
Einführung in Computational Engineering

Zusammenfassung der Vorlesung im WS 2010/2011
Dominik Schreiber <ow91fibo@rbg.informatik.tu-darmstadt.de>



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1	Einführung	3
1.1	Begriffsbildung	3
1.2	Schritte einer Simulationsstudie	3
1.2.1	Problemspezifikation	3
1.2.2	Modellierung	4
1.2.3	Implementierung	4
1.2.4	Validierung	4
2	Diskrete Modellierung und Simulation	5
2.1	Einleitung und Grundbegriffe	5
2.2	Funktionsweise einer Discrete-Event-Simulation (DEVS)	5
2.2.1	Ereignisalgorithmus	6
2.3	Petrinetze	6
2.3.1	Beschreibung von Petrinetzen	7
2.3.2	Mathematische Darstellung zeitdiskreter Petrinetze	7
2.3.3	Eigenschaften von Petrinetzen	8
2.4	Endliche Zustandsautomaten, XABSL	8
3	Zeitkontinuierliche Modellierung und Simulation	9
3.1	Einleitung	9
3.2	Beschreibung zeitkontinuierlicher Systeme	9
3.3	Modellanalyse	10
3.3.1	Lösbarkeit	10
3.3.2	Gleichgewichtslösungen	10
3.3.3	Jacobi-Matrix	10
3.3.4	Linearisierung um die Ruhelage	10
3.3.5	Lösung von $(\Delta x) = A\Delta x$	11
3.3.6	Stabilität	11
3.3.7	Zeitcharakteristik	11
3.3.8	Steife Differentialgleichungen	11
3.3.9	Unstetige rechte Seite	11
3.3.10	Linearisierung um eine Referenztrajektorie	11
3.3.11	PD-Regelung linearer Systeme	12
3.4	Grundlagen der numerischen Simulation	12
3.4.1	Zahldarstellung	12
3.4.2	Rundungsfehler	12
3.4.3	Fortpflanzung von Rundungsfehlern	13
3.4.4	Kondition	13
3.4.5	Numerische Stabilität	13
3.4.6	Rückwärtsanalyse	13
3.5	Berechnung nichtlinearer Gleichgewichtslösungen	13
3.5.1	Differentialgleichungslöser	13
3.5.2	Fixpunktiteration	14
3.5.3	Newton-Verfahren	14
3.6	Numerische Lösung der nichtlinearen Zustandsdifferentialgleichungen	15
3.6.1	Explizites Euler-Verfahren	15
3.6.2	Implizites Euler-Verfahren	15
3.6.3	Heun-Verfahren	16
3.6.4	Runge-Kutta-Verfahren	16
3.6.5	Schrittweitensteuerung	17
3.7	Integration von Zustands-Differentialgleichungen mit Unstetigkeiten	17
3.7.1	Schaltfunktionen	17

3.8	Zeitkontinuierliche Simulationswerkzeuge	18
3.8.1	Level 0: Matlab	18
3.8.2	Level 1: Differentialgleichungslöser in Matlab	18
3.8.3	Blockdiagrammsymbole	19
4	Interpretation und Validierung	20
5	Modulare Modellbildung und Simulation komplexer Systeme	21
5.1	Modulare Modellbildung	21
5.1.1	Elektrische Schaltungen	21
5.1.2	Mechanik	22

1 Einführung

Überblick: Definition von wichtigen Begriffen, Ausführung der *5 Schritte einer Simulationsstudie*.

1.1 Begriffsbildung

Simulation Nachahmung eines realen Systems am Computer. Mögliche Zwecke einer Simulation:

- bekanntes Szenario verstehen oder nachvollziehen,
- bekanntes Szenario optimieren,
- unbekanntes Szenario vorhersagen.

Simulationsstudie in 5 Schritten:

Problemspezifikation Grund/Zweck der Simulation?

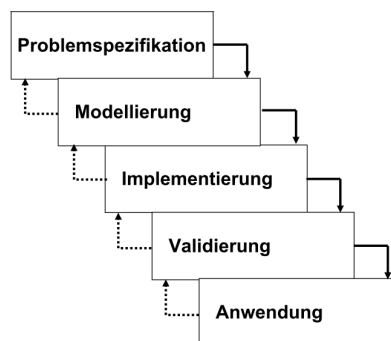
Modellierung Beschreibung der Realität durch Mathematik, dadurch Abstraktion. Modell muss sowohl ausreichend genau als auch handhabbar (daher von unwichtigen Details befreit) sein. Größen fallen *qualitativ* und *quantitativ* auf.

Implementierung des Modells.

Validierung erfüllt das implementierte Modell seinen Zweck?

Anwendung

1.2 Schritte einer Simulationsstudie



1.2.1 Problemspezifikation

Aufgabenstellungen bei der Simulation System hat Ein- und Ausgänge, Daten darüber sind nur teilweise bekannt:

Simulationsproblem Eingänge, Systemmodell bekannt; Ausgang unbekannt.

Strukturidentifikationsproblem Eingänge, Ausgang bekannt; Systemmodell unbekannt.

Steuerungsproblem Systemmodell, Ausgang bekannt; Eingänge unbekannt.

Parameterschätzproblem Ausgang, Eingang, Systemmodell bekannt; Systemparameter unbekannt.

1.2.2 Modellierung

System Menge miteinander verbundener (komplexer) Komponenten mit Attributen und Zustandsvariablen. Beschreibung durch

algebraische (Un-)Gleichungen z.B. $E = mc^2$

gewöhnliche Differentialgleichungen eine unabhängige Variable (meist t), z.B. $\dot{y}(t) = y(t)$

partielle Differentialgleichungen mehrere unabhängige Variablen, z.B. $u_{xx} + u_{yy} = f, u = \theta$ für $(x, y) \in \Omega$

Automaten, Zustandsübergangsdiagramme z.B. für Warteschlangen, Texterkennung

Graphen z.B. für Rundreisen (z.B. „Handlungsreisender“), Reihenfolgen, Rechensysteme, Abläufe

Wahrscheinlichkeitsverteilungen z.B. für Zustimmungen, Heuristiken

Fuzzy Logic z.B. für Regelung von Haushaltsgeräten

neuronale Netze

algebraische Strukturen z.B. Gruppen (Quantenmechanik), endliche Körper (Kryptographie)

Modell gibt Realität mit gewissen Einschränkungen wieder.

Zustandsvariablen nicht-redundante, zeitabhängige Größen, die die aktuelle Konfiguration und den zukünftigen Verlauf eines Systems genau festlegen (deterministisch).

Zustandsübergänge Art hängt von Zeit- und Wertübergängen ab:

zeitkontinuierlich&wertkontinuierlich, zeitdiskret&wertkontinuierlich, zeitdiskret&wertdiskret, ereignisdiskret&wertdiskret, stochastisch

1.2.3 Implementierung

Beinhaltet die Auswahl/Entwicklung eines *Berechnungsverfahrens*, die Programmierung von Modell und Berechnungsverfahren, sowie die Visualisierung der Ergebnisse.

Lösungsansätze für mathematische Modelle

analytisch Existenz- / Eindeutigkeitsnachweis sowie Konstruktion erfolgen formal/analytisch. Fast nie erreichbarer Optimalfall.

heuristisch „trial&error“ nach bestimmter Heuristik (z.B. Greedy)

direkt-numerisch numerischer Algorithmus liefert exakte Lösung

approximativ-numerisch iteratives Näherungsverfahren für angenäherte Beziehungen (= diskretisierte Gleichungen). Häufigster Fall.

1.2.4 Validierung

Ist systematische Plausibilitätsprüfung des Simulationsmodells (erfüllt es die Anforderungen?). Ausreichende Glaubwürdigkeit des Simulationsmodells bzgl. der Problemspezifikation soll nachgewiesen werden. Dies macht falsche Schlussfolgerungen aus der Simulation unwahrscheinlicher. Validierung beruht auf sorgfältig ausgewählten Tests. Möglichkeiten:

Vergleich mit Experimenten z.B. Laborexperimente (Prototyping), 1:1 Experimente (Crashtest).

A-posteriori Beobachtungen z.B. Realitäts-Test (Wetter nach Vorhersage), Zufriedenheits-Test.

Plausibilitäts-Test Simulationsergebnisse auf Konsistenz mit bestehenden Theorien getestet.

Modellvergleich Vergleich von Simulationen unterschiedlicher Modelle.

2 Diskrete Modellierung und Simulation

Überblick: Zeit- oder ereignisdiskrete Modelle werden mittels Petrinetzen oder endlichen Zustandsautomaten modelliert.

2.1 Einleitung und Grundbegriffe

zeitdiskrete Modelle nur zu ausgewählten Zeiten t Beobachtung des Systemausgangs. Häufig äquidistant diskreditierte Variante des zeitkontinuierlichen Modells.

ereignisdiskrete Modelle Zeitachse nicht äquidistant. Änderungen bei Ereignissen.

ereignisdiskrete Simulation Terminologie:

System Ansammlung von Objekten, interagieren zeitabhängig nach spezifizierten Regeln.

Modell abstrakte Darstellung der Objekte und Wechselbeziehungen in einem *System*.

Entität (= entity) einzelnes Objekt, in *Modell* explizit dargestellt.

Attribut (= attribute) Zustand einer *Entität* beschreibende Variable.

Ereignis (= event) Aktualisierung des Modellzustands.

