

2.4 Software-Praxis „NetLogo“

Die Simulation von Umwelten und Mikrowelten am Computer wird durch unterschiedliche Entwicklungsumgebungen und in einer Vielzahl von Programmiersprachen realisiert. Wir wollen uns im folgenden Exkurs auf die Entwicklungsumgebung NetLogo (Wilensky, 1999) konzentrieren.

NetLogo ist eine programmierbare Entwicklungsumgebung zur Simulation von natürlichen und sozialen Phänomenen und Systemen. Diese unter der OpenSource-Lizenz zugängliche Software ist aus der Entwicklungsumgebung *StarLogo* des *MIT Media Lab* hervorgegangen. Ihr Autor Uri Wilensky hat NetLogo am *Center for Connected Learning and Computer-Based Modeling (CCL)* seit 1997 zunächst an der Tufts University in der Nähe von Boston und seit 2000 an der Northwestern University nahe Chicago entworfen und weiterentwickelt.

NetLogo basiert auf den Sprachen Scala und Java und bietet dem Anwender bereits bei der Installation eine umfangreiche Model-Bibliothek an, die Simulationen aus den Disziplinen Biologie, Medizin, Physik, Chemie, Mathematik, Computerwissenschaften, Ökonomie und Sozialpsychologie umfasst. Anwender können nicht nur die Parameter in den Simulationen verändern, sondern vollständig in den Code der Modelle eingreifen.

Die Modellierung eigener Systeme kann sowohl programmiert im softwareeigenen auf Logo basierenden Dialekt erfolgen, als aber auch in einem speziellen Editor in der Flussdiagramm-Syntax von *System Dynamics* (Bossel, 2004). Die anschließende Übersetzung der Diagramme in lauffähige Systeme wird von NetLogo übernommen.

Eine besondere Stärke von NetLogo sind die umfassenden Interfaces und Formen der Darstellungsmöglichkeiten; so lassen sich alle lauffähigen Simulationen nicht nur numerisch oder in Form von unterschiedlichen Diagrammen abbilden, sondern können als zweidimensionale oder dreidimensionale Umwelten am Monitor visualisiert werden.

Wir betrachten zunächst ein Modell aus der Bibliothek, bevor wir uns der codebasierten Entwicklung eines Modells zuwenden. Alle Aktivitäten innerhalb der Software werden zur Übersichtlichkeit im Folgenden in einem eigenen Schrifttypen im Text eingefügt sein, zusätzlich sind alle Aktionen durchnummeriert.

2.4.1 Modelle aus der Bibliothek nutzen

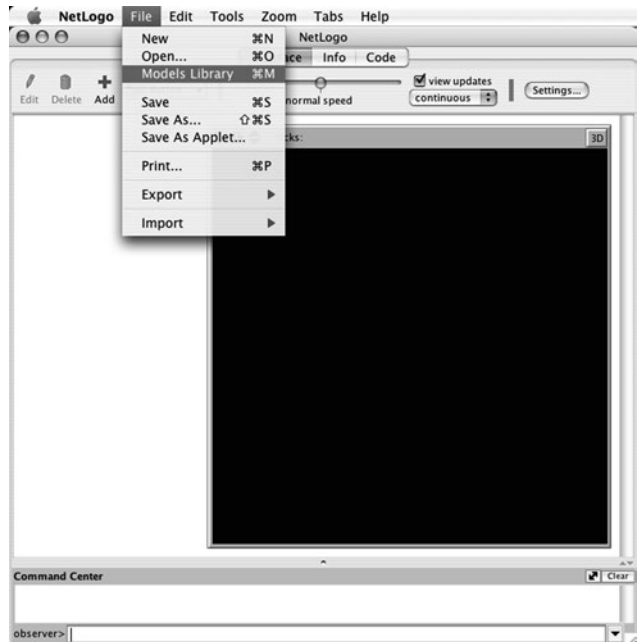
Beobachtung des Modellverhaltens

- ✓ #1 Wir öffnen unter dem Hauptmenüpunkt File die Modell-Bibliothek (Models Library) (■ Abb. 2.13) und wählen den Unterordner Social Science, hier wählen wir das Modell Traffic Basic.

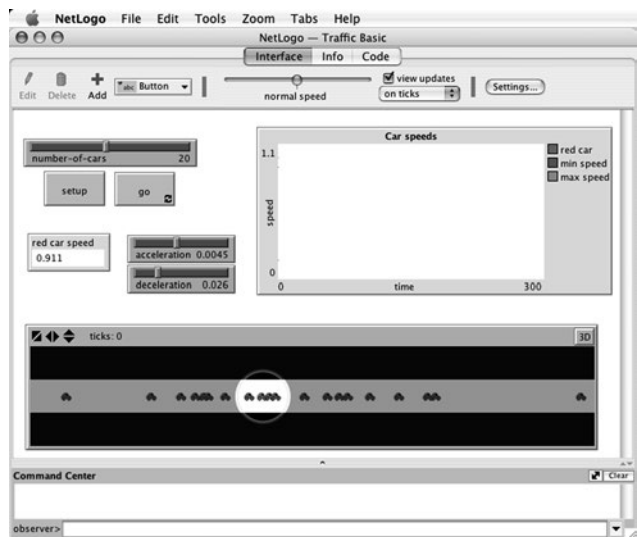
Das zentralste Element von NetLogo ist der sog. *View*, der nach dem ersten Laden eines Modells zunächst als schwarzes Fenster erscheint. Ebenso zentral sind die Buttons *go* und *setup* (■ Abb. 2.14). Bevor ein dynamisches Modell in Bewegung gebracht werden kann, muss es über den Button *setup* erstmalig aufgebaut werden; wenn zu einem späteren Zeitpunkt einzelne Variablen im Modell verändert werden, ist dieser Knopf zum Aufsetzen des modifizierten Modells ebenfalls notwendig.

- ✓ #2 Über das Betätigen des Buttons *setup* wird das Modell aufgebaut.

■ **Abb. 2.13** Das Benutzer-Interface von NetLogo erscheint zunächst ohne Modell. Dieses muss entweder selbst programmiert oder aus der Modell-Bibliothek (Models Library) geladen werden



■ **Abb. 2.14** Das Modell „Traffic Basic“ stellt ein relativ übersichtliches Szenario dar. Zu erkennen sind die unterschiedlichen Repräsentationsformate



Im Beobachtungsfenster erscheint nun das Modell zum Straßenverkehr als grafische zweidimensionale Darstellung. Dieses Szenario kann über den Button go „zum Leben erweckt“ werden.

