Automatic Differentiation in Linear Algebra and Statistics

Abstract

Automatic Differentiation (AD) refers to a number of techniques for evaluating computational sensitivities. These techniques are very different from Symbolic Differentiation and Finite Differencing and almost certainly should be used more broadly. I will talk about two very different versions of AD : the first augments computer code to generate derivative values while the second applies standard calculus tools in unusual ways. Historically the second approach originated (1950s) in Maximum Likelihood Estimation and error estimates in Numercal Linear Algebra. Recently both approaches have been used in economic forecasting and options pricing. The presentation should be accessible to anyone with minimal linear algebra background. Specifically I intend to exhibit derivatives of LU, QR, and various spectral decompositions.

References:Selected

1. Code Modification. Uwe Naumann, The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation, SIAM, 2012.
2. Linear Algebra. Mike, Giles. “Collected matrix derivative results for forward and reverse mode algorithmic differentiation”, pp.35-44 in Advances in Automatic Differentiation, Springer, 2008.
3. Linear Algebra. G. W. Stewart, Error and Perturbation Bounds for Subspaces Associated with Certain Eigenvalue Problems, SIAM Rev., 15(4), 727–764, 1973.
4. Linear Algebra and Economic Forecasting. Gary Anderson, “A procedure for differentiating perfect-foresight-model reduced-from coefficients”, Journal of Economic Dynamics and Control Volume 11, Issue 4, December 1987.
5. Linear Algebra and Economic Forecasting. H. Bastani and L. Guerrieri, “On the application of automatic differentiation to the likelihood function for dynamic general equilibrium models”, Lecture Notes in Computational Science and Engineering Volume 64, 303-313, 2008.
6. Linear Algebra and Statistics. F. R. De Hoog, R. S. Anderssen, and M. A. Lukas, Differentiation of Matrix Functionals Using Triangular Factorizations, Mathematics of Computation, V80, #275, pp1585-1600, 2011.

AD: Code Modification

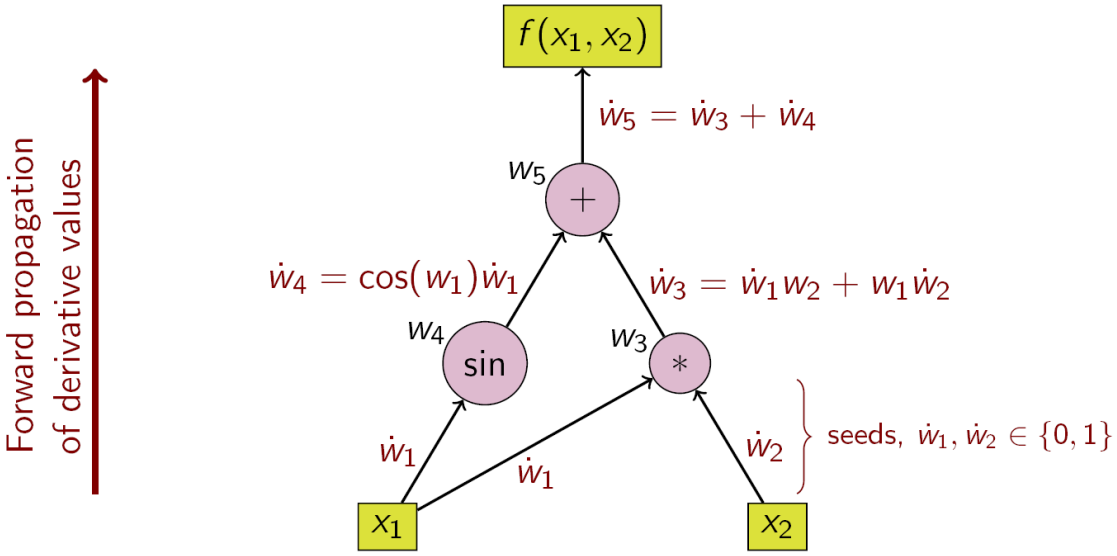
- All computer code is simply a lot of arithmetic (+, *, ÷) and logical operations.
- Freezing the logical operations leaves the arithmetic which we learnt how to differentiate in Calc 1.
- This is what Algorithmic Differentiation does!
- Operator overloading is the easiest (but in practice least computationaly efficient) way to understand this.

