Supplement of Geosci. Model Dev., 16, 47–74, 2023
https://doi.org/10.5194/gmd-16-47-2023-supplement

Geoscientific Model Development
Open Access

Supplement of

ICLASS 1.1, a variational Inverse modelling framework for the Chemistry Land-surface Atmosphere Soil Slab model: description, validation, and application

Peter J. M. Bosman and Maarten C. Krol

Correspondence to: Peter J. M. Bosman (peter.bosman.publicaddress@gmail.com)

S1 Adjoint model

In order to obtain an analytical local derivative, we constructed the adjoint of CLASS. To illustrate the employed technique of adjoint coding, we show in this section some examples of the implementation for a number of forward-model code structures. We have slightly simplified the examples for clarity. For a more detailed background on adjoint models, see Errico (1997) and Giering and Kaminski (1998). Note that the tangent linear code itself is not used in an actual optimisation, but it is used for the construction of the adjoint, for validating the written adjoint code, and it can also be used for calculating model output sensitivities.

S1.1 Single line code example

We start with a short example for one line of forward-model (CLASS) code. For this we chose the calculation of net radiation:

```
Q = Swin - Swout + Lwin - Lwout
```

Where *Swin*, *Swout*, *Lwin* and *Lwout* are the incoming shortwave, outgoing shortwave, incoming longwave and outgoing longwave radiation respectively. We construct the tangent linear code of this statement as follows. We first calculate the derivatives of *Q* to the variables at the right-hand side of the equation. We then multiply these derivatives with a perturbation in the parameter to which we took the derivative. Finally, we add everything up, resulting in the following expression:

```
dQ = dSwin - dSwout + dLwin - dLwout
```

where *dSwin*, *dSwout*, *dLwin* and *dLwout* are small perturbations to variables *Swin*, *Swout*, *Lwin* and *Lwout* respectively. *dQ* is the change in *Q* due to these perturbations. Generally, this tangent linear equation is exact only if the perturbations are infinitesimally small, otherwise the result is a linear approximation to the change in *Q*. In other words, the tangent linear model provides an exact approximation to the change in *Q* at location (*Swin*=*Swin*$_0$, *Swout*=*Swout*$_0$,*Lwin*=*Lwin*$_0$,*Lwout*=*Lwout*$_0$) in parameter space (denoting the values of the variables at the location where we take the derivative with index 0). Note that for this case the forward-model equation is linear, meaning that the tangent linear of the forward-model statement provides an exact approximation for any perturbation size. For general adjoint coding a matrix notation of a tangent linear statement is convenient:

$$\begin{bmatrix} dSwin \\ dSwout \\ dLwin \\ dLwout \\ dQ \end{bmatrix}_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & -1 & 1 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} dSwin \\ dSwout \\ dLwin \\ dLwout \\ dQ \end{bmatrix}_{k-1} \tag{S1}$$

Here, $k-1$ and k are indices for the corresponding position in the forward-model code, i.e. before and after execution of the forward-model code line respectively. Now we switch from the tangent linear variables *dSwin* etc. to the adjoint variables *adSwin*, *adSwout*, *adLwin*, *adLwout* and *adQ*. While doing so we also transpose the matrix with numbers:

$$\begin{bmatrix} adSwin \\ adSwout \\ adLwin \\ adLwout \\ adQ \end{bmatrix}_{k-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} adSwin \\ adSwout \\ adLwin \\ adLwout \\ adQ \end{bmatrix}_k \tag{S2}$$

Note that we have also reversed the indices $k-1$ and k now (Giering and Kaminski, 1998), the adjoint model is "reversed" compared to the tangent linear and forward models. From the matrix we obtain five adjoint equations:

```
adSwin = adQ + adSwin
```

```
adSwout = -adQ + adSwout
adLwin = adQ + adLwin
adLwout = -adQ + adLwout
adQ = 0
```

Observations of variable *Q* could then be used to obtain the model-observation mismatch, which can subsequently be used in the cost function and forcing vector (see Eq. 11 in main text). The forcing vector can subsequently be used to force the adjoint model (see Eq. 10 in main text). In that case, the variables *adSwin*, *adSwout*, *adLwin* and *adLwout* are proportional to the derivatives of the data part of the cost function to the incoming shortwave, outgoing shortwave, incoming longwave and outgoing longwave radiation respectively. Multiplying those variables with a factor 2 gives the actual derivatives (see also Eq. 10 in main text). Regarding the physical units, when the units of the forward-model variable are $\mathrm{W\,m^{-2}}$ (e.g. *Swin*), the units of the adjoint variable (e.g. *adSwin*) will be $\mathrm{m^2W^{-1}}$.

