# Not So Short Tutorial On Algorithmic Differentiation

Sebastian F. Walter, HU Berlin

Wednesday, 04.06.2010

## What is Algorithmic Differentiation?

- Name confusion: Algorithmic Differentiation aka Automatic Differentiation aka Computational Differentiation aka **AD**
- Considered one of the most important algorithmic techniques invented in the 20'th century[1]
- Has distinct advantages over Finite Differences (FD) and Symbolic Differentiation (SD): more efficient **and** numerically stable **and** *relatively* easy to use.
- Can also be done by hand instead of symbolic differentiation (examples follow).

[1] Nick Trefethen, http://www.comlab.ox.ac.uk/nick.trefethen/inventorstalk.pdf

## Computational Model

- computer programs are a sequence of elementary functions $\phi_l \in \{+, -, *, /, \sin, \exp, \dots\}$
- Example: Evaluate Function $f(3,7)$:

$$
\begin{aligned}
f : \mathbb{R}^2 &\to \mathbb{R} \\
x &\mapsto y = f(x) = \sin(x_1 + \cos(x_2)x_1)
\end{aligned}
$$

- Computational Trace:

| | | | | | |
|---|---|---|---|---|---|
| independent | $v_{-1}$ | = | $x_1$ | = | 3 |
| independent | $v_0$ | = | $x_2$ | = | 7 |
| | $v_1$ | = | $\phi_1(v_0)$ | = | $\cos(v_0)$ |
| | $v_2$ | = | $\phi_2(v_1, v_{-1})$ | = | $v_1 v_{-1}$ |
| | $v_3$ | = | $\phi_3(v_{-1}, v_2)$ | = | $v_{-1} + v_2$ |
| | $v_4$ | = | $\phi_4(v_3)$ | = | $\sin(v_3)$ |
| dependent | $y$ | = | $v_4$ | | |

# Code Tracing with PYADOLC

```python
import numpy; from numpy import sin,cos; import adolc
def f(x):
    return sin(x[0] + cos(x[1])*x[0])

adolc.trace_on(1)
x = adolc.adouble([3,7]);   adolc.independent(x)
y = f(x)
adolc.dependent(y); adolc.trace_off()
adolc.tape_to_latex(1,[3,7],[0.])
```

| code | op | loc | loc | loc | loc | double | double | value | value | va |
|---|---|---|---|---|---|---|---|---|---|---|
| 33 | start of tape | | | | | | | | | |
| 39 | take stock op | | | 2 | 0 | | $3.000000e+00$ | | | $n$ |
| 1 | assign ind | | | | 0 | | $3.000000e+00$ | | | |
| 1 | assign ind | | | | 1 | | $7.000000e+00$ | | | |
| 20 | cos op | | 1 | 3 | 2 | | | | $7.000000e+00$ | $6.569866e-$ |
| 15 | mult a a | | 2 | 0 | 3 | | | | $7.539023e-01$ | $3.000000e+$ |
| 11 | plus a a | | 0 | 3 | 4 | | | | $3.000000e+00$ | $2.261707e+$ |
| 21 | sin op | | 4 | 6 | 5 | | | | $5.261707e+00$ | $5.221055e-$ |
| 2 | assign dep | | | | 5 | | | | | |
| 0 | death not | | | 0 | 6 | | | | | $-8.528809e-$ |
| 32 | end of tape | | | | | | | | | |

All AD tools (implicitly) work on the computational trace.

# PART I:

## The Forward Mode of AD: **Taylor Polynomial Arithmetic**

## Algorithmic Differentiation $\leftrightarrow$ Taylor Polynomial Arithmetic

- Basic Observation: Let $f : \mathbb{R}^N \rightarrow \mathbb{R}$, then

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x + e_n t)\Big|_{t=0} \quad = \quad (\nabla_x f(x))^T \cdot e_n = \frac{\partial f}{\partial x_n} \ ,$$

  where $e_n$ is the $n$'th cartesian basis vector.
- complete gradient can be obtained by taking $e_1, e_2, \ldots, e_N$
- for notational brevity: introduction of the *seed matrix*

$$\begin{aligned} (\nabla_x f(x))^T \cdot S \quad &= \quad \frac{\mathrm{d}}{\mathrm{d}t} f(x + St)\Big|_{t=0} \\ \text{e.g.} \quad S \quad &= \quad \begin{pmatrix} 1 & \\ & 1 \end{pmatrix} \end{aligned}$$

- $S \in \mathbb{R}^{N \times P}$ doesn't have to be square and not necessarily be the identity matrix
- "using" a seed matrix also leads computationally more efficient algorithms (vectorized AD, e.g. `adolc.hov_forward`)

## Univariate Taylor Polynomial Arithmetic (UTP)

- Notation for UTPs:

$$[x]_D \quad := \quad [x_0, x_1, \ldots, x_{D-1}] = \sum_{d=0}^{D-1} x_d T^d \in \mathbb{R}[T]/(T^D) \ ,$$

