Theory and Tools for Algorithmic Differentiation

An Informal Overview

Sebastian F. Walter

Montag, 29. April 2010

Very Brief Overview of Taylor Arithmetic

- Computation of directional derivatives (forward mode of AD) should be performed in Taylor arithmetic. For simplicity: **Univariate Taylor Polynomials** (**UTP**):

$$
\begin{aligned}
f : \mathbb{R}^N &\rightarrow \mathbb{R}^M \\
\frac{\mathrm{d}f}{\mathrm{d}x}(x_0)x_1 &= \frac{\mathrm{d}}{\mathrm{d}t} f(x_0 + x_1 t)\Big|_{t=0} \quad , \quad x_1 \in R^N \\
x_1^T \frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0)x_1 &= \frac{\mathrm{d}^2}{\mathrm{d}t^2} f(x_0 + x_1 t + 0t^2)\Big|_{t=0} \\
\frac{\mathrm{d}^d f}{\mathrm{d}x^d}(x_0)\{x_1, \dots, x_1\} &= \frac{\mathrm{d}^d}{\mathrm{d}t^d} f(x_0 + x_1 t + \dots + 0t^d)\Big|_{t=0} \\
[f]_D &= \sum_{d=0}^{D-1} f_d t^d = \sum_{d=0}^{D-1} \frac{1}{d!} \frac{\mathrm{d}^d}{\mathrm{d}t^d} f([x]_D)|_{t=0} t^D
\end{aligned}
$$

- f is a composite function of *elementary functions* $\phi_l \in \{+, -, *, /, \sin, \dots\}$, i.e. $f = \phi_L \circ \phi_{L-1} \circ \dots \phi_1$.
- it suffices to provide Taylor arithmetic implementations for all elementary functions $\{+, -, *, /, \sin, \dots\}$.
- Theory should generalize easily to **multivariate Taylor propagation**, but we only have limited experience with it.
- the UTP algorithms are also used in the **reverse mode of AD**

Algorithms for Univariate Taylor Polynomials over Scalars (UTPS)

- binary operations

$z = \phi(x, y)$	$d = 0, \ldots, D$	OPS	MOVES
$x + cy$	$z_d = x_d + c y_d$	$2D$	$3D$
$x \times y$	$z_d = \sum_{k=0}^{d} x_k y_{d-k}$	D^2	$3D$
x/y	$z_d = \frac{1}{y_0}\left[x_d - \sum_{k=0}^{d-1} z_k y_{d-k}\right]$	D^2	$3D$

- unary operations

$y = \phi(x)$	$d = 0, \ldots, D$	OPS	MOVES
$\ln(x)$	$\tilde{y}_d = \frac{1}{x_0}\left[\tilde{x}_d - \sum_{k=1}^{d-1} x_{d-k}\tilde{y}_k\right]$	D^2	$2D$
$\exp(x)$	$\tilde{y}_d = \sum_{k=1}^{d} y_{d-k}\tilde{x}_k$	D^2	$2D$
$\sqrt{x}$	$y_d = \frac{1}{2y_0}\left[x_d - \sum_{k=1}^{d-1} y_k y_{d-k}\right]$	$\frac{1}{2}D^2$	$3D$
x^r	$\tilde{y}_d = \frac{1}{x_0}\left[r\sum_{k=1}^{d} y_{d-k}\tilde{x}_k - \sum_{k=1}^{d-1} x_{d-k}\tilde{y}_k\right]$	$2D^2$	$2D$
$\sin(v)$ $\cos(v)$	$\tilde{s}_d = \sum_{j=1}^{d} \tilde{v}_j c_{d-j}$ $\tilde{c}_d = \sum_{j=1}^{d} -\tilde{v}_j s_{d-j}$	$2D^2$	$3D$
$\tan(v)$	$\tilde{\phi}_d = \sum_{j=1}^{d} w_{d-j}\tilde{v}_j$ $\tilde{w}_d = 2\sum_{j=1}^{d} \phi_{d-j}\tilde{\phi}_j$		
$\arcsin(v)$	$\tilde{\phi}_d = w_0^{-1}\left(\tilde{v}_d - \sum_{j=1}^{d-1} w_{d-j}\tilde{\phi}_j\right)$ $\tilde{w}_d = -\sum_{j=1}^{d} v_{d-j}\tilde{\phi}_j$		
$\arctan(v)$	$\tilde{\phi}_d = w_0^{-1}\left(\tilde{v}_d - \sum_{j=1}^{d-1} w_{d-j}\tilde{\phi}_j\right)$ $\tilde{w}_d = 2\sum_{j=1}^{d} v_{d-j}\tilde{v}_j$		

