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 4

Lernziele dieses Übungsblattes

- Mehrdimensionale, gekoppelte Systeme
- Berechnung der Energie in mechanischen Systemen
- Explizites Euler-Verfahren
- Instabilitäten und Analytische Stabilitätsanalyse

Aufgabenmodus

Die Aufgabe 10 dieses Arbeitsblattes verfügt über ein Tutorial, das 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 ggf. in früheren Übungen nachlesen.

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

Prüfungsvorleistung

Die Bearbeitung von Aufgabe 12 ist die zweite von vier unbenoteten Prüfungsvorleistungen. Stellen Sie Ihre Bearbeitung bis zum 16. Mai, 12:00 Uhr über Gitlab zur Verfügung.

Ihr/e Tutor/in wird sich dann mit Ihnen in Kontakt setzen und ggf. notwendige Anpassungen mit Ihnen besprechen.

Übungsaufgaben

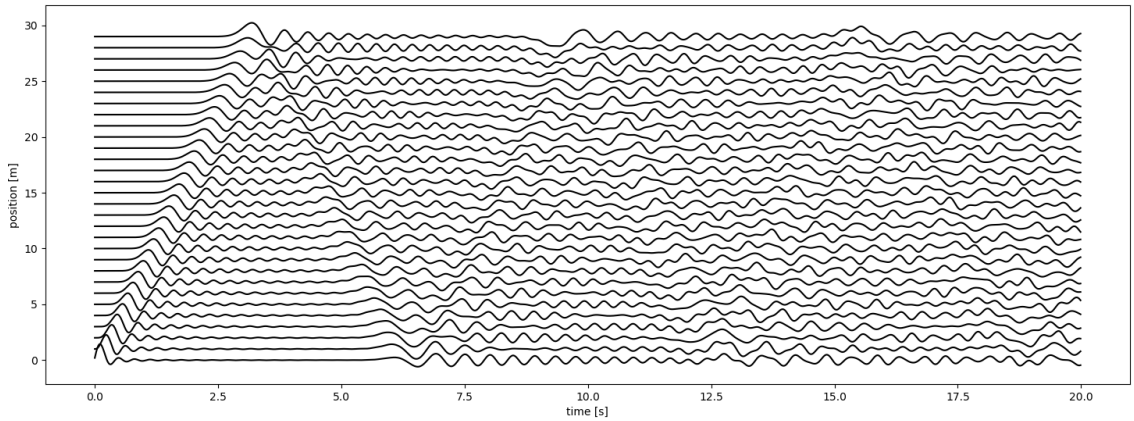
Aufgabe 10 *Gekoppelte Pendel mit GSL (Tutorial)*

Hinweis: Sie benötigen diese Pendelsimulation und Ihre Umsetzung auch auf dem nächsten Übungsblatt. Diese Aufgabe verfügt über ein Tutorial.

Sie beschäftigen sich nun mit einem einfachen System gekoppelter Pendelmassen. Anders als das SEIR-System auf dem letzten Übungsblatt ist dies ein System mit hoher Dimensionalität, weswegen ein strukturierter Ansatz zur Programmierung unablässig ist.

$N = 30$ Kugeln der Masse $m = 1\,\mathrm{kg}$ starten in einem Abstand von $L_0 = 1\,\mathrm{m}$ zueinander und sind mit Federn der Härte $k = 100\,\mathrm{N\,m^{-1}}$ miteinander verbunden. Die Ruhelänge der Federn sei ebenfalls L_0. Die erste Feder ist außerdem mit einer Feder der Ruhelänge $0\,\mathrm{m}$ an den Ursprung gebunden.

Zum Zeitpunkt t_0 erhält das erste Pendel einen Stoß, der es auf eine Geschwindigkeit von $20\,\mathrm{m\,s^{-1}}$ beschleunigt. Darauf bewegt sich eine Stoßwelle durch das System, die am freien Ende reflektiert wird. Trägt man die Positionen der Pendel gegen die Zeit auf, so ergibt sich folgender Graph:



In dieser Aufgabe werden Sie zunächst das System mit einem Integrator der GSL Bibliothek (*GNU Scientific Library*) lösen. GSL stellt eine breite Palette an numerischen Funktionen bereit, darunter auch Integratoren für gewöhnliche Differentialgleichungen.

