User Guide for `uc_algorithm`

Calculating Unimodular Completions for Hyperregular Polynomial Matrices with Entries in $\frac{\mathrm{d}}{\mathrm{d}t}$ [6, 7]

Klemens Fritzsche[1]

[1]Institute of Control Theory, Dresden University of Technology,
Klemens.Fritzsche@tu-dresden.de

1 Introduction

Unimodular matrices are a topic related to differential flatness in the context of nonlinear control. For an introduction see e.g. [8].

Given a dynamical systems in implicit form

$$\mathbf{0} = \mathbf{F}\left(\mathbf{x}, \dot{\mathbf{x}}\right), \quad \mathbf{x}\left(t\right) \in \mathbb{R}^n \tag{1}$$

with $p < n$ first order differential equations, the so called *tangent system* is determined by

$$\mathbf{0} = \left.\left(\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{x}}}\frac{\mathrm{d}}{\mathrm{d}t} + \frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)\right|_{\mathbf{F}=\mathbf{0}} \mathbf{v} = \mathbf{P}\left(\tfrac{\mathrm{d}}{\mathrm{d}t}\right)\mathbf{v} \tag{2}$$

and $\mathbf{v}\left(t\right) \in \mathbb{R}^n, \mathbf{P}_1\frac{\mathrm{d}}{\mathrm{d}t} + \mathbf{P}_0 =: \mathbf{P}\left(\frac{\mathrm{d}}{\mathrm{d}t}\right) \in \mathcal{M}^{p\times n}[\frac{\mathrm{d}}{\mathrm{d}t}]$, where $\mathcal{M}^{p\times n}[\frac{\mathrm{d}}{\mathrm{d}t}]$ denotes the set of $(p \times n)$-matrices with entries in $\frac{\mathrm{d}}{\mathrm{d}t}$. The matrix $\mathbf{P}\left(\frac{\mathrm{d}}{\mathrm{d}t}\right)$ is assumed to be hyperregular, such that there always exists a completion $\mathbf{H} \in \mathcal{M}^{(n-p)\times n}$ with

$$\begin{pmatrix} \mathbf{P}\left(\frac{\mathrm{d}}{\mathrm{d}t}\right) \\ \mathbf{H} \end{pmatrix} \in \mathcal{U}^n[\tfrac{\mathrm{d}}{\mathrm{d}t}], \tag{3}$$

where $\mathcal{U}^n[\frac{\mathrm{d}}{\mathrm{d}t}]$ denotes matrices of $\mathcal{M}^{n\times n}[\frac{\mathrm{d}}{\mathrm{d}t}]$ that possess a polynomial inverse in $\mathcal{M}^{n\times n}[\frac{\mathrm{d}}{\mathrm{d}t}]$ and will be called *unimodular*. The matrix $\mathbf{H}$ is called a *unimodular completion of* $\mathbf{P}\left(\frac{\mathrm{d}}{\mathrm{d}t}\right)$ and provides information about the flatness property of the system (1) (see [5, 7, 8, 9]).

The algorithm at hand performs both steps (2) and (3) for a specified system (1) and determines a matrix $\mathbf{H}$. In addition, this guide outlines how to interpret this result.

2 Installation

2.1 Prerequisities

The algorithm is implemented in Python 2.7.11, so make sure it is installed on your system. To check your Python version from the command line type

```
$ python --version
Python 2.7.11
```

Additionally, the algorithm is based on the following packages that need to be up to date:

- sympy
- numpy
- symbtools
- pycartan

It is recommended to install these using the the Python package index PyPI (see [1] for instructions):

```
$ pip install sympy numpy symbtools pycartan
```

2.2 Installing the algorithm

The quickest way to install the algorithm is to clone the git repository:

```
$ git clone https://github.com/klim-/franke_algorithm.git
```

3 Template example

Given an implicit system of the form (1), the following template code shows how it can be prepared for the algorithm:

```
# -*- coding: utf-8 -*-
import sympy as sp
import symbtools as st
from sympy import sin, cos, tan

# Number of state variables
n = 6

vec_x = st.symb_vector('x1:%i' % (n+1))
vec_xdot = st.time_deriv(vec_x, vec_x)
st.make_global(vec_x)
st.make_global(vec_xdot)

# Additional symbols
g, k1, k2 = sp.symbols("g, k1, k2")

# Time-dependent symbols
diff_symbols = sp.Matrix([k1, k2])

# Nonlinear system (state space representation)
# 0 = F_eq(x, xdot)
F_eq = sp.Matrix([
            [ xdot1 - x4 ],
            [ xdot2 - x5 ],
            [ xdot3 - x6 ],
            [ g*sin(x1) + xdot4*x3 ]])
```

The example file should be located within the example folder.