Ereignisliste Liste von $\langle \text{Zeitpunkt}, \text{Ereignistyp} \rangle$, nach Zeitpunkt aufsteigend geordnet.

Aktivität (= activity) zeitlich erstreckter, Zustand einer *Entität* verändernder Vorgang mit initiiertem und abschließendem *Ereignis*.

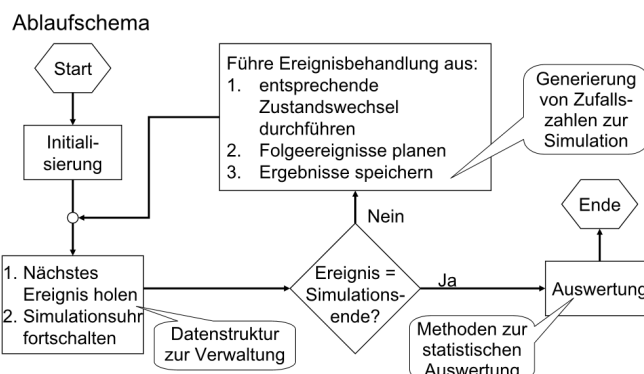
Simulationsuhr (= simulation clock) aktuelle Simulationszeit, Variable.

Zeitführungsroutine (= timing) Prozedur; Auswahl des nächsten *Ereignisses* aus *Ereignisliste*, *Simulationsuhr* auf nächsten Ereigniszeitpunkt vorstellen.

Ergebnisroutine Prozedur; Berechnung statistischer Schätzwerte der Ergebnisvariablen, Ausgabe des Ergebnisprotokolls.

Steuerprogramm Prozedur; ruft wiederholt *Zeitführungs-* und *Ergebnisroutine* auf bis Simulation beendet.

2.2 Funktionsweise einer Discrete-Event-Simulation (DEVS)



2.2.1 Ereignisalgorithmus

```
int N; // Länge der Warteschlange
{idle, busy} S; // Serverzustand
int exp(1); // Ankunftszeiten
int norm(m, s); // Serverbelegungszeiten; m Mittelwert, s Standardabweichung
List L; // nach Zeitpunkten aufsteigend sortierte Ereignisliste
int t, tMax; // Start-, Endzeitpunkt

DEVS() {
    L = { (t1, E1) }; t = 0;
    initializeSystem();

    while (t < tMax) {
        t := t1;
        E := E1;
        L := L \ (t1, E1);
        eventRoutine(E);
        sort(L);
    }
}

initializeSystem() {
    N = 0;
    S = idle;
}

eventRoutine(E) {
    switch E:
        A: arrivalRoutine();
        D: departureRoutine();
}

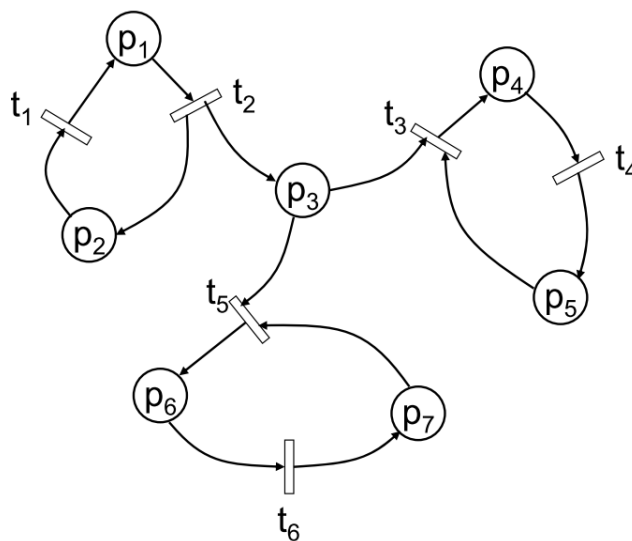
arrivalRoutine() {
    L += (t+exp(1), A);
    if (S == busy) N++;
    else {
        S = busy;
        L += (t+norm(m, s), D);
    }
}

departureRoutine() {
    if (N == 0) S = idle;
    else {
        N--;
        L += (t+norm(m, s), D);
    }
}
```

2.3 Petrinetze

Überblick: Ein Petrinetz ist ein gerichteter, bipartiter Graph mit Plätzen und Transitionen (2 Arten von Knoten), wobei Kanten nur zwischen Platz und Transition existieren.

2.3.1 Beschreibung von Petrinetzen



Petrinetz gerichteter, bipartiter Graph $PN = (P, T, A, K, M')$ mit 2 Arten von Knoten (Plätze, Transitionen) und Kanten nur zwischen *unterschiedlichen* Knoten (Platz und Transition).

zeitdiskret (kausal) beschreibt logisch, *was* in welcher Sequenz passiert.

zeitkontinuierlich enthält Vorhersage, *wann* ein Ereignis auftritt.

Plätze (= places) (Kreis) mögliche Zustände. $P = \{p_1, p_2, \dots, p_n\}$.

Transitionen (Rechteck) mögliche Ereignisse. $T = \{t_1, t_2, \dots, t_m\}$.

gerichtete Kanten (= directed edges) verbinden Plätze und Transitionen. $A = \{a_1, a_2, \dots, a_k\}$.

Markierungen (= tokens) (Punkt in Kreis) definieren aktuellen Zustand des Petrinetzes. Zustand des Petrinetzes: $M = \{m_1, m_2, \dots, m_n\}$ mit m_i = Anzahl der Marken in p_i . Anfangsmarkierung $M' = \{m'_1, m'_2, \dots, m'_n\}$. Maximale Kapazität der Plätze $K = \{k_1, k_2, \dots, k_n\}$.

dynamische Eigenschaften resultieren aus Ausführung des Petrinetzes

- Markierungen werden erzeugt/gelöscht/verschoben durch Schalten (= Feuern) von Transitionen.
- Transitionen können schalten, wenn alle Eingangsplätze mind. eine Markierung enthalten.
- Feuern Transitionen, wird jedem ihrer Eingangsplätze eine Markierung entnommen und jedem ihrer Ausgangsplätze eine hinzugefügt.

2.3.2 Mathematische Darstellung zeitdiskreter Petrinetze

pre-weights W^- Verbindungen $p_i \rightarrow t_j$ von Platz zu Transition.

post-weights W^+ Verbindungen $t_j \rightarrow p_i$ von Transition zu Platz.

Markierungsvektor m Anzahl von Markierungen m_i in allen Plätzen p_1, \dots, p_n zu diskreten Zeitpunkten r : $m(r) = [m_1(r), m_2(r), \dots, m_n(r)]^T$.

Kapazität k Kapazitäten der Plätze p_i : $k = [k_1, k_2, \dots, k_n]^T$. Es gilt $\forall i \leq n : m_i(r) \leq k_i$.

Transitionen t $t = [t_1, t_2, \dots, t_m]^T$

Aktivierung einer Transition t_i zum Ändern des Markierungsvektors $m(r-1) \rightarrow m(r)$. Bedingungen:

- $\forall p_i \in \cdot t_i : m_i(r-1) \geq w_{ij}^- \neq 0$ - hinreichend viele Markierungen in Eingangsplätzen.
- $\forall p_i \in t_i \cdot : m_i(r-1) \geq k_i + w_{ij}^- - w_{ij}^+$ - ausreichend Kapazität in Ausgangsplätzen.

Nach Feuern einer Transition t_i gilt $m_i(r) = m_i(r-1) - w_{ij}^- + w_{ij}^+ \leq k_i$.

Aktivierungsfunktion u_j für Transition t_j : $(\forall t_j \text{ aktiviert})?1 : 0$.

Schaltvektor u $u(r) = [u_1(r), u_2(r), \dots, u_m(r)]^T$

Nach Feuern aller aktiver Transitionen gilt $m_i(r) = m_i(r-1) + \sum_{j=1}^n u_j(r) \cdot (w_{ij}^+ - w_{ij}^-)$.

Neuer Markierungsvektor $m(r) = m(r-1) + u(r) \cdot (W^+ - W^-)$.

2.3.3 Eigenschaften von Petrinetzen

Erreichbarkeit (= reachability) Ein Zustand m heißt *erreichbar* von einem Anfangszustand m' , falls eine Schaltsequenz $m' \rightarrow \dots \rightarrow m$ existiert.

Beschränktheit (= boundedness) Ein Petrinetz heißt k_i -*beschränkt*, falls an keinem Platz p_i je mehr als k_i Markierungen vorhanden sind. Ist $k_i = 1$, nennt man das Petrinetz *sicher* (= safe).

Verklemmung (= deadlock) ist ein Zustand eines Petrinetzes, in dem keine Transition mehr schalten kann.

Lebendigkeit (= liveness) ein Petrinetz heißt *lebendig*, falls es keine *tote* Transition enthält. Eine Transition heißt *tot* (= dead, non-live), falls sie durch keine Folgemarkierung mehr aktivierbar ist.

2.4 Endliche Zustandsautomaten, XABSL

Das Entscheidungsverhalten eines autonomen Agenten wird mit hierarchischen Zustandsautomaten modelliert, dessen Verhalten über eine Menge von endlichen Zustandsautomaten und elementaren Verhaltensroutinen spezifiziert wird, die hierarchisch zu einem gerichteten, azyklischen Graphen aufgebaut werden.