The General Method: Newton-Hensel Lifting

- Many functions are implicitly defined by an algebraic or differential equation:
  - multiplicative inverse: $y = x^{-1}$ by $0 = xy - 1$
  - exponential $y = e^x$ by $0 = \frac{\mathrm{d}y}{\mathrm{d}x} - y(x)$.
  - in general for independent x and dependent y:

$$0 \quad = \quad F(x, y)$$

- **Newton-Hensel Lifting**: Let $F([x], [y]_D) \stackrel{D}{=} 0$ and $F'([x], [y]_D) \mod t^D$ invertible. Then

$$\begin{aligned} 0 &\stackrel{D+E}{=} F([x], [y]_{D+E}) \\ 0 &\stackrel{D+E}{=} F([x], [y]_D) + F'([x], [y]_D)[\Delta y]_E t^D \\ [\Delta y]_E &\stackrel{E}{=} -\left(F'([x], [y]_E\right)^{-1} [\Delta F]_E \end{aligned}$$

- $[X]_D \equiv [X_0, \ldots, x_{D-1}] \equiv \sum_{d=0}^{D-1} x_d t^d$
- $[\Delta F]_E t^D \stackrel{D+E}{=} F([x], [y]_D)$

Example: Newton-Hensel Lifting applied to $y = x^{-1}$

- given $[x] \equiv \sum_{d=0}^{\infty} x_d t^d$ and x_0 invertible
- compute $[y]$ s.t. $1 = [x][y]$.
- let $0 \stackrel{D}{=} [x]_D [y]_D - 1$ already be satisfied. Then

$$
\begin{aligned}
0 &\stackrel{D+E}{=} [x]_{D+E}([y]_D + [\Delta y]_E t^D) - 1 \\
&\stackrel{D+E}{=} ([x]_{D+E}[y]_D - 1) + [x]_D [\Delta y]_E t^D \\
\therefore \quad 0 &\stackrel{E}{=} [\Delta F]_E + [x]_E [\Delta y]_E \\
\therefore \quad [\Delta y]_E &\stackrel{E}{=} -[y]_E [\Delta F]_E
\end{aligned}
$$

- Setting $E = D$ means that at each step the number of correct coefficients is doubled. The quantity $[\Delta F]_E$ can be computed by a convolution of $[x]_D$ and $[y]_D$ which can be done in $\mathcal{O}(D \log D)$ by using the FFT. Also, the multiplications can also be accelerated with the FFT. That means that the division is only a small constant more expensive than the multiplication.

Univariate Taylor Propagation on Matrices (UTPM)

- Application of Newton-Hensel lifting to the defining equations of matrix-valued functions.
- **Defining equations** of the QR decomposition:

$$
\begin{aligned}
0 &\stackrel{D}{=} [Q]_D[R]_D - [A]_D \\
0 &\stackrel{D}{=} [Q]_D^T[Q]_D - \mathrm{I} \\
0 &\stackrel{D}{=} P_L \circ [R]_D \ ,
\end{aligned}
$$

  where $(P_L)_{ij} = \delta_{i>j}$ and elementwise multiplication $\circ$.
- **Defining equations** of the symmetric eigenvalue decomposition

$$
\begin{aligned}
0 &\stackrel{D}{=} [Q]_D^T[A]_D[Q]_D - [\Lambda]_D \\
0 &\stackrel{D}{=} [Q]_D^T[Q]_D - \mathrm{I} \\
0 &\stackrel{D}{=} (P_L + P_R) \circ [\Lambda]_D \ .
\end{aligned}
$$

- etc.