4 Usage of the algorithm

It is recommended to use the IPython Qt Console by running the bash command

```
$ ipython qtconsole
```

This will render the output of the algorithm in LaTeX. Inside the Qt Console run

```
>>> run uc_algorithm example/my_example.py
```

to start the script. Alternatively, the command from a standard shell is

```
$ python uc_algorithm example/my_example.py
```

The algorithm is iterative and the intermediate steps are not unique. This degree of freedom may well have an impact on the usability of the resulting matrix **H** or the script itself such that the user can choose between a *manual* and an *automatic* mode. However, we believe the automatic mode is utilizing this freedom fairly well and the bottlenecks are to be found elsewhere.

4.1 Example output

The following system is an academic example taken from [5]

```python
# -*- coding: utf-8 -*-
import sympy as sp
import symbtools as st

vec_x = st.symb_vector('x1:6')
vec_xdot = st.time_deriv(vec_x, vec_x)

st.make_global(vec_x)
st.make_global(vec_xdot)

F_eq = sp.Matrix([
        [xdot1 - x3*x4],
        [xdot2 - x4],
        [xdot3 - x5]])
```

Running the algorithm results in the following output:

```
$ python uc_algorithm.py examples/franke_ex1.py
x =
Matrix([
[x1],
[x2],
[x3],
[x4],
[x5]])

xdot =
Matrix([
[xdot1],
[xdot2],
[xdot3],
[xdot4],
[xdot5]])

0 = F(x,xdot) =
Matrix([
[-x3*x4 + xdot1],
[   -x4 + xdot2],
[   -x5 + xdot3]])

Enter "manual" for manual mode or hit enter:
```

Enter is parsed as automatic mode by default and gives:

```
i = 0 ##################################################################


P10 [3 x 5] =
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, 0]])


P00 [3 x 5] =
Matrix([
[0, 0, -x4, -x3,  0],
[0, 0,   0,  -1,  0],
[0, 0,   0,   0, -1]])

P10_roc [5 x 2] =
Matrix([
[0, 0],
[0, 0],
[0, 0],
[1, 0],
[0, 1]])


P10_rpinv [5 x 3] =
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1],
[0, 0, 0],
[0, 0, 0]])


P10_dot [3 x 5] =
Matrix([
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]])


A0 [3 x 3] =
Matrix([
[0, 0, -x4],
[0, 0,   0],
[0, 0,   0]])
```

```
B0 [3 x 2] =
Matrix([
[-x3,  0],
[ -1,  0],
[  0, -1]])


B0_loc [1 x 3] =
Matrix([[-1, x3, 0]])

i = 1 ################################################################


P11 [1 x 3] =
Matrix([[-1, x3, 0]])


P01 [1 x 3] =
Matrix([[0, 0, x4]])


P11_roc [3 x 2] =
Matrix([
[x3, 0],
[ 1, 0],
[ 0, 1]])


P11_rpinv [3 x 1] =
Matrix([
[-1],
[ 0],
[ 0]])


P11_dot [1 x 3] =
Matrix([[0, xdot3, 0]])


A1 [1 x 1] =
Matrix([[0]])


B1 [1 x 2] =
Matrix([[-xdot3, x4]])


--- special case -------------------------------------------------------


B1_tilde [1 x 1] =
Matrix([[-xdot3]])


P11_tilde_roc [3 x 1] =
Matrix([
[x3],
[ 1],
[ 0]])


Z1 [3 x 1] =
Matrix([
[x3*x4/xdot3],
[   x4/xdot3],
[          1]])


Z1_lpinv [1 x 3] =
Matrix([[0, 0, 1]])


########################################################################


Exit condition satisfied
```

```
H-matrix =
Matrix([
[-1, x3, 0, 0, 0],
[ 0,  0, 1, 0, 0]])


w[0] = (-1)dx1  +  (x3)dx2

w[1] = (1)dx3

############################################################################


Data saved to  examples/franke_ex1.pcl

############################################################################
```