3 Zeitkontinuierliche Modellierung und Simulation

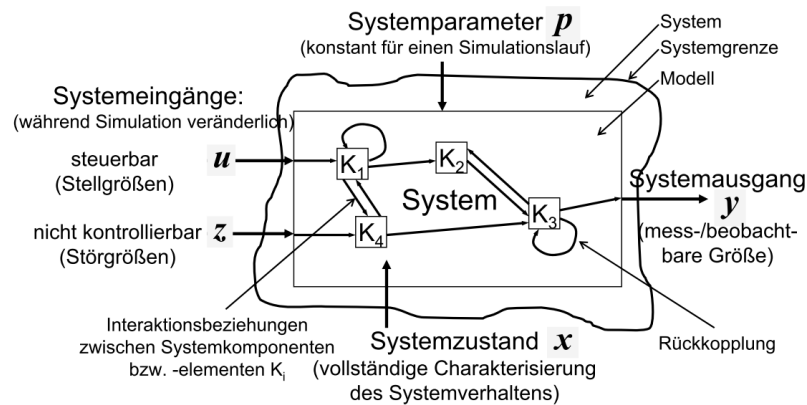
Überblick: Zeitkontinuierliche Systeme werden durch Differentialgleichungen ihrer Zustandsvariablen beschrieben, die mit mathematischen und numerischen Methoden betrachtet werden können.

3.1 Einleitung

örtlich konzentrierter Systemzustand ein System mit örtlich konzentrierten Systemzuständen wird durch *gewöhnliche* Differentialgleichungen (= Gleichung einer Funktion einer unabhängigen Veränderlichen) beschrieben.

örtlich verteilter Systemzustand ein System mit örtlich verteilten Systemzuständen wird durch *partielle* Differentialgleichungen (= Gleichung einer Funktion *mehr als* einer unabhängiger Veränderlicher, wenn mindestens eine ihrer Ableitung wieder in der Gleichung erscheint) beschrieben.

3.2 Beschreibung zeitkontinuierlicher Systeme



System gewöhnlicher Differentialgleichungen erster Ordnung:

$$\dot{x}(t) = \frac{dx(t)}{dt} = \begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \vdots \\ \dot{x}_n(t) \end{pmatrix} = \begin{pmatrix} f_1(x(t), u(t), t) \\ f_2(x(t), u(t), t) \\ \vdots \\ f_n(x(t), u(t), t) \end{pmatrix} = f(x(t), u(t), t)$$

Jedes System gewöhnlicher Differentialgleichungen *höherer* Ordnung kann auf ein System *erster* Ordnung transformiert werden. Beispiel:

$$\ddot{x} + \frac{c}{m}x = 0 \implies x_1 := x, x_2 := \dot{x}, \dot{x} := \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \implies \dot{x} = f(x) = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} := \begin{pmatrix} x_2 \\ -\frac{c}{m}x_1 \end{pmatrix}$$

Ein Differentialgleichungssystem heißt *autonom*, wenn f nicht explizit von t abhängt (t kommt nicht explizit vor). Jedes nicht-autonome Differentialgleichungssystem kann in ein autonomes Differentialgleichungssystem transformiert werden, indem eine „Uhrzeitvariable“ $x_{n+1} = t \Rightarrow \dot{x}_{n+1} = 1, x_{n+1}(0) = 0$ eingeführt wird. Beispiel:

$$\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} t^2 \cdot x_2 \\ \sqrt{t} \cdot x_1 \end{pmatrix}, x(0) = x_0 \in \mathbb{R}^2 \implies \dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{pmatrix} = \begin{pmatrix} x_3^2 \cdot x_2 \\ \sqrt{x_3} \cdot x_1 \\ 1 \end{pmatrix}, x(0) = \begin{pmatrix} x_1(0) \\ x_2(0) \\ 0 \end{pmatrix}$$

Damit wird ein gewöhnliches Differentialgleichungssystem o.B.d.A. als *autonom* und *erster Ordnung* angenommen. Wenn nicht anders angegeben hat eine solche Differentialgleichung im Folgenden die Form $\dot{x}(t) = f(x, u), x(0) = x_0 \in \mathbb{R}^n$ und wird auch als DGL bezeichnet.

3.3 Modellanalyse

3.3.1 Lösbarkeit

Richtungsfeld einer gewöhnlichen Differentialgleichung 1. Ordnung: für jeden (zulässigen) Punkt $(x, t) \in \mathbb{R}^2$ wird die Steigung $\dot{x}(t)$ in ein zweidimensionales Koordinatensystem (Horizontale: t , Vertikale: x) eingetragen.

Lösungstrajektorie $x(t)$ aus Richtungsfeld: Anfangswert festlegen ($x(0) = x_0$), dann „dem Richtungsfeld folgen“.

allgemeine Lösung einer Differentialgleichung $\dot{x}(t) = f(x(t))$ mit $x \in \mathbb{R}^n$ hängt von n Integrationskonstanten (Anfangsbedingungen) ab.

eindeutige Lösung ein Differentialgleichungssystem $\dot{x}(t) = f(x(t)), x(0) = x_0 \in \mathbb{R}^n$ hat eine eindeutige Lösung, falls die *Lipschitz-Bedingung* erfüllt ist:

$$\exists L \in \mathbb{R}^+ \forall x_1, x_2 \in \mathbb{R}^n : \|f(x_1) - f(x_2)\| \leq L \cdot \|x_1 - x_2\|$$

3.3.2 Gleichgewichtslösungen

Ruhezustand (= stationärer Zustand, Arbeitstrajektorie) einer DGL: $0 = f(x_s, u_s)$. (Allgemein $\lim_{t \rightarrow \infty} x(t) = x_s$).

Gleichgewichtslösung Lösung x_s des Systems von n Gleichungen f_i zur Bestimmung von n Unbekannten $x_{s,i}$ zu einem stationären Zustand. Man unterscheidet:

lineare Systemdynamik ($= \dot{x} = Ax + Bu$). Dann muss $Ax_s = -Bu_s$ gelten. Eine solche Lösung existiert, wenn $Rg(A) = n$ oder $\det(A) \neq 0$.

nichtlineare Systemdynamik ($= \dot{x} = f(x, u)$). Dann muss $0 = f(x_s, u_s)$ gelten. Dazu kann es *keine, genau eine, mehrere* oder auch *unendlich viele* Lösungen geben.

3.3.3 Jacobi-Matrix

Die Jacobi-Matrix der rechten Seite $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ einer Differentialgleichung hinsichtlich x ist folgendermaßen definiert:

$$\frac{df}{dx} = \left(\frac{df_i}{dx_j} \right)_{i,j=1,\dots,n} = \begin{pmatrix} \frac{df_1}{dx_1} & \dots & \frac{df_1}{dx_n} \\ \vdots & \ddots & \vdots \\ \frac{df_n}{dx_1} & \dots & \frac{df_n}{dx_n} \end{pmatrix}$$

Berechnungsmöglichkeiten

analytisch (exakt) „von Hand“, mittels Formelmanipulationsprogrammen.

numerisch (approximativ) (= Vorwärtsdifferenzenquotient) Berechnung der partiellen Ableitungen $\frac{df_i}{dx_j}$ durch Annäherung

der Tangente: $\frac{df_i}{dx_j} = \lim_{\delta_j \rightarrow 0} \frac{1}{\delta_j} (f(x + e_j \delta_j) - f(x))$. Wahl der Schrittweite $\delta_j = \epsilon(1 + |x_j|)$ relativ zu ϵ_{mach} ,

der relativen Maschinengenauigkeit: $\epsilon = \sqrt{\epsilon_{mach}}$. Genauigkeit: $\left| \frac{df_i(x)}{dx_j} - \frac{1}{\delta_j} (f_i(x + e_j \delta_j) - f_i(x)) \right| \leq |\delta_j| \cdot \left| \frac{d^2 f_i(\hat{x})}{dx_j^2} \right|$

mit $\hat{x} \in]x, x + e_j \delta_j[$. Als Faustregel erhält man maximal die *Halbte* der gültigen Dezimalstellen von f .

symbolisches Differenzieren algorithmische Anwendung von Ketten-, Produkt- und anderen Differentiationsregeln. Daher hoher Berechnungsaufwand.

automatisches Differenzieren Programm von $f(x)$ wird in Programm zur Auswertung der Jacobi-Matrix transformiert. Dabei Vorwärts- und Rückwärtsmodus (dabei effiziente Gradientenberechnung) möglich.

3.3.4 Linearisierung um die Ruhelage

Man betrachtet die Abweichung $\Delta x_i(t) := x_i(t) - x_{s,i}$ von einer Ruhelage x_s und deren Auswirkungen. Es wird eine DGL für $\Delta x(t)$ gesucht. Mit Taylor-Entwicklung erhält man die lineare DGL (geht nur, wenn f in x_s stetig differenzierbar):

$$(\dot{\Delta x}_i) = \dot{x}_i = f_i(x, u) = f_i(x_s + \Delta x, u_s + \Delta u) = \frac{df}{dx}|_{x_s, u_s} \cdot \Delta x + \frac{df}{du}|_{x_s, u_s} \cdot \Delta u$$

3.3.5 Lösung von $(\dot{\Delta x}) = A\Delta x$

skalare Differentialgleichung $(\dot{\Delta x}) = a \cdot \Delta x$ wird mit $\Delta x(t) = c \cdot e^{\lambda t}$ gelöst.