S1.2 Multiple lines code

For two consecutive forward-model statements, two consecutive tangent linear statements are constructed using the principle described above (intermediate tangent linear statements can be useful for more complex statements). Two adjoint statements are also constructed, however, those statements are placed in reverse order. The forward model in essence consists of a collection of main Python functions, and we wrote a separate tangent linear and adjoint function for each of these functions. In the forward model a timestep includes a sequence of calls to these functions. In the tangent linear the same sequence of calls to the tangent linear functions belonging to the forward-model functions is executed. In the adjoint model however, the adjoint functions belonging to the tangent linear functions are called in reverse order. For example, when in the forward model calls are made to the statistics, radiation and surface layer functions, then the adjoint will call the adjoint surface layer function before the adjoint radiation and adjoint statistics functions. The adjoint model also runs backward in time, in contrast to the forward model or tangent linear model, so the order of the time steps is also reversed.

Note that there is no need to calculate derivatives of all model variables, only of those that depend on one of the possible state variables and have a possible influence on the cost function. Those are our "active variables" (Giering and Kaminski, 1998). Similarly, we only need to calculate derivatives with respect to possible state variables and to variables depending on those state variables.

S1.3 Storing variables and functions with adjoint arguments

Some model statements are slightly more complex, requiring additional techniques. The first one is the storing of the value of forward-model variables, for use in the tangent linear and adjoint models. As an example, take the following line of forward-model code:

```
ueff = np.sqrt(u ** 2. + v ** 2. + wstar**2.)
```

Where *u* and *v* are the zonal and meridional wind components, *wstar* is the convective velocity scale (Stull, 1988) and *np.sqrt* is a function that calculates the square root of a variable. In the tangent linear model the statement becomes:

```
dueff = 0.5 * (u**2. + v**2. + wstar**2.)**(-0.5) *
(2 * u * du + 2 * v * dv + 2 * wstar * dwstar)
```

Where *du*, *dv* and *dwstar* are perturbations just as in the earlier example. We give also the corresponding adjoint code here:

```
adu += 0.5 * (u**2. + v**2. + wstar**2.) **(-0.5) * 2 * u * adueff
adv += 0.5 * (u**2. + v**2. + wstar**2.) **(-0.5) * 2 * v * adueff
adwstar += 0.5 * (u**2. + v**2. + wstar**2.) **(-0.5) * 2 * wstar * adueff
adueff = 0
```

Note that *u*, *v* and *wstar* occur also in the tangent linear and adjoint equations. This means that we need to have the values of these variables at all modelled times and iterations available for use in the adjoint and tangent linear models. As the adjoint should provide us a local analytical derivative, we need to have the value of the model variables at exactly the point in state space where we want to calculate the gradient. Therefore, before every call to the adjoint model, we store the necessary information while running the forward model at the same point in state space as the upcoming adjoint call. Since our memory requirements for this are not excessive, we can store all required variables at all time steps and there is no need to employ a more advanced checkpointing scheme (Charpentier, 2001).

The second technique we want to illustrate is the use of a function with adjoint arguments. We take the following example from the forward-model code:

```
esatvar    = esat(theta)
```

Which calculates the saturation vapour pressure based on mixed-layer potential temperature (*theta*). It refers to a function *esat*, which is a simple function given below:

```
def esat(T):
    return 0.611e3 * np.exp(17.2694 * (T - 273.16) / (T - 35.86))
```

Where *T* is temperature and np.exp refers to the exponential function from the Numpy (van der Walt et al. (2011); https://numpy.org) library. For the tangent linear model this leads to the following code:

```
def desat(T,dT):
    return 0.611e3 * np.exp(17.2694 * (T - 273.16) / (T - 35.86)) *
    (-17.2694)*(35.86-273.16)/(T-35.86)**2 * dT
desatvar    = desat(theta,dtheta)
```

For the adjoint code, there is no need to write an adjoint function for *desat*, instead we can write the adjoint code as follows:

```
adtheta += desat(theta,adesatvar)
adesatvar = 0
```

where function *desat* is called with variable *adesatvar* as second argument, instead of the argument *dtheta* that is used in the tangent linear model.

