Univariate Taylor Polynomial Arithmetic Applied to Matrix Factorizations in the Forward and Reverse Mode

EuroAD 2010
Paderborn, 03.06.2010

Sebastian F. Walter\textsuperscript{3}, Lutz Lehmann\textsuperscript{4}

Montag, 30. April 2010

\textsuperscript{3}sebastian.walter@gmail.com
\textsuperscript{4}llehmann@mathematik.hu-berlin.de
PART I:
Motivation for Forward/Reverse Univariate Taylor Polynomial Arithmetic: Optimum Experimental Design
non-catalyzed and catalyzed reaction path
deactivation of the catalyst
batch process
measurements: product mass concentration
control of educt molar numbers, catalyst concentration, temperature profile
five unknown model parameters

\[ \dot{n}_1 = -k \cdot \frac{n_1 \cdot n_2}{m_{tot}}, \quad n_1(0) = n_{a1} \]
\[ \dot{n}_2 = -k \cdot \frac{n_1 \cdot n_2}{m_{tot}}, \quad n_2(0) = n_{a2} \]
\[ \dot{n}_3 = k \cdot \frac{n_1 \cdot n_2}{m_{tot}}, \quad n_3(0) = 0 \]

\[ k = k_1 \cdot \exp \left( -\frac{E_1}{R} \cdot \left( \frac{1}{T} - \frac{1}{T_{ref}} \right) \right) \]
\[ + k_{kat} \cdot c_{kat} \cdot \exp (-\lambda \cdot t) \cdot \exp \left( -\frac{E_{kat}}{R} \cdot \left( \frac{1}{T} - \frac{1}{T_{ref}} \right) \right) \]
\[ n_4 = n_{a4} \quad T = \vartheta + 273 \]
\[ m_{tot} = n_1 \cdot M_1 + n_2 \cdot M_2 + n_3 \cdot M_3 + n_4 \cdot M_4 \]
Optimum Experimental Design in Chemical Engineering (Cont.)

- **Dynamics**: Defined by ODE
- **Goal**: Estimate parameters $p = (k_1, k_{kat}, E_{kat}, \lambda, E_1)$
- **Problem**: Errors in the measurements $\eta$ result in errors in parameters $p$.
- **nonlinear regression** with additive iid normal errors

\[ \eta_m = h_m(t_m, x(t_m), p, q) + \varepsilon_m, \quad m = 1, \ldots, N_M \]
\[ \varepsilon_m \sim \mathcal{N}(0, \sigma_m^2) \]

- $\eta_m$ are measurements, $h$ **measurement model function** (connects model to the real world)
- Controls $q = (n_{a1}, n_{a2}, n_{a4}, c_{kat}, \theta)$ influence the error propagation.
- Therefore: Find controls $q$ such that the “uncertainty” in $p$ is as “small” as possible.
Simplified Derivation of an Uncertainty Measure

- **Unconstrained Nonlinear Parameter Estimation:**
  \[ \hat{p} = \arg\min_p \| F(p) \|_2^2, \]
  where \( F(p) = \Sigma^{-1}(\eta - h) \)
  measurements \( \eta \), measurement function \( h \in \mathbb{R}^{NM}, \Sigma \in \mathbb{R}^{NM \times NM} \),

- **Solution Operator:**
  \( J^\dagger : F \mapsto p \) of linearized parameter estimation
  \[ J^\dagger = (J(\hat{p})^T J(\hat{p}))^{-1} J(\hat{p})^T \]
  \[ J(p) = \frac{dF}{dp}(p) \]

- **Linear Error Propagation:**
  (computation of the covariance matrix \( C \)):
  \[ C := \mathbb{E}[(\hat{p} - \mathbb{E}[p])(\hat{p} - \mathbb{E}[p])^T] = J^\dagger \underbrace{\mathbb{E}[(\hat{F} - \mathbb{E}[F])(\hat{F} - \mathbb{E}[F])^T]}_{=I} (J^\dagger)^T \]
  \[ = (J^T J)^{-1} \]
  (independent of \( \hat{\eta} \))
Simplified Derivation of a Uncertainty Measure (cont.)