Vektordifferentialgleichung $(\dot{\Delta x}) = A \cdot \Delta x$ (z.B. $A = \frac{df(x,u)}{dx}|_{x_s, u_s}$) hat die allgemeine komplexe Lösung $\Delta x(t) = \sum_{i=1}^n c_i \cdot e^{\lambda_i t}$, wobei λ_i der i -te Eigenwert von A und c_i der zugehörige Eigenvektor ist. Ist λ_i komplex, sind jeweils Real- und Imaginärteil von $c_i \cdot e^{\lambda_i t} = I_{i,Re}(t) + i \cdot I_{i,Im}(t)$ Lösungen. $I_{i,Re} := e^{Re(\lambda_i)t}(\gamma_i \cos(Im(\lambda_i)t) - \beta_i \sin(Im(\lambda_i)t))$, $I_{i,Im} := e^{Re(\lambda_i)t}(\gamma_i \sin(Im(\lambda_i)t) + \beta_i \cos(Im(\lambda_i)t))$. Existiert eine Basis aus Eigenvektoren, erhält man die allgemeine Lösung durch Linearkombination der $I_{i,Re}, I_{i,Im}$. Durch eine Anfangsbedingung werden γ_i, β_i festgelegt.

3.3.6 Stabilität

Eine Ruhelage heißt *stabil*, wenn eine durch eine kurze Störung verursachte Abweichung aus ihr mit der Zeit gegen 0 strebt. Hierbei spielen die (Realteile der) Eigenwerte von $\frac{df}{dx}|_{x_s, u_s}$ die entscheidende Rolle: sind alle negativ, ist die Ruhelage *stabil*. Ist einer 0 und alle anderen negativ, kann man keine Aussage über die Stabilität (anhand dieses Kriteriums) machen. Ist einer positiv, ist die Ruhelage *instabil*.

3.3.7 Zeitcharakteristik

Zeitcharakteristika T_i eines linearisierten Systems können über die Eigenwerte λ_i bestimmt werden:

reelles λ_i : $T_i = \frac{1}{|\lambda_i|}$

rein imaginäres λ_i : $T_i = \frac{2\pi}{|\lambda_i|}$

konjugiert komplexes λ_i : $T_i = \min\left(\frac{1}{|Re(\lambda_i)|}, \frac{2\pi}{|Im(\lambda_i)|}\right)$

Somit ergibt sich $T_{max} = \max(T_i)$ und $T_{min} = \min(T_i)$. Man kann dann eine sinnvolle Simulationsdauer abschätzen (als Faustregel):

stabiles System $t_f = 5 \times T_{max}$

instabiles System wähle t_f so dass $|x(t_f)| \geq M$ mit $M \in \mathbb{R}$.

3.3.8 Steife Differentialgleichungen

Eine stabile Differentialgleichung mit sehr unterschiedlichen Zeitcharakteristika wird *steif* genannt. Ein Maß für Steifheit ist $\frac{T_{max}}{T_{min}}$. Je größer dieser Wert, desto „steifer“ ist die DGL. Ab $\frac{T_{max}}{T_{min}} > 10^3 \dots 10^7$ spricht man von einer *steifen Differentialgleichung*. Die numerische Lösung solcher DGLn ist aufwendig und erfordert *implizite Verfahren*.

3.3.9 Unstetige rechte Seite

Ist die rechte Seite einer Differentialgleichung *unstetig* (z.B. durch unstetige Steuerung $u(t)$) kann dies erhebliche Schwierigkeiten bei der numerischen Integration verursachen (die meist auf mehrfacher stetiger Differenzierbarkeit der rechten Seite beruhen). Durch Detektion der Unstetigkeit und abschnittsweise Berechnung kann die numerische Integration verbessert werden.

3.3.10 Linearisierung um eine Referenztrajektorie

Linearisierung um eine Referenztrajektorie kann interpretiert werden als Linearisierung um eine *zeitveränderliche* Ruhelage. $x_s = x_s(t), u_s = u_s(t) : \dot{x}_s(t) = f(x_s(t), u_s(t))$. Durch Linearisieren erhält man $(\dot{\Delta x}) = A(t) \cdot \Delta x + B(t) \cdot \Delta u = \left(\frac{df_i(x_s(t), u_s(t))}{dx_j}\right)_{i,j=1,\dots,n} \cdot \Delta x + \left(\frac{df_i(x_s(t), u_s(t))}{du_j}\right)_{i,j=1,\dots,n} \cdot \Delta u$.

3.3.11 PD-Regelung linearer Systeme

Beobachtung des Bewegungsverhaltens einer linearen Systemdynamik (am Beispiel $m\ddot{x}(t) + b\dot{x}(t) + kx(t) = 0$): wie kehrt das System nach einer Auslenkung wieder zu einer stabilen Ruhelage zurück?

Bringe dazu die Differentialgleichung zunächst in *Normalform*: $\ddot{x}(t) + \frac{b}{m}\dot{x}(t) + \frac{k}{m}x(t) = 0$. Bilde dann charakteristische Gleichung $\lambda^2 + \frac{b}{m}\lambda + \frac{k}{m} = 0$ und finde Nullstellen: $\lambda_{1,2} = -\frac{b}{2m} \pm \frac{\sqrt{b^2 - 4mk}}{2m}$. Anhand dieser Nullstellen (= Pole) kann man das Bewegungsverhalten der Masse bestimmen:

$\lambda_{1,2}$ einfache, reelle Nullstellen im Beispiel $b^2 - 4mk > 0 \Leftrightarrow b^2 > 4mk \Rightarrow \lambda_{1,2} < 0$. Die allgemeine Lösung lautet $x(t) = c_1 e^{\lambda_1 t} + c_2 e^{\lambda_2 t}$. Dies bedeutet „dominierende Reibungskraft“. Es gilt $\lim_{t \rightarrow \infty} x(t) = 0$, da $\lambda_{1,2} < 0$. Dieses Verhalten nennt man *überkritisch gedämpft* (= overdamped).

$\lambda_{1,2}$ doppelte, reelle Nullstelle im Beispiel $b^2 - 4mk = 0 \Leftrightarrow b^2 = 4mk \Rightarrow \lambda_{1,2} = -\frac{b}{2m} < 0$. Dies bedeutet „ausgewogene Reibung und Federsteifigkeit“. Die allgemeine Lösung $x(t) = c_1 e^{\lambda_1 t} + c_2 e^{\lambda_2 t}$ ist wegen $\lim_{t \rightarrow \infty} x(t) = 0$ wieder abklingend. Das Verhalten nennt man *kritisch gedämpft* (= critically damped).

$\lambda_{1,2}$ einfache, komplexe Nullstelle im Beispiel $b^2 - 4mk < 0 \Leftrightarrow b^2 < 4mk \Rightarrow \lambda_{1,2} = -\frac{b}{2m} \pm i \frac{\sqrt{b^2 - 4mk}}{2m} := \lambda_r \pm i\lambda_i$. Dies bedeutet „dominierende Federkraft“. Die allgemeine Lösung ist nun $x(t) = c_1 e^{\lambda_r t} \cos(\lambda_i t) + c_2 e^{\lambda_r t} \sin(\lambda_i t)$. Da $\lambda_r < 0$ gilt weiterhin $\lim_{t \rightarrow \infty} x(t) = 0$, allerdings oszilliert die Bewegung dabei um die Ruhelage. Dieses Verhalten nennt man *unterkritisch gedämpft* (= underdamped).

Um das natürliche Bewegungsverhalten eines Systems hin zu einem *kritisch gedämpften* zu verändern, stellt man ein *Regelgesetz* auf, das die anzuwendende Antriebskraft f als Funktion der Zustandswerte bestimmt: $f = -k_p x - k_v \dot{x}$, wobei k_p, k_v reelle Konstanten sind, die den Proportionalanteil $-k_p x$ bzw. den Differentialanteil $-k_v \dot{x}$ bezeichnen. So versucht man, die Masse unabhängig von Störeinflüssen in einer konstanten Normalposition zu halten. Wendet man dieses Regelgesetz auf die Bewegungsdifferentialgleichung an, erhält man veränderte Faktoren, die das System ein (von k_p, k_v abhängiges) gedämpftes Verhalten bekommen lassen:

$$m\ddot{x}(t) + b\dot{x}(t) + kx(t) = -k_p x(t) - k_v \dot{x}(t) \Leftrightarrow m\ddot{x}(t) + (b + k_v)\dot{x}(t) + (k + k_p)x(t) = 0$$

3.4 Grundlagen der numerischen Simulation

3.4.1 Zahldarstellung

Reelle Zahlen werden auf Computern als *normalisierte Gleitpunktzahlen* dargestellt. Allgemein hat dies die Form $z = \pm(d_1 \cdot B^0 + d_2 \cdot B^{-1} + \dots + d_n \cdot B^{-n+1}) \cdot B^E$, wobei B die Basis, E der Exponent mit $E_{\min} \leq E \leq E_{\max}$ und d_i die Ziffern sind. Da z normalisiert ist, ist $d_1 \neq 0$. Die $d_1 \dots d_n$ sind die Mantisse M der Zahl. Da n, E_{\min}, E_{\max} endlich und konstant gibt es nur endlich viele darstellbare Zahlen, daher Rundungsfehler.

Eine 32-bit Gleitkommazahl (nach IEEE 754 Standard) besteht aus: 1 Bit Vorzeichen (S), 8 Bit Exponent (E) (mit Bias ($bias$)) und 23 Bit Mantisse (M) (normalisiert). Der Wert berechnet sich also durch $(-1)^S \cdot (1 + M) \cdot 2^{E-bias}$.

