Systemtheorie und ein virtueller Trichter

Seminar Syntax und Semantik
ZHAW, Zürich

Florian Lüthi*

3. Juni 2013

*luethifl@students.zhaw.ch

Inhaltsverzeichnis

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

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

  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 [4]:

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

- 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 $h(t) = \frac{v^2(t)}{2 \cdot g}$) 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` [3] zur Vereinfachung einiger DOM-Operationen sowie wegen einiger nützlicher Routinen zur funktionialen Programmierung wie `$.map()` und `$.grep()`,
- `gRaphaël` [1] 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.

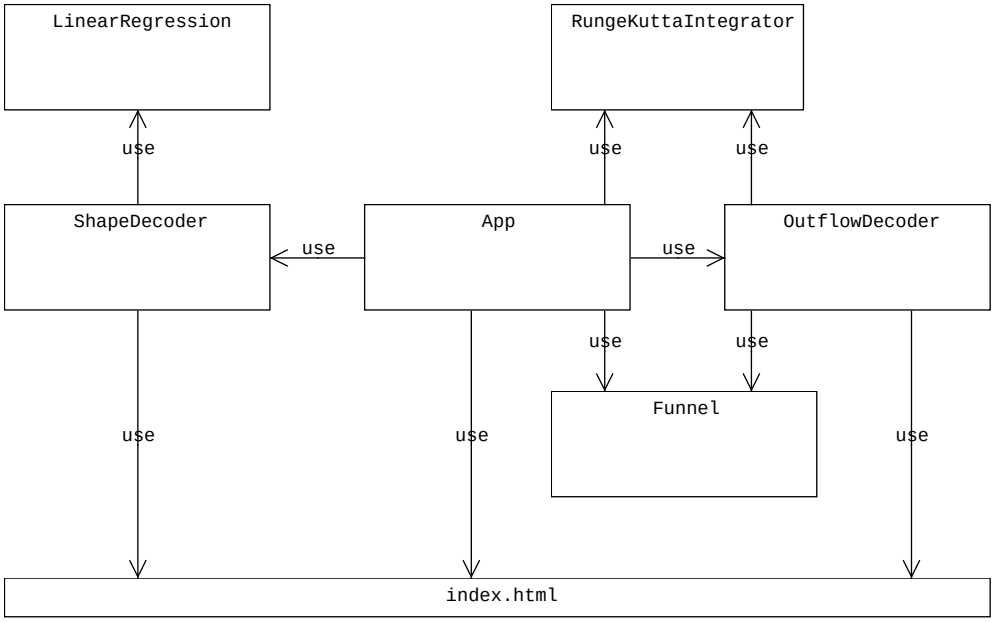
4.3.1 `App`

Diese Klasse hält den momentanen Zustand des Modells. Ausserdem ist sie verantwortlich für das Zeichnen des Trichters sowie weiter 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 $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ sowie den unbekannten Koeffizienten $x^* = \{x_1^*, x_2^*, \ldots\}$ das Ausgleichsproblem

$$||Ax^* - b||_2 = \min_{x \in \mathbb{R}^n} ||Ax - b||_2$$

ableitet [2]. Durch eine QR-Zerlegung von A mittels Householder-Transformation ([6]) kann das äquivalente Ausgleichsproblem

$$||QAx^* - Qb||_2 = \min_{x \in \mathbb{R}^n} ||Ax - b||_2$$

einfach gelöst werden, weil R eine obere Dreiecksgestalt hat. Die Residuen werden von den durch die Überdeterminierung überflüssigen Komponenten von Qb repräsentiert.

Die implementierte Methode verdeutlicht das Vorgehen:

```
this.calculate = function (u, v, mode) {

  var vs = [];

  var A = mode.getA(u);
  var R = self.getR(A, vs);
  var b = mode.getB(v);
  var Qb = self.transformB(b, vs, mode.coeffs);
  var x = self.solve(Qb, R, mode.coeffs);
  var res = self.getResidual(Qb, x);

  return {
    x: x,
    func: mode.getFunction(x),
    residual: res
  };

}
```

4.3.6 `RungeKuttaIntegrator`

Zur Berechnung der anfallenden Integrale implementiert die Klasse `RungeKuttaIntegrator` das Bogacki-Shampine-Verfahren ([5]). Es handelt sich dabei um ein Verfahren aus der Familie der expliziten Runge-Kutta-Verfahren, welches die Ordnung 3 hat und zur automatischen Steuerung der Schrittweite h ein eingebettetes Verfahren der Ordnung 2 aufweist.

