Zusammenfassung Computersimulationen

Mladen Ivkovic
mladen.ivkovic@uzh.ch

Mai 2014

1 Algorithmen

1.1 Nullstellen finden

1.1.1 Allgemeines

- Wir suchen die Nullstelle einer Funktion $f(x)$. Die zu erfüllende Bedingung ist offensichtlich $f(x) = 0$. Die Nullstellensuche ist u.a. nützlich, da eine Funktion $g(x) = b$ in $f(x) \equiv g(x) - b$ umgeschrieben werden kann und so beliebige Werte b einer Funktion gesucht werden können.
- Der einfachste Weg wäre es, $f(x)$ graphisch darzustellen und die Nullstelle von Auge zu suchen, um darauf das Plot-Intervall für die $x-$Werte anzupassen. Ein anderer Weg[1] wäre es, alle x-Werte abzusuchen und zu Prüfen, für welches x die Funktion $y = 0$ herausgibt.[2]
- Ist $f(x)$ mehrdimensional, haben wir ein $N \times N$ Gleichungssystem, welches keine, eine, mehrere oder unendlich viele Nullstellen besitzen kann.
- Bei Interationsmethoden ist der Erfolg von der Funktion selber, von der Anzahl Nullstellen und vom Startwert x_0 abhängig.

1.1.2 Bisektions-Methode

- Die Bisektions-Methode basiert auf dem Zwischenwertsatz.[3] Daraus folgt, dass wenn $f(a)$ und $f(b)$ verschiedene Vorzeichen haben, es mindestens eine Nullstelle in $[a, b]$ hat.
- Man berechnet nun den Wert von $f(x)$ in der Mitte des Intervalls $[a, b]$, $f\left(\frac{a+b}{2}\right)$ und ersetzt damit a oder b so, dass 0 immer noch dazwischen ist. (Man ersetzt also dasjenige, welches das selbe Vorzeichen wie $f\left(\frac{a+b}{2}\right)$ hat.) Programmbeispiel: Abb. 3.1.
- Die Bisektionsmethode ist robust, konvergiert immer und die Konvergenzrate ist bekannt. Jedoch konvergiert sie relativ langsam: Sie hat eine 'Lineare Konvergenz', d.h. mit jedem Iterationsschritt gewinnt sie eine signifikante Binärziffer dazu.
- **Varianten**
  - **Regula Falsi**: Statt $\frac{a+b}{2}$ zu berechnen, wird eine Gerade von $(a, f(a))$ nach $(b, f(b))$ gezogen. Die Nullstelle dieser Gerade wird als neuer Punkt a oder b gewählt, so dass die Nullstelle immer noch im Intervall liegt.[4]

[1] Ich nenne es die Brute-Force-Methode

[2] Da wir nicht analytisch, sondern numerisch arbeiten, muss berücksichtigt werden, dass die Funktion wohl selten genau null sein wird. Insofern muss $y = 0 \pm \epsilon$ gelten, wobei das ϵ genügend klein, und dennoch genügend gross gewählt werden muss. Bei Java wäre das etwa $\epsilon \approx 10^{-14}$.

[3] Eine reelle, stetige Funktion $f\ [a, b] \to \mathbb{R}$, $x \mapsto f(x)$ nimmt jeden Wert zwischen $f(a)$ und $f(b)$ an.

[4] Vgl. `http://upload.wikimedia.org/wikipedia/commons/thumb/9/99/Regula_falsi.gif/759px-Regula_falsi.gif`

– **Sekantenverfahren**: Wie bei der Regula-Falsi-Methode werden die Nullstellen von Sekanten zwischen zwei Punkten verwendet. a und b werden aber nicht mehr wahlweise ersetzt, sondern es werden immer nur die neusten zwei Punkte verwendet. Es kann hier vorkommen, dass die Nullstelle für einige Iterationsschritte nicht mehr zwischen den Punkten x_n und x_{n+1} liegt.

1.1.3 Newton-Raphson-Verfahren

- Beim Newton-Raphson-Verfahren gehen wir von einem Startwert x aus. Nun berechnen wir $f(x)$ und $f'(x)$.[5] Das neue x' ist: $x' = x - \frac{f(x)}{f'(x)}$. Widerhole, bis $f(x')$ genügend klein. Programmbeispiel: Abb. 3.2.

- Erklärung: Gesucht ist x_{n+1}, welches näher bei der Nullstelle ist als x_n. Dazu wird $f(x)$ linearisiert, indem die Tangente am Punkt $(x_n, f(x_n))$ bestimmt wird. Diese hat die Gleichung: $t(x) = f(x_n) + f'(x_n)(x - x_n)$. Wir wählen dann x_{n+1} als einzige Nullstelle dieser Gleichung: $0 = t(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n)$. Daraus folgt: $x_{n+1} = x_n - \frac{f(x)}{f'(x)}$.

- Das Newton-Raphson-Verfahren ist schneller als die Bisektionsmethode: Sie konvergiert 'quadratisch', sprich es verdoppeln sich die Anzahl Stellen bei jeder Iteration. Jedoch kann es passieren, dass (sollte $f'(x) = 0$ sein) $x_{n+1} \to \infty$ strebt, oder es kann in einen ewigen loop hineinfallen.

1.2 Lösung von Anfangswertproblemen

1.2.1 Euler-Verfahren

- Das Euler-Verfahren ist das einfachste Verfahren zur numerischen Lösung eines Anfangswertproblems. Gegeben sei die Gleichung $\dot{y} = f(y,t)$ [6] mit $y(t_0) = y_0$. Wir wählen nun eine Schrittweite $\Delta t > 0$. Mit dieser Schrittweite können die diskreten Zeitwerte $t_k = t_0 + k \cdot \Delta t$ berechnet werden, wobei $k = 0, 1, 2...$ die Schrittnummer ist. Mit diesen berechnen wir die diskret angenäherten Werte für y: $y_{k+1} = y_k + \Delta t \cdot f(y_k, t_k)$

- Die Grundidee ist, y 'grob zu integrieren'. Hier wird die Näherung $f(y,t) = \frac{dy}{dt} \approx \frac{\Delta y}{\Delta t}$ gemacht und daraus $\Delta y = f(y,t) \cdot \Delta t$, wobei wir bei einem $y = y_0$ anfangen und uns zum gesuchten $y_k = y_0 + k \cdot \Delta y$ 'aufintegrieren'. [7] Ist einem noch die Treppenfunktions-Herleitung von (Riemann-) Integralen noch präsent, ergibt diese Annäherung für genügend kleine Δt Sinn.

