Georg-August-Universität Göttingen
Institut für Theoretische Physik
Prof. Dr. F. Heidrich-Meisner
Dr. E. Bothmann
S. L. Villani
SoSe 2021

Übungen zur Vorlesung
Computergestütztes Wissenschaftliches Rechnen
Blatt 3

Lernziele dieses Übungsblattes

- Numerische Nullstellensuche: Newton-Raphson-Algorithmus
- Explizites Euler-Verfahren
- Mehrdimensionale, gekoppelte Systeme
- Programmierkonzepte: Arrays, Pointer

Aufgabenmodus

Die Aufgaben 7 und 9 dieses Arbeitsblattes verfügen über Tutorials, die Sie am Ende des Dokumentes finden. Abhängig von Ihren Vorkenntnissen könne Sie die Aufgaben entweder eigenständig bearbeiten, oder dem dazugehörigen Tutorial folgen. Ziel der Tutorials ist es, Sie durch die grundlegenden Lernkonzepte zu führen und Ihre C-Kenntnisse aufzufrischen. Nach Abschluss eines Tutorials haben Sie ein funktionierendes Programm vorliegen und die entsprechende (Teil-)Aufgabe hinreichend bearbeitet.

Die Tutorials fallen zunächst sehr feinschrittig aus, werden aber im Laufe des Semesters zunehmend aufeinander aufbauen. Bereits erläuterte Konzepte müssen Sie dann ggfs. in früheren Übungen nachlesen.

Die Tutorials sind lediglich als Lösungs-Vorschlag zu verstehen. Wir ermutigen Sie, auch Ihre eigenen Implementierungen auszuprobieren.

Übungsaufgaben

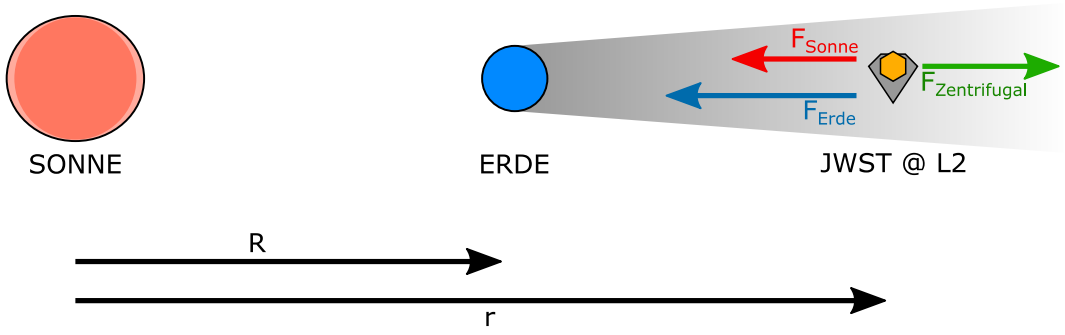
Aufgabe 7 *Newton-Raphson-Algorithmus (Tutorial)*

Hinweis: Diese Aufgabe hat ein Tutorial.

Das James Webb Space Telescope (JWST) soll im Oktober 2021 seinen Dienst als Nachfolger des Hubble-Teleskops antreten. Das JWST arbeitet im fernen Infrarotbereich und seine Instrumente benötigen hierfür ein komplexes Kühlungskonzept. Essentiell dabei ist die Positionierung des Teleskops: Um jegliche Sonneneinstrahlung zu vermeiden wird das JWST im Schatten der Erde auf dem sonnenabgewandten Lagrangepunkt L2 geparkt.

Ziel dieser Aufgabe ist es, die Position von L2 und seine Entfernung zur Erde zu berechnen. Die zugrundeliegende Gleichung ist, als Polynom fünften Grades, analytisch nicht lösbar. Mit einem numerischen Nullstellensucher wie dem Newton-Raphson-Algorithmus (der weiter unten beschrieben wird) kann der Lagrangepunkt aber leicht bestimmt werden.

Wir betrachten das Sonne-Erde-System in einem mitrotierenden Koordinatensystem. Die Erde bewege sich auf einer perfekten Kreisbahn um die Sonne und stehe somit in den gewählten Koordinaten still. Die Bewegung der Sonne sei ebenfalls vernachlässigbar.



