

Systemtheorie und ein virtueller Trichter

**Seminar Syntax und Semantik
ZHAW, Zürich**

Florian Lüthi*

2. Juni 2013

*luethifl@students.zhaw.ch

Inhaltsverzeichnis

1	Einführung	3
2	Syntax und Semantik in der Systemtheorie	4
3	Illustrierendes System	5
3.1	Setup	5
3.2	Ermittlung der Ausgabe-Semantik	6
3.3	Ermittlung der Eingabe-Semantik	6
3.4	Ermittlung der Syntax	6
4	Bemerkungen zur Implementation	7
4.1	Umgebung	7
4.2	Libraries	7
4.3	Architektur	7
4.3.1	App	7
4.3.2	Funnel	7
4.3.3	OutflowDecoder	9
4.3.4	ShapeDecoder	10
4.3.5	LinearRegression	11
5	Schlussfolgerungen	12
	Literaturverzeichnis	13

1 Einführung

Diese Seminararbeit ist Teil des Seminars *Syntax und Semantik* an der ZHAW Zürich im Frühlingssemester 2013. Die Arbeit beschäftigt sich mit der allgemeinen Abgrenzung der Begriffe *Syntax* und *Semantik* in der Systemtheorie für physikalische Systeme. Das dazugehörige Programm **VirtualFauceteering** soll die Thematik beispielhaft an einem einfachen System, bestehend aus einem Wasserhahn und einem Trichter, illustrieren.

VirtualFauceteering wie auch dieses Dokument können von <https://github.com/foyan/VirtualFaucet> bezogen werden.

Der Autor bedankt sich bei den Dozenten Albert Heuberger und Beat Seeliger für die Kursführung.

2 Syntax und Semantik in der Systemtheorie

3 Illustrierendes System

3.1 Setup

Um die Wechselwirkung von Syntax und Semantik an einem physikalischen System illustrieren und simulieren zu können, wird die folgende Versuchsanordnung angenommen. Man beachte, dass wir uns mit unserer Anordnung im Flächenland befinden; dies bedeutet, dass alle involvierten Volumina zweidimensional als Flächen berechnet werden, in der Realität aber Rotationskörper wären.

- Ein Wasserhahn, der mit konstantem Ausflussvolumen entweder läuft oder nicht:

$$\Delta V_{\text{out}} \in \{0, 200\}$$

- Ein Trichter, dessen Wand durch die Radiusfunktion (abhängig von der Höhe h)

$$r(h) = \begin{cases} 4 & (h = 0) \\ r'(h) + 4 & (h > 0) \end{cases}$$

mit

$$r' : \mathbb{R} \rightarrow \mathbb{R}$$

beschrieben wird und der eine untere Trichteröffnung von $r(0) = 4$ besitzt.

Da das Volumen des Wassers innerhalb des Trichters bei bekannter Pegelhöhe der Integralfunktion

$$V(h) = 2 \cdot \int_0^h r(x) dx$$

entspricht, ist die Funktion der Pegelhöhe die Lösung h der folgenden Gleichung:

$$\int_0^h r(x) dx - \frac{1}{2} \cdot V_h = 0$$

Die Berechnung der Ausflussgeschwindigkeit v des Wassers aus dem Trichter ergibt sich gemäss dem Torricellischen Gesetz [?]:

$$v = \sqrt{2gh}$$

- Eine Messvorrichtung, welche die momentane Ausflussgeschwindigkeit messen kann.

3.2 Ermittlung der Ausgabe-Semantik

Bei gegebener Eingabe und bekannter Syntax ist die Ermittlung der Ausgabe trivial, denn sie entspricht direkt der gemessenen Ausflussgeschwindigkeit.

3.3 Ermittlung der Eingabe-Semantik

Bei gegebener Ausgabe und bekannter Syntax gestaltet sich die Ermittlung der Eingabe folgendermassen:

1. Ermittle die Pegelhöhe h innerhalb des Trichters mit der Umkehrung des Torricellischen Gesetzes:

$$h(t) = \frac{v^2(t)}{2 \cdot g}$$

2. Ermittle das Volumen des sich momentan im Trichter befindenden Wassers durch Integration über $r(h)$:

$$V(t) = 2 \cdot \int_0^{h(t)} r(x) dx$$

sowie das ausgeflossene Volumen durch

$$\Delta V_{\text{out}}(t) = 2 \cdot v(t) \cdot r(0) \cdot \Delta t.$$

3. Ermittle die totale Änderung des Volumens durch

$$\Delta V(t) = V_t - V_{t-\Delta t}$$

sowie daraus folgend das vom Wasserhahn in den Trichter geflossene Volumen durch

$$\Delta V_{\text{in}}(t) = \Delta V(t) + \Delta V_{\text{out}}(t).$$

3.4 Ermittlung der Syntax

Dies ist der schwierigste Teil. Bei gegebener Eingabe $\Delta V_{\text{in}}(t)$ und gegebenen Ausgaben $v(t)$ und $\Delta V_{\text{out}}(t)$ (sowie daraus abgeleitet der Höhe $h(t) = \frac{v^2(t)}{2g}$) ergeben sich die Messpunkte \tilde{r} durch

$$\tilde{r}(h(t)) = \frac{\Delta V_t}{2 \cdot \Delta h_t} = \frac{\Delta V_{\text{in}}(t) - \Delta V_{\text{out}}(t)}{2 \cdot h(t) - h(t - \Delta t)}.$$

Die notwendige analytische Beschreibung der Syntax kann in beschränktem Masse durch Regression erreicht werden.