✓ #3 Über den Button go startet das Modell.

In unserem Beispiel zeigt sich, dass zwanzig Fahrzeuge auf einem runden Parcours fahren, wenn ein Auto am rechten Bildschirmrand die Strecke verlässt, taucht es am linken Bildschirmrand wieder auf.

Wir können nun das spezifische Verhalten unseres Modells über einen längeren Zeitraum beobachten. Wir sehen nach einiger Zeit, dass sich nicht immer alle Fahrzeuge in Fahrt befinden, sondern dass es immer wieder zu Stillständen auf der Strecke kommt. Die Fahrzeuge sind also in unterschiedlichen Geschwindigkeiten und unterschiedlichen Abständen zueinander auf der Strecke unterwegs. Unser Modell kann als sehr simples und reduziertes Modell einer Autobahn betrachtet werden. Die Anzahl der Spuren ist auf eine einzige reduziert worden und die unterschiedlichen Fahrzeugmodelle sind ebenfalls zu einem Automodell reduziert. Weitere Faktoren, wie etwa die Persönlichkeit der Autofahrer oder die aktuelle Witterung, sind vorerst nicht in das Modell aufgenommen worden. Dennoch reichen die verwendeten Systemvariablen aus, um eine Autobahn mit dem Phänomen Stau zu simulieren. Wollen wir das Modell zum Stoppen bringen, so inaktivieren wir den Button go.

✓ #4 Ebenfalls über den Button go stoppt das laufende Modell.

Wir können unsere Simulation jederzeit über den Button go starten und unterbrechen. Die Zeit wird in NetLogo über Zeitschritte, sog. *Ticks*, simuliert. Die Anzahl der Zeitschritte wird am oberen Rand des Beobachtungsfensters angezeigt. Mit Betätigung des Buttons go wird also die Zeitvariable zum Laufen oder Pausieren gebracht und damit das ganze System gesteuert.

✓ #4 Über den Button go wird das pausierende Modell wieder gestartet.

Zur genaueren Beobachtung eines Modells kann es manchmal notwendig sein, das komplette System langsamer laufen zu lassen um einzelne Prozesse besser beobachten zu können oder aber man möchte das System über einen längeren Zeitraum betrachten und eine Beschleunigung des gesamten Modellablaufes wäre sinnvoll. Beides lässt sich über den Geschwindigkeitsschieberegler am oberen Bildschirmrand modifizieren, der zunächst als *normal speed* voreingestellt ist. Über die Nutzung des Geschwindigkeitsschiebereglers verändert sich der Intervall zwischen den einzelnen Zeitschritten bzw. Ticks.

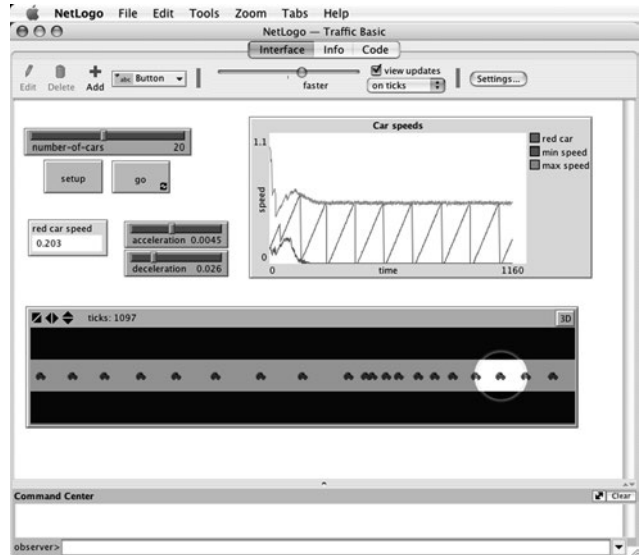
✓ #5 Über den Geschwindigkeitsschieberegler wird das gesamte Modell in seiner Abfolge beschleunigt (faster) oder entschleunigt (slower).

Die Historie der jeweiligen Modellsimulation zeigt sich folglich an der Anzahl der jeweiligen Ticks. Möchte man das Modellverhalten über längere Zeiträume beobachten, wird man in NetLogo durch ein Zeitdiagramm (*plot*) unterstützt. In unserem Beispiel befinden sich im Zeitdiagramm drei Linien: die Maximalgeschwindigkeit im System (*max speed*), die Minimalgeschwindigkeit (*min speed*) und die Geschwindigkeit eines Referenzfahrzeugs (*red car*). Alle Werte sind standardisiert und bewegen sich zwischen 0 (Stillstand) und 1.0 (potenzielle Höchstgeschwindigkeit).

✓ #6 Wir stoppen die aktuelle Simulation und stellen über den Button setup eine neue Modellsimulation mit dem Startpunkt tick 0 her.

Sofern wir uns noch an den Startzustand unserer ersten Simulation erinnern können, wäre jetzt schon ein Unterschied in der Startkonfiguration zu erkennen. Über den Button setup wird nie derselbe Startzustand installiert, sondern immer eine andere zufällige Anordnung durch einen randomisierten Abstand der Fahrzeuge zueinander aufgebaut. Wir könnten dies durch mehrfachen Drücken des Buttons setup testen (■ Abb. 2.15).

■ **Abb. 2.15** Das Verlaufsdiagramm bietet einen sehr guten Überblick über das Modell-Verhalten.

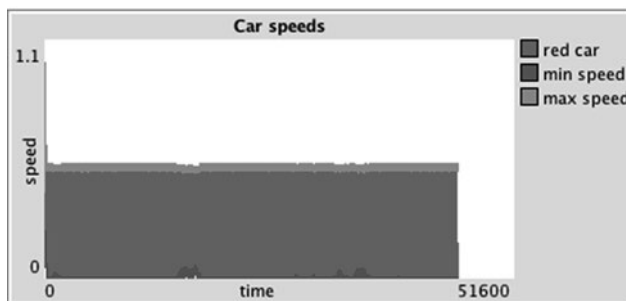


- ✓ #7 Wir lassen die aktuelle Simulation bis zu einem Zeitpunkt von über 1.000 Ticks durchlaufen, um sie dann zu pausieren.

