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 2

Lernziele dieses Übungsblattes

- Genauigkeit von Gleitkommazahlen
- Numerische Ableitungen und ihre Genauigkeit
- Programmierkonzepte: Code-Wiederverwendbarkeit, Linker, Makefiles

Aufgabenmodus

Die Aufgabe 5) verfügt über ein Tutorial, das Sie am Ende des Dokumentes finden. Abhängig von Ihren Vorkenntnissen können Sie die Aufgabe 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 ggf. in früheren Übungen nachlesen.

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

Manche Teilaufgaben sind als **Bonus**-Aufgaben gekennzeichnet und haben einen fortgeschrittenen Schwierigkeitsgrad. Sie dienen der Vertiefung Ihres Verständnisses der Algorithmen. Wir empfehlen Ihnen, zunächst die normalen Aufgaben zu bearbeiten.

Übungsaufgaben

Aufgabe 4 *Gleitkommagenauigkeit*

In dieser Aufgabe wird demonstriert, wie analytisch äquivalente Formeln bzw. Ausdrücke bei numerischer Behandlung aufgrund der zugrundeliegenden beschränkten Genauigkeit zu verschiedenen Ergebnissen führen können.

Das quadratische Polynom $ax^2 + bx + c = 0$ hat bekannterweise die beiden Lösungen

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\,.$$

1. Schreiben Sie ein C-Programm, das eine Quadratische Gleichung mit Hilfe dieser Darstellung löst.

2. Lösen Sie mit Ihrem Programm die Gleichung

$$x^2 - \frac{10^{16}+1}{10^8}x + 1 = 0.$$

   Die analytischen Lösungen sind $x_1 = 10^8$ und 10^{-8} (bestätigen Sie dies durch eigene Rechnung). Berechnen Sie den *relativen Fehler*, den Ihre numerischen Lösungen gegenüber der analytischen Lösung aufweisen. Zeigen Sie damit, dass eine der beiden Lösungen unbrauchbar ist.

   Hinweis: Benutzen Sie im `printf` Befehl für die Ausgabe von `double`-Werten den Platzhalter `%g`.

3. Erweitert man den Bruch der obigen Gleichung mit $-b \mp \sqrt{b^2 - 4ac}$, so lassen sich die beiden Lösungen des Polynoms äquivalent als

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

   darstellen. (Beachten Sie den Wechsel der Vorzeichen $\pm$ zu $\mp$!) Erweitern Sie Ihr Programm und bestimmen Sie die Lösung der Gleichung auch mit dieser neuen Darstellung. Zeigen Sie, dass nun die andere Lösung starke Abweichungen aufweist.

4. Erklären Sie diese Beobachtungen indem Sie sich überlegen, welche Terme besonders groß oder besonders klein werden und bei welcher Gleitkomma-Operation das zu Schwierigkeiten führt.

5. Schreiben Sie eine Funktion `solve_quadratic`, die die beiden Ansätze so kombiniert, dass beide Nullstellen mit bestmöglicher Genauigkeit berechnet werden. Da die Funktion zwei verschiedene Werte zurückgeben soll, müssen Sie zunächst einen neuen Datentyp in Form einer Struktur anlegen:

```
typedef struct {
    double x1;
    double x2;
} Tuple;
```

   In den Tipps am Ende des Dokumentes finden Sie eine kurze Anleitung zur Verwendung von Strukturen.

   Erstellen Sie dann die Funktion in folgender Form:

```
Tuple solve_quadratic(double a, double b, double c)
{
    Tuple solution;
    solution.x1 = ...
    solution.x2 = ...
    return solution;
}
```

Hinweis: Sie benötigen eine Fallunterscheidung bezüglich des Vorzeichens von b. Überprüfen Sie Ihren Algorithmus an der Gleichung

$$-x^2 + \frac{10^{16}+1}{10^8}x - 1 = 0,$$

welche die selben analytischen Lösungen aufweist.

