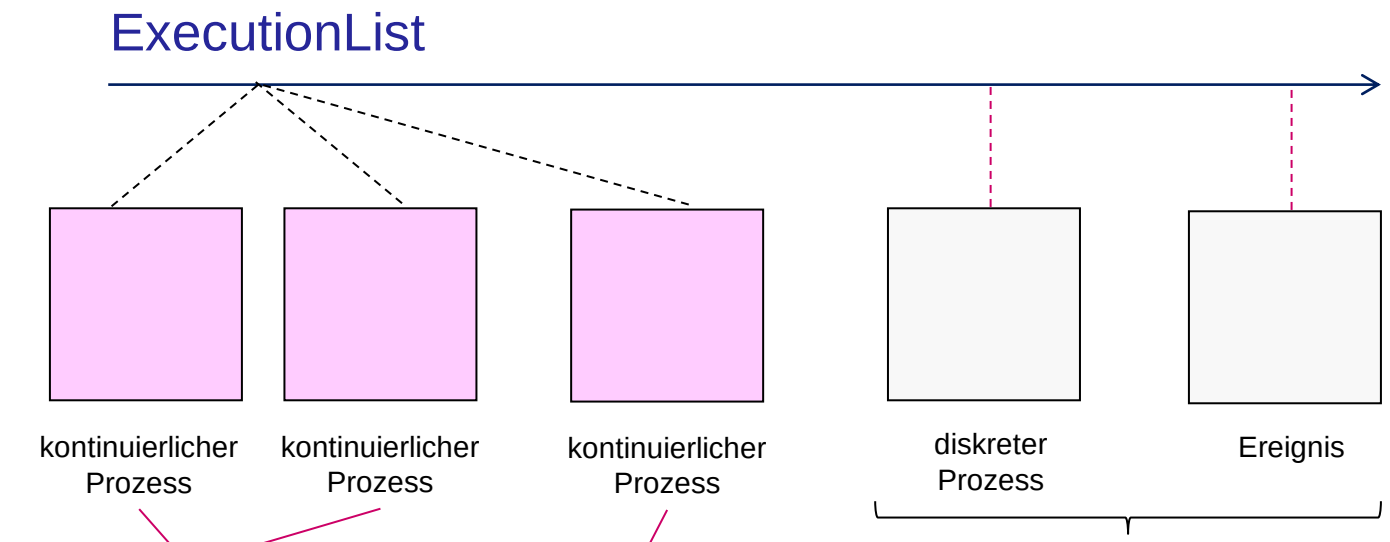
# *Modul OMSI-2*
## *im SoSe 2011*

## *Objektorientierte Simulation mit ODEMx*

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage
Dipl.-Inf. Andreas Blunk

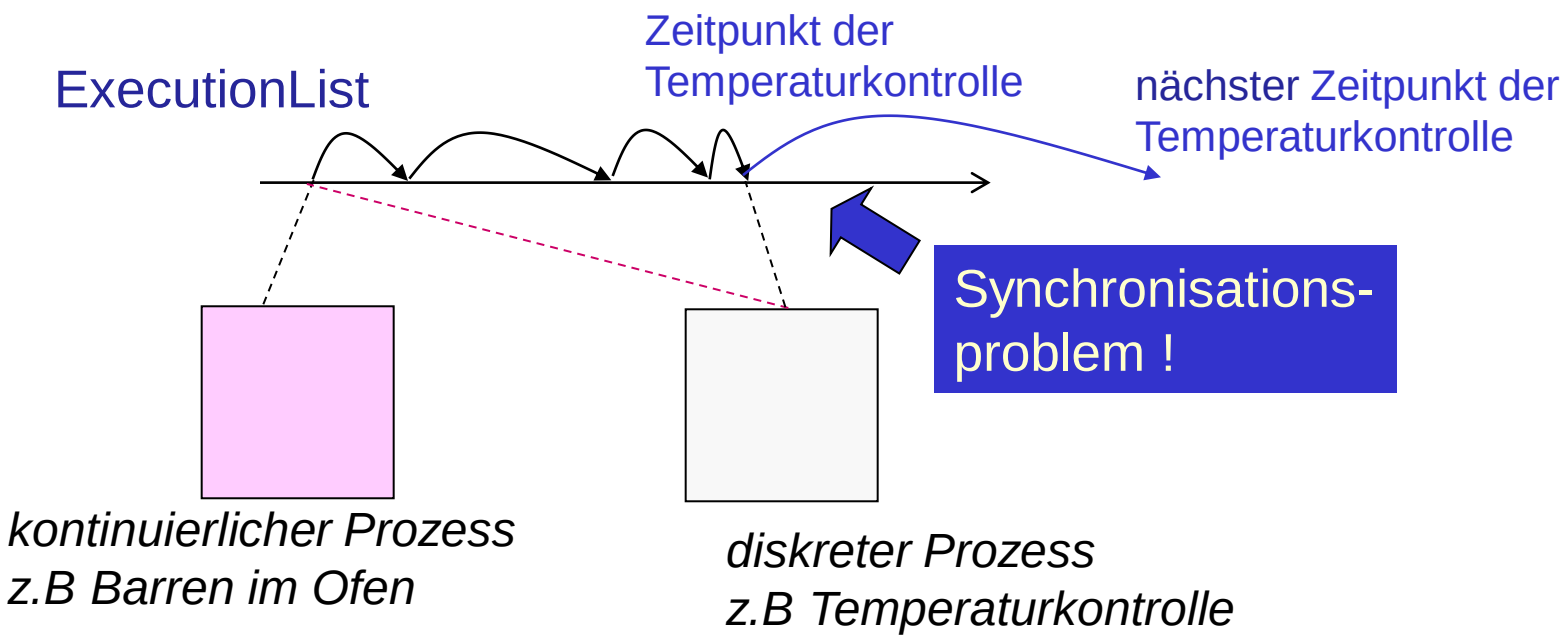
fischer|ahrens|eveslage|blunk@informatik.hu-berlin.de

# Zustandsänderungsarten in ODEMx (Wdh.)



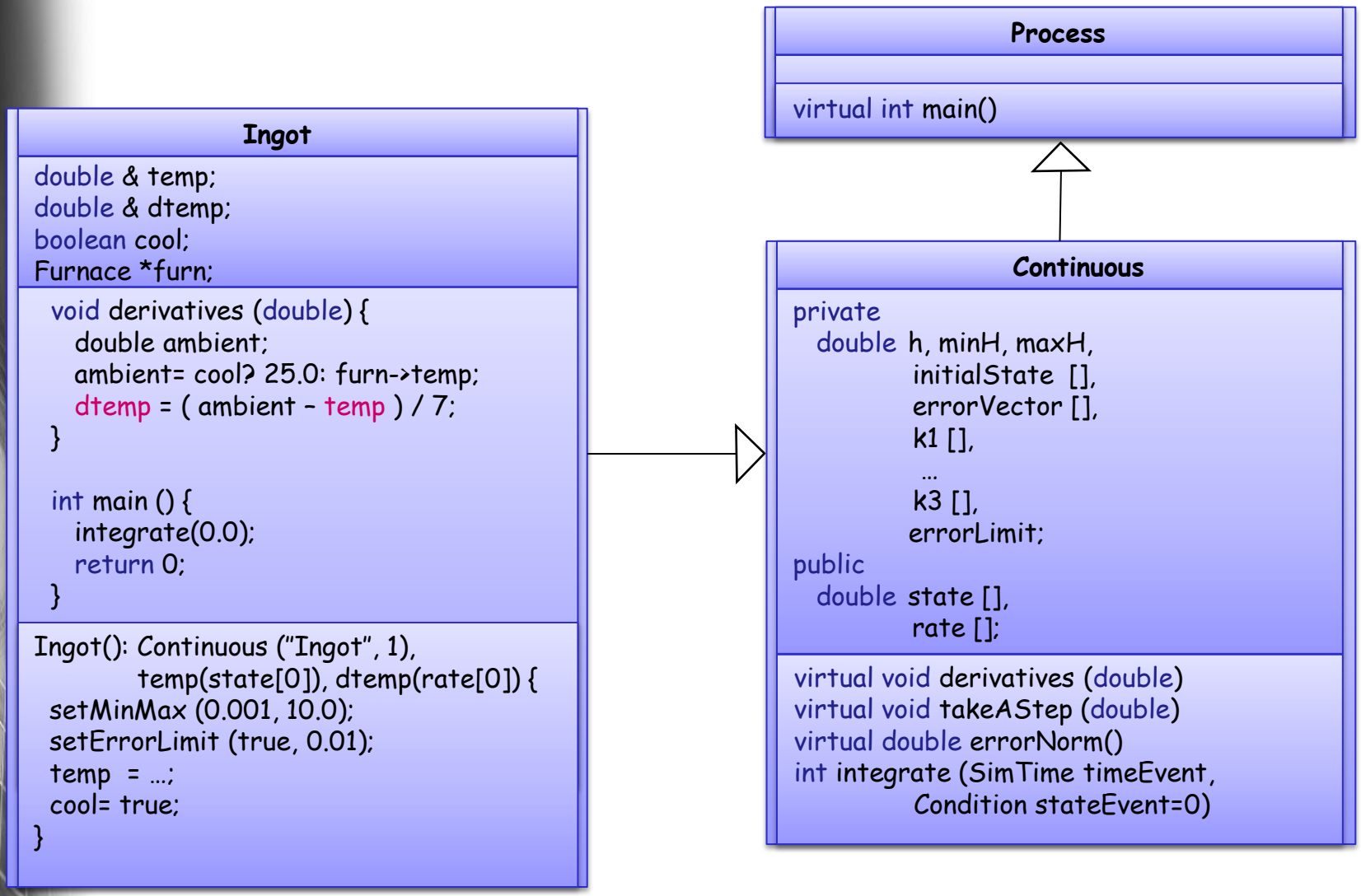
# Synchronisationsproblem (Wdh.)

1. Diskrete Prozesse/Ereignisse sind höher priorisiert als kontinuierliche Prozesse
2. Kontinuierliche Prozesse passen ihre Schrittweite dynamisch an, um Ereigniszeiten diskreter Prozesse exakt zu treffen