Anhand der grünen Linie ist zu erkennen, dass das System von zwanzig Fahrzeugen zu jedem Zeitpunkt eine Höchstgeschwindigkeit aufweisen konnte. Diese war nicht konstant sondern streut um einen Bereich von 0.5, wobei die potenzielle Höchstgeschwindigkeit von 1.0 nie erreicht worden ist. Die Minimalgeschwindigkeit im System lag hingegen nach ungefähr 200 Ticks konstant bei der minimal zu erreichenden Geschwindigkeit von 0. Diese Linien zeigen uns also an, dass es immer mindestens ein Fahrzeug mit einer relativen Höchstgeschwindigkeit gab und ebenso immer mindestens ein Fahrzeug im System stillstand. Es gab also Stau auf unserem Parcours. Um das Systemverhalten dem Verhalten der Individuen im System gegenüberstellen zu können, wird in unserem Modell ein Fahrzeug als Referenzfahrzeug genutzt. Dieses Beobachtungsfahrzeug hat einen eigenen sog. *Monitor* erhalten, der zu jedem Tick die aktuelle Geschwindigkeit anzeigt während das Modell läuft, und eine eigene Linie im Zeitdiagramm, um das Geschwindigkeitsverhalten über den gesamten Beobachtungszeitraum abzubilden.

Betrachten wir nun alle drei Linien im Zusammenspiel, so kann anhand der Abszisse bzw. X-Achse der Simulationsverlauf zunächst in zwei Phasen unterteilt werden. Eine unregelmäßige Startphase wird von einer zweiten stabiler erscheinenden Phase abgelöst, in der sich Wiederholungen ähnlicher aber nicht unbedingt identischer Muster der jeweiligen Werte zeigen. In der zweiten Phase gibt es einen konstanten Minimalwert von 0 und im Vergleich zur Startphase streut auch der Maximalwert in einem kleineren Bereich. Zudem zeigt sich im Verhalten des roten Fahrzeugs eine stabile Abfolge in einer nahezu linearen Zunahme der Geschwindigkeit, die in einer Bremsung und dem Stillstand endet, um dann wieder relativ konstant zuzunehmen. In der Startphase zeigt sich deutlich, dass die Minimalgeschwindigkeit sehr stark streut und noch nicht den minimalen Wert von 0 erreicht hat, ebenso variiert die Maximalgeschwindigkeit wesentlich stärker und weist Werte auf, die dem potenziellen Maximum von 1 deutlich näher sind. Es lassen sich also anhand des Zeitdiagramms instabile Phasen, wie der Startphase, deutlich von stabileren Phasen abgrenzen. Auch wenn sich ein komplexes System sehr deutlich eingependelt hat und die Werte relativ stabil zu oszillieren scheinen, kann nicht ausgeschlossen werden, dass nicht

■ **Abb. 2.16** Die Skalierung des Verlaufsdiagramms passt sich automatisch der Anzahl an Zeiteinheiten an



zu einem späteren Zeitpunkt wieder eine instabile Phase auftritt. Dies gilt auch für solch relativ kleine komplexe Systeme wie unserem Stau-Szenario. Daher kann es sich lohnen, Beobachtungsphasen unterschiedlich lange auszudehnen. Für fortgeschrittene Benutzer bietet NetLogo später die Funktion, sich alle Daten zudem als Datenblatt im csv-Format ausgeben zu lassen, um für die genauere Analyse Statistiksoftware wie SPSS oder R hinzuzunehmen.

- ✓ #8 Wir lassen die aktuelle Simulation bis zu einem Zeitpunkt von über 400.000 Ticks durchlaufen und verwenden hierzu den Geschwindigkeitsschieberegler.

Das Zeitdiagramm ändert mit der Anzahl der Ticks seine Relationen, sodass sich der Gesamtverlauf des Systems im Fenster anzeigen lässt (■ Abb. 2.16). Über eine wesentlich ausgedehnte Beobachtungsphase zeigt sich, dass eine stabile oszillierende Phase in unserem Verkehrssystem nach einiger Zeit wieder durch instabile Phasen unterbrochen wird; in diesen Phasen befindet sich kein Stau im System. Das dominierende Verhalten des gegebenen Systems ist allerdings das Vorherrschen von Stau. Könnte das Simulationsmodell mit den gegebenen Variablen auch ohne Stau laufen? Was wäre zu modifizieren?

2.4.2 Experimentieren mit Modellen

Die größte Stärke einer Computersimulation liegt in der gegebenen Möglichkeit, mit dieser zu experimentieren und Hypothesen zu testen. Eine Computersimulation ist gewissermaßen ein geschützter Raum, bevor ein Modell in die Realität umgesetzt wird oder Veränderungen in einem realen System vorgenommen werden. Je besser ein Modell simuliert wird, umso höher ist die Validität der aus ihr gewonnenen Prädiktionen. Bleiben wir in unserem Szenario, so könnte eine für uns wichtige und interessante Frage lauten: Kann auf der gegebenen Teststrecke mit einer Anzahl von 20 Fahrzeugen alleine durch die Veränderung der Variablen Beschleunigung oder Entschleunigung ein Stau vermieden werden? Die für uns wesentliche abhängige Variable (AV) ist also die Minimalgeschwindigkeit, die nicht den Wert 0 annehmen sollte. Für die Erstellung unserer Hypothesen finden wir folgende unabhängigen Variablen (UVn): Beschleunigung (*acceleration*) und Entschleunigung/Bremsweg (*deceleration*).

Wir haben unser Modell bislang unter den Default-Werten 0.0045 von *acceleration* und 0.026 von *deceleration* getestet. Diese Werte werden uns von den in NetLogo üblichen Schiebereglern angezeigt. Hinsichtlich unserer Hypothesen bietet es sich an, vorerst zwischen niedrigen und hohen Wertebereichen bzgl. unserer UVn zu unterscheiden. Wir könnten also folgende Hypothese formulieren: Befindet sich die Beschleunigung in einem hohen Bereich und der Bremsweg

in einem niedrigen Bereich so erwarten wir ein signifikant abweichendes Systemverhalten, das sich durch fehlenden Stau auszeichnet. Eine solche Vorhersage unterscheidet sich deutlich von einem Vorgehen, das zunächst nur explorativ agiert und dabei alle möglichen UVn-Einstellungen durchprobieren würde.

- ✓ #1 Wir laden das Modell Traffic Basic aus der Bibliothek und setzen den Schieberegler acceleration auf den Wert 0.0075. Nach dem setup lassen wir das Modell auf ungefähr 2.500 Zeitpunkte (Ticks) vorlaufen.

Während der Beobachtung des Systemverhaltens ist uns aufgefallen, dass einzelne Fahrzeuge zu Beginn sehr langsam sind und es zu stauähnlichen Zuständen im System kommen kann. Dennoch zeigt sich in der weiteren Beobachtung, dass sich der Abstand zwischen den Fahrzeugen aus einer anfänglich großen Streuung zu einem sehr einheitlichen Maß reguliert und die Fahrzeuge im späteren Verlauf alle auf einer sehr einheitlichen und hohen Geschwindigkeit über den Parcours fahren, ein Stau oder stauähnlicher Zustand ist abschließend nicht mehr zu erkennen.