3.4.2 Rundungsfehler

Da es nur endlich viele Gleitpunktzahlen gibt, müssen reelle Zahlen x auf die „am nächsten liegende“ Gleitpunktzahl abgebildet (= gerundet) werden: $\forall g : |x - rd(g)| \leq |x - g|$. Dabei gibt es 4 Rundungsarten (nach IEEE 754):

- aufrunden („nach ∞ “)
- abrunden („nach $-\infty$ “)
- Stellen abschneiden („nach 0“)
- zur nächsten geraden Gleitpunktzahl

Der dabei entstehende Rundungsfehler ϵ ist $\epsilon(x) := \frac{x - rd(x)}{x} \Leftrightarrow rd(x) = x \cdot (1 - \epsilon(x))$. Sei $gl(x)$ die Gleitpunktdarstellung der Zahl $x \in \mathbb{R}$. Für die arithmetischen Gleitpunktoperationen $+, -, \cdot, /$ und $|\epsilon_i| \leq eps, \epsilon_i = \epsilon_i(x, y)$ gilt nun:

- $gl(x + y) = (x + y)(1 + \epsilon_1)$
- $gl(x - y) = (x - y)(1 + \epsilon_2)$

- $gl(x \cdot y) = (x \cdot y)(1 + \epsilon_3)$
- $gl(x/y) = (x/y)(1 + \epsilon_4)$

3.4.3 Fortpflanzung von Rundungsfehlern

Sei f ein Algorithmus, der aus einer endlichen Anzahl von Elementaroperationen zusammen gesetzt ist: $f = f^{(r)} \circ f^{(r-1)} \circ \dots \circ f^{(0)}$. Den Rundungsfehler analysiert man, indem in allen Ausdrücken $gl(x \square y) = (x \square y)(1 + \epsilon)$ ersetzt wird. So ermittelt man eine Darstellung $gl(f) = f \cdot (1 + (\dots)\epsilon_1 + (\dots)\epsilon_2 + \dots)$.

Ursache für Rundungsfehler können Auslöschungen sein (Unterschiede zwischen zwei reellen Zahlen sind geringer als die Maschine darstellen kann).

3.4.4 Kondition

In einer Berechnung $y = f(x)$ mit gegebenem absolutem Fehler $\Delta x_j = |\tilde{x}_j - x_j|$ und $\Delta y_i = |\tilde{y}_i - y_i| = |f_i(\tilde{x}) - f_i(x)|$ berechnet sich der relative Fehler in y durch Taylor-Entwicklung: $\epsilon_{y_i} = \frac{\Delta y_i}{y_i} \approx \sum_{j=1}^n \left(\frac{x_j}{f_i(x)} \cdot \frac{df_i(x)}{dx_j} \right) \epsilon_{x_j}$. Dabei nennt man $\frac{x_j}{f_i(x)} \cdot \frac{df_i(x)}{dx_j}$ die *Konditionszahl* der Funktion f . Je größer diese ist, desto schlechter ist das Problem konditioniert (d.h. kleine Änderungen der Eingabe erzeugen eine große Änderung der Ausgabe).

Schlechte Konditionierung tritt z.B. auf bei der Subtraktion zweier nahezu gleich großer Zahlen oder bei Berechnungen mit relativ großen Zwischenwerten, wenn das Ergebnis recht klein ist.

3.4.5 Numerische Stabilität

numerisch stabil ein Berechnungsverfahren für ein gut konditioniertes Problem, bei dem sich der relative Eingabefehler nicht vergrößert.

numerisch instabil ein Berechnungsverfahren, das trotz kleiner Konditionszahl zu falschen Ergebnissen führt.

Durch mathematisch äquivalente Umformung zu einer Implementierung, die die Rundungsfehler nicht verstärkt lässt sich dieses Problem manchmal beheben.

3.4.6 Rückwärtsanalyse

Zum Nachweis der numerischen Stabilität von $\tilde{y} = y + \Delta y$ kann man auch wie folgt vorgehen:

1. zeige $\tilde{y} = f(x + \Delta x) - \tilde{y}$ kann als Ergebnis einer exakten Rechnung zu geänderten Eingabedaten erhalten werden.
2. zeige $|\Delta x| \leq C \cdot eps \cdot \|x\|$ – Änderungen in den Eingabedaten liegen in Größenordnung der Rundungsfehler bei der Eingabe.

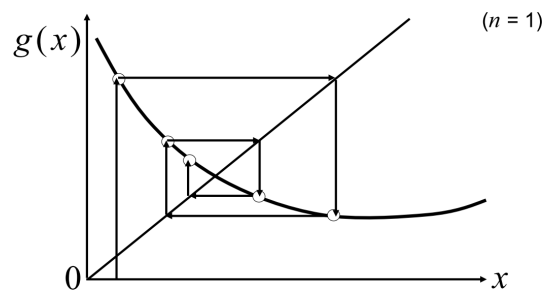
3.5 Berechnung nichtlinearer Gleichgewichtslösungen

Das Bestimmen der Punkte x_s sodass $f(x_s) = 0$ ist in vielfältigen Anwendungen wichtig (z.B. Bestimmung stationärer Punkte dynamischer Systeme, Kinematik, Optimierung). Mögliche (numerische Ansätze):

3.5.1 Differentialgleichungslöser

Wähle x_0 nahe genug an x_s und verwende einen Differentialgleichungslöser bis der stationäre Zustand erreicht ist.

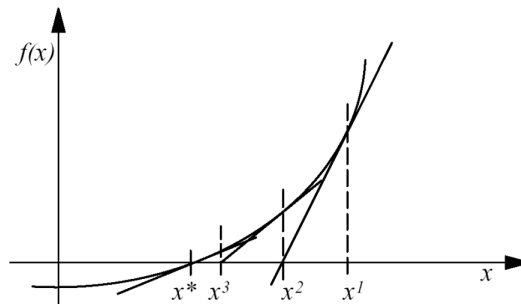
3.5.2 Fixpunktiteration



Wandele $f(x) = 0$ um in $x = g(x)$ (z.B. $g(x) := f(x) + x$) und einen Startwert x_0 nahe genug an x_s . Sei x_k die k -te Näherung der Lösung. Dann ist $x_{k+1} = g(x_k)$. Gilt die Konvergenzbedingung, dass x_0 nahe genug an x_s und die Eigenwerte von $\frac{dg}{dx}(x_s)$ im Einheitskreis, dann konvergiert x_k gegen x_s .

Die Konvergenz kann erreicht werden durch Wahl einer *Relaxationsmatrix* A : $0 = f(x) \Leftrightarrow 0 = A \cdot f(x)$. Wählt man $A = -\left[\frac{df}{dx}(x_s)\right]^{-1}$ mit $f(x_s) = 0$, sind alle Eigenwerte von $\frac{dg}{dx}(x_s) = 0$.

3.5.3 Newton-Verfahren



Das Newton-Verfahren ist ein spezialisiertes Fixpunktiterations-Verfahren: $x_{neu} := g(x) = x + \Delta x = x - \left[\frac{df}{dx}(x)\right]^{-1} \cdot f(x)$. Ablauf:

Initialisierung ($k = 0$) wähle Startvektor $x^{(0)}$

Iterationsschritt ($k \rightarrow k + 1$)

- berechne $f(x^{(k)})$
- berechne Jacobimatrix $\frac{df}{dx}(x^{(k)})$
- berechne Korrektur $\Delta x^{(k)}$ als Lösung des Gleichungssystems $\frac{df}{dx}(x^{(k)}) \cdot \Delta x^{(k)} = -f(x^{(k)})$
- berechne neue Näherung $x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$

Terminierungstest Beende Verfahren, wenn

„nahe genug“ an der Lösung

keine Änderung in $x^{(k+1)}$: $\|\Delta x^{(k)}\| \leq \epsilon_1 \cdot (1 + \|x^{(k)}\|)$

Funktionswerte klein genug: $\|f(x^{(k+1)})\| \leq \epsilon_2$

„kein Fortschritt“ mehr

keine Abnahme $\|f(x^{(k+1)})\| \geq \|f(x^{(k)})\| - \epsilon_3$

zu viele Iterationsschritte $k + 1 \geq k_{max}$

Sonst erhöhe k um 1 und führe nächsten Iterationsschritt durch.

Schwierigkeiten bei der Anwendung des Newton-Verfahrens stellen die Wahl eines geeigneten Startvektors $x^{(0)}$ (nahe genug an x_s), die Einzugsbereiche mehrerer Lösungen (Korrektur wird von anderer Lösung „angezogen“), Divergenz

und Singularitäten sowie hochgradig nichtlineare Probleme dar. Durch Schrittweitensteuerung kann diesen Problemen teilweise begegnet werden.

Das Berechnen der Jacobi-Matrix in jedem Iterationsschritt ist häufig aufwendig, daher wird diese häufig angenähert. Man untersucht z.B. die Jacobi-Matrix auf *Dünnbesetztheit* und approximiert nur die von 0 verschiedenen Einträge. Eine weitere Möglichkeit ist, die Jacobi Matrix $\frac{df}{dx}(x^{(k)})$ durch eine konstante Matrix (z.B. $\frac{df}{dx}(x^{(0)})$) zu ersetzen, allerdings ist das Newton-Verfahren dann nicht mehr quadratisch, sondern noch höchstens linear konvergent.

