Tangent Linear and Adjoint Coding Short Course
Day 2
Adjoint Coding

Thomas J. Kleespies
Room 810

Lessons Learned from Yesterday's TL

- **ALL input levels must be perturbed simultaneously in both TL and perturbed forward model**
- **The limit test is performed on output channel Tb_TL**
- **I suggest picking channels that reflect different physics:**
  - **1: a temperature sensitive channel, 2: a water vapor channel,**
  - **3: An ozone sensitive channel, 4: a window channel**
- **When in doubt, check all channels**

- **I had suggested passing forward variables through COMMON. Passing through calling parameters works just as well.**
- **Active variables for this problem are:**
  - **T, Tskin, tau, emiss**
  - **Make TL variables for all of these and differentiate.**
- **Who claims the the prize for first correct solution?**

Further Lessons Learned

Enter Class Comments here:

Review of Previous Day Problem Set

```
!Code excerpt from Compbright_Save_TL.f90

! named common saves intermediate forward model values for use in TL,AD,K codes
Real B(46,19),Bs(19),TotalRad(19)
Common /radiances/ B,Bs,TotalRad

Do ichan = 1 , M

! Initialize integrator
 Sum_TL=0.

! Compute tl radiance for first level
 Call Planck_TL(Vnu(ichan),T(1),T_TL(1),B(1,Ichan),B_TL(1))

! Now compute radiances for the rest of the levels
 Do level=2,N
  Call Planck_TL(Vnu(ichan),T(level),T_TL(level),B(level,Ichan),B_TL(level))

!  Sum=Sum+.5*(B1+B2)*(Tau1-Tau2)  ! forward left commented in place
  Sum_TL = Sum_TL +.5*(                                                        &
   (B_TL(level-1)      +B_TL(level      ))*(Tau    (level-1,ichan)-Tau    (level,ichan)) &
 + (B    (level-1,ichan)+B    (level,ichan))*(Tau_TL(level-1,ichan)-Tau_TL(level,ichan)) &
                                     )
 EndDo

! Surface term, ignoring downward reflected
 Call Planck_TL(Vnu(ichan),Tskin,Tskin_TL,Bs(Ichan),Bs_TL)

!Sum=Sum+Bs*Tau(N,ichan)*Emiss(ichan)  ! forward left commented in place
 Sum_TL = Sum_TL + Bs_TL       *Tau(N,ichan)    *Emiss(ichan)    &
                 + Bs(ichan)   *Tau_TL(N,ichan)*Emiss(ichan)    &
                 + Bs(ichan)   *Tau(N,ichan)    *Emiss_TL(ichan)

! Now brightness temperature
 Tb_TL(ichan) = 0
 If(TotalRad(Ichan).gt.0.) Then
   Tb_TL(ichan) = Bright_TL(Vnu(ichan),TotalRad(Ichan),Sum_TL,BC1(ichan),BC2(ichan))
 EndIf

EndDo ! ichan
```

```
! Code excerpt from Test_Compbright_Save_AD.f90
 Do i = 1 , Niter ! outer loop emulates taking the limit
   Sign = -1.0
   Do isign = 1 , 2 ! inner loop delta x -> 0 +-

    Sign = -Sign

! compute perturbed basic state
    Call Compbright_Save(   Vnu,                            &
                            T+Sign*T_TL,                    &
                            Tau+Sign*Tau_TL,                &
                            Tskin+Sign*Tskin_TL,            &
                            Emiss+Sign*Emiss_TL,            &
                            BC1,BC2,Nlevel,Nchan,           &
                            TbP)

! compute forward model values and variables here for use with TL
    Call Compbright_Save(Vnu,T,Tau,Tskin,Emiss,BC1,BC2,Nlevel,Nchan,Tb)

    Call Compbright_Save_TL(Vnu,                            &
                            T,Sign*T_TL,                    &
                            Tau,Sign*Tau_TL,                &
                            Tskin,Sign*Tskin_TL,            &
                            Emiss,Sign*Emiss_TL,            &
                            BC1,BC2,Nlevel,Nchan,           &
                            Tb,Tb_TL)

    Ratio(isign) = (TbP(Ichan) - Tb(Ichan) ) / Tb_TL(Ichan)         ! ratio

   EndDo ! sign

   Write(6,6120) i, Ratio(1),Ratio(2)

   TLIn =TLin*.5 ! halve perturbation

  EndDo  ! iter
 EndDo
```