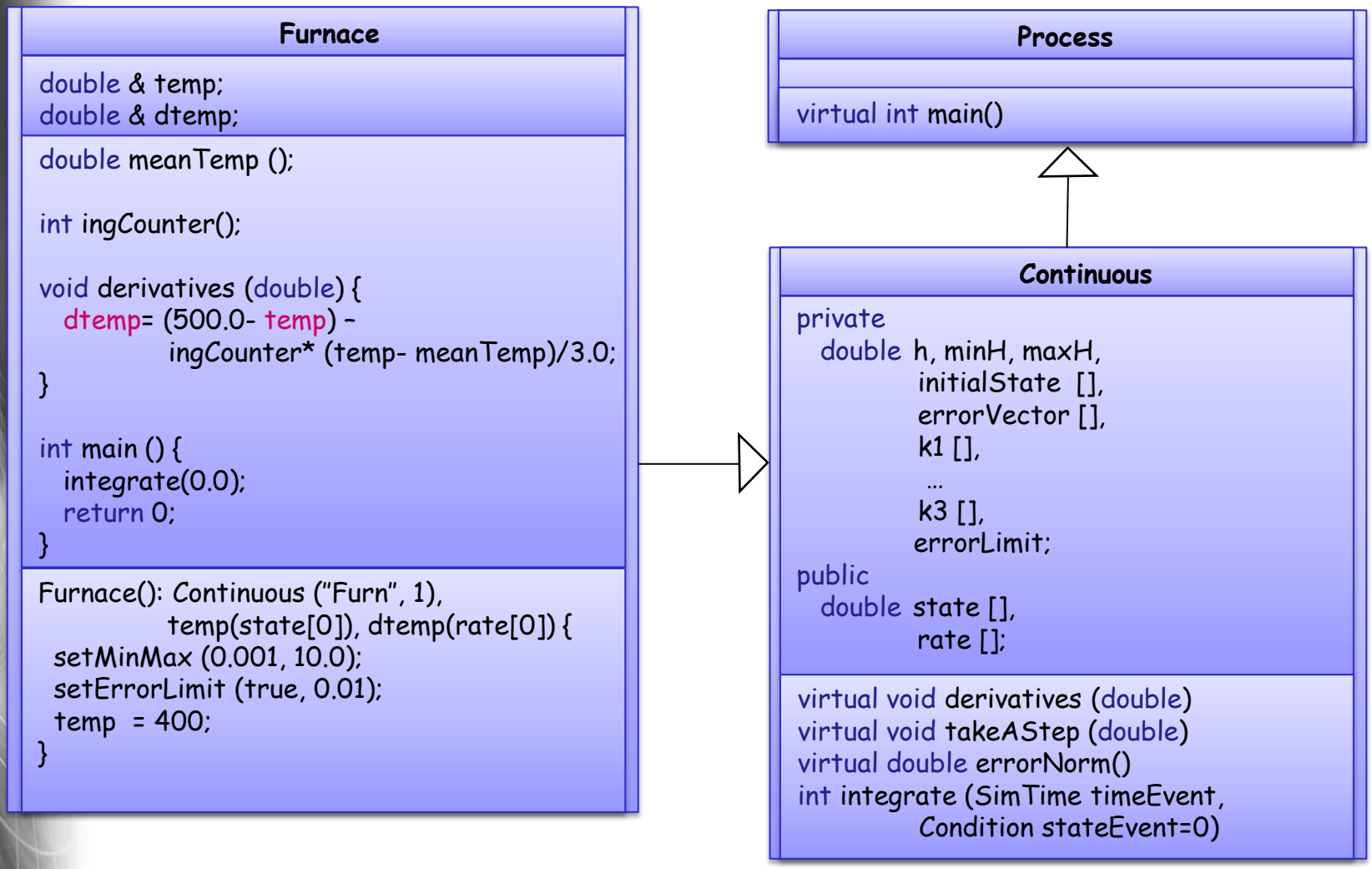



Fazit: Jeder Barren (Continuous-Objekt) im Ofen und der Ofen (Continuous-Objekt) selbst werden mit dem diskreten Prozess der Temperaturkontrolle synchronisiert

# Die Klasse Ingot (Eisenbarren)



# Die Klasse Furnace (Tiefofen)

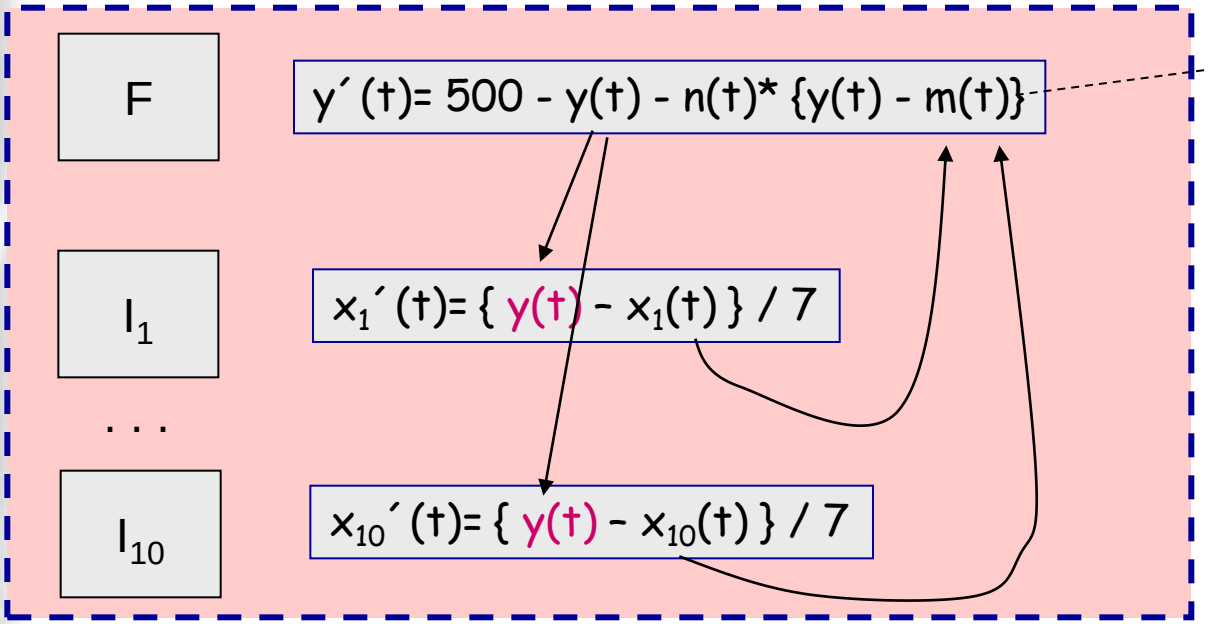


# 1. Behandlung zeitkontinuierlicher Prozesse

1. Scheduling von Process- und Continuous-Objekten
2. DGL, Richtungsfeld(Phasenraum), Trajektorie, Zustandsraum
3. Näherungsverfahren, Diskretisierungs/Verfahrensfehler
4. Runge-Kutta-Merson-Verfahren (ODEMx-Bibliothek)
5. Continuous-Implementierungskonzept
6. Ablauf am Beispiel (Barrentemperatur)
7. Synchronisationsprobleme und alternative Modellierungsmöglichkeiten in ODEMx
8. Bestimmung von Zustandsereignissen
9. ODEMx-Beispiel (Doko), graphische Kurvendarstellung

# Synchronisationsprobleme im kontinuierlichen Fall am Beispiel (1)

dynamische Abhängigkeit zeitkontinuierlicher Prozesse




stark gekoppelte Prozesse (da sogar Rückkopplungen)

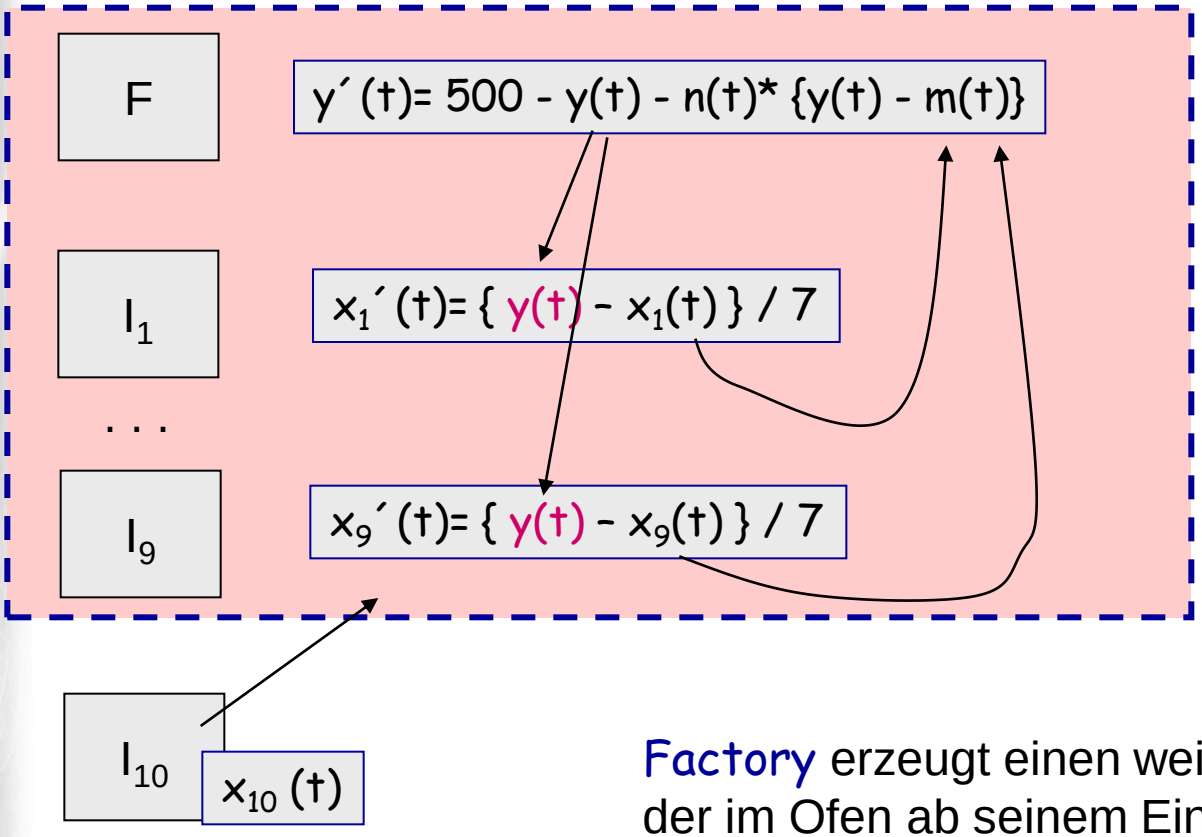
$I_{11}$ $x_{11}´(t)= \{ 25 - x_{11}(t) \} / 7$

$I_{12}$ $x_{12}´(t)= \{ 25 - x_{12}(t) \} / 7$

unabhängige Prozesse

# Synchronisationsprobleme im kontinuierlichen Fall am Beispiel (2)

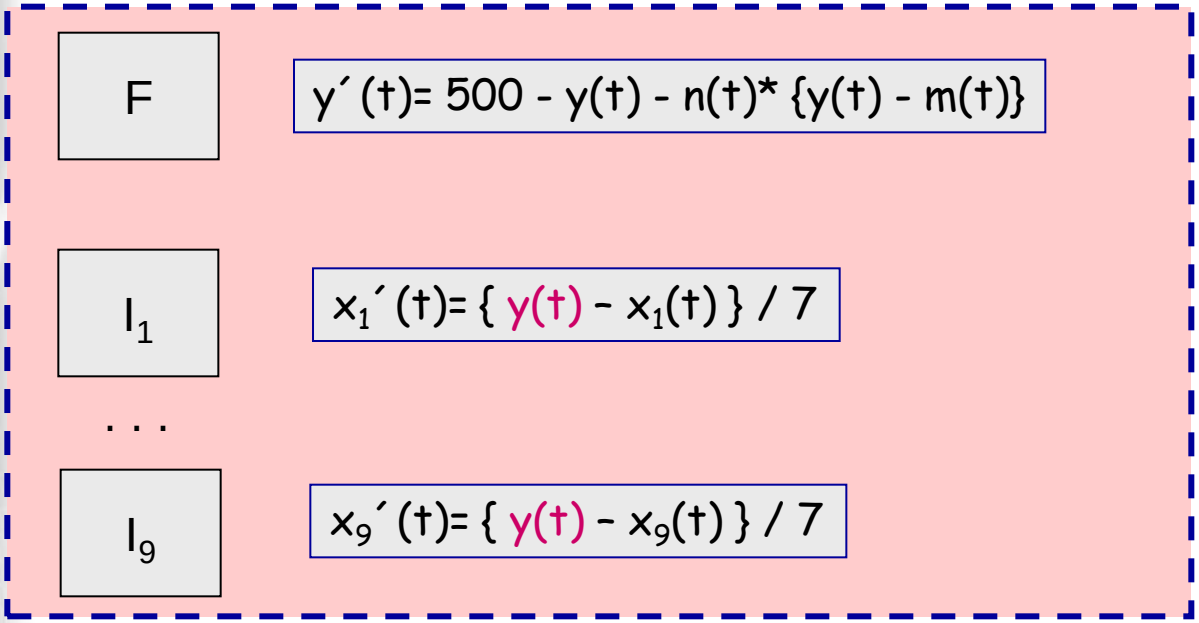
Abhängigkeit zeitkontinuierlicher Prozesse von diskreten Prozessen




