichen = zeichen;                     // Defini
    char stern_zeiger_zeichen = *zeiger_zeichen; // char \

    int ganze_zahl = 234767;                                 // D
    int* zeiger_ganze_zahl = &ganze_zahl;                    // D
    int& ref_ganze_zahl = ganze_zahl;                        // D
    int stern_zeiger_ganze_zahl = *zeiger_ganze_zahl; // i

    double zahl = 2.47654673;                  // Definitior
    double* zeiger_zahl = &zahl;               // Definitior
    double& ref_zahl = zahl;                   // Definitior
    double stern_zeiger_zahl = *zeiger_zahl; // double Var

    printf("Wert der Variable 'zeichen' ist: %c \n", zeichen);                          // Ausgabe des Wertes der Variable
    printf("Wert der Variable 'zeiger_zeichen' ist: %p \n", zeiger_zeichen);            // Ausgabe des Zeigers auf die Adresse der Variable
    printf("Wert der Variable 'stern_zeiger_zeichen' ist: %c \n", stern_zeiger_zeichen); // Ausgabe der Dereferenzierung des Zeigers
    printf("Wert der Variable 'ref_zeichen' ist: %c \n", ref_zeichen);                  // Ausgabe der Referenz (konstanter Zeiger, der direkt dereferenziert wird)
    printf("------------------------------\n");
    printf("Wert der Variable 'ganze_zahl' ist: %i \n", ganze_zahl);                    // Ausgabe des Wertes der Variable
    printf("Wert der Variable 'zeiger_ganze_zahl' ist: %p \n", zeiger_ganze_zahl);      // Ausgabe des Zeigers auf die Adresse der Variable
    printf("Wert der Variable 'stern_zeiger_ganze_zahl' ist: %i \n", stern_zeiger_ganze_zahl); // ....
    printf("Wert der Variable 'ref_ganze_zahl' ist: %i \n", ref_ganze_zahl);
    printf("------------------------------\n");
    printf("Wert der Variable 'zahl' ist: %f \n", zahl);
    printf("Wert der Variable 'zeiger_zahl' ist: %p \n", zeiger_zahl);
    printf("Wert der Variable 'stern_zeiger_zahl' ist: %f \n", stern_zeiger_zahl);
    printf("Wert der Variable 'ref_zahl' ist: %f \n", ref_zahl);
}
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ g++ Zeiger_1.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ ./a.out
Wert der Variable 'zeichen' ist: H
Wert der Variable 'zeiger_zeichen' ist: 0x7ffc0bf541be
Wert der Variable 'stern_zeiger_zeichen' ist: H
Wert der Variable 'ref_zeichen' ist: H
------------------------------
Wert der Variable 'ganze_zahl' ist: 234767
Wert der Variable 'zeiger_ganze_zahl' ist: 0x7ffc0bf541c0
Wert der Variable 'stern_zeiger_ganze_zahl' ist: 234767
Wert der Variable 'ref_ganze_zahl' ist: 234767
------------------------------
Wert der Variable 'zahl' ist: 2.476547
Wert der Variable 'zeiger_zahl' ist: 0x7ffc0bf541c8
Wert der Variable 'stern_zeiger_zahl' ist: 2.476547
Wert der Variable 'ref_zahl' ist: 2.476547
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$
```

Eindimensionale integrierte C++ Arrays *(Vektoren)*

Arrays und Zeiger

Alle die bis zu diesem Abschnitt besprochenen Datentypen verbinden den Namen des deklarierten Typs mit einem einzelnen Wert. Um die in der Physik und Mathematik definierten Größen wie Vektoren und Matrizen in einem Computerprogramm adäquat abbilden zu können, gibt es die sogenannten Datenarrays. Für einen Typ T ist "T Name[N];" die Deklaration eines Typs "Array mit N Elementen vom Typ T", wobei die Elemente von 0 bis N-1 indiziert werden. Diese Konstruktion ist der mathematischen Definition eines N-dimensionalen Vektors $\vec{v} = (v_0, v_1, \ldots, v_{N-1})$ nicht unähnlich, wobei ein C++ Array natürlich von viel allgemeinerer Struktur ist, da der Typ des Arrays ja nicht nur auf Zahlen-Arrays limitiert ist. C++ Daten-Arrays und das Konstrukt des Zeigers sind eng miteinander verwandt und der Name eines Arrays kann als Zeiger auf sein erstes Element verstanden werden. Diese Eigenschaft wollen wir, am Beispiel eines eindimensionalen int-Arrays, im Folgenden näher betrachten.

Array_1_zeiger.cpp

```cpp
#include <iostream>                         // Ein- und Ausgabebibliothek
using namespace std;                        // Benutze den Namensraum std

int main(){                                 // Hauptfunktion
    int v[7] = {1,4,6,8,9,5,3};             // Definition eines Integer-Arrays mit sieben Einträgen
    int* zeiger_v = v;                      // Definition des Zeigers auf das Integer-Array v
    int dim_v = sizeof(v)/sizeof(v[0]);     // Dimension des Arrays

    cout << "Das Array v ist ein Zeiger auf sein erstes Element: v=" << v << " und &v[0]=" << &v[0] << endl;
    cout << "Die Dimension unseres Arrays v ist: dim(v)=" << dim_v << endl << endl;

    printf("%20s %20s %20s %25s %30s %30s \n", "Index i Array", "Wert v[i]", "Referenz &v[i]", "Zeiger zeiger_v+i", "Deref. Zeiger *(zeiger_v+i)", "Deref. Adresse *(&v[i])");

    for(int i=0; i<dim_v; ++i){         // Schleifen Anfang ueber alle
        printf("%20i ", i);             // Ausgabe des Indexes i
        printf("%20i ", v[i]);          // Ausgabe des i-ten Wertes des Arrays
        printf("%20p ", &v[i]);         // Ausgabe der Referenz (Addresse) des i-ten Eintrages im Array
        printf("%25p ", zeiger_v+i);    // Ausgabe des Zeigers auf den i-ten Eintrag im Array
        printf("%30i ", *(zeiger_v+i)); // Dereferenzierung des Zeigers auf den i-ten Eintrag im Array
        printf("%30i \n", *(&v[i]));    // Dereferenzierung der Addresse des i-ten Eintrages im Array
    }                                   // Ende der Schleife
    printf("\nEs gilt z.B. für i=3:\nv[3] = %i \n*(v+3) = %i \n*(&v[0]+3) = %i \n", v[3], *(v+3), *(&v[0]+3));
}
```

Eindimensionale integrierte C++ Arrays *(Vektoren)*

Navigieren in Arrays

Der Zugriff auf den Wert eines Array-Elementes kann entweder durch die Angabe des entsprechenden Vektor-Index (v[i]), oder durch den dereferenzierten Wert seiner Zeigerposition im Array erfolgen: *(zeiger_v+ i). Es gelten dabei die folgenden äquivalenten Formulierungen, um den Wert des i-ten Eintrages im Array zu erhalten: v[i] =*(v+i) = *(&v[o]+i) . Diese Eigenschaft wird in den vier letzten Zeilen der Terminalausgabe für i=3 überprüft.

Array_1_zeiger.cpp

```cpp
#include <iostream>                         // Ein- und Ausgabebibliothek
using namespace std;                        // Benutze den Namensraum std

int main(){                                 // Hauptfunktion
    int v[7] = {1,4,6,8,9,5,3};             // Definition eines Integer-Arrays mit sieben Einträgen
    int* zeiger_v = v;                      // Definition des Zeigers auf das Integer-Array v
    int dim_v = sizeof(v)/sizeof(v[0]);     // Dimension des Arrays

    cout << "Das Array v ist ein Zeiger auf sein erstes Element: v=" << v << " und &v[0]=" << &v[0] << endl;
    cout << "Die Dimension unseres Arrays v ist: dim(v)=" << dim_v << endl << endl;

    printf("%20s %20s %20s %25s %30s %30s \n", "Index i Array", "Wert v[i]", "Referenz &v[i]", "Zeiger zeiger_v+i", "Deref. Zeiger *(zeiger_v+i)", "Deref. Adresse *(&v[i])");

    for(int i=0; i<dim_v; ++i){         // Schleifen Anfang ueber alle
        printf("%20i ", i);             // Ausgabe des Indexes i
        printf("%20i ", v[i]);          // Ausgabe des i-ten Wertes des Arrays
        printf("%20p ", &v[i]);         // Ausgabe der Referenz (Addresse) des i-ten Eintrages im Array
        printf("%25p ", zeiger_v+i);    // Ausgabe des Zeigers auf den i-ten Eintrag im Array
        printf("%30i ", *(zeiger_v+i)); // Dereferenzierung des Zeigers auf den i-ten Eintrag im Array
        printf("%30i \n", *(&v[i]));    // Dereferenzierung der Addresse des i-ten Eintrages im Array
    }                                   // Ende der Schleife
    printf("\nEs gilt z.B. für i=3:\nv[3] = %i \n*(v+3) = %i \n*(&v[0]+3) = %i \n", v[3], *(v+3), *(&v[0]+3));
}
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ g++ Array_1_zeiger.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ ./a.out
Das Array v ist ein Zeiger auf sein erstes Element: v=0x7ffe227c9b40 und &v[0]=0x7ffe227c9b40
Die Dimension unseres Arrays v ist: dim(v)=7

      Index i Array           Wert v[i]      Referenz &v[i]        Zeiger zeiger_v+i    Deref. Zeiger *(zeiger_v+i)       Deref. Adresse *(&v[i])
                0                   1      0x7ffe227c9b40           0x7ffe227c9b40                            1                             1
                1                   4      0x7ffe227c9b44           0x7ffe227c9b44                            4                             4
                2                   6      0x7ffe227c9b48           0x7ffe227c9b48                            6                             6
                3                   8      0x7ffe227c9b4c           0x7ffe227c9b4c                            8                             8
                4                   9      0x7ffe227c9b50           0x7ffe227c9b50                            9                             9
                5                   5      0x7ffe227c9b54           0x7ffe227c9b54                            5                             5
                6                   3      0x7ffe227c9b58           0x7ffe227c9b58                            3                             3

Es gilt z.B. für i=3:
v[3] = 8
*(v+3) = 8
*(&v[0]+3) = 8
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$
```

Array_1_zeiger.cpp

```cpp
#include <iostream>                         // Ein- und Ausgabebibliothek
using namespace std;                        // Benutze den Namensraum std

int main(){                                 // Hauptfunktion
    int v[7] = {1,4,6,8,9,5,3};             // Definition eines Integer-Arrays mit sieben Einträgen
    int* zeiger_v = v;                      // Definition des Zeigers auf das Integer-Array v
    int dim_v = sizeof(v)/sizeof(v[0]);     // Dimension des Arrays

    cout << "Das Array v ist ein Zeiger auf sein erstes Element: v=" << v << " und &v[0]=" << &v[0] << endl;
    cout << "Die Dimension unseres Arrays v ist: dim(v)=" << dim_v << endl << endl;

    printf("%20s %20s %20s %25s %30s %30s \n", "Index i Array", "Wert v[i]", "Referenz &v[i]", "Zeiger zeiger_v+i", "Deref. Zeiger *(zeiger_v+i)", "Deref. Adresse *(&v[i])");

    for(int i=0; i<dim_v; ++i){             // Schleifen Anfang ueber alle
        printf("%20i ", i);                 // Ausgabe des Indexes i
        printf("%20i ", v[i]);              // Ausgabe des i-ten Wertes des Arrays
        printf("%20p ", &v[i]);             // Ausgabe der Referenz (Addresse) des i-ten Eintrages im Array
        printf("%25p ", zeiger_v+i);        // Ausgabe des Zeigers auf den i-ten Eintrag im Array
        printf("%30i ", *(zeiger_v+i));     // Dereferenzierung des Zeigers auf den i-ten Eintrag im Array
        printf("%30i \n", *(&v[i]));        // Dereferenzierung der Addresse des i-ten Eintrages im Array
    }                                       // Ende der Schleife
    printf("\nEs gilt z.B. für i=3:\nv[3] = %i \n*(v+3) = %i \n*(&v[0]+3) = %i \n", v[3], *(v+3), *(&v[0]+3));
}
```

Strings als eindimensionale Arrays von Zeichen

am Beispiel des „Hallo Welt" Programms

Strings als Arrays von Zeichen

Obwohl wir es im Großteil der Vorlesung mit Zahlen-Arrays zu tun haben werden, wollen wir die Verwendung von anderen C++ Array-Typen auch einmal am Beispiel eines Arrays bestehend aus Zeichen (ein String, ein eindimensionales Array vom Typ char) verdeutlichen. Ein String ist eine Abfolge (ein Array) von mehreren Zeichen. Nehmen wir z.B. den Satz "Hallo, du schoene Welt!". Dieser String stellt ein Array des Typs char dar, wobei seine Dimension N die Anzahl der Zeichen (hier speziell N=23) ist. Im folgenden Programm wurde dieses char-Array mittels der Anweisung char satz[] = {'H','a', ... ,'!'}; deklariert und sofort mit den einzelnen Zeichen initialisiert. Initialisiert man ein Array direkt bei seiner Deklaration, muss man die Dimension des Arrays (hier N=23) nicht explizit angeben. In gleicher Weise wie im vorigen Beispiel des int-Arrays, wird der Zeiger auf das Array definiert, die Dimension des Arrays berechnet und mehrere Größen der jeweiligen Elemente des Arrays ausgegeben.

HelloWorld_1_zeiger.cpp

```
#include <iostream>                         // Ein- und Ausgabebibliothek

int main(){                                 // Hauptfunktion
    char satz[] = {'H','a','l','l','o',',',' ','d','u',' ','s','c','h','o','e','n','e',' ','W','e','l','t','!'}; // Deklaration des eindimensionalen Arrays von Zeichentypen und Initialisierung
    char* zeiger_satz = satz;                       // Deklaration des Zeigers auf das eindimensionalen Arrays von Zeichentypen
    int anz_zeichen = sizeof(satz)/sizeof(satz[0]); // Dimension des eindimensionalen Arrays (dim= 23 = 92/4 = (Byte fuer den gesamten Satz)/(Byte für ein Zeichen)

    printf("%20s %20s %20s %25s %35s %35s \n", "Index i Array", "Zeichen satz[i]", "Referenz &satz[i]", "Zeiger zeiger_satz+i", "Deref. Zeiger *(zeiger_satz+i)", "Deref. Adresse *(&satz[i])");

    for(int i=0; i<anz_zeichen; ++i){       // Schleifen Anfang ueber alle Zeichen im Satz
        printf("%20i ", i);                 // Ausgabe des Indexes i
        printf("%20c ", satz[i]);           // Ausgabe des i-ten Wertes des Arrays
        printf("%20p ", &satz[i]);          // Ausgabe der Referenz (Addresse) des i-ten Eintrages im Array
        printf("%25p ", zeiger_satz+i);     // Ausgabe des Zeigers auf den i-ten Eintrag im Array
        printf("%35c ", *(zeiger_satz+i)); // Dereferenzierung des Zeigers auf den i-ten Eintrag im Array
        printf("%35c \n", *(&satz[i]));     // Dereferenzierung der Addresse des i-ten Eintrages im Array
    }                                       // Ende der Schleife
}
```

Arrays : bash — Konsole

Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ g++ HelloWorld_1_zeiger.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ ./a.out
      Index i Array      Zeichen satz[i]    Referenz &satz[i]      Zeiger zeiger_satz+i      Deref. Zeiger *(zeiger_satz+i)         Deref. Adresse *(&satz[i])
                0                    H       0x7fff1f6a2160           0x7fff1f6a2160                                H                                H
                1                    a       0x7fff1f6a2161           0x7fff1f6a2161                                a                                a
                2                    l       0x7fff1f6a2162           0x7fff1f6a2162                                l                                l
                3                    l       0x7fff1f6a2163           0x7fff1f6a2163                                l                                l
                4                    o       0x7fff1f6a2164           0x7fff1f6a2164                                o                                o
                5                    ,       0x7fff1f6a2165           0x7fff1f6a2165                                ,                                ,
                6                            0x7fff1f6a2166           0x7fff1f6a2166                                                                 
                7                    d       0x7fff1f6a2167           0x7fff1f6a2167                                d                                d
                8                    u       0x7fff1f6a2168           0x7fff1f6a2168                                u                                u
                9                            0x7fff1f6a2169           0x7fff1f6a2169                                                                 
               10                    s       0x7fff1f6a216a           0x7fff1f6a216a                                s                                s
               11                    c       0x7fff1f6a216b           0x7fff1f6a216b                                c                                c
               12                    h       0x7fff1f6a216c           0x7fff1f6a216c                                h                                h
               13                    o       0x7fff1f6a216d           0x7fff1f6a216d                                o                                o
               14                    e       0x7fff1f6a216e           0x7fff1f6a216e                                e                                e
               15                    n       0x7fff1f6a216f           0x7fff1f6a216f                                n                                n
               16                    e       0x7fff1f6a2170           0x7fff1f6a2170                                e                                e
               17                            0x7fff1f6a2171           0x7fff1f6a2171                                                                 
               18                    W       0x7fff1f6a2172           0x7fff1f6a2172                                W                                W
               19                    e       0x7fff1f6a2173           0x7fff1f6a2173                                e                                e
               20                    l       0x7fff1f6a2174           0x7fff1f6a2174                                l                                l
               21                    t       0x7fff1f6a2175           0x7fff1f6a2175                                t                                t
               22                    !       0x7fff1f6a2176           0x7fff1f6a2176                                !                                !
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ 
```

```cpp
int main(){                                 // Hauptfunktion
    char satz[] = {'H','a','l','l','o',',',' ','d','u',' ','s','c','h','o','e','n','e',' ','W','e','l','t','!'}; // Deklaration des eindimensionalen Arrays von Zeichentypen und Initialisierung
    char* zeiger_satz = satz;                   // Deklaration des Zeigers auf das eindimensionalen Arrays von Zeichentypen
    int anz_zeichen = sizeof(satz)/sizeof(satz[0]); // Dimension des eindimensionalen Arrays (dim= 23 = 92/4 = (Byte fuer den gesamten Satz)/(Byte für ein Zeichen)

    printf("%20s %20s %20s %25s %35s %35s \n", "Index i Array", "Zeichen satz[i]", "Referenz &satz[i]", "Zeiger zeiger_satz+i", "Deref. Zeiger *(zeiger_satz+i)", "Deref. Adresse *(&satz[i])");

    for(int i=0; i<anz_zeichen; ++i){       // Schleifen Anfang ueber alle Zeichen im Satz
        printf("%20i ", i);                 // Ausgabe des Indexes i
        printf("%20c ", satz[i]);           // Ausgabe des i-ten Wertes des Arrays
        printf("%20p ", &satz[i]);          // Ausgabe der Referenz (Addresse) des i-ten Eintrages im Array
        printf("%25p ", zeiger_satz+i);     // Ausgabe des Zeigers auf den i-ten Eintrag im Array
        printf("%35c ", *(zeiger_satz+i)); // Dereferenzierung des Zeigers auf den i-ten Eintrag im Array
        printf("%35c \n", *(&satz[i]));     // Dereferenzierung der Addresse des i-ten Eintrages im Array
    }                                       // Ende der Schleife
}
```