AD: Operator Overloading

- $a + b \longrightarrow \{a + b, \Delta a + \Delta b\}$
- $a - b \longrightarrow \{a - b, \Delta a - \Delta b\}$
- $a * b \longrightarrow \{a * b, \Delta a * b + a * \Delta b\}$
- $a / b \longrightarrow \{a / b, (\Delta a * b - a * \Delta b) / b^2\}$
- $\sqrt{a} \longrightarrow \left\{\sqrt{a}, \Delta a / \left(2\sqrt{a}\right)\right\}$
- $\text{Sin}[a] \longrightarrow \{\text{Sin}[a], \text{Cos}[a] * \Delta a\}$
- …

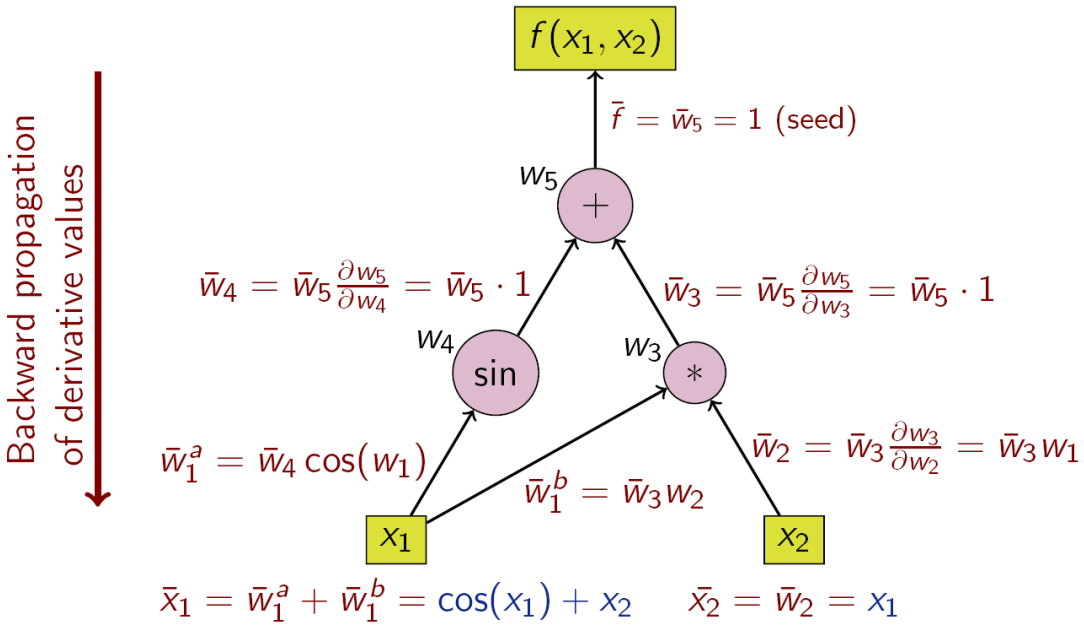
AD: Forward Mode

From wiki. Standard notation "$\dot{w}$" means forward mode derivative.



AD: Reverse Mode

From wiki. Standard notation "$\bar{w}$" means backward mode derivative



AD: Multiple Parameters

- Abstractly computational functions are maps
  $$f : \mathbb{R}^m \longrightarrow \mathbb{R}^n$$
  realized by explicit code.
- Calc III talks about directional derivatives and gradients.
  - Directional Derivatives $\longleftrightarrow$ Forward Mode AD
  - Gradients $\longleftrightarrow$ Reverse Mode AD
- Directional Derivatives and Gradients are different.

Directional Derivatives

Directional derivatives answer the question.
How does the output change if I make small changes to some inputs?

- Directional Derivatives:
  - Given direction $\vec{u} \in \mathbb{R}^m$ define $g[\alpha] = f\left[\vec{x} + \alpha\, \vec{u}\right]$ then $\partial_u f = g'[0]$
  - Given multiple directions $\left\{\vec{u}_1, \ldots, \vec{u}_r\right\} \subset \mathbb{R}^m$ define $g[\vec{\alpha}] = f\left[\vec{x} + \alpha_1\, \vec{u}_1 + \ldots \alpha_r\, \vec{u}_1\right]$ then $\partial_{u_i} f = \partial_i [\vec{0}_r]$
  - Higher directional derivatives are defined similarly

Gradients

Gradients answer the question.
What is the minimal input change to create a small output change?

- Gradient
  - $\nabla f : \mathbb{R}^m \longrightarrow \mathbb{R}^{n \times m}$
  - In terms of the unit vector $u = \frac{\nabla f}{\| \nabla f \|}$

    $f[p_0 + \alpha u] \approx f[p_0] + \alpha \| \nabla f \|$
  - Higher gradients are defined similarly.

h

```
Show[CPic, Epilog → {RGBColor[1, 0, 0], Point[{1, 1}], Arrow[{{1, 1},
    {1 + 0.5 Df[{1, 1}] / (Abs[Df[{1, 1}]] Norm[Df[{1, 1}]]), 1 + 0.5 Df[{1, 1}] / (Abs[Df[{1, 1}]] Norm[Df[{1, 1}]])}}]}]
```