Ein Blick in das Zeitdiagramm zeigt uns, dass wir wieder von einer Startphase sprechen dürfen, in der die Differenz zwischen Minimal- und Maximalgeschwindigkeit der einzelnen Fahrzeuge gemessen am Referenzfahrzeug noch sehr stark variiert, während sich beide Mittelwerte immer mehr dem optimalen Wert des Maximalwertes annähern, sodass am Übergang zwischen den Phasen alle Fahrzeuge die Maximalgeschwindigkeit im System erreichen und stabil halten. Wir wollen unsere Hypothese daher als angenommen gelten lassen und abschließend testen wir diese noch einmal in einem längeren Simulationsverlauf.

- ✓ #2 Wir lassen die aktuelle Simulation bis zu einem Zeitpunkt von über 400.000 Ticks durchlaufen und verwenden hierzu den Geschwindigkeitsschieberegler (acceleration).

In einem langen Simulationsverlauf zeigt sich zu keinem anderen Zeitpunkt mehr ein Verlassen des stabilen Systemzustandes, sodass wir von einem sehr stabilen System sprechen können. Wir haben einen ersten Weg gefunden, das Phänomen Stau aus dem simulierten Verkehrssystem zu entfernen. Um das Systemverhalten unter den beiden gegebenen unabhängigen Variablen Beschleunigung und Entschleunigung weiter zu analysieren bietet es sich an, die Schieberegler in drei Wertebereiche zu unterteilen und die Auswirkungen aller möglichen Einstellungen auf das System systematisch zu explorieren. Während diese beiden Variablen systematisch über die Schieberegler beeinflusst werden können, ist die Variable „Abstand zwischen den Fahrzeugen am Startpunkt“ nur unsystematisch über den Button setup zu beeinflussen. Dennoch bietet es sich an, den Einfluss dieser Variable auf das Systemverhalten mittels einer Versuchsreihe zu untersuchen.

- ✓ #3 Wir setzen den Schieberegler acceleration auf den Wert 0.0075 und den Schieberegler deceleration auf den Wert 0.026. Nach dem setup lassen wir das Modell auf ungefähr 1.500 Zeitpunkte (Ticks) vorlaufen und setzen zuvor den Geschwindigkeitsschieberegler zwei Stufen nach rechts auf fast. Wir notieren uns den x-Wert des Ticks, an dem die blaue Linie (min speed) die grüne Linie (max speed) erreicht. Diesen Vorgang wiederholen wir bis zum zehnten Durchgang.

Wir haben nun eine Versuchsreihe, die uns sehr schön das Verhalten komplexer Systeme unter Variation des Startzustandes präsentiert. Die Wertereihe des Autors lautet: 694, 926, 395, 420, 691, 542, 1283, 1139, 762, 602. Es ist eine Stärke des Systems, dass Wertereihen von der hier

Präsentierten enorm abweichen, da die Startzustände des Systems durch errechnete Zufallszahlen bestimmt sind. Für fortgeschrittene Anwender sind diese Variablen beeinflussbar und die verwendeten Zufallszahlen sind im System dokumentiert, um die Reproduzierbarkeit von Experimenten zu ermöglichen.

Das Beispiel-Modell bietet einen dritten Schieberegler zum Experimentieren an: es kann die Anzahl der Fahrzeuge variiert werden. Der Stau in unserem System ist natürlich so wie in der Realwelt darauf zurückzuführen, dass die Autobahn überlastet ist. Megastädte wie Shanghai lösen solche Probleme mittlerweile über die Regelung, dass ungerade Kfz-Nummern an anderen Tagen fahren dürfen als gerade Kfz-Nummern. Für unser Szenario ist es interessant, herauszufinden, ab welcher Anzahl an Fahrzeugen das Straßensystem zu Stau oder stauähnlichen Zuständen neigt. Hierzu reduzieren wir schrittweise die Anzahl der Fahrzeuge. Es sind gleiche Effekte zu erwarten, wenn man anstatt der Reduktion der Fahrzeuganzahl eine Verlängerung der Strecke vornehmen würde. Dies ist nicht mehr über Schieberegler möglich, sondern verlangt einen tieferen Eingriff in das Szenario und gibt einen ersten Einblick in den Aufbau von NetLogo-Simulationen, mit dem wir das Kapitel abschließen werden.

Der statische Aufbau einer NetLogo-Simulation basiert auf einem unsichtbaren Gitternetz von Kacheln (*patches*), auf dem eine Microwelt für den Viewer aufgebaut werden kann. Der Ursprung dieser Microwelt liegt als Default-Wert in der Mitte des Fensters *View* mit den Koordinaten 0,0; von hier spannt sich die Welt über die Kacheln auf. Wir können den Wert der Länge unserer Straße erfahren, indem wir in die Settings des Modells gehen.

- ✓ #4 Die Settings des Modells werden entweder über den Button *Settings* oder über den Menüpunkt *Edit* des Kontextmenüs im Fenster *View* geöffnet.

Wir erkennen am oberen Menüpunkt, dass man den Ursprung der Microwelt im Fenster *View* verlegen könnte, belassen diesen aber in der Mitte. Über die Felder *max-pxcor* und *max-pycor* legt man nun die Grenzen der Microwelt fest. Um den *Parcours* zu verlängern, verändern wir die maximalen Koordinaten für die X-Achse, wobei der Negativwert vom Editor automatisch angepasst wird.

- ✓ #5 Im Eingabefeld *max-pxcor* erweitern wir den voreingestellten Wert von 25 zu 40. Wir bestätigen unsere Eingabe mit dem Button *OK* im Fenster und aktualisieren unser Szenario über den Button *setup*, anschließend lassen wir die Simulation bis über 1.000 Ticks laufen.