- Statistical Interpretation of the Covariance Matrix $C$:
  Defines **Confidence Region** $\text{CR}$:

$$\text{CR} := \left\{ p : (p - \hat{p})^T C^{-1} (p - \hat{p}) \leq N_p \hat{\sigma}^2 F(N_p, N_M - N_p, 1 - \alpha) \right\}$$

where $\alpha$ is statistical significance level, $F$ the F-distribution, $\hat{\sigma}$ unbiased estimate of the std

- **Typical Choices of Obj. Function:**

$$\Phi_A(q) = \frac{1}{N_P} \text{tr}(C) = \frac{1}{N_P} \text{tr}(J^T J)^{-1} \quad \text{A-criterion}$$

$$\Phi_D(q) = \det(K^T CK)^{\frac{1}{N_P}} \quad \text{D-criterion}$$

$$\Phi_E(q) = \max \left\{ \lambda : \lambda \text{ eigenvalue of } C \right\} \quad \text{E-criterion}$$

$$\Phi_M(q) = \max \left\{ \sqrt{C_{ii}}, i = 1, \ldots, N_P \right\} \quad \text{M-criterion}$$

- $K$ is a projection s.t. $K^T CK$ is regular
Overall Objective Function

- **Part I: Computation of $J_1$ and $J_2$**

  
  $J_1[n_{mts}, :] = \frac{\sqrt{\sigma_{mts}}}{\sigma_{mts}} \frac{\partial}{\partial (p, s)} (h(t_{n_{mts}}, x(t_{n_{mts}}; s, u(t_{n_{mts}}; q), q)))$

  $J_2 = \frac{\partial}{\partial (p, s)} r(q, p, s)$

- **Part II: Numerical Linear Algebra**

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

  $= \begin{pmatrix} Q_2^T (Q_2 J_1 J_2 Q_2)^{-1} Q_2 \end{pmatrix}$

  $\Phi = \lambda_1(C)$, max. eigenvalue

  where $J_2^T = (Q_1^T, Q_2^T)(L, 0)^T$

- **Computational Graph**

  ![Computational Graph](image)

  $N_{mts}$ Number measurement times, $\sigma$ std of a measurement, $q$ controls, $p$ nature given parameter, $s$

  pseudo-Parameter (e.g. initial values), $u$ control functions
Experimental Design Optimization: Required Derivatives

- Gradient-type optimizers require the gradient $\nabla_q \Phi(q)$
- thus: second order derivatives (mixed partial derivatives in parameters $p$ and control vector $q$)
- parameter robust OED:

$$\Phi_{\text{robust}}(q) := \phi(C(p, q)) + \gamma \left\| \frac{d}{dp} \phi(C(p, q)) \right\|_{2,\Sigma}$$

i.e. requires third order derivatives (twice in parameters $p$ and once in control vector $q$).
- Other objective functions may require even four’th and higher derivatives
- Matrices have often very high condition numbers (e.g. $J$)
- Number of controls $N_q$ is much larger than number of parameters $N_p \Rightarrow$ reverse mode of AD
- Want efficient, easy to use, flexible, numerically robust methods for forward/reverse mode AD
PART II:
Theory, Algorithms and Software
Reminder: Taylor Arithmetic

- Forward mode AD can be done by **Univariate Taylor Polynomial (UTP)** arithmetic:

\[
f : \mathbb{R}^N \rightarrow \mathbb{R}^M \\
\frac{df}{dx}(x_0)x_1 = \left. \frac{d}{dt} f(x_0 + x_1 t) \right|_{t=0}, \quad x_1 \in \mathbb{R}^N
\]

- \( f \) is a composite function of *elementary functions* \( \phi_l \in \{+, -, *, /, \sin, \ldots \} \), i.e. \( f = \phi_L \circ \phi_{L-1} \circ \ldots \phi_1 \).

- it suffices to provide Taylor arithmetic implementations for all elementary functions \( \{+, -, *, /, \sin, \ldots \} \).