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from matplotlib.ticker import NullFormatter
from PIL import Image

anz_zeichen = len(Hello_World)

im = Image.open('./illustration_DeborahMoldawski.jpg')
x_max=5334
y_max=4000

params = {
    'figure.figsize'    : [17.34,13]
}
matplotlib.rcParams.update(params)
plt.style.use('dark_background')
nullfmt = NullFormatter()

fig, ax = plt.subplots()
ax.yaxis.set_major_formatter(nullfmt)
ax.xaxis.set_major_formatter(nullfmt)

ax.imshow(im)

# define the colormap
cmap = plt.cm.Blues
zeichen_colors = cmap(np.linspace(0,1,2*anz_zeichen))

textstr0=r'$Zeiger \, und \, Arrays'+'$'
textstr1=r'$\vec{v}=v_i \Rightarrow'+'$'
textstr2=r'$char \,\, satz[23]'+'$'
props = dict(boxstyle='round', facecolor='black', alpha=0.95)
props1 = dict(boxstyle='square', facecolor='black', alpha=0.65)
ax.text(2750, 1500, textstr0, fontsize=37, verticalalignment='top', horizontalalignment='center', bbox=props)
ax.text(2200, 2320, textstr1, fontsize=37, verticalalignment='top', horizontalalignment='right', bbox=props)
ax.text(2500, 2520, textstr2, fontsize=35, verticalalignment='top', horizontalalignment='left', bbox=props)

r=y_max/2
r1=r*0.65
i=0
for addresse in Hello_World_Adressen:
    props2 = dict(boxstyle='round', facecolor=zeichen_colors[anz_zeichen+i], alpha=0.95)

    phi = - i*1.8*np.pi/len(Hello_World) - np.pi/6
    x_0 = x_max/2 + 1.4*r1*np.sin(phi)
    y_0 = y_max/2 + r1*np.cos(phi)
    R_x_0 = x_max/2 - 600 + 1.4*r*np.sin(phi)
    R_y_0 = y_max/2 + r*np.cos(phi)

    ax.text(x_0, y_0, Hello_World[i], fontsize=37, verticalalignment='top', horizontalalignment='left',
    bbox=props2)
    ax.text(R_x_0, R_y_0, addresse, fontsize=25, verticalalignment='top', horizontalalignment='left',
    bbox=props1, color="white")
    i = i + 1

plt.savefig("HelloWorld_Zeiger.png", bbox_inches="tight")
```

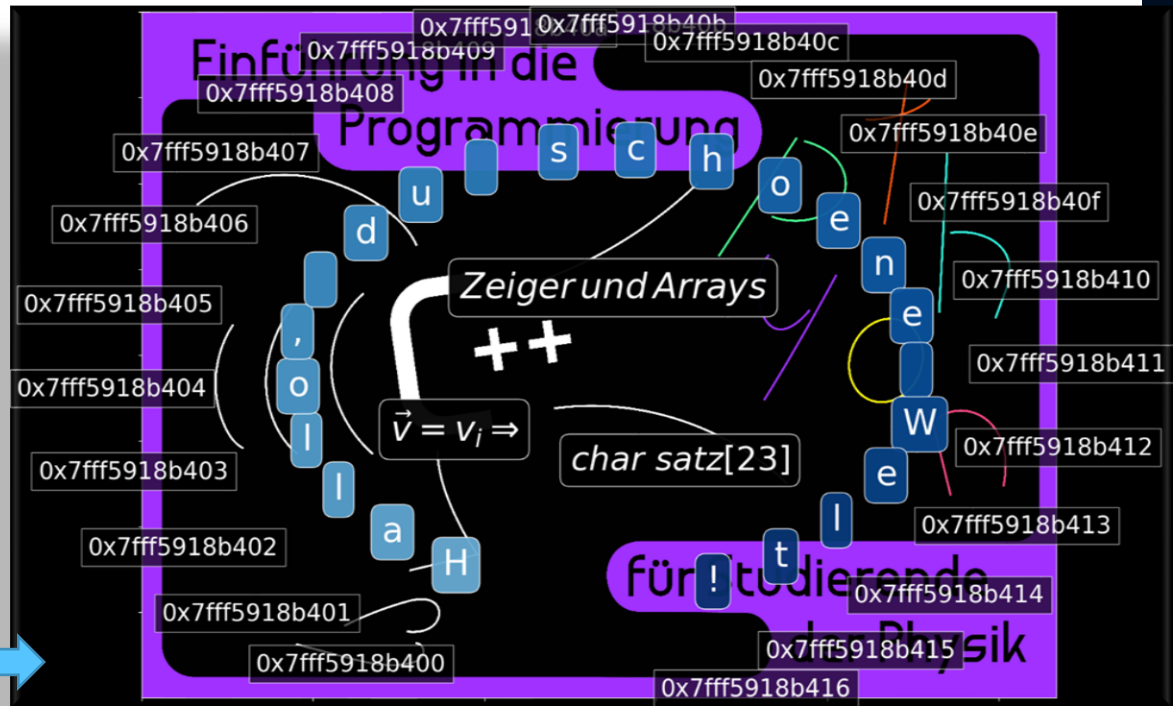
Python Skript „plot_bild.py“

```cpp
#include <iostream>                     // Ein- und Ausgabebibliothek

int main(){                             // Hauptfunktion
    int i;                              // Index des eindimensionalen Arrays
    char satz[] = {'H','a','l','l','o',',',' ','d','u',' ','s','c','h','o','e','n','e',' ','W','e','l','t','!'}; // Deklaration des eindimensionalen Arrays von
    Zeichentypen und Initialisierung
    int anz_zeichen = sizeof(satz)/sizeof(satz[0]); // Dimension des eindimensionalen Arrays (dim= 23 = 92/4 = (Byte fuer den gesamten Satz)/(Byte für ein Zeichen)

    printf("Hello_World=[");
    for(i=0; i<anz_zeichen-1; ++i){     // Schleifen Anfang ueber alle
        printf("'%c',", satz[i]);       // Ausgabe des i-ten Wertes des Arrays
    }                                   // Ende der Schleife
    printf("'%c'] \n", satz[i]);

    printf("Hello_World_Adressen=[");
    for(i=0; i<anz_zeichen-1; ++i){     // Schleifen Anfang ueber alle
        printf("'%p',", &satz[i]);      // Ausgabe der Referenz (Addresse) des i-ten Eintrages im Array
    }                                   // Ende der Schleife
    printf("'%p'] \n", &satz[i]);
}
```

C++ Programm
„HelloWorld_1_zeiger_python.cpp“

```
g++ HelloWorld_1_zeiger_python.cpp
./a.out > Datenliste.py
cat Datenliste.py plot_bild.py > HelloWorld_1_zeiger_python.py
python3 HelloWorld_1_zeiger_python.py
```

Shell Skript „plot_bild.sh“



Arrays an Funktionen übergeben

Möchte man ein Array an eine Funktion als Argument übergeben, sollte man dies nicht über die einzelnen Werte des Arrays machen, da man dann die Array-Einträge nicht verändern kann. Stattdessen übergibt man ein Array (dies gilt auch für mehrdimensionale Arrays) als Zeiger auf sein erstes Element mit einem zusätzlichen Vermerk zu seiner Dimension.

Als ein einfaches Beispiel betrachten wir uns die folgende Programmieraufgabe. Sie sollen eine C++ Funktion schreiben, die den 4. und 6. Eintrag eines eindimensionalen Arrays vertauscht. Nehmen wir z.B. den Vektor $\vec{v} = (v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) = (1, 4, 6, 8, 5, 3, 8, 9, 3)$ und vertauschen seinen 4. und 6. Eintrag. Ein solches Vertauschen ist gleichbedeutend mit dem Tausch der Werte von v_3 mit v_5 und das Ergebnis wäre der Vektor $\vec{v} = (1, 4, 6, 3, 5, 8, 8, 9, 3)$. Die zu entwerfende Tausch-Funktion sollte natürlich nur funktionieren, wenn die Dimension des Arrays mindestens 6 ist. Das nebenstehende C++ Programm stellt einen solchen Komponentenaustausch des Vektors dar und verwendet dabei eine definierte Tausch-Funktion. Die konstruierte Tausch-Funktion "void Tausche_35(int* pv, int dim){ ... }" hat zwar keinen Rückgabewert, dennoch verändert sie die Komponenten des int-Arrays, indem sie auf Zeigerebene die Werte des Arrays abändert und vertauscht. Im main()-Programm wird zunächst das Array definiert, dann mittels einer normalen (Index-basierten) for-Schleife im Terminal ausgegeben, danach vertauscht und dann nochmals im Terminal ausgegeben. Die zweite Terminalausgabe des Vektors benutzt dabei eine Bereichsbasierte for-Schleife.

C++ Arrays und Funktionen

Array_funktionen_1.cpp

```cpp
#include <iostream>                                 // Ein- und Ausgabebibliothek

/** Funktion ohne Rueckgabewert, die als Argument den Zeiger auf
 * ein eindimensionales int-Array und seine Dimension hat.
 * Falls die Dimension groesser als 5 ist, vertauscht sie den vierten
 * Eintrag im Array (v[3]) mit dem sechsten (v[5])
 **/
void Tausche_35(int* pv, int dim){
    if( dim > 5 ){
        printf("Vertausche v[3] mit v[5] \n");
        int tmp = *(pv+3);                          // Speichert den Wert von v[3]
        *(pv+3) = *(pv+5);                          // Schreibt den Wert von v[5] in v[3]
        *(pv+5) = tmp;                              // Schreibt den alten Wert von v[3] in v[5]
    }
}

int main(){                                         // Hauptfunktion
    int v[] = {1,4,6,8,5,3,8,9,3};                  // Definition eines Integer-Arrays mit neun Einträgen
    int dim_v = sizeof(v)/sizeof(v[0]);             // Dimension des Arrays

    //Ausgabe von v mittels einer Index-basierten for-Schleife
    printf("v = ( %2i", v[0]);
    for(int i=1; i<dim_v; ++i){
        printf(" ,%2i", v[i]);
    }
    printf(" )\n");

    Tausche_35(v, dim_v);

    //Ausgabe von v mittels einer Bereichsbasierten for-Schleife
    printf("v = ( ");
    for(int n : v){          // Man haette an dieser Stelle auch "for(auto n : v){" schreiben koennen
        printf("%2i, ", n);
    }
    printf("\b\b )\n");
}
```

Array_funktionen_1.cpp

```cpp
#include <iostream>                         // Ein- und Ausgabebibliothek

/** Funktion ohne Rueckgabewert, die als Argument den Zeiger auf
 * ein eindimensionales int-Array und seine Dimension hat.
 * Falls die Dimension groesser als 5 ist, vertauscht sie den vierten
 * Eintrag im Array (v[3]) mit dem sechsten (v[5])
 **/
void Tausche_35(int* pv, int dim){
    if( dim > 5 ){
        printf("Vertausche v[3] mit v[5] \n");
        int tmp = *(pv+3);                  // Speichert den Wert von v[3]
        *(pv+3) = *(pv+5);                  // Schreibt den Wert von v[5] in v[3]
        *(pv+5) = tmp;                      // Schreibt den alten Wert von v[3] in v[5]
    }
}

int main(){                                 // Hauptfunktion
    int v[] = {1,4,6,8,5,3,8,9,3};          // Definition eines Integer-Arrays mit neun Einträgen
    int dim_v = sizeof(v)/sizeof(v[0]);     // Dimension des Arrays

    //Ausgabe von v mittels einer Index-basierten for-Schleife
    printf("v = ( %2i", v[0]);
    for(int i=1; i<dim_v; ++i){
        printf(" ,%2i", v[i]);
    }
    printf(" )\n");

    Tausche_35(v, dim_v);