- Das Euler-Verfahren ist ein Einschrittverfahren. Einschrittverfahren benutzen immer nur den zuletzt berechneten Wert, um auf den nächsten zu schliessen. (Zur Berechnung von y_{k+1} benutzen wir nur y_k.) Je kleiner Δt ist, desto genauer wird die Approximation, jedoch ist mehr Rechenarbeit nötig, das Verfahren wird langsamer. Die Abweichungen von den exakten Werten 'pflanzen sich fort': Der Fehler auf dem approximierten Wert steigt mit jedem Rechnungsschritt. Die Unsicherheit liegt in der Grössenordnung von $\approx \Delta t^2$.

[5] Ist $f'(x)$ unbekannt, kann es mit $\approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$ angenähert werden.
[6] WICHTIG! $f(y,t) = \frac{dy}{dt}$!!!
[7] Diese Gleichung soll zur Veranschaulichung dienen, ist aber so nicht korrekt, denn bei jedem Iterationsschritt wird in der Funktion $f(y_k, t_k)$ ein neues y_k und t_k eingesetzt.

1.2.2 Runge-Kutta-Verfahren

- Das Runge-Kutta-Verfahren ist auch ein Einschrittverfahren zur approximation von gewöhnlichen Differentialgleichungen. Anders als beim Euler-Verfahren verwendet man s Zwischenschritte zwischen den zwei Stützstellen t_n und t_{n+1}. Nach Anzahl Zwischenschritte s wird das jeweilige Verfahren als s-stufig bezeichnet. (Das Euler-Verfahren wäre ein Runge-Kutta-Verfahren erster Ordnung.)

- **Beispiele**
  Für die nachfolgenden Beispiele gelten immer die gleichen Ausgangslagen: Sei $y' = f(y,t)$ und $y(t_0) = y_0$, die Schrittweite sei Δt, n die Schrittzahl und k_s der s-te Zwischenschritt. Programmbeispiele sind in den Abbildungen 3.3, 3.4 und 3.5.

  – **Das zweistufige Runge-Kutta-Verfahren**:

$$\begin{aligned} k_1 &= f(y_n, t_n) \\ k_2 &= f\left(y_n + \tfrac{\Delta t}{2} \cdot k_1, t_n + \tfrac{\Delta t}{2}\right) \\ y_{n+1} &= y_n + \Delta t \cdot k_2 \end{aligned}$$

  Der Fehler liegt in der Grössenordnung von $\approx \Delta t^3$.

  – **Das vierstufige Runge-Kutta-Verfahren**:

$$\begin{aligned} k_1 &= f(y_n, t_n) \\ k_2 &= f\left(y_n + \tfrac{\Delta t}{2} \cdot k_1, t_n + \tfrac{\Delta t}{2}\right) \\ k_3 &= f\left(y_n + \tfrac{\Delta t}{2} \cdot k_2, t_n + \tfrac{\Delta t}{2}\right) \\ k_4 &= f\left(y_n + \Delta t \cdot k_3, t_n + \Delta t\right) \\ y_{n+1} &= y_n + \Delta t \cdot \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned}$$

  Der Fehler liegt in der Grössenordnung von $\approx \Delta t^5$.

- Die Runge-Kutta-Methoden brauchen mehr Rechenschritte als die Euler-Methode, sind jedoch genauer.

1.2.3 Leapfrog-Verfahren

- Das Leapfrog-Verfahren ist eine Methode zur numerischen Integration von Differentialgleichungen vom Typ $\ddot{x} = \dot{v} = f(x)$. Auch beim Leapfrog-Verfahren werden Zwischenwerte berücksichtigt. Dies kann geschehen, indem die Momentangeschwindigkeit v_n zum Zeitpunkt t_n nicht für einen ganzen, sondern einen halben Zeitschritt gelten lässt. So bekommt man die Position zum Zeitpunkt $t_n + \frac{\Delta t}{2}$. Aus diesen Zwischenwerten können wir auf genauere Werte für die Positionen und Geschwindigkeiten zum Zeitpunkt $t_n + \Delta t$ schliessen. Um von einer alten zu einer neuen Position zu gelangen, arbeitet man nun mit dem Mittel der alten und neuen Momentangeschwindigkeiten bzw. Positionen.[8]

[8] Zur Verdeutlichung: `http://upload.wikimedia.org/wikibooks/de/0/02/Das_Mehrk%C3%B6rperproblem_in_der_Astronomie_Prinzip_des_Leapfrog-Verfahrens.png`

- Die Berechnung der Geschwindigkeitsänderung nennt man **Kick**, diejenige der Position **Drift**. Es ist möglich, die Zwischenpositionen über die Geschwindigkeit (Kick-Drift-Kick), aber auch über die Position (Drift-Kick-Drift) zu berechnen. Hier die Formeln:

"Drift-Kick-Drift" version

$$\begin{aligned} x_{n+\frac{1}{2}} &= x_n + v_n \frac{\Delta t}{2} \\ v_{n+1} &= v_n + f(x_{n+\frac{1}{2}})\Delta t \\ x_{n+1} &= x_{n+\frac{1}{2}} + v_{n+1}\frac{\Delta t}{2} \end{aligned}$$

"Kick-Drift-Kick" version

$$\begin{aligned} v_{n+\frac{1}{2}} &= v_n + f(x_n)\frac{\Delta t}{2} \\ x_{n+1} &= x_n + v_{n+\frac{1}{2}}\frac{\Delta t}{2} \\ v_{n+1} &= v_{n+\frac{1}{2}} + f(x_{n+1})\frac{\Delta t}{2} \end{aligned}$$

- Die Leapfrog-Integration liefert i.A. genauere Ergebnisse als das Euler-Verfahren. Es ist invariant unter Zeitumkehr und erhält exakt Erhaltungsgrössen wie den Impuls und Energie. Die Verwendung des Mittels der Momentangeschwindigkeiten zu Anfang und Ende eines Zeitschritts bedeutet, dass man die wahre Fläche unter der Geschwindigkeitskurve nicht mehr durch Rechtecke, sondern Trapeze ersetzt.[9] Das Problem am Leapfrog-Verfahren ist aber, dass es neben x_0 und v_0 noch x_1 oder v_1 zum Anfangen braucht. (Dies kann zwar durch Euler/Runge-Kutta berechnet werden, der Rest der Simulation ist aber sehr stark von diesem Anfangswert abhängig.) Das Leapfrog-Verfahren hat einen Fehler von der Grössenordnung $\approx \Delta t^2$.