  - $T$ is an *indeterminate*, i.e. a formal parameter
  - $x_d \in \mathbb{R}$ is called *Taylor coefficient*
  - a UTP is defined by the $D$ coefficients $x_0, \ldots, x_{D-1}$
- Definition of Functions on UTPs:

$$\begin{aligned} E_D(f) : \mathbb{R}[T]/(T^D) \quad &\to \quad \mathbb{R}[T]/(T^D) \\ [x]_D \quad &\mapsto \quad [y]_D := \sum_{d=0} \underbrace{\frac{1}{d!} \frac{\mathrm{d}^d}{\mathrm{d}t^d} f(\sum_{k=0}^{D-1} x_d t^d) \Big|_{t=0}}_{\equiv y_d} T^d \ , \end{aligned}$$

  where $f : \mathbb{R} \to \mathbb{R}$, $y = f(x)$
- Required: algorithms to compute $y_d$ for the elem. funcs. $+, -, *, /, \ldots$

## UTP of the Multiplication and Interpretation of UTP

- goal: compute $[z]_D = [x]_D [y]_D$

$$\begin{aligned} \sum_{d=0}^{D-1} z_d T^d &= \sum_{d=0}^{D-1} x_d T^d \sum_{d=0}^{D-1} y_d T^d \\ &= \sum_{d=0}^{D-1} \underbrace{\Big(\sum_{k=0}^{d} x_{d-k} y_k\Big)}_{=z_d} T^d + \mathcal{O}(T^D) \end{aligned}$$

- Interpretation: The coefficients $y_d$ of $[y]_D = E_D(f)([x]_D)$ satisfy

$$\begin{aligned} y_d &= \frac{\mathrm{d}^d f}{\mathrm{d}^d t}(x_0 + 1t)\Big|_{t=0} \\ &= \frac{\mathrm{d}^d f}{\mathrm{d}^d x}(x_0) \end{aligned}$$

if $[x]_D = [x_0, 1, 0, \ldots, 0]$

- i.e. one gets **higher-order derivatives** by UTP arithmetic.

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

## Live Example: Gradient Evaluation using TAYLORPOLY

- Function:

$$
\begin{aligned}
f : \mathbb{R}^2 &\to \mathbb{R} \\
x &\mapsto y = f(x) = \sin(x_1 + \cos(x_2)x_1)
\end{aligned}
$$

- Compute Gradient$\nabla_x f(3, 7)$
- seed matrix $S = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

```
import numpy; from numpy import sin,cos; from taylorpoly import UTPS

def f(x):
    return sin(x[0] + cos(x[1])*x[0]) + x[1]*x[0]

x = [UTPS([3,1,0],P=2), UTPS([7,0,1],P=2)]
y = f(x)

print 'normal function evaluation y_0 = f(x_0) = ', y.data[0]
print 'gradient evaluation g(x_0) = ', y.data[1:]
```

## Higher-Order Mixed Partial Derivatives by UTP Arithmetic

- Always possible to compute higher-order mixed partial derivatives by evaluation/interpolation[2]
- Example: Hessian $H = [[f_{xx}, f_{xy}], [f_{yx}, f_{yy}]]$

$$\begin{aligned}
\langle s_1 | H | s_2 \rangle &= \frac{1}{2} \left[ \langle s_1 | H | s_2 \rangle + \langle s_2 | H | s_1 \rangle \right] \\
&= \frac{1}{2} \left[ \langle s_1 + s_2 - s_2 | H | s_2 \rangle + \langle s_2 + s_1 - s_1 | H | s_1 \rangle \right] \\
&= \frac{1}{2} \left[ \langle s_1 + s_2 | H | s_1 + s_2 \rangle - \langle s_1 | H | s_1 \rangle - \langle s_2 | H | s_2 \rangle \right] \ .
\end{aligned}$$

$$\begin{aligned}
s_1^T \nabla^2 f(x) s_2 &= \left. \frac{\partial^2 f(x + t_1 s_1 + t_2 s_2)}{\partial t_1 \partial t_2} \right|_{t_1 = t_2 = 0} \\
&= \frac{1}{2} \left[ \left. \frac{\partial^2 f(x + t(s_1 + s_2))}{\partial t^2} \right|_{t=0} - \frac{\partial^2 f(x + s_1 t)}{\partial t^2} - \frac{\partial^2 f(x + s_2 t)}{\partial t^2} \right] \ .
\end{aligned}$$

- to get $H_{ij}$ use $s_1 = e_i$ and $s_2 = e_j$ cartesian basis vectors

[2] Griewank et al., **Evaluating Higher Derivative Tensors by Forward Propagation of Univariate Taylor Series**, Mathematics of Computation, 2000