Hinweis: Wenn Sie GSL auf Ihrem eigenen Rechner verwenden möchten, müssen Sie zunächst die notwendigen Pakete installieren. Beachten Sie die Anleitungen zur Einrichtung für die verschiedenen Betriebssystem im Stud.IP.

1. Machen Sie sich klar, wie das Differentialgleichungssystem der Pendel aufgebaut ist. Der Zustand des gesamten Systems wird durch den Vektor $\mathbf{y}$ beschrieben, der aus den Positionen und Geschwindigkeit der einzelnen Pendel aufgebaut ist:

$$\dot{\mathbf{y}} = \mathbf{F}(\mathbf{y}) \qquad \mathbf{y} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ v_1 \\ v_2 \\ \vdots \end{pmatrix} \qquad \mathbf{F}(\mathbf{y}) = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ F_1/m \\ F_2/m \\ \vdots \end{pmatrix}$$

Dabei sind F_n die Federkräfte, die auf das entsprechende Pendel wirken.

2. Wir haben eine Programmvorlage für Sie vorbereitet. Laden Sie aus Stud.IP das Archiv

   `10_Pendel_GSL_Vorlage.zip`

   herunter und entpacken Sie es in Ihren CWR-Ordner. Das Projekt enthält auch ein `makefile`, das Sie zur Kompilierung verwenden können. Momentan sind jedoch noch Syntax-Fehler vorhanden. Achten Sie auf `/* TODO */`-Kommentare, die die Stellen markieren, an denen das Programm noch nicht vollständig ist.

   Machen Sie sich mit der Struktur von `pendulums.c` vertraut.

3. Definieren Sie im Kopfbereich die physikalischen Konstanten.

4. Schreiben Sie die Funktion für $\mathbf{F}(\mathbf{y})$ in folgender Form:

   ```
   int pendulums_ode(double t, const double y[],
                     double f[], void *params)
   {
      ...
      return GSL_SUCCESS;
   }
   ```

   Der Rahmen der Funktion ist bereits in `pendulums.c` vorgegeben. Hierbei sind `t` die aktuelle Systemzeit und `y[]` ist ein Array mit dem Systemzustand $\mathbf{y}$.

   Benutzen Sie `y[]` um alle auftretenden Ableitungen zu berechnen und schreiben Sie dann die korrekten Terme in `f[]`. Anschließend wird der Integer `GSL_SUCCESS` zurückgegeben, um dem Integrator den Erfolg der Berechnung zu signalisieren.

   Wichtig: Schreiben Sie `pendulums_ode` so, dass die Funktion auch für ein einzelnes Pendel mit $N = 1$ funktioniert!

5. Gehen Sie durch die `main`-Funktion und korrigieren Sie die Ausdrücke mit `???`. Folgen Sie den Anweisungen in den Kommentaren, um alles korrekt zu initialisieren und die Startbedingungen zu setzen.

6. Im Programm finden Sie Befehle, die mit `gsl_` beginnen. Lesen Sie sich die Kommentare zu den Befehlen durch. Am wichtigsten ist für Sie der `step_apply`-Befehl, der einen Zeitschritt auf das Array `y[]` anwendet. Dabei wird zuvor im Code Ihre `pendulums_ode()`-Funktion als Ableitung übergeben.

   Der `step_apply`-Befehl wird in einer `while`-Schleife wiederholt aufgerufen, bis die gewünschte Simulationszeit erreicht wurde. Fügen Sie am Ende der Schleife Routinen hinzu, mit denen Sie die Positionen der Pendel zusammen mit der aktuellen Zeit im CSV-Format in eine Datei schreiben. Verwenden Sie die schon geöffnete `pos_file`-Datei.