1.3 Weitere Verfahren

1.3.1 Ableitung

- Näherung für Ableitung:

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1/2} - u_{i-1/2}}{\Delta x} \qquad \text{oder} \qquad \frac{u_i - u_{i-1}}{\Delta x}$$

- Für die zweite Ableitung kann man dann folgende Näherung wählen:

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$$

[9] Siehe: http://upload.wikimedia.org/wikibooks/de/2/25/Das_Mehrk%C3%B6rperproblem_in_der_Astronomie_Strecke_und_momentane_Geschwindigkeit_im_Leapfrog-Verfahren.png

2 Simulationen

2.1 Game of Life

- **Aufgabe**
  Wir simulieren das Leben von Zellen über einem Gebiet. Vereinfacht betrachten wir das Gebiet als ein Gitter. Die Zellen haben genau zwei Existenzmöglichkeiten: 'Lebt' und 'Tot'. Die Auswertung geschieht über einzelne Zeitschritte, welche für alle Zellen gelten. Die Spielregeln sind die folgenden:
  1. Eine lebende Zelle mit weniger als 2 lebenden Nachbarn stirbt im nächsten Zeitschritt.
  2. Eine lebende Zelle mit 2 oder 3 lebenden Nachbarn lebt weiter im nächsten Zeitschritt.
  3. Eine lebende Zelle mit mehr als 3 Nachbarn stirbt im nächsten Zeitschritt.
  4. Eine tote Zelle mit genau 3 Nachbarn lebt im nächsten Zeitschritt.

- **Vorgehen**
  - Man definiere das Gitter mit $x \times y$ Zellen. (z.B. als Array (nennen wir es $grid1$) bestehend aus y weiteren Arrays, welche jeweils x Elemente enthalten, und behandeln dieses als Gitter mit y Zeilen und x Spalten. Die Position der Zelle Z(2,3) wäre demnach $grid1[y-2][3]$.)
  - Man definiere die Zustände 'Lebt' und 'Tot' mit 1 = lebt, 0 = tot.
  - Als Startbedingung können z.B. bestimmte Zellen als lebend gesetzt werden, oder man lässt den Zufall entscheiden. (z.B. ein Durchlauf mit *if random_int < 0.1, grid1[i][j] = 1*)
  - Durchlauf jedes Zeitschrittes:
    * Man definiere ein neues Gitter, $grid\_new$, mit derselben Anzahl Zellen.
    * Man setze den Rand beider Gitter (Alle Zellen mit $i = 0$ oder $i = x$ oder $j = 0$ oder $j = y$) als tot. (Die x-Koordinate wird mit i, die y-Koordinate mit j bezeichnet.)
    * Für alle i mit $1 \leq i \leq x - 1$, tue folgendes: Für alle j mit $1 \leq j \leq y - 1$, zähle die Werte der Nachbarn (Insgesamt 8 Nachbarn) zusammen. Dann wende man die obigen Regeln (einfache if-Prüfung) an. Z.B: Falls $grid1[i][j] =$ tot und $neighbours = 3$, $grid\_new[i][j] =$ lebt.
    * Wichtig: Die Änderungen im Gitter, die nun erfolgen, müssen im $grid\_new$ abgespeichert werden, um die Berechnungen für $grid1$ nicht zu stören.
    * Man plotte nun das neue Gitter $grid\_new$, und speichere es als $grid1$ ab, damit diese neue Situation im nächsten Durchlauf die Anfangsbedingungen sind.

2.2 Fluiddynamik und Gitter-Gas-Modelle

- **Aufgabe**
  Wir wollen Fluiddynamik numerisch behandeln. Die Standard-Methode wäre, die Navier-Stokes-Gleichungen[1] zu diskretisieren (finite volume method) und numerisch zu integrieren. Eine Alternative wäre es, mikriskopische Modelle basierend auf stark vereinfachten Gasteilchen-Kollisionen zu betrachten. Die Hoffnung dabei ist, im Mittel über viele Teilchen die Navier-Stokes-Gleichung mit einem einfacheren, schnelleren, effizienteren Verfahren zu lösen. Beim **HPP Gitter-Gas-Modell**[2] brauchen wir ein 2-dimensionales, orthogonales Gittermodell. Die Regeln dabei sind:

  1. Es gibt 4 erlaubte Geschwindigkeiten: N,S, E, W
  2. Pro Gitterzelle dürfen 0 oder 1 Teilchen pro Geschwindigkeit (also 0-4 Teilchen pro Zelle)
  3. Ein Zeitschritt besteht aus Propagation und Kollision
  4. Regeln für Kollision: Bei horizontaler Kollision (W und E) gehen die Teilchen vertikal auseinander (N und S). Bei vertikaler Kollision (N und S) gehen die Teilchen horizontal auseinander (W und E).
  5. Der Rand soll die Teilchen reflektieren.

  Simuliert werden soll die Teilchenbewegung und die Teilchendichte (Anzahl Teilchen in Zelle).

- **Vorgehen**
  - Wir benutzen bitweise Operationen für Geschwindigkeit und Speicheroptimierung. Wir definieren jede Geschwindigkeit als einen anderen Bit:

Geschw.	Bits	Integer
E	00001	1
N	00010	2
W	00100	4
S	01000	8
Rand	10000	16

    Nicht am Rand (<16) gilt für die Kollisionen: Aus 00101 (5 als int, EW) wird 01010 (10 als int, NS) und umgekehrt. Am reflektierenden Rand ($\geq$ 16) gilt: Aus 10001 wird 10100 und umgekehrt, aus 10010 wird 11000 und umgekehrt.
  - Definiere $x \times y$ Gitter *grid*1 wie beim Game of Life für den momentanen Zustand. Dann ein weiteres *grid*2 für die Berechnung der Bewegung nach einem Zeitschritt und ein weiteres *count* für die Dichte.
  - Initialisierung beispielsweise durch Zufall. Setze in *grid*2 und *count* alle Zellen auf 0.
  - Setze den Rand des Gitter als Rand mit Wert 16.
  - Führe Bewegung eines Zeitschrittes aus: Für jede Geschwindigkeit, prüfe ob Zelle *ij* von *grid*1 diese Geschwindigkeit enthält. Falls ja, hat im nächsten Zeitschritt die

[1] $\rho \frac{D\mathbf{v}}{Dt} = -\nabla p + \nabla \cdot \mathbf{T} + \mathbf{f}$
[2] Hardy, Pomeau und de Pazzis, 1973

Zelle, in der das Teilchen hingewandert wäre, nun ein Teilchen mit dieser Geschwindigkeit. Trage dieses in $grid2$ und $count$ ein. ($count$ braucht keine Geschwindigkeit, sondern nur Teilchenzahl.) Beispiel: $if((grid1[i][j]\&1) == 1)\{grid2[i][j+1]| = 1; count[i][j+1] += 1;\}$. Setze danach $grid1$ auf 0.

- Führe am $grid2$ die Randbedingungen aus: 16 bleibt 16, aus 17 wird 20 und umgekehrt, aus 18 wird 24 und umgekehrt. Danach Führe Kollisionsbedingungen aus: Aus 5 wird 10 und umgekehrt.
- $grid1 = grid2$ setzen. Diesen Zeitschritt graphisch darstellen. Wiederhole Verfahren, bis zufrieden.