4 Bemerkungen zur Implementation

4.1 Umgebung

Das verwendete Programmierumfeld ist HTML und JavaScript. Dadurch ist gewährleistet, dass das Programm auf allen möglichen Plattformen lauffähig ist. Ausserdem verlangt die sinnvolle Umsetzung der Thematik ein relativ stark ausgebautes User Interface zur Illustration. Dies ist mit HTML relativ einfach und mit geringem Aufwand möglich.

Alternativen wurden bewusst nicht evaluiert.

4.2 Libraries

Die eingesetzten JavaScript-Libraries sind:

- `jQuery` [1] zur Vereinfachung einiger DOM-Operationen sowie wegen einiger nützlicher Routinen zur funktionalen Programmierung wie `$.map()` und `$.grep()`,
- `gRaphaël` [?] für die Plots,
- `knockout.js` [?] für das einfache Binding von Modell und User Interface.

4.3 Architektur

Die Architektur des Programms ist in Abbildung 4.1 skizziert. In den folgenden Abschnitten werden die Komponenten kurz erläutert und wo notwendig, interessant und/oder sinnvoll Kommentare zum Code abgegeben.

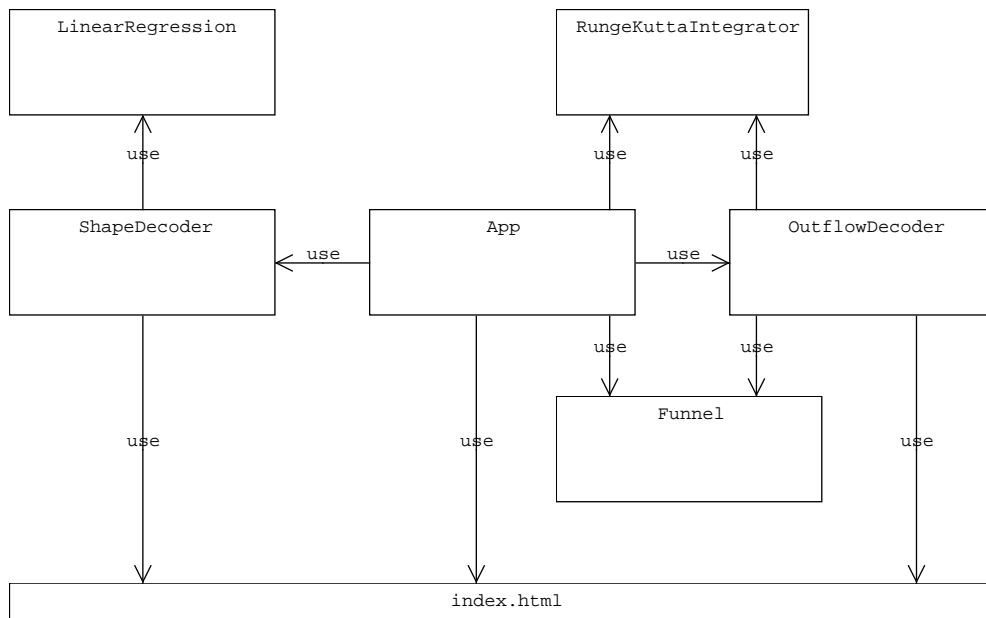
4.3.1 App

Diese Klasse hält den momentanen Zustand des Modells. Ausserdem ist sie verantwortlich für das Zeichnen des Trichters sowie weitere Teile des User Interfaces.

4.3.2 Funnel

Diese Klasse definiert den Trichter. Sie besteht mehr oder weniger aus der Methode `radius()`, welche den Radius des Trichters in Abhängigkeit von der Höhe zurückgibt:

Abbildung 4.1: Architektur




```

this.radius = function (h) {
  if (h <= 1) {
    return 4;
  }
  return 4 + self.r_h()(h-1);
}

```

`r_h` ist ein knockout-Observable, das die benutzerdefinierbare innere Funktion speichert.