5 Interpreting the results, checking for integrability

As indicated, the resulting matrix **H** may help to decide whether the nonlinear system is flat and if so in determining a flat output. The details about this process are stated in [7]. Helpful for this analysis are the toolboxes `pycartan` and `symbtools`. The results of the algorithm are stored in a `pcl`-file and can be parsed from a fresh Python-shell. For this, type

```
>>> import symbtools as st
>>> data = st.pickle_full_load("/path/to/franke_ex1.pcl")
```

```
>>> data.F_eq
Matrix([
[-x3*x4 + xdot1],
[   -x4 + xdot2],
[   -x5 + xdot3]])
>>> data.H
Matrix([
[-1, x3, 0, 0, 0],
[ 0,  0, 1, 0, 0]])
```

In order to study the vector 1-form $\boldsymbol{\omega} = \mathbf{H}\mathrm{d}\mathbf{x}$, a basis vector needs to be generated.

```
>>> basis_vec = data.vec_x
>>> basis_vec
Matrix([
[x1],
[x2],
[x3],
[x4],
[x5]])
```

The vector 1-form can now be generated using the package `pycartan`:

```
>>> import pycartan as pc
>>> omega = pc.VectorDifferentialForm(1, basis_vec, coeff=data.H)
>>> omega
Matrix([
[-1, x3, 0, 0, 0],
[ 0,  0, 1, 0, 0]])"dX"
```

The resulting vector 1-form `omega` can now be unpacked into ordinary 1-forms:

```
>>> omega1, omega2 = omega.unpack()
>>> omega1
(-1)dx1  +  (x3)dx2
>>> omega2
(1)dx3
```

Checking a 1-form for integrability can be achieved by computing its exterior derivative:

```
>>> omega1.d
(-1)dx2^dx3
>>> omega2.d
(0)dx1^dx2
```

Since the exterior derivative of the 1-form `omega2` is zero, it can be integrated which yields one of two components of a flat output $\mathbf{y} = (\mathtt{y1}, \mathtt{y2})$:

```
>>> y2 = omega2.integrate()
>>> y2
x3
```

For the second component `y1`, it can be shown that the frobenius theorem is fulfilled [3]:

```
>>> omega1.d^omega2
(0)dx1^dx2^dx3^dx4
```

Thus, the system is flat and there exists a factor $\mu\,(x_1, x_2, x_3)$ such that $\mathrm{d}\tilde{\omega}_1 = 0$ with $\tilde{\omega}_1 = \omega_1 + \mu\omega_2$. In order to determine μ, the exterior derivative of $\tilde{\omega}_1$ can be calculated which results in $\mathrm{d}\tilde{\omega}_1 = \mathrm{d}x_3 \wedge \mathrm{d}x_2 + \mathrm{d}\mu \wedge \mathrm{d}x_3 \overset{!}{=} 0$. It can easily be seen that $\mathrm{d}\mu = \mathrm{d}x_2$ and therefore $\mu = x_2$. This can be verified:

```
>>> st.makeGlobal(data.vec_x) # load elements from vec_x into namespace
>>> mu = x2
>>> omega1_tilde = omega1 + mu*omega2
>>> omega1_tilde.d
(0)dx1^dx2
```

To determine the remaining component `y1` of a flat output $\mathbf{y}$, we can integrate `omega1_tilde` as before:

```
>>> y1 = omega1_tilde.integrate()
>>> y1
-x1 + x2*x3
```

A flat output of this example system is $\mathbf{y} = (-x_1 + x_2x_3, x_3)$.

6 Implementation details

The algorithm described in [4] makes use of pseudo inverses and orthogonal complements of rectangular matrices. Since unimodular completions of hyper-regular matrices are not unique, there are degrees of freedom in the algorithm. Additionally, the iterative character makes the symbolic computation of intermediate steps slow if these freedoms are not chosen well. This section describes the underlying heuristics.