Das *Quasi-Newton-Verfahren* addiert in jedem Schritt die Approximation der Jacobi-Matrix auf: $J^{(k)} \approx \frac{df}{dx}(x^{(k)}) = J^{(k-1)} + U^{(k)}$, wobei $U^{(k)}$ eine $n \times n$ -Matrix von Rang 1 oder 2 ist, sodass $J^{(k)}$ die *Sekanten-Bedingung* erfüllt: $J^{(k)} \cdot (x^{(k)} - x^{(k-1)}) = f(x^{(k)}) - f(x^{(k-1)})$.

3.6 Numerische Lösung der nichtlinearen Zustandsdifferentialgleichungen

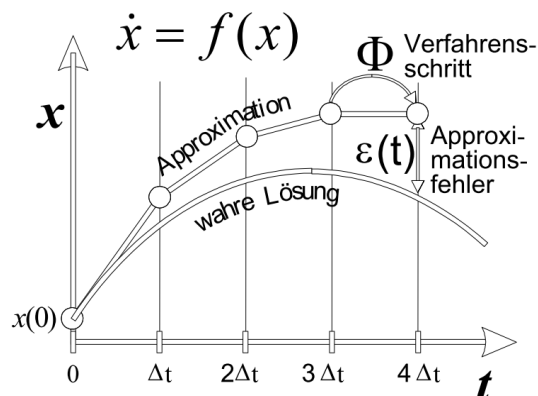
Zur Zustands-DGL $\dot{x} = f(x), x(0) = x_0$ wird die Lösung an diskreten Zeitpunkten $0 \leq t \leq t_f$ gesucht. Die Formale Integration der DGL liefert

$$\int_0^t \dot{x}(\tau) d\tau = x(t) - x(0) - \int_0^t f(x(\tau)) d\tau$$

$$\Rightarrow x(t_{k+1}) = x(0) + \int_0^{t_k} f(x(\tau)) d\tau + \int_{t_k}^{t_{k+1}} f(x(\tau)) d\tau = x(t_k) + \int_{t_k}^{t_{k+1}} f(x(\tau)) d\tau$$

Bei der numerischen Integration approximiert man die Fläche unter der Kurve (z.B. durch Rechtecke) zwischen t_k und t_{k+1} , wobei $t_{k+1} - t_k = h_k$. Bei *Einschrittverfahren* ist der allgemeine Ansatz zur Approximation $x_{k+1} = x_k + h \cdot \Phi(t_k, x_k, x_{k+1}, h; f)$ mit der Konsistenzbedingung $\lim_{h \rightarrow 0} \Phi(t_k, x_k, x_{k+1}, h; f) = f(x_k)$.

3.6.1 Explizites Euler-Verfahren

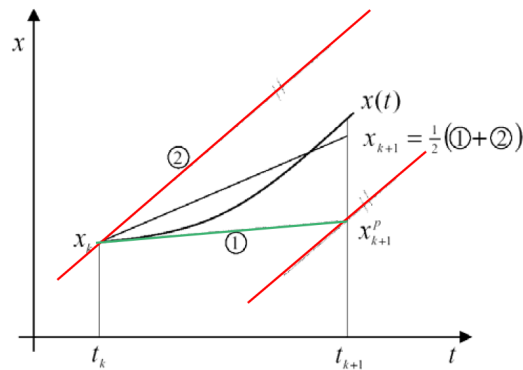


Beim Expliziten Euler-Verfahren approximiert man die Fläche zwischen t_k und t_{k+1} als $x(t_{k+1}) = x(t_k) + h_k \cdot f_k$ mit $f_k := f(x(t_k))$. Der Approximationsfehler beträgt $|x_{n_f} - x(t_f)| \leq C \cdot \max(h_k)$, das heißt, Explizite Euler-Verfahren hat lineare Konvergenz ($O(h^1)$).

3.6.2 Implizites Euler-Verfahren

Ähnlich dem Expliziten Euler-Verfahren wird die Fläche durch ein Rechteck approximiert, allerdings durch $x_{k+1} = x_k + h_k \cdot f(x_{k+1})$. Dazu muss in jedem Schritt die nichtlineare Gleichung $0 = x_{k+1} - x_k - h_k \cdot f(x_{k+1})$ gelöst werden. Dies macht das Verfahren wesentlich aufwendiger, allerdings hat es bei *steifen Differentialgleichungen* bessere Stabilitätseigenschaften.

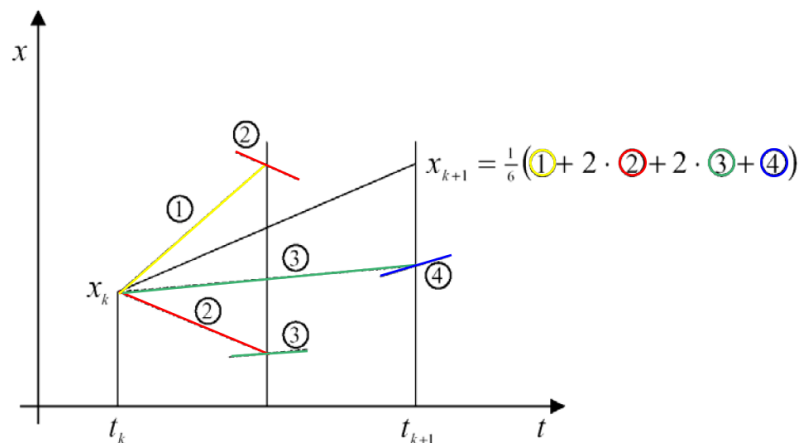
3.6.3 Heun-Verfahren



Das Heun-Verfahren ist ein *Prädiktor-Korrektor-Verfahren* und besteht aus zwei Schritten. Prädiktor: $x_{k+1}^p = x_k + h_k \cdot f(x_k)$ (= explizites Euler-Verfahren), Korrektor: $x_{k+1} = x_k + h_k \cdot \frac{1}{2} (f_k + f(x_{k+1}^p))$. Es ergibt sich:

- $s_1 = f(x_k)$
- $s_2 = f(x_k + h_k \cdot s_1)$
- $x_{k+1} = x_k + \frac{h_k}{2} (s_1 + s_2)$

3.6.4 Runge-Kutta-Verfahren



Das Runge-Kutta-Verfahren erweitert die Idee des Heun-Verfahrens und kommt so durch mehr Berechnungen in jeder Iteration auf eine Komplexität in $O(h^4)$. Allgemeiner Ansatz:

$$\begin{aligned}
 s_1: s_1 &= f(x_k) \\
 s_2: s_2 &= f(x_k + h \cdot \alpha_1 \cdot s_1) \\
 s_3: s_3 &= f(x_k + h \cdot \beta_1 \cdot s_1 + h \cdot \alpha_2 \cdot s_2) \\
 s_4: s_4 &= f(x_k + h \cdot \gamma_1 \cdot s_1 + h \cdot \beta_2 \cdot s_2 + h \cdot \alpha_3 \cdot s_3) \\
 x_{k+1}: x_{k+1} &= x_k + h \cdot \underbrace{(\delta_1 s_1 + \delta_2 s_2 + \delta_3 s_3 + \delta_4 s_4)}_{=\Phi}
 \end{aligned}$$

Man sucht nun die Koeffiziente $\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \gamma_1, \delta_1, \delta_2, \delta_3, \delta_4$ so, dass die Konsistenzbedingung $\lim_{h \rightarrow 0} \Phi(t_k, x_k, x_{k+1}, h; f) = f(x_k)$ erfüllt ist und der Approximationsfehler möglichst klein wird.

Das „klassische“ Runge-Kutta-Verfahren 4. Ordnung läuft dabei wie folgt ab:

$$s_1: s_1 = f(x_k)$$

$$s_2: s_2 = f(x_k + \frac{h}{2}s_1)$$

$$s_3: s_3 = f(x_k + \frac{h}{2}s_2)$$

$$s_4: s_4 = f(x_k + h \cdot s_3)$$

$$x_{k+1}: x_{k+1} = x_k + \frac{h}{6} \underbrace{((s_1 + 2s_2 + 2s_3 + s_4))}_{=\Phi}$$

3.6.5 Schrittweitensteuerung

Eine konstante Schrittweite h_k kann ungenau und ineffizient sein, daher wird h_k in jedem Schritt „so groß wie möglich“ und „so klein wie nötig“ gewählt.

2x Einschrittverfahren berechne zwei Näherungen von x_k für Schrittweite h_k und $\frac{h_k}{2}$, schätze daraus den lokalen Fehler. Passe h_k an, so dass lokale Fehlerschätzung unter vorgegebener Schranke liegt. Ist lokaler Fehler unter vorgegebener Schranke, akzeptiere Schritt (nutze $\frac{h_k}{2}$ für Näherung, da genauer) und mache Schrittweitevorschlag für nächsten Schritt. Sonst: wiederhole Schritt mit verändertem h_k .

2 Verfahren unterschiedlicher Ordnung berechne Näherung für x_k mit beiden Verfahren für h_k , schätze daraus lokalen Fehler. Dies ist wesentlich effizienter, da bei geeigneter Wahl der Verfahren (z.B. Runge-Kutta-Verfahren der 4. und 5. Ordnung) nur die letzten δ_i Koeffizienten neu berechnet werden.