- **Anmerkungen**
  Die Vorteile dieses Verfahrens sind dass es exakt und deterministisch ist, es ist kompakt (5 bits pro Gitterzelle reichen aus) und es ist schnell. Die Nachteile sind dass es stark anisotrop ist und statistisches Rauschen enthält. Eine Weiterführung dieses Modells wäre das FHP-Modell bit gleichseitigen Dreiecken und 6 Geschwindigkeiten.

2.3 Gewöhnliche Differentialgleichungen

2.3.1 SIR (Susceptible Infected Recovered) - Modell

- **Aufgabe**
  Wir betrachten die Ausbreitung einer Krankheit in einer konstanten Population P. Anfällige (Susceptibles S) können mit einer Infektionsrate $\frac{dI}{dt} = -\frac{dS}{dt} - \frac{dR}{dt}$ angesteckt werden. Die Infizierten I können mit einer Genesungsrate $\frac{dR}{dt} = a \cdot I$ genesen, wobei a die Genesungszeit bzw. Wahrscheinlichkeit ist. Für die Anfälligen S gilt: $\frac{dS}{dt} = -r \cdot S \cdot I$. Die genesene Population ist immun und kann nicht wieder angesteckt werden. Konstant bleibt $P = I + R + S$.

- **Vorgehen**
  - Man wähle Anfangsbedingungen für I, R, S, r und a.
  - Man löse mit dem (Euler, aber genauer mit der) Runge-Kutta-2-Methode die Anfangswertprobleme:

$$\frac{dS}{dt} = -r \cdot S \cdot I$$
$$\frac{dR}{dt} = a \cdot I$$
$$\frac{dI}{dt} = -\frac{dS}{dt} - \frac{dR}{dt}$$

    in dieser Reihenfolge. Die Resultate nach jedem Zeitschritt speichere man in einem (oder 3) Arrays.
  - Arrays plotten bei Bedarf.

2.3.2 Jäger-Beute-Modell / Lotka-Volterra-Modell

- **Aufgabe**
  Beim Jäger-Beute-Modell werden die Populationen von Räubern (Falken h) und Beute

(Eichhörnchen s) betrachtet. Gibt es genug wenige Räuber, vergrössert sich die Eichhörnchenpopulation, da sie sich schneller vermehren als aussterben. Mit steigender Eichhörnchenpopulation steigt aber auch die Falkenpopulation, denn mehr Nahrung ist vorhanden und mehr Falken überleben. Das geht so weiter, bis die Falkenpopulation gross genug ist, um zu viele Eichhörnchen zu fressen, so dass die Eichhörnchenpopulation sinkt. Mit ihr sinkt auch die Falkenpopulation, da wieder weniger Nahrung vorhanden ist. Das geht so weiter, bis die Falkenpopulation wieder klein genug ist, dass sich die Eichhörnchenpopulation wieder vergrössern kann. Dabei gelten die Gleichungen:

$$\frac{ds}{dt} = s(k_s - k_f \cdot h) \qquad \frac{dh}{dt} = h(k_h \cdot s - k_d)$$

wobei die Konstante k_s ist Reproduktionsrate der Eichhörnchen, k_d ist die die Sterberate der Falken, wenn keine Nahrung vorhanden ist, k_f die Rate, mit der Eichhörnchen gefressen werden (Sterberate pro Räuber) und k_h die Reproduktionsrate der Falken pro Beutelebewesen.

- **Vorgehen:** Löse Anfangswertproblem mit Euler oder besser Runge-Kutta, Resultate nach jedem Zeitschritt in einem Array abspeichern, bei Belieben plotten.

2.4 Daisyworld

- **Aufgabe**
  Ziel dieser Übung ist die Simulation von Daisyworld. Daisyworld ist eine Welt, in welcher der Planet nur aus weissen Daisies[3], schwarzen Daisies und barer Erde besteht. Die Sonne strahlt auf die Erde ein und überträgt somit Energie. Die Einfallende Energie E_{in} ist $E_{in} = l \cdot c_s$ mit l als Luminositätsfaktor (Welcher Teil der Sonneneinstrahlung auf die Erde einwirkt, zwischen 0.6 und 1.8) und mit der konstanten Solarkonstante c_s (mittlere Sonnenbestrahlungsintensität, 1370W/m^2). Von dieser einfallenden Energie E_{in} wird ein Teil von der Erde reflektiert: $E_r = E_{in} \cdot A_{tot}$. A_{tot} nennt sich Albedo und ist der Faktor der einfallenden Strahlung, die reflektiert wird. Die absorbierte Energie beträgt also: $E_{abs} = E_{in} - E_r = (1 - A_{tot})E_{in}$.
  Der Albedo setzt sich folgendermassen zusammen: weisse Daisies, schwarze Daisies und bare Erde haben einen eigenen Faktor: $A_w = 0.75$, $A_e = 0.5$, $A_b = 0.25$. Der gesamte Albedo A_{tot} setzt sich zusammen aus dem Anteil der einzelnen Flächen und dem zugehörigen Albedo: $A_{tot} = f_w \cdot A_w + f_e \cdot A_e + f_b \cdot A_b$, wobei f_w, f_e und f_b die Anteile der von der Erde, schwarzen und weissen Daisies eingenommenen Fläche sind. Offensichtlich muss gelten: $f_w + f_e + f_b = 1$.
  Damit ein Gleichgewicht eintritt, muss gelten: Der Planet absorbiert und emittiert die gleiche Energie. So schliessen wir auf die Erdtemperatur: $E_{abs} = E_{em} = \sigma \cdot T^4$. σ ist die Stefan-Boltzmann-Konstante, (den Emissionsfaktor ϵ und die Fläche A setzen wir auf 1). Daraus folgt: $T = \left(\sqrt{\frac{l \cdot c_s \cdot (1 - A_{tot})}{\sigma}}\right)^{1/4}$. Die Temperatur bestimmt die Wachstumsrate der Daisies (schwarze und weisse). Es gilt für beide Daisies die Flächenänderung: $\frac{\partial f_i}{\partial t} = f_i \cdot (f_e \cdot g(T) - d)$. d ist die Sterberate, $g(T)$ der Temperaturabhängige Wachstumsfaktor: $g(T) = 1 - 0.003265 \cdot (22.5 - T)^2$. [4]

[3] engl. für Gänseblümchen
[4] Gemeint sind hier 22.5°C

Zu berechnen sind das Wachstum der Daisies und die Temperatur, nachdem sich das Gleichgewicht eingestellt hat.