Is this a bad result?

```
HIRS channel           5           Single precision
Iter      Pos ratio      Neg ratio
   1    1.034600735    0.968000233
   2    1.016974330    0.983676672
   3    1.008406639    0.991757333
   4    1.004179955    0.995861471
   5    1.002081037    0.997931898
   6    1.001035571    0.998961031
   7    1.000512838    0.999500036
   8    1.000218749    0.999761403
   9    1.000088096    0.999826789
  10    0.999957442    1.000218749
  11    0.999957442    0.999957442
  12    0.999434710    1.000480175
  13    0.999434710    1.001525640
  14    0.995252967    1.003616452
  15    0.995252967    1.003616452
```

Ratio starts to wander at iteration 10

HIRS channel	5	**Double Precision**
Iter	Pos ratio	Neg ratio
1	1.034600773	0.968000178
2	1.016974788	0.983675551
3	1.008406038	0.991756553
4	1.004182687	0.995857961
5	1.002086261	0.997923901
6	1.001041860	0.998960680
7	1.000520613	0.999480023
8	1.000260227	0.999739932
9	1.000130094	0.999869946
10	1.000065042	0.999934968
11	1.000032520	0.999967483
12	1.000016260	0.999983741
13	1.000008130	0.999991870
14	1.000004065	0.999995935
15	1.000002032	0.999997968

Double precision reveals that wandering is a precision issue.
This passes the limit test.

Remember: direction of approach for pos and neg ratio may vary from variable to variable.

Check each variable class.

What good are adjoints?

If your pickup is broken, your girl has left you, and your dog has died:

Using adjoint techniques, you can :

- **fix your pickup,**
- **get your girl back,**
- **and bring your dog back to life, as long as they have been properly linearized. (at least in theory)**

Adjoint coding objective

- **To make the linearized code run backwards.**
- **E.g.: TL code inputs linearized temperature profile and outputs linearized brightness temperature**
- **Adjoint code inputs linearized brightness temperatures and outputs linearized temperature profile**

- **Note that I often interchange ‘linearized’ and ‘derivative’**

Our objective is the Jacobian

$$K(x)^T = \begin{bmatrix} \frac{\partial R_1}{\partial T_1} & \frac{\partial R_2}{\partial T_1} & \frac{\partial R_3}{\partial T_1} & \cdots & \frac{\partial R_m}{\partial T_1} \\ \frac{\partial R_1}{\partial T_2} & \frac{\partial R_2}{\partial T_2} & \frac{\partial R_3}{\partial T_2} & \cdots & \frac{\partial R_m}{\partial T_2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial R_1}{\partial T_n} & \frac{\partial R_2}{\partial T_n} & \frac{\partial R_3}{\partial T_n} & \cdots & \frac{\partial R_m}{\partial T_n} \\ \frac{\partial R_1}{\partial q_1} & \frac{\partial R_2}{\partial q_1} & \frac{\partial R_3}{\partial q_1} & \cdots & \frac{\partial R_m}{\partial q_1} \\ \frac{\partial R_1}{\partial q_2} & \frac{\partial R_2}{\partial q_2} & \frac{\partial R_3}{\partial q_2} & \cdots & \frac{\partial R_m}{\partial q_2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial R_1}{\partial q_n} & \frac{\partial R_2}{\partial q_n} & \frac{\partial R_3}{\partial q_n} & \cdots & \frac{\partial R_m}{\partial q_n} \end{bmatrix}$$

Recommended AD Naming Conventions

There is no ‘standard’ naming convention. Here is what I recommend:

- **Keep forward model variable names the same**
- **Append “ _AD” to forward model variable and routine names to describe adjoint variables and routines**

How do we derive the Adjoint Code

- **By taking the transpose of the Tangent Linear Code**

- **It’s that simple.**

Huh?