# Live Example: Hessian Evaluation using TAYLORPOLY

- $f : \mathbb{R}^2 \to \mathbb{R}$, use seed matrix $S = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$ and propagate

  $$[x]_3 = \begin{pmatrix} 3 \\ 7 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} T$$

```
from taylorpoly import UTPS
def f_fcn(x):
    return sin(x[0] + cos(x[1])*x[0])

S = array([[1,0,1],[0,1,1]], dtype=float)
P = S.shape[1]
print 'seed matrix with P = %d directions S = \n'%P, S
x1 = UTPS(zeros(1+2*P), P = P)
x2 = UTPS(zeros(1+2*P), P = P)
x1.data[0] = 3; x1.data[1::2] = S[0,:]
x2.data[0] = 7; x2.data[1::2] = S[1,:]
y = f_fcn([x1,x2])
print 'x1=',x1; print 'x2=',x2; print 'y=',y
H = zeros((2,2),dtype=float)
H[0,0] = 2*y.coeff[0,2]
H[1,0] = H[0,1] = (y.coeff[2,2] - y.coeff[0,2] - y.coeff[1,2])
H[1,1] = 2*y.coeff[1,2]
```

# PART II:
## The Reverse Mode of AD

# The Reverse Mode by Hand:

- Recall: $y = f(x) = \sin(x_1 + \cos(x_2)x_1)$

| | | | | | | |
|---|---|---|---|---|---|---|
| independent | $v_{-1}$ | = | $x_1$ | = | 3 |
| independent | $v_0$ | = | $x_2$ | = | 7 |
| | $v_1$ | = | $\phi_1(v_0)$ | = | $\cos(v_0)$ |
| | $v_2$ | = | $\phi_2(v_1, v_{-1})$ | = | $v_1 v_{-1}$ |
| | $v_3$ | = | $\phi_3(v_{-1}, v_2)$ | = | $v_{-1} + v_2$ |
| | $v_4$ | = | $\phi_4(v_3)$ | = | $\sin(v_3)$ |
| dependent | $y$ | = | $v_4$ | | |

- Reverse Mode by Hand: Successive Pullbacks

$$
\begin{aligned}
\mathrm{d}y &= \mathrm{d}\phi_4(v_3) = \left.\frac{\partial \phi_4(z)}{\partial z}\right|_{z=v_3} \mathrm{d}v_3 = \underbrace{\cos(v_3)}_{=\bar{v}_3} \mathrm{d}v_3 \\
&= \bar{v}_3 \mathrm{d}\phi_3(v_{-1}, v_2) = \underbrace{\bar{v}_3}_{=\bar{v}_{-1}} \mathrm{d}v_{-1} + \underbrace{\bar{v}_3}_{=\bar{v}_2} \mathrm{d}v_2 \\
&= \underbrace{(\bar{v}_{-1} + \bar{v}_2 v_1)}_{=\bar{v}_{-1}} \mathrm{d}v_{-1} + \underbrace{\bar{v}_2 v_{-1}}_{=\bar{v}_1} \mathrm{d}v_1 \\
&= \bar{v}_{-1} \mathrm{d}v_{-1} + \underbrace{(-\bar{v}_1 \sin(v_0))}_{=\bar{v}_0} \mathrm{d}v_0
\end{aligned}
$$

- Interpretation: $\bar{v}_{-1} \equiv \frac{\mathrm{d}f}{\mathrm{d}x_1}$ and $\bar{v}_0 \equiv \frac{\mathrm{d}f}{\mathrm{d}x_2}$
- Need to **store** $v_0, v_1, v_3, v_4$ for the reverse mode!

# Live Example: Semi-Automatic Reverse Mode

```
import numpy; from numpy import sin,cos; from taylorpoly import UTPS
x1 = UTPS([3,1,0],P=2); x2 = UTPS([7,0,1],P=2)
# forward mode
vm1 = x1
v0  = x2
v1 = cos(v0)
v2 = v1 * vm1
v3 = vm1 + v2
v4 = sin(v3)
y  = v4
# reverse mode
v4bar = UTPS([0,0,0],P=2);  v3bar = UTPS([0,0,0],P=2)
v2bar = UTPS([0,0,0],P=2);  v1bar = UTPS([0,0,0],P=2)
v0bar = UTPS([0,0,0],P=2); vm1bar = UTPS([0,0,0],P=2)
v4bar.data[0] = 1.
v3bar += v4bar*cos(v3)
vm1bar += v3bar; v2bar += v3bar
v1bar += v2bar * vm1;  vm1bar += v2bar * v1
v0bar -= v1bar * sin(v0)
g1 = y.data[1:]; g2 = numpy.array([vm1bar.data[0], v0bar.data[0]])
print 'UTPS gradient g(x_0)=', g1
print 'reverse gradient g(x_0)=', g2
print 'Hessian H(x_0)=\n',numpy.vstack([vm1bar.data[1:], v0bar.data[1:]])
```