Factory erzeugt einen weiteren Barren, der im Ofen ab seinem Eintrittszeitpunkt die kontinuierlichen Prozesse im Ofen beeinflusst

# Synchronisationsprobleme im kontinuierlichen Fall am Beispiel (3)

Abhängigkeit diskreter Prozesse von kontinuierlichen Prozessen




Aktionen des Controller-Prozesses hängen vom Erreichen eines bestimmten Temperaturwertes der Barren im Ofen ab

# Strukturierungsvarianten (1)

**Benutzung eines <u>zentralen</u> Arrays**

– als zentralisierte Speicher von Zustandsgrößen und ihrer Ableitungen zu verschiedenen Zwischenpunkten:
mit jeweils genau einem state-, rate-, $k_1$, …,$k_n$-Vektor

→ d.h. es gibt <u>nur ein</u> Continuous-Objekt für alle zeitkontinuierlichen Vorgänge (Ofen-Temp und aller enthaltener Barren)

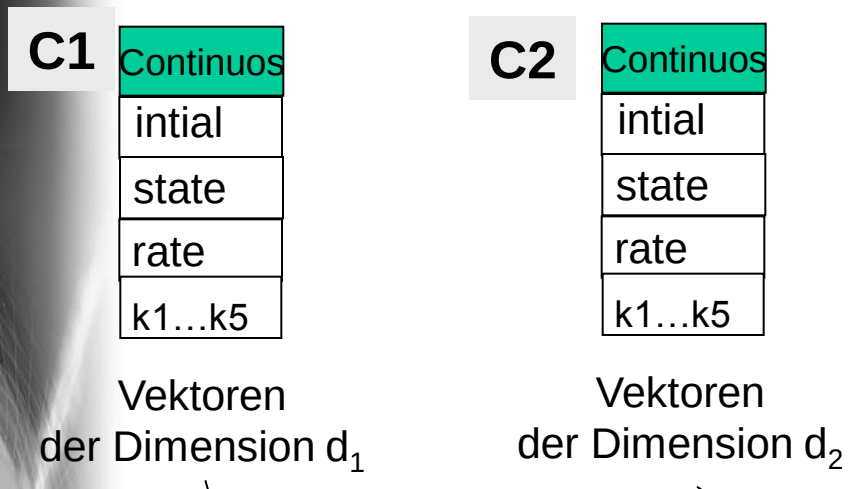
– sichern Effizienz der Berechnung ohne Synchronisationsprobleme, **solange** die Dimension des Zustandsraumes fest bleibt
und Strukturänderungen über Parameteränderungen ausgedrückt werden können

aber ungünstig: Tiefofen-Beispiel
(dynamische Änderung der Dimension,
Barren kommen und gehen)

State[0] - Ofentemperatur
State[1] - Temperatur des 1.Barren
State[i] - Temperatur des i-ten Barren

# Strukturierungsvarianten (2)

## Benutzung dezentraler Arrays




**Vorteil**
- dynamische Anzahl von Continuous-Objekten
- jedes Continuous-Objekt kann individuelle Dimension haben

→ Strukturänderungen sind leichter

**Nachteil**: keine exakte Synchronisation bei gekoppelten Continuous-Objekten

→ numerische Integration wird verfälscht

```
void derivatives( double t) {
      rate[0]= state[1];
      rate[1]= -d*state[1] – C2->state[2];
}
```

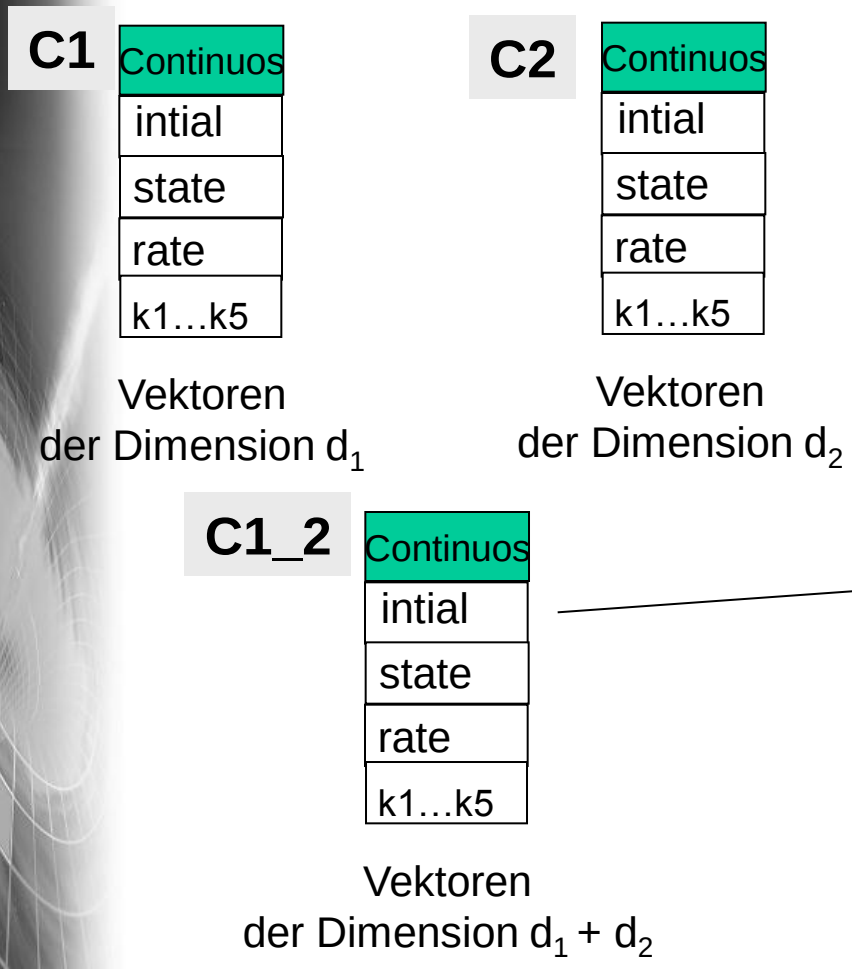
```
void derivatives( double t) {
      rate[0]= - a*state[1];
      rate[1]= - C1->state[1] – state[1];
      rate[2]= - d*state[1] * C1->state[0];
}
```

Dilemma der aktuellen ODEMx-Implementierung

denn: - zentrales GLS sichert a priori die exakte Synchronisation,
- dynamische Strukturänderungen werden dagegen nicht adäquat unterstützt

# Strukturierungsvarianten (3)

## Behebung des Synchronisationsproblems




**Methode**
- ein neues Continuous-Objekt kann ältere dynamisch ersetzen (Strukturänderungen)

**Nachteil**: Übernahme der einzelnen State-Vektoren in Kopie

```
void derivatives( double t) {
    rate[0]= state[1];
    rate[1]= -d*state[1] – state[3];
    rate[2]= - a*state[3];
    rate[3]= - state[1] – state[3];
    rate[4]= - d*state[4] * state[0];
}
```

später
verbesserte ODEMx-Lösung

(Übernahme des bekannten Master-Slave-Prinzips für den zeitkontinuierlichen Kooperationsfall)

# 1. Behandlung zeitkontinuierlicher Prozesse

1. Scheduling von Process- und Continuous-Objekten
2. DGL, Richtungsfeld(Phasenraum), Trajektorie, Zustandsraum
3. Näherungsverfahren, Diskretisierungs/Verfahrensfehler
4. Runge-Kutta-Merson-Verfahren (ODEMx-Bibliothek)
5. Continuous-Implementierungskonzept
6. Ablauf am Beispiel (Barrentemperatur)
7. Synchronisationsprobleme und alternative Modellierungsmöglichkeiten in ODEMx
8. Bestimmung von Zustandsereignissen
9. ODEMx-Beispiel (Doko), graphische Kurvendarstellung

# Abbruch der zeitkontinuierlichen Zustandsänderung

- Wie lange wird die Simulation der Zustandsgrößen ausgeführt?
  → unter welchen Bedingungen bricht integrate ab?

- Abbruchvarianten, die sich überlagern können
  1. Vorgabe eines Zeitintervalls (ab Start der Simulation des Prozesses)
     → Abbruch durch ein Zeitereignis
  2. Vorgabe einer Abbruchbedingung (Funktionszeiger)
     - über die eigenen sich entwickelnden Zustandsgrößen des Prozesses: State-Vektor) oder
     - über beliebigen Zustandsgrößen

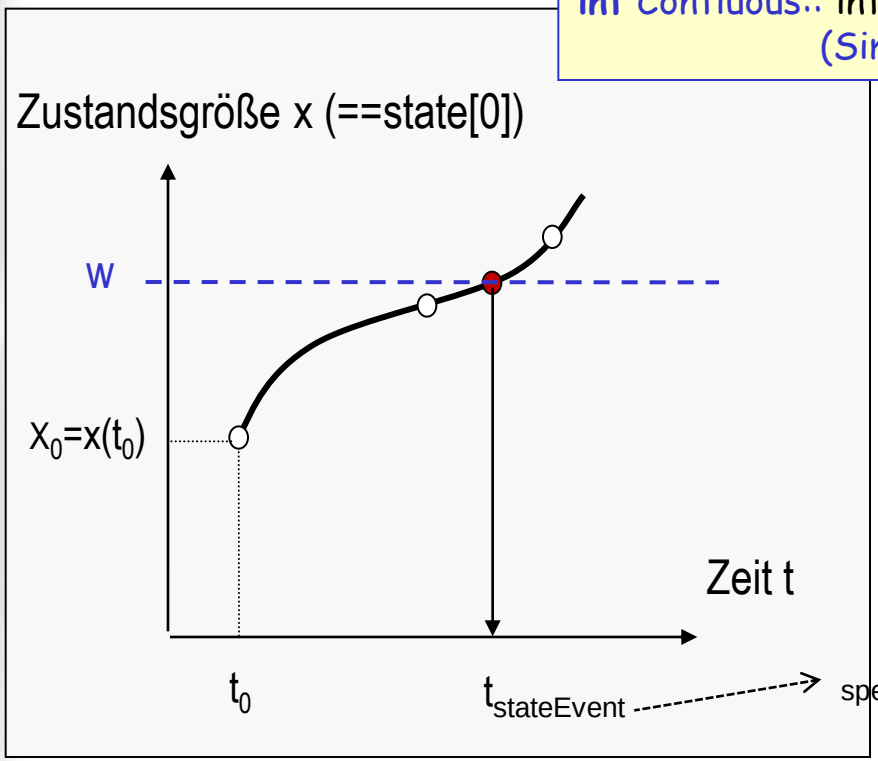
     Bedingung ist nach jedem Integrationsschritt auszuwerten

     → Abbruch durch ein lokales/globales Zustandsereignis
  3. Abbruch durch einen anderen Prozess per Interrupt-Ruf

- Rückgabewert von integrate:
  - 0: Zeitereignis
  - 1: Zustandsereignis
  - 2: Interrupt

# Zustandsereignisbestimmung

```
int Contiuous:: integrate
                (SimTime timeEvent, Condition stateEvent=0);
```




```
typedef bool(Process::*Condition)();
//definiert in Process
```