Well, maybe it’s not quite that simple.

Adjoint Coding Rules

- **Call forward model first to initialize forward variables**
- **Reverse the order of TL routine calls**
- **Convert Functions to Subroutines**
- **Reverse the order of active loop indices**
- **Reverse the order of code within loops and routines**
- **Reverse the inputs and outputs of assignment statements**
- **Accumulate the outputs of the assignment statements**
- **Rename TL variables and routines to AD**

- **Initializing output accumulators is VERY important**

Example 1: reverse order of routines

TL

```
Program Main_TL
 Call Sub1
 Call Sub2
 Call Sub3

 Call Sub1_TL
 Call Sub2_TL
 Call Sub3_TL
End Program Main_TL
```

Adjoint

```
Program Main_AD
 Call Sub1
 Call Sub2
 Call Sub3

 Call Sub3_AD
 Call Sub2_AD
 Call Sub1_AD
End Program Main_AD
```

Example 2: Functions to Subroutines

Reverse code order, reverse assignment I/O&accumulate

TL

```
Real Function Bright_TL
(V,Radiance,Radiance_TL,BC1,BC2)
K2 = C2*V
K1 = C1*V*V*V

TempTb_TL =
    K2*Alog(K1/Radiance + 1.)**(-2.)
    * Radiance_TL/(K1+Radiance) *
    K1/Radiance        (1)

Bright_TL = BC2*TempTb_TL     (2)

 Return
End Function Bright_TL
```

Adjoint

```
Subroutine Bright_AD
    (V,Radiance,Radiance_AD,BC1,
    BC2,TB_AD)
K2 = C2*V
K1 = C1*V*V*V  !inactive constants
TempTb_AD = 0 ! initialize for each
    invocation
TempTb_AD = TempTb_AD +
    BC2*Tb_AD       (2)

Radiance_AD = Radiance_AD +
    K2*Alog(K1/Radiance + 1.)**(-2.)
    * TempTb_AD/(K1+Radiance) *
    K1/Radiance       (1)
 Return
End Subroutine Bright_AD
```

Example 3 – from Compbright_AD: Reverse inputs and outputs of assignments

```
   1                  2
Sum_TL = Sum_TL  + Bs_TL      *Tau(N,ichan)   *Emiss(ichan)    &
                 + Bs(ichan)  *Tau_TL(N,ichan)*Emiss(ichan)    &
                                  3
                 + Bs(ichan)  *Tau(N,ichan)   *Emiss_TL(ichan)
                                                    4
```

```
Sum_AD = Sum_AD ! Doesn't do anything, we can toss this statement
Bs_AD           = Bs_AD            + Sum_AD    *Tau(N,ichan)*Emiss(ichan)
Tau_AD(N,ichan)= Tau_AD(N,ichan) + Bs(ichan) *Sum_AD      *Emiss(ichan)
Emiss_AD(ichan)= Emiss_AD(ichan) + Bs(ichan) *Tau(N,ichan)*Sum_AD
```

Accumulate Reverse inputs and outputs

Example 3 revisited ala G&K pg 12

m is the current realization of the values

TL

$$\begin{bmatrix} sum' \\ B_s' \\ \tau' \\ \varepsilon' \end{bmatrix}^{m} = \begin{bmatrix} 1 & \tau'\varepsilon' & B_s'\varepsilon' & B_s'\tau' \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sum' \\ B_s' \\ \tau' \\ \varepsilon' \end{bmatrix}^{m-1}$$