    //Ausgabe von v mittels einer Bereichsbasierten for-Schleife
    printf("v = ( ");
    for(int n : v){         // Man haette an dieser Stelle auch "for(auto n : v){" schreiben koennen
        printf("%2i, ", n);
    }
    printf("\b\b )\n");
}
```

Möchte man ein Array an eine Funktion als Argument übergeben, sollte man dies nicht über die einzelnen Werte des Arrays machen, da man dann die Array-Einträge nicht verändern kann. Stattdessen übergibt man ein Array (dies gilt auch für mehrdimensionale Arrays) als Zeiger auf sein erstes Element mit einem zusätzlichen Vermerk zu seiner Dimension.

C++ Arrays und Funktionen

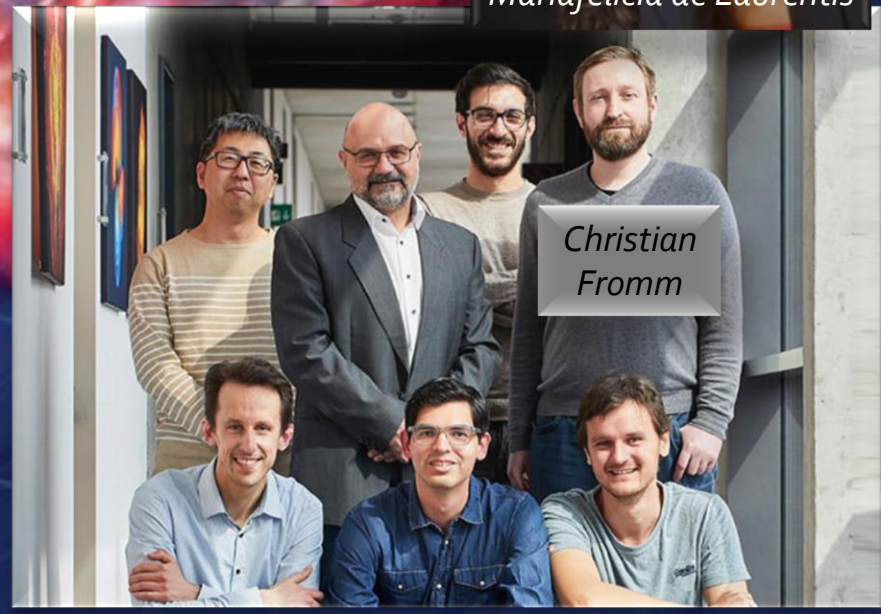
Pressekonferenz am 12. Mai 2022, 15.00 Uhr

Neue Erkenntnisse über das Schwarze Loch in unserer Milchstraße

Mariafelicia de Laurentis

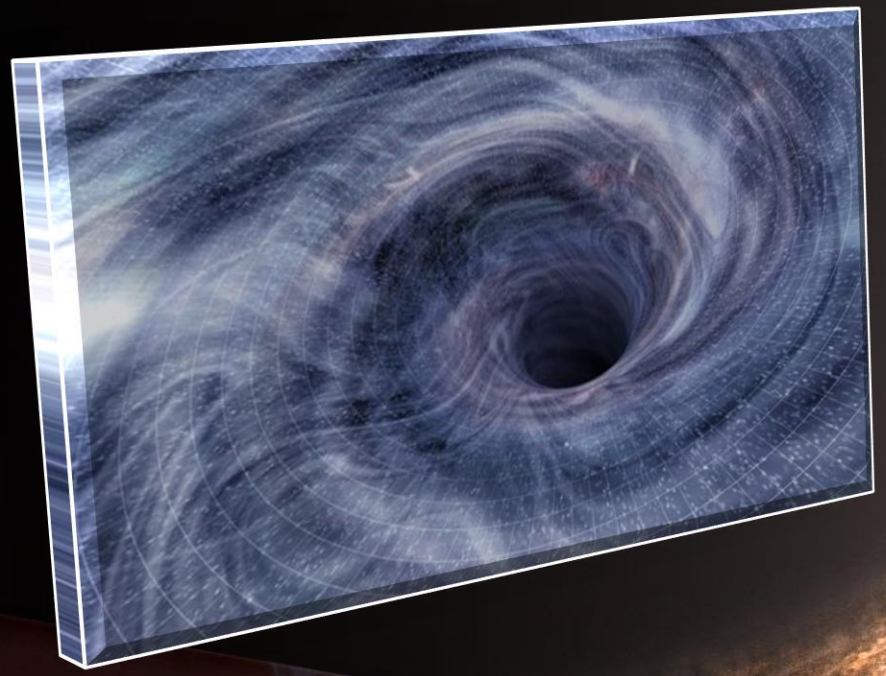
Die Europäische Südsternwarte (ESO) und das Event Horizon Telescope (EHT) Projekt halten eine Pressekonferenz ab, bei der neue Ergebnisse vom EHT zur Milchstraße präsentiert werden.

- **Wann:** Am 12. Mai um 15:00 Uhr MESZ
- **Wo:** Eridanus-Auditorium, ESO-Hauptsitz, Garching bei München, und online
- **Was:** Eine Pressekonferenz, auf der bahnbrechende Ergebnisse des EHT zur Milchstraße präsentiert werden
- **Wer:** Der ESO-Generaldirektor wird die einleitenden Worte sprechen. EHT-Projektleiter Huib Jan van Langevelde und Anton Zensus, Gründungs-Vorsitzender der EHT-Kollaboration, werden ebenfalls sprechen. Eine Runde von EHT-Forschenden werden die Ergebnisse erläutern und Fragen beantworten. Diese Runde besteht aus
  - Thomas Krichbaum, Max-Planck-Institut für Radioastronomie, Deutschland
  - Sara Issaoun, Center for Astrophysics | Harvard & Smithsonian, US und Radboud University, Niederlande
  - José L. Gómez, Instituto de Astrofísica de Andalucía (CSIC), Spanien
  - Christian Fromm, Universität Würzburg, Deutschland
  - Mariafelicia de Laurentis, University of Naples "Federico II" und the National Institute for Nuclear Physics (INFN), Italien



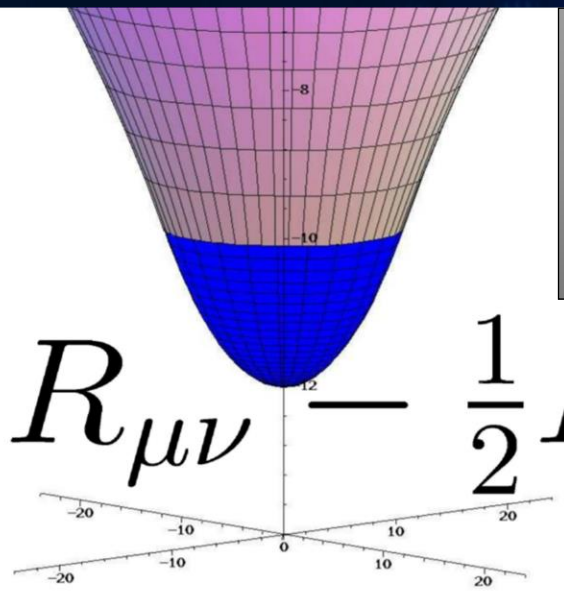
Christian Fromm

Was sind schwarze Löcher?



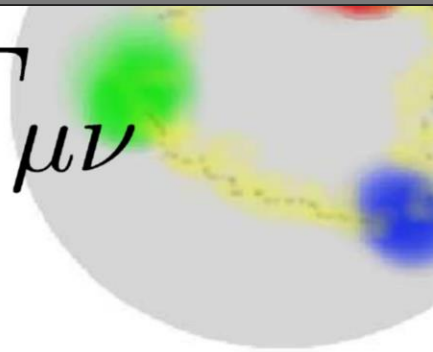
und wie sehen sie aus?

Grundlagen der Allgemeinen Relativitätstheorie



Vor etwa hundert Jahren (1915) stellte Albert Einstein seine „Allgemeine Relativitätstheorie“ (ART) der Öffentlichkeit vor.

$$R_{\mu\nu} - \frac{1}{2} R\, g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$

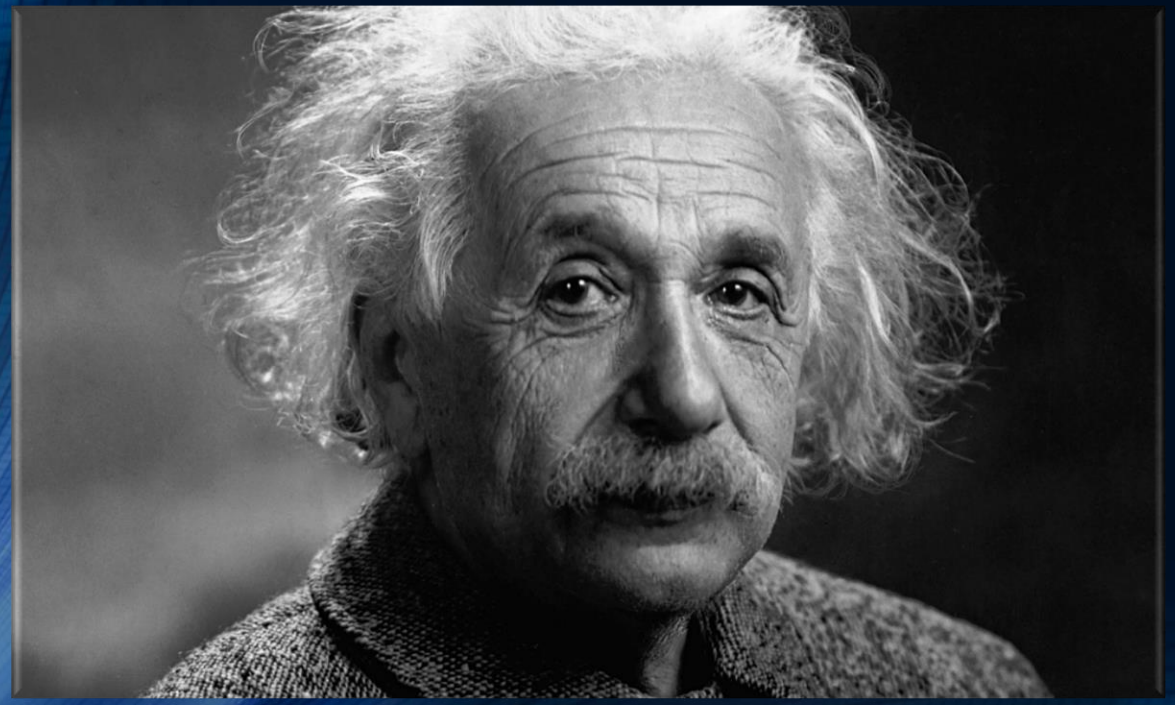


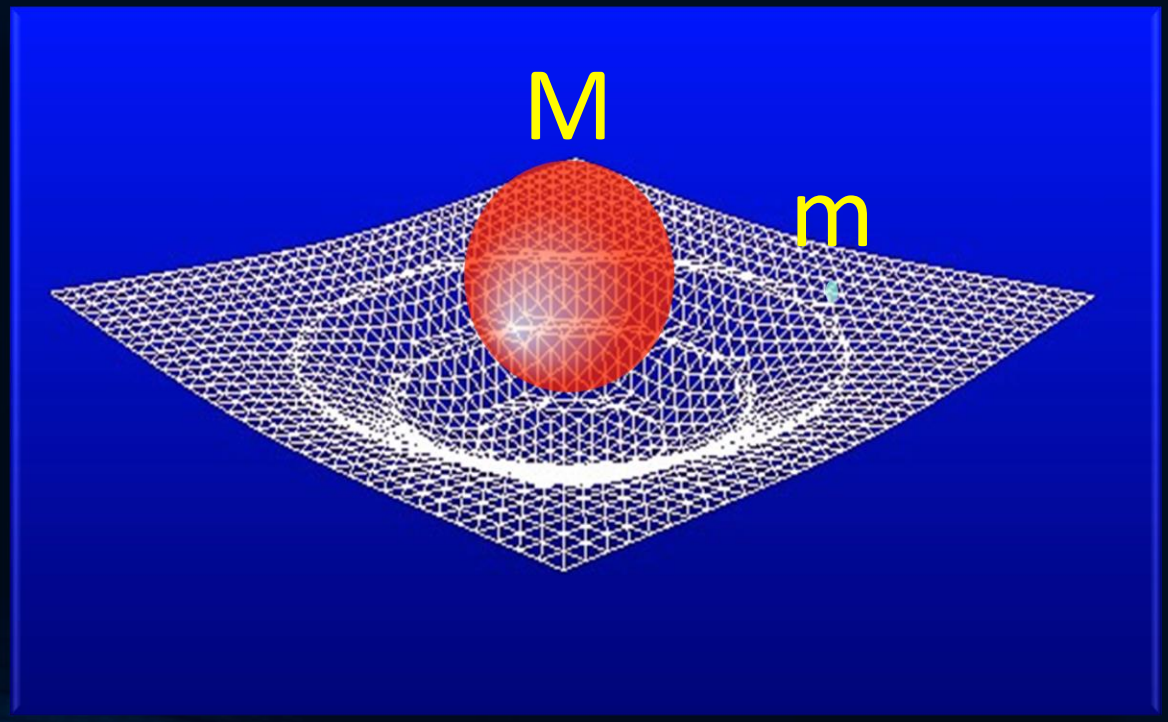
Die ART ist eine sehr revolutionäre Theorie. Sie besagt, dass jegliche Energieformen (z.B. Masse eines Körpers) die „Raumzeit“ verbiegen und durch diese Krümmung des Raumes und der Zeit die Gravitation (Schwerkraft) resultiert. -> Raumzeit-Krümmung = Energie

Raumzeit-Krümmung ist Gravitation?

Betrachten wir einen Objekt kleiner Masse m das um ein Objekt großer Masse M kreist (z.B. Erde um die Sonne)

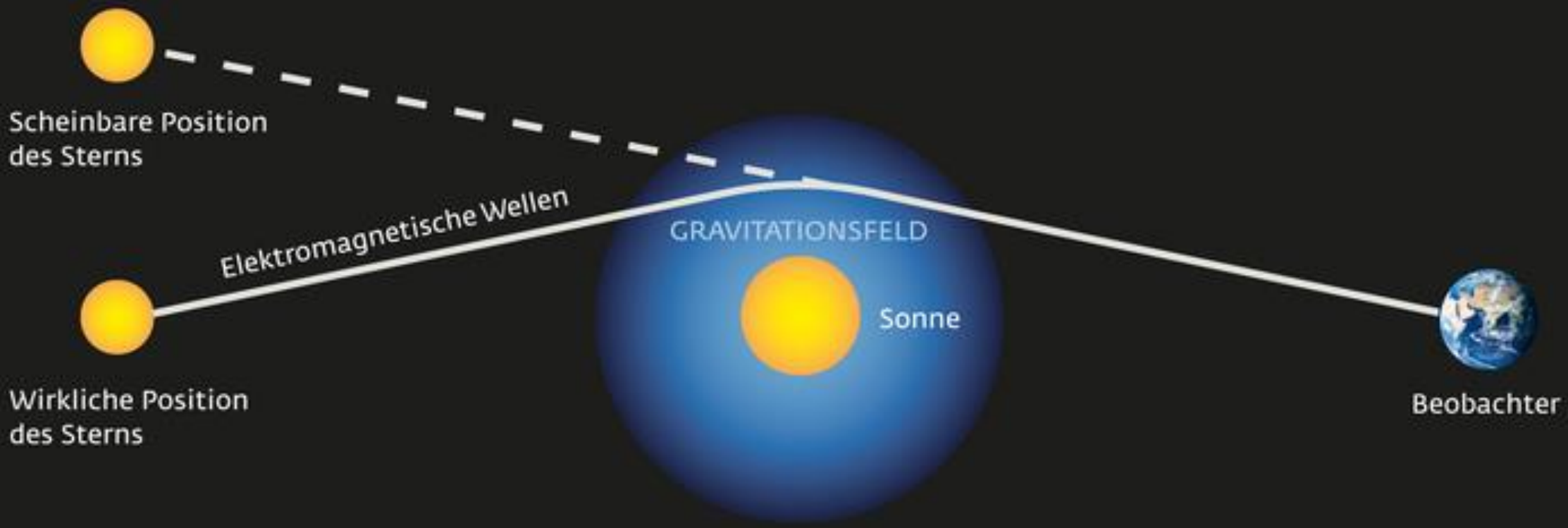
Einstein: Die Krümmung der Raumzeit, verursacht durch die große Masse, bestimmt die Umlaufbahn des kleinen Körpers und ist ursächlicher Grund der gravitativen Wechselwirkung





Erste Bestätigung der ART: Sonnenfinsternis 1919

Aufgrund des extrem revolutionären Charakters der ART glaubten viele Physiker zunächst nicht an Einsteins Theorie. Das änderte sich schlagartig im Jahre 1919:



Der Einstein-Ring

LRG 3-757: im Jahre 2007 mit dem Hubble Space Teleskop aufgenommen

Gravitative Zeitdilatation

Den Effekt der Zeitverbiegung kann man heutzutage sogar auf der Erde nachweisen -> Uhren ticken in den Bergen ein wenig schneller als im Tal.

News
12.02.2018
Drucken
Teilen

RELATIVITÄTSTHEORIE

Warum die Zeit im Gebirge schneller vergeht

Mit einem surrealen Effekt der Gravitationsphysik haben Wissenschaftler die Höhe eines Tunnels in den französischen Alpen bestimmt.

von Robert Gast

© ISTOCK / SKOUATROULIO (AUSSCHNITT)

2018 auf www.spektrum.de

Frankfurter Allgemeine

Physik & Mehr

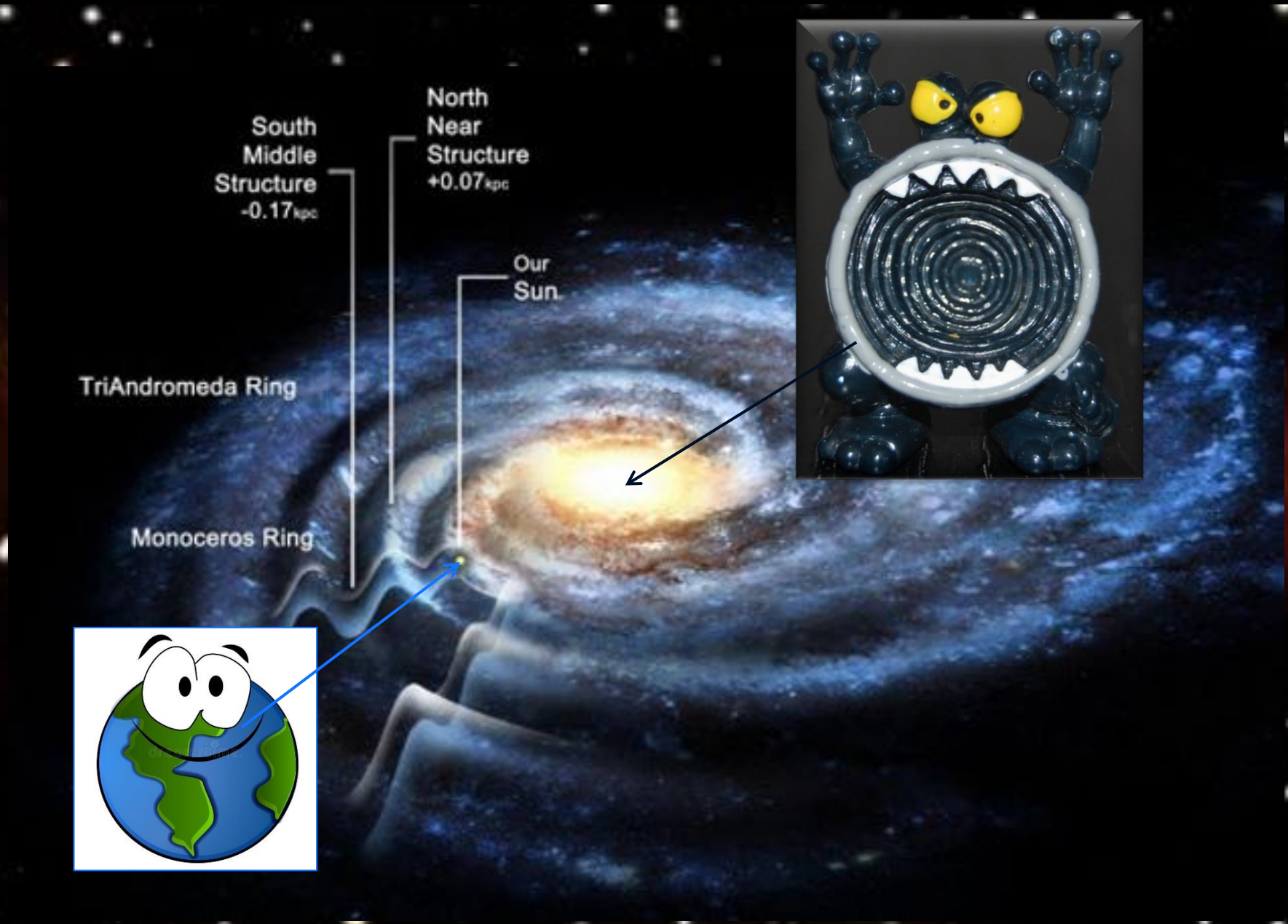
IZIN GENE KLIMA WELTRAUM GARTEN NETZRÄTSEL

ALLGEMEINE RELATIVITÄTSTHEORIE

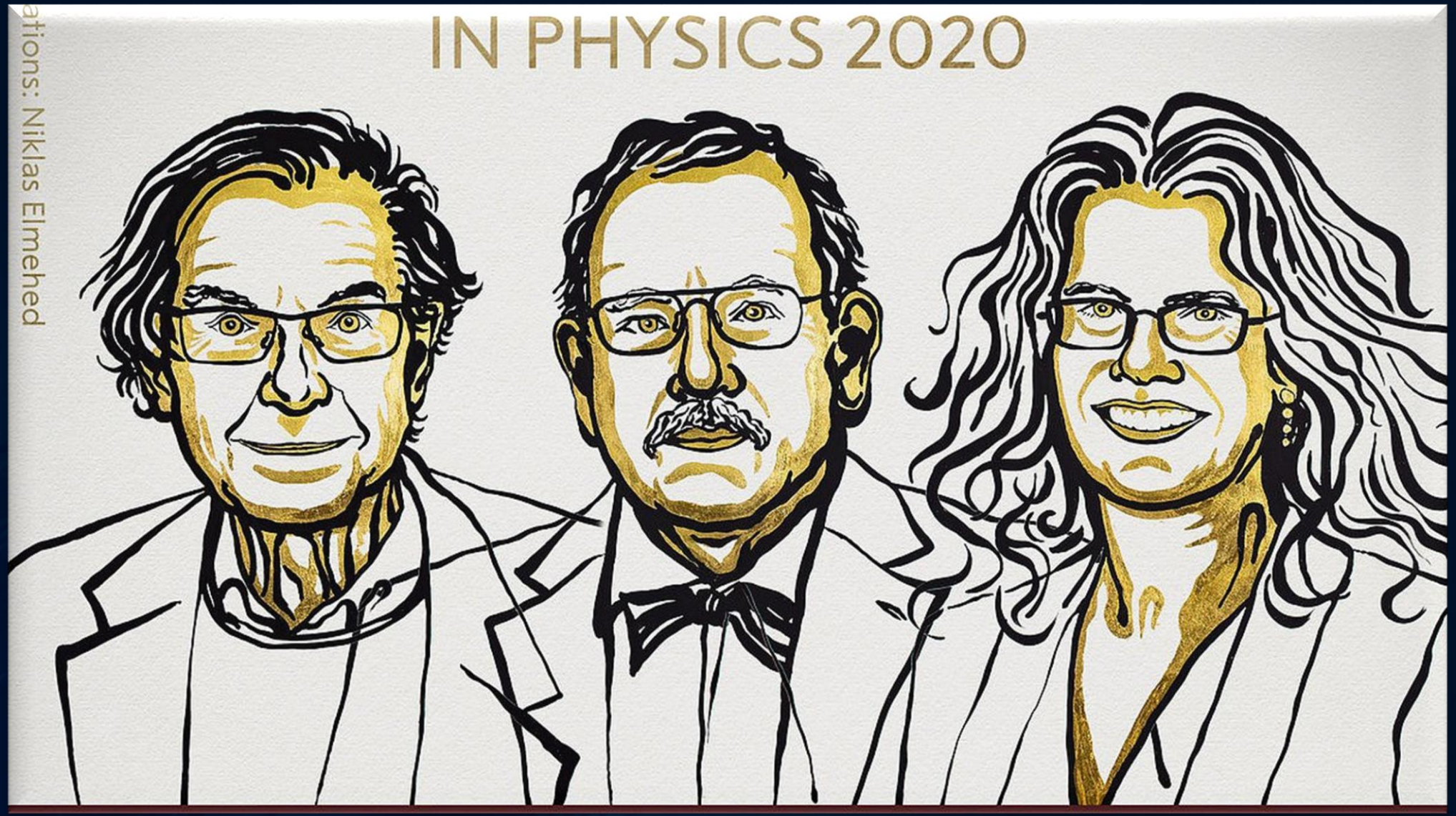
Hurra, wir hier unten leben länger!

VON ANNE HARDY - AKTUALISIERT AM 19.10.2010 - 06:00

Wie sieht das schwarze Loch im Zentrum unserer Galaxie aus?



Nobel Preis 2020



Für die Entdeckung, dass die Bildung von Schwarzen Löchern eine robuste Vorhersage der allgemeinen Relativitätstheorie ist (R. Penrose) und die Entdeckung eines supermassiven kompakten Objekts im Zentrum unserer Galaxie (R. Genzel und).

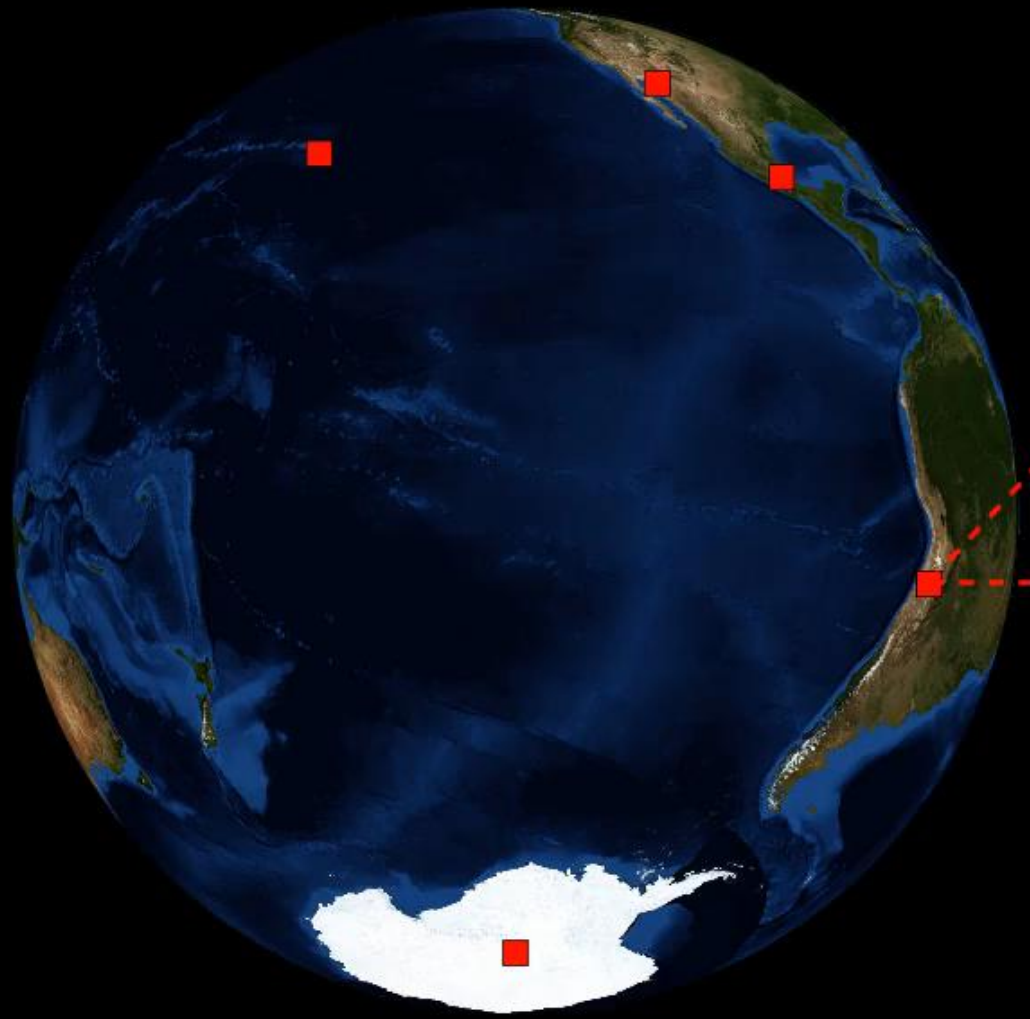
Vorige Woche hatte R.Genzel in Frankfurt einen Vortrag

Wie sieht das schwarze Loch im Zentrum unserer Galaxie aus?

Das EU-Projekt **BlackHoleCam**

L.Rezzolla, H.Falke und M.Kramer

Event Horizon Telescope



Atacama Large
Millimeter Array (ALMA)

Coordinates: $23^\circ\,01'09''\mathrm{S},\quad 67^\circ\,45'12''\mathrm{W}$

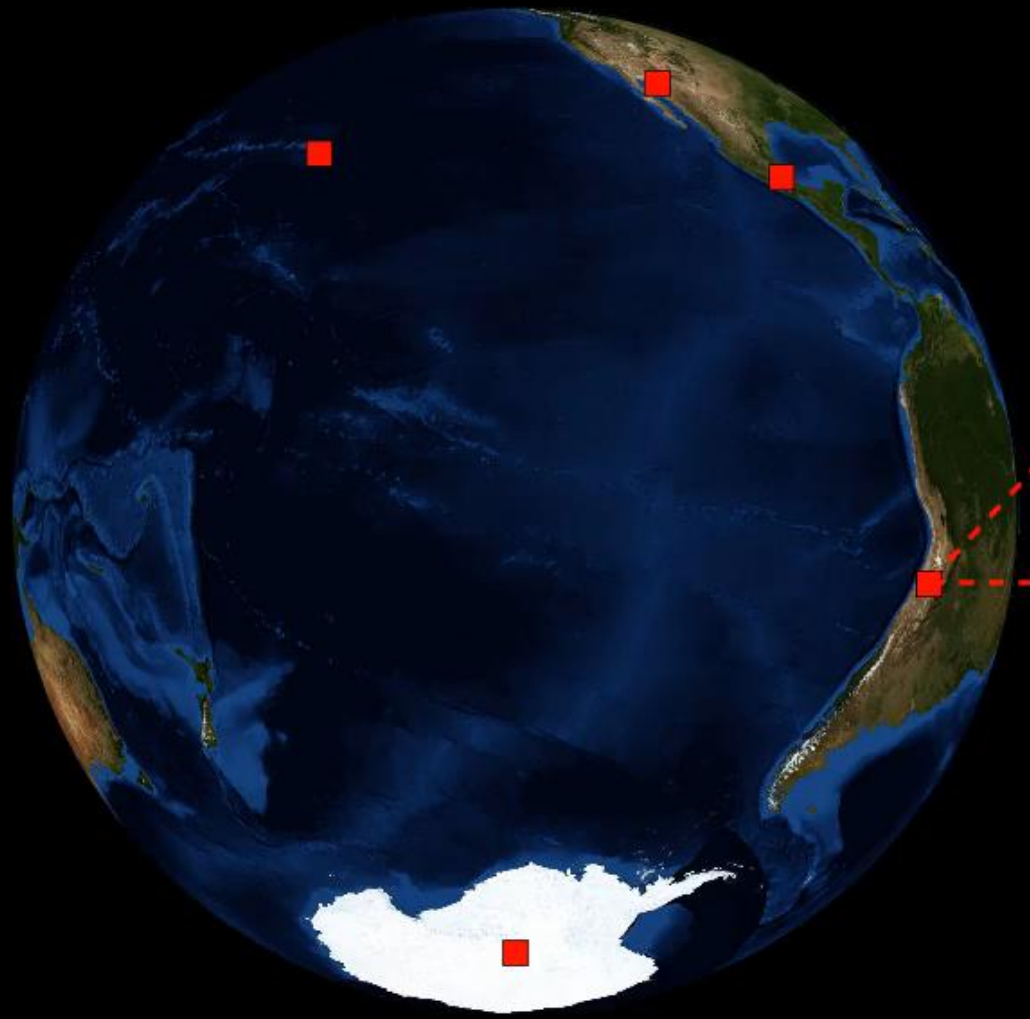
Diameter: 12m

Das EHT ist ein
virtuelles Radioteleskop
der Größe der Erde

Das Event Horizon Teleskop (EHT) ist eine hochgradig internationale Kooperation von Radioteleskopen die mittels Langbasisinterferometrie das erste Bild eines schwarzen Loches aufzeichnen konnte