3.7 Integration von Zustands-Differentialgleichungen mit Unstetigkeiten

Numerische Integrationsverfahren erfordern bisher, dass f mindestens so oft stetig differenzierbar ist, wie ihre Ordnung beträgt. Dies ist aber nicht zwangsläufig der Fall. Angenommen, $f(x, t)$ ist abschnittsweise mehrfach stetig differenzierbar, an den Übergängen (die durch Ereignisse zustandsabhängig oder zeitgesteuert eintreten können) möglicherweise unstetig. Ursachen dieser Unstetigkeiten können sein:

Stoßvorgänge sprunghafte Änderung der Geschwindigkeit durch Kollisionen

Reibung an den Übergängen zwischen Gleit- und Haftreibung

Strukturvariable Systeme die Anzahl der Freiheitsgrade/Dimensionen von x ändert sich

Approximation von Teilmodellen mit stückweise stetigen Funktionen (z.B. aus Tabellendaten)

Hysterese f hängt von der aktuellen Vorgeschichte von x ab

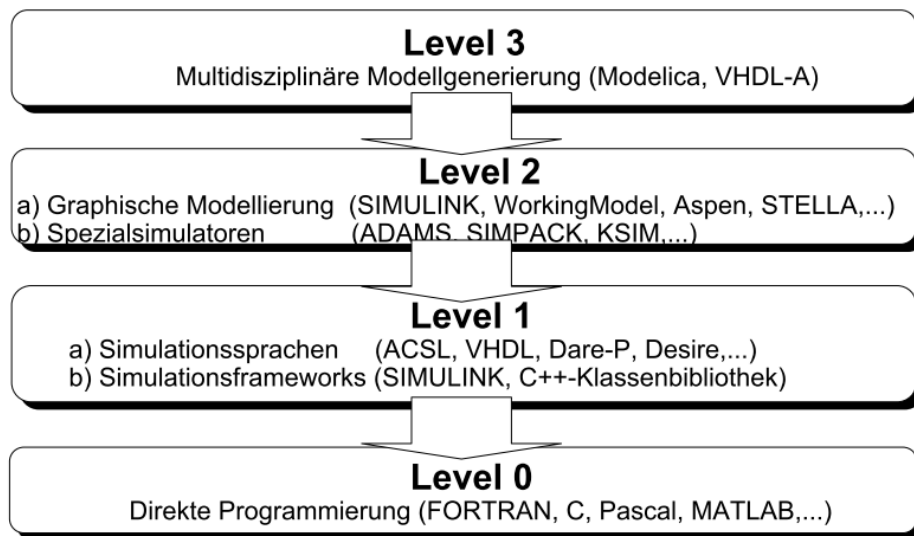
Unstetige Eingänge/Steuerungsgrößen ...

3.7.1 Schaltfunktionen

Schaltzeitpunkte $t_{s,i}$ werden im Allgemeinen als Nullstellen n_q von Schaltfunktionen $q_l(x(t_{s,i}), t_{s,i}) = 0, l \in \{1, \dots, n_q\}$ charakterisiert. Bei der numerischen Integration müssen nun die Vorzeichen der Schaltfunktion beobachtet werden: gibt es zwischen $x_k \approx x(t_k)$ und $x_{k+1} \approx x(t_{k+1})$ einen Vorzeichenwechsel in (mindestens) einer Schaltfunktion, muss der dazwischen liegende Schaltzeitpunkt bestimmt werden. So wird numerische Integration mit Nullstellensuche kombiniert. Vorgehen:

1. Prüfe Modell (Differentialgleichungen) auf mögliche Unstetigkeiten in x , der rechten Seite oder den ersten Ableitungen der rechten Seite.
 2. Bestimme Schaltzeitpunkte (events): Zeitgesteuert? Dann zu bekannten $t_{s,i}$. Zustandsabhängig? Dann Schaltpunkt als Nullstelle der Schaltfunktion $q_k(x(t_{s,i}), t_{s,i}) = 0, k \in \{1, \dots, n_q\}$.
 3. Integriere numerisch von einem Schaltpunkt zum nächsten, halte Schaltpunkte dabei genau ein (z.B. durch „root finding“).
 4. Setze Problem nach Schaltpunkt wieder auf: als neues Anfangswertproblem mit Anfangswert x aus einer Zustandsübergangsbedingung (z.B. $x(t_{s,i} + 0) = \xi(x(t_{s,i} - 0), t_{s,i})$).
-

3.8 Zeitkontinuierliche Simulationswerkzeuge



3.8.1 Level 0: Matlab

Beispiel: Anwendung des Euler-Verfahrens auf eine Schiffschaukel.

```
% Modellparameter
d = 100.0; % Reibungskonstante
m = 100.0; % Schaukelmasse incl. Mensch
l = 2.5;   % Abstand zu Drehachse
g = 10.0;  % Erdbeschleunigung

% Simulationsparameter
deltat = 0.1; % Schrittweite
tEnd = 20.0; % Endzeit

% Startwerte
t = 0.0; % Startzeit
phi = 1.0; % Startwinkel
omega = 0.0; % Startwinkelgeschwindigkeit

% Zeitschleife
while t <= tEnd
    % Ausgabe
    disp(sprintf('%8.4f %8.4f %8.4f', t, phi, omega));
    % rechte Seite
    dphi_dt = omega;
    domega_dt = -d/(m*l^2) * omega - g / l * sin(phi);
    % Euler-Schritt
    t = t + deltat;
    phi = phi + deltat * dphi_dt;
    omega = omega + deltat*domega_dt;
end;
```

3.8.2 Level 1: Differentialgleichungslöser in Matlab

Matlab bietet diverse numerische Methoden zum Lösen von Differentialgleichungen:

ode1 Explizites Euler-Verfahren

ode3 Bogacki-Shampine-Formel (ode23 mit fester Schrittweite)

ode2 Heun-Verfahren

ode4 Runge-Kutte-Verfahren 4. Ordnung

ode5 Dormand-Prince-Formel (ode45 mit fester Schrittweite)

ode113 Adams-Bashforth-Moulton PECE, variable Ordnung

ode15s Verfahren mit variabler Ordnung, basiert auf numerischer Differentiation

ode23 explizites Runge-Kutta (2,3) Paar

ode23s modifizierte Rosenbrock-Formel 2. Ordnung


ode23t Implementierung der Trapezoidschen Regel, nutzt eine „freie“ Interpolierende

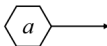
ode23tb TR-BDF2, implizites Runge-Kutta-Verfahren

ode45 explizite Runge-Kutta (4,5) Formel

3.8.3 Blockdiagrammsymbole

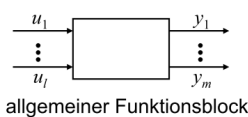
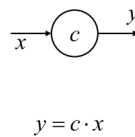
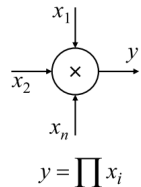
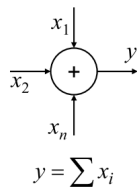
Eine beliebige Gleichung $\dot{x}(t) = f(x(t), u(t))$, $x(0) = x_0 \in \mathbb{R}^n$ kann als Blockdiagramm dargestellt werden. Dabei gibt es folgende Symbole:

Zustandsgrößen: 

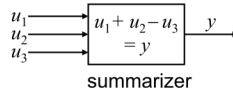
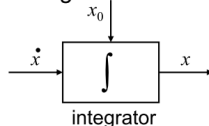
Eingaben und Parameter: 

Wirkfaktoren: $x \xrightarrow{-a} -ax$

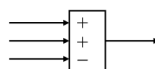
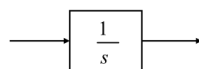
Funktionsblock: Ausgabe ist Funktion der Eingaben



Einige wichtige Funktionsblöcke



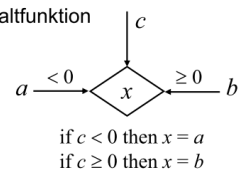
Darstellung in Matlab/Simulink



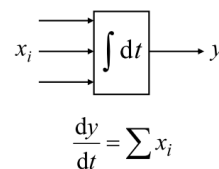
Funktionen



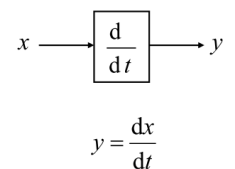
Schaltfunktion



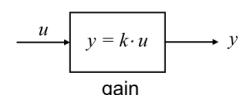
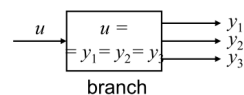
Integration



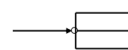
Differentiation



Einige wichtige Funktionsblöcke



Darstellung in Matlab/Simulink



4 Interpretation und Validierung

Überblick: Validierung entspricht einer systematischen Plausibilitätsüberprüfung. Fehlersuche, Konsistenzprüfung und Daten- und Parameterabgleich spielen eine Rolle.

Bei Modellierung und Simulation gibt es 4 hauptsächliche Fehlerquellen: Modellierungsfehler, Approximationsfehler (des iterativen Berechnungsverfahrens), Rundungsfehler und Programmier-/Implementierungsfehler. Dies erfordert Validierung.

Im Unterschied zu *Verifikation*, bei der der (meist mathematische) Nachweis der Korrektheit eines Programms geführt wird (was in der Regel bei komplexen Simulationsmodellen unmöglich ist), wird bei der *Validierung* durch die systematische Untersuchung und systematisches Testen die Wahrscheinlichkeit der Korrektheit eines Simulationsmodells lediglich erhöht.