Aufgabe 5 *Wiederverwendung von Code (Tutorial)*

Hinweis: Diese Aufgabe hat ein Tutorial.

Sie haben nun bereits einige numerische Algorithmen implementiert:

- `integrate()`
- `erf_midpoint()`
- `erf_simpson()`
- `solve_quadratic()`

Im Laufe des Semesters werden noch viele weitere Funktionen hinzukommen. Ziel dieser Aufgabe ist es, eine eigene Numerik-Bibliothek `my_numerics` anzulegen und Ihre Algorithmen dort zur Wiederverwendung einzupflegen. Sie finden im Zusatzmaterial der Vorlesung (StudIP) ein Dokument, das Ihnen etwas detaillierter eine mögliche Projektstruktur erläutert.

1. Erstellen Sie zwei neue Dateien `my_numerics.c` und `my_numerics.h`. Statten Sie den Header mit einem Include-Guard aus und binden Sie ihn in die neue `.c`-Quelle ein.
2. Kopieren Sie den Quellcode der Funktionen `integrate`, `solve_quadratic`, `erf_simpson` und `erf_midpoint` in `my_numerics.c`. Sie benötigen außerdem Ihren Integranden der Fehlerfunktion. Beschränken Sie mit `static` den Scope des Integranden auf die Übersetzungseinheit der Bibliothek.

   Schreiben Sie die Funktionsdeklarationen in den Header. Die Strukturdefinition von `Tupel` kommt ebenfalls in den Header.
3. Kompilieren Sie `my_numerics.c` zu einem Object. Benutzen Sie hierfür in `gcc` den Parameter `-c`.
4. Erstellen Sie ein kleines C-Projekt, in das Sie die Header-Datei `my_numerics.h` einbinden und einige der Funktionen testen. Kompilieren Sie das Projekt, verlinken Sie es mit dem Object und führen Sie es zum Test aus.
5. Schreiben Sie ein kurzes Makefile, das den gesamten Vorgang des Kompilierens und Linkens automatisiert.

Aufgabe 6 *Numerische Ableitungen*

Um die Ableitung $f'(x)$ einer Funktion numerisch zu bestimmen, bedient man sich des Differenzenquotienten

$$f'(x) = \frac{f(x+\delta) - f(x)}{\delta}.$$

Während in der Analysis hier ein Grenzübergang $\delta \to 0$ angesetzt wird, belässt es die Numerik bei endlichen Differenzen, $\delta > 0$. Die richtige Wahl der Schrittweite δ ist hierbei entscheidend für die Genauigkeit der berechneten Ableitung.

Ziel dieser Aufgabe ist es, Sie mit der Problematik der Schrittweitenwahl vertraut zu machen.

1. Schreiben Sie die Funktion

   ```
   double diff(double x, double delta,
               double func(double));
   ```

   Sie soll die Ableitung einer Funktion `func` an der Stelle `x` berechnen. Nutzen Sie dafür den oben genannten Differenzenquotient mit Schrittweite `delta`. Die Funktion soll Ihrer Numerik-Bibliothek hinzugefügt werden; denken Sie daran, die Deklaration im dazugehörigen Header zu publizieren.

2. Bestimmen Sie die Ableitung der Funktion $f(x) = x^2(x-1)$ am Punkt $x = 1$ analytisch. Schreiben Sie dann eine C-Funktion, die den Wert der Funktion f zurückgibt.

3. Berechnen Sie mithilfe der selbst geschriebenen Funktion `diff` aus Ihrer Bibliothek die Ableitung der Funktion $f(x)$ an der Stelle $x = 1$ numerisch. Führen Sie diese Berechnung 100 mal durch, und zwar für Werte von δ, die im Bereich 10^0 bis 10^{-16} logarithmisch verteilt sind. Schreiben Sie den Betrag der Differenz zwischen analytischem und numerischem Wert der Ableitung zusammen mit dem Wert für δ in eine Datei auf der Festplatte.