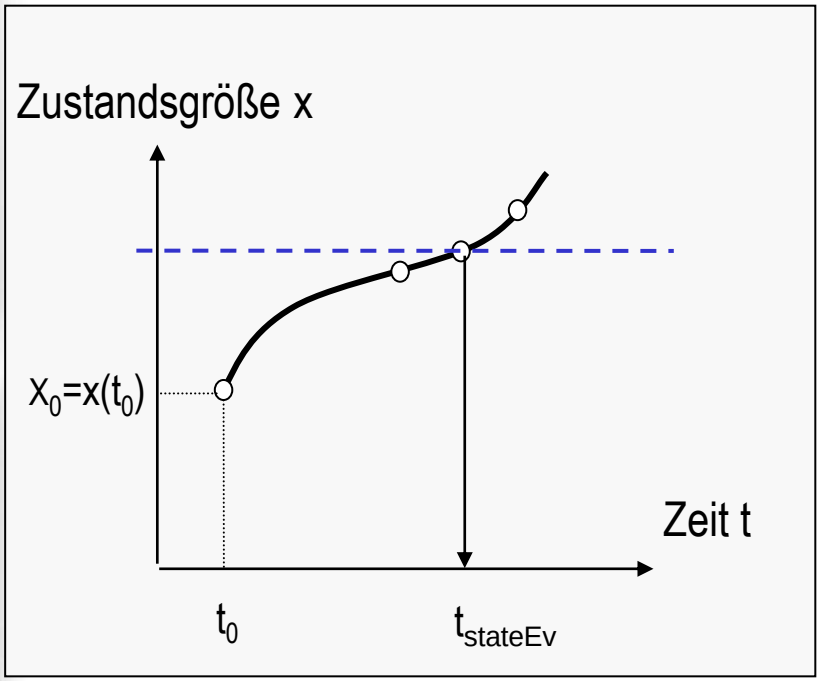
```
bool specialEv() { return state[0] >=w; }
```

```
integrate(20.0, (Condition)&X::specialEv)
```

Näherungsverfahren zur Zustandsereignisbestimmung

1. Intervallschachtelung
2. Interpolation (quadratisches Polynom)

# Intervallschachtelung



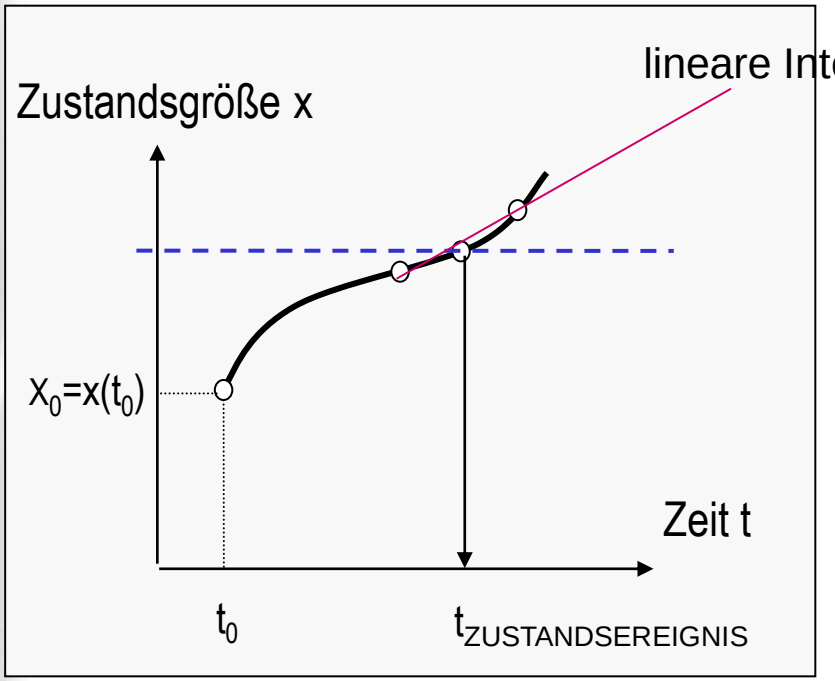

```
bool specialEv() { return state[0] >=w; }
```

```
integrate(20.0, (Condition)&X::specialEv)
```

**möglicher Effekt:**
integrate bricht ab mit erkanntem Zustandsereignis, aber
die Zustandsbedingung ist dennoch nicht erfüllt

- integrate erhält Adresse einer bool-Funktion als Zustandsbedingung
- nach jedem Integrationsschritt (takeAstep) wird Zustandsbedingung ausgewertet
  - erfüllt: Wiederholung des Integrationsschrittes mit halber SW
  - nicht erfüllt: Akzeptanz des Schrittes (Fortsetzung mit dieser SW)
- Abbruch bei Unterschreiten der minimalen SW minh

# Interpolation




integrate benutzt **quadratisches** Interpolationspolynom, da neben
- dem Zustand (vor Zustandsereignis) und
- dem Zustand (nach Zustandsereignis) auch
- die Ableitung des Zustandes (vor dem Zustandsereignis)

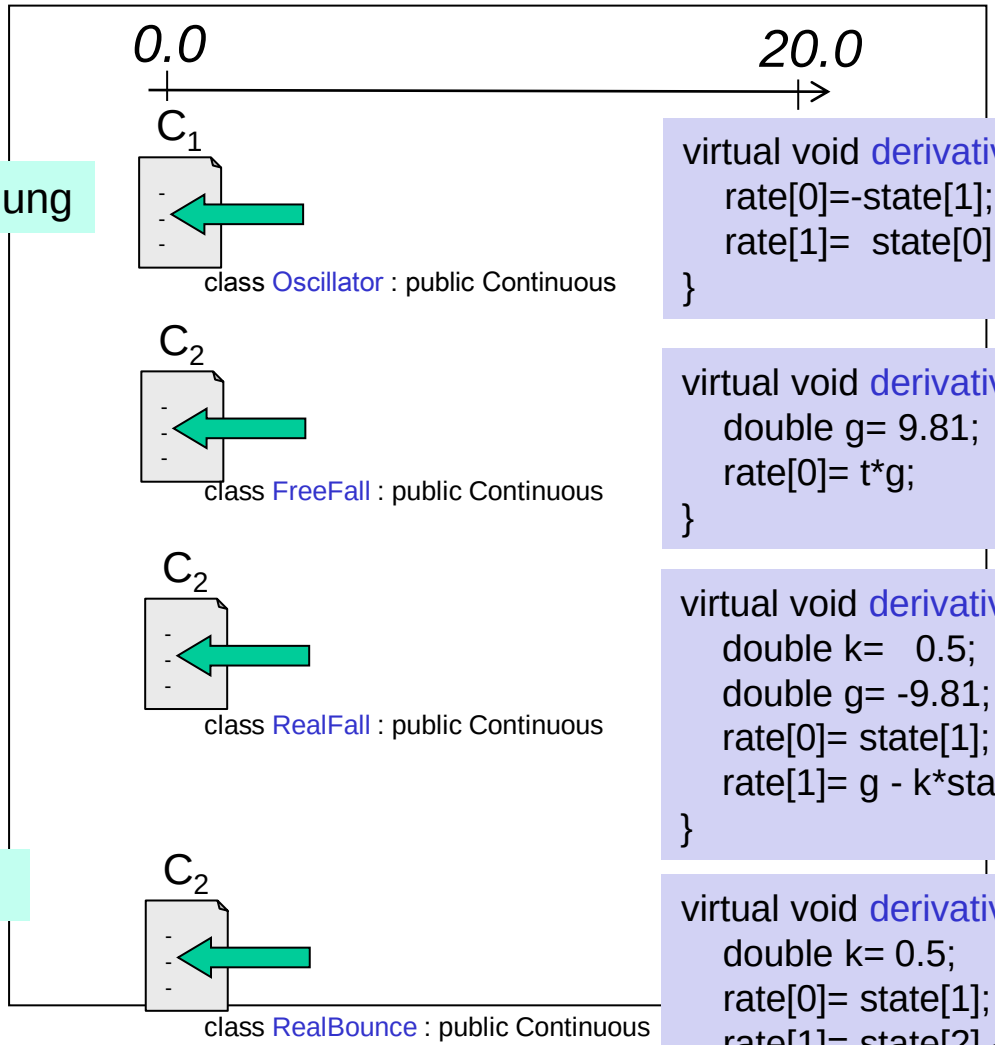
bekannt ist

noch nicht
von ODEM nach ODEMx
portiert

# 1. Behandlung zeitkontinuierlicher Prozesse

1. Scheduling von Process- und Continuous-Objekten
2. DGL, Richtungsfeld(Phasenraum), Trajektorie, Zustandsraum
3. Näherungsverfahren, Diskretisierungs/Verfahrensfehler
4. Runge-Kutta-Merson-Verfahren (ODEMx-Bibliothek)
5. Continuous-Implementierungskonzept
6. Ablauf am Beispiel (Barrentemperatur)
7. Synchronisationsprobleme und alternative Modellierungsmöglichkeiten in ODEMx
8. Bestimmung von Zustandsereignissen
9. ODEMx-Beispiel (Doko), graphische Kurvendarstellung

# Beispiel (ODEMx-Doko)




sin/cos-Schwingung

```
virtual void derivatives (double t) {
    rate[0]=-state[1];
    rate[1]=  state[0];
}
```

freier Fall

```
virtual void derivatives (double t) {
    double g= 9.81;
    rate[0]= t*g;
}
```

Zeitabhängigkeit

Fall mit Luftwiderstand

```
virtual void derivatives (double t) {
    double k=   0.5;
    double g= -9.81;
    rate[0]= state[1];
    rate[1]= g - k*state[1];
}
```

springender Ball

```
virtual void derivatives (double t) {
    double k= 0.5;
    rate[0]= state[1];
    rate[1]= state[2] - k*fabs(state[1]);
    rate[2]= 0.0; }
```

Folge von Zustandsereignissen

# Continuous-Objekte und Trace

```
class Oscillator : public Continuous {
      double k;
public:
      Oscillator( double pk) :
         Continuous ( getDefaultSimulation(), "Oscillator", 2),
         k(pk) {};

protected:
      virtual int main() {
                state[0]=1;
                state[1]=0;

                setStepLength(0.01, 0.1);
                setErrorlimit(0, 0.1);

                integrate(20.0, 0);

                return 0;
      }

      virtual void derivatives (double t) {
                //never use 'getCurrentTime()' here
                //to get the time.
                rate[0]= - k *state[1];
                rate[1]= state[0];
      }
};
```

```
int main( int argc, char* argv[]) {
                Oscillator osci1 (100.0);
                Oscillator osci2 (200.0);

                ContuTrace tracer[] = { ContuTrace(&osci1),
                                        ContuTrace(&osci2)
                                      };
                osci1.activate();
                osci2.activate();

                getDefaultSimulation()->run();

                return 0;
}
```

es werden zwei Trace-Ausgabe-Files generiert (eins je *Continuous*- Objekt)