- the UTP algorithms are also used in the **reverse mode of AD**
Algorithms for **Univariate Taylor Polynomials over Scalars** (UTPS)

### Binary Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Formula</th>
<th>Operations (OPS)</th>
<th>Moves (MOVES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z = \phi(x, y)$</td>
<td>$d = 0, \ldots, D$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x + cy$</td>
<td>$z_d = x_d + cy_d$</td>
<td>$2D$</td>
<td>$3D$</td>
</tr>
<tr>
<td>$x \times y$</td>
<td>$z_d = \sum_{k=0}^{d} x_k y_d - k$</td>
<td>$D^2$</td>
<td>$3D$</td>
</tr>
<tr>
<td>$x/y$</td>
<td>$z_d = \frac{1}{y_0} \left[ x_d - \sum_{k=0}^{d-1} z_k y_d - k \right]$</td>
<td>$D^2$</td>
<td>$3D$</td>
</tr>
</tbody>
</table>

### Unary Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Formula</th>
<th>Operations (OPS)</th>
<th>Moves (MOVES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = \phi(x)$</td>
<td>$d = 0, \ldots, D$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\ln(x)$</td>
<td>$\tilde{y}<em>d = \frac{1}{x_0} \left[ x_d - \sum</em>{k=1}^{d-1} x_d - k \tilde{y}_k \right]$</td>
<td>$D^2$</td>
<td>$2D$</td>
</tr>
<tr>
<td>$\exp(x)$</td>
<td>$\tilde{y}<em>d = \sum</em>{k=1}^{d} y_d - k \tilde{x}_k$</td>
<td>$D^2$</td>
<td>$2D$</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>$y_d = \frac{1}{2y_0} \left[ x_d - \sum_{k=1}^{d-1} y k y_d - k \right]$</td>
<td>$\frac{1}{2} D^2$</td>
<td>$3D$</td>
</tr>
<tr>
<td>$x^r$</td>
<td>$\tilde{y}<em>d = \frac{1}{x_0} \left[ r \sum</em>{k=1}^{d} y_d - k \tilde{x}<em>k - \sum</em>{k=1}^{d-1} x_d - k \tilde{y}_k \right]$</td>
<td>$2D^2$</td>
<td>$2D$</td>
</tr>
<tr>
<td>$\sin(v)$</td>
<td>$\tilde{s}<em>d = \sum</em>{j=1}^{d} \tilde{v}<em>j c</em>{d-j}$</td>
<td>$2D^2$</td>
<td>$3D$</td>
</tr>
<tr>
<td>$\cos(v)$</td>
<td>$\tilde{c}<em>d = \sum</em>{j=1}^{d} -\tilde{v}<em>j s</em>{d-j}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tan(v)$</td>
<td>$\tilde{\phi}<em>d = \sum</em>{j=1}^{d} w_{d-j} \tilde{v}_j$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\tilde{w}<em>d = 2 \sum</em>{j=1}^{d} \tilde{\phi}_{d-j} \tilde{\phi}_j$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\arcsin(v)$</td>
<td>$\tilde{\phi}<em>d = w_0^{-1} \left( \tilde{v}<em>d - \sum</em>{j=1}^{d-1} w</em>{d-j} \tilde{\phi}_j \right)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\tilde{w}<em>d = -\sum</em>{j=1}^{d} v_{d-j} \tilde{\phi}_j$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\arctan(v)$</td>
<td>$\tilde{\phi}<em>d = w_0^{-1} \left( \tilde{v}<em>d - \sum</em>{j=1}^{d-1} w</em>{d-j} \tilde{\phi}_j \right)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\tilde{w}<em>d = 2 \sum</em>{j=1}^{d} v_{d-j} \tilde{v}_j$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Apply UTP to Numerical Linear Algebra (NLA) Algorithms

- **Possibility 1**: Apply standard AD techniques to the NLA algorithms
  - will non-differentiable operations cause problems? (e.g. pivoting or treatment of degenerate cases)
  - how treat factorizations that are **not unique** in nominal solution (e.g eigenvalue decomposition with repeated eigenvalues). Possibly higher-order information makes it unique. That means that e.g. for \([y]_D = f([x]_D)\) it happens that \(y_0 = y_0(x_0, x_1, x_2, \ldots)\) and **not** \(y_0 = y_0(x_0)\) as usually assumed.