4. Plotten Sie den numerischen Fehler der Ableitung gegen die verwendete Schrittweite und bestimmen Sie das Minimum des Fehlers. Nutzen Sie hierfür eine doppellogarithmische Darstellung. Können Sie den Verlauf der Kurve erklären?

5. Verbessern Sie Ihre `diff`-Funktion, indem Sie die numerische Ableitung mithilfe der *Zentralen Differenz*

   $$f'(x) = \frac{f(x+\delta) - f(x-\delta)}{2\delta}$$

   berechnen. Während die Konvergenzordnung des bisher benutzten Differenzenquotienten (hier: Vorwärtsdifferenz) von Ordnung $\mathcal{O}(\delta)$ ist, ist diese bei der Zentralen Differenz $\mathcal{O}(\delta^2)$. Wie wirkt sich dies auf den Kurvenverlauf in Ihrem doppellogarithmischen Plot aus?

Selbsttest

- Wieso kann man mit einem Computer nicht beliebig genau rechnen?

- Was ist die grundlegende Vorgehensweise, wenn ich Code wiederverwenden möchte bzw. für zukünftige Programme wiederverwendbar designen möchte?
- Was versteht man unter einem 'Object' beim Kompilieren?
- Kann man eine Funktion an eine Funktion übergeben?
- Wie bestimmt man numerisch eine Ableitung, und was bestimmt die Genauigkeit des Ergebnisses?
- Was ist ein Include-Guard?
- Was ist die grundlegende Struktur und funktionsweise eines `make`-Files?
- Wann ist es sinnvoll, die `const`-Deklaration zu verwenden?
- Was kann man tun, um beim Aufrufen von Funktionen von Matrizen zu vermeiden, dass unnötigerweise Daten hin- und her kopiert werden?

Tutorials

Aufgabe 5 - Wiederverwendung von Code

1) Kopieren Sie die C-Datei Ihrer Bearbeitung der Aufgabe 3 vom Blatt 1 (d.h. von Ihrem Fehlerfunktions-Projekt) in ein neues Verzeichnis. Sie sollten dort folgende Funktionen erstellt haben:

```
double gaussian(double y)
{
    return exp(-y*y);
}

double integrate(double left, double right,
                 int N, double integrand(double))
{
    ...
}

double erf_simpson(double x, double delta_x)
{
    ...
}

double erf_midpoint(double x, double delta_x)
{
    ...
}

int main()
{
    ...
}
```

Dabei ist `gaussian` der verwendete Integrand für die Fehlerfunktion, dem Sie evtl. aber einen anderen Namen gegeben haben.

2) Erstellen Sie eine neue Datei namens `my_numerics.c`. Bewegen Sie alle Funktionen von letzter Woche (außer `main()`!) in diese neue Datei. Schreiben Sie `static` vor den Integranden:

```
static double gaussian(double y){
    return exp(-y*y);
}
```

Auf diese Weise ist die Funktion ausschließlich in der Datei `my_numerics.c` sichtbar. Das ist wichtig, da Sie sonst den Funktionsnamen "`gaussian`" nirgends mehr verwenden können, sobald gegen die Bibliothek verlinkt werden soll ("*polluting the namespace*"). Wenn Sie besonders gründlich sein wollen, dann benennen Sie sämtliche Funktionen Ihrer Bibliothek mit einem Präfix, der ihre Bibliothek kennzeichnet, wie z.B. `mn_integrate` und `mn_erf`. Das wird für diesen Kurs nicht notwendig sein, Sie werden aber sehen, dass z.B. sämtliche Funktionen der GNU Scientific Library den Präfix "`gsl_`" tragen.

Da eine Ihrer Funktionen den `<tgmath.h>`-Header benötigt, inkludieren Sie diesen oben in `my_numerics.c`.

3) Erstellen Sie eine neue Datei namens `my_numerics.h`. Fügen Sie folgenden Code, einen sogenannten Include-Guard, in den Header ein:

```
#ifndef MYNUMERICS_H
#define MYNUMERICS_H

#endif
```

Alles was Sie in diese Datei schreiben, muss zwischen den Zeilen `#define` und `#endif` stehen.