Wir können den Effekt des Verhältnisses zwischen Anzahl der Autos und Streckenlänge unmittelbar beobachten. Nach einer Verlängerung der Teststrecke um 30 Kacheln ergibt sich ein stabiles System ohne Stau oder stauähnliche Zustände auf dem *Parcours*. Zur weiteren Beobachtung kann nun die Teststrecke wieder verkürzt werden. Zudem lässt sich eine Streckenlänge herausfinden, die aufgrund der unterschiedlichen Fahrzeugabstände am Startpunkt, sowohl in ein stabiles System mit Stau als auch in ein stabiles System ohne Stau führen kann. Dem experimentieren mit gegebenen Modellen ist nun kaum noch eine Grenze gesetzt. Kaum jemand hat das Phänomen Stau schöner beschrieben als die deutsche Hip-Hop-Band die Fantastischen Vier: „Du stehst nicht im, Du bist der Stau!“ Zur Reduzierung dieses Phänomens gibt uns NetLogo eine mächtige Simulationsmöglichkeit an die Hand, die hier auf einer sehr reduzierten Form dargestellt worden ist. Möchte man bestehende Modelle um Variablen erweitern oder eigene Modelle entwickeln, so ist eine Programmierung im Code von NetLogo notwendig. Im folgenden ► Abschn. 2.1.3 werden wir uns mit der Programmierung einer eigenen Microwelt beschäftigen.

2.4.3 Ein eigenes Modell programmieren

Bislang haben wir uns mit Modellen der von NetLogo angebotenen Modell-Bibliothek beschäftigt. Der eigentliche Kern der Software liegt allerdings in der Erstellung eigener Simulationsmodelle. Wie bereits beschrieben, besteht der statische Anteil eines Simulationsmodells in NetLogo aus einer Anordnung von Kacheln (patches), die über die Oberfläche der Welt verteilt werden können. Die aktiven Agenten in der Simulation werden als Turtles bezeichnet. Alle Agenten, auch Patches, werden über von NetLogo zur Verfügung gestellten Kommandos (commands) gesteuert. In Prozeduren können viele unterschiedliche Kommandos zusammengefügt werden.

Das erste Setup

Bevor wir uns der Programmierung zuwenden, die sich zur Einführung von NetLogo in Deutschland eng am Original-Tutorium aus NetLogo orientiert (Wilensky, 1999), starten wir mit den notwendigen Rahmenbedingungen eines jeglichen Modells in NetLogo.

- ✓ #1 Um ein eigenes Modell zu programmieren starten wir NetLogo ohne ein Modell zu laden oder wählen im Menüpunkt File die Option New aus.

Wir starten mit dem Einfügen eines *setup*-Buttons als erstes Element in unserer Microwelt.

- ✓ #2 Da im Drop-down-Menü bereits Button als Default-Wert aktiviert ist, muss nur das Icon Add angeklickt werden und anschließend mit dem Steuerkreuz ein Platz im Fenster bestimmt werden, um einen leeren Button zu platzieren.
- ✓ #3 Es öffnet sich eine Dialogbox in deren Fenster unter Commands das Wort *setup* eingefügt werden muss.

Der Button *setup* bezieht sich nun auf die Prozedur *setup*, die wir noch nicht programmiert haben, daher erscheint der Name auf dem Button vorerst in Rot und bei der Benutzung des Buttons würden wir aktuell noch eine Fehlermeldung erhalten. Wir programmieren nun unseren ersten Code für die Prozedur *setup*.

- ✓ #4 Wir wählen den Reiter Code aus der Reiterliste am oberen Fensterrand.
- ✓ #5 Es wird folgender Code eingefügt:

```
to setup
  clear-all
  create-turtles 100 [setxy random-xxcor random-ycor]
  reset-ticks
end
```

Jede Prozedur endet mit einem *to* und endet mit einem *end*, wobei dem *to* der Name der Prozedur folgt, in unserem Fall ist es der Name *setup*.

Das Kommando `clear-all` führt einen Reset in der Microwelt aus, was wichtig ist, wenn zuvor schon ein Modell simuliert worden ist.

Die aktiven Elemente namens Turtles werden in NetLogo mit dem Kommando `create-turtles` erzeugt, wobei der anschließende Parameter die Anzahl, in unserem Fall also 100 Agenten, festlegt. In eckigen Klammern können dem Agenten weitere Kommandos und Reports übergeben werden. Die Reporter `random-xcor` und `random-ycor` überliefern jeweils Zufallszahlen aus einem erlaubten Bereich der X- und Y-Koordinaten. Diese Zufallszahlen werden an das Kommando `setxy` übergeben, das wiederum die einzelnen Turtles in die Microwelt setzen wird. Es werden also Agenten zufallsverteilt im Koordinatenbereich des Viewers eingefügt.

Das Kommando `reset-ticks` setzt die Simulationszeit auf den Startpunkt 0 zurück und das Kommando `end` definiert den Endpunkt der Prozedur `setup`.

- ✓ #6 Wir testen unseren ersten Code, indem wir über den Reiter Interface auf die Microwelt gehen und dort den Button `setup` nutzen. Über mehrfaches Anklicken dieses Buttons testen wir die Zufallszahlen der Kommandos `random-xcor` und `random-ycor`.

Making the go Button

Bisher haben wir eine statische Microwelt von 100 Agenten erzeugt, nun möchten wir die erste einfache Dynamik in unsere Simulation integrieren. Modelle werden in NetLogo über den Button `go` gestartet und gestoppt, sodass wir diesen zuerst in unsere Oberfläche integrieren.

- ✓ #7 Wir erzeugen eine Button mit dem Kommando `go` und aktivieren in der Dialogbox die Funktionen `Forever` und `Disable until Ticks start`.

Die Funktion *Forever* bestimmt, dass der Button `go` nach dem Anklicken gedrückt bleibt und das Modell fortwährend weiterläuft bis der Button ein weiteres Mal gedrückt wird. Die Funktion `Disable until Ticks start` sorgt dafür, dass das Model erst gestartet werden kann, wenn es über den Button `setup` aufgebaut worden ist.

Der Button `go` bezieht aktuell noch auf eine nicht existente Prozedur namens `go`, die wir nun im Code-Reiter erstellen werden.

- ✓ #8 Wir erzeugen folgenden Code:

```
to go
  move-turtles
  tick
end
```

Die Prozedur `go` ist nun erstellt und erhält eine weitere Prozedur namens `move-turtles`, die noch nicht im Code existiert. Das Kommando `tick` ist bereits in NetLogo integriert und setzt den Counter `tick` um eine Zeiteinheit nach oben. Nun fügen wir noch die Prozedur *move-turtles* hinzu.

✓ #8 Wir fügen folgenden Code hinzu:

```
to move-turtles
  ask turtles [right random 360 forward 1]
end
```

Durch eine einzige Zeile Code wird unser simples Beispiel-Modell zu erstem Leben erweckt. Hierzu wird über das Kommando `ask` an alle Agenten *turtles* die Instruktion gegeben sich nach rechts zu drehen. Das Kommando `right` wird über die Parameter `random 360` dazu verwendet, einen zufälligen Winkel für die Rechtsdrehung zu wählen, und eine anschließende Vorwärtsbewegung um einen Schritt wird durch `forward 1` erwirkt.