Python-Animation erstellt von Dr. Christian Fromm

Event Horizon Telescope



Atacama Large
Millimeter Array (ALMA)

Coordinates: $23^\circ\,01'09''\mathrm{S},\quad 67^\circ\,45'12''\mathrm{W}$

Diameter: 12m

Das EHT ist ein
virtuelles Radioteleskop
der Größe der Erde

Das Event Horizon Teleskop (EHT) ist eine hochgradig internationale Kooperation von Radioteleskopen die mittels Langbasisinterferometrie das erste Bild eines schwarzen Loches aufzeichnen konnte

Python-Animation erstellt von Dr. Christian Fromm

Bilder von zwei schwarzen Löchern werden erwartet

(Stand 04.2017)

	M87	Sgr A*
Mass (M_{sun})	3-6 x 10^9 (?)	4 x 10^6
Distance	16 Mpc	8.5 kpc
Luminosity	10^{44} erg/s	10^{36} erg/s
Mdot (M_{edd})	10^{-4}	10^{-8}
BH Spin Axis	Gal disk?	10-25 deg los
@ the BH?	Maybe	Yes
B field @ BH	60-130 G	10-100 G
Scattered?	No	yes
Shadow Size	640 AU	0.5 AU
Shadow Angle	20-40 μas	52 μas
GM/c^3	8 hrs	20 sec
Jet Power	10^{42}-10^{43} erg/s	?

Die ersten Bilder eines Schwarzen Lochs

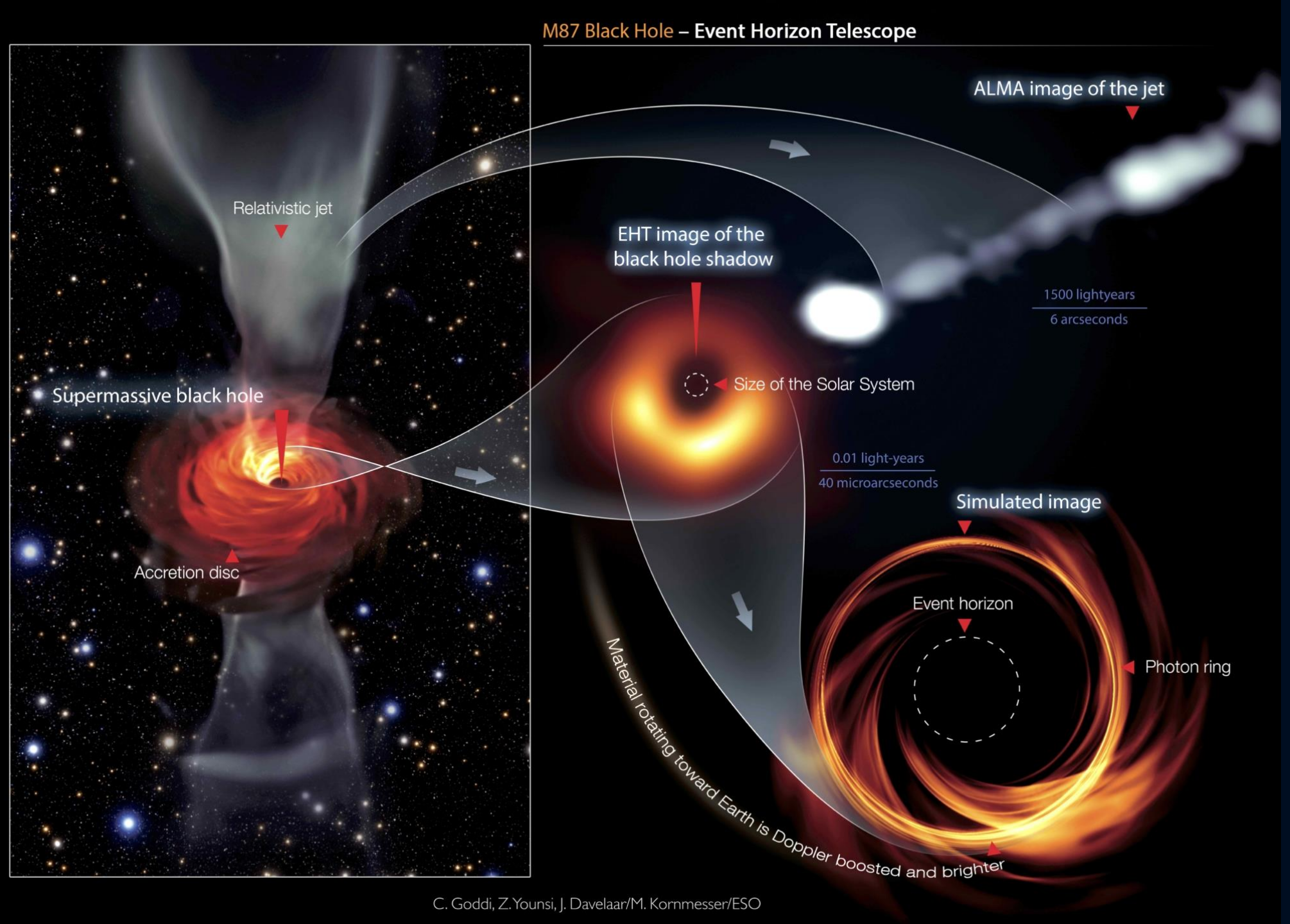
Ein Meilenstein in der Geschichte der Astronomie

Ein wenig mehr als hundert Jahre nachdem Albert Einstein seine Feldgleichungen der *Allgemeinen Relativiätstheorie* formulierte, und er damit die Grundlage für Gravitations-wellen und schwarzer Löcher legte, wurde im Jahre 2019 das erstes Bild eines schwarzen Lochs (siehe rechte Abbildung) der Öffentlichkeit präsentiert.

YouTube Video: https://www.youtube.com/watch?v=Zh5p9Sro_VU&list=PLn5gYfEKIag8nps1GKLqUW35AOgQY7aM2

Anlässlich der bahnbrechenden Aufnahme des ersten Bildes eines schwarzen Lochs im Zentrum unserer Nachbargalaxie M87, wurde am 17. April 2019 um 20 Uhr ein öffentlicher, populärwissenschaftlicher Abendvortrag im Otto Stern Zentrum (OSZ H1) am Campus Riedberg der Goethe Universität gehalten. Es sprachen die drei „Principal Investigators" des europäischen Black Hole Cam-Projekts (L.Rezzolla, M.Kramer und H.Falke), welches neben der EHT-Kolloboration für das Bild verantwortlich ist.

Eine Reise in das Herz von Messier 87


M87 Black Hole – Event Horizon Telescope
ALMA image of the jet
Relativistic jet
EHT image of the black hole shadow
1500 lightyears
6 arcseconds
Supermassive black hole
Size of the Solar System
0.01 light-years
40 microarcseconds
Simulated image
Accretion disc
Event horizon
Photon ring
Material rotating toward Earth is Doppler boosted and brighter
C. Goddi, Z. Younsi, J. Davelaar/M. Kornmesser/ESO

Die ersten Bilder eines Schwarzen Lochs

interaktiven Vorlesung liegt sowohl auf der Allgemeine Relativitätstheorie als auch auf der Vermittlung spezieller Programmierkenntnisse.

Grundlegende Größen der Allgemeinen Relativitätstheorie

Im folgenden werden die Grundlagen der allgemeinen Relativitätstheorie und im besonderen die Einsteingleichung

$$R_{\mu\nu} - \frac{1}{2}\, g_{\mu\nu} R \;=\; -\,8\pi\, T_{\mu\nu}$$

und die Geodatengleichung

$$\frac{d^2 x^\mu}{d\tau^2} + \Gamma^\mu_{\nu\rho}\, \frac{dx^\nu}{d\tau}\, \frac{dx^\rho}{d\tau} \;=\; 0$$

als bekannt vorausgesetzt. Die griechischen, raumzeitlichen Indices $\mu, \nu, \rho \ldots$ laufen von 0..3, wobei, falls nicht anders angegeben, diese den folgenden kartesischen Raumzeitkoordinaten entsprechen: $x^\mu = \left(x^0, x^1, x^2, x^3\right) = (t, x, y, z)$.

Im folgenden werden einige grundlegende Größen der allgemeinen Relativitätstheorie am Beispiel der allgemeinen statischen, isotropen Metrik erläutert und aufgezeigt, wie man diese in Maple berechnet. Zunächst wird das "tensor"-Paket eingebunden. Die mit roter Schrift gekennzeichneten Wörter stellen die vom User eingegebenen Befehle dar und die blauen Wörter sind die vom Maple-Program ausgegebenen Größen. Hier werden im speziellen die im "tensor"-Paket neu definierten Befehle ausgegeben. Möchte man die eingegebenen Befehle zwar ausführen, aber nicht ausgeben lassen, so hat man am Ende des Befehls einen Doppelpunkt und kein Semikolon zu schreiben.

Die Geodätengleichung

Ein System gekoppelter nicht-linearer Differentialgleichungen zweiter Ordnung?

, wobei wir wieder ein sphärisches Koordinatensystem benutzen ($x^\mu = (t, r, \theta, \phi)$). Die Geodätengleichung stellt ein System gekoppelter nichtlinearer Differentialgleichungen dar

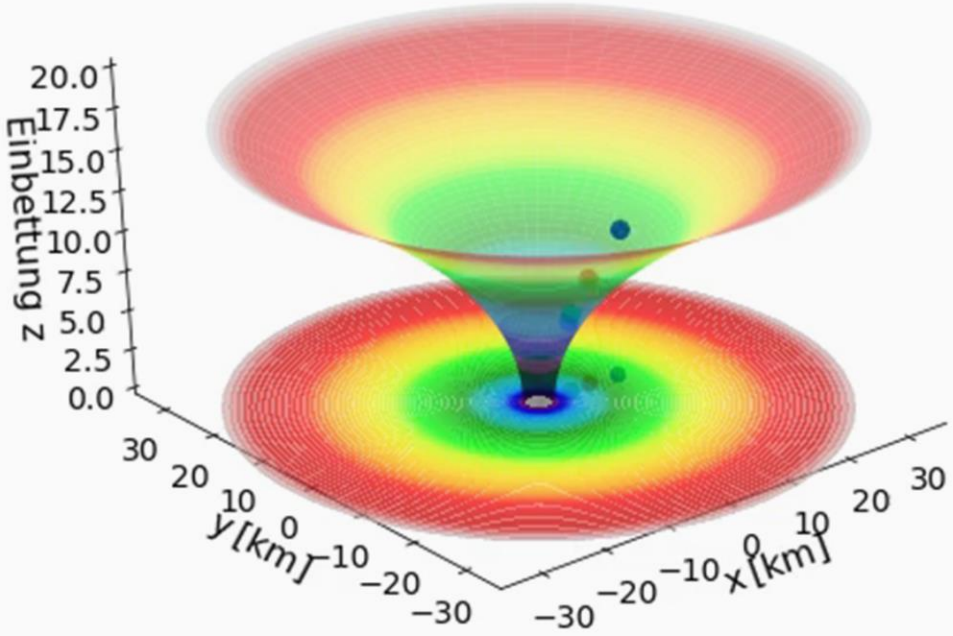
$$\frac{d^2t}{d\lambda^2} = -\Gamma^0_{\nu\rho}\frac{dx^\nu}{d\lambda}\frac{dx^\rho}{d\lambda}$$
$$\frac{d^2r}{d\lambda^2} = -\Gamma^1_{\nu\rho}\frac{dx^\nu}{d\lambda}\frac{dx^\rho}{d\lambda}$$
$$\frac{d^2\theta}{d\lambda^2} = -\Gamma^2_{\nu\rho}\frac{dx^\nu}{d\lambda}\frac{dx^\rho}{d\lambda}$$
$$\frac{d^2\phi}{d\lambda^2} = -\Gamma^3_{\nu\rho}\frac{dx^\nu}{d\lambda}\frac{dx^\rho}{d\lambda} \quad ,$$

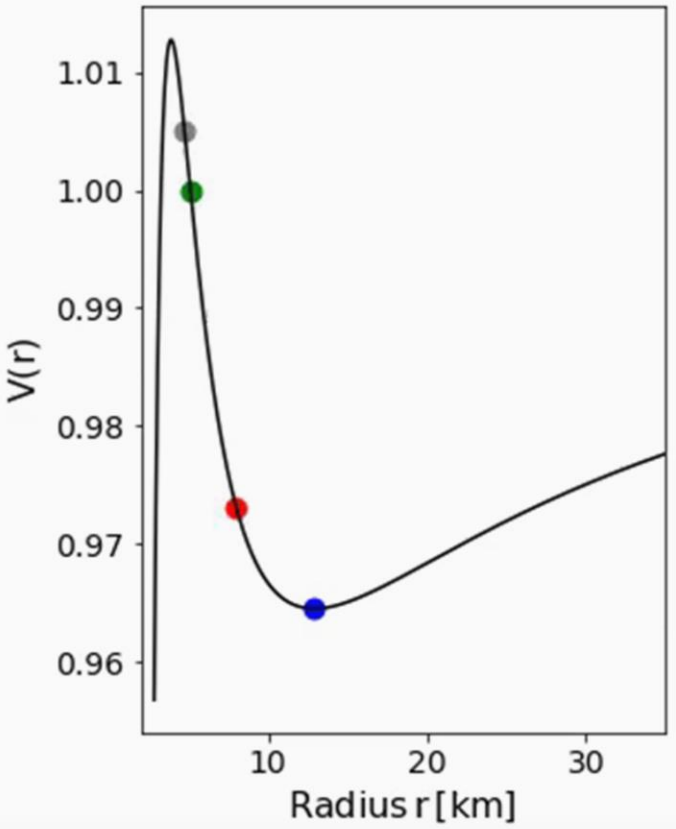
wobei λ ein affiner Parameter (z.B. die Eigenzeit τ), t, r, θ und ϕ die Koordinaten und $\Gamma^\mu_{\nu\rho}$ die Christoffel Symbole zweiter Art darstellen.

Klassifizierung der möglichen Bahnbewegungen eines Probekörpers um ein nicht-rotierendes schwarzes Loch

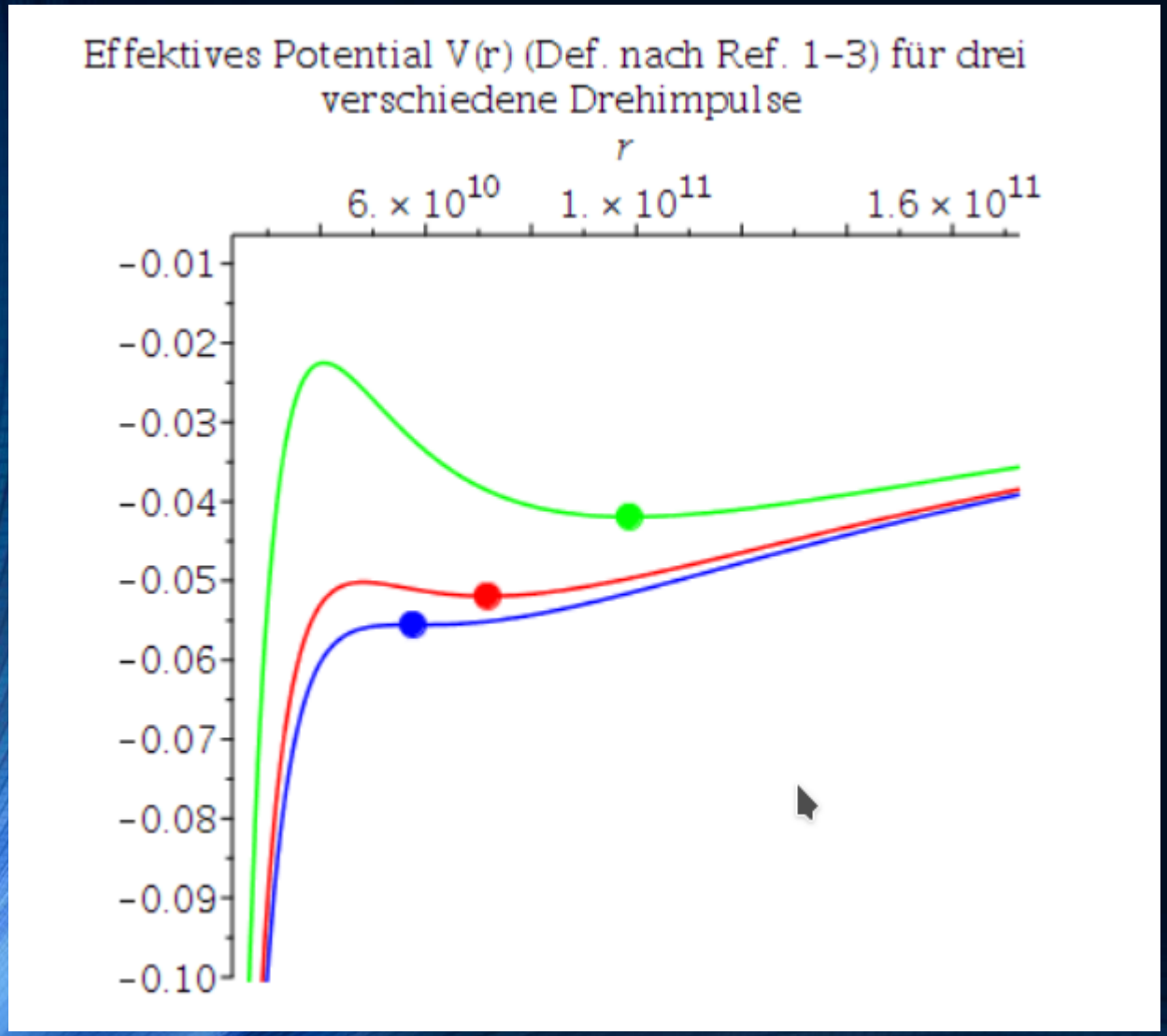