S1.4 If-statements

The adjoint code provides us in principle a locally exact analytical derivative. However, since the functions in the equations implemented in the model are not all continuous over their full domain, they are consequently not all differentiable over their full domain. For example, the following model statement occurs in the forward-model code:

```
ueff           = max(0.01, np.sqrt(u**2. + v**2. + wstar**2.))
```

Now if *v* and *wstar* are 0 and *u* equals 0.01, the derivative $\frac{dueff}{du}$ is undefined, since the left derivative and right derivative are not equal, i.e. the left derivative (moving to lower values of *u*) equals 0 and the right derivative (moving to higher values of *u*) equals 1. We have written the following tangent linear code belonging to this model statement (only *du* term shown here):

```
if np.sqrt(u**2. + v**2. + wstar**2.) < 0.01:
    dueff = 0
else:
    dueff = 0.5 * (u**2. + v**2. + wstar**2.)**(-0.5) * (2 * u * du)
```

Which means that at the point $u = 0.01\ \mathrm{m\,s^{-1}}$, we have chosen to implement the right derivative instead of the left derivative. This is however an arbitrary choice, the right derivative is the correct one to use in case we (or the optimisation algorithm) are interested in the behaviour of *ueff* for increasing *u*, in the other case the left derivative is the correct one. This is a potential source of error for the written adjoint code, however, we have to keep in mind that (in this specific example) the calculated derivative is only (possibly) wrong at the exact value of $u = 0.01$. Furthermore, the model is very non-linear anyway, which makes the calculated gradients only valid locally. Note also that, when running the adjoint code of ICLASS on a computer, small errors that are related to finite machine precision can occur.

S1.5 Loops

For-loops in the forwardmodel result in for-loops in the tangent linear and adjoint models. As an example, the following forward-model code:

```
for i in range(nr_of_surf_lay_its):
    run_surface_layer(call_from_init=True,iterationnumber=i)
```

results in the following tangent linear code:

```
for i in range(nr_of_surf_lay_its):
    tl_run_surface_layer(checkpoint_init[i])
```

and the corresponding adjoint code:

```
for i in range(nr_of_surf_lay_its-1,-1,-1):
    adj_run_surface_layer(checkpoint_init[i])
```

where the function argument `checkpoint_init[i]` contains stored forward model variables, as explained in Sect. S1.3. Note that if the for-loop would contain multiple statements, the order would be reversed in the adjoint code. The forward model also contains a while-loop. This loop can be transformed into a for-loop in the tangent linear and adjoint code (and dealt with similarly as above), when the number of performed iterations for this while-loop is known. Therefore the value of the variable counting the number of iterations performed in the while-loop is stored during a forward-model run. As mentioned earlier, the forward model is run before calling the adjoint, so we have the necessary information.

S2 A-priori error covariance matrix

The a-priori error covariance matrix $\mathbf{S_A}$ is defined as:

$$\begin{bmatrix} \text{var}((\boldsymbol{x}^{\{t\}} - \boldsymbol{x_A})_1) & \dots & \text{cov}((\boldsymbol{x}^{\{t\}} - \boldsymbol{x_A})_1, (\boldsymbol{x}^{\{t\}} - \boldsymbol{x_A})_n) \\ \vdots & \ddots & \vdots \\ \text{cov}((\boldsymbol{x}^{\{t\}} - \boldsymbol{x_A})_n, (\boldsymbol{x}^{\{t\}} - \boldsymbol{x_A})_1) & \dots & \text{var}((\boldsymbol{x}^{\{t\}} - \boldsymbol{x_A})_n) \end{bmatrix} \tag{S3}$$

where $\boldsymbol{x}^{\{t\}}$ is the unknown vector of 'true' values of the parameters in the state vector, $\boldsymbol{x_A}$ represents the a-priori estimate of the state vector, and n is the number of variables in the state vector. The matrix has size $n \times n$. See also Brasseur and Jacob (2017).

S3 Observational error and weight information for application example

In Table S1 we provide the measurement error and model error standard deviations used in the application example, as well as the employed weights in the cost function. The representation errors were set to 0 for all observation streams. The measurement errors and model errors were assumed to follow normal distributions. The measurement error standard deviations and weights were chosen to be independent of time.

Table S1. Used measurement error and model error standard deviations at time t ($\sigma_{I,t}$ and $\sigma_{M,t}$ respectively) in the application example, as well as weights used in the cost function. 'abs' means absolute value. The length of the vector containing the boundary height observations ($\boldsymbol{y_h}$) is used as reference value for setting the weights.