Show::gtype : Symbolis not a type of graphics ≫

Gradients: Pictures

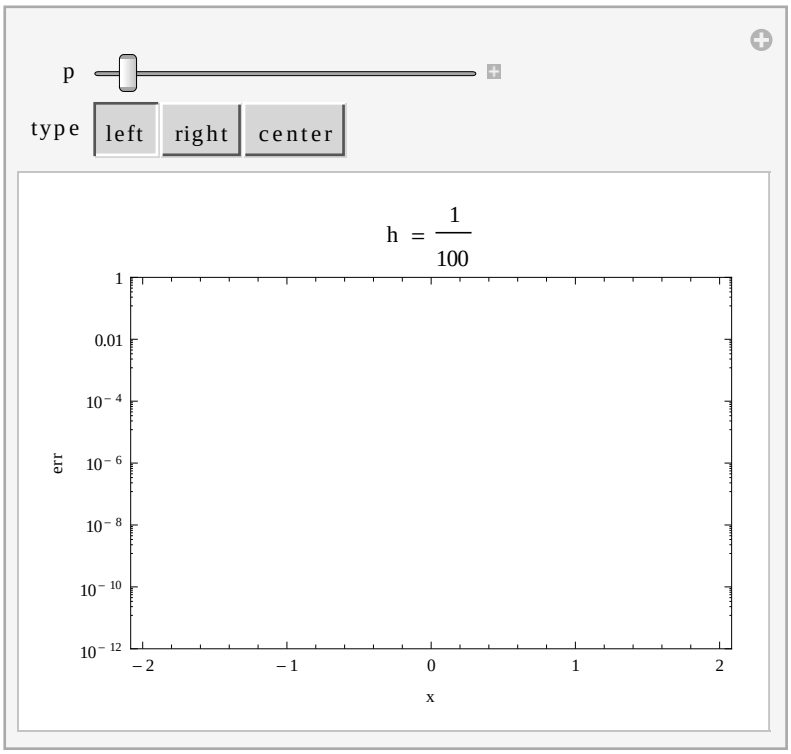
h

```
Show[CPic, Epilog → {RGBColor[1, 0, 0], Point[{0.5, 0.5}],
    Arrow[{{0.5, 0.5}, {0.5 + 0.5 Df[{0.5, 0.5}] / (Abs[Df[{0.5, 0.5}]] Norm[Df[{0.5, 0.5}]]),
      0.5 + 0.5 Df[{0.5, 0.5}] / (Abs[Df[{0.5, 0.5}]] Norm[Df[{0.5, 0.5}]])}}]}]
```

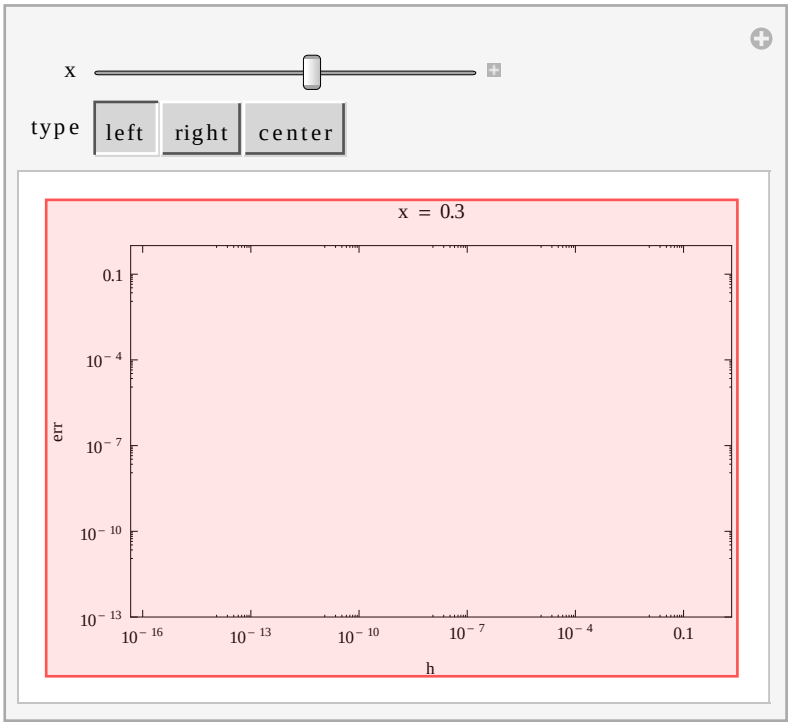
Show::gtype : Symbolis not a type of graphics ≫

Why is AD better than FD

One word: Cancellation



How to Pick “h” in FD



AD Tools

- Tapenade: http:/ / www-sop.inria.fr/ tropics/ tapenade.html
  - Code rewrite, Open Source, Gov sponsored, web interface, inactive since 2012.
- DCC: http:/ / www.stce.rwth-aachen.de/ software/ dcc.html
  - Code rewrite and overloading, Open Source, Associated with Naumann text, active.
- SACADO and Stokhos: http:/ / trilinos.sandia.gov/
  - Code rewrite and overloading, C++, Open Source, Gov sponsored, installed in MathLab (part of trilinos), used MA4620 Fall 2012 for gas dynamics FEA.