- **Vorgehen**
  - Man wähle die Anfangsbedingungen und definiere die Funktionen für T und für $g(T)$ und A_{tot}.
  - Für jeden Iterationsschritt: Mit den momentanen Flächen berechne man den Albedo A_tot. Damit berechne man die Temperatur T und mit ihr $g(T)$. Mit $g(T)$ berechne man die Flächenänderung mit der Runge-Kutta-Methode. Hiermit wäre ein Zeitschritt getan.
  - Es ist möglich, den Luminositätsfaktor l zu variieren. (Auch für verschiedene l ganze Durchläufe machen...)

2.5 Planetenbahnen

- **Aufgabe**
  Ziel dieser Simulation ist die Berechnung von Planetenbahnen.
  Seien n Planeten mit Masse M_i, Anfangsposition (x_i, y_i, z_i) und Anfangsgeschwindigkeiten (v_{xi}, v_{yi}, v_{zi}) gegeben. Die auf den Planeten p wirkende Beschleunigung ist die Summe der Gravitationsbeschleunigung aller anderen Planeten: $\ddot{x} = a_p = \sum_{i=1, i\neq p}^{n} \frac{G \cdot M_i}{|r_i|^3} \cdot \vec{r_i}$. G ist die Gravitationskonstante, M_i die Masse des anderen Planeten und $\vec{r_i}$ der Abstand zwischen den zwei Planeten: $\vec{r_i} = \vec{x_p} - \vec{x_i}$, wobei x_p und x_i die Ortsvektoren der beiden Planeten sind.

- **Vorgehen**
  - Pro Zeitschritt Leapfrog-Verfahren für alle Planeten, alle Orte abspeichern. Wiederholen.

2.6 (Reaktion-) Diffusionsgleichungen

- **Aufgabe**
  Die Diffusionsgleichung ist $\frac{\partial u}{\partial t} = D \cdot \frac{\partial^2 u}{\partial x^2} \approx D \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$.
  Zur Beschreibung von Reaktionen gilt die Fisher-Kolmogorov-Gleichung: $\frac{\partial u}{\partial t} = D \cdot \frac{\partial^2 u}{\partial x^2} + u(1-u) \approx D \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + u(1-u)$.
  Das Gierer-Meinhart-Modell für Reaktionsgleichungen ist:

$$\frac{\partial a}{\partial t} = p \cdot \frac{a^2}{h} - M_a \cdot a + D_a \cdot \frac{\partial^2 a}{\partial x^2} + p_0 \qquad \frac{\partial h}{\partial t} = p \cdot a^2 - M_h \cdot h + D_h \cdot \frac{\partial^2 h}{\partial x^2}$$

- **Vorgehen**
  - Approximation $\frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$ für zweite Ableitung, Runge-Kutta für erste verwenden.

2.7 Logistische Gleichungen

Die logistische Gleichung wird dazu gebraucht, um chaotisches Verhalten zu simulieren. Eine einfache Form der Gleichung ist $x_{n+1} = r \cdot x_n(1 - x_n)$. Ihr Verhalten ist stark von r abhängig.

Ein Beispiel wäre mit $x_n = sin^2(\Theta_n)$. Dadurch ergibt sich für $r = 4$: $x_{n+1} = sin^2(\Theta_{n+1}) = 4x_n(1 - x_n) = 4sin^2(\Theta_n)(1 - sin^2(\Theta_n)) = 4sin^2(\Theta_n)cos^2(\Theta_n) = sin^2(2\Theta_n)$. Das Chaos kommt bei der Definition des Θ_{n+1} ins Spiel: $\Theta_{n+1} = 2\Theta_n mod(2\pi)$. Das $mod(2\pi)$ gibt nur den Rest bei einer Division durch 2π an, was die Rechnung einfacher macht, da $sin(x)$ ja 2π-periodisch ist. Einfach gesagt verdoppeln wir den Winkel Θ bei jeder Iteration und schauen, was dabei rauskommt.
Eine andere Möglichkeit zur 'Chaos-Erstellung' wäre es, eine irrationale Zahl z (z.B. $\sqrt{2}$, π, e...) zu wählen, diese bei jeder Iteration mit 10 zu multiplizieren und den Integer-Anteil abzuschneiden. So 'verschieben' wir die keiner Symmetrie folgenden Nachkommastellen immer um eine weitere Stelle nach links: $y_0 = z$, $y_{i+1} = 10 \cdot y_i - int(10 \cdot y_i)$.

2.8 Julia-Menge und Mandelbrot-Menge

Die Mandelbrot-Menge ist die Menge aller komplexen Zahlen c, für welche die Reihe $z_0 = c$, $z_{n+1} = z_n^2 + c$ gilt. Sobald $|z_i| \geq 2$, kann man Konvergenz ausschliessen. (D.h. bei jedem Schritt prüfen, ob $|z_n| \geq 2$. Falls ja, nicht abspeichern.)
Ist z_0 gegeben und c der Parameter, nennt man die Menge Mandelbrot-Menge. Ist der Anfangswert z_0 und c gegeben, so nennt man die Menge Julia-Menge.

3 Grobzusammenfassung

3.1 Algorithmen

3.1.1 Nullstellen

- **Bisektions-Methode**
  - Sei $f(a)$ und $f(b)$ mit unterschiedlichen Vorzeichen. Berechne $f\left(\frac{a+b}{2}\right)$ und ersetze a oder b so, dass Nullstelle immer noch dazwischen liegt, bzw. ersetze dasjenige mit dem selben Vorzeichen wie $f\left(\frac{a+b}{2}\right)$. Wiederhole bis $|f\left(\frac{a+b}{2}\right)| \leq \epsilon$
  - **Vorteile:** robust, konvergiert immer, Konvergenzrate bekannt.
  - **Nachteile:** konvergiert rel. langsam, Lineare Konvergenz (Eine Binärziffer pro Interationsschritt)
- **Newton-Raphson-Verfahren**
  - Beim Newton-Raphson-Verfahren gehen wir von einem Startwert x aus. Nun berechnen wir $f(x)$ und $f'(x)$.[1] Das neue x' ist: $x' = x - \frac{f(x)}{f'(x)}$. Wiederhole, bis $f(x')$ genügend klein.
  - **Vorteil:** ist schneller als die Bisektionsmethode, konvergiert 'quadratisch' (Anzahl Stellen verdoppeln sich bei jeder Iteration).
  - **Nachteile:** sollte $f'(x) = 0$ sein, strebt $x_{n+1} \to \infty$; es kann in einen ewigen loop hineinfallen.