✓ #9 Bevor wir nun im Interface den Button `go` nutzen, ist es wichtig, dass die Auswahlbox von `view updates` nicht den Default-Wert `continuous` beibehält, sondern auf `on Ticks` umgestellt wird. Ist dies geschehen, testen wir unseren neuen Code.

Die Agenten bewegen sich zufällig durch die Microwelt und beim Verlassen des Bildschirmes tauchen sie auf der anderen Seite wieder auf, so wie es auch schon auf unserem Autoparcours im vorherigen Abschnitt „Das erste Setup“ in ► Abschn. 2.1.3 geschehen ist. Dieser Vorgang wird als *wrap* bezeichnet und kann von einem erfahrenen Nutzer unter dem Stichwort Topologie der Microwelt ebenfalls umdefiniert werden.

Patches and variables

Bislang haben wir noch keinen statischen Hintergrund in Form von Kacheln (*patches*) erzeugt. Dies werden wir nun ändern.

✓ #10 Unter die Prozedur `setup` fügen wir folgenden neuen Code hinzu:

```
to setup
  clear-all
  setup-patches
  create-turtles 100 [setxy random-xcor random-ycor]
  reset-ticks
end
```

✓ #11 Die Prozedur `setup-patches` müssen wir ebenfalls neu durch folgenden Code definieren:

```
to setup-patches
  ask patches [set pcolor green]
end
```

Es werden also zunächst alle Hintergrund-Kacheln auf den Farbwert Grün gesetzt, unsere Microwelt bildet auf einfache Art eine Wiese ab. Um den Code innerhalb der Prozedur gut leserlich und übersichtlich zu halten, passen wir nun noch das setup der Agenten *turtles* an, indem wir zwei neue Prozeduren schreiben.

✓ #12 Die Prozedur setup wird folgendermaßen erweitert:

```
to setup
  clear-all
  setup-patches
  setup-turtles
  reset-ticks
end
```

✓ #13 Die Prozedur setup-turtles wird durch bereits bekannten Zeilen erzeugt:

```
to setup-turtles
  create-turtles 100
  ask turtles [setxy random-xcor random-ycor]
end
```

Die beiden letzten Änderungen dienen der Übersicht im Code und haben keine Auswirkungen auf das Erscheinen der Microwelt im Interface. Bei größeren Projekten ist ein sauberer und übersichtlicher Code allerdings ein sehr wichtiger Aspekt.

Eigene Variablen Interaktion zwischen turtles und patches

Bislang bewegen sich die Agenten, unangebunden und ziellos über den Hintergrund. Wir möchten eine Interaktion erstellen, indem die Agenten das Gras fressen.

✓ #14 Die Prozedur go wird um eine Prozedur erweitert:

```
to go
  move-turtles
  eat-grass
  tick
end
```

✓ #15 Die Prozedur eat-grass verwendet eine Bedingungsabfrage:

```

to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
    ]
  ]
end

```

Die Interaktion zwischen den simulierten Tieren und ihrer Umwelt wird zunächst relativ simpel hergestellt. Ein Agent erhält vom Patch auf dem er sich befindet immer alle notwendigen Informationen. In unserem Fall wird abgefragt ob die Patch-Farbe `pcolor` grün ist. Es können nur die Werte `TRUE` oder `FALSE` empfangen werden, im Falle von `TRUE` wird die Kachel, auf der sich der Agent befindet auf den Farbwert Schwarz gesetzt, hierzu dient die Bedingungsabfrage `if`. Wir können nun in der Simulation beobachten, wie unsere Agenten alle grünen Kacheln schwarz färben, oder dass die simulierten Insekten die grüne Wiese fressen. Um dieser Interaktion noch etwas mehr Sinn zu verleihen und unser Modell immer mehr einer realen Umwelt anzupassen, fügen wir dem System einen Metabolismus hinzu. Die Insekten ernähren sich vom Gras, um Energie zu erhalten und damit zu überleben. Die Eigenschaft der Energie ist eine spezifische Variable für unsere Agenten und muss im Modell an oberster Stelle im Code deklariert werden. Andere Eigenschaften, wie etwa die Farbe der Agenten, sind bereits in NetLogo vorhanden.

✓ #16 Die Deklaration `turtles-own` wird im Code an die oberste Stelle in die erste Zeile gesetzt:

```

turtles-own [energy]

```

Damit diese Variable in den Metabolismus des Systems integriert wird, wollen wir sowohl einen Zuwachs an Energie durch das Fressen von Gras implementieren, als aber auch einen Energieverlust durch Arbeit. Der Energieverlust entsteht folglich schon bei der Bewegung der Agenten.

✓ #17 Die Prozedur eat-grass wird erweitert um die Energiezunahme:

```

to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy energy + 10
    ]
  ]
end

```

Die vorhandene Energie wird mit jedem gegessenen Grasfleck um 10 Energiepunkte erhöht. Wir müssen also in der Syntax darauf achten, dass die Deklaration `energy` auch in der Addition auftaucht. Den Verbrauch der Energie fügen wir in die Prozedur `move-turtles` ein.

✓ #18 Um Energie bei der Bewegung zu verbrauchen, lautet der zusätzliche Code:

```
to move-turtles
  ask turtles [
    right random 360
    forward 1
    set energy energy - 1
  ]
end
```

Noch zeigen Energiegewinn und Energieverlust keine Auswirkungen in der Simulation. Ein negativer Energiehaushalt sollte daher realistische Auswirkungen auf die simulierten Organismen haben und zum Ableben dieser führen.

✓ #19 Die Prozedur `go` wird um `check-energy` erweitert:

```
to go
  move-turtles
  eat-grass
  check-energy
  tick
end
```

✓ #20 Die Prozedur `check-energy` wird in den Code eingefügt:

```
to check-energy
  ask turtles [
    if energy <= 0 [
      die
    ]
  ]
end
```

Haben wir den Code mit seiner Bedingungsabfrage richtig eingefügt, können wir nun in unserem Modell nicht nur beobachten, wie mit dem Verhalten der Insekten das Gras verschwindet, sondern auch, wie sich die Anzahl der Insekten reduziert und diese langsam aussterben. Das Kommando `die` ist ein fester Bestandteil der NetLogo-Bibliothek. Bei mehreren Versuchen ist zu erkennen, dass meistens etwas Gras in der Mikrowelt übrig bleibt. Damit unsere Mikrowelt etwas länger überlebt, fügen wir zunächst dem Gras noch die Möglichkeit der Reproduktion hinzu.