AD: Forward vs Backwards

- Forward
  - Conceptually simple
  - Easily SIMD parallelized at various levels
    - Simple code
    - Simple memory access
  - Can compute selected directional derivatives
    - Practically, each directional derivative is proportional to 1-2 function evaluations
- Reverse
  - Theoretically good complexity
  - Complex memory access
  - All or nothing
    - Theoretically, entire gradient is proportional to 1-10 function evaluations

AD: Forward vs Backwards Cont

- $f : \mathbb{R}^m \longrightarrow \mathbb{R}^n$
  - Forward computation of ∇f m sweeps (one for each input)
  - Reverse computation of ∇f n sweeps (one for each output)
  - Forward can readily exploit sparsity

AD for Matrix Computations

- The basic building block for most technical computations are BLAS and LINPACKish libary calls.
  - The logic in such algorithms is simply (implicit or explicit) pivoting.
  - Every algorithm in this class has "essentially" explicit AD algorithms for derivatives of any order.
    - The pivoting strategy is determined by the base algorithm
  - Extremely efficient code on vast range of architectures
  - The algorithmic output is completely determined.
    - The theoretical ouput is not completely determined.
    - Basis choice for two (or higher) dimensional subspaces is a common example
- Aim is to create effective implementations of these algorithms in BLAS and LINPACKish code
  - Always a triangular formulation
  - Non-unique theoretical output
    - Potentially consistent but rank deficient linear systems

Matrix Arithmetic AD

It is pretty clear how to differentiate Matrix Arithmetic.

- $C = A + B \longrightarrow \text{dA} = \text{dA} + \text{dB}$
- $C = A\ B \longrightarrow \text{dC} = \text{dA}\ B + A\ \text{dB}$
- $C = f(A, B) \longrightarrow \text{dC} = \partial_A f\ \text{dA} + \partial_B f\ \text{dB}$

Forward and backwards chain rules put these pieces together.

Chain Rule: Forward

Consider a procedure $F : x \longrightarrow y$ of the form

$$x \;\rightarrow\; C = f(A, B) \;\rightarrow\; y$$

Forward mode (written with $\overset{\circ}{C}$ or Cd) computes sensitivites wrt x starting at x. At the middle step:

- Input is A and B along with Ad and Bd
- $C = f(A, B)$ and the sensitivies $\partial_A f$ and $\partial_B f$ are computed
- The infinimitesimal expresion gives
  - Cd dx $= \partial_A f$ Ad dx $+ \partial_B f$ Bd dx
  - So... Cd $= \partial_A f$ Ad $+ \partial_B f$ Bd

The backwards chain rule may be less clear.

Chain Rule: Backward

Consider a procedure $F : x \longrightarrow y$ of the form

$$x \;\rightarrow\; C = f(A, B) \;\rightarrow\; y$$

Reverse mode (written $\overline{C}$ or Cb) computes sensitivites wrt y starting at y. At the middle step:

- Input is C and Cb
- A and B have been retrieved from a "tape".
  - In practice this is the big drawback for reverse mode.
- The sensitivies $\partial_A f$ and $\partial_B f$ are computed
- The definition of Cb is that Cb : dC = dy
- Need to compute Ab and Bb.
- Infinimitesimal gives
  - Cb : dC = Cb : ($\partial_A f$ dA + $\partial_B f$ dB)
  - Cb : dC = $\partial_A f^T$ Cb : dA + $\partial_B f^T$ Cb : dB)
  - So Ab = $\partial_A f^T$ Cb and Bd = $\partial_B f^T$ Cb

SVD:

$A \to \{U, \Sigma, V\}$

- U and V are orthogonal $U\ U^T = \mathrm{Id}$ and $V\ V^T = \mathrm{Id}$
  Σ is "diagonal"
- $A = U\ \Sigma\ V^T$ so $\mathrm{Ad} = \mathrm{Ud}\ \Sigma\ V^T + U\ \Sigma\ \mathrm{Vd}^T + U\ \Sigma\mathrm{d}\ V^T$
- $U^T\ U = \mathrm{Id}$ so $\mathrm{Ud}^T\ U + U^T\ \mathrm{Ud} = 0$
  $\mathrm{Pu} = U^T\ \mathrm{Ud}$ and $\mathrm{Pv} = V^T\ \mathrm{Vd}$ are skew.
- $U^T\ \mathrm{Ad}\ V = \mathrm{Pu}\ \Sigma - \Sigma\ \mathrm{Pv} + \Sigma\mathrm{d}$
  - $\mathrm{Pu}\ \Sigma$ and $\Sigma\ \mathrm{Pv}$ have zero diagonals
    $\Longrightarrow \Sigma\mathrm{d} = \mathrm{Diagonal}\left[U^T\ \mathrm{Ad}\ V\right]$
  - Transposing gives a second equation
    $V^T\ \mathrm{Ad}^T\ U = -\Sigma\ \mathrm{Pu} + \mathrm{Pv}\ \Sigma + \Sigma\mathrm{d}$
    $\Longrightarrow \mathrm{Pu} = \Sigma^{-1}\left(\mathrm{Pv}\ \Sigma + \Sigma\mathrm{d} - V^T\ \mathrm{Ad}^T\ U\right)$