3.1.2 Lösung von Anfangswertproblemen

- **Euler-Verfahren**
  - Gegeben sei die Gleichung $\dot{y} = f(y,t)$ mit $y(t_0) = y_0$. Wir wählen nun eine Schrittweite $\Delta t > 0$. Mit dieser Schrittweite können die diskreten Zeitwerte $t_k = t_0 + k \cdot \Delta t$ berechnet werden, wobei $k = 0, 1, 2...$ die Schrittnummer ist. Mit diesen berechnen wir die diskret angenäherten Werte für y: $y_{k+1} = y_k + \Delta t \cdot f(y_k, t_k)$
  - Das Euler-Verfahren ist ein Einschrittverfahren. Je kleiner Δt ist, desto genauer wird die Approximation, jedoch ist mehr Rechenarbeit nötig, das Verfahren wird langsamer. Die Abweichungen von den exakten Werten 'pflanzen sich fort': Der Fehler auf dem approximierten Wert steigt mit jedem Rechnungsschritt. Die Unsicherheit liegt in der Grössenordnung von $\approx \Delta t^2$.
- **Runge-Kutta-Verfahren**
  - Anders als beim Euler-Verfahren verwendet man s Zwischenschritte zwischen den zwei Stützstellen t_n und t_{n+1}. Nach Anzahl Zwischenschritte s wird das jeweilige

[1] Ist $f'(x)$ unbekannt, kann es mit $\approx \frac{f(x+\Delta x)-f(x)}{\Delta x}$ angenähert werden.

Verfahren als s-stufig bezeichnet. Der Fehler bei der 2-stufigen Methode liegt in der Grössenordnung von $\approx \Delta t^3$, bei der 4-stufigen bei $\approx \Delta t^5$. Die Runge-Kutta-Methoden brauchen mehr Rechenschritte als die Euler-Methode, sind jedoch genauer.

– Für die nachfolgenden Beispiele gelten immer die gleichen Ausgangslagen: Sei $y' = f(y,t)$ und $y(t_0) = y_0$, die Schrittweite sei Δt, n die Schrittzahl und k_s der s-te Zwischenschritt.

2-stufige Runge-Kutta-Methode

$$\begin{aligned} k_1 &= f(y_n, t_n) \\ k_2 &= f\left(y_n + \tfrac{\Delta t}{2} \cdot k_1, t_n + \tfrac{\Delta t}{2}\right) \\ y_{n+1} &= y_n + \Delta t \cdot k_2 \end{aligned}$$

4-Stufige Runge-Kutta-Methode

$$\begin{aligned} k_1 &= f(y_n, t_n) \\ k_2 &= f\left(y_n + \tfrac{\Delta t}{2} \cdot k_1, t_n + \tfrac{\Delta t}{2}\right) \\ k_3 &= f\left(y_n + \tfrac{\Delta t}{2} \cdot k_2, t_n + \tfrac{\Delta t}{2}\right) \\ k_4 &= f\left(y_n + \Delta t \cdot k_3, t_n + \Delta t\right) \\ y_{n+1} &= y_n + \Delta t \cdot \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned}$$

- **Das Leapfrog-Verfahren**
  – Das Leapfrog-Verfahren ist eine Methode zur numerischen Integration von Differentialgleichungen vom Typ $\ddot{x} = \dot{v} = f(x)$. Beim Leapfrog-Verfahren werden Zwischenwerte berücksichtigt. Dies kann geschehen, indem v_n zum Zeitpunkt t_n nicht für einen ganzen, sondern einen halben Zeitschritt gilt. lässt. Von der Position bei $t_n + \frac{\Delta t}{2}$ wird mit v_{n+1} für x_{n+1} zur Zeit $t_n + \Delta t$ gerechnet. Um von einer alten zu einer neuen Position zu gelangen, arbeitet man nun mit dem Mittel der alten und neuen Momentangeschwindigkeiten bzw. Positionen.

"Drift-Kick-Drift" version

$$\begin{aligned} x_{n+\frac{1}{2}} &= x_n + v_n \frac{\Delta t}{2} \\ v_{n+1} &= v_n + f(x_{n+\frac{1}{2}})\Delta t \\ x_{n+1} &= x_{n+\frac{1}{2}} + v_{n+1}\frac{\Delta t}{2} \end{aligned}$$

"Kick-Drift-Kick" version

$$\begin{aligned} v_{n+\frac{1}{2}} &= v_n + f(x_n)\frac{\Delta t}{2} \\ x_{n+1} &= x_n + v_{n+\frac{1}{2}}\frac{\Delta t}{2} \\ v_{n+1} &= v_{n+\frac{1}{2}} + f(x_{n+1})\frac{\Delta t}{2} \end{aligned}$$

– Die Leapfrog-Integration liefert i.A. genauere Ergebnisse als das Euler-Verfahren. Es ist **invariant** unter Zeitumkehr und erhält exakt Erhaltungsgrössen wie den **Impuls und Energie**. Bei der Integration wird nicht mehr durch Rechtecke angenähert, sondern **Trapeze** ersetzt, da man nun Mittelwerte benutzt.[2] Das **Problem** am Leapfrog-Verfahren ist aber, dass es neben x_0 und v_0 noch x_1 oder v_1 zum Anfangen braucht. (Dies kann zwar durch Euler/Runge-Kutta berechnet werden, der Rest der Simulation ist aber sehr stark von diesem Anfangswert abhängig.) Der Fehler liegt bei der Grössenordnung $\approx \Delta t^2$.

- **Ableitungen**

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1/2} - u_{i-1/2}}{\Delta x} \qquad \text{oder} \qquad \frac{u_i - u_{i-1}}{\Delta x}$$
$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$$

[2] Siehe: http://upload.wikimedia.org/wikibooks/de/2/25/Das_Mehrk%C3%B6rperproblem_in_der_Astronomie_Strecke_und_momentane_Geschwindigkeit_im_Leapfrog-Verfahren.png

3.2 Zu simulierende Situationen

3.2.1 Game of Life

Wir simulieren das Leben von Zellen über einem Gebiet. Vereinfacht betrachten wir das Gebiet als ein Gitter. Die Zellen haben genau zwei Existenzmöglichkeiten: 'Lebt' und 'Tot'. Die Auswertung geschieht über einzelne Zeitschritte, welche für alle Zellen gelten. Die Spielregeln sind:

1. Eine lebende Zelle mit weniger als 2 lebenden Nachbarn stirbt im nächsten Zeitschritt.
2. Eine lebende Zelle mit 2 oder 3 lebenden Nachbarn lebt weiter im nächsten Zeitschritt.
3. Eine lebende Zelle mit mehr als 3 Nachbarn stirbt im nächsten Zeitschritt.
4. Eine tote Zelle mit genau 3 Nachbarn lebt im nächsten Zeitschritt.