7. Kompilieren Sie das Programm und lassen Sie die Simulation laufen.

8. Plotten Sie die Pendelpositionen gegen die Zeit und reproduzieren Sie den Graph aus der obigen Abbildung.

9. Schreiben Sie die Funktion

`double pendulums_energy(const double y[])`

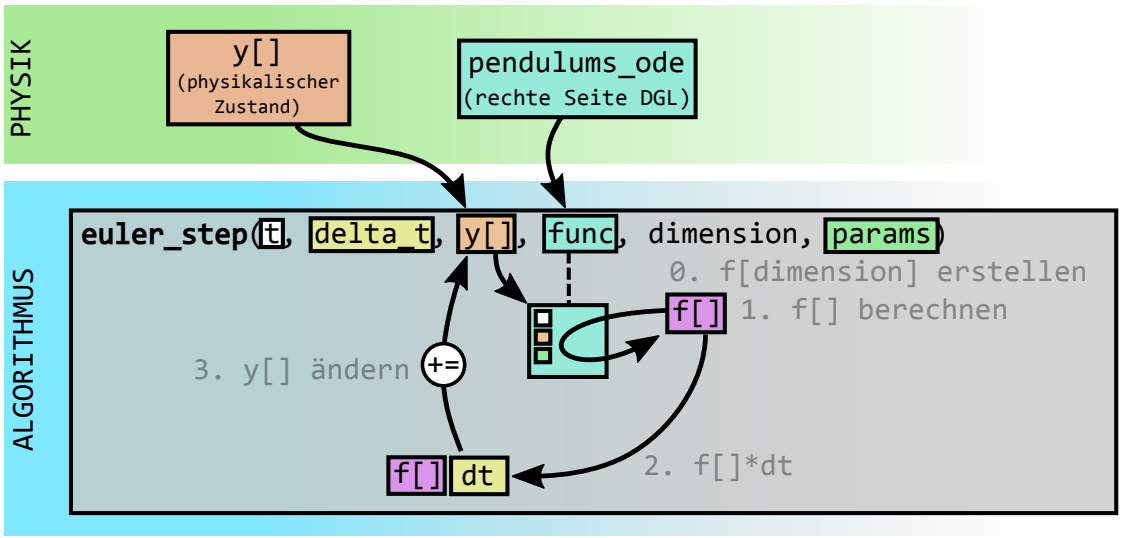
Sie finden die Deklaration unterhalb von `pendulums_ode` – erweitern Sie sie zu einer Definition. Die Funktion soll die Gesamtenergie des Pendelsystems berechnen. Dazu erhält sie den Zustandsvektor `y[]`. Das `const` sorgt dafür, dass der Inhalt von `y[]` innerhalb von `pendulums_energy` schreibgeschützt ist. Denken Sie an alle Federspann-Energien (auch die einzelne Feder am Ursprung) und die kinetischen Energien der Pendel.

10. Schreiben Sie in der Simulationsschleife die Zeit `t` und die Systemenergie CSV-formatiert in `energy_file`.
11. Simulieren Sie das System und plotten Sie den Verlauf der Energie. Der RK4-Algorithmus sollte für kleine Zeitschritte eine hinreichend stabile Energie ausgeben.

Aufgabe 11 *Euler-Integration (Pendel)*

Nachdem Sie die Pendel-Simulation in Aufgabe 10 mit einer vorgefertigten *Blackbox* durchgeführt haben, werden Sie nun einen eigenen Simulationsalgorithmus schreiben. Ziel ist es, die Funktion `gsl_odeiv2_step_apply` durch eine eigene Schritt-Funktion `euler_step` zu ersetzen.

(Sie haben gesehen, dass die GSL viel Verwaltungsfunktionalität bereit stellt, z.B. in Form der Funktion `gsl_odeiv2_step_alloc` oder der Stukturen `gsl_odeiv2_system` und `gsl_odeiv2_step`. Das müssen Sie hier nicht berücksichtigen, Sie sollen nur eine einzelne Funktion schreiben.)