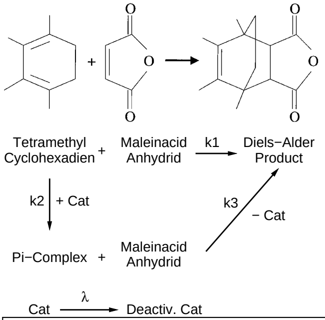
## Summary: Forward Mode and Reverse Mode

- Goal: compute Jacobian $J = \frac{\mathrm{d}F}{\mathrm{d}x}$ of function $F : \mathbb{R}^N \to \mathbb{R}^M$
- Forward Mode: $J = \frac{\mathrm{d}F}{\mathrm{d}x} \cdot S$, where $S = \mathrm{I} \in \mathbb{R}^{N \times N}$
  needs $N$ $\mathrm{OPS}(F)$ and $\mathrm{MEM}(J) \leq N\ \mathrm{MEM}(F)$
- Reverse Mode: $J = \bar{S}^T \cdot \frac{\mathrm{d}F}{\mathrm{d}x}$, where $S = \mathrm{I} \in \mathbb{R}^{M \times M}$
  $\mathrm{OPS}(J) \approx M\ \mathrm{OPS}(F)$ but $\mathrm{MEM}(J) \sim \mathrm{OPS}(F)$
- Rule of Thumb: if $5M < N$ then reverse mode *most likely* more efficient, but only **if and only if** there is enough RAM!
- Partial Solution: can use *Checkpointing* to obtain logarithmic growth in the memory.
- For $M \ll N$ reverse mode will be the method of choice.

# PART III:

## Advanced AD and Applications

# Optimum Experimental Design in Chemical Engineering




- 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

$$\dot{n}_1 = -k \cdot \frac{n_1 \cdot n_2}{m_{tot}}, \qquad n_1(0) = n_{a1}$$

$$\dot{n}_2 = -k \cdot \frac{n_1 \cdot n_2}{m_{tot}}, \qquad n_2(0) = n_{a2}$$

$$\dot{n}_3 = k \cdot \frac{n_1 \cdot n_2}{m_{tot}}, \qquad n_3(0) = 0$$

$$k = k_1 \cdot \exp\left(-\frac{E_1}{R} \cdot \left(\frac{1}{T} - \frac{1}{T_{ref}}\right)\right) + k_{kat} \cdot c_{kat} \cdot \exp\left(-\lambda \cdot t\right) \cdot \exp\left(-\frac{E_{kat}}{R} \cdot \left(\frac{1}{T} - \frac{1}{T_{ref}}\right)\right)$$

$$n_4 = n_{a4} \qquad T = \vartheta + 273$$

$$m_{tot} = n_1 \cdot M_1 + n_2 \cdot M_2 + n_3 \cdot M_3 + n_4 \cdot M_4$$

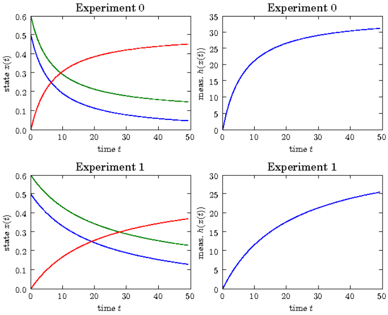
# Optimum Experimental Design in Chemical Engineering (Cont.)

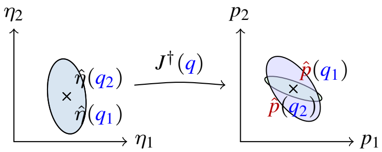
- **Dynamics**: Defined by ODE
- **Goal**: Estimate parameters $p = (k_1, k_{\text{kat}}, E_{\text{kat}})$
- **Problem**: Errors in the measurements $\eta$ result in errors in parameters $p$.
- **nonlinear regression** with additive iid normal errors

$$\eta_m = h_m(t_m, x(t_m), p, q) + \varepsilon_m \ , \quad m = 1, \ldots, N_M$$

$$\varepsilon_m \sim \mathcal{N}(0, \sigma_m^2)$$

- $\eta_m$ 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.



# Overall Objective Function

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

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

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

- **Part II: Numerical Linear Algebra**

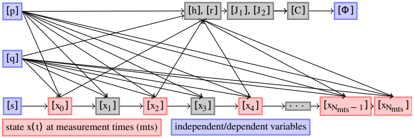
$$C(J_1, J_2) = (\mathrm{I}, 0) \begin{pmatrix} J_1^T J_1 & J_2^T \\ J_2 & 0 \end{pmatrix}^{-1} \begin{pmatrix} \mathrm{I} \\ 0 \end{pmatrix}$$

$$= \left( Q_2^T (Q_2 J_1^T J_1 Q_2^T)^{-1} Q_2 \right)$$

$$\Phi = \lambda_1(C) \quad \text{, max. eigenvalue}$$

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

- **Computational Graph**



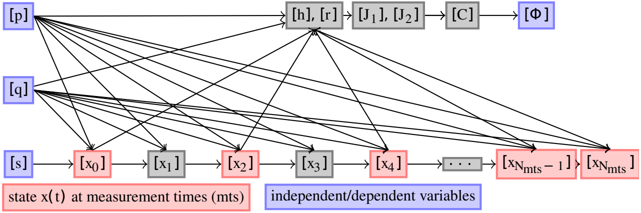

$N_{mts}$ Number measurment times, $\sigma$ std of a measurement, $q$ controls, $p$ nature given parameter, $s$ pseudo-Parameter (e.g. initial values), $u$ control functions