UTPM Rectangular QR Decomposition, $E = 1$

input : $[A]_D = [A_0, \ldots, A_{D-1}]$, where $A_d \in \mathbb{R}^{M \times N}$, $d = 0, \ldots, D{-}1$, $M \geq N$.
output: $[Q]_D = [Q_0, \ldots, Q_{D-1}]$ matrix with orthonormal column vectors, where $Q_d \in \mathbb{R}^{M \times N}$, $d = 0, \ldots, D-1$
output: $[R]_D = [R_0, \ldots, R_{D-1}]$ upper triangular, where $R_d \in \mathbb{R}^{N \times N}$, $d = 0, \ldots, D-1$

$Q_0, R_0 = \mathrm{qr}\,(A_0)$

for $d = 1$ **to** $D - 1$ **do**

$$\Delta F = A_d - \sum_{k=1}^{d-1} Q_{d-k} R_k$$
$$S = -\frac{1}{2} \sum_{k=1}^{d-1} Q_{d-k}^T Q_k$$
$$P_L \circ X = P_L \circ (Q^T \Delta F R^{-1} - S)$$
$$X = P_L \circ X - (P_L \circ X)^T$$
$$\Delta R = Q^T \Delta F - (S + X) R$$
$$\Delta Q = (\Delta F - Q \Delta R) R^{-1}$$

end

Hands-On Example 1

- Recursive Function:

$$e(x,n) = \begin{cases} \frac{x^n}{n!} + e(x,n-1) & \text{if } n > 0 \\ 1 & \text{otherwise} \end{cases}$$

- Task: compute $\frac{\mathrm{d}^3}{\mathrm{d}x^3} e(x,n)$ for $n = 20$.

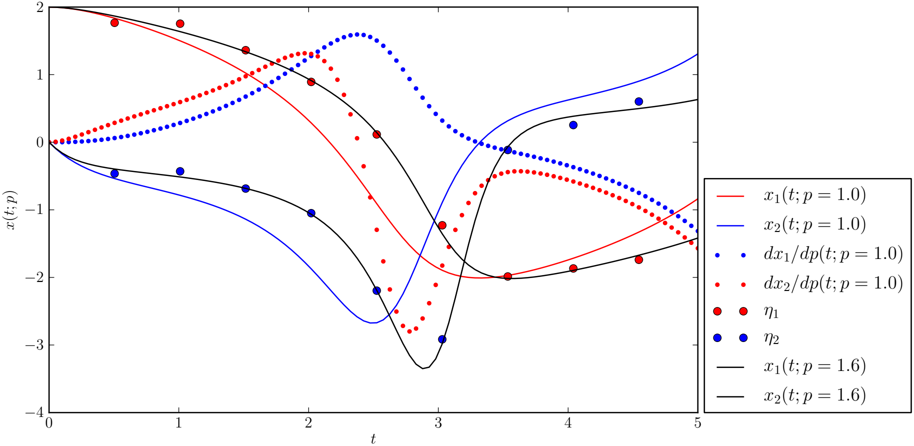
```
import numpy; import scipy
import taylorpoly

def e(x,n):
    """ recursive implementation of the exponential function"""
    if n == 0: return 1.
    else: return x**n/scipy.factorial(n) + e(x,n-1)

x = 1.

# Taylor Arithmetic
ax = taylorpoly.UTPS([x,1,0,0])
print 'error in first derivative = ', e(ax,20).data[1] - numpy.exp(x)
print 'error in third derivative = ', e(ax,20).data[3] - numpy.exp(x)/6.
```

Hands-On Example 2: General Least-Squares Fitting of an ODE Model



Van-Der-Pol Oscillator, $x(t)$ computed with PYSOLVIND, $h(t, x(t)) = x(t)$, least squares solution computed with scipy.optimize.leastsq

Example 2: General Least-Squares Fitting of an ODE Model (cont.)