Observation stream	$\sigma_{I,t}$	$\sigma_{M,t}$	Weights
T_{200} (K)	0.1	0.5 for t <= 10:30 UTC, 0.3 otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{T_{200}}})}$
T_{140} (K)	0.1	0.5 for t <= 10:30 UTC, 0.3 otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{T_{140}}})}$
T_{80} (K)	0.1	0.5 for t <= 10:30 UTC, 0.3 otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{T_{80}}})}$
T_{40} (K)	0.1	0.5 for t <= 10:30 UTC, 0.3 otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{T_{40}}})}$
T_{20} (K)	0.1	0.5 for t <= 10:30 UTC, 0.3 otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{T_{20}}})}$
T_{10} (K)	0.1	0.5 for t <= 10:30 UTC, 0.3 otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{T_{10}}})}$
T_2 (K)	0.1	0.5 for t <= 10:30 UTC, 0.3 otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{T_2}})}$
q_{200} ($\text{kg}\,\text{kg}^{-1}$)	1.0×10^{-4}	3.0×10^{-4} for t <= 10:30 UTC, 2.0×10^{-4} otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{q_{200}}})}$
q_{140} ($\text{kg}\,\text{kg}^{-1}$)	1.0×10^{-4}	3.0×10^{-4} for t <= 10:30 UTC, 2.0×10^{-4} otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{q_{140}}})}$
q_{80} ($\text{kg}\,\text{kg}^{-1}$)	1.0×10^{-4}	3.0×10^{-4} for t <= 10:30 UTC, 2.0×10^{-4} otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{q_{80}}})}$
q_{40} ($\text{kg}\,\text{kg}^{-1}$)	1.0×10^{-4}	3.0×10^{-4} for t <= 10:30 UTC, 2.0×10^{-4} otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{q_{40}}})}$
q_{20} ($\text{kg}\,\text{kg}^{-1}$)	1.0×10^{-4}	3.0×10^{-4} for t <= 10:30 UTC, 2.0×10^{-4} otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{q_{20}}})}$
q_{10} ($\text{kg}\,\text{kg}^{-1}$)	1.0×10^{-4}	3.0×10^{-4} for t <= 10:30 UTC, 2.0×10^{-4} otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{q_{10}}})}$
q_2 ($\text{kg}\,\text{kg}^{-1}$)	1.0×10^{-4}	3.0×10^{-4} for t <= 10:30 UTC, 2.0×10^{-4} otherwise	$\frac{1}{7} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{q_2}})}$
$CO2_{207}$ (ppm)	1.0	3.0 for t <= 10:30 UTC, 1.0 otherwise	$\frac{1}{4} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{CO2_{207}}})}$
$CO2_{127}$ (ppm)	1.0	3.0 for t <= 10:30 UTC, 1.0 otherwise	$\frac{1}{4} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{CO2_{127}}})}$
$CO2_{67}$ (ppm)	1.0	3.0 for t <= 10:30 UTC, 1.0 otherwise	$\frac{1}{4} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{CO2_{67}}})}$
$CO2_{27}$ (ppm)	1.0	3.0 for t <= 10:30 UTC, 1.0 otherwise	$\frac{1}{4} \times \frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{CO2_{27}}})}$
h (m)	100.0	60.0	1.0
LE ($\text{W}\,\text{m}^{-2}$)	13.0	$\text{abs}(0.12 \times y_{LE,t})$	$\frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{LE}})}$
H ($\text{W}\,\text{m}^{-2}$)	13.0	$\text{abs}(0.12 \times y_{H,t})$	$\frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{H}})}$
F_{CO2} ($\text{mg}\,\text{CO}_2\,\text{m}^{-2}\,\text{s}^{-1}$)	0.08	$\text{abs}(0.25 \times y_{F_{CO2},t})$	$\frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{F_{CO2}}})}$
S_{out} ($\text{W}\,\text{m}^{-2}$)	2.0	15.0 if t < 13:00 UTC and > 10:30 UTC, 3.0 otherwise	$\frac{\text{length}(\boldsymbol{y_h})}{\text{length}(\boldsymbol{y_{S_{out}}})}$

References

145 Brasseur, G. and Jacob, D.: Inverse Modeling for Atmospheric Chemistry, in: Modeling of Atmospheric Chemistry, pp. 487–537, Cambridge University Press, Cambridge, https://doi.org/10.1017/9781316544754.012, 2017.

Charpentier, I.: Checkpointing Schemes for Adjoint Codes: Application to the Meteorological Model Meso-NH, SIAM Journal on Scientific Computing, 22, 2135–2151, https://doi.org/10.1137/S1064827598343735, 2001.

Errico, R. M.: What Is an Adjoint Model?, Bulletin of the American Meteorological Society, 78, 2577–2591, https://doi.org/10.1175/1520-0477(1997)078<2577:WIAAM>2.0.CO;2, 1997.

Giering, R. and Kaminski, T.: Recipes for adjoint code construction, ACM Transactions on Mathematical Software, 24, 437–474, https://doi.org/10.1145/293686.293695, 1998.

Stull, R. B.: An introduction to boundary layer meteorology, Kluwer Academic Publishers, Dordrecht, 1988.

van der Walt, S., Colbert, S., and Varoquaux, G.: The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22–30, https://doi.org/10.1109/MCSE.2011.37, 2011.