Führen Sie alle Berechnungen in 1D durch, auf dem Verbindungsstrahl von der Sonne durch die Erde.

1. Auf das JWST wirken drei beschleunigende Kräfte: die Gravitationskräfte von Sonne und Erde, sowie eine Zentrifugalkraft durch die Wahl der Koordinaten. Der Lagrangepunkt L2 markiert den Punkt, an denen sich diese drei Kräfte gerade zu Null aufheben.

   Stellen Sie die Gleichung für ein Kräftegleichgewicht auf. Die Masse des JWST lässt sich aus der Gleichung herauskürzen:

   $$\pm a_{\text{Erde}} \pm a_{\text{Sonne}} \pm a_{\text{zentrifugal}} \stackrel{!}{=} 0$$

   Sie dürfen annehmen, dass sich der L2 hinter der Erde befindet und die Vorzeichen entsprechend wählen.

   Die Gravitationsbeschleunigungen sind von der Form $-\frac{\mu}{\Delta x^2}$, die Zentrifugalbeschleunigung von der Form $\omega^2 r$.

2. Schreiben Sie eine Funktion `acceleration`, die die linke Seite der Gleichung zurückgibt. Der Parameter `r` ist der Abstand des Teleskops von der Sonne:

```
double acceleration(double r)
{
    double a_earth = ...
    double a_sun = ...
    double a_centrifugal = ...
    return a_earth + a_sun + a_centrifugal;
}
```

Sie benötigen folgende physikalische Konstanten:

(a) Gravitationsparameter der Erde $\mu_{\text{Erde}} = 3.986 \cdot 10^{14} \frac{\text{m}^3}{\text{s}^2}$

(b) Gravitationsparameter der Sonne $\mu_{\text{Sonne}} = 1.327 \cdot 10^{20} \frac{\text{m}^3}{\text{s}^2}$

(c) Kreisfrequenz des Erdorbits $\omega = 1.991 \cdot 10^{-7}\text{Hz}$

(d) Radius des Erdorbits $R = 1.496 \cdot 10^{11}\text{m}$

3. Fügen Sie Ihrer Numerik-Bibliothek die neue Funktion

```
double find_root(double func(double), double x0,
                 double delta, double rel_tol,
                 int max_iter)
{ ... }
```

hinzu. Die Funktion berechnet eine Nullstelle einer Funktion mit dem Newton-Raphson-Algorithmus. Die übergebenen Parameter sollen folgende Funktionsweise haben:

`func`	Die Funktion deren Nullstelle gefunden werden soll.
`x0`	Der Startwert (initial guess) für die Iteration.
`delta`	Die Schrittweite bei der Berechnung der Ableitung.
`rel_tol`	Relative Toleranz zwischen den Iterationen. Wenn die prozentuale Abweichung der Lösung zwischen zwei aufeinander folgenden Iterationen kleiner ist als `rel_tol`, dann wird die Lösung akzeptiert.
`max_iter`	Die maximale Anzahl an Iterationen bevor die Berechnung erzwungen beendet wird.

Der Newton-Raphson-Algorithmus ist ein iteratives Verfahren, das durch wiederholte Anwendung eine Nullstelle einer Funktion $f(x)$ bestimmt. Ausgehend von einem Anfangswert x_0 wird folgende Vorschrift benutzt, um die Lösung x_i zunehmend zu verbessern:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Benutzen Sie Ihre Funktion `diff` vom letzten Übungszettel um die Ableitung zu berechnen.

4. Finden Sie die Nullstelle der Funktion `acceleration` mit der eben erstellten Routine `find_root` und berechnen Sie den Abstand $R-r$ von L2 zur Erde. Achten Sie darauf, einen sinnvollen Startwert `x0` zu wählen. Als Schrittweite ist $\delta = 1\text{km}$ eine geeignete Wahl. Vergleichen Sie Ihre Berechnung mit Literaturwerten.