- **memory consumption**: NLA algorithms often have \(O(N^3)\) complexity, therefore also \(O(N^3)\) memory requirement? Always possible to reduce to \(O(N^2)\)?

- source trafo software featuring UTP?

- operator overloading software for UTP exists (ADOL-C, CppAD) but is relatively slow and needs retaping for program branches (pivoting...)

- code reuse of existing algorithms?

- performance: how hard to parallelize? Optimized implementations ala ATLAS? NLA is going to stay. But what about new coding paradigms?

- **Possibility 2**: Matrix Calculus Approach, topic of this talk
Newton’s Method

- Many functions are implicitly defined by algebraic equations:
  - multiplicative inverse: \( y = x^{-1} \) by \( 0 = xy - 1 \)
  - in general for independent \( x \) and dependent \( y \):
    \[
    0 = F(x, y)
    \]

- **Newton’s Method**\(^{29}\): Let \( F([x], [y]_D) \equiv 0 \) and \( F'([x], [y]_D) \mod t^D \) invertible. Then
  \[
  0 \equiv_{D+E} F([x], [y]_{D+E}) \\
  0 \equiv_{D+E} F([x], [y]_D) + F'([x], [y]_D)[\Delta y]_E t^D \\
  [\Delta y]_E \equiv_E - (F'([x], [y]_E)^{-1} [\Delta F]_E)
  \]

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

- if \( E = D \) then number of correct coefficients is doubled

\(^{29}\) also called Newton-Hensel lifting or Hensel lifting
Application of Newton’s Method to defining equations

**Defining equations** of the QR decomposition:

\[
0 \overset{D}{=} [Q]_D [R]_D - [A]_D \\
0 \overset{D}{=} [Q]_D^T [Q]_D - I \\
0 \overset{D}{=} P_L \circ [R]_D ,
\]

where \((P_L)_{ij} = \delta_{i>j}\) and element-wise multiplication \(\circ\).

**Defining equations** of the symmetric eigenvalue decomposition

\[
0 \overset{D}{=} [Q]_D^T [A]_D [Q]_D - [\Lambda]_D \\
0 \overset{D}{=} [Q]_D^T [Q]_D - I \\
0 \overset{D}{=} (P_L + P_R) \circ [\Lambda]_D .
\]

**Defining equations** of the Cholesky Decomposition

\[
0 \overset{D}{=} [L]_D [L]_D^T - [a]_D \\
0 \overset{D}{=} P_D \circ [L]_D - I \\
0 \overset{D}{=} P_R \circ [L]_D .
\]

etc...
Algorithm: Forward UTPM of the Rectangular QR Decomposition

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 $R_d \in \mathbb{R}^{N \times N}$, $d = 0, \ldots, D-1$.