Neben den gebundenen kreisförmigen (blau) und elliptischen (rot) Bahnen, den parabolischen (grün) und hyperbolischen (grau) Bahnverläufen ist auch eine durch das schwarze Loch eingefangene Bahn (schwarz: capture orbit) möglich

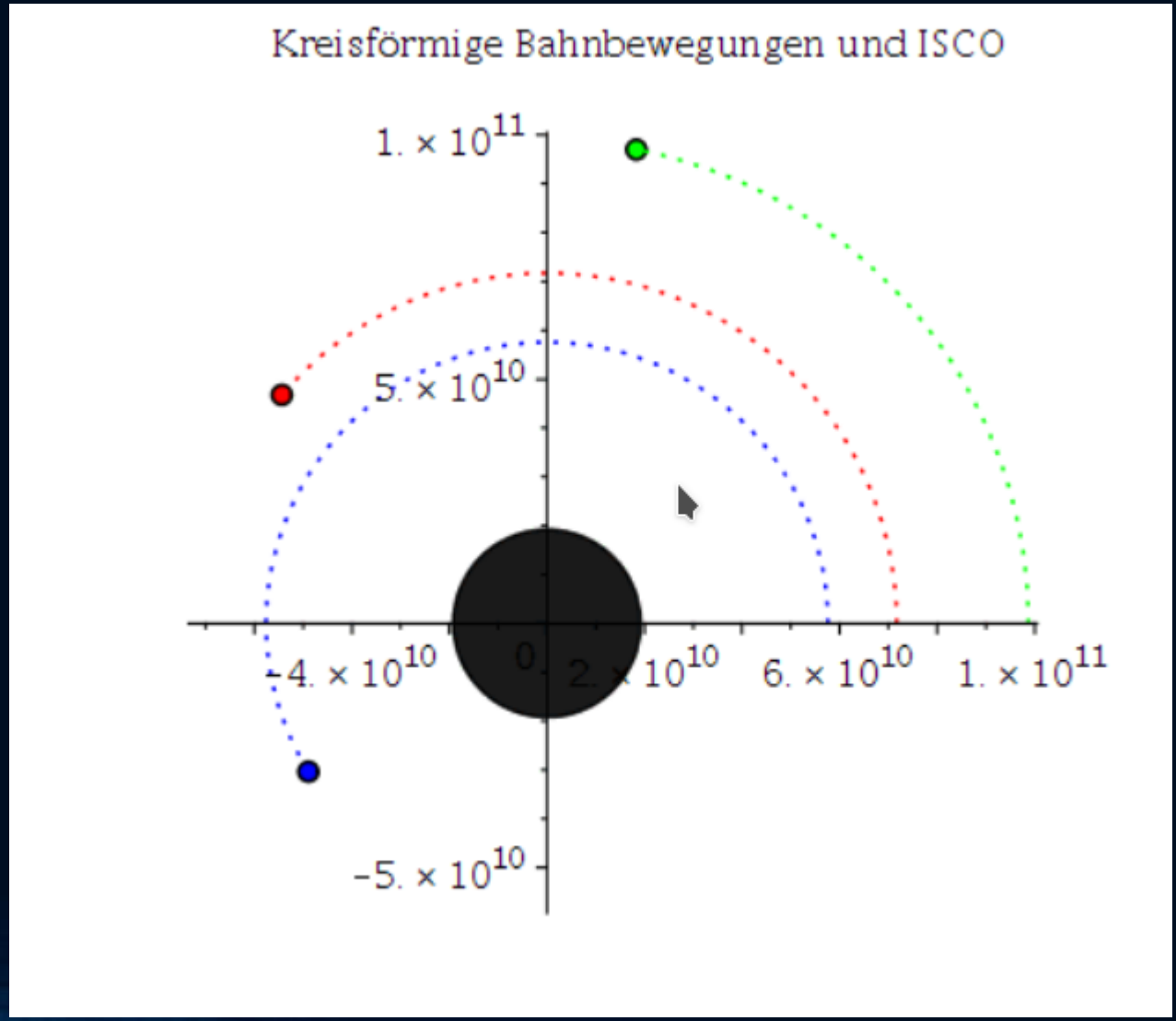
Animation wurde im Python Jupyter Notebook *„Klassifizierung unterschiedlicher Bahnbewegungen um ein nicht-rotierendes Schwarzes Loch“* erstellt





Das effektive Potential eines Probekörpers hat am ISCO (*Innermost stable circular Orbit*) hat eine Sattelpunktseigenschaft

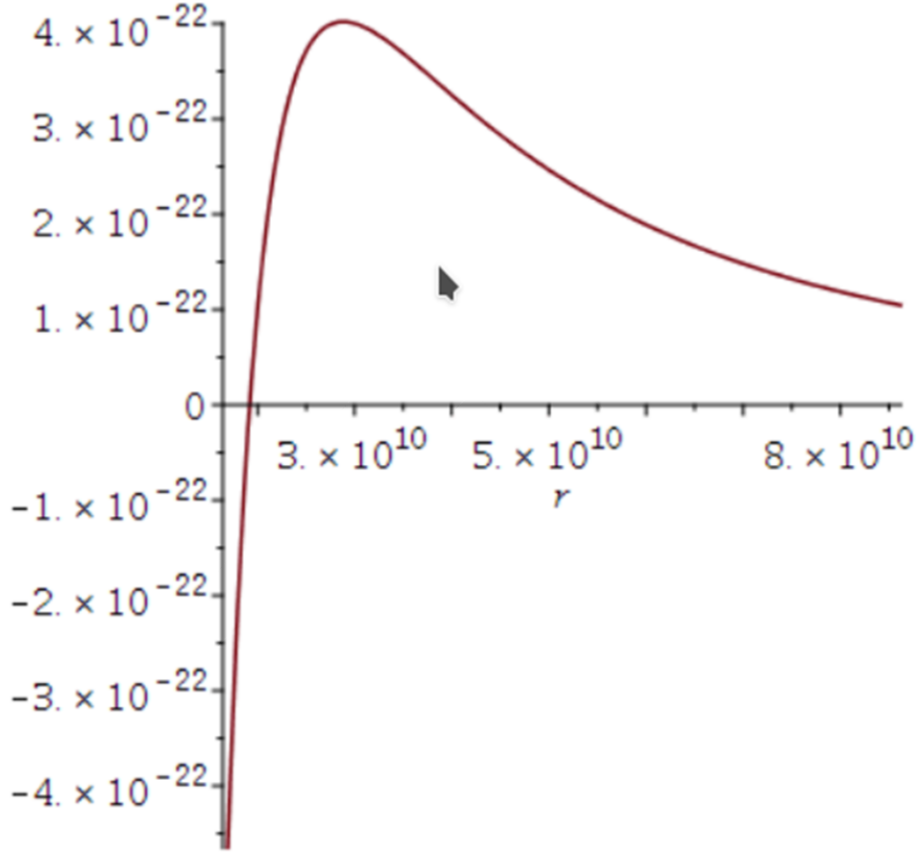


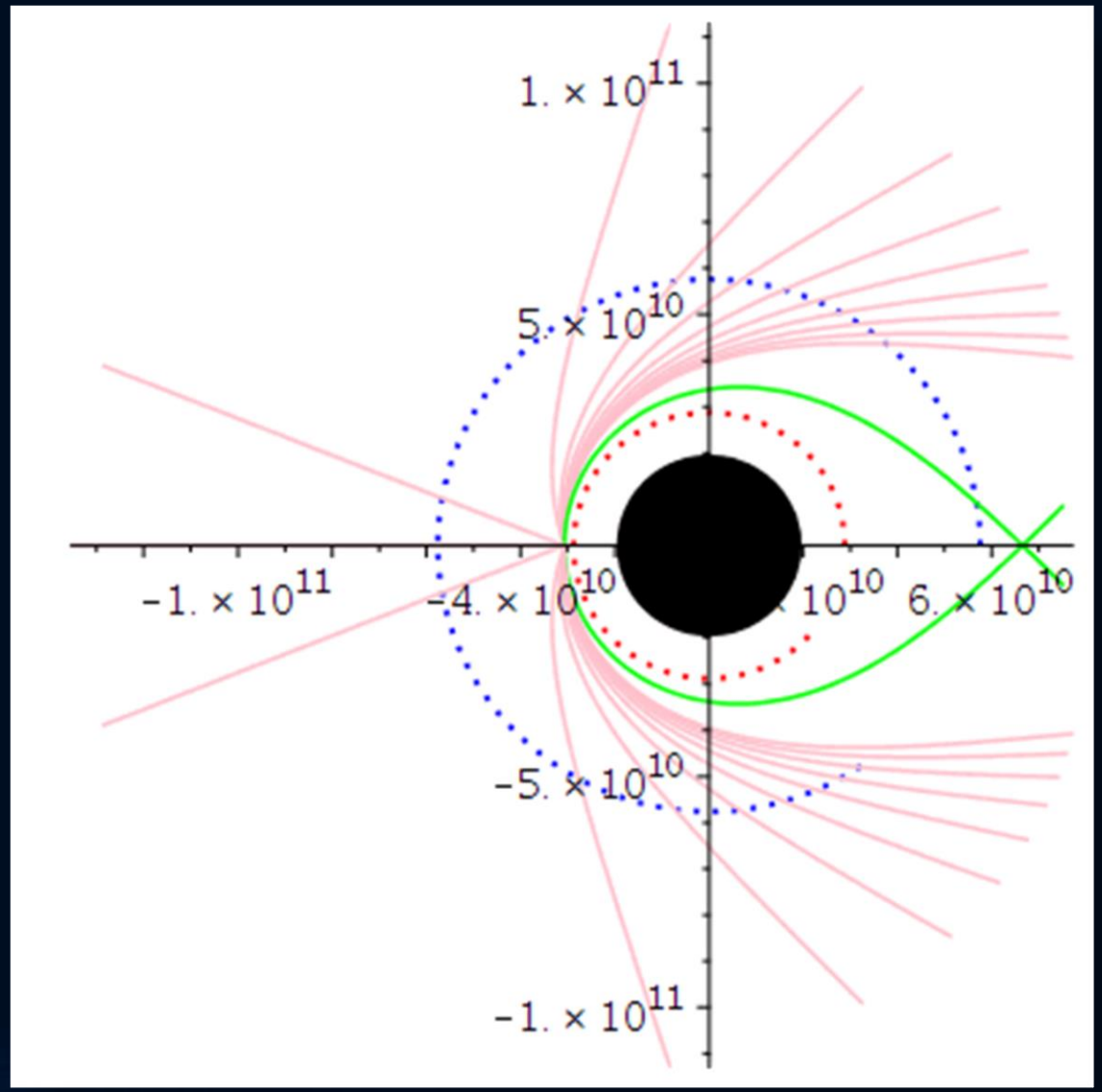


Masselose Teilchen (Photonen):
Das effektive Potential und die Photonensphäre bei 3M

$$VeffPhotonHobson := (r, M) \mapsto \frac{1 - \frac{2M}{r}}{r^2}$$

Effektives Potential V(r) (Def. nach Ref. 1-3) für Photonen

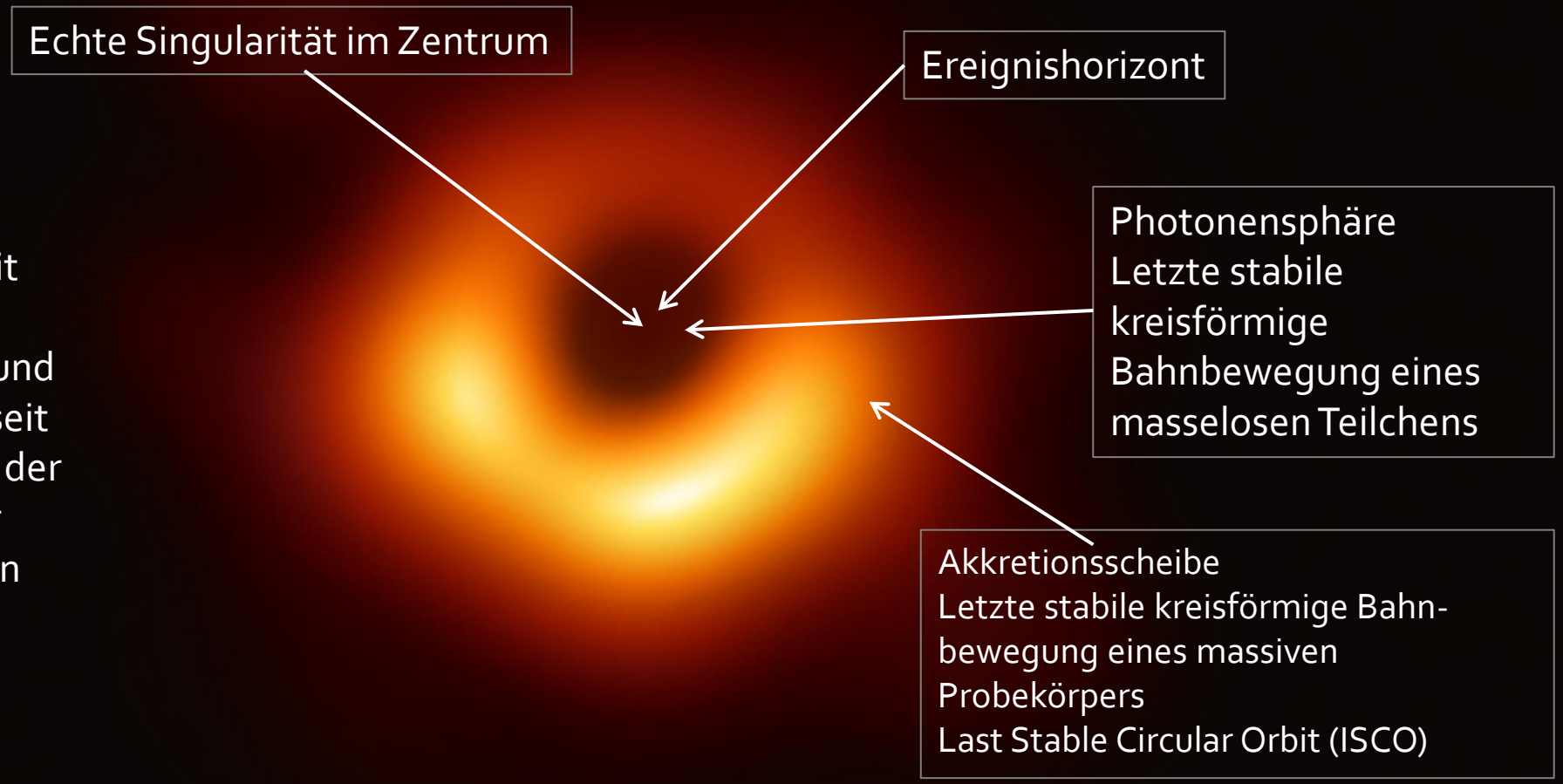




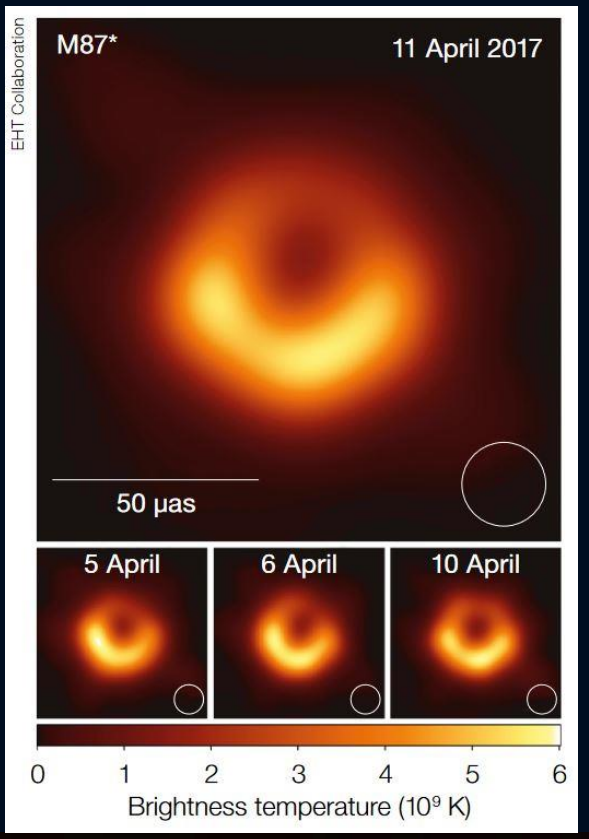
Blau: ISCO , Rot: Photonensphäre

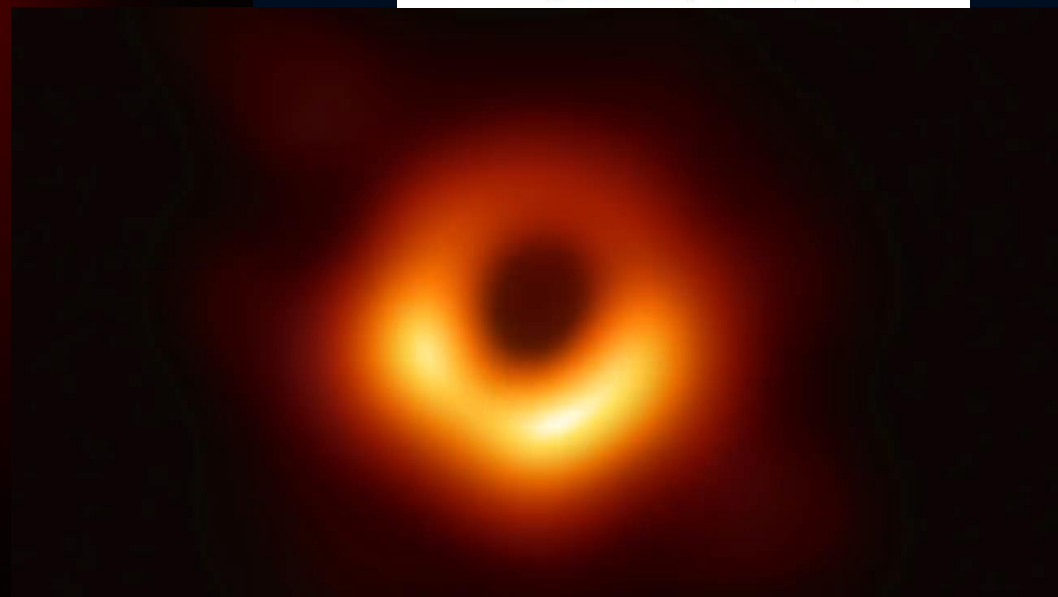
das Bild des schwarzen Lochs

Ein wenig mehr als hundert Jahre nachdem Albert Einstein seine Feldgleichungen der *Allgemeine Relativiätstheorie* der Öffentlichkeit präsentierte, und er damit die Grundlage für Gravitationswellen und schwarzer Löcher formulierte, ist seit einigen Wochen ein Meilenstein in der Geschichte der Astronomie in aller Munde (erstes Bild eines schwarzen Lochs, siehe rechte Abbildung).



Aktuelle und zukünftige Bilder



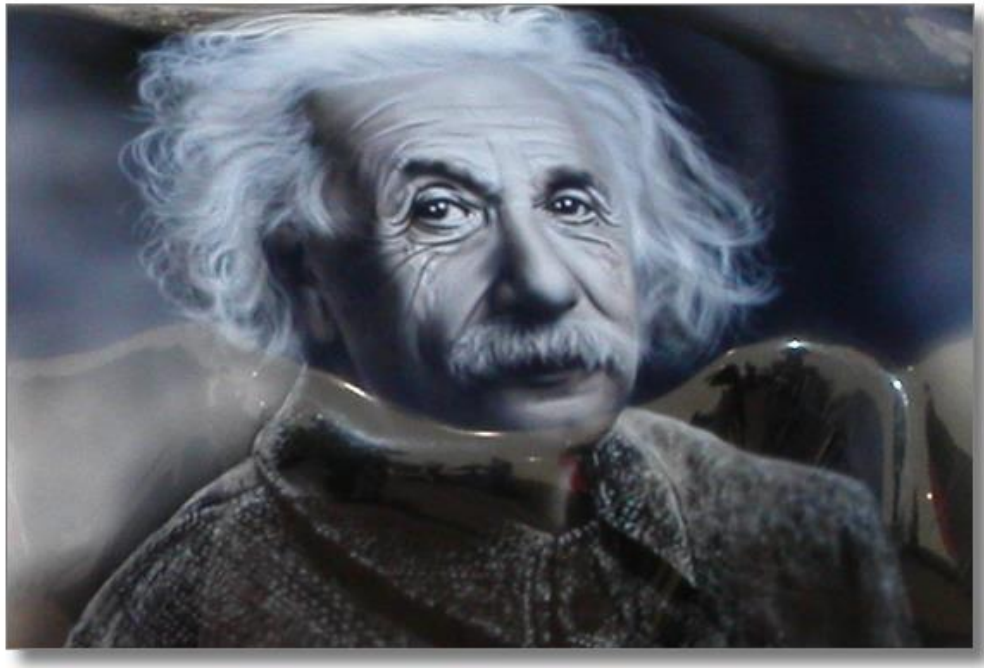


Vorlesung: Allgemeine Relativitätstheorie mit dem Computer

ART mit dem Computer (Online) *von* Dr.phil.nat.Dr.rer.pol. Matthias Hanauske

Nächster Zoom Link am 16.04.2021, 15.00-17.00 Uhr:
ID: 794 847 5614, PWD: 785453

Vorwort



Die Vorlesung *Allgemeine Relativitätstheorie mit dem Computer* wurde im Sommersemester 2016 das erste Mal gehalten und viele der auf dieser Hauptseite erreichbaren Internetseiten basieren grundsätzlich auf dem damals erstellten Kurs. In der Vorlesung werden die mathematisch anspruchsvollen Gleichungen der Allgemeinen Relativitätstheorie (ART) in diversen Programmierumgebungen analysiert. Im ersten Teil des Kurses erlernen die Studierenden die Verwendung von Computeralgebra-Systemen (Python Jupyter Notebooks, Maple und Mathematica). Die oft komplizierten und zeitaufwendigen Berechnungen der tensoriellen Gleichungen der ART können mithilfe dieser Programme erleichtert werden. Diverse Anwendungen der Einstein- und [illegible] interaktive Programmierung [illegible] geführt und

Allgemeine Relativitätstheorie mit dem Computer
(General Theory of Relativity on the Computer)
Vorlesung SS 2021

Aufgrund der andauernden Corona-Krise findet die Vorlesung und die Übungstermine auch in diesem Semester nur Online statt!

Diese Internetseite fasst die Online-Angebote der Vorlesung *Allgemeine Relativitätstheorie mit dem Computer* zusammen. Auf der linken Seite finden Sie die einzelnen Vorlesungsaufzeichnungen (Videos), Vorlesungspräsentationen (pdf-Dateien) und weiterführende Links. Die Vorlesungstermine (Zoom Meetings, synchrones Lehrangebot) finden jeweils freitags von 15.00-17.00 Uhr statt. Die Termine der Online-Übungen (ca. 1.5 Stunden pro Woche) werden in der ersten Vorlesungseinheit gemeinsam festgelegt. Alle Lehrangebote werden mittel der Zoom Meeting Software gemacht und die jeweiligen Zoom-Links sind in der rechten oberen Ecke dieser Internetseite angegeben. Die Inhalte der Vorlesung gliedern sich in drei Teile (Teil I, Teil II, Teil III), die Sie in der zweiten oberen Spalte einsehen können. Weiteres Zusatzmaterial und diverse Online-Aufgaben sind über die Online-Lernplattformen OLAT und Lon Capa erhältlich (siehe *E-Learning*). Der Schwerpunkt der gesamten interaktiven Vorlesung liegt sowohl auf der Allgemeine Relativitätstheorie als auch auf der Vermittlung

http://itp.uni-frankfurt/~hanauske/VARTC/

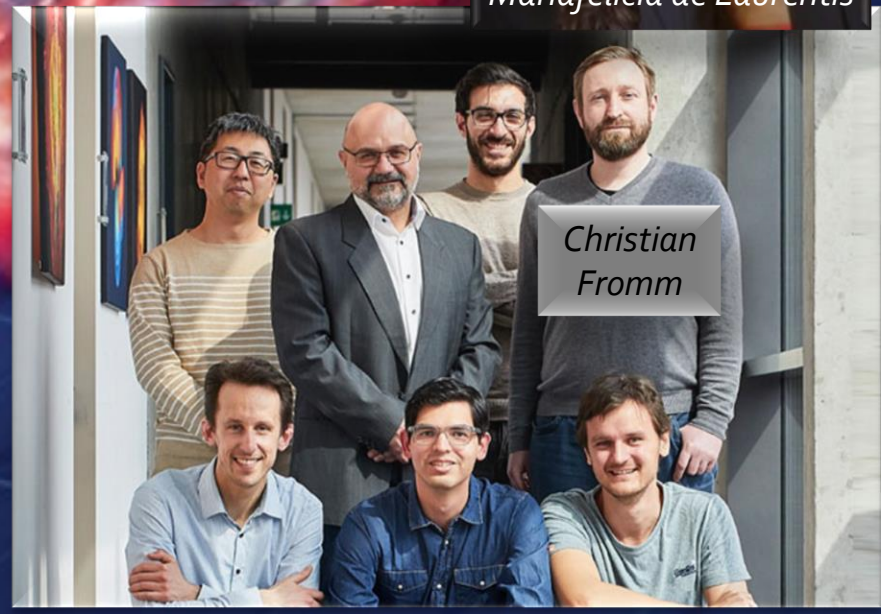
Pressekonferenz am 12. Mai 2022, 15.00 Uhr

Neue Erkenntnisse über das Schwarze Loch in unserer Milchstraße

Mariafelicia de Laurentis

Die Europäische Südsternwarte (ESO) und das Event Horizon Telescope (EHT) Projekt halten eine Pressekonferenz ab, bei der neue Ergebnisse vom EHT zur Milchstraße präsentiert werden.

- **Wann:** Am 12. Mai um 15:00 Uhr MESZ
- **Wo:** Eridanus-Auditorium, ESO-Hauptsitz, Garching bei München, und online
- **Was:** Eine Pressekonferenz, auf der bahnbrechende Ergebnisse des EHT zur Milchstraße präsentiert werden
- **Wer:** Der ESO-Generaldirektor wird die einleitenden Worte sprechen. EHT-Projektleiter Huib Jan van Langevelde und Anton Zensus, Gründungs-Vorsitzender der EHT-Kollaboration, werden ebenfalls sprechen. Eine Runde von EHT-Forschenden werden die Ergebnisse erläutern und Fragen beantworten. Diese Runde besteht aus
  - Thomas Krichbaum, Max-Planck-Institut für Radioastronomie, Deutschland
  - Sara Issaoun, Center for Astrophysics | Harvard & Smithsonian, US und Radboud University, Niederlande
  - José L. Gómez, Instituto de Astrofísica de Andalucía (CSIC), Spanien
  - Christian Fromm, Universität Würzburg, Deutschland
  - Mariafelicia de Laurentis, University of Naples "Federico II" und the National Institute for Nuclear Physics (INFN), Italien



Christian Fromm

Anwendungsbeispiel
Interpolation und Polynomapproximation