3.2.2 Fluiddynamik und Gitter-Gas-Modelle

Beim **HPP Gitter-Gas-Modell** brauchen wir ein 2-dimensionales, orthogonales Gittermodell. Die Regeln dabei sind:

1. Es gibt 4 erlaubte Geschwindigkeiten: N,S, E, W
2. Pro Gitterzelle dürfen 0 oder 1 Teilchen pro Geschwindigkeit (also 0-4 Teilchen pro Zelle)
3. Ein Zeitschritt besteht aus Propagation und Kollision
4. Regeln für Kollision: Bei horizontaler Kollision (W und E) gehen die Teilchen vertikal auseinander (N und S). Bei vertikaler Kollision (N und S) gehen die Teilchen horizontal auseinander (W und E).
5. Der Rand soll die Teilchen reflektieren.

Simuliert werden soll die Teilchenbewegung und die Teilchendichte (Anzahl Teilchen in Zelle).

Wir benutzen bitweise Operationen für Geschwindigkeit und Speicheroptimierung. Wir definieren jede Geschwindigkeit als einen anderen Bit:

Geschw.	Bits	Integer
E	00001	1
N	00010	2
W	00100	4
S	01000	8
Rand	10000	16

Nicht am Rand (<16) gilt für die Kollisionen: Aus 00101 (5 als int, EW) wird 01010 (10 als int, NS) und umgekehrt. Am reflektierenden Rand ($\geq$ 16) gilt: Aus 10001 wird 10100 und umgekehrt, aus 10010 wird 11000 und umgekehrt.

3.2.3 SIR (Susceptible Infected Recovered) - Modell

Wir betrachten die Ausbreitung einer Krankheit in einer konstanten Population P. Anfällige (Susceptibles S) können mit einer Infektionsrate angesteckt werden. Die Infizierten I können mit einer Genesungsrate genesen, wobei a die Genesungszeit bzw. -wahrscheinlichkeit ist. r ist die Erkrankungswahrscheinlichkeit. Die genesene Population ist immun und kann nicht wieder angesteckt werden. Es gelten die Gleichungen:

$$\frac{dS}{dt} = -r \cdot S \cdot I \qquad \frac{dR}{dt} = a \cdot I \qquad \frac{dI}{dt} = -\frac{dS}{dt} - \frac{dR}{dt}$$

3.2.4 Jäger-Beute-Modell / Lotka-Volterra-Modell

Beim Jäger-Beute-Modell werden die Populationen von Räubern (Falken h) und Beute (Eichhörnchen s) betrachtet. Gibt es genug wenige Räuber, vergrössert sich die Eichhörnchenpopulation, da sie sich schneller vermehren als aussterben. Mit steigender Eichhörnchenpopulation steigt aber auch die Falkenpopulation, denn mehr Nahrung ist vorhanden und mehr Falken überleben. Das geht so weiter, bis die Falkenpopulation gross genug ist, um zu viele Eichhörnchen zu fressen, so dass die Eichhörnchenpopulation sinkt. Mit ihr sinkt auch die Falkenpopulation, da wieder weniger Nahrung vorhanden ist. Das geht so weiter, bis die Falkenpopulation wieder klein genug ist, dass sich die Eichhörnchenpopulation wieder vergrössern kann. Dabei gelten die Gleichungen:

$$\frac{ds}{dt} = s(k_s - k_f \cdot h) \qquad \frac{dh}{dt} = h(k_h \cdot s - k_d)$$

wobei die Konstante k_s ist Reproduktionsrate der Eichhörnchen, k_d ist die die Sterberate der Falken, wenn keine Nahrung vorhanden ist, k_f die Rate, mit der Eichhörnchen gefressen werden (Sterberate pro Räuber) und k_h die Reproduktionsrate der Falken pro Beutelebewesen.

3.2.5 Daisyworld

Daisyworld ist eine Welt, in welcher der Planet nur aus weissen Daisies[3], schwarzen Daisies und barer Erde besteht. Die Sonne strahlt auf die Erde ein und überträgt somit Energie. Die einfallende Energie E_{in} ist $E_{in} = l \cdot c_s$ mit l als Luminositätsfaktor (Welcher Teil der Sonneneinstrahlung auf die Erde einwirkt, zwischen 0.6 und 1.8) und mit der konstanten Solarkonstante c_s (mittlere Sonnenbestrahlungsintensität, 1370W/m^2). Von dieser einfallenden Energie E_{in} wird ein Teil von der Erde reflektiert: $E_r = E_{in} \cdot A_{tot}$. A_{tot} nennt sich Albedo und ist der Faktor der einfallenden Strahlung, die reflektiert wird. Die absorbierte Energie beträgt also: $E_{abs} = E_{in} - E_r = (1 - A_{tot})E_{in}$.
Der Albedo setzt sich folgendermassen zusammen: weisse Daisies, schwarze Daisies und bare Erde haben einen eigenen Faktor: $A_w = 0.75$, $A_e = 0.5$, $A_b = 0.25$. Der gesamte Albedo A_{tot} setzt sich zusammen aus dem Anteil der einzelnen Flächen und dem zugehörigen Albedo: $A_{tot} = f_w \cdot A_w + f_e \cdot A_e + f_b \cdot A_b$, wobei f_w, f_e und f_b die Anteile der von der Erde, schwarzen und weissen Daisies eingenommenen Fläche sind. Offensichtlich muss gelten: $f_w + f_e + f_b = 1$.
Damit ein Gleichgewicht eintritt, muss gelten: Der Planet absorbiert und emittiert die gleiche Energie. So schliessen wir auf die Erdtemperatur: $E_{abs} = E_{em} = \sigma \cdot T^4$. σ ist die Stefan-Boltzmann-Konstante. Daraus folgt: $T = \left(\sqrt{\frac{l \cdot c_s \cdot (1 - A_{tot})}{\sigma}}\right)^{1/4}$. Die Temperatur bestimmt die Wachstumsrate der Daisies (schwarze und weisse). Es gilt für beide Daisies die Flächenänderung:

[3] engl. für Gänseblümchen

$\frac{\partial f_i}{\partial t} = f_i \cdot (f_e \cdot g(T) - d)$. d ist die Sterberate, $g(T)$ der Temperaturabhängige Wachstumsfaktor: $g(T) = 1 - 0.003265 \cdot (22.5 - T)^2$. [4]
Zu berechnen sind das Wachstum der Daisies und die Temperatur, nachdem sich das Gleichgewicht eingestellt hat.