## Differentiation of ODE Solvers

- Explicit Euler:

$$x_{k+1} \quad = \quad x_k + h_k f(t_k, x_k, p)$$

- want $x(t)$ and $\frac{\mathrm{d}x}{\mathrm{d}p}(t)$ at $t = [t_1, t_2, \ldots, t_{N_{\mathrm{mts}}}]$
- Idea: look at computation trace of the $x_k$
- Apply Standard AD techniques + some tricks
- Approach also works for implicit schemes as needed for DAEs (DAESOL-II)



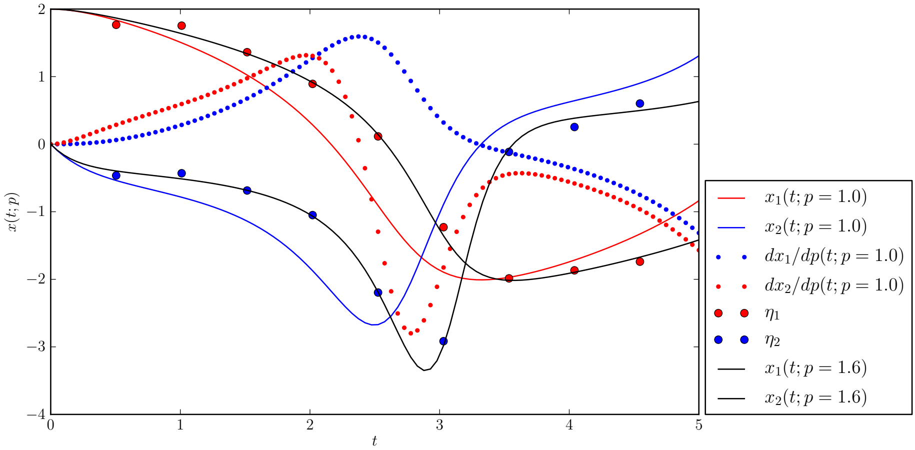
## Live Example: UTPS of Explicit Euler

```
import numpy; from numpy import sin,cos; from taylorpoly import UTPS
x = numpy.array([UTPS([1,0,0],P=2), UTPS([0,0,0],P=2)])
p = UTPS([3,1,0],P=2)
def f(x):
    return numpy.array([x[1],-p * x[0]])

ts = numpy.linspace(0,2*numpy.pi,100)
x_list = [[xi.data.copy() for xi in x]]
for nts in range(ts.size-1):
    h = ts[nts+1] - ts[nts]
    x = x + h * f(x)
    x_list.append([xi.data.copy() for xi in x])

xs = numpy.array(x_list)
import matplotlib.pyplot as pyplot
pyplot.plot(ts, xs[:,0,0], '.k-', label = r'$x(t)$')
pyplot.plot(ts, xs[:,0,1], '.r-', label = r'$x_p(t)$')
pyplot.xlabel('time $t$')
pyplot.legend(loc='best')
pyplot.grid()
pyplot.show()
```

# Application Example: DAESOL-II for Least-Squares




- DAESOL-II can already do forward/reverse UTP
- UTP in DAESOL-II is approaching maturity now

# Differentiating Implicitly Defined Function with 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**[3]: Let $F([x], [y]_D) \overset{D}{=} 0$ and $F'([x], [y]_D) \mod t^D$ invertible. Then

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

- $[X]_D \equiv [X_0, \dots, x_{D-1}] \equiv \sum_{d=0}^{D-1} x_d t^d, \quad [\Delta F]_E t^D \overset{D+E}{=} F([x], [y]_D)$
- if $E = D$ then number of correct coefficients is doubled

[3] also called Newton-Hensel lifting or Hensel lifting

# Univariate Taylor Propagation on Matrices (UTPM)

- Application of Newton's Method to defining equations
- **Defining equations** of the $QR$ decomposition:

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

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

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

- **Defining equations** of the Cholesky Decomposition

$$
\begin{aligned}
0 &\overset{D}{=} [L]_D [L]_D^T - [a]_D \\
0 &\overset{D}{=} P_D \circ [L]_D - \mathrm{I} \\
0 &\overset{D}{=} P_R \circ [L]_D \; .
\end{aligned}
$$

- etc...