6.1 Pseudoinverses

Definition 6.1. *Given a matrix* $\mathbf{P} \in \mathcal{M}^{p\times n}$ *with* $\operatorname{rank}\mathbf{P} = p < n$*, the matrix* $\mathbf{P}^{+R} \in \mathcal{M}^{n\times m}$ *is called right pseudo inverse, if* $\mathbf{P}\mathbf{P}^{+R} = \mathbf{I}_p$ *is fulfilled.*

Definition 6.2. *Given a matrix* $\mathbf{P} \in \mathcal{M}^{p\times n}$ *with* $\operatorname{rank}\mathbf{P} = n < p$*, the matrix* $\mathbf{P}^{+L} \in \mathcal{M}^{n\times p}$ *is called left pseudo inverse, if* $\mathbf{P}^{+L}\mathbf{P} = \mathbf{I}_n$ *is fulfilled.*

As indicated, the goal is to choose pseudo inverses as *simple* as possible such that further iterations can be computed efficiently. While Moore-Penrose pseudo inverses are easy to determine [2], they lead to matrices with complicated entries in general and due to the iterative nature of the algorithm, the entries grow rapidly. Instead, pseudo inverses with preferably many 0 or 1 entries have proven favorable for further computations.

Let $\mathbf{P}$ be matrix in $\mathcal{M}^{p\times n}$ with $\operatorname{rank}\mathbf{P} = p < n$. By right multiplying with a permutation matrix $\mathbf{V}_\pi \in \mathbb{R}^n$, $\mathbf{P}$ can be sorted column-wise such that

$$\tilde{\mathbf{P}} = \mathbf{P}\mathbf{V}_\pi = (\mathbf{A}, \mathbf{B}), \quad \mathbf{A} \in \mathcal{M}^{p\times p}, \mathbf{B} \in \mathcal{M}^{p\times(n-p)}, \operatorname{rank}\mathbf{A} = p. \tag{4}$$

A right pseudo inverse of $\tilde{\mathbf{P}}$ can now be specified with

$$\tilde{\mathbf{P}}^{+R} = \begin{pmatrix} \mathbf{A}^{-1} \\ \mathbf{0}_{(n-p)\times p} \end{pmatrix}. \tag{5}$$

This guarantees at least $m \cdot (n-p)$ zeros in $\tilde{\mathbf{P}}^{+R}$. With $\mathbf{P}\mathbf{V}_\pi\tilde{\mathbf{P}}^{+R} = \mathbf{I}_p$ this leads to a right pseudo inverse for $\mathbf{P}$:

$$\mathbf{P}^{+R} = \mathbf{V}_\pi\tilde{\mathbf{P}}^{+R} = \mathbf{V}_\pi \begin{pmatrix} \mathbf{A}^{-1} \\ \mathbf{0}_{(n-p)\times m} \end{pmatrix}. \tag{6}$$

The computation of the left pseudo inverse can be done with equation (6) as well:

Let $\mathbf{T}$ be a matrix in $\mathcal{M}^{p\times n}$ with $\operatorname{rank}\mathbf{T} = n < p$. Transposing the equation $\mathbf{T}^{+L}\mathbf{T} = \mathbf{I}_n$ gives

$$\mathbf{I}_n = \left(\mathbf{T}^{+L}\mathbf{T}\right)^\top = \mathbf{T}^\top\left(\mathbf{T}^{+L}\right)^\top \equiv \mathbf{T}^\top\left(\mathbf{T}^\top\right)^{+R}. \tag{7}$$

Transposing this equation again yields

$$\mathbf{T}^{+L} = \left(\left(\mathbf{T}^\top\right)^{+R}\right)^\top. \tag{8}$$

The heuristic for these computations is in $\mathbf{V}_\pi$ and is about the choice of *simple* and linearly independent columns whereas *simple* is not specified yet. Criterias for that are *preferably many zeros*, *preferably few mathematical operations* or *preferably few variables.* In order to choose the criteria, the variable `pinv_optimization` in `core/algebra.py` needs to be set accordingly.

6.2 Orthogonal complements

The computation of orthogonal complements is not unique, either. Again, we are interested in *simple* complements such that further computations are efficient.