Anwendungsbeispiel: Interpolation und Polynomapproximation

Die Interpolation ist ein Teilgebiet der numerischen Mathematik und es befasst sich mit der Problematik, eine analytische Funktion mittels einer gegebenen Menge von Werten zu bestimmen. Wir werden in diesem Unterpunkt den wohl elementarsten Algorithmus einer Interpolation vorstellen, bei dem ein Polynom $P(x)$ mittels einer gegebenen Liste von Werten konstruiert wird. Aufgrund des *Weierstraßschen Approximationssatzes* gibt es für jede, auf einem Teilintervall $[a, b]$ definierte, stetige Funktion, $f(x)$ ein Polynom $P(x)$ mit der Eigenschaft, dass für jedes $\epsilon > 0$ die Differenz der Funktion mit dem Polynom verschwindend klein wird ($|f(x) - P(x)| < \epsilon \; \forall \, x \in [a, b]$). Die Taylorschen Polynome sind hierbei wohl das bekannteste Beispiel einer Entwicklung einer Funktion in ein Polynom. Leider haben die Taylorschen Polynome den Nachteil, dass sie von ihrer Konstruktion her nur eine Stützstelle $x_0 \in [a, b]$ verwenden und somit die Funktion $f(x)$ zwar gut in dem Bereich um die Stützstelle beschreiben, aber nicht im gesamten Teilintervall $[a, b]$. Bei einer Interpolation mittels der *Methode der Lagrange Polynome* verwendet man hingegen mehrere Stützstellenpunkte für die Konstruktion des approximierenden Polynoms. Dieses Verfahren ist Gegenstand dieses Anwendungsbeispiels.

Die Theorie der Entwicklung einer Funktion in ein Lagrange Polynom

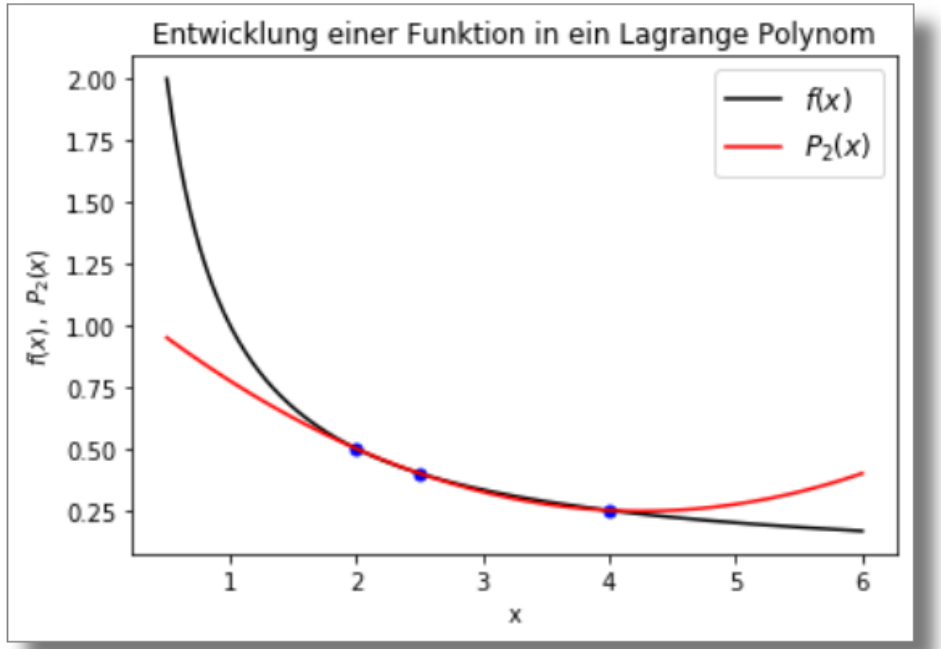
Bei der Entwicklung einer Funktion in ein Lagrange Polynom steht man vor der Aufgabe eine Funktion $f(x)$ mittels $(n + 1)$ vorgegebener Punkte $((x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n)),)$ in ein Lagrange Polynom vom Grade n zu entwickeln. Das n-te Lagrange Polynom $P_n(x)$ einer Funktion $f(x)$ ist wie folgt definiert:

$$P_n(x) = \sum_{k=0}^{n} f(x_k) \cdot L_{n,k}(x) \quad \text{, wobei} \quad L_{n,k}(x) = \prod_{i=0,\, i \neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)} \quad .$$

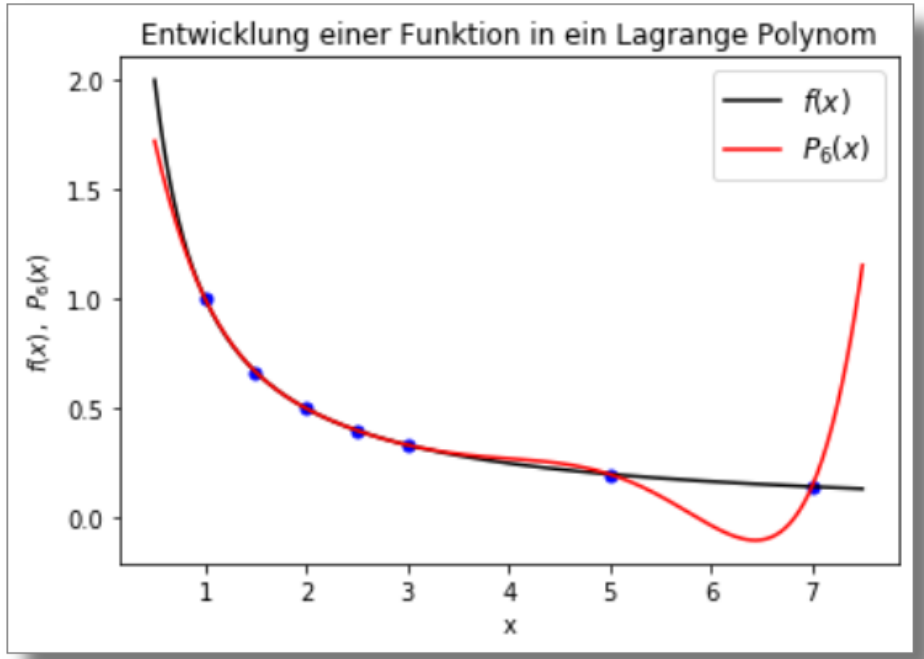
Lagrange Polynome

In diesem Unterpunkt werden wir die Interpolation einer Funktion f(x) mittels der *Methode der Lagrange Polynome* vorstellen, bei der man mehrere Stützstellenpunkte für die Konstruktion des approximierenden Polynoms verwendet.

Im Gegensatz zu Taylorschen Polynomen, benutzt die Methode der Lagrange Polynome mehrere unterschiedliche Punkte der Funktion bei der Entwicklung der Funktion in ein Polynom. Taylorschen Polynome haben somit den Nachteil, dass sie von ihrer Konstruktion her nur eine Stützstelle $x_0 \in [a,b]$ verwenden und somit die Funktion f(x) zwar gut in dem Bereich um die Stützstelle beschreiben, aber nicht im gesamten Teilintervall [a,b].



Die nebenstehende linke Abbildung verdeutlicht die oben beschriebene Lagrange Polynom Entwicklung, wobei die blauen Punkte die drei vorgegebenen Stützstellen des Polynoms darstellen, bei welchen die Übereinstimmung $f(x) = P_2(x)$ exakt gilt. Die rechte Abbildung zeigt das entsprechende Polynom $P_6(x)$, wobei hierbei die sieben Stützstellen an den folgenden x-Werten genommen wurden: $\vec{x} = (1, 1.5, 2, 2.5, 3, 5, 7)$.



$$P_n(x) = \sum_{k=0}^{n} f(x_k) \cdot L_{n,k}(x) \quad \text{, wobei} \quad L_{n,k}(x) = \prod_{i=0,\, i\neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)} \quad .$$

Approximiert man z.B. die Funktion $f(x) = \frac{1}{x}$ im Teilintervall $[a, b] = [0.5, 6]$ mittels dreier vorgegebener Punkte ($n = 2$, mit $n + 1$ Punkten $(2, f(2)), (2.5, f(2.5))$ und $(4, f(4))$) in ein Lagrange Polynom vom Grade 2, so erhält man:

$$P_2(x) = \sum_{k=0}^{2} f(x_k) \cdot L_{2,k}(x) = f(x_0) \cdot L_{2,0}(x) + f(x_1) \cdot L_{2,1}(x) + f(x_2) \cdot L_{2,2}(x)$$

$$= \frac{1}{2} \cdot L_{2,0}(x) + \frac{1}{2.5} \cdot L_{2,1}(x) + \frac{1}{4} \cdot L_{2,2}(x)$$

mit:

$$L_{2,0}(x) = \prod_{i=0,\, i\neq 0}^{2} \frac{(x - x_i)}{(x_0 - x_i)} = \frac{(x - x_1)}{(x_0 - x_1)} \cdot \frac{(x - x_2)}{(x_0 - x_2)} = \frac{(x - 2.5)}{(2 - 2.5)} \cdot \frac{(x - 4)}{(2 - 4)}$$

$$L_{2,1}(x) = \prod_{i=0,\, i\neq 1}^{2} \frac{(x - x_i)}{(x_1 - x_i)} = \frac{(x - x_0)}{(x_1 - x_0)} \cdot \frac{(x - x_2)}{(x_1 - x_2)} = \dots$$

$$L_{2,2}(x) = \prod_{i=0,\, i\neq 2}^{2} \frac{(x - x_i)}{(x_2 - x_i)} = \frac{(x - x_0)}{(x_2 - x_0)} \cdot \frac{(x - x_1)}{(x_2 - x_1)} = \dots$$

Beispiel:
Lagrange Polynom vom Grade n=2
3 Stützpunkte

$$\Longrightarrow P_2(x) = 0.05\, x^2 - 0.425\, x + 1.15$$

Fehlerformel der Lagrange Polynom Methode

Die Abbildungen zeigen, dass die Übereinstimmung des berechneten Polynoms $P_n(x)$ mit der Funktion $f(x)$ nur an den Stützstellen exakt ist. Betrachtet man sich den Unterschied beider Funktionen an einem beliebigen x-Wert im Bereich der Stützstellen, so kann man die folgende Fehlerformel des Lagrangeschen Polynoms herleiten

$$f(x) = P_n(x) + \underbrace{\frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i)}_{\text{Fehler}} \ , \ \forall\ \xi(x) \in [x_0, x_n] \quad ,$$

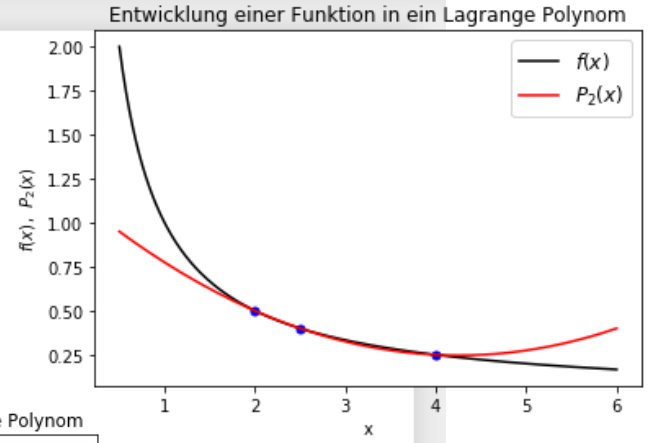
wobei $f^{(n+1)(x)}$ die $(n+1)$-te Ableitung der Funktion darstellt und $\xi(x)$ ein x-Wert im Bereich der Stützstellen ist.