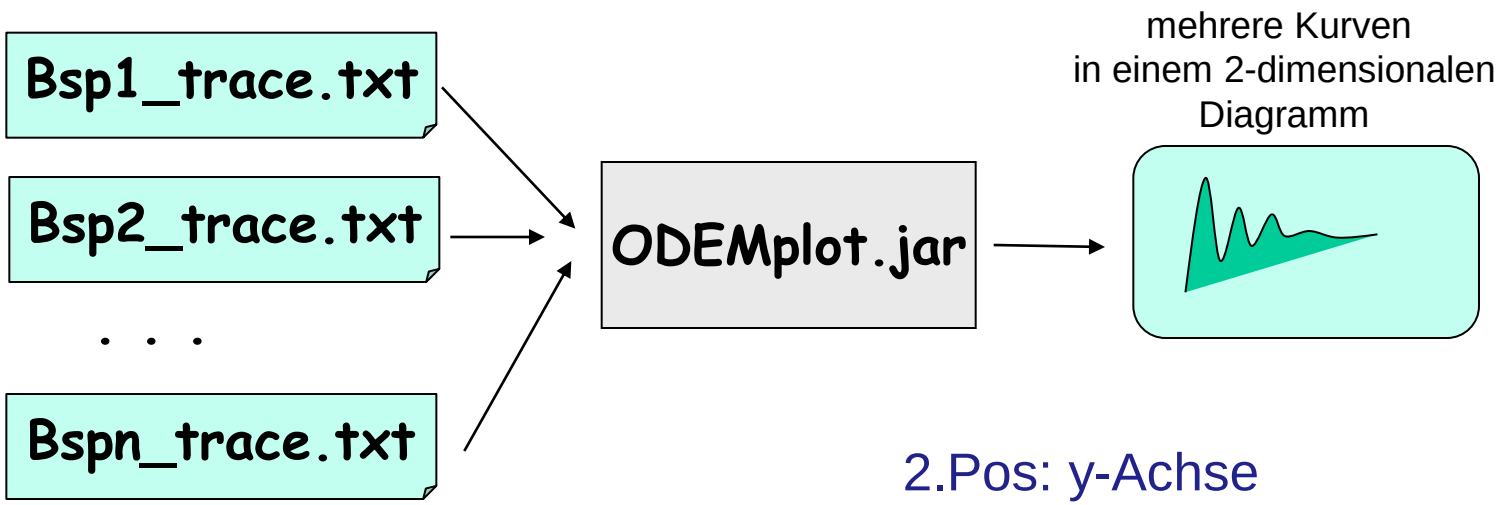
- "Oscillator-1_trace.txt"
- "Oscillator-2_trace.txt"

# ODEMx Trace-File

T R A C E  F I L E  F O R  Oscillator-1

| Time | Steplength | Error | state[0] | state[1] | rate[0] | rate[1] |
|---|---|---|---|---|---|---|
| 0.1000000 | 0.1000000 | 0.0000000 | 0.9950042 | 0.0998334 | -0.0998333 | 0.9950042 |
| 0.2000000 | 0.1000000 | 0.0000000 | 0.9800666 | 0.1986693 | -0.1986692 | 0.9800666 |
| 0.3000000 | 0.1000000 | 0.0000000 | 0.9553365 | 0.2955202 | -0.2955201 | 0.9553365 |
| 0.4000000 | 0.1000000 | 0.0000000 | 0.9210610 | 0.3894183 | -0.3894182 | 0.9210610 |
| 0.5000000 | 0.1000000 | 0.0000000 | 0.8775826 | 0.4794255 | -0.4794254 | 0.8775826 |
| 0.6000000 | 0.1000000 | 0.0000000 | 0.8253357 | 0.5646424 | -0.5646423 | 0.8253357 |
| 0.7000000 | 0.1000000 | 0.0000000 | 0.7648423 | 0.6442176 | -0.6442176 | 0.7648423 |
| 0.8000000 | 0.1000000 | 0.0000000 | 0.6967068 | 0.7173560 | -0.7173560 | 0.6967068 |
| 0.9000000 | 0.1000000 | 0.0000000 | 0.6216101 | 0.7833268 | -0.7833268 | 0.6216101 |
| 1.0000000 | 0.1000000 | 0.0000000 | 0.5403024 | 0.8414709 | -0.8414709 | 0.5403025 |

# Hinweise: Graphische Ausgabe (Zeitverhalten)




2.Pos: y-Achse

1.Pos: x-Achse

```
java -jar ODEMPlot.jar 2    Bsp1_trace.txt 01    Bsp2_trace.txt 01
```

Dimension

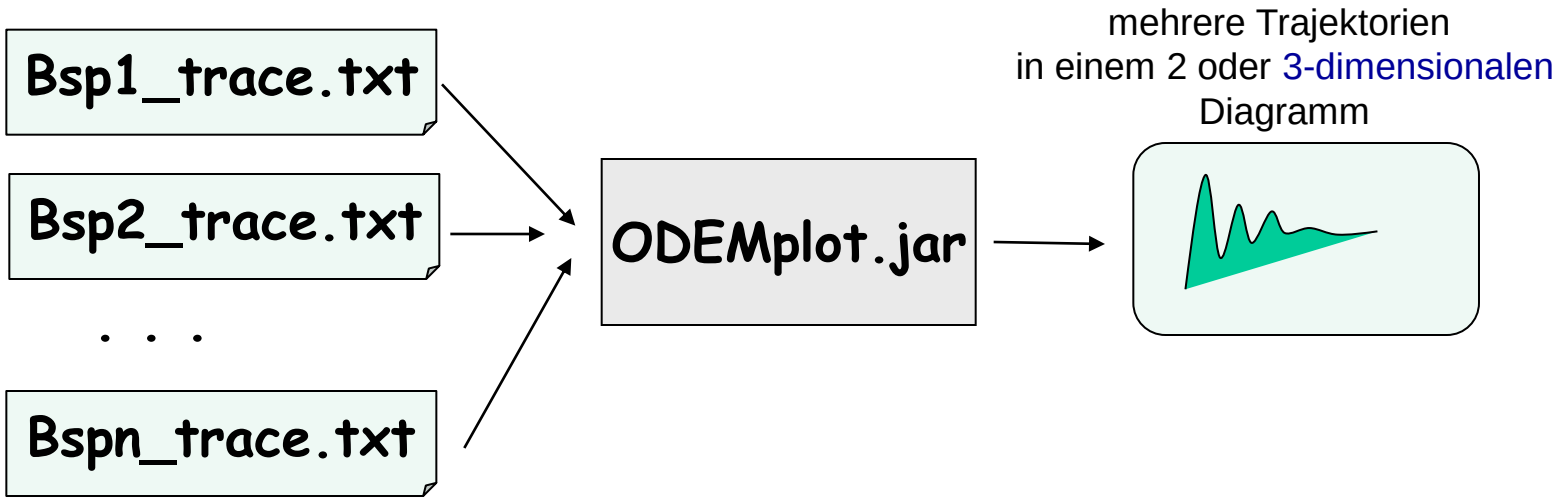
0→Zeit

1→state[0]

Überlagerung zweier Zustandskurven unterschiedlicher Continuous-Objekte bei

- individueller Skalierung der einzelnen Kurven
- Koordinatenermittlung

# Hinweise: Graphische Ausgabe (Zustandsraum)



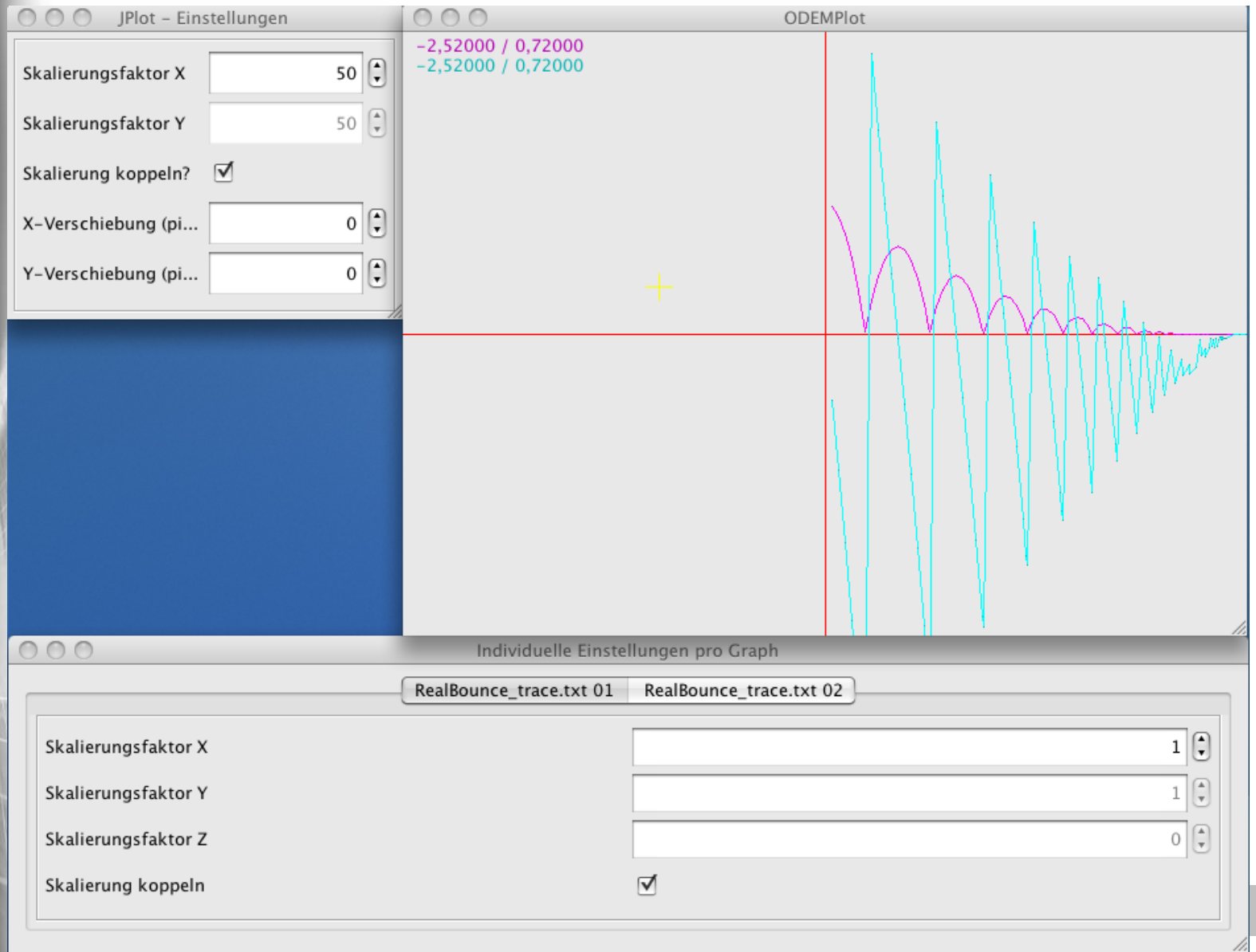

```
java -jar ODEMPlot.jar 2     Bsp1_trace.txt 12
```

Dimension

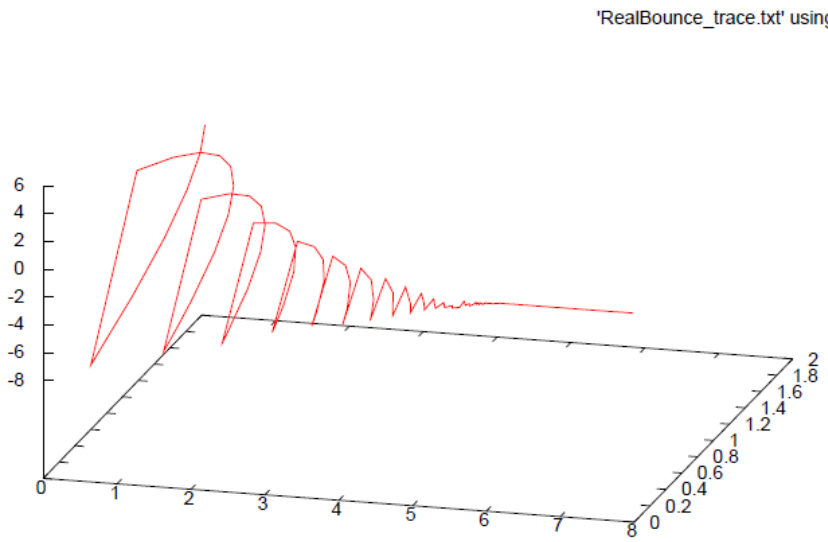
state[0]
state[1]