Definition 6.3. *The right ortho complement* $\mathbf{P}^{\perp R}$ *of a matrix* $\mathbf{P} \in \mathcal{M}^{p\times n}$ *with* $\operatorname{rank}\mathbf{P} = p < n$ *is defined by* $\mathbf{P}\mathbf{P}^{\perp R} = \mathbf{0}_{p\times(n-p)}$*, where* $\mathbf{P}^{\perp R} \in \mathcal{M}^{n\times(n-p)}$ *and* $\operatorname{rank}\mathbf{P}^{\perp R} = n - p$ *holds.*

Definition 6.4. *The left ortho complement* $\mathbf{P}^{\perp L}$ *of a matrix* $\mathbf{P} \in \mathcal{M}^{p\times n}$ *with* $\operatorname{rank}\mathbf{P} = n < p$ *is defined by* $\mathbf{P}^{\perp L}\mathbf{P} = \mathbf{0}_{(p-n)\times n}$*, where* $\mathbf{P}^{\perp L} \in \mathcal{M}^{(p-n)\times p}$ *and* $\operatorname{rank}\mathbf{P}^{\perp L} = p - n$ *holds.*

The preceding heuristic can be used for the computation of orthogonal complements, too:

Let $\mathbf{P}$ be a matrix in $\mathcal{M}^{p\times n}$ with $\operatorname{rank}\mathbf{P} = p < n$. The matrix $\mathbf{K} := \left(\mathbf{P}^{+R}, \mathbf{P}^{\perp R}\right) \in \mathcal{M}^{n\times n}$ is regular and there exists an inverse $\mathbf{K}^{-1} \in \mathcal{M}^{n\times n}$ which can be arrange into two hyper-columns $\mathbf{\Delta}_1$ and $\mathbf{\Delta}_2$ such that

$$\mathbf{I}_n = \mathbf{K}^{-1}\mathbf{K} = \begin{pmatrix}\mathbf{\Delta}_1\\ \mathbf{\Delta}_2\end{pmatrix}\begin{pmatrix}\mathbf{P}^{+R} & \mathbf{P}^{\perp R}\end{pmatrix} = \begin{pmatrix}\mathbf{\Delta}_1\mathbf{P}^{+R} & \mathbf{\Delta}_1\mathbf{P}^{\perp R}\\ \mathbf{\Delta}_2\mathbf{P}^{+R} & \mathbf{\Delta}_2\mathbf{P}^{\perp R}\end{pmatrix}. \tag{9}$$

It is obvious that $\mathbf{\Delta}_1 = \mathbf{P}$. As explained in the previous section, $\mathbf{P}$ can be sorted column-wise such that $\tilde{\mathbf{P}} = \mathbf{P}\mathbf{V}_\pi = (\mathbf{A}, \mathbf{B})$ with $\mathbf{A} \in \mathcal{M}^{p\times p}$ and $\operatorname{rank}\mathbf{A} = p$. The matrix $\tilde{\mathbf{P}}$ can then be completed such that

$$\tilde{\mathbf{K}}^{-1} := \mathbf{K}\mathbf{V}_\pi = \begin{pmatrix}\mathbf{A} & \mathbf{B}\\ \mathbf{0}_{(n-p)\times p} & \mathbf{I}_{n-p}\end{pmatrix}. \tag{10}$$

Due to equation (5), this results in

$$\begin{aligned}\mathbf{I}_n = \tilde{\mathbf{K}}^{-1}\tilde{\mathbf{K}} &= \begin{pmatrix}\mathbf{A} & \mathbf{B}\\ \mathbf{0}_{(n-p)\times p} & \mathbf{I}_{n-p}\end{pmatrix}\begin{pmatrix}\mathbf{A}^{-1} & \mathbf{C}\\ \mathbf{0}_{(n-m)\times m} & \mathbf{D}\end{pmatrix} &\text{(11a)}\\ &= \begin{pmatrix}\mathbf{I}_m & \mathbf{A}\mathbf{C} + \mathbf{B}\mathbf{D}\\ \mathbf{0}_{(n-p)\times p} & \mathbf{D}\end{pmatrix}. &\text{(11b)}\end{aligned}$$