```cpp
/* Entwicklung einer Funktion in ein Lagrange Polynom
 * Mittels der Methode der Lagrange Polynome entwickelt man eine Funktion ( hier speziell f(x)=1/x )
 * durch Angabe von n+1 vorgegebener Punkte in ein Lagrange Polynom vom Grade n.
 * Hier speziell 3 Punkte ( (2,f(2)), (2.5,f(2.5)), (4,f(4)) )
 * Ausgabe zum Plotten (Gnuplot oder Python) mittels: "./a.out > Lagrange_Polynom_1.dat" */
#include <iostream>                                                 // Ein- und Ausgabebibliothek

double f(double x){                                                 // Deklaration und Definition der Funktion f(x) die approximiert werden soll
    double wert;
    wert = 1.0/x;                                                   // Eigentliche Definition der Funktion
    return wert;                                                    // Rueckgabewert der Funktion f(x)
}                                                                   // Ende der Funktion f(x)

int main(){                                                         // Hauptfunktion
    double points[] = { 2, 2.5, 4 };                                // Deklaration und Initialisierung der Stützstellen als double-Array
    const unsigned int N_points = sizeof(points)/sizeof(points[0]); // Anzahl der Punkte die zur Approximation verwendet werden
    double plot_a=0.5;                                              // Untergrenze des x-Intervalls in dem die Ergebnisse ausgegeben werden sollen
    double plot_b=6;                                                // Obergrenze des x-Intervalls in dem die Ergebnisse ausgegeben werden sollen
    const unsigned int N_xp=300;                                    // Anzahl der Punkte in die das x-Intervall aufgeteilt wird
    double dx = (plot_b - plot_a)/N_xp;                             // Abstand dx zwischen den aequidistanten Punkten des x-Intervalls
    double x = plot_a-dx;                                           // Aktueller x-Wert
    double xp[N_xp+1];                                              // Deklaration der x-Ausgabe-Punkte als double-Array
    double fp[N_xp+1];                                              // Deklaration der f(x)-Ausgabe-Punkte als double-Array
    double Pfp[N_xp+1];                                             // Deklaration der Ausgabe-Punkte des approximierten Polynoms als double-Array
    double Lk;                                                      // Deklaration einer Variable, die fuer Zwischenrechnungen benoetigt wird