- Van-Der-Pol Oscillator

$$
\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= p(1 - x_1(t)^2)x_2(t) - x_1(t) \\
x(0) &= 2\ ; \quad v(0) = 0\ .
\end{aligned}
$$

- measurement function

$$
h(t_i, x(t_i), p) = \sin(xt_i)/p\ ,
$$

- error model

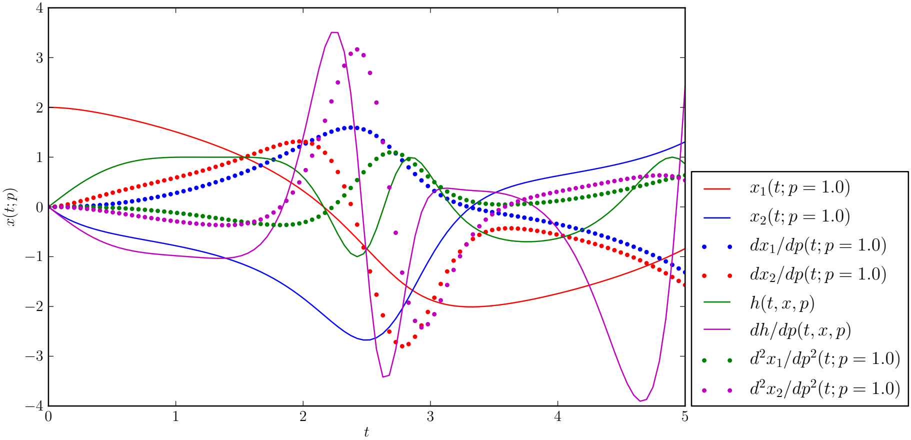
$$
\eta_i = h(t_i, x(t_i), p) + \epsilon_i\ ,
$$

  where ϵ_i iid $\sim \mathcal{N}(0, \sigma^2)$
- Gauss-Newton solver requires $F(p)$ and $\frac{\mathrm{d}F}{\mathrm{d}p}$, where

$$
F(p) = \Sigma^{-1}(h(t, x(t), p) - \eta)\ .
$$

- DAESOL-II returns $x(t_i, p)$, i.e. it is **necessary** to have algorithms for UTP to compute $h(t, x(t))$!

Example 2: (cont.)



Van-Der-Pol Oscillator, $x(t)$ computed with PYSOLVIND and $h(t, x(t)) = \sin(x(t)t)/p$ with PYADOLC.

Hands-On Example 3: Matrix Valued Functions

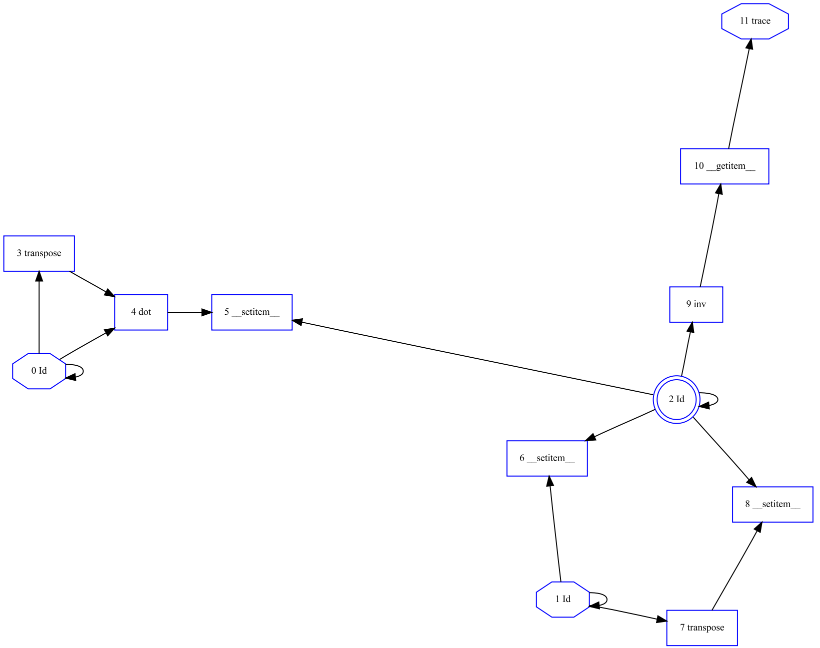
- Compute $\nabla_q^2 \mathrm{eigh}(C(q))$, where