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

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

$\Delta F = A_d - \sum_{k=1}^{d-1} Q_{d-k}R_k$

$S = -\frac{1}{2} \sum_{k=1}^{d-1} Q_{d-k}^T Q_k$

$P_L \circ X = P_L \circ (Q_0^T \Delta F R_0^{-1} - S)$

$X = P_L \circ X - (P_L \circ X)^T$

$R_d = Q_0^T \Delta F - (S + X)R_0$

$Q_d = (\Delta F - Q_0R_d)R_0^{-1}$

end
Algorithm: Reverse UTPM of the Rectangular QR Decomposition

\[
\begin{align*}
\text{input} & : [A]_D = [A_0, \ldots, A_{D-1}], \text{ where } A_d \in \mathbb{R}^{M \times N}, d = 0, \ldots, D-1, M \geq N. \\
\text{input} & : [Q]_D = [Q_0, \ldots, Q_{D-1}] \text{ matrix with orthonormal column vectors, where } Q_d \in \mathbb{R}^{M \times N}, d = 0, \ldots, D-1 \\
\text{input} & : [R]_D = [R_0, \ldots, R_{D-1}] \text{ upper triangular, where } R_d \in \mathbb{R}^{N \times N}, d = 0, \ldots, D-1 \\
\text{input/output} & : [\bar{A}]_D = [\bar{A}_0, \ldots, \bar{A}_{D-1}], \text{ where } \bar{A}_d \in \mathbb{R}^{M \times N}, d = 0, \ldots, D-1, M \geq N. \\
\text{input} & : [\bar{Q}]_D = [\bar{Q}_0, \ldots, \bar{Q}_{D-1}], \text{ where } \bar{Q}_d \in \mathbb{R}^{M \times N}, d = 0, \ldots, D-1 \\
\text{input} & : [\bar{R}]_D = [\bar{R}_0, \ldots, \bar{R}_{D-1}], \text{ where } \bar{R}_d \in \mathbb{R}^{N \times N}, d = 0, \ldots, D-1 \\
\end{align*}
\]

\[
[\bar{A}]_D = \bar{A}_D + ([\bar{Q}]_D - [Q]_D [Q]_D^T [\overline{Q}]_D [R]_D^{-T}) \\
+ [Q]_D \left( [\bar{R}]_D + P_L \circ ([R]_D [\bar{R}]_D^T - [\bar{R}]_D [R]_D^T + [Q]_D^T [\bar{Q}]_D - [\bar{Q}]_D^T [Q]_D)[R]_D^{-T} \right)
\]
ALGOPY Live Example: QR decomposition

```python
import numpy; from algopy import UTPM

# QR decomposition, UTPM forward
D, P, M, N = 3, 1, 5, 2
A = UTPM(numpy.random.rand(D, P, M, N))
Q, R = UTPM.qr(A)
B = UTPM.dot(Q, R)

# check that the results are correct
print 'Q.T Q - 1\n', UTPM.dot(Q.T, Q) - numpy.eye(N)
print 'QR - A\n', B - A
print 'triu(R) - R\n', UTPM.triu(R) - R

# QR decomposition, UTPM reverse
Bbar = UTPM(numpy.random.rand(D, P, M, N))
Qbar, Rbar = UTPM.pb_dot(Bbar, Q, R, B)
Abar = UTPM.pb_qr(Qbar, Rbar, A, Q, R)

print 'Abar - Bbar\n', Abar - Bbar
```

Sebastian F. Walter, Lutz Lehmann
Univariate Taylor Polynomial Arithmetic Applied to Matrix Factorizations in the Forward and Reverse Mode
Montag, 30. April 2010 17 / 28
import numpy; from algopy import CGraph, Function, UTPM, dot, qr, eigh, inv

D, P, M, N = 2, 1, 5, 2

# generate badly conditioned matrix A
A = UTPM(numpy.zeros((D,P,M,N)))

x = UTPM(numpy.zeros((D,P,M,1))); y = UTPM(numpy.zeros((D,P,M,1)))

x.data[0,0,:,:] = [1,1,1,1,1]; x.data[1,0,:,:] = [1,1,1,1,1]
y.data[0,0,:,:] = [1,2,1,2,1]; y.data[1,0,:,:] = [1,2,1,2,1]

alpha = 10**-5; A = dot(x,x.T) + alpha*dot(y,y.T); A = A[:,:,2]

# Method 1: Naive approach
Apinv = dot(inv(dot(A.T,A)),A.T)

print 'naive approach: A Apinv A - A = 0 
', dot(dot(A, Apinv) , A) - A

print 'naive approach: Apinv A Apinv - Apinv = 0 
', dot(dot(Apinv, A) , Apinv) - Apinv

print 'naive approach: (Apinv A)^T - Apinv A = 0 
', dot(Apinv, A.T) - dot(Apinv, A)

print 'naive approach: (A Apinv)^T - A Apinv = 0 
', dot(A, Apinv.T) - dot(A, Apinv)