Taking the transpose

AD

$$\begin{bmatrix} sum' \\ B_s' \\ \tau' \\ \varepsilon' \end{bmatrix}^{m-1*} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \tau'\varepsilon' & 1 & 0 & 0 \\ B_s'\varepsilon' & 0 & 1 & 0 \\ B_s'\tau' & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sum' \\ B_s' \\ \tau' \\ \varepsilon \end{bmatrix}^{m*}$$

Example 4: Reverse indexing of loops

TL

```
Do I = 1 , Nlevel
 B_Tl(I) = T_TL(I)*Tau(I) + T(I)*Tau_TL(I)
EndDo
```

AD

```
Do I = Nlevel, 1 , -1
 T_AD(I) = T_AD(I) + B_AD(I)*Tau(I)
 Tau_AD(I) = Tau_AD(I) + T(I)*B_AD(I)
EndDo
```

This illustrates reversing loop flow. Doesn’t make any difference for this particular code fragment, but in general it does.

Initializing Accumulators

G&K say zero accumulators after done using them.

However, you have to zero them before you use them the first time,
so just zero them before you start.

AD variables local to a routine should be zeroed there.

Adjoint testing

- **Objective: Assure that the adjoint is the transpose of the tangent linear**
- **Method: Construct Jacobians from TL and AD and compare**

N inputs -> TL -> M outputs

M inputs -> AD -> N outputs

Call TL N times with the ith element=1, all other elements =0

Put output into ith row of an NxM array

Call AD M times with the jth element=1, all other elements=0

Put output into a jth row of an MxN array

Verify that AD = TL^T to within machine precision

Tangent-Linear Output

$$K(X) = \left[\frac{\partial R_1}{\partial X} \frac{\partial R_2}{\partial X} \frac{\partial R_3}{\partial X} \cdots \frac{\partial R_m}{\partial X} \right]$$

For a single call to TL, output is derivative of each channel radiance with respect to whole input state vector.

Adjoint Output

$$K(x)^T = \begin{bmatrix} \frac{\partial R}{\partial T_1} \\ \frac{\partial R}{\partial T_2} \\ \vdots \\ \frac{\partial R}{\partial T_n} \\ \frac{\partial R}{\partial q_1} \\ \frac{\partial R}{\partial q_2} \\ \vdots \\ \frac{\partial R}{\partial q_n} \end{bmatrix}$$

For a single call to AD, output is derivative of all channel radiances with respect to each element of the input state vector.

Filling the Jacobian

We call the TL and AD with all input elements set to zero except one so as to isolate the derivative to a specific element of the Jacobian. This gives the derivative $\frac{\partial R_j}{\partial x_i}$

TL Jacobian Construction

$$K(x_1) = \left[\frac{\partial R_1}{\partial x_1} \; \frac{\partial R_2}{\partial x_1} \; \frac{\partial R_3}{\partial x_1} \cdots \frac{\partial R_m}{\partial x_1} \right]$$

$$K(x_2) = \left[\frac{\partial R_1}{\partial x_2} \; \frac{\partial R_2}{\partial x_2} \; \frac{\partial R_3}{\partial x_2} \cdots \frac{\partial R_m}{\partial x_2} \right]$$

$$\vdots$$

$$K(x_n) = \left[\frac{\partial R_1}{\partial x_n} \; \frac{\partial R_2}{\partial x_n} \; \frac{\partial R_3}{\partial x_n} \cdots \frac{\partial R_m}{\partial x_n} \right]$$