$$C = (\mathrm{I}, 0) \begin{pmatrix} J_1^T J_1 & J_2^T \\ J_2 & 0 \end{pmatrix}^{-1} \begin{pmatrix} \mathrm{I} \\ 0 \end{pmatrix} .$$

```
import numpy
from algopy import CGraph, Function, UTPM; from algopy.globalfuncs import

def C(J1,J2):
    Np = J1.shape[1];   Nr = J2.shape[0]
    tmp = zeros((Np+Nr, Np+Nr), dtype=J1)
    tmp[:Np,:Np] = dot(J1.T,J1)
    tmp[Np:,:Np] = J2
    tmp[:Np,Np:] = J2.T
    return inv(tmp)[:Np,:Np]

D,P,Nm,Np,Nr = 2,1,50,4,3
cg = CGraph()
J1 = Function(UTPM(numpy.random.rand(D,P,Nm,Np)))
J2 = Function(UTPM(numpy.random.rand(D,P,Nr,Np)))
Phi = Function.eigh(C(J1,J2))[0][0]
cg.independentFunctionList = [J1,J2]; cg.dependentFunctionList = [Phi]
cg.plot('pics/cgraph.svg')
```


11 trace
10 __getitem__
3 transpose
4 dot
5 __setitem__
9 inv
0 Id
2 Id
6 __setitem__
8 __setitem__
1 Id
7 transpose

Connecting UTPM and UTPS

- Program:

$$
\begin{aligned}
A &= \begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix} \; ; \qquad B = \begin{pmatrix} x_3 & x_2 \\ x_1 & x_1 \end{pmatrix} \\
C &= AB \; ; \qquad \Phi = \mathrm{tr}(C)
\end{aligned}
$$

- Reverse Mode

$$
\begin{aligned}
\bar{\Phi} &= 1 \; ; \quad \bar{C} = \text{zeros\_like } (C) \; ; \\
\bar{B} &= \text{zeros\_like } (B) \\
\bar{A} &= \text{zeros\_like } (A) \; ; \\
\bar{x}_1 &= \bar{x}_2 = \bar{x}_3 = \bar{x}_4 = 0 \\
\bar{C} &+= \mathrm{I}\bar{\Phi} \\
\bar{A} &+= \bar{C}B^T \\
\bar{B} &+= \bar{C}^T A \\
\bar{x}_1 &+= \bar{A}_{11} \; ; \bar{x}_2 += \bar{A}_{12} \; \bar{x}_3 += \bar{A}_{21} \; ; \bar{x}_4 += \bar{A}_{22} \\
\bar{x}_3 &+= \bar{B}_{11} \; ; \bar{x}_2 += \bar{B}_{12} \; \bar{x}_1 += \bar{B}_{21} \; ; \bar{x}_1 += \bar{B}_{22}
\end{aligned}
$$