1. Fügen Sie Ihrer Numerik-Bibliothek eine neue Funktion folgender Form hinzu:

```
typedef
int ode_func(double, const double[], double[], void*);

void euler_step(double t, double delta_t, double y[],
                ode_func func, int dimension, void *params)
```

```
{
    ...
}
```

(Den `typedef` können Sie in Ihren Header `my_numerics.h` schreiben. Sie müssen dann nicht mehr den gesamten Funktions-Pointer in die Funktions-Deklaration schreiben.)

`t` ist die aktuelle Zeit des Systems, `delta_t` ist die Größe des durchzuführenden Zeitschritts. `y[]` ist der aktuelle Zustandsvektor des Systems und hat die Größe `dimension`.

Die Funktion `func` ist die Systemfunktion, die die Ableitungen der Systemgrößen berechnet. `func` nimmt eine System-Zeit `t`, einen Systemzustand `y`, ein Zielarray `f` und eine Reihe von Parametern `params` entgegen und berechnet daraus die rechte Seite einer gewöhnlichen Differentialgleichung. Der Aufruf

```
func(t, y, f, params);
```

beschreibt also das Array `f[]` mit der zeitlichen Ableitung eines ODE-Systems. Ziehen Sie Ihre Implementierung von `pendulums_ode` zu Rate, falls Sie sich unsicher sind, wie ein konkretes Beispiel für `func` funktionieren könnte. `euler_step` soll aber ein vollständiger Ersatz für den GSL-Integrator sein, der auf beliebige Systeme angewandt werden kann! Die Grafik oben soll Ihnen verdeutlichen, wie der Integrationsalgorithmus und die (physikalische) Problemstellung voneinander getrennt werden.

Hinweis: Der `params` Pointer dient dann dazu, Parameter an die Systemfunktion `func` durchzugeben. Wir werden später sehen, wie das funktioniert. Wenn eine konkrete Funktion keine Parameter benötigt (wie etwa Ihre `pendulums_ode`), können Sie an `euler_step` einfach `NULL` übergeben.

2. Implementieren Sie in `euler_step` den expliziten Euler-Algorithmus, um den Zustandsvektor `y[]` zu aktualisieren. Am Ende des Dokuments finden Sie auch noch ein paar Hinweise zum Aufbau der Funktion.

3. Ersetzen Sie in Ihrem Hauptprogramm den Befehl `gsl_odeiv2_step_apply` durch einen Aufruf von `euler_step`. Benutzen Sie dafür den Befehl

   ```
   euler_step(t, delta_t, y,
              pendulums_ode, gsl_dimension, NULL);
   ```

   Da sie nicht mehr benötigt werden, sollten sie der besseren Übersicht halber die restlichen GSL-Befehle aus dem Programm entfernen.

4. Plotten Sie die Energie und die Positionen der mit dem Euler-Verfahren durchgeführten Simulation. Sie sollten ein bedeutend schlechteres Verhalten erkennen. Sie müssen den Zeitschritt möglicherweise deutlich verkleinern um überhaupt sinnvolle Ergebnisse zu erhalten, da das Euler-Verfahren leichter zu Instabilitäten neigt als der in der GSL-Routine angewandte Runge-Kutta-Algorithmus 4. Ordnung (RK4).

Aufgabe 12 *Stabilitätsanalyse (Prüfungsvorleistung)*

Hinweis: Beachten Sie bitte die Hinweise zur Abgabe auf der ersten Seite.

Das Pearl-Verhulst-Modell für das Wachstum von Populationen geht davon aus, dass das rein exponentielle Wachstum durch *limitierende Faktoren* begrenzt wird, so dass die Netto-Reproduktionsrate bei hohen Populationsgrößen effektiv abnimmt (entweder weil weniger geboren werden oder – z.B. durch Nahrungsmangel – mehr sterben). Die ODE des Modells hat die Form

$$\frac{dn}{dt} = r_0(1 - Kn(t))n(t). \tag{1}$$

Diese ODE wird auch Bernoullische Differentialgleichung genannt. Die Limitierung wird hier durch die Konstante K gesteuert. Für $K = 0$ erhält man ein *unbeschränktes* exponentielles Anwachsen der Population, das durch die Reproduktionsrate r_0 gesteuert wird. Der Parameter K heißt in der Ökologie-Literatur auch *Kapazität* des Lebensraums.