Derlei Verfahren mit Schrittweitensteuerung sind im Allgemeinen massiv effizienter als Verfahren mit fester Schrittweite, haben allerdings das Problem, dass die genaue Berechnung eines Integrals zwischen zwei Grenzen nicht möglich ist.

Aus diesem Grund kombiniert diese Klasse die beiden Verfahren: Es wird mittels Bogacki-Shampine möglichst nahe an die obere Grenze heranintegriert. Danach wird für das restliche Intervall das klassische Runge-Kutta-4-Verfahren herangezogen:

```
this.integrate = function (y0, t0, y, tend) {
  var r = self.bogackiShampine(y0, t0, y, tend, 0.01, 0.00001);
  return self.rungeKutta4(r.ym1, r.tm1, y, tend).y;
}
```

Die Berechnung der Pegelhöhe h als Nullstelle der nichtlinearen Funktion $f(h) = \int_0^h r(x)dx - V_h$ verlangt eigentlich nach einem passenden Verfahren. Dies ist allerdings sehr ineffizient, da die mehrfache Auswertung der Integralfunktion sehr ineffizient ist.

Allerdings wissen wir, dass die Funktion $f(h) = \int_0^h r(x)dx$ (wie jede Integralfunktion) monoton steigend sein muss. Ausserdem wissen wir, dass sie sogar streng monoton steigend sein muss – ansonsten gäbe es ein x^*, sodass $r(x^*) = 0$; dies ist nicht möglich (denn dann wäre der Trichter nicht mehr durchlässig).

Durch dieses zusätzliche Problemwissen ist es uns möglich, die Nullstelle der Funktion wie folgt anzunähern:

1. Führe eine Bogacki-Shampine-Integration wie oben durch. Ändere aber die Abbruchbedingung dahingehend, dass anstelle auf $\texttt{t}_{\texttt{end}}$ auf das bekannte Endresultat $\texttt{y}_{\texttt{end}}$ geprüft wird:

   ```
   this.bogackiShampine = function (y0, t0, y, end, h, tol, breakonY) {

     /* [...] */

     while ((!breakonY && tn < end) || (breakonY && yn < end)) {
       /* [...] */
     }

     /* [...] */
   }
   ```

2. Das obige Problem mit den ungenauen Grenzen gilt natürlich auch für die inverse Integrationsmethode; das erhaltene $\texttt{t}_{\texttt{end}}$ ist also falsch und insbesondere zu tief. Aus diesem Grund nähern wir uns mit Bisektion der Nullstelle an, indem wir iterativ das Runge-Kutta-Verfahren mit Schrittweite $h = 0.5 \cdot (y_{\text{end}} - y_0)$ durchführen:

```
this.integrateReverse = function (y0, t0, y, yend) {
  var r = self.bogackiShampine(y0, t0, y, yend, 0.01,0.00001, true);

  var t0 = r.tm1;
  var tend = r.t;
  var y0 = r.ym1;

  for (var i = 0; i < 5; i++) {
    var r2 = self.rungeKutta4(y0, t0, y, tend, 0.5);
    if (r2.ym1 > yend) {
      tend = r2.tm1;
    } else {
      t0 = r2.tm1;
      y0 = r2.ym1;
    }
  }

  return t0;
}
```

5 Schlussfolgerungen

Literaturverzeichnis

[1] gRaphael. gRaphael. http://g.raphaeljs.com/, 2013. [Online; Stand 2. Juni 2013].

[2] Ralph Massjung. Linearer Ausgleich, 2013.

[3] The jQuery Foundation. jQuery. http://www.jquery.com, 2013. [Online; Stand 2. Juni 2013].

[4] Wikipedia. Ausflussgeschwindigkeit — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Ausflussgeschwindigkeit&oldid=118792897, 2013. [Online; Stand 2. Juni 2013].

[5] Wikipedia. Bogacki–Shampine method — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Bogacki%E2%80%93Shampine_method&oldid=551439672, 2013. [Online; accessed 2-June-2013].

[6] Wikipedia. QR decomposition — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=QR_decomposition&oldid=550994991, 2013. [Online; accessed 2-June-2013].