SVD:
$A \to \{U,\Sigma,V\}$ (cont)

- $U^T \ \text{Ad}\ V = \Sigma^{-1}\left(\text{Pv}\ \Sigma\ +\ \Sigma\text{d} - V^T\ \text{Ad}^T\ U\right)\ \Sigma\ - \Sigma\ \text{Pv} + \Sigma\text{d}$
  $\Longrightarrow \Sigma\ U^T\ \text{Ad}\ V + V^T\ \text{Ad}^T\ U\ \Sigma = \text{Pv}\ \Sigma^2\ - \Sigma^2\ \text{Pv} + 2\ \Sigma\ \Sigma\text{d}$
  $\Longrightarrow \text{Pv} = F \circ \left(\Sigma\ U^T\ \text{Ad}\ V + V^T\ \text{Ad}^T\ U\ \Sigma\right)$
  where $F_{i,i} = 0$ and $F_{i,j} = \left(s_j{}^2 - s_i{}^2\right)^{-1}$ for $i \neq j$ and $\circ$ is the Hadamard product.
- Similar expression hold for Pu and for non-square matrices.
- Note: The AD derivative always exists (it can be computed) so the problem is non-uniqueness rather than non-existence.

LUdecomposition:

$A \rightarrow \{L,U\}$

- U is Upper triangular
  L is Lower triangular with 1s on the diagonal
- $A = L\ U$ so $\text{Ad} = \text{Ld}\ U + L\ \text{Ud}$ m^2 entries or equations
- Ud upper triangular $m \times (m+1)/2$ DOF
  Ld strictly lower triangular $m \times (m-1)/2$ DOF
- $L^{-1}\ \text{Ad}\ U^{-1} = L^{-1}\ \text{Ld} + \text{Ud}\ U^{-1}$ If inverses exist
  $\text{Ld} = L\left(L^{-1}\ \text{Ad}\ U^{-1}\right)_{\triangledown}$ and $\text{Ud} = \left(L^{-1}\ \text{Ad}\ U^{-1}\right)_{\blacktriangle} U$
  Note: Inverses are triangular matrices

QRdecomposition:
$A \to \{Q,R\}$

- $A = Q\ R$
  R is Upper triangular
  Q is orth i.e. $Q^T\ Q = \mathrm{Id}$
- $\mathrm{Ad} = \mathrm{Qd}\ R + Q\ R\mathrm{d}$ $\quad$ m^2 entries or equations
  $\mathrm{Qd}^T\ Q + Q^T\ \mathrm{Qd} = 0$ $\quad$ $P = Q^T\ \mathrm{Qd}$ is skew
- Rd upper triangular $\quad$ $m \times (m+1)/2$ DOF
  $P = -P^T$ $\quad$ $m \times (m-1)/2$ DOF
- $Q^T\ \mathrm{Ad}\ R^{-1} = Q^T\ \mathrm{Qd} + \mathrm{Rd}\ R^{-1}$ $\quad$ If inverses exist
  $\mathrm{Qd} = Q\ \left(Q^T\ \mathrm{Ad}\ R^{-1}\right)_{\mathrm{sk}}$ and $\mathrm{Rd} = \left(Q^T\ \mathrm{Ad}\ R^{-1}\right)_{\blacktriangle}\ R$
  Note: Inverse is triangular.

Schur Decomposition:
$A \rightarrow \{Q,T\}$

- $A = Q\ T\ Q^T$
  T is pseudo upper triangular
  Q is orth i.e. $Q^T\ Q$ = Id
- Ad = Qd $T\ Q^T + Q\ T$ $\text{Qd}^T + Q$ Td Q^T m^2 entries or equations
  Q^T Ad $Q = P\ T - T\ P +$ Td P skew
- Td $m \times (m+1)/2$ DOF
  $P = -P^T$ $m \times (m-1)/2$ DOF
- Triangular* solve of a very sparse system.

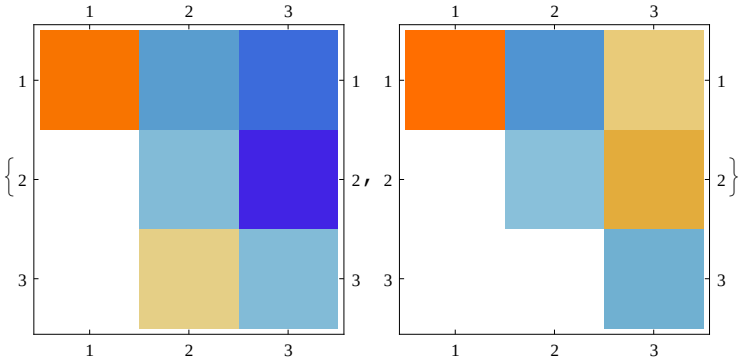
*Except for a some 2×2 diagonal sub blocks.