1. Zeigen Sie, dass die Lösung der Bernoullische DGL (1)

   $$n(t) = \frac{n_0 e^{r_0 t}}{1 + K n_0 \left(e^{r_0 t} - 1\right)} \tag{2}$$

   lautet, wobei $n_0 = n(t = 0)$ die Anfangsgröße der Population bezeichnet. Zeigen Sie, dass $n(t) \to 1/K$ für $t \to \infty$.

2. Bevor wir die numerische Lösung von Gl. (1) mittels Euler-Cauchy studieren, schreiben wir den darin vorkommenden Iterationsschritt um:

   $$n_{i+1} = n_i + \Delta t \cdot r_0 n_i(1 - K n_i) = (1 + \Delta t\, r_0) n_i \left(1 - \frac{\Delta t\, r_0 K}{1 + \Delta t\, r_0} n_i\right).$$

   Außerdem reskalieren wir gemäß

   $$x_i = \frac{\Delta t\, r_0 K}{1 + \Delta t\, r_0} n_i,$$

   und führen schließlich die Bezeichnung

   $$4\mu = 1 + \Delta t\, r_0$$

   ein. So nimmt der Euler-Schritt die Form

   $$x_{i+1} = 4\mu x_i(1 - x_i) \tag{3}$$

   an, die man als *logistische Abbildung* bezeichnet. Bei festem Δt nimmt μ mit wachsendem r_0 zu. Implementieren Sie Gl. (3).

   Hinweis: Natürlich können Sie auch Gl. (1) mit Hilfe Ihrer Implementation des Euler-Algorithmus aus Aufgabe 11 lösen. Wie oben gezeigt, ist das vollkommen äquivalent. Sie müssen dann lediglich zwischen (μ, x) und $(\Delta t, n)$ umrechnen, da wir im Folgenden von der Form (3) ausgehen und nur die reskalierten Größen (μ, x) verwenden.

3. Untersuchen Sie die Ergebnisse Ihrer numerischen Lösung, indem Sie x_i gegen $i = 0, \ldots, 100$ auftragen für die Schrittgrößen $\mu = 0.4$, 0.74 und 0.77. Wählen Sie Ihren Anfangswert x_0 klein genug, so dass stets $0 < x \leq 1$ gilt. Beschreiben Sie das Verhalten der drei Simulationen und vergleichen Sie dieses mit der analytischen Lösung Gl. (2).

4. Tragen Sie in einem zweiten Plot $x_{i=1000}$ gegen die Schrittgröße $\mu = 0.4 \ldots 1.0$ auf. Verwenden Sie dabei eine sehr feine Unterteilung für μ, um die komplexe Dynamik insbesondere für größere $\mu \lesssim 1.0$ auflösen zu können. Beschreiben Sie Ihre Resultate. Welche Aussage lässt sich über die Stabilität der implementierten numerischen Lösung treffen?

5. Führen Sie eine analytische Stabilitätsanalyse für den Iterationsschritt in Gl. (3) durch. Leiten Sie insbesondere unter Zuhilfenahme des Grenzwerts der analytischen Lösung das Stabilitätskriterium $\mu \leq 3/4$ her. Deckt sich dies mit Ihren Beobachtungen aus der vorherigen Teilaufgabe?

Selbsttest

- Wie formalisiere ich eine System gewöhnlicher Differentialgleichungen in Vektorschreibweise?
- Wie überführe ich ein DGL System 2. Ordnung in ein System 1. Ordnung?
- Was muss für die Ableitung einer Iterationsvorschrift gelten, damit der Algorithmus stabil ist (d.h. der Fehler nicht divergiert).

Tutorials

Aufgabe 10 - GSL

1. Machen Sie sich mit der Aufgabenstellung vertraut. Wie in Punkt 1.) angegeben, werden der Systemzustand $\mathbf{y}$ und die Zeitableitung $\mathbf{F}(\mathbf{y})$ als Vektoren aufgefasst.