# Algorithm: Forward UTPM of the Rectangular $QR$ Decomposition

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

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

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

$\Delta F = A_d - \sum_{k=1}^{d-1} Q_{d-k} R_k$
$S = -\frac{1}{2} \sum_{k=1}^{d-1} Q_{d-k}^T Q_k$
$P_L \circ X = P_L \circ (Q_0^T \Delta F R_0^{-1} - S)$
$X = P_L \circ X - (P_L \circ X)^T$
$R_d = Q_0^T \Delta F - (S + X) R_0$
$Q_d = (\Delta F - Q_0 R_d) R_0^{-1}$

**end**

## Algorithm: Reverse UTPM of the Rectangular $QR$ Decomposition

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

$$\begin{aligned}
[\bar{A}]_D &= [\bar{A}]_D + ([\bar{Q}]_D - [Q]_D [Q]_D^T [\bar{Q}]_D)[R]_D^{-T}) \\
&\quad + [Q]_D \left( [\bar{R}]_D + P_L \circ \left( [R]_D [\bar{R}]_D^T - [\bar{R}]_D [R]_D^T + [Q]_D^T [\bar{Q}]_D - [\bar{Q}]_D^T [Q]_D \right) [R]_D^{-T} \right)
\end{aligned}$$

# ALGOPY Live Example: *QR* decomposition

```
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
```

## ALGOPY Live Example: Moore-Penrose Pseudo inverse

```
import numpy; from algopy import CGraph, Function, UTPM, dot, qr, eigh, in
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,:,0] = [1,1,1,1,1]; x.data[1,0,:,0] = [1,1,1,1,1]
y.data[0,0,:,0] = [1,2,1,2,1]; y.data[1,0,:,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 \n', dot(dot(A, Apinv),A) - A
print 'naive approach: Apinv A Apinv - Apinv = 0 \n', dot(dot(Apinv, A),Ap
print 'naive approach: (Apinv A)^T - Apinv A = 0 \n', dot(Apinv, A).T - d
print 'naive approach: (A Apinv)^T - A Apinv = 0 \n', dot(A, Apinv).T - d
# 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 \n',  dot(dot(A, Apinv),A) - A
print 'QR approach: Apinv A Apinv - Apinv = 0 \n', dot(dot(Apinv, A),Apinv
print 'QR approach: (Apinv A)^T - Apinv A = 0 \n', dot(Apinv, A).T - dot(
print 'QR approach: (A Apinv)^T - A Apinv = 0 \n', dot(A, Apinv).T - dot(
```

# Algorithm: Forward UTPM of Symmetric Eigenvalue Decomposition

**input** : $[A]_D = [A_0, \dots, A_{D-1}]$, where $A_d \in \mathbb{R}^{N \times N}$ symmetric positive definite, $d = 0, \dots, D-1$
**output**: $[\tilde{\Lambda}]_D = [\tilde{\Lambda}_0, \dots, \tilde{\Lambda}_{D-1}]$, where $\Lambda_0 \in \mathbb{R}^{N \times N}$ diagonal and $\Lambda_d \in \mathbb{R}^{N \times N}$ block diagonal $d = 1, \dots, D-1$.
**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 $\mathrm{sl} = b[n_b] : b[n_b+1]$ one has $(Q_0[:, \mathrm{sl}])^T A_0 Q_0[:, \mathrm{sl}] = \lambda_{n_b} \mathrm{I}$.

$\Lambda_0, Q_0 = \mathrm{eigh}\,(A_0)$

$E_{ij} = (\Lambda_0)_{jj} - (\Lambda_0)_{ii}$
$H = P_B \circ (1/E)$

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

$S = -\frac{1}{2} \sum_{k=1}^{d-1} Q_{d-k}^T Q_k$
$K = \Delta F + \tilde{Q}_0^T A_d \tilde{Q}_0 + S\Lambda_0 + \Lambda_0 S$
$\tilde{Q}_d = Q_0 (S + H \circ K)$
$\tilde{\Lambda}_d = \bar{P}_B \circ K$

**end**

- 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[4]

- Orthonormal Matrix:

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

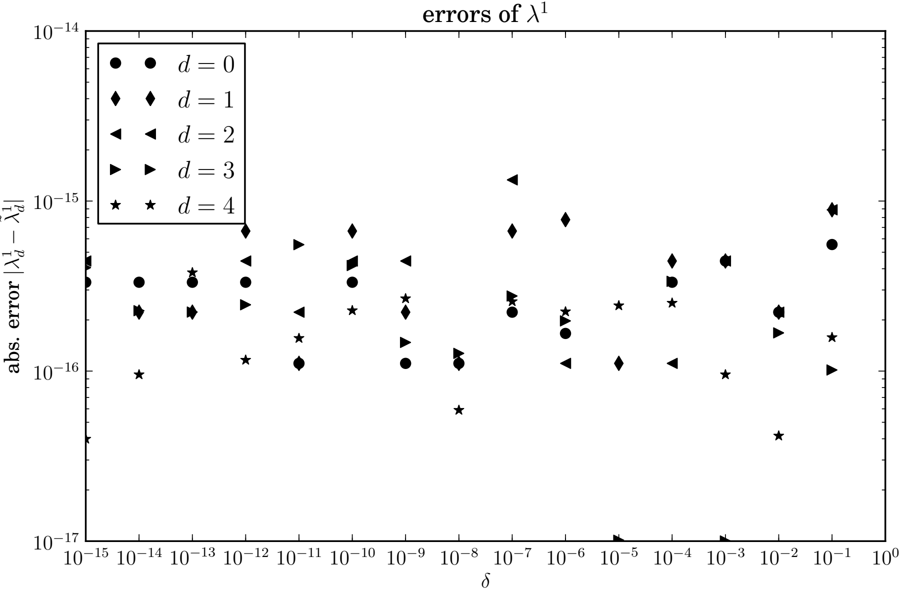
  where $x \equiv x(t) := 1 + t$.
- constant $\delta = 0$ means repeated eigenvalues, $\delta > 0$ distinct but close
- In Taylor arithmetic one obtains