Schur Decomposition: Pictures

```
m = 3;
A = RandomReal[{-1, 1}, {m, m}];
{Q, T} = SchurDecomposition[A];
{QI, TI} = SchurDecomposition[A, RealBlockDiagonalForm → False];
Map[MatrixForm, {T, TI}]
Map[MatrixPlot, {T, TI}]
```

$$\left\{\begin{pmatrix} 0.83372 & -0.139854 & -0.347039 \\ 0. & -0.0275551 & -0.483611 \\ 0. & 0.0899205 & -0.0275551 \end{pmatrix},\right.$$

$$\left.\begin{pmatrix} 0.83372 - 8.32667 \times 10^{-17}\, i & -0.160909 + 0.0973818\, i & 0.25414 + 0.200082\, i \\ 0. + 0.\, i & -0.0275551 + 0.208534\, i & 0.332712 + 0.210465\, i \\ 0. + 0.\, i & 0. + 0.\, i & -0.0275551 - 0.208534\, i \end{pmatrix}\right\}$$



Schur Decomposition: Recursive Solve

- Given B and $T_{\blacktriangle}$
  Find $\text{Td}_{\blacktriangle}$ and P_{sk} satisfying $B = P\ T\ -\ T\ P\ +\ \text{Td}$
- Find the splitting point nearest the middle

$$\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} P_{11} & -P_{21}^{T} \\ P_{21} & P_{22} \end{pmatrix} \cdot \begin{pmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{pmatrix} - \begin{pmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{pmatrix} \cdot \begin{pmatrix} P_{11} & -P_{21}^{T} \\ P_{21} & P_{22} \end{pmatrix} + \begin{pmatrix} \text{Td}_{11} & \text{Td}_{12} \\ 0 & \text{Td}_{22} \end{pmatrix}$$

  - $B_{21} = P_{21}\ T_{11} - T_{22}\ P_{21} + 0$ Triangular Lyapunov/ Sylvester Eq for P21
- Recursively split subblocks.
- Solve the remaining 2×2 diagonal blocks
- Finally compute Td using original equation.

Schur Decomposition: Non-RecursiveSolve

- Given B and $T_{\blacktriangle}$
  Find $\mathrm{Td}_{\blacktriangle}$ and P_{sk} satisfying $B = P\ T\ -\ T\ P\ +\ \mathrm{Td}$
- If $T[[2,\ 1]] = 0$

$$\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} 0 & -P_{21}^{T} \\ P_{21} & P_{22} \end{pmatrix} . \begin{pmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{pmatrix} - \begin{pmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{pmatrix} . \begin{pmatrix} 0 & -P_{21}^{T} \\ P_{21} & P_{22} \end{pmatrix} + \begin{pmatrix} \mathrm{Td}_{11} & \mathrm{Td}_{12} \\ 0 & \mathrm{Td}_{22} \end{pmatrix}$$

  - $B_{21} = P_{21}\ T_{11} - T_{22}\ P_{21} + 0$ Simple division for (n-1)×1 P21
    $B_{11} = -T_{12}\ P_{21} + \mathrm{Td}_{11}$ Determines Td_1 directly
- If $T[[2,\ 1]] \neq 0$

$$\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} 0 & -P_{21}^{T} \\ P_{21} & P_{22} \end{pmatrix} . \begin{pmatrix} \begin{pmatrix} t_{11} & t_{21} \\ t_{12} & t_{11} \end{pmatrix} & T_{12} \\ 0 & T_{22} \end{pmatrix} - \begin{pmatrix} \begin{pmatrix} t_{11} & t_{21} \\ t_{12} & t_{11} \end{pmatrix} & T_{12} \\ 0 & T_{22} \end{pmatrix} . \begin{pmatrix} 0 & -P_{21}^{T} \\ P_{21} & P_{22} \end{pmatrix} + \begin{pmatrix} \begin{pmatrix} td_{11} & td_{21} \\ td_{12} & td_{11} \end{pmatrix} & \mathrm{Td}_{12} \\ 0 & \mathrm{Td}_{22} \end{pmatrix}$$