Aufgabe 8 *Epidemie-Modellierung*

Ein einfaches Modell für die Dynamik einer Epidemie ist das SEIR-Modell. Es betrachtet vier Gruppen von Personen, die an der Dynamik der Epidemie in unterschiedlicher Weise teilnehmen:

- **Susceptible**: Personen der Gruppe S sind anfällig für die Krankheit.
- **Exposed**: Personen der Gruppe E wurden der Krankheit ausgesetzt und tragen den Erreger in sich. Die Gruppe ist noch nicht ansteckend.
- **Infectious**: Personen der Gruppe I haben die Krankheit soweit ausgebildet, dass sie nun Personen der Gruppe S anstecken können.
- **Removed**: Personen der Gruppe R nehmen nicht mehr an der Dynamik der Epidemie teil, weil sie entweder gestorben oder gesundet und immunisiert sind.

Der Übergang von einer Gruppe zur anderen geschieht mit einer gewissen Rate β, a oder γ und folgt dem folgenden Schema:

$$S \xrightarrow{\beta} E \xrightarrow{a} I \xrightarrow{\gamma} R.$$

Die Dynamik der Größen wird durch das folgende System von Differentialgleichungen beschrieben:

$$\frac{dS}{dt} = -\beta\frac{I \cdot S}{N}, \qquad \frac{dE}{dt} = \beta\frac{I \cdot S}{N} - aE, \qquad \frac{dI}{dt} = aE - \gamma I. \tag{1}$$

Vitaldynamiken (Geburten und natürliche Sterbefälle) werden hier ignoriert. Der Parameter β ist die durchschnittliche Ansteckungsrate pro Person, a ist der Kehrwert der mittleren (exponentiell verteilten) *präinfektiösen Zeit* und γ der Kehrwert der mittleren *infektiösen Zeit*. Die Ansteckungsrate kann alternativ auch über die Basisreprodukionszahl $R_0 = \beta/\gamma$ angegeben werden.

Verwenden Sie die folgenden Werte für a und γ:

- $a = (6{,}1 \text{ Tage})^{-1}$
- $\gamma = (3{,}7 \text{ Tage})^{-1}$

1. Diskretisieren Sie das SEIR-Modell unter Anwendung des expliziten Euler-Verfahrens. Überführen Sie hierfür die Zeitableitung einer Größe A in eine Vorwärtsdifferenz und nummerieren Sie die diskreten Schritte mit einem Index i:

$$\frac{dA}{dt} \quad \longrightarrow \quad \frac{A(t+\Delta t) - A(t)}{\Delta t} \equiv \frac{A_{i+1} - A_i}{\Delta t}\,.$$

   Für die explizite Version des Verfahrens wird die rechte Seite des Gleichungssystems dann zum Zeitpunkt i ausgewertet. Stellen Sie die Gleichungen nach den Größen A_{i+1} des nächsten Zeitschrittes um.

2. Schreiben Sie ein C-Programm, das die diskretisierten SEIR-Gleichungen löst. Schreiben Sie den zeitlichen Verlauf der Populationsgrößen in eine CSV-Datei. Verwenden Sie Anfangswerte von $E(t=0) = 30000$ und $I(t=0) = 9000$. Die Gesamtpopulation beträgt $83 \cdot 10^6$.

Untersuchen Sie das Verhalten für die verschiedenen Basisreproduktionszahlen $R_0 = [1.25, 1.5, 2]$.

Als Zeitschritt ist $\Delta t = 0.01$ geeignet, also 100 Samples pro Tag.

3. Plotten Sie die Populationsgrößen gegen die Zeit. Bestimmen Sie die maximale Größe der Infektiösen Gruppe.

Aufgabe 9 *Arrays und Pointer (Tutorial)*

Diese Aufgabe dient zur Auffrischung Ihrer C-Kenntnisse über Arrays und Pointer. Sie können die Aufgabe überspringen, wenn Sie sich sicher in diesen Konzepten fühlen. Die Aufgabe verfügt über ein Tutorial.

1. Erstellen Sie ein `int`-Array der Länge 100 und füllen Sie es mit aufsteigenden natürlichen Zahlen beginnend bei 10. Geben Sie den Inhalt auf die Konsole aus.
2. Erstellen Sie einen Pointer auf einen Speicherbereich, der 100 `int`-Werte umfasst und füllen Sie ihn mit aufsteigenden natürlichen Zahlen beginnend bei 10. Geben Sie den Inhalt auf die Konsole aus.
3. Schreiben Sie eine Funktion, die einen Pointer entgegen nimmt und in dem angezeigten Speicherbereich 100 Zahlen mit 2 multipliziert.