Sie haben letzte Woche bereits gesehen, wie Sie den I/O-Header mit

```
#include <stdio.h>
```

einfügen. Der `#include` Befehl funktioniert denkbar einfach: Er kopiert vor dem eigentlichen Kompiliervorgang den Inhalt einer Textdatei an die Stelle des Quellcodes. Weil hierbei zunächst keine weitere logische Prüfung des Codes stattfindet, kann das zu Problemen führen. Es kommt häufig vor, dass durch verschachtelte `#include`-Befehle der selbe Text mehrmals im selben Quellcode landet. Wenn dadurch z.B. eine Variable mehrere Male deklariert wird, führt das zu einem Kompilierfehler.

Der Include-Guard ist ein einfaches Makro, das verhindert, dass Code doppelt an den Compiler übergeben wird. Sollte nach Abarbeiten der `include`-Befehle Code mehrfach im Quelltext stehen, so wird er trotzdem nur beim ersten Mal vom Compiler berücksichtigt.

4) Fügen Sie dem Header die Deklarationen von `integrate`, `erf_simpson` und `erf_midpoint` hinzu:

```
/* Numerical Integration of scalar 1D function */
double integrate(double left, double right,
                 int N, double integrand(double));

/* Implementations of the error function */
double erf_simpson(double x, double delta_x);
double erf_midpoint(double x, double delta_x);
```

Die Deklaration enthält Informationen über Ein- und Ausgabewerte einer Funktion (die *Signatur* der Funktion), nicht aber den eigentlichen Quellcode. Über den Header werden die Funktionen Ihrer Bibliothek für andere Programmteile publiziert, so dass Sie diese dort verwenden können. Den statischen Integranden deklarieren wir nicht im Header, da dieser lediglich als "Hilfsfunktion" für die `erf`-Implementationen dient. Die Header sind die erste Anlaufstelle, um sich über die verfügbaren Funktionen in einer Bibliothek zu informieren, es ist also eine gute Idee, Header sorgfältig zu kommentieren.

5) Kompilieren Sie Ihre Numerikbibliothek mit dem Befehl

   ```
   gcc -c my_numerics.c
   ```

   Dies erzeugt das Object `my_numerics.o`. Das Argument `-c` veranlasst gcc dazu, die Datei nur zu kompilieren, nicht aber zu verlinken und lauffähig zu machen.

   Das Object enthält also eine Sammlung an Funktionen, die jetzt von anderen Programmen benutzt werden kann und stellt damit schon eine sehr einfache Library dar. Typischerweise wird in der Praxis allerdings erst ein Archiv bestehend aus mehreren Objects als Library bezeichnet. Hier werden wir `my_numerics.o` aber nicht weiter verpacken.

6) Gehen Sie zurück in Ihre Hauptquelle, aus der Sie die Funktionen extrahiert haben. Inkludieren Sie Ihren Header:

   ```
   #include "my_numerics.h"
   ```

   Dadurch werden vor der Kompilierung die Deklarationen in den Quellcode kopiert. Die Deklarationen teilen dem Compiler mit, welche Funktionen existieren und wie mit ihnen kommuniziert wird.

7) Kompilieren Sie die Hauptdatei mit dem Befehl

   ```
   gcc -c exercise_3.c
   ```

   Sie haben jetzt zwei `.o`-Dateien, die sogenannten Objects. Sie enthalten den Maschinencode Ihrer Funktionen und Querverweise zwischen den Funktionen. Wenn Sie in `main` beispielsweise die Funktion `integrate` aufrufen, dann findet sich im Maschinencode von `main` momentan noch ein Hinweis auf einen fehlenden Sprungbefehl.

   Das Zusammenführen aller Maschinencode-Ausschnitte und das Einfügen der korrekten Sprungbefehle zwischen den Funktionen wird als *Verlinken* bezeichnet.