   Wir verwenden hier die Konvention, dass $\mathbf{y}$ so aufgebaut ist, dass in den ersten N Einträgen die Positionen der N Pendel stehen und danach die N Geschwindigkeiten folgen. Damit ist das physikalische System zu einem Zeitpunkt vollständig durch $\mathbf{y}$ charakterisiert.

2. Laden Sie die Vorlage wie in 2) angegeben runter und bearbeiten Sie dann Aufgabe 3) der Übung.

3. Analysieren Sie die vier Parameter der gesuchten Funktion `pendulums_ode`. Innerhalb der Funktion stehen Ihnen die Zeit `t` und der Zustandsvektor `y[]` zur Verfügung, um die Kräfte zu berechnen. Außerdem natürlich alle globalen Variablen. Die "Kräfte" (bzw. Ableitungen) sollen dann in das Array `f[]` geschrieben werden. Sie dürfen davon ausgehen, dass `f` auf einen hinreichend großen Speicherbereich zeigt. Wir ignorieren hier den Pointer `params`. Da die Kräfte nicht explizit von der Zeit abhängen, werden Sie auch `t` nicht gebrauchen müssen.

4. Schreiben Sie zunächst eine Null in alle Einträge von `f[]`:

   ```
   for(int i = ; i < 2*N; ++i)
       f[i] = 0;
   ```

5. Die Ableitungen der Pendelpositionen sind jeweils ihre Geschwindigkeit. Schreiben Sie also in die ersten N Einträge von `f[]` die Geschwindigkeiten:

   ```
   for(int i = 0; i < N; ++i)
   ```

```
f[i] += y[N+i];
```

6. Nun zu den Ableitungen der Geschwindigkeiten, gegeben durch die Pendelkräfte und die Pendelmasse. Das erste Pendel ist mit einer Feder an den Ursprung gekoppelt. Berechnen Sie die resultierende Kraft und addieren Sie sie auf die Ableitung der ersten Geschwindigkeit auf:

   ```
   f[N] += -k*y[0]/mass;
   ```

7. Als letztes müssen noch die Federn zwischen den Pendeln berücksichtigt werden. Gehen Sie mit einer `for`-Schleife durch alle $N-1$ Federn durch. Berechnen Sie ihre tatsächliche Länge, die resultierende Kraft basierend auf der Ruhelänge und addieren Sie die Kraft mit richtigem Vorzeichen auf die entsprechenden Ableitungen in `f[]`. Achten Sie unbedingt darauf, dass die gesamte `pendulums_ode`-Funktion auch für ein Einzelpendel $N = 1$ funktioniert!

8. Bearbeiten Sie den Rest der Aufgabe, angefangen bei 5.).

Tipps

Aufgabe 11 - Expliziter Euler-Algorithmus

Der explizite Euler-Schritt aktualisiert den Zustandsvektor gemäß folgender Vorschrift:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \Delta t \cdot \mathbf{F}\left(\mathbf{y}_i\right)$$

1. Zunächst muss $\mathbf{F}\left(\mathbf{y}_i\right)$ mit der Funktion `func` berechnet werden. Erstellen Sie in `euler_step` einen Pointer `double *f`. Weisen Sie ihm mit `malloc` einen Speicherbereich zu, der groß genug ist, das Resultat von `func` aufzunehmen. Der Zustandsvektor `y[]`, sowie seine Ableitung `f[]` haben jeweils `dimension` Einträge.

2. Beschreiben Sie `f[]` mithilfe der Funktion `func`. Der Befehl hierfür ist bereits in der 1. Unteraufgabe angegeben, machen Sie sich aber klar, warum der Befehl funktioniert! Denken Sie daran, den `params`-Pointer an `func` weiterzureichen.

3. Aktualisieren Sie den Zustandsvektor gemäß der obigen Vorschrift. Das gelingt mit einer `for`-Schleife über alle Einträge von `y[]`.

4. Geben Sie den Speicher von `f` wieder frei.