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

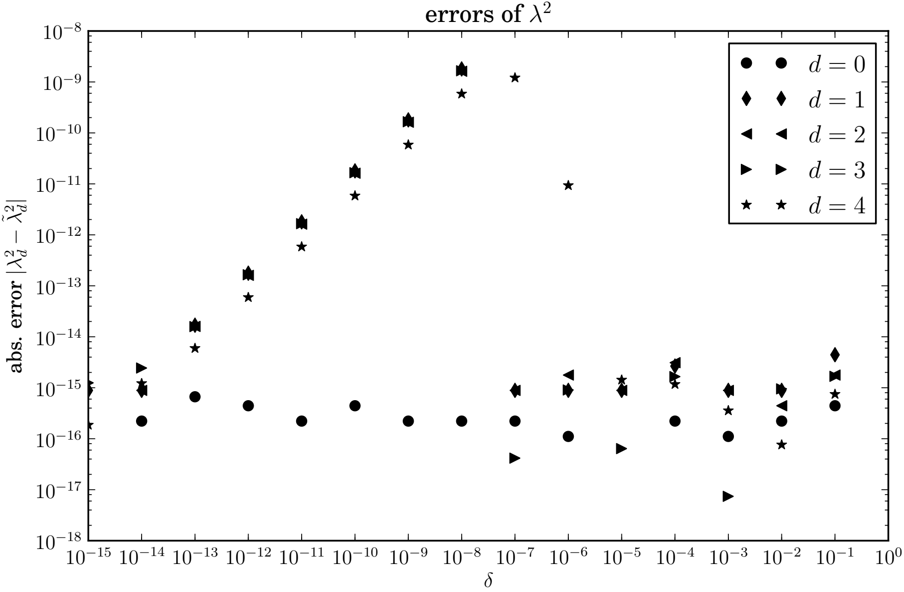
- Define $A(t) = Q(t)\Lambda(t)Q(t)$ and try to reconstruct $\Lambda(t)$ and $Q(t)$.

[4]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

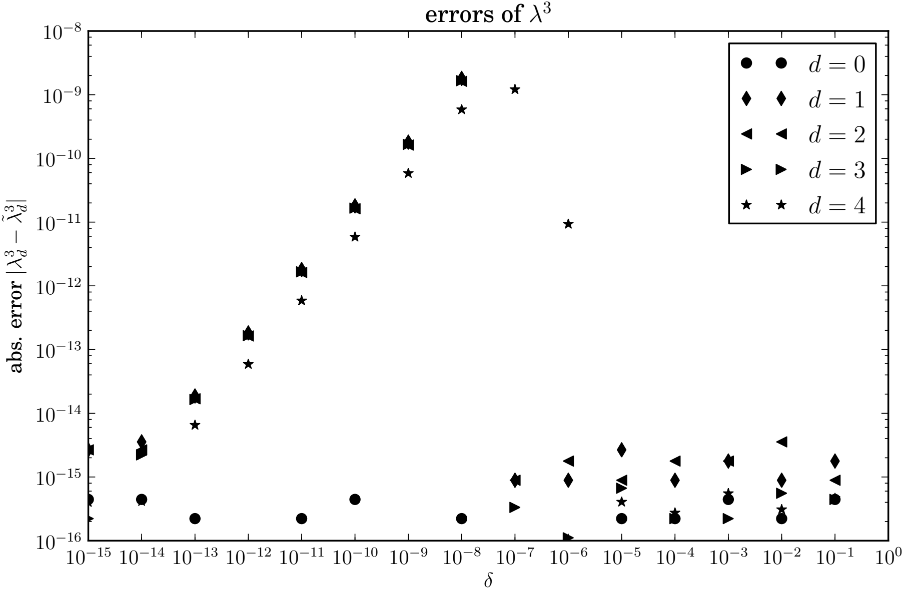
# Test Example for the Symmetric Eigenvalue Decomposition (cont.)