- Derivations:

$$
\begin{aligned}
\mathrm{tr}(\bar{\Phi} \mathrm{d}\Phi) &= \mathrm{tr}(\bar{\Phi} \sum_i \mathrm{d}C_{ii}) \\
&= \mathrm{tr}(\mathrm{I}\bar{\Phi} \mathrm{d}C) = \mathrm{tr}(\bar{C}^T \mathrm{d}C) \\
\mathrm{tr}(\bar{C}^T \mathrm{d}C) &= \mathrm{tr}(\bar{C}^T \mathrm{d}(AB)) \\
&= \mathrm{tr}(\bar{C}^T \mathrm{d}AB) + \mathrm{tr}(\bar{C}^T A \mathrm{d}B) \\
&= \mathrm{tr}(B\bar{C}^T \mathrm{d}A) + \mathrm{tr}(\bar{C}^T A \mathrm{d}B) \\
&= \mathrm{tr}(\bar{A} \mathrm{d}A) + \mathrm{tr}(\bar{B} \mathrm{d}B)
\end{aligned}
$$

Example: Symmetric Eigenvalue and QR decomposition

```
import numpy; from algopy import UTPM

# QR decomposition
D,P,M,N = 3,1,9,2
A = UTPM(numpy.random.rand(D,P,M,N))
Q,R = UTPM.qr(A)
print UTPM.dot(Q.T,Q) - numpy.eye(N)
print UTPM.dot(Q,R) - A

# symmetric eigenvalue decomposition
D,P,M,N = 3,1,4,4
Q,R = UTPM.qr(UTPM(numpy.random.rand(D,P,M,N)))
D = UTPM(numpy.zeros((D,P,N)))
D.data[0,0,:4] = [1,3,1,2]
D.data[1,0,:4] = [7,5,7,1]
D.data[2,0,:4] = [3,8,5,6]

D = UTPM.diag(D)
B = UTPM.dot(Q,UTPM.dot(D,Q.T))

D2,Q2 = UTPM.eigh(B)
print D2
```

Test Example for the Symmetric Eigenvalue Decomposition

- Orthonormal Matrix:[1]

$$
\begin{aligned}
Q(t) &= \frac{1}{\sqrt{3}} \begin{pmatrix} \cos(x(t)) & 1 & \sin(x(t)) & -1 \\ -\sin(x(t)) & -1 & \cos(x(t)) & -1 \\ 1 & -\sin(x(t)) & 1 & \cos(x(t)) \\ -1 & \cos(x(t)) & 1 & \sin(x(t)) \end{pmatrix} \\
\Lambda(t) &= \operatorname{diag}(x^2 - x + \frac{1}{2}, 4x^2 - 3x, \delta(-\frac{1}{2}x^3 + 2x^2 - \frac{3}{2}x + 1) + (x^3 + x^2 - 1), 3x - 1) \ ,
\end{aligned}
$$

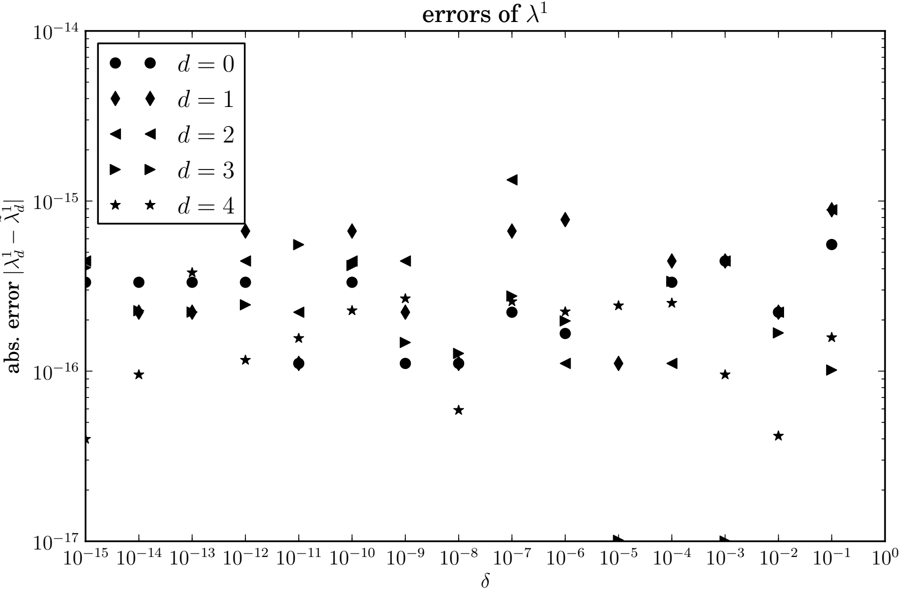
  where $x \equiv x(t) := 1 + t$.
- constant $\delta = 0$ means repeated eigenvalues, $\delta > 0$ distinct but close
- In Taylor arithmetic one obtains