   Wenden Sie die Funktion auf das Array und auf den allozierten Speicherbereich an. Geben Sie dann deren Inhalt auf die Konsole aus.
4. Arrays und Pointer scheinen in diesen einfachen Anwendungen sehr ähnlich zu funktionieren. Tatsächlich sind die technischen Unterschiede aber enorm. Untersuchen Sie folgende Fragen:
   - Welche Größen gibt die Funktion `sizeof()` für das Array und den Pointer aus?
   - Welchen Unterschied gibt es zwischen den Funktionsparametern `double *D` und `double D[]`?
   - Was passiert wenn Sie auf das Array oder auf den Pointer 1 addieren und dann den ersten Wert ausgeben?
   - Was passiert wenn Sie ein Array von 10MB Größe anlegen und beschreiben? Was passiert, wenn Sie einen Speicherbereich von 10MB mit `malloc` anlegen und beschreiben?
5. Geben Sie das Ergebnis von `A[10]` aus (`A` sei Ihr Array). Was passiert wenn Sie versuchen `10[A]` auszugeben? Wie erklärt sich das Verhalten?

Selbsttest

- Wie funktioniert der Newton-Raphson-Algorithmus?
- Wie funktioniert das explizite Euler-Verfahren?
- Wann benutze ich Arrays und wann `malloc`?
- Warum ist es wichtig, nicht mehr benötigten Speicher freizugeben?
- Warum sind Änderungen an einem Array durch eine Funktion außerhalb der Funktion sichtbar?

Tutorials

Die Übungen und die Tutorials sollen Ihnen die *divide and conquer* Strategie nahebringen. *Teile und herrsche* bedeutet, ein komplexes Problem in viele kleine Einzelprobleme zu zerlegen, die für sich gesehen einfach zu lösen sind. Wenn ein (Einzel)-Problem dann gelöst ist, müssen Sie sich keine weitere Gedanken darum machen und können die Komponente wie eine Blackbox als *gegeben* ansehen. Am Ende werden die Komponenten zusammengesetzt und lösen das eigentliche Problem.

Genau diese Strategie haben Sie schon auf dem ersten Übungsblatt gesehen, als Sie die Integrationsfunktion `integrate` geschrieben haben und darauf basierend dann die Fehlerfunktionen `my_erf_...` erstellt haben. Nachdem die Behandlung von Integralen abgehandelt wurde, ist es leicht, komplexere mathematische Funktionen darauf aufzubauen.

Bevorzugen Sie *divide and conquer* immer vor einer mutmaßlich "optimierten" Umsetzung, in der sämtliche Systeme miteinander verwoben in einem einzelnen Codeblock umgesetzt werden. Ihre Programme werden dadurch deutlich übersichtlicher, verständlicher und leichter wartbar. Optimieren Sie Ihren Code erst am Ende, wenn Sie tatsächliche Daten über das Laufzeitverhalten sammeln können (performance profiling, z.B. mit `gprof`).

Aufgabe 7 - Newton-Raphson-Algorithmus

1. Stellen Sie zunächst die Gleichung wie in 1 gefordert auf.
2. Erstellen Sie eine neue Datei `lagrange.c`. Binden Sie die Header `stdio.h`, `stdlib.h`- und `tgmath.h` ein.
3. Definieren Sie unterhalb der Headereinbindung direkt die physikalischen Konstanten:

   ```
   const double mu_earth = 3.986e14;
   const double mu_sun = 1.327e20;
   const double omega = 1.991e-7;
   const double radius_orbit_earth = 1.496e11;
   ```

   Das Schlüsselwort `const` weist den Compiler an, keine weitere Änderungen an dieser Variable zuzulassen. Sie sollten jede Variable, die keiner Veränderung unterliegt, als `const` deklarieren. Damit können eine Vielzahl an Bugs schon während der Kompilierung erkannt und dann behoben werden.

Die eben definierten Variablen sind außerhalb einer Funktion definiert und stehen damit im *File Scope*. Das bedeutet, an jeder Stelle im Code dieser Quelltext-Datei sind die Variablen verfügbar. Der Compiler stellt sicher, dass diese Variablen vor Beginn der `main`-Funktion initialisiert werden (Variablen im File Scope werden garantiert zu Null gesetzt, im Gegensatz zu automatischen Variablen innerhalb einer Funktion).