3.2.6 Planetenbahnen

Ziel dieser Simulation ist die Berechnung von Planetenbahnen. Seien n Planeten mit Masse M_i, Anfangsposition (x_i, y_i) und Anfangsgeschwindigkeiten (v_{xi}, v_{yi}) gegeben. Die auf den Planeten p wirkende Beschleunigung ist die Summe der Gravitationsbeschleunigung aller anderen Planeten: $\ddot{x} = a_p = \sum_{i=1, i \neq p}^{n} \frac{G \cdot M_i}{|r_i|^3} \cdot \vec{r}_i$. G ist die Gravitationskonstante, M_i die Masse des anderen Planeten und $\vec{r}_i$ der Abstand zwischen den zwei Planeten: $\vec{r}_i = \vec{x}_p - \vec{x}_i$, wobei x_p und x_i die Ortsvektoren der beiden Planeten sind.

3.3 (Reaktion-) Diffusionsgleichungen

Die Diffusionsgleichung ist $\frac{\partial u}{\partial t} = D \cdot \frac{\partial^2 u}{\partial x^2} \approx D \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$.
Zur Beschreibung von Reaktionen gilt die Fisher-Kolmogorov-Gleichung: $\frac{\partial u}{\partial t} = D \cdot \frac{\partial^2 u}{\partial x^2} + u(1 - u) \approx D \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + u(1 - u)$.
Das Gierer-Meinhart-Modell für Reaktionsgleichungen ist:

$$\frac{\partial a}{\partial t} = p \cdot \frac{a^2}{h} - M_a \cdot a + D_a \cdot \frac{\partial^2 a}{\partial x^2} + p_0 \qquad \frac{\partial h}{\partial t} = p \cdot a^2 - M_h \cdot h + D_h \cdot \frac{\partial^2 h}{\partial x^2}$$

3.3.1 Logistische Gleichungen

Die logistische Gleichung wird dazu gebraucht, um chaotisches Verhalten zu simulieren. Eine einfache Form der Gleichung ist $x_{n+1} = r \cdot x_n(1 - x_n)$. Ihr Verhalten ist stark von r abhängig. Ein Beispiel wäre mit $x_n = sin^2(\Theta_n)$. Dadurch ergibt sich für $r = 4$: $x_{n+1} = sin^2(\Theta_{n+1}) = 4x_n(1 - x_n) = 4sin^2(\Theta_n)(1 - sin^2(\Theta_n)) = 4sin^2(\Theta_n)cos^2(\Theta_n) = sin^2(2\Theta_n)$. Das Chaos kommt bei der Definition des Θ_{n+1} ins Spiel: $\Theta_{n+1} = 2\Theta_n mod(2\pi)$. Das $mod(2\pi)$ gibt nur den Rest bei einer Division durch 2π an, was die Rechnung einfacher macht, da $sin(x)$ ja 2π-periodisch ist. Einfach gesagt verdoppeln wir den Winkel Θ bei jeder Iteration und schauen, was dabei rauskommt.
Eine andere Möglichkeit zur 'Chaos-Erstellung' wäre es, eine irrationale Zahl z (z.B. $\sqrt{2}$, π, e...) zu wählen, diese bei jeder Iteration mit 10 zu multiplizieren und den Integer-Anteil abzuschneiden. So 'verschieben' wir die keiner Symmetrie folgenden Nachkommastellen immer um eine weitere Stelle nach links: $y_0 = z$, $y_{i+1} = 10 \cdot y_i - int(10 \cdot y_i)$.

3.3.2 Julia-Menge und Mandelbrot-Menge

Die Mandelbrot-Menge ist die Menge aller komplexen Zahlen c, für welche die Reihe $z_0 = c$, $z_{n+1} = z_n^2 + c$ gilt. Sobald $|z_i| \geq 2$, kann man Konvergenz ausschliessen. (D.h. bei jedem Schritt prüfen, ob $|z_n| \geq 2$. Falls ja, nicht abspeichern.)
Ist z_0 gegeben und c der Parameter, nennt man die Menge Mandelbrot-Menge. Ist der Anfangswert z_0 und c gegeben, so nennt man die Menge Julia-Menge.

[4] Gemeint sind hier 22.5°C

Programmbeispiele (Python)

```python
def bisektion(f, a, b, epsilon):
    '''
    Unter der Annahme, dass f(a)<0 und f(b)>0.
    Sonst müsste das zuerst noch geprüft werden.
    '''
    while abs(f(0.5*(a+b))) > epsilon:

        if f(0.5*(a+b)) < 0:
            a = 0.5*(a+b)

        else:
            b = 0.5*(a+b)

    return 0.5*(a+b)
```

Abbildung 3.1: Programmbeispiel Bisektionsverfahren

```python
def newton_raphson(func, x0):
    """
    Hier mit abgeschätzter Ableitungsfunktion
    """

    delta = 1e-15
    xn = x0
    xm = x0

    def abl(x):
        return (func(x+delta)-func(x))/delta

    while abs(func(xm)) <= delta:
        xn = xm
        xm = xn - func(xn)/abl(xn)

    return xm
```

Abbildung 3.2: Programmbeispiel Newton-Raphson-Verfahren

```
def euler(func, y0, t):
    '''
    t ist hier ein Array von den Stellen, an
    denen das Verfahren durchgeführt wird.
    (Die Abstände müssen auch nicht zwingender-
    massen gleich sein.)
    '''
    y = y0
    y_list = [1*y]
    for i in range(len(t)-1):
        h = t[i+1]-t[i]
        dy = h*func(y, t[i])
        y += dy
        y_list.append(1*y)
    return array(y_list)
```

Abbildung 3.3: Programmbeispiel Euler-Verfahren

```
def rungekutta2(func, y0, t):
    '''
    t ist hier ein Array von den Stellen, an
    denen das Verfahren durchgeführt wird.
    (Die Abstände müssen auch nicht zwingender-
    massen gleich sein.)
    '''

    y = y0
    y_list = [1*y]

    for i in range(len(t)-1):
        h = t[i+1]-t[i]
        k2 = func(y + h/2*func(y,t[i-1]), t[i-1]+h/2)
        y += h*k2
        y_list.append(1*y)
    return array(y_list)
```

Abbildung 3.4: Programmbeispiel Zweistufiges-Runge-Kutta-Verfahren

```
def rungekutta4(func, y0, t):
    '''
    t ist hier ein Array von den Stellen, an
    denen das Verfahren durchgeführt wird.
    (Die Abstände müssen auch nicht zwingender-
    massen gleich sein.)
    '''

    y = y0
    y_list = [1*y]

    for i in range(len(t)-1):
        h = t[i+1]-t[i]
        f1 = func(y,t[i])
        f2 = func(y + h/2*f1, t[i]+h/2)
        f3 = func(y + h/2*f2, t[i]+h/2)
        f4 = func(y + h/2*f3, t[i]+h/2)
        y += h/6*(f1+2*f2+2*f3+f4)
        y_list.append(1*y)
    return array(y_list)
```

Abbildung 3.5: Programmbeispiel Vierstufiges-Runge-Kutta-Verfahren