# Method 2: Using the differentiated QR decomposition
Q,R = qr(A)
tmp1 = solve(R.T, A.T)
tmp2 = solve(R, tmp1)

Apinv = tmp2

print 'QR approach: A Apinv A - A = 0 
', dot(dot(A, Apinv) , A) - A

print 'QR approach: Apinv A Apinv - Apinv = 0 
', dot(dot(Apinv, A) , Apinv) - Apinv

print 'QR approach: (Apinv A)^T - Apinv A = 0 
', dot(Apinv, A.T) - dot(Apinv, A)

print 'QR approach: (A Apinv)^T - A Apinv = 0 
', dot(A, Apinv.T) - dot(A, Apinv)
Algorithm: Forward UTPM of Symmetric Eigenvalue Decomposition

\begin{algorithm}
\textbf{input} : \([A]_D = [A_0, \ldots, A_{D-1}]\), where \(A_d \in \mathbb{R}^{N \times N}\) symmetric positive definite, \(d = 0, \ldots, D-1\)
\textbf{output} : \([\tilde{\Lambda}]_D = [\tilde{\Lambda}_0, \ldots, \tilde{\Lambda}_{D-1}]\), where \(\Lambda_0 \in \mathbb{R}^{N \times N}\) diagonal and \(\Lambda_d \in \mathbb{R}^{N \times N}\) block diagonal
\(d = 1, \ldots, D-1\).
\textbf{output} : \(b \in \mathbb{N}^{N_{b}+1}\), array of integers defining the blocks. The integer \(N_B\) is the number of blocks. Each block has the size of the multiplicity of an eigenvalue \(\lambda_{n_b}\) of \(\Lambda_0\) s.t. for \(s[l] = b[n_b] : b[n_b + 1]\) one has \((Q_0[:], s[l])^TA_0Q_0[:], s[l]) = \lambda_{n_b}I.\)
\end{algorithm}

\begin{align*}
\Lambda_0, Q_0 &= \text{eigh} (A_0) \\
E_{ij} &= (\Lambda_0)_{ij} - (\Lambda_0)_{ii} \\
H &= P_B \circ (1/E)
\end{align*}

\begin{algorithm}
\textbf{for} \(d = 1\) \textbf{to} \(D - 1\) \textbf{do}
\begin{align*}
S &= -\frac{1}{2} \sum_{k=1}^{d-1} Q_{d-k}^T Q_k \\
K &= \Delta F + \tilde{Q}_0^T A_d \tilde{Q}_0 + S \Lambda_0 + \Lambda_0 S \\
\tilde{Q}_d &= Q_0 (S + H \circ K) \\
\tilde{\Lambda}_d &= P_B \circ K
\end{align*}
\textbf{end}
\end{algorithm}

- for the special case of distinct eigenvalues, this algorithm suffices
- for repeated eigenvalues this algorithm is one step in a little more involved algorithm
Test Example for the Symmetric Eigenvalue Decomposition\textsuperscript{44}

- Orthonormal Matrix:

\[
Q(t) = \frac{1}{\sqrt{3}} \begin{pmatrix}
\cos(x(t)) & 1 & \sin(x(t)) & -1 \\
-\sin(x(t)) & -1 & \cos(x(t)) & -1 \\
1 & -\sin(x(t)) & 1 & \cos(x(t)) \\
-1 & \cos(x(t)) & 1 & \sin(x(t)) \\
\end{pmatrix}
\]

\[
\Lambda(t) = \text{diag}(x^2 - x + \frac{1}{2}, 4x^2 - 3x, \delta(-\frac{1}{2}x^3 + 2x^2 - \frac{3}{2}x + 1) + (x^3 + x^2 - 1), 3x - 1),
\]

where \( x \equiv x(t) := 1 + t. \)

- constant \( \delta = 0 \) means repeated eigenvalues, \( \delta > 0 \) distinct but close

- In Taylor arithmetic one obtains

\[
\Lambda_0 = \text{diag}(1/2, 1, 1 + \delta, 2) \\
\Lambda_1 = \text{diag}(1, 5, 5 + \delta, 3) \\
\Lambda_2 = \text{diag}(2, 8, 8 + \delta, 0) \\
\Lambda_3 = \text{diag}(0, 0, 6 - 3\delta, 0) \\
\Lambda_d = \text{diag}(0, 0, 0, 0), \quad \forall d \geq 4.
\]