File Scope Variablen haben einige Vorteile, besonders hier im Kontext physikalischer Konstanten. Sie stellen aber auch eine Klasse von Globalen Variablen dar. Globale Variablen bedürfen besonderer Vorsicht, denn sie führen bei unachtsamem Programmdesign schnell zu undurchschaubaren und extrem schwer zu debuggenden kausalen Strukturen, besonders wenn verschiedene Teile des Programmes über eine Globale Variable hinweg kommunizieren. Wenn Sie den Tutorials folgen, werden Sie nur unbedenkliche Anwendungen dieser Variablen vorgeführt bekommen.

Benutzen Sie keine `#define` Befehle, um Konstanten zu definieren. Der Compiler kann keine Überprüfung der Typen durchführen. Das Programm läuft dadurch auch nicht schneller oder benötigt weniger Speicher, da der Compiler unnötige Variablen erkennt und wegoptimiert. Z.B. können viele numerischen Konstanten direkt im Maschinenbefehl (als sog. *immediate*) eingebettet werden, so dass die Variable erst gar nicht im Speicher auftaucht.

4. Erstellen Sie jetzt die Funktion

```
double acceleration(const double r)
{
    double a_earth = ???
    double a_sun = ???
    double a_centrifugal = omega*omega*r;

    return a_earth + a_sun + a_centrifugal;
}
```

   Definieren Sie am besten auch konstante Funktions-Parameter wie `r` als `const`. Das ist gute Praxis und kann Ihnen viele Fehlersuchen ersparen.

   Die Zentrifugal-Beschleunigung greift hier auf die globale Variable `omega` zu. Ergänzen Sie bei `???` noch die korrekte Berechnung für die Gravitations-Terme.

5. Kopieren Sie Ihre Numerik-Bibliothek (`my_numerics.c` und `my_numerics.h`) in denselben Ordner wie `lagrange.c`. Alternativ berücksichtigen Sie die Anweisungen aus dem Projekstruktur-Tutorial. Erstellen Sie dann passend dazu folgendes makefile:

```
lagrange: lagrange.o my_numerics.o
    gcc lagrange.o my_numerics.o -lm -o lagrange

lagrange.o: my_numerics.h lagrange.c
    gcc -c lagrange.c

my_numerics.o: my_numerics.h my_numerics.c
    gcc -c my_numerics.c
```

6. Erstellen Sie in `my_numerics.c` die neue Funktion

```
double find_root(double func(double), double x0,
                 double delta, const double rel_tol,
                 const int max_iter)
{
    ...
}
```

7. Fügen Sie der Funktion einen Iterationszähler und die gesuchte Nullstelle `x` hinzu:

```
    double x = x0;
    int iteration = 0;
```

8. Benutzen Sie eine `while`-Schleife, um die Iteration wiederholt zu berechnen. Benutzen Sie dafür folgenden Code und ergänzen Sie die fehlenden Ausdrücke bei `???`.

   Vergessen Sie nicht, ihr Ergebnis mithilfe des `return`-Befehls zurückzugeben!

```
    while(iteration < ???) {

        // evaluate function and derivative
        double  f = ???;
        double df = diff(???);

        // store current x for tolerance comparison
        double x_old = x;

        // do iteration step
        x -= ???

        if(fabs(x_old-x)/fabs(x) < rel_tol)
            break;

        iteration++;
    }

    return x;
```

9. Fügen Sie die Funktionsdeklaration dem Header `my_numerics.h` hinzu:

```
double find_root(double func(double), double x0,
                 double delta, const double rel_tol,
                 const int max_iter);
```

10. Binden Sie den Header `my_numerics.h` in die C-Quelle Ihres Hauptprogramms ein.

11. Erstellen Sie dort auch eine `main`-Funktion und bearbeiten darin den Rest der Aufgaben.

Aufgabe 9 - Arrays und Pointer

Dieses Tutorial zeigt Ihnen einige grundlegende Anwendungen von Arrays bzw. Heap-Speicher. Bearbeiten Sie die Aufgaben nebenher, während Sie das Tutorial durchgehen.