Ein Simulationsmodell ist die *Implementierung* eines *Modells* und *Berechnungsverfahrens*. Somit müssen bei der Validierung 3 Aspekte überprüft werden:

Implementierung Syntaktische Fehlerfreiheit, Plausibilität („naturgetreues“ Verhalten) sowie numerische Korrektheit der Implementierung

Modell Zuverlässigkeit und logische Konsistenz (Anwendbarkeit), ausreichende Detailliertheit sowie Korrektheit der Modellparameter

Berechnungsverfahren Geeignetheit (für die numerische Lösung – explizit/implizit?), Approximationsfehler (Schrittweitensteuerung), Rundungsfehler

Mögliche Tests auf Plausibilität und Konsistenz:

Reproduktion von beobachtetem / „natürlichem“ Systemverhalten

Vorhersage von Verhalten: wie *plausibel* ist das simulierte Systemverhalten?

Verhaltensanomalien starke Unterschiede / Widersprüche des Simulationsmodells gegenüber dem realen System

Extremsituationen Simulation extremer Szenarien

Parametervariationen / -sensitivität Simulationsmodell sensitiv zu plausiblen Variationen der Modellparameter?

Anhand der Messwerte können die Modellparameter optimiert werden:

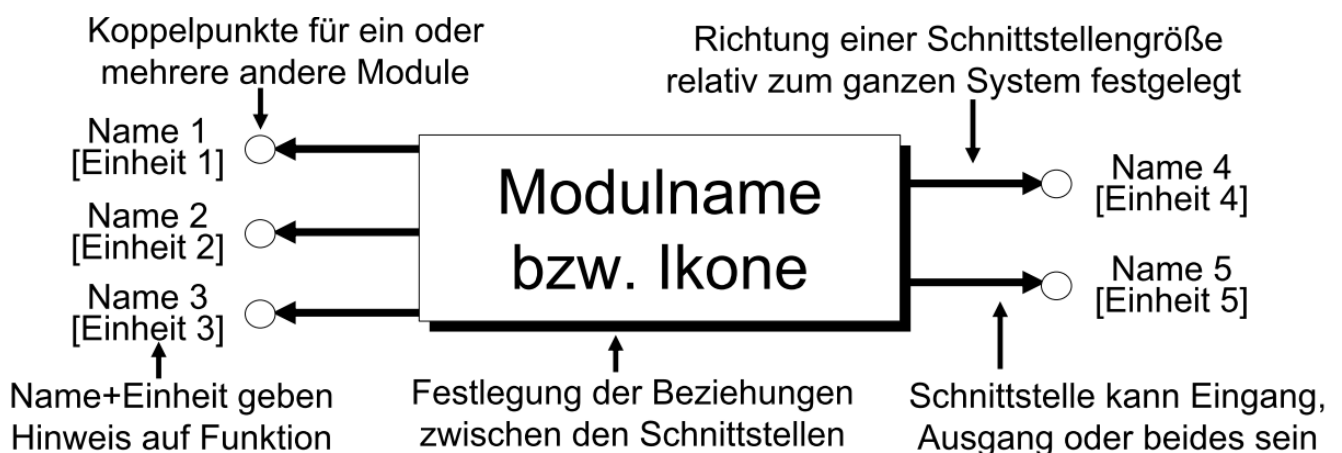
$$\min \varphi(x), \varphi(q) = \frac{1}{2} \sum_{j=1}^n \omega_j \|\hat{x}_j - x(t_j; p)\|^2 \in \mathbb{R}, \omega_j = \text{const} > 0, p \in \mathbb{R}^{n_p}$$

Kann so ein Parametersatz p gefunden werden, so dass die Abweichung des Systemverhaltens vom Experiment „klein“ ist, sind Simulationsmodell und Parameter „stimmig“. Andernfalls gibt es eine Reihe möglicher Erklärungen: das verwendete Optimierungsverfahren war ungeeignet, die experimentellen Daten waren nicht ausreichend relevant, die Messfehler in den Messdaten waren zu groß, oder das verwendete Modell war nicht detailliert genug.

5 Modulare Modellbildung und Simulation komplexer Systeme

Überblick: Modelle werden aus Modulen aufgebaut, für die Potential- und Flussgleichungen sowie Gesetze gelten. Daraus kann man differential-algebraische Gleichungen für das Modell ableiten.

5.1 Modulare Modellbildung



Jedes Modul ist über Schnittstellen S_i mit den übrigen Modulen verbunden. Für jedes Modul gibt es eigene Modellgleichungen, die die Schnittstellenvariablen miteinander verknüpfen.

Module werden an *Knoten* verbunden. Für die verschiedenen Variablen gilt dabei:

Flussvariablen Summe ist 0: $i_1 + i_2 + i_3 = 0$

Potentialvariablen sind gleich: $u_1 = u_2 = u_3$

Regeln für die automatische Modellgenerierung:

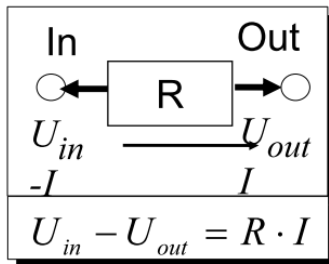
1. *Zerlegung* des Systems in Module mit Schnittstellen
2. Zuordnung einer *Potential-/Flussvariable* pro Schnittstelle
3. Formulierung von *Gleichungen* pro Modul unter ausschließlicher Benutzung von Schnittstellen / Konstanten
4. *Kopplung* zweier Module über kompatible Schnittstellen induziert Potential- und Flussgleichung.

Nach Aufstellen der Gleichungen (geschieht diese von Hand) sollten diese so vereinfacht werden bis alle verbleibenden Fluss- und Potentialvariablen zu einem einzigen Modul gehören.

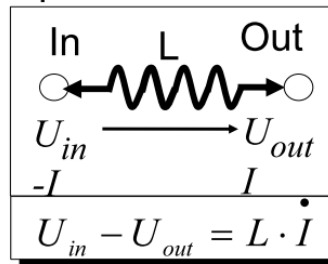
5.1.1 Elektrische Schaltungen

Elektrische Schaltungen werden aus folgenden Elementen zusammengebaut:

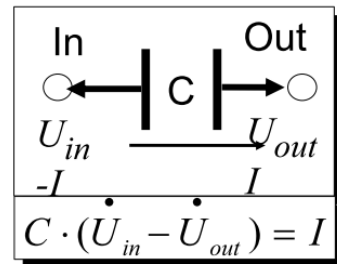
Widerstand



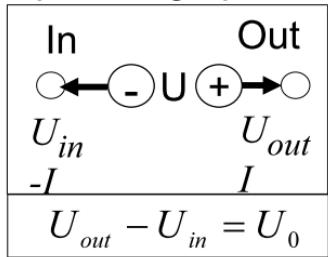
Spule



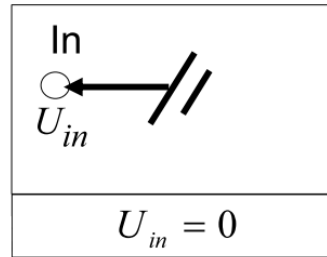
Kondensator



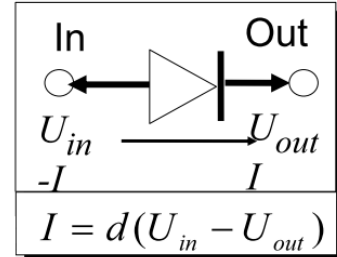
Spannungsquelle



Erde



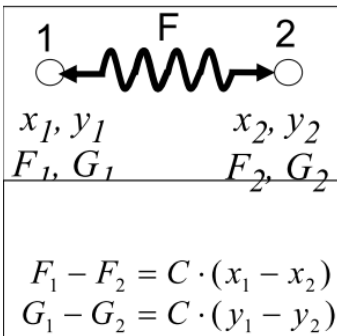
Diode



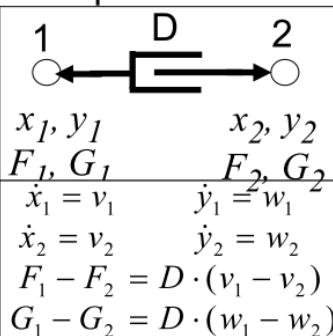
5.1.2 Mechanik

Mechanische Modelle werden aus folgenden Modulen zusammengebaut:

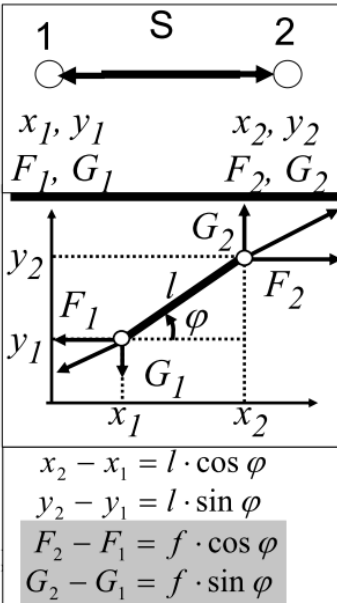
Feder



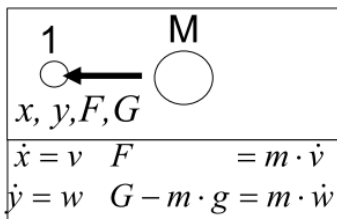
Dämpfer



Stab



Punktmasse



Bezugssystem