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

\textsuperscript{44} 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
Test Example for the Symmetric Eigenvalue Decomposition (cont.)

```plaintext
\begin{align*}
\text{abs. error } |\lambda_1^d - \tilde{\lambda}_1^d| \\
\text{errors of } \lambda_1^d = 0, 1, 2, 3, 4
\end{align*}
```

![Graph showing errors of \( \lambda^1 \) vs. \( \delta \)]
Test Example for the Symmetric Eigenvalue Decomposition (cont.)

errors of $\lambda^2$

<table>
<thead>
<tr>
<th>doi</th>
<th>10−15</th>
<th>10−14</th>
<th>10−13</th>
<th>10−12</th>
<th>10−11</th>
<th>10−10</th>
<th>10−9</th>
<th>10−8</th>
<th>10−7</th>
<th>10−6</th>
<th>10−5</th>
<th>10−4</th>
<th>10−3</th>
<th>10−2</th>
<th>10−1</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta$</td>
<td>10−18</td>
<td>10−17</td>
<td>10−16</td>
<td>10−15</td>
<td>10−14</td>
<td>10−13</td>
<td>10−12</td>
<td>10−11</td>
<td>10−10</td>
<td>10−9</td>
<td>10−8</td>
<td>10−7</td>
<td>10−6</td>
<td>10−5</td>
<td>10−4</td>
<td>10−3</td>
</tr>
<tr>
<td>abs. error $</td>
<td>\lambda^2_d - \tilde{\lambda}^2_d</td>
<td>$</td>
<td>errors of $\lambda^2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sebastian F. Walter, Lutz Lehmann
Test Example for the Symmetric Eigenvalue Decomposition (cont.)

![Graph showing errors of $\lambda^3$ vs $\delta$]

-the errors $|\lambda^3 - \tilde{\lambda}^3|$ for different values of $d$ (from 0 to 4).
Sebastian F. Walter\textsuperscript{53}, Lutz Lehmann\textsuperscript{54} () Univariate Taylor Polynomial ArithmeticApplied t
The E-Criterion of the Opt. Exp. Design Problem

- Compute $\nabla_q \text{eigh}(C(q))$, where

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

```python
import numpy
from algopy import CGraph, Function, UTPM, dot, inv, zeros, eigh

def C(J1, J2):
    """generic implementation of the covariance computation""
    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.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')
```
# Some Software for Forward/Reverse UTP

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Status</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>algopy</td>
<td>forward/reverse UTPM in Python</td>
<td>alpha</td>
<td>10388</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.github.com/b45ch1/algopy">www.github.com/b45ch1/algopy</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pysolvind</td>
<td>Python Bindings to SolvIND/DAESOL-II</td>
<td>alpha</td>
<td>9743</td>
</tr>
<tr>
<td>pyadolc</td>
<td>Python Bindings to ADOL-C (C++)</td>
<td>stable</td>
<td>6895</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.github.com/b45ch1/pyadolc">www.github.com/b45ch1/pyadolc</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pycppad</td>
<td>Python Bindings to CppAD (C++)</td>
<td>stable</td>
<td>1334</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.github.com/b45ch1/pycppad">www.github.com/b45ch1/pycppad</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>taylorpoly</td>
<td>forward/reverse UTPS/UTPM (C)</td>
<td>alpha</td>
<td>9276</td>
</tr>
<tr>
<td></td>
<td>includes Python bindings</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.github.com/b45ch1/taylorpoly">www.github.com/b45ch1/taylorpoly</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LOC include unit tests but exclude comments (about 25% of the line count are comments)
- **Summary:**
  - Have a fairly complete set of useful tools in Python now
  - TAYLORPOLY hosts ANSI-C algorithms that can be used from basically all programming languages

- **Outlook:**
  - Reverse mode of $QR$ decomposition of quadratic by singular matrices
  - Reverse mode of the symmetric eigenvalue decomposition for the case of repeated eigenvalues
  - Derive UTPM algorithm for the Singular Value Decomposition and generalized eigenvalue decomposition
  - Port all existing algorithms from ALGOPY to TAYLORPOLY