- ✓ #21 Die Prozedur go wird um regrow-grass erweitert:

```

to go
  move-turtles
  eat-grass
  check-energy
  regrow-grass
  tick
end

```

- ✓ #22 regrow-grass funktioniert über eine Zufallsabfrage:

```

to regrow-grass
  ask patches [
    if random 100 < 3 [
      set pcolor green
    ]
  ]
end

```

Für das Nachwachsen des Grases wird eine Zufallsziffer aus 100 gezogen und wenn diese kleiner 3 ist, dann wächst Gras auf dem jeweiligen Patch nach. Die Prozedur wird also für jeden patch einzeln durchgeführt und die Chance des Nachwachsens liegt bei 3 %, weil drei Ziffern (0, 1, 2) möglich sind. In der Simulation läuft das System nun wieder sehr stabil weiter. Abschließen wollen wir unser kleines Ökosystem mit der Möglichkeit der Vermehrung der Insekten.

- ✓ #23 Die Prozedur reproduce wird in go eingefügt:

```

to go
  move-turtles
  eat-grass
  check-energy
  reproduce

  regrow-grass
  tick
end

```

- ✓ #24 Der Code zur Prozedur reproduce:

```

to reproduce
  ask turtles [
    if energy > 50 [

```

```

        set energy energy - 50
        hatch 1 [
            set energy 50]
    ]
end

```

Die Reproduktion der Insekten wird zunächst nur an ihren Energiehaushalt gekoppelt, sodass die Bedingungsabfrage ab einem Energieniveau von über 50 Energieeinheiten diese dem Individuum abzieht und dafür ein neues Individuum mittels dem Kommando *hatch* in die Microwelt setzt, das die abgezogenen Energieeinheiten ererbt. Auch wenn dieses Verhalten eher einem Immobilienmarkt ähnelt, haben wir ein kleines Ökosystem erzeugt, auch Räuber-Beute-System genannt. Es wäre natürlich praktisch, das Verhalten unseres Ökosystems über weitere Darstellungsformen beobachten zu können.

Werte, Daten und Monitore

Das Interface bildet bislang im Fenster View unsere Microwelt als beobachtbares System zweidimensional ab. Bestimmte Informationen aber auch über andere Parameter abzurufen, erweist sich oft als sehr praktisch. Bislang ist es z. B. sehr umständlich die Anzahl der Insekten und die Anzahl der Grasflächen zu zählen. Auch wenn wir mit 100 Insekten jedes Mal starten, so wäre es interessant zu wissen, um welche Größe das System nach der Startphase oszilliert. Diese Arbeit können uns sog. *Monitore* abnehmen. Wir werden zwei Monitore auf unserer Oberfläche erstellen.

- ✓ #25 Auf dem Interface nutzen wir das gleich Drop-down-Menü, mit dem wir unsere Buttons erstellt haben, um dort die Funktion Monitor zu aktivieren. Wir pressen den Button Add und platzieren den Monitor an einer geeigneten Stelle auf unserer Oberfläche.
- ✓ #26 Es öffnet sich die Dialogbox in deren Feld wir folgenden Code eingeben:

```
count turtles
```

Die Funktion des Zählens wird über das Kommando *count* aktiviert, dem dann noch übergeben wird, was zu zählen ist. In diesem Fall wird die aktuelle Anzahl aller Agenten *turtles* verlangt; man bezeichnet dies in NetLogo als *agentset*.

Der zweite Monitor soll das Gras zählen. Wir wissen, dass dieses durch grüne Kacheln symbolisiert ist. Anstatt nun alle Kacheln zählen zu lassen, was zu einer statischen Angabe im Monitor führen würde (bei Standardeinstellungen 1.089 patches), werden wir uns auf die grünen Kacheln konzentrieren. Diese weisen bislang augenscheinlich eine Fluktuation auf.

- ✓ #27 Wir erstellen wieder einen Monitor, in dessen Dialogbox wir nun folgenden Code eingeben:

```
count patches with [pcolor = green]
```

- ✓ #28 Zusätzlich geben wir einen Displaynamen im Namensfeld ein: count gras

Hätten wir keinen Displaynamen eingegeben, würde NetLogo den vollständigen Befehl als Monitornamen anzeigen. Durch den Zusatz `with` können wir uns auf die grünen Kacheln konzentrieren, die uns als Nahrungsquelle im Ökosystem interessieren. Wir brauchen nun also nicht mehr zu versuchen, die Anzahl der Räuber und der Beute im System zu zählen, sondern wir können diese numerische Information bequem von den Monitoren ablesen.

Numerische Werte können auch direkt im *Viewer* angezeigt werden. Dies ist allerdings nicht immer übersichtlich und vielleicht nur für wenige Situationen von Interesse, sodass wir eine solche Möglichkeit über einen Schalter gezielt aktivieren wollen. In unserem Beispiel könnte es interessant sein, hin und wieder den Energiewert unserer Insekten zu betrachten. Wir wollen diesen Wert über einen Schalter aktivieren.

- ✓ #29 Ebenso wie einen Button legen wir auch einen Schalter bzw. Switch über den Button Add und dem Drop-down-Menü an. Dieser wird im Interface an eine passende Stelle gelegt.

- ✓ #30 In das Feld Global variable der Dialogbox geben wir folgenden Code ein:

```
show-energy?
```

Nachdem wir das Interface vorbereitet haben, muss nun noch der Code der Simulation an der passenden Stelle erweitert werden.

- ✓ #31 Im Code erweitern wir die Prozedur `eat-grass`:

```
to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy energy + 10
    ]
    ifelse show-energy?
      [set label energy]
      [set label ""]
  ]
end
```

Der Code zu dieser Funktion ist insofern interessant, als dass er uns weitere Informationen zur Syntax von NetLogo liefert. Das Kommando `ifelse` funktioniert ähnlich wie das Kommando `if`, nur dass es für beide möglichen Fälle bestimmte Kommandos angibt. Für den Wert `TRUE` wird bei den Insekten der aktuelle Wert der Energie als Label angeführt. Wenn allerdings der Schalter zu `show-energy?` auf `off` gesetzt ist, dann wird der Wert `FALSE` übergeben. In diesem Fall wird dem Label ein leerer String, in Form von zwei Anführungszeichen, übergeben. Wie in vielen Programmiersprachen, ist ein String eine Sequenz von Buchstaben, Ziffern oder anderen Charakteren, der zwischen Anführungszeichen gesetzt übergeben wird.