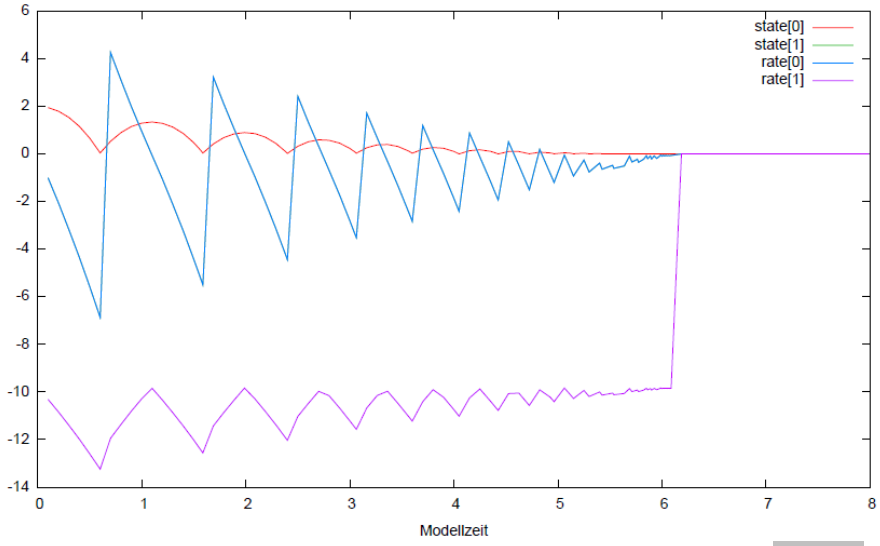
Phasendarstellung bei
- individueller Skalierung der einzelnen Kurven
- Koordinatenermittlung
- möglicher Wechsel der Beobachterposition (im 3-dim. Fall)

# ODEMplot


JPlot – Einstellungen
Skalierungsfaktor X
50
Skalierungsfaktor Y
50
Skalierung koppeln?
X-Verschiebung (pi...
0
Y-Verschiebung (pi...
0
ODEMPlot
-2,52000 / 0,72000
-2,52000 / 0,72000
Individuelle Einstellungen pro Graph
RealBounce_trace.txt 01
RealBounce_trace.txt 02
Skalierungsfaktor X
1
Skalierungsfaktor Y
1
Skalierungsfaktor Z
0
Skalierung koppeln

# GNUplot


'RealBounce_trace.txt' using 1:4:5
6
4
2
0
-2
-4
-6
-8
0
1
2
3
4
5
6
7
8
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


state[0]
state[1]
rate[0]
rate[1]
6
4
2
0
-2
-4
-6
-8
-10
-12
-14
0
1
2
3
4
5
6
7
8
Modellzeit

# 2. Analyse zeitkontinuierlicher Systeme

1. Wirkungsstrukturen
2. Beispiel: Einfaches Räuber-Beute-System
3. Gleichgewichtspunkte und ihre Stabilität
4. Systematische Betrachtung von Systemstrukturen
5. Stabilitätsanalyse linearer und nichtlinearer Systeme
6. Analytische Stabilitätsanalyse (Prinzip)
7. Besonderheiten nichtlinearer Systeme

# *Die Wirkungsstruktur bestimmt Zustandsänderung*

## Niedrigtemperaturofen-Beispiel:

**zwei Ursachen für Zustandsänderungen**

*System*

- Einwirkungen von außen
  ankommender Barren
- Einwirkungen von innen/
  Verhalten der Objekte im System selbst
  - Ofentemperatur ändert sich
  - Barren werden von Abkühlungs- in Erwärmungszustand überführt
  - Barren werden aus dem Ofen entfernt

*System-Komponente (Ofen)*

- Einwirkungen von außen
  Barrenzuführung und Entnahme
- Einwirkungen von innen/
  Verhalten der Objekte im System selbst
  - Temperaturänderungen sind von aktuellen Temperaturen des Ofens und der Barren abhängig

# *Die Wirkungsstruktur bestimmt Zustandsänderung (2)*

**allgemeine Annahmen**

(1) Einwirkungen auf das System sind unabhängig von Systemreaktion
→ Rückkopplungsfreiheit von der Umgebung (sinnvolle Forderung)

falls
- Anzahl wartender Barren vor dem Ofen oder
- andere Zustandsgrößen

Einfluss auf die Ankunftsintensität der Barren hätte, wäre das System nicht rückkopplungsfrei

(2) zwischen den Systemkomponenten bestehen Kopplungen und Rückkopplungen
→ diese bestimmen zusammen mit den Kopplungen zur Systemumgebung die Wirkungsstruktur des Systems

# Bestimmung des Verhaltens

**wichtiger Sachverhalt**:

Der Zustandsvektor eines deterministischen Systems (und damit jede Zustandsgröße) kann zu jedem Zeitpunkt $t > t_0$ eindeutig bestimmt werden, **wenn**

- die Anfangszustände $z(t_0)$ für jede Zustandsgröße
- der Eingangsvektor $u = u(t)$ im Zeitraum $[t_0, t_{max}]$ und
- die Zustandsfunktion $f = f(z,u,t)$ für den Zeitraum $[t_0, t_{max}]$

bekannt sind.

trotzdem kann es u.U. zu chaotischem Verhalten kommen !!!

# *Die Rolle von Rückkopplungen (generell)*

gilt bereits bei einfachsten Systemen

- aus spontaner Beobachtung der Zustandsänderung allein kann man nicht die weitere Zustandsentwicklung ableiten
  Anfangswert und sämtliche Eingänge (für komplettes Beobachtungsintervall) werden benötigt

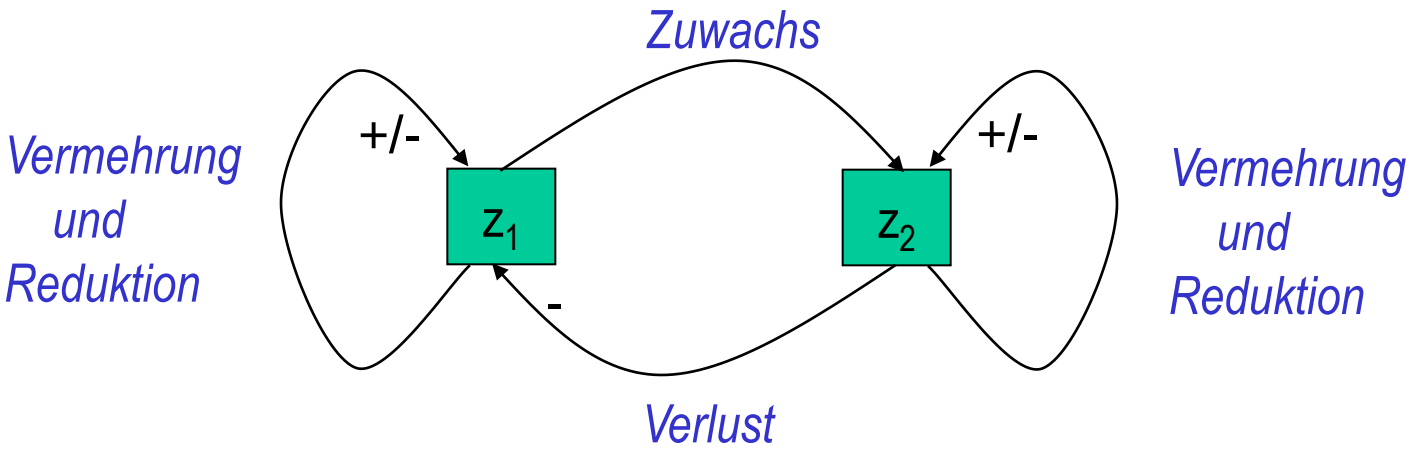
- Systeme mit **mindestens zwei** Zustandsgrößen können bereits innere Rückkopplungen aufweisen:
  Größe *A* verändert den Zufluss von Größe *B* und umgekehrt
  Systemverhalten mit Schwingung, ohne dass diese von außen erzwungen wäre
  (auch Zyklen in volkswirtschaftlichen Prozessen)

- **innere Rückkopplungen** können zu (determiniertem) chaotischen Verhalten führen
  man kann nicht mehr sicher Verhalten vorhersagen, nur noch potentielle alternative Verhaltensbereiche

# Wirkungsstruktur: Kopplungen u. Rückkopplungen

zwei Zustandsgrößen

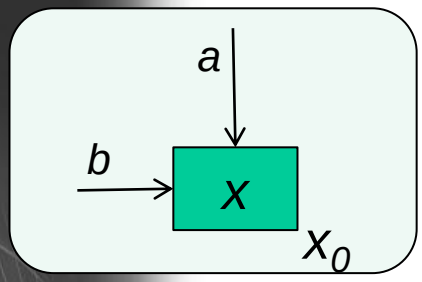
- eine Zustandsgröße beeinflusst die Änderung der jeweils anderen Größe (Kopplung) und umgekeht (Rückkopplung)
- Änderung einer Zustandsgröße ist von sich selbst abhängig (Eigenkopplung)



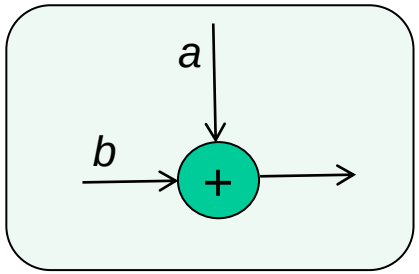

$z_1$ Beutepopulation

$z_2$ Räuberpopulation

# Wirkungsdiagramme, Symbole



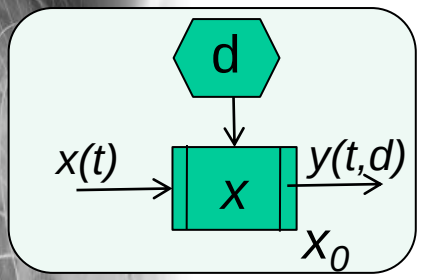

Integration $\frac{dy}{dt} = a + b$



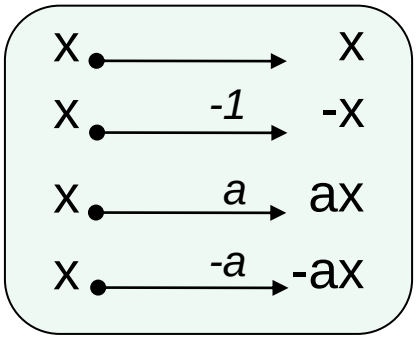

Addition a+b

weitere Operatoren:

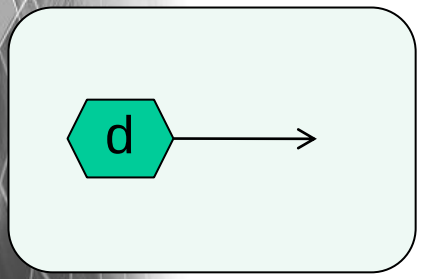
-, *, /



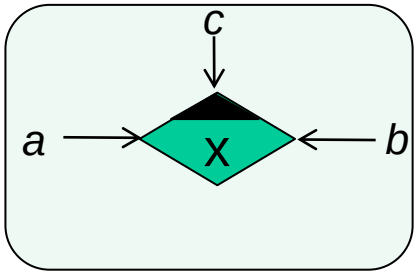

Verzögerung




Wirkungsfaktoren




Parameter



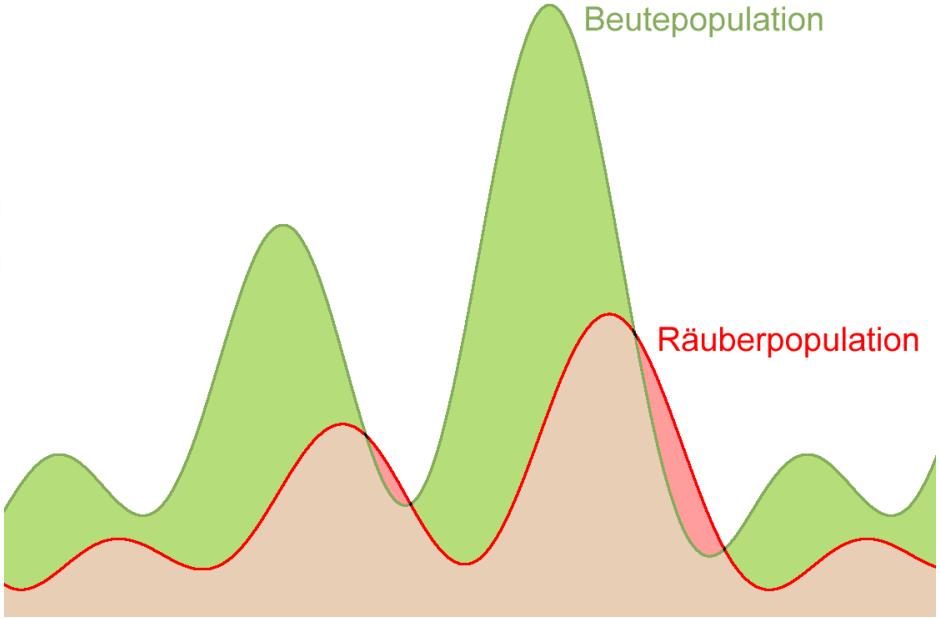

Schalter

*if c < 0 then x=a*
*if c ≥ 0 then x=b*

# 2. Analyse zeitkontinuierlicher Systeme

1. Wirkungsstrukturen
2. Beispiel: Einfaches Räuber-Beute-System
3. Gleichgewichtspunkte und ihre Stabilität
4. Systematische Betrachtung von Systemstrukturen
5. Stabilitätsanalyse linearer und nichtlinearer Systeme
6. Analytische Stabilitätsanalyse (Prinzip)
7. Besonderheiten nichtlinearer Systeme

# Prinzip von Räuber-Beute-Beziehungen



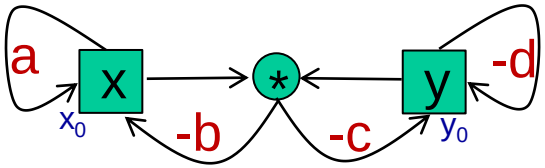

- je mehr Beutetiere vorhanden sind, desto mehr Räuber finden Nahrung

  die Population der Räuber nimmt daher zu
  – zeitlich verschoben zur Population der Beutetiere –

- durch die Vernichtung der Beutetiere sinkt
  auf Grund der fehlenden Nahrung die Anzahl der Räuber
  – ebenfalls mit einem gewissen zeitlichen Verzug –

- zwischen Räuber und Beutetier entwickelt sich ein biologisches Gleichgewicht,
  das die Populationsdichten der betreffenden Arten relativ stabil hält

# Räuber-Beute-System ohne Kapazitätsgrenze




- Systemgleichungen

$$dx/dt = a*x - b*x*y$$
$$dy/dt = c*x*y - d*y$$

a - Wachstumsrate der Beute = 1.0
b - Beuteverlustrate = 1.0
c - Beutegewinnrate = 1.0
d - Atmungsrate des Räubers = 1.0
$x_0$= 0.1
$y_0$= 0.1

- startet man für x und y mit kleinen Werten, wächst zunächst nur die Beutepopulation (exponentiell), die Räuberpopulation entwickelt sich etwas später (aber auch rasant)
- drastischer Einbruch der Beutepopulation und verzögerter drastischer Einbruch der Räuberpopulation
- völliges Verschwinden der Beutepopulation ist unmöglich

→**ungedämpfte Schwingung**
(Schwingung um einen Gleichgewichtszustand, der nicht angenommen wird)

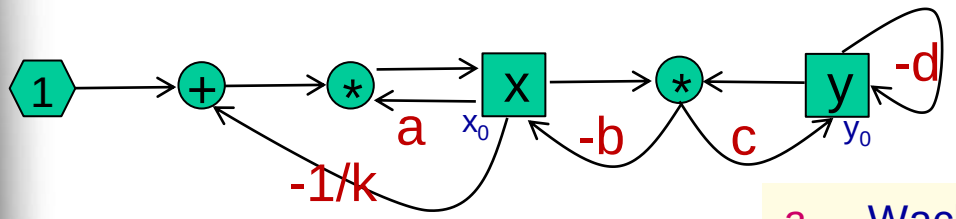
# Beutepopulation mit Kapazitätsgrenze

Räuber-Beute-Beziehung zwischen Bisamratten und Minks (amerikan. Nerz)

- Mink ist wichtigster Räuber der Bisamratte

- Populationsgröße der Bisamratte abhängig von
  - Räuberpopulation
  - Besatzdichte des Territoriums

# Räuber-Beute-System mit Kapazitätsgrenze




*a* - Wachstumsrate der Beute = 1.0
*b* - Beuteverlustrate = 1.0
*c* - Beutegewinnrate = 1.0
*d* - Atmungsrate des Räubers = 1.0
*k* - Tragfähigkeit für Beutepopulation = 1.0
$x_0$= 0.1
$y_0$= 0.1

$$x'(t) = ax(1-\frac{x}{k}) - bxy$$
$$y'(t) = cxy - dy$$

**Zustandsgrößen**:
*x*- Beutepopulation (logistisches Wachstum mit Wachstumsbeschränkung: Kapazitätsgrenze *k*)
*y*- Räuberpopulation

- *x* würde ohne Räuber allmählich anwachsen bis Tragfähigkeit des Nahrungsgebietes erreicht ist und bleibt dann konstant
- *y* würde ohne Beute schließlich verhungern
- leben beide zusammen, beeinflussen sich die Populationen

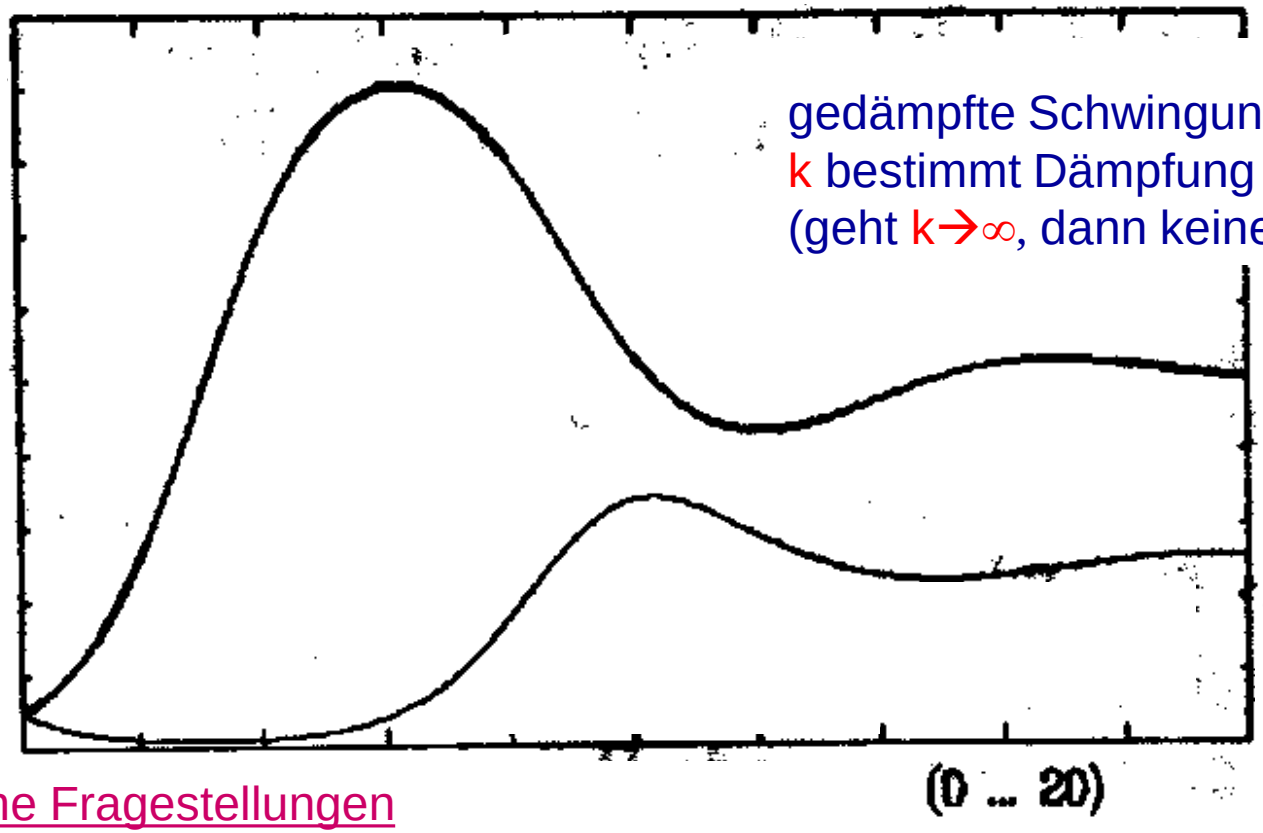
**→ gedämpfte Schwingung ➔ Gleichgewichtszustand**

# Räuber-Beute-System mit Kapazitätsgrenze

Systementwicklung für Anfangswerte

$x$= 0.1

$y$= 0.1




gedämpfte Schwingung
k bestimmt Dämpfung
(geht k→∞, dann keine Dämpfung)

*x* (Beute)

*y* (Räuber)

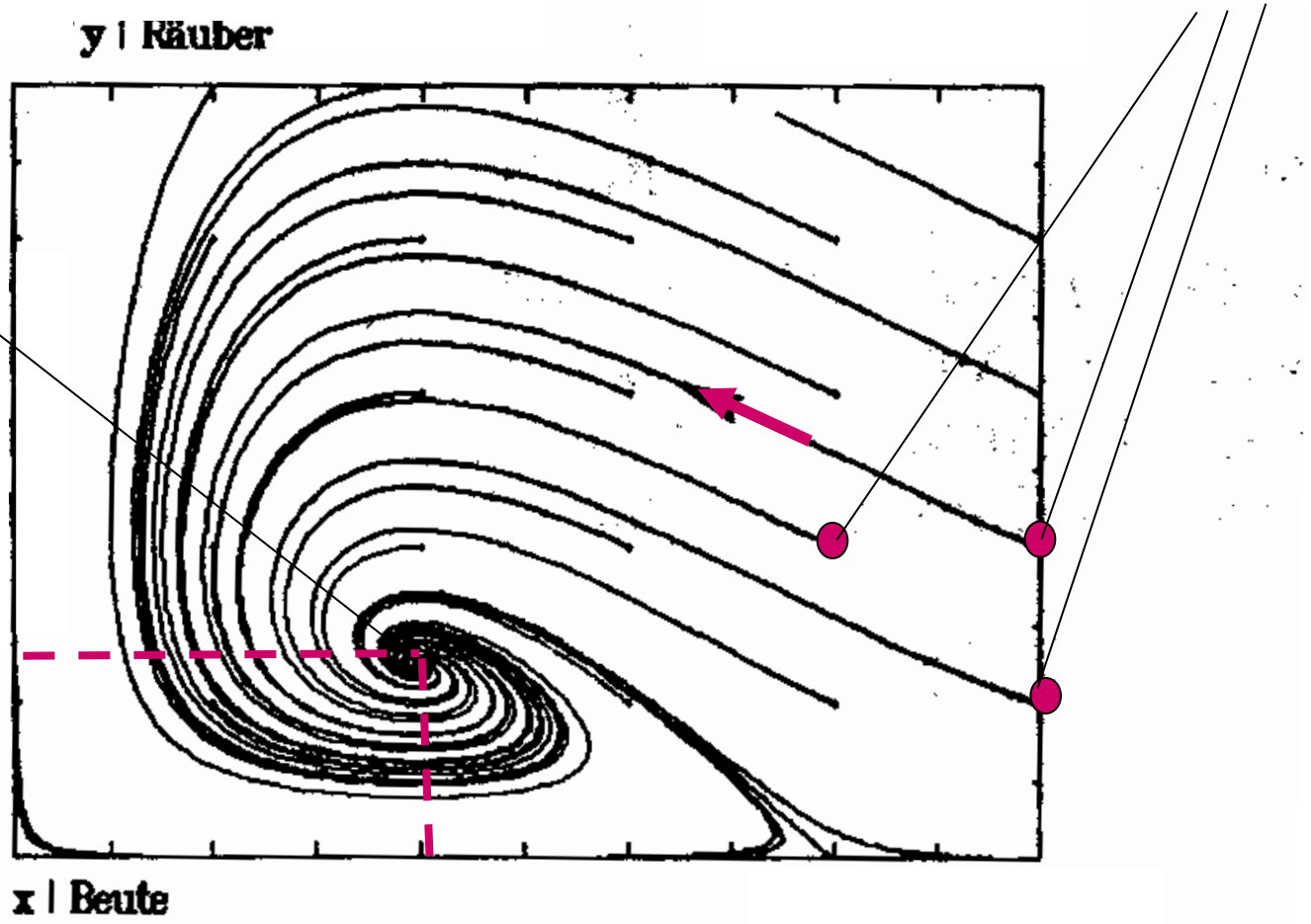
typische Fragestellungen

- Gibt es Gleichgewichtspunkte ungleich (0.0) ?
- Ändern sich Gleichgewichtspunkte bei Änderung der Startbedingungen?