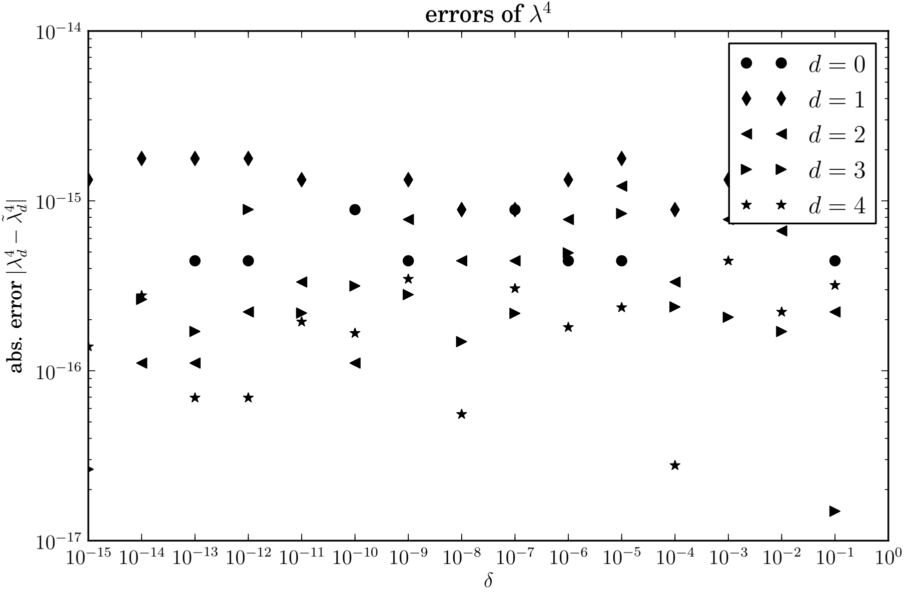
# Test Example for the Symmetric Eigenvalue Decomposition (cont.)



# Test Example for the Symmetric Eigenvalue Decomposition (cont.)



# Test Example for the Symmetric Eigenvalue Decomposition (cont.)



## The $E$-Criterion of the Opt. Exp. Design Problem

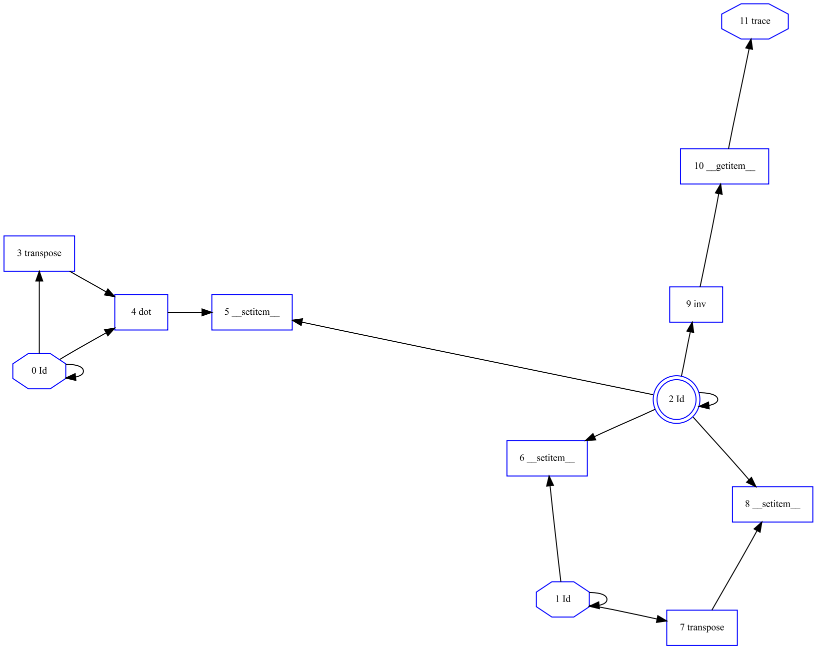
- Compute $\nabla_q^2 \mathrm{eigh}(C(q))$, where

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

```
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

## Some Software for Forward/Reverse UTP

| Name | Description | Status | LOC |
|---|---|---|---|
| **algopy** | forward/reverse UTPM in Python<br>www.github.com/b45ch1/algopy | alpha | 10388 |
| **pysolvind** | Python Bindings to SolvIND/DAESOL-II | alpha | 9743 |
| **pyadolc** | Python Bindings to ADOL-C (C++)<br>www.github.com/b45ch1/pyadolc | stable | 6895 |
| **pycppad** | Python Bindings to CppAD (C++ )<br>www.github.com/b45ch1/pycppad | stable | 1334 |
| **taylorpoly** | ANSI-C with Python bindings<br>www.github.com/b45ch1/taylorpoly | alpha | 9276 |

- Summary:
  - Have shown many aspects of AD, in particular Univariate Taylor Polynomial arithmetic and the Reverse Mode
  - There are many useful tools in Python that ease prototyping
  - 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