    printf("# x-Werte der %3d Stuetzstellen-Punkte: \n", N_points); // Beschreibung der ausgegebenen Groessen
    for(int k = 0; k < N_points; ++k){                              // For-Schleife der Ausgabe der Stuetzstellen x-Werte
        printf("%10.5f",points[k]);                                 // Ausgabe der Stuetzpunkte
    }                                                               // Ende for-Schleife der Ausgabe
    printf("\n");                                                   // Zeilenumbruch

    printf("# 0: Index j \n# 1: x-Wert \n# 2: f(x)-Wert \n");       // Beschreibung der ausgegebenen Groessen
    printf("# 3: Approximierter Wert des Lagrange Polynoms P(x) \n"); // Beschreibung der ausgegebenen Groessen
    printf("# 4: Fehler zum wirklichen Wert f(x)-P(x) \n");         // Beschreibung der ausgegebenen Groessen

    for(int j = 0; j <= N_xp; ++j){                                 // For-Schleife die ueber die einzelnen Punkte des x-Intervalls geht
        x=x+dx;                                                     // Aktueller x-Wert
        xp[j]=x;                                                    // Eintrag des aktuellen x-Wertes in das x-Array
        fp[j]=f(x);                                                 // Eintrag des aktuellen f(x)-Wertes in das f(x)-Array

        Pfp[j]=0;                                                   // Initialisierung des j-ten Polynom-Arrays mit 0 (Wert beim Anfang der Summenbildung)
        for(int k = 0; k < N_points; ++k){                          // For-Schleife der Summation in der Lagrange Polynom Methode
            Lk=1;                                                   // Initialisierung der Produktvariable Lk mit 1
            for(int i = 0; i < N_points; ++i){                      // For-Schleife der Produktbildung in der Lagrange Polynom Methode
                if(i != k){                                         // Die Produktbildung soll nur fuer (i ungleich k) erfolgen
                    Lk = Lk * (x - points[i])/(points[k] - points[i]); // Berechnung der Lk-Werte in der Lagrange Polynom Methode
                }                                                   // Ende if-Bedingung
            }                                                       // Ende for-Schleife der Produktbildung
            Pfp[j] = Pfp[j] + f(points[k])*Lk;                      // Kern-Gleichung in der Lagrange Polynom Methode
        }                                                           // Ende for-Schleife der Summenbildung
    }                                                               // Ende der for-Schleife ueber die einzelnen Punkte des x-Intervalls

    for(int j = 0; j <= N_xp; ++j){                                 // For-Schleife der separaten Ausgabe der berechneten Werte
        printf("%3d %14.10f %14.10f %14.10f %14.10f \n",j, xp[j], fp[j], Pfp[j], (fp[j] - Pfp[j])); // Ausgabe der berechneten Werte
    }                                                               // Ende for-Schleife der Ausgabe
}                                                                   // Ende der Hauptfunktion
```

```cpp
/* Entwicklung einer Funktion in ein Lagrange Polynom
 * Mittels der Methode der Lagrange Polynome entwickelt man eine Funktion ( hier speziell f(x)=1/x )
 * durch Angabe von n+1 vorgegebener Punkte in ein Lagrange Polynom vom Grade n.
 * Hier speziell 3 Punkte ( (2,f(2)), (2.5,f(2.5)), (4,f(4)) )
 * Ausgabe zum Plotten (Gnuplot oder Python) mittels: "./a.out > Lagrange_Polynom_1.dat" */
#include <iostream>                                                 // Ein- und Ausgabebibliothek

double f(double x){                                                 // Deklaration und Definition der
    double wert;
    wert = 1.0/x;                                                   // Eigentliche Definition der Funk
    return wert;                                                    // Rueckgabewert der Funktion f(x)
}                                                                   // Ende der Funktion f(x)

int main(){                                                         // Hauptfunktion
    double points[] = { 2, 2.5, 4 };                                // Deklaration und Initialisierung
    const unsigned int N_points = sizeof(points)/sizeof(points[0]); // Anzahl der Punkte die zur Appro
    double plot_a=0.5;                                              // Untergrenze des x-Intervalls in
    double plot_b=6;                                                // Obergrenze des x-Intervalls in
    const unsigned int N_xp=300;                                    // Anzahl der Punkte in die das x-
    double dx = (plot_b - plot_a)/N_xp;                             // Abstand dx zwischen den aequidi
    double x = plot_a-dx;                                           // Aktueller x-Wert
    double xp[N_xp+1];                                              // Deklaration der x-Ausgabe-Punkt
    double fp[N_xp+1];                                              // Deklaration der f(x)-Ausgabe-Pu
    double Pfp[N_xp+1];                                             // Deklaration der Ausgabe-Punkte
    double Lk;                                                      // Deklaration einer Variable, die

    printf("# x-Werte der %3d Stuetzstellen-Punkte: \n", N_points); // Beschreibung der ausgegebenen G
    for(int k = 0; k < N_points; ++k){                              // For-Schleife der Ausgabe der St
        printf("%10.5f",points[k]);                                 // Ausgabe der Stuetzpunkte
    }                                                               // Ende for-Schleife der Ausgabe
    printf("\n");                                                   // Zeilenumbruch

    printf("# 0: Index j \n# 1: x-Wert \n# 2: f(x)-Wert \n");       // Beschreibung der ausgegebenen G
    printf("# 3: Approximierter Wert des Lagrange Polynoms P(x) \n"); // Beschreibung der ausgegebenen G
    printf("# 4: Fehler zum wirklichen Wert f(x)-P(x) \n");         // Beschreibung der ausgegebenen Groessen

    for(int j = 0; j <= N_xp; ++j){                                 // For-Schleife die ueber die ei
        x=x+dx;                                                     // Aktueller x-Wert
        xp[j]=x;                                                    // Eintrag des aktuellen x-Wertes
        fp[j]=f(x);                                                 // Eintrag des aktuellen f(x)-We

        Pfp[j]=0;                                                   // Initialisierung des j-ten Poly
        for(int k = 0; k < N_points; ++k){                          // For-Schleife der Summation in
            Lk=1;                                                   // Initialisierung der Produktva
            for(int i = 0; i < N_points; ++i){                      // For-Schleife der Produktbildun
                if(i != k){                                         // Die Produktbildung soll nur f
                    Lk = Lk * (x - points[i])/(points[k] - points[i]); // Berechnung der Lk-Werte in de
                }                                                   // Ende if-Bedingung
            }                                                       // Ende for-Schleife der Produkt
            Pfp[j] = Pfp[j] + f(points[k])*Lk;                      // Kern-Gleichung in der Lagrange
        }                                                           // Ende for-Schleife der Summenbi
    }                                                               // Ende der for-Schleife ueber di

    for(int j = 0; j <= N_xp; ++j){                                 //
        printf("%3d %14.10f %14.10f %14.10f %14.10f \n",j, xp[j], fp[j], Pfp[j], (fp[j] - Pfp[j])); //
    }                                                               // Ende for-Schleife der Ausgabe
}                                                                   // Ende der Hauptfunktion
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V4$ ./a.out
# x-Werte der   3 Stuetzstellen-Punkte:
   2.00000   2.50000   4.00000
# 0: Index j
# 1: x-Wert
# 2: f(x)-Wert
# 3: Approximierter Wert des Lagrange Polynoms P(x)
# 4: Fehler zum wirklichen Wert f(x)-P(x)
  0   0.5000000000   2.0000000000   0.9500000000   1.0500000000
  1   0.5183333333   1.9292604502   0.9431418056   0.9861186446
  2   0.5366666667   1.8633540373   0.9363172222   0.9270368150
  3   0.5550000000   1.8018018018   0.9295262500   0.8722755518
  4   0.5733333333   1.7441860465   0.9227688889   0.8214171576
  5   0.5916666667   1.6901408451   0.9160451389   0.7740957062
  6   0.6100000000   1.6393442623   0.9093550000   0.7299892623
  7   0.6283333333   1.5915119363   0.9026984722   0.6888134641
  8   0.6466666667   1.5463917526   0.8960755556   0.6503161970
  9   0.6650000000   1.5037593985   0.8894862500   0.6142731485
 10   0.6833333333   1.4634146341   0.8829305556   0.5804840786
 11   0.7016666667   1.4251781473   0.8764084722   0.5487696750
 12   0.7200000000   1.3888888889   0.8699200000   0.5189688889
 13   0.7383333333   1.3544018059   0.8634651389   0.4909366670
```

```
290   5.8166666667   0.1719197708   0.3695972222  -0.1976774514
291   5.8350000000   0.1713796058   0.3724862500  -0.2011066442
292   5.8533333333   0.1708428246   0.3754088889  -0.2045660643
293   5.8716666667   0.1703093954   0.3783651389  -0.2080557435
294   5.8900000000   0.1697792869   0.3813550000  -0.2115757131
295   5.9083333333   0.1692524683   0.3843784722  -0.2151260040
296   5.9266666667   0.1687289089   0.3874355556  -0.2187066467
297   5.9450000000   0.1682085786   0.3905262500  -0.2223176714
298   5.9633333333   0.1676914477   0.3936505556  -0.2259591078
299   5.9816666667   0.1671774868   0.3968084722  -0.2296309855
300   6.0000000000   0.1666666667   0.4000000000  -0.2333333333
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V4$
```



Einführung in die Programmierung für Studierende der Physik

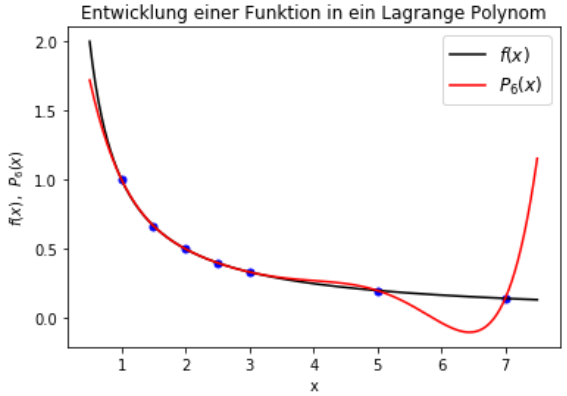
(Introduction to Programming for Physicists)

Vorlesung gehalten an der J.W.Goethe-Universität in Frankfurt am Main

(Sommersemester 2022)

von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

Frankfurt am Main 01.04.2022



Entwicklung einer Funktion in ein Lagrange Polynom

Zunächst wird das Python Modul "sympy" eingebunden, das ein Computer-Algebra-System für Python bereitstellt und eine Vielzahl an symbolischen Berechnungen im Bereich der Mathematik und Physik relativ einfach möglich macht. Falls Sie das "sympy" Modul das erste Mal verwenden, müssen Sie es zunächst in Ihrer Python 3 Umgebung installieren (z.B. in einem Linux Terminal mit "pip3 install sympy").

In [1]:
```
from sympy import *
init_printing()
```

Wir möchten nun die Funktion $f(x) = \frac{1}{x}$ mittels $(n+1)$ vorgegebener Punkte $((x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n)),)$ in ein Lagrange Polynom vom Grade N entwickeln. Hierzu definieren wir zunächst die Funktion:

Übungsblatt Nr. 5

Aufgabe 1 *(7.5 Punkte)*

Der im C++ Programm Lagrange_Polynom_1.cpp implementierte Algorithmus stellt eine nützliche Implementierung der Lagrange Polynom Methode dar, die wir vielleicht auch in zukünftigen Programmen verwenden können. Definieren Sie eine C++ Funktion, die den Kern-Algorithmus des Programms Lagrange_Polynom_1.cpp beinhaltet. Die Struktur der Funktion sollte dem allgemeinen C++ Funktionenschema "'Rückgabe Typ' **Funktionsname** ('Argumentenliste') { 'Block von Anweisungen' }" folgen (siehe Funktionen in C++) und in der 'Argumentenliste' sollte einerseits der Zeiger auf das im Hauptprogramm deklarierte Stützstellen-Array und seine Dimension stehen und andererseits sollten die im Hauptprogramm deklarierten Arrays double xp[N_xp+1], double fp[N_xp+1] und double Pfp[N_xp+1] mittels eines Aufrufs der Funktion mit den entsprechenden Datenwerten der Berechnung gefüllt werden. Testen Sie das Programm zunächst mittels des vorgegebenen Beispieles ($f(x) = 1/x$ und $\vec{x} = (2, 2.5, 4)$) und berechnen dann das Lagrange Polynom vom Grade $n = 8$ im Teilintervall $[a, b] = [0.91, 9.07]$ mit $f(x) = 10\,e^{-x/5} \cdot \sin(3\,x)$ und $\vec{x} = (1, 2, 3, 4, 5, 6, 7, 8, 9)$.

Aufgabe 2 *(7.5 Punkte)*

Die im C++ Programm Lagrange_Polynom_1.cpp implementierte Lagrange Polynom Methode approximiert eine vorgegebene Funktion $f(x)$ (hier speziell $f(x) = 1/x$) durch ein Lagrangepolynom. Wir nehmen jedoch im Folgenden an, dass die wirkliche Funktion $f(x)$ unbekannt ist, und wir lediglich an den Stützstellen die Funktionswerte kennen. Gegeben seien die folgenden x- und y-Werte der Stützstellen: $\vec{x} = (2, 10.1, 15, 20, 40, 60, 90)$ und $\vec{y} = (2.1, 41, 43, 40.2, 12, 5, 17.5)$. Berechnen Sie das Lagrange Polynom $P_7(x)$ im Teilintervall $[a, b] = [0, 100]$ mittels eines C++ Programms und stellen es grafisch mittels Python dar.

Aufgabe 3 *(5 Punkte)*

Im Unterpunkt C++ Arrays, Zeiger und Referenzen hatten wir gesehen, wie man ein Array an eine Funktion als Argument übergeben sollte: "Möchte man ein Array an eine Funktion als Argument übergeben, sollte man dies nicht über die einzelnen Werte des Arrays machen, da man dann die Array-Einträge nicht verändern kann. Stattdessen übergibt man ein Array (dies gilt auch für mehrdimensionale Arrays) als Zeiger auf sein erstes Element mit einem zusätzlichen Vermerk zu seiner Dimension." In gleicher Weise gilt dies auch für die gewöhnlichen skalaren Datentypen (int, double, ...). Konstruieren Sie eine Tausch-Funktion für zwei skalare double-Datentypen ("double a" und "double b"). Die Funktion soll dabei die im Hauptprogramm initialisierten Werte von "a" und "b" vertauschen, sodass nach dem Funktionsaufruf die Variable "a" den Wert von "b" und die Variable "b" den ursprünglich initialisierten Wert von "a" besitzt. Verwenden Sie hierbei als 'Rückgabe Typ' void und als 'Argumentenliste' den Zeiger auf a und auf b. Was wäre geschehen, hätten Sie anstatt der Zeiger die Werte der Variablen an die Funktion übergeben?