  - Almost as direct

Schur Decomposition: Code

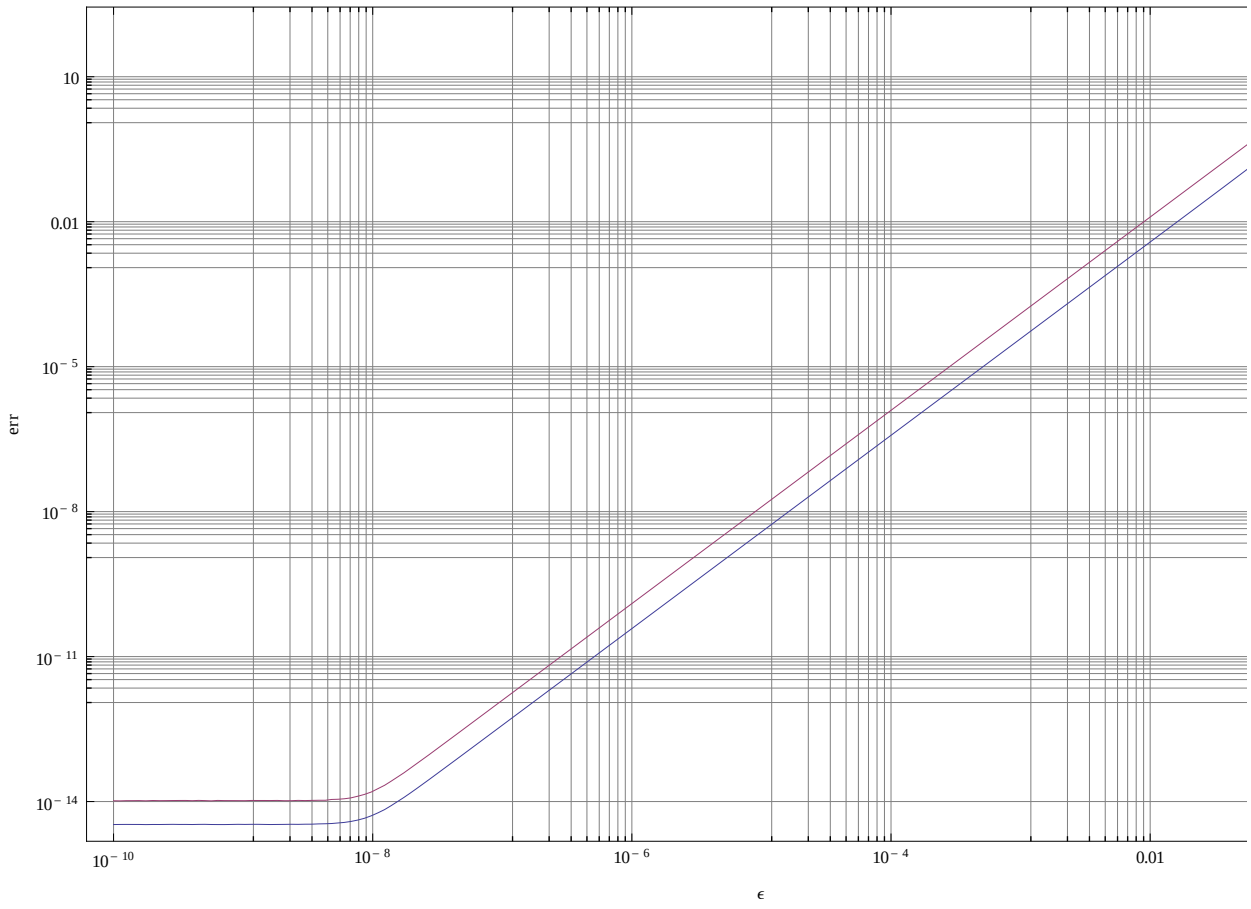
Recursive | Lyapunv Solver | Split Matrix | Split Point

```
SetAttributes[SchurADSolveForP, HoldFirst];
SchurADSolveForP[P_, m_, {T_, B_}] := Module[
  {m1, Tt, Bt, P21, n1, n2},
  m1 = SchurADSplit[T];
  (* If no zero then it is either a 2x2 block or it is done *)
  If[m1 == {}, If[Dimensions[T] == {2, 2}, P[[m + 2, m + 1]] = -1/2 (B[[2, 2]] - B[[1, 1]])/(T[[1, 2]] + T[[2, 1]])]; Return[]];
  (* Implement the splitting algorithm *)
  Tt = ADSplit[T, m1]; Bt = ADSplit[B, m1];
  P21 = LyapunovSolve[Tt[[2, 2]], -Tt[[1, 1]], -Bt[[2, 1]]];
  {n1, n2} = Dimensions[P21]; P[[m + m1 + 1 ;; m + m1 + n1, m + 1 ;; m + n2]] = P21;
  (* recurse on remaining two blocks *)
  SchurADSolveForP[P, m, {Tt[[1, 1]], Bt[[1, 1]] + Tt[[1, 2]].P21}];
  SchurADSolveForP[P, m + m1, {Tt[[2, 2]], Bt[[2, 2]] - P21.Tt[[1, 2]]}]
 ]
```