$$
\begin{aligned}
\Lambda_0 &= \operatorname{diag}(1/2, 1, 1 + \delta, 2) \\
\Lambda_1 &= \operatorname{diag}(1, 5, 5 + \delta, 3) \\
\Lambda_2 &= \operatorname{diag}(2, 8, 8 + \delta, 0) \\
\Lambda_3 &= \operatorname{diag}(0, 0, 6 - 3\delta, 0) \\
\Lambda_d &= \operatorname{diag}(0, 0, 0, 0), \quad \forall d \geq 4 \ .
\end{aligned}
$$

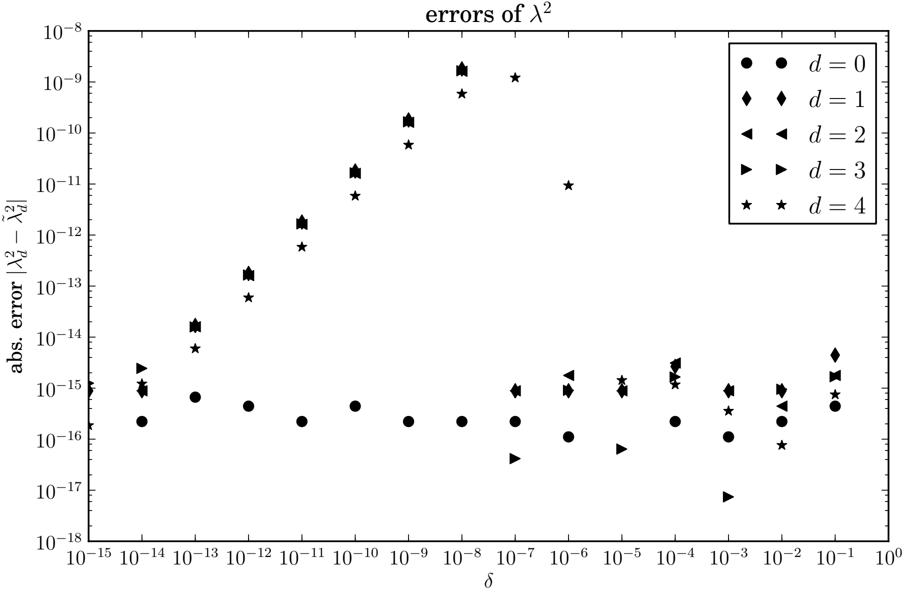
- Define $A(t) = Q(t)\Lambda(t)Q(t)$ and try to reconstruct $\Lambda(t)$ and $Q(t)$.

[1] Example adapted from Andrew and Tan, Computation of Derivatives of Repeated Eigenvalues and the Corresponding Eigenvectors of Symmetric Matrix Pencils, SIAM Journal on Matrix Analysis and Applications