AD Jacobian Construction

$$K_1(x)^T = \begin{bmatrix} \frac{\partial R_1}{\partial T_1} \\ \frac{\partial R_1}{\partial T_2} \\ \vdots \\ \frac{\partial R_1}{\partial T_n} \\ \frac{\partial R}{\partial q_1} \\ \frac{\partial R_1}{\partial q_2} \\ \vdots \\ \frac{\partial R_1}{\partial q_n} \end{bmatrix}, K_2(x)^T = \begin{bmatrix} \frac{\partial R_2}{\partial T_1} \\ \frac{\partial R_2}{\partial T_2} \\ \vdots \\ \frac{\partial R_2}{\partial T_n} \\ \frac{\partial R_2}{\partial q_1} \\ \frac{\partial R_2}{\partial q_2} \\ \vdots \\ \frac{\partial R_2}{\partial q_n} \end{bmatrix}, \cdots, K_m(x)^T = \begin{bmatrix} \frac{\partial R_m}{\partial T_1} \\ \frac{\partial R_m}{\partial T_2} \\ \vdots \\ \frac{\partial R_m}{\partial T_n} \\ \frac{\partial R_m}{\partial q_1} \\ \frac{\partial R_m}{\partial q_2} \\ \vdots \\ \frac{\partial R_m}{\partial q_n} \end{bmatrix}$$

Machine Precision Considerations

Test that

Abs(TL-AD)/TL < MP

Rule of thumb:

MP = 1.e-7 for Single precision

MP = 1.e-12 for Double precision

Use errors intelligently

- **If the AD /= TL^T , use the location in the matrix to find the error in the code.**
- **E.G. if Tsfc_AD /= $Tsfc\_T^T$, look where Tsfc_AD is computed for the error.**

- **Make sure AD variables are initialized to zero.**

Adjoint Testing Example

```
! Compute forward model radiance

 Call Planck(Vnu(Ichan),Temp,B)

! Compute TL values
 Temp_TL = 1.0   ! Initialize input
 B_TL = 0.0      ! Initialize output

 Call Planck_TL(Vnu(Ichan),Temp,Temp_TL,B,B_TL) ! tangent linear model

! Compute AD values
 B_AD = 1.0    ! Initialize input
 Temp_AD = 0.0 ! Initialize output (accumulator)
 Call Planck_AD(Vnu(Ichan),Temp,Temp_AD,B,B_AD) ! Adjoint model

! Here the output of the TL is 1x1 and the output of the AD is 1x1,
! so Transpose(TL) = AD ==> B_TL = Temp_AD

 Write(6,*) B_TL, Temp_AD, B_TL-Temp_AD
```

Problem Set for Tomorrow:

Construct routine COMPBRIGHT_SAVE_AD.F90 from TL code COMPBRIGHT_TL_SAVE.F90 and test using techniques learned today.

Low level routines PLANCK.F90, BRIGHT.F90, PLANCK_TL.F90, Bright_AD.F90, Planck_AD.F90, Bright_AD.F90, COMPBRIGHT_SAVE.F90 and COMPBRIGHT_SAVE_TL.FOR

and their testing routines are provided.

Hint: use of EQUIVALENCE greatly eases the testing.

```
! TL input vector
Equivalence (TLin(1  ), T_TL     )
Equivalence (TLin(47 ), Tau_TL   )
Equivalence (TLin(921), Emiss_TL )
Equivalence (TLin(940), Tskin_TL )

! AD output vector
Equivalence (ADout(1  ), T_AD     )
Equivalence (ADout(47 ), Tau_AD   )
Equivalence (ADout(921), Emiss_AD )
Equivalence (ADout(940), Tskin_AD )

Real Tb   (Nchan)   ! brightness temperature
Real Tb_TL(nTLout)  ! brightness temperature TL
Real Tb_AD(nADin)   ! brightness temperature AD

Real TLout(nTLout)
Real ADin (nADin)

Equivalence(TLout,Tb_TL)
Equivalence(ADin, Tb_AD)

Real TL(nTLin,nTLout)
Real AD(nADin,nADout)
```

Problem Set for Tomorrow cont: Things to watch out for:

If test fails, don't assume that it is in the AD code… it could be in the testing logic.

Remember to zero outputs (accumulators).

Only set one input element to unity at a time. Rest are set to zero.

I find that maybe half of the errors that I chase down are in the testing logic.

Don’t wait to do this assignment.
It is difficult.

Good fun and have luck.