Schur Decomposition: Test for Correctness

The correct way to test this stuff is to see how well the computed derivatives approximate the original matrix.

```
Clear[m, A, Ad, Qtest, Ttest, Atest, ϵ]
m = 20;
{A, Ad} = RandomReal[{-1, 1}, {2, m, m}];
{Timing[{Q, T} = SchurDecomposition[A];],
 Timing[{Qd, Td} = SchurAD[Ad, {Q, T}];]};
Qtest[ϵ_] := Q + ϵ Qd;
Ttest[ϵ_] := T + ϵ Td;
Atest[ϵ_] := Qtest[ϵ].Ttest[ϵ].Transpose[Qtest[ϵ]]
NormA = Norm[A, "Frobenius"];
ErrQ[ϵ_] = Transpose[Qtest[ϵ]].Qtest[ϵ] - IdentityMatrix[m];
ErrA[ϵ_] = Atest[ϵ] - (A + ϵ Ad);
LogLogPlot[ {
   Norm[ErrA[ϵ], "Frobenius"]
  ------------------------------ ,
              NormA
   Norm[ErrQ[ϵ], "Frobenius"]
 }, {ϵ, 10^-10, 1},
 Frame → True,
 MaxRecursion → 2,
 GridLines → Automatic,
 FrameLabel → {"ϵ", "err"},
 PlotLegends → {"A", "Q"}]
```



Schur Decomposition: Test for Speed

The AD computation is roughly the same time as the original

```
Clear[m, A, Ad, Qtest, Ttest, Atest, ϵ]
m = 400;
{A, Ad} = RandomReal[{-1, 1}, {2, m, m}];
{Timing[{Q, T} = SchurDecomposition[A];],
 Timing[{Qd, Td} = SchurAD[Ad, {Q, T}];]}
```

```
{{0.440933, Null}, {0.373942, Null}}
```

Higher Derivatives: Schur Decomposition

- $A = Q\ T\ Q^T$
  T is pseudo upper triangular
  Q is orth i.e. $Q^T\ Q = \text{Id}$
  $\text{Ad} = \text{Qd}\ T\ Q^T + Q\ T\ \text{Qd}^T + Q\ \text{Td}\ Q^T$ m^2 entries or equations
  $Q^T\ \text{Ad}\ Q = P\ T - T\ P + \text{Td}$ P skew
- Differentiate again
  $\text{Qd}^T\ \text{Ad}\ Q + Q^T\ \text{Add}\ Q + Q^T\ \text{Ad}\ \text{Qd} = \text{Pd}\ T + P\ \text{Td} - \text{Td}\ P - T\ \text{Pd} + \text{Tdd}$
- Highlight knowns and unknowns in above
  $\text{Qd}^T\ \text{Ad}\ Q + Q^T\ \text{Add}\ Q + Q^T\ \text{Ad}\ \text{Qd} = \text{Pd}\ T - T\ \text{Pd} + P\ \text{Td} - \text{Td}\ P + \text{Tdd}$
  Rearrange
  $\text{Qd}^T\ \text{Ad}\ Q + Q^T\ \text{Add}\ Q + Q^T\ \text{Ad}\ \text{Qd} - P\ \text{Td} + \text{Td}\ P = \text{Pd}\ T - T\ \text{Pd} + \text{Tdd}$
- "Turtles all the way down!" Same Eq for all higher derivatives.
  - These equations can be cascaded

Schur Decomposition: Bulge Chasing Application

A standard computational strategy for computing all the eigenvalues of m×m dense non-symmetric matrix is a two-tone bulge-chasing multishift Implicit QR algorithm.

- K. Braman, R. Byers, and R. Mathias: The multishift QR algorithm. Part I: Maintaining well-focused shifts and level 3 performance, SIAM J. Matrix Anal. Appl., 23 (2002), pp. 929–947.
- K. Braman, R. Byers, and R. Mathias: The multishift QR algorithm. Part II: Aggressive Early Deflation, SIAM J. Matrix Anal. Appl., 23 (2002), pp. 948–973
- Karlsson, L., Kågström, B., Wadbro, E. (2013)
  Fine-Grained Bulge-Chasing Kernels for Strongly Scalable Parallel QR Algorithms.
  Parallel Computing, 2014

Such algorithms are based on computing the upper triangular T of a Schur decomposition by iteratively driving an upper hessenberg matrix H to T using an artfully chosen sequence of orthogonal similarity transformations.

Schur Decomposition: Bulge Chasing Application (cont)

Mulit-shift Implicit QR:

- Chooses a small number of shifts.
  - In practice eigenvalue estimates
- Creates a bulge at the top of the matrix based on these shifts.
- Systematically moves the bulge down and off the bottom of the matrix.
- Chooses new shifts based on the bottom entries of the matrix.
- Repeats until the problem reduces.
  - Currently some subdiagonal entry at the bottom is small

Schur Decomposition: Bulge Chasing Application (cont)

Mulit-shift Implicit QR:

- AED uses a block-sized Schur decomposition at the bottom to improve deflation.
  - In practice multiple passes to achieve reduction
  - AD can enhance shift selection. Work from MA5580 Fall 2013 with Peter Solfest, Alex Hirzel, and Ryan Bruner.
- Middle Deflation (proposed by Brannan) would use similar techniques to target the middle of the matrix.
  - AD can enhance shift selection. Work from MA5580 Fall 2013 with Peter Solfest, Alex Hirzel, and Ryan Bruner.