Wir sehen anhand der Monitore zur Anzahl von Insekten und Gräsern und des Energiewertes, wie viele Daten in unserem Ökosystem permanent anfallen. Wollen wir unser System allerdings in seinem Verhalten genauer analysieren, wäre eine Aufzeichnung der wesentlichen Daten sinnvoll. Eine solche Funktion haben wir in NetLogo bereits im vorherigen Kapitel zur Modell-Bibliothek kennengelernt, nun wollen wir diese selbst anlegen.

Plotting

Das Aufzeichnen von Daten erfolgt in NetLogo sehr komfortabel in einem Zeitdiagramm *Plot*, das genauso wie ein Button oder ein Monitor direkt im Interface angelegt wird.

- ✓ #29 Wir legen über den Button Add ein Plot auf die Oberfläche.
- ✓ #30 Wir fügen in das Feld Name den Namen Anzahl ein und benennen die X-Achse (X axis label) als Zeit und die Y-Achse (Y axis label) als Summe.
- ✓ #31 In der Tabelle vergeben wir in der Spalte Pen Name für den Wert default den Namen Insekten und in der Spalte Pen Update Commands sollte folgender Code stehen:

```
plot count turtles
```

- ✓ #32 Wir nutzen den Button Add Pen
- ✓ #33 In der neuen Zeile wählen wir im Feld Color einen grünen Farbton (63) aus, schreiben in die Spalte Pen Name den Namen Gras und geben in die Spalte Pen Update Commands folgenden Code ein:

```
plot count patches with [pcolor = green]
```

- ✓ #34 Wir schließen den Vorgang mit dem Button OK ab

Wir haben nun unseren ersten Zeitdiagramm in unserer Oberfläche angelegt und können dem Entwicklungsverlauf unseres Ökosystems in seiner Historie am Bildschirm folgen. Auch wenn wir einen längeren Zeitverlauf beobachten, reicht unser Fenster aus, da wir die Funktion Auto Scale in der Dialogbox zu unserem Diagramm aktiviert gelassen haben, so skaliert sich die X-Achse automatisch zur Zeitreihe. Ebenso hatten wir die Default-Werte für die Größe unseres Diagramms belassen, hätten aber beide Achsen nach unseren Wünschen anpassen können. Dies kann jeder Zeit nachträglich in der Dialogbox geändert werden.

Wollen wir unterschiedliche Verläufe in unserem Zeitdiagramm vergleichen, kann es sich lohnen, exakt gleichlange Simulationen aufzuzeichnen. Für unser Szenario wäre es z. B. interessant, wie der Augenblick streut, in dem die Anzahl der Räuber die der Beute übertrifft. Eine einfache Möglichkeit bietet sich darin, die Anzahl der Ticks direkt in der Prozedur go anzulegen.

- ✓ #35 Die Anzahl der Ticks kann direkt in der Prozedur `go` bestimmt werden, hierzu erweitern wir den Code folgendermaßen:

```

to go
  if ticks >= 100 [stop]
  move-turtles
  eat-grass
  check-energy
  reproduce
  regrow-grass
  tick
end

```

Wir haben uns also dazu entschieden, die Prozedur `go` bis zu 100 *Ticks* laufen zu lassen und anschließend die Simulation zu stoppen. Jeder beliebige Zeitraum ist hier natürlich denkbar. Wir sehen insgesamt in unseren Aufzeichnungen, dass sich die Varianzen in sehr engen Bereichen halten und sich die Verläufe sehr ähneln. Für ein wenig mehr Experimentierfreude können sich kleine Erweiterungen auf unserer Oberfläche anbieten.

2.4.4 Der letzte Feinschliff für das Explorieren

Bislang haben wir in das Verhalten des Ökosystems über den Code eingegriffen. Um die Interaktion mit dem Simulationsmodell zu erhöhen, bietet es sich an, bestimmte Variablen über das Interface anzusteuern und die Veränderungen im Monitor und Zeitdiagramm zu betrachten.

Um z. B. die Anzahl der Insekten zu variieren, bietet es sich an, auf der Oberfläche einen Schieberegler anzubringen.

- ✓ #36 Über den Button Add installieren wir einen Slider auf dem Interface.
- ✓ #37 In das Feld Global variable geben wir `number` ein und das Maximum setzen wir auf 200.
- ✓ #38 Im Code der Prozedur `setup-turtles` ersetzen wir die Anzahl 100 durch die Variable `number`.

Wir können nun die Simulation mit einem Insekt oder direkt mit 200 Insekten starten und die Auswirkungen auf den Systemverlauf beobachten.

Natürlich könnten wir auch Interaktiv auf den Energiegewinn durch das Gras einwirken; hierzu legen wir einen weiteren Schieberegler an.

- ✓ #39 Ein weiterer Slider wird über den Button Add eingefügt.
- ✓ #40 Es wird `energy-from-grass` als Name für die globale Variable gewählt und die Werte bei ihrem Default von 0 und 100 belassen.
- ✓ #41 Die Prozedur `eat-grass` erhält folgende geänderte Zeile:

```
set energy (energy + energy-from-grass)
```

Unterschiede im Systemverlauf sind bei diesem Parameter im sehr kleinen Wertebereich von 1 bis 10 besonders spannend. Zuletzt wollen wir noch die Energievererbung bei der Reproduktion variieren.

- ✓ #42 Es wird ein Slider auf dem Interface eingefügt, der für die globale Variable den Namen birth-energy erhält.
- ✓ #43 Es wird der Code der Prozedur reproduce wie folgt geändert:

```
to reproduce
  ask turtles [
    if energy > birth-energy [
      set energy energy - birth-energy
      hatch 1 [
        set energy birth-energy
      ]
    ]
  ]
end
```

Wir haben nun eine Menge Schieberegler für die Exploration des Systemverhaltens in unser Beispiel eingefügt. Mit Sicherheit wäre noch eine Menge weiterer Interaktionen mit dem Modell denkbar und der Anpassung an die Realität sind dem Modellentwickler in NetLogo nahezu keine Grenzen gesetzt. Diese Grenzen ergeben sich manchmal eher durch die verfügbare Hardware als durch die Mächtigkeit der hier vorgestellten Software. Einen letzten Feinschliff wollen wir unserem Ökosystem mit der letzten Zeile Code einfügen und damit das Kapitel beschließen.

- ✓ #44 Wir fügen folgende Zeile in den Code

```
Problemstellung: hier das Shape anpassen :-)
```

Für die eigene Erstellung von Simulationen stehen Ihnen nun alle Funktionen offen. Das User Manual zur Software finden Sie auf der Internetseite <https://ccl.northwestern.edu/netlogo/>.