It follows that $\mathbf{D} = \mathbf{I}_{n-p}$ and thus $\mathbf{C} = -\mathbf{A}^{-1}\mathbf{B}$. From

$$\tilde{\mathbf{K}} = \left(\tilde{\mathbf{P}}^{+R}, \tilde{\mathbf{P}}^{\perp R}\right) = \begin{pmatrix}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\\ \mathbf{0}_{(n-p)\times m} & \mathbf{I}_{n-p}\end{pmatrix} \tag{12}$$

follows

$$\left(\mathbf{I}_p, \mathbf{0}_{m\times(n-p)}\right) = \mathbf{P}\mathbf{K} = \mathbf{P}\mathbf{V}_\pi\begin{pmatrix}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\\ \mathbf{0}_{(n-p)\times m} & \mathbf{I}_{n-p}\end{pmatrix}. \tag{13}$$

A right orthogonal complement can be computed as

$$\mathbf{P}^{\perp R} = \mathbf{V}_\pi\begin{pmatrix}-\mathbf{A}^{-1}\mathbf{B}\\ \mathbf{I}_{n-p}\end{pmatrix}. \tag{14}$$

The computation of a left orthogonal complement can be deduced from that:

Let $\mathbf{T}$ be a matrix in $\mathcal{M}^{p\times n}$ with $\operatorname{rank}\mathbf{T} = n < p$. Transposing the equation $\mathbf{T}^{\perp L}\mathbf{T} = \mathbf{0}_{(p-n)\times n}$ leads to

$$\mathbf{0}_{n\times(p-n)} = \left(\mathbf{T}^{\perp L}\mathbf{T}\right)^{\top} = \mathbf{T}^{\top}\left(\mathbf{T}^{\perp L}\right)^{\top} \equiv \mathbf{T}^{\top}\left(\mathbf{T}^{\top}\right)^{\perp R}. \tag{15}$$

Transposing this equation again yields

$$\mathbf{T}^{\perp L} = \left(\left(\mathbf{T}^{\top}\right)^{\perp R}\right)^{\top}. \tag{16}$$

6.3 Heuristic for pseudo inverses and orthogonal complements

The heuristics for computing preferably simple pseudo inverses and orthogonal complements of $\mathbf{P} \in \mathcal{M}^{p\times n}$ with $\operatorname{rank}\mathbf{P} = p < n$ is in the function `reshape_matrix_columns(P)` in `core/algebra.py`. This function works as follows:

1. Remove zero columns of $\mathbf{P}$
2. Sort columns by *complexity* specified by `pinv_optimization` (free symbols or count operations)
3. Sort columns by number of zeros
4. Pick the first p linear independent columns and add to a matrix $\mathbf{A}$
5. Compare $\mathbf{A}$ and $\mathbf{P}$ and calculate permutation matrix $\mathbf{V}_\pi$ from that
6. Use $\mathbf{V}_\pi$ to calculate $\mathbf{B}$
7. Verify that $\mathbf{P}\mathbf{V}_\pi = (\mathbf{A}, \mathbf{B})$
8. Return $\mathbf{A}$, $\mathbf{B}$, $\mathbf{V}_\pi$

References

[1] Python package index `pypi`. `https://pypi.python.org/pypi`, February 2016.

[2] S. L. Campbell and C. D. Meyer. *Generalized inverses of linear transformations.* Classics in applied mathematics. SIAM, Philadelphia, 2009.

[3] G. Conte, C. H. Moog, and A. M. Perdon. *Algebraic Methods for Nonlinear Control Systems.* Springer, 2007.

[4] M. Franke and K. Röbenack. On the computation of flat outputs for nonlinear control systems. *European Control Conference (ECC)*, 2013.

[5] Matthias Franke. *Über die Konstruktion flacher Ausgänge für nichtlineare Systeme und zur Polzuweisung durch statische Ausgangsrückführungen.* Dissertation, Logos Verlag Berlin, 2014.

[6] Matthias Franke and Klaus Röbenack. Some remarks concerning differential flatness and tangent systems. *Proceedings in Applied Mathematics and Mechanics (PAMM)*, 2012.

[7] K. Fritzsche, M. Franke, C. Knoll, and K. Röbenack. Über die systematische bestimmung flacher ausgänge nichtlinearer mehrgrößensysteme. *at - Automatisierungstechnik*, submitted, 2016.

[8] Jean Lévine. *Analysis and Control of Nonlinear Systems*. Springer, 2009.

[9] Jean Lévine. On necessary and sufficient conditions for differential flatness. *Applicable Algebra in Engineering, Communication and Computing (AAECC)*, 2011.