Sie können Arrays anlegen, um auf eine Vielzahl von Daten über einen Index strukturiert zuzugreifen. Der Befehl

```
int a[100];
```

legt ein Array aus 100 Integers auf dem Stapel (Stack) an. Analog können Sie mit

```
double a[100];
```

ein Array aus 100 Doubles anlegen. Das funktioniert mit allen Datentypen, auch Strukturen. Der Zugriff auf die einzelnen Elemente des Arrays erfolgt über einen Index (beginnend bei 0) in eckigen Klammern. Der Befehl

```
a[10] = 1001;
```

weist dem elften Integer den Wert 1001 zu.

Die Stärke von Arrays liegt darin, dass der Index dynamisch programmiert werden kann. Sie können z.B. mit einer `for`-Schleife durch das gesamte Array gehen und alle Werte ausgeben:

```
for(int i = 0; i < 100; ++i)
    printf("a[%d] = %d\n", i, a[i]);
```

Analog können Sie ein Array mit Werten füllen:

```
for(int i = 0; i < 100; ++i)
    a[i] = 2*i;
```

Arrays werden auf dem Stack initialisiert, wodurch ihre Größe beschränkt ist. Wenn Sie große Arrays benötigen, so müssen Sie den notwendigen Speicherplatz aus dem Heap anfordern, was mit dem Befehl `malloc` gelingt.

Sie initialisieren einen Speicherbereich auf dem Heap mit folgendem Befehl:

```
int *p = malloc(100*sizeof(int));
```

`malloc` nimmt als Argument die Anzahl der Bytes entgegen, die reserviert werden sollen. Der Befehl `sizeof` liefert die Anzahl der Bytes des übergebenen Datentyps zurück. Somit reserviert dieser Befehl Speicher für genau 100 Integers. (Benutzen Sie `sizeof()` um die 10MB-Aufgabe zu bearbeiten.)

`malloc` liefert eine Speicheradresse des reservierten Bereichs zurück, die in der Variable `p` gespeichert wird. Hierbei ist `p` vom Datentyp `int*`, d.h. ein Integer-Pointer. Pointer dienen zum Speichern von Speicheradressen (mehr dazu in späteren Übungen). Sie können auf die Integers im Heap dann mit der gleichen Syntax zugreifen, wie schon beim Array:

```
for(int i = 0; i < 100; ++i)
    p[i] = 2*i;
```

Wenn Sie den Speicher nicht mehr brauchen, müssen Sie ihn unbedingt mit

```
free(p);
```

wieder freigeben, damit das Programm nicht immer mehr Speicher akkumuliert und dann vom Betriebssystem terminiert wird. Das ist besonders in Funktionen wichtig, wo der Zeiger am Ende der Funktion seine Gültigkeit verliert und damit die Adresse des reservierten Speichers verloren geht (auch wenn die Funktion später erneut aufgerufen wird).

Der Inhalt von Arrays und von Heap-Speicher kann von Funktionen bearbeitet werden. Die Funktion

```
void add_one(int ARR_SIZE, int ARR[ARR_SIZE])
{
    for(int i = 0; i < ARR_SIZE; ++i)
        ARR[i] += 1;
}
```

addiert zu den ersten 100 Einträgen des Speichers 1. Das funktioniert, weil hier (neben der Anzahl der Einträge) lediglich die Speicher*adresse* übergeben wird, nicht der Speicher selbst. Beachten Sie daher, dass hier das originale Array verändert wird, die Änderungen also ohne `return` außerhalb der Funktion sichtbar sind. Sie können der Funktion entweder einen Pointer übergeben, oder ein Array. C wandelt dann das Array automatisch zu einem Pointer auf das erste Element um:

```
add_one(100, p);
add_one(100, a);
```

Sie müssen dafür Sorge tragen, dass Sie die Array-, bzw. Speichergrenzen einhalten. Das gelingt für Funktionen am einfachsten mit der oben gezeigten Notation von `add_one`. Bei ungültigen Zugriffen im Heap meldet Ihnen das Betriebssystem in der Regel eine Speicherschutzverletzung. Falsche Zugriffe auf ein Array können aber leicht unentdeckt bleiben und mitunter zu völlig unerwartetem und absurdem Programmverhalten führen. Zugriffe außerhalb des Arrays betreffen nämlich in der Regel andere Variablen und andere Strukturen auf dem Stapel.