# Räuber-Beute-System mit Kapazitätsgrenze

Beantwortung durch vollständige Zustandsraumanalyse:
Trajektoriendarstellungen für unterschiedliche Anfangswerte

unabhängig vom Startwert $(x_0,y_0)$ gibt es genau einen nichttrivialen Gleichgewichtspunkt (für **dieses** System)




Trajektorien im 2-dimensionalen Zustandsraum

# 2. Analyse zeitkontinuierlicher Systeme

1. Wirkungsstrukturen
2. Beispiel: Einfaches Räuber-Beute-System
3. Gleichgewichtspunkte und ihre Stabilität
4. Systematische Betrachtung von Systemstrukturen
5. Stabilitätsanalyse linearer und nichtlinearer Systeme
6. Analytische Stabilitätsanalyse (Prinzip)
7. Besonderheiten nichtlinearer Systeme

# Gleichgewichtspunkte

- (oder stationärer Punkte, Fixpunkte) im Zustandsraum
  sind natürliche Ruhepunkte eines Systems
  ihre Kenntnis ist für die Beurteilung des Systemverhaltens wichtig

- an diesen Stellen verschwinden die Ableitungen der Zustandsgrößen nach der Zeit $\frac{d\vec{y}}{dt} \rightarrow 0$

- nichtlineare Systeme können mehrere Gleichgewichtspunkte besitzen, welcher angenommen wird, hängt häufig von Anfangswerten der Zustandsgrößen ab

- besonders interessant ist die jeweilige Bewegung des Systems (Zustandsbahnen) in der Umgebung seiner Gleichgewichtspunkte

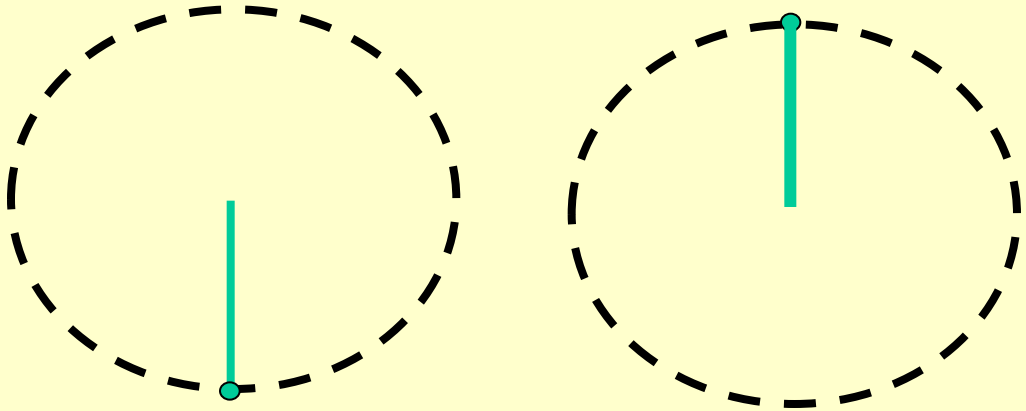
Stabilität von Gleichgewichtspunkten

# *Stabile und instabile Gleichgewichtspunkte*

- Gleichgewichtspunkt (GP): System befindet sich in Ruhe,
  d.h.: Zustandsgrößen ändern sich nicht (mehr)

- anschaulich Pendel-Beispiel
  (fester, drehbarer Stab, Punktmasse, kein Luftwiderstand)

– zwei Gleichgewichtspunkte (für Zustandsgröße *Auslenkung*)



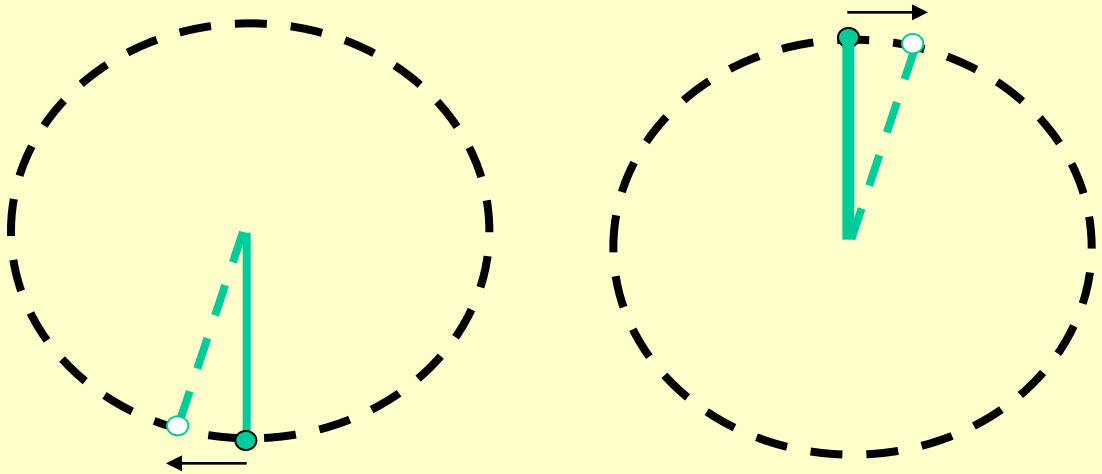
**Entscheidungskriterium für Stabilität/ Instabilität von Gleichgewichtspunkten im Systemverhalten: ???**

# Stabile und instabile Gleichgewichtspunkte

**Entscheidungskriterium :** Welches Verhalten ergibt sich bei kleinster Bewegung aus dem Gleichgewichtspunkt ?

**stabil**: Rückkehr in den Gleichgewichtspunkt
**instabil**: Verlassen des Gleichgewichtspunktes **(meist Übergang in anderen Gleichgewichtspunkt)**

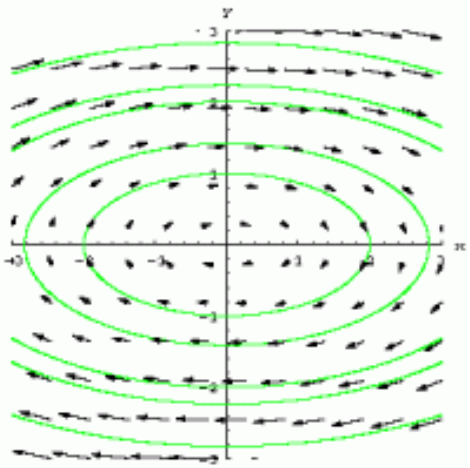
## Pendel-Beispiel

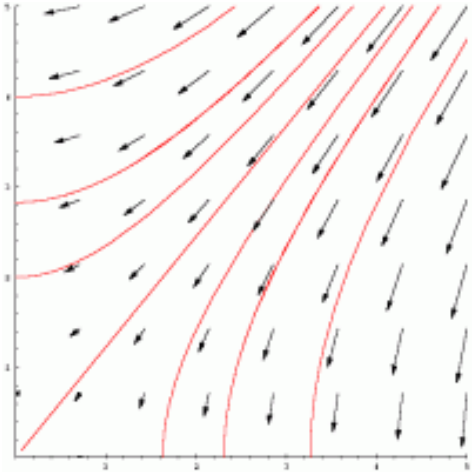


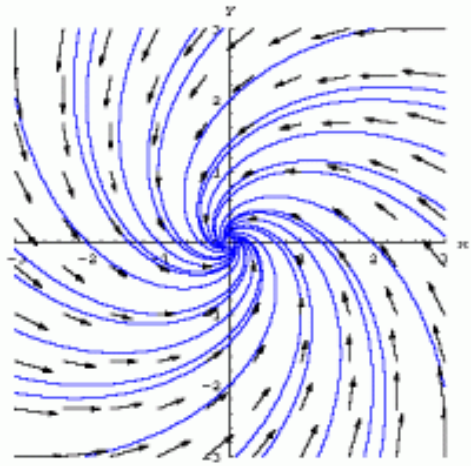
unterer GP ist stabil, der obere dagegen instabil

Es gibt mathematische Verfahren zur Bestimmung der Qualität von Gleichgewichtspunkten

# Gleichgewichtspunkte







wichtiges Problem bei der Systemmodellierung mit Differentialgleichungen ist die Stabilität von Gleichgewichtspunkten (Ruhezustände).

# *Wichtige Eigenschaften von Trajektorien*

(1) Verlauf einer Trajektorie (Zustandsbahn) ist durch **Anfangswert** eindeutig bestimmt.
Verschiedene Trajektorien (gestartet aus unterschiedlichen Anfangswerten) schneiden sich (nichtdeterministisches Verhalten ausgeschlossen) nicht

(2) Eine Trajektorie kann u.U. nur aus einem Punkt (dem Anfangswert) bestehen
Hier sind die Ableitungen der Zustandsgrößen nach der Zeit Null

(3) alle Punkte im Zustandsraum, wo sich die Trajektorie nicht mehr bewegt, heißen **Gleichgewichtspunkte** oder stationäre Punkte

es gibt auch Gleichgewichtspunkte, die durch das System gar nicht angenommen werden können

(4) Systeme können **periodisches Verhalten** zeigen,
Trajektorienmuster können Kreise, Ellipsen oder andere geschlossene Kurvendarstellungen sein

# *Wichtige Eigenschaften von Trajektorien*

- ein System bewegt sich
  - ausgehend von einem oder
    mehreren Anfangswert(en) auf gesonderten Bahnen

    einem **Gleichgewichtspunkt** zu,

    **ohne diesen zu erreichen**

# *Chaotisches Systemverhalten*

- sogar bei determinierten Systemen möglich

  Tiefofen-Beispiel
  war stochastisch (Ankunftszeit, Anfangstemperatur) und dennoch **nicht** chaotisch!

- seien Umwelteinflüsse und Anfangszustand determiniert,
  ist chaotisches Systemverhalten dadurch gegeben, dass sich
  - im Fall **kleinster** Änderungen in den Umwelteinflüssen oder/und Anfangswerten
  - die Systementwicklungen **dramatisch** ändern (Zustandsbahnen streben exponentiell beschleunigt auseinander)

  ➔ Unbestimmbarkeit des künftigen Verhaltens