8) Verlinken Sie die beiden Objects zu einem lauffähigen Programm. Denken Sie daran auch die Mathematik-Funktionen mit `-lm` zu verlinken, damit auch Funtionen

wie `exp()` eingebaut werden:

```
gcc exercise_3.o my_numerics.o -lm -o exercise_3
```

9) Überprüfen Sie, dass Ihr Programm noch immer das gleiche Verhalten wie letzte Woche zeigt.

10) Fügen Sie nun auch noch die Funktion `solve_quadratic` in Ihre Bibliothek ein. Die Struktur-Definition schreiben Sie hierfür in den Header, der Quellcode kommt in die C-Datei. Fügen Sie eine Funktionsdeklaration in den Header ein. Da die Funktion die Definition der Struktur benötigt, muss der `my_numerics.h`-Header in der Quelle `my_numerics.c` selbst inkludiert werden!

11) Kompilieren Sie alles neu, und überprüfen Sie die Funktion Ihrer Algorithmen.

12) Damit Sie nicht bei jeder Änderung dreimal per Hand den `gcc` aufrufen müssen, kann der Kompiliervorgang mit Hilfe eines `makefile`s automatisiert werden. Ein `makefile` hat folgende, einfache Struktur:

```
ZIEL: ABHÄNGIGKEITEN
    BEFEHL
```

`ZIEL` bezeichnet eine Datei, die erstellt werden soll. Das können z.B. Objects oder Ihr fertiges Programm sein. `ABHÄNGIGKEITEN` ist eine Liste aller Dateien, von denen das `ZIEL` abhängt. Wenn die Abhängigkeiten verändert wurden, dann wird das Ziel neu erstellt. `BEFEHL` ist der Befehl, der das Ziel erstellt. Beachten Sie, dass Sie den Befehl mit einem Tabulatorschritt einrücken müssen!

Wenn eine der Dateien aus der Liste der Abhängigkeiten überhaupt nicht existiert, dann sucht `make` nach einer passenden Herstellungsvorschrift dafür, also ein anderes ZIEL.

Erstellen Sie eine Datei namens `makefile` und fügen Sie folgenden Code ein:

```
exercise_3: exercise_3.o my_numerics.o
    gcc exercise_3.o my_numerics.o -lm -o exercise_3

exercise_3.o: my_numerics.h exercise_3.c
    gcc -c exercise_3.c

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

Machen Sie sich klar, wie diese Datei den Kompiliervorgang strukturiert: In der ersten Zeile soll `exercise_3` erstellt werden, wozu aber z.B. `exercise_3.o` benötigt wird. Falls `exercise_3.o` nicht existiert, oder veraltet ist, wird zunächst Zeile 4 bearbeitet, um das Object `exercise_3.o` zu erstellen, usw.

Beachten Sie, dass natürlich keine Erstellvorschrift für die Quelldateien existiert: Diese stellen Sie als Programmierer bereit.

13) Löschen Sie alle Objects und das executable des Hauptprogramms (nicht den Code!) aus Ihrem Projektordner. Führen Sie in dem Ordner in einer Konsole den Befehl `make` aus, und überprüfen Sie, dass Ihr Projekt korrekt kompiliert und gelinkt

wird. Wenn Sie etwas am Projekt verändern, können Sie es mit `make` neu kompilieren. Wenn die Abhängigkeiten in dem makefile korrekt aufgestellt sind, dann werden nur diejenigen Teile des Projektes neu erstellt, die sich tatsächlich verändert haben.

Hinweis: Wenn Sie sich an den Tipps aus "CWR21-Projektstruktur.pdf" orientieren wollen, dann werden Sie das `makefile` und Ihre Dateiordnung etwas umgestalten müssen.

Hinweis: Die Dateien, die automatisiert durch ein Makefile erstellt werden, fügt man üblicherweise nicht zu dem git-Repository hinzu. Damit diese Dateien auch in der Ausgabe von `git status` nicht als unbekannte Dateien aufgeführt werden, kann man git anweisen, sie zu ignorieren, siehe den entsprechenden Tipp. Das Makefile sollte natürlich zum Repository hinzugefügt werden.

Tipps

Strukturen zur Zusammenfassung von Variablen

Sie können Strukturen in C benutzen, um Variablen in logische Einheiten zusammenzufassen. Möchten Sie z.B. einen 3D-Vektor erstellen, können Sie eine Struktur benutzen, um die Koordinaten x, y und z unter einer einzigen Variable zu speichern. Definieren Sie hierfür die Struktur `Vector3` mit folgendem Befehl:

```
struct Vector3 {
    double x;
    double y;
    double z;
};
```

Achten Sie auf das Semikolon nach der geschweiften Klammer. Sie können die Struktur wie einen normalen Datentyp verwenden, um eine neue Variable anzulegen.

```
struct Vector3 v;
```

Sie können auf die Komponenten des Vektors dann mit dem Punktoperator "`.`" wie folgt zugreifen:

```
v.x = 1.2;
v.y = 0.9;
v.z = 1.3;
```

Mit Hilfe von `typedef` können Sie die Struktur so definieren, dass Sie nicht jedes Mal `struct` schreiben müssen:

```
typedef struct {
    double x;
    double y;
    double z;
} Vector3;

Vector3 v;
v.x = 1;
```

Sie können Strukturen schnell mit folgender Konstruktion initialisieren:

```
Vector3 v = {.x=1.2, .y=0.9, .z=1.3};
```

Strukturen stellen auch eine Möglichkeit dar, mehrere Rückgabewerte von einer Funktion zu erhalten. Benutzen Sie hierfür die Struktur als Rückgabetyp oder Parametertyp:

```
Vector3 my_function(Vector3 w)
{
    Vector3 v;
    ...
    return v;
}
```

Wenn Sie große Datenmengen zwischen Funktionen austauschen wollen, sollten Sie, wann immer möglich, lieber Zeiger auf Strukturen und/oder Arrays austauschen, sonst werden ggf. unnötigerweise viele Daten hin- und her kopiert, was Zeit und Speicher verbraucht, aber sonst keine Vorteile mit sich bringt.

Ignorieren von abhängigen Dateien in git

Abhängige Dateien, die durch Befehle aus anderen Dateien erzeugt werden, möchten Sie typischerweise nicht Ihrem git-Repository hinzufügen, denn Sie können ja bei Bedarf einfach neu erstellt werden (z.B. bequem durch ein Makefile), und würden sonst nur das Repository "vollmüllen", so dass es schwieriger sein würde, die wirklich relevanten Änderungen zwischen Commits zu finden. Auch die Größe des Repository würde unnötig anwachsen, insbesondere da abhängige Dateien (Object-Dateien, Plots, ausführbare Dateien, ...) meist deutlich größer sind als die Quelldateien, die häufig von Menschen geschriebene Textdateien sind.

Listen Sie einfach alle Dateien, die git ignorieren soll, in einer Datei mit den Namen `.gitignore` auf. Dabei können Sie ein Sternchen als Platzhalter verwenden. Für die bisherigen Aufgaben könnte Ihr `.gitignore` z.B. so aussehen:

```
exercise_3
*.o
```

Diese Dateien werden jetzt nicht mehr in Befehlen wie `git status` als unbekannt aufgelistet. Solche Regeln sind natürlich für alle Nutzer dieses Repository nützlich. Deshalb sollte man `.gitignore`-Dateien dem Repository hinzufügen:

```
git add .gitignore
git commit -m "Add git-ignore file"
```

Dateien, die Sie vor der Definition der `.gitignore`-Regeln hinzugefügt haben, bleiben übrigens weiterhin Teil des Repository. Falls Sie sie löschen wollen, geht das mit `git rm <file>` und `git commit -m "Delete <file>"`. Falls die Datei wieder neu erstellt wird (z.B. durch `make`), wird sie dann in Zukunft ignoriert.