4.3.3 OutflowDecoder

Diese Klasse repräsentiert den Decoder, welcher den Ausfluss aus dem Trichter analysiert und daraus den Einfluss wieder herstellt. Die Analyse passiert folgendermassen:

```

this.report = function (x, v) {

  var h = v > 0 ? v * v / 2.0 / 9.81 : 0;
  var V = self.integrator.integrate(0, 0, self.funnel.radius, h) * 2;

  var dV = V - self.prevV;
  var dVo = self.prevv * self.funnel.radius(0) * 2 * dt;
  var dVi = dV + dVo;

  /* [...] */

  var bit = dVi > 90.0;
  self.prevV = V;
  self.prevv = v;

  /* [...] */
}

```

Im Falle von Semantik in Form von Text muss noch der Text aus der binären Repräsentation wiederhergestellt werden:

```

this.report = function (x, v) {

  /* [...] */

  self.currentByte = ((self.currentByte << 1) + bit) & 255;
  var ch = self.isPrintable(self.currentByte) ? String.fromCharCode(self.currentByte) : "\u00A0";
  var newText = self.plainText[x % 8]() + ch;
  if (newText.length > 100) {
    newText = newText.substr(newText.length - 100, 100);
  }

  /* [...] */

  self.plainText[x % 8](newText);
}

```

`plainText[]` enthält acht `knockout`-Observables mit Text. Dies ist nötig, weil der Decoder nicht weiss, in welchem Zeitschritt die Übertragung begonnen hat, und es deswegen acht verschiedene Möglichkeiten gibt, den Bitstrom zu Bytes zusammenzusetzen.

Trotzdem macht natürlich nur eine dieser acht Textsequenzen Sinn. Welche das ist, versucht die eingebaute Gibberish Detection zu erkennen:

```
this.report = function (x, v) {  
  
    /* [...] */  
    for (var i = 0; i < 8; i++) {  
        self.gibberishFactors[i] *= 0.99;  
    }  
  
    self.gibberishFactors[x % 8] += self.isGibberish(self.currentByte) ?  
        0.0 : 0.125;  
  
    var minIndex = self.minGibberishIndex();  
    var minGibberish = self.gibberishFactors[minIndex];  
    for (var i = 0; i < 8; i++) {  
        if (self.gibberishFactors[i] > minGibberish) {  
            minGibberish = self.gibberishFactors[i];  
            minIndex = i;  
        }  
    }  
    self.minGibberishIndex(minIndex);  
  
    /* [...] */  
}
```

Die Gibberish Detection hält für jede der acht Sequenzen einem numerischen Wert vor, der sich um 0.125 erhöht, wenn an die Sequenz ein druckbares Zeichen angefügt wird. Vorgängig wird jeweils der Wert für alle Sequenzen um ein Prozent verringert – dies führt dazu, dass neue Übertragungen einigermaßen unabhängig von der jeweils vorhergehenden analysiert werden können.

Die verwendeten Werte sind empirisch, scheinen sich aber im Praxiseinsatz zu bewähren.

4.3.4 ShapeDecoder

Diese Klasse hat die Aufgabe, aus gegebener Eingabe und gegebener Ausgabe die Syntax wiederherzustellen – sprich die Radiusfunktion des Trichters.

Dies tut sie, indem sie lineare Regression für verschiedene Funktions-Skelette durchführen lässt und dann dasjenige mit dem kleinsten normierten Residuum selektiert:

```
this.guess = function () {  
    var reg = new LinearRegression();  
  
    var minR = 99999999;  
    var minRguess = -1;  
    for (var i = 0; i < self.guesses.length; i++) {  
        var res = reg.calculate(self.x, self.yr, self.guesses[i].mode);
```

```

        self.guesses[i].func = res.func;
        self.guesses[i].x(res.x);
        self.guesses[i].residual(res.residual);
        self.guesses[i].isLeader(false);

        if (res.residual < minR) {
            minRguess = i;
            minR = res.residual;
        }
    }

    if (minRguess != -1) {
        self.guesses[minRguess].isLeader(true);
    }
}

```

4.3.5 LinearRegression

Diese Klasse nähert eine Reihe von Messpunkten durch eine Funktion mit linearen Koeffizienten an. Das tut sie, indem sie aus den Messpunkten $(u_1, v_1), (u_2, v_2), \dots, (u_n, v_n)$ sowie den unbekannten Koeffizienten $x = \{x_1, x_2, \dots\}$ das überdeterminierte orthogonale Gleichungssystem

$$A^T A x = A^T b$$

mit

$$b = \{v_1, v_2, \dots\}$$

und

$$A = \begin{bmatrix} f_1(u_1) & f_2(u_1) & \dots & f_m(u_1) \\ f_1(u_2) & f_2(u_2) & \dots & f_m(u_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(u_n) & f_2(u_n) & \dots & f_m(u_n) \end{bmatrix}$$

ableitet. Durch eine *QR*-Zerlegung von $A^T A$ mit einer Householder-Transformation ([?])

5 Schlussfolgerungen

Literaturverzeichnis

- [1] The jQuery Foundation. jQuery, 2013.