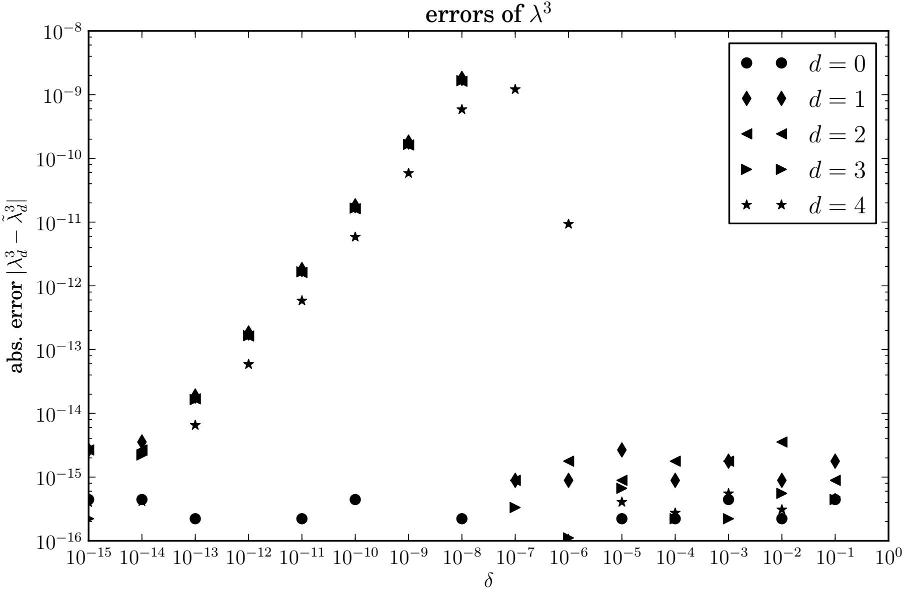
Illustrative example (cont.)



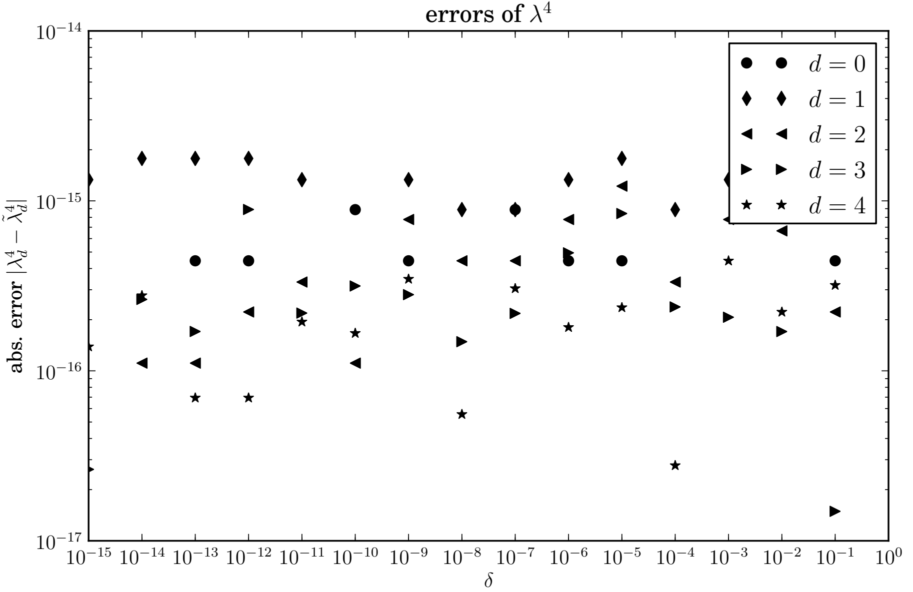
Illustrative example (cont.)



Illustrative example (cont.)



Illustrative example (cont.)



- Summary:
  - Have shown an overview of what can be done with UTPS and UTPM
  - outlined how UTPM and UTPS are connected
  - have shown software examples how to apply UTPS and UTPM in practice
  - have shown that one can connect the UTPS and UTPM algorithms to DAESOL-II
  - presented an algorithm to differentiate the symmetric eigenvalue decomposition with repeated eigenvalues
- Outlook:
  - Now it the goal to implement these algorithms efficiently in C to have a **library of building blocks for UTP**
  - New project: **TAYLORPOLY**[2]. Many useful algorithms already implemented with almost 100% test coverage in a unit test.
  - If you want to have a look: it's open source and www.github.com is useful to share and share back
  - liberal licence, i.e. you may take code snippets and use it in proprietary software

[2] www.github.com/b45ch1/taylorpoly