UM perturbation sensitivity project

Dealing with rounding issues

Background

The UM perturbation sensitivity project is currently in the process of identifying coding issues that lead to excessive perturbation growth in the model. Currently, all problems are emerging at `IF` tests that contain comparisons between real numbers; for example `IF (qCL(i) > 0.0 ) ....` In this test, `qCL(i)` is being used to represent one of two states; "no liquid cloud", or "some liquid cloud". This is fine, but it is then important to ensure that rounding issues do not lead to unintended changes of state prior to the test, such as slightly non-zero `qCL(i)` values when there is supposed to be no liquid cloud. If such problems occur at discontinuous branches in the code, the result is spurious perturbation growth.

This document collects together some typical examples of what can go wrong, and how to deal with them. First, though, it is worth making a quick note of some of the characteristics of floating-point arithmetic.

Floating-point identities and non-identities

In floating-point arithmetic many of the identities that hold in normal arithmetic no longer hold, basically because of the limited precision available to represent real numbers. Thus, it is often important that coders know which algebraic identities pass through to floating-point arithmetic and which don't, and how results can be affected by the way the calculations are implemented by the compiler. The following is based on what I have found through internet searches. However, I haven't been able to track down any particularly good references for this, so please let me know if you spot any errors. For chapter and verse on floating-point arithmetic, a good reference is David Goldberg's article (http://docs.sun.com/source/806-3568/ncg_goldberg.html).

The following floating-point identity always holds:

```
  0.0 * x ≡ 0.0
```

The following also hold, but only if the numbers that go into the calculations have the same precision:

```
  0.0 + x ≡ x
  1.0 * x ≡ x
    x / x ≡ 1.0
    x - x ≡ 0.0
    x - y ≡ x + (-y)
    x + y ≡ y + x
    x * y ≡ y * x
  2.0 * x ≡ x + x
  0.5 * x ≡ x / 2.0
```

For example, optimisation may lead to some variables being held in cache and others in main memory, and these will generally store numbers with different levels of precision. Thus, coding based on these identities will probably work as intended in most circumstances, but may be vulnerable to higher levels of optimisation.

The following are non-identities:

```
x + (y + z) !≡ (x + y) + z
x * (y * z) !≡ (x * y) * z
x * (y / z) !≡ (x * y) / z
```

These say that, unlike in normal arithmetic, the order of the calculations matters. Failure to recognise this can cause problems, as in example 1 below. (Note that putting brackets around calculations to try and impose the "correct" order of calculation will not necessarily work; the compiler will decide for itself!)

Example 1: Non-distributive arithmetic

At UM vn7.4, the routine `LSP_DEPOSITION` contains the following calculation:

```
          ! Deposition removes some liquid water content
          ! First estimate of the liquid water removed is explicit
          dqil(i) = max (min ( dqi_dep(i)*area_mix(i)                 &
     &                /(area_mix(i)+area_ice_1(i)),                   &
     &                qcl(i)*area_mix(i)/cfliq(i)) ,0.0)
...
        If (l_seq) Then
          qcl(i) = qcl(i) - dqil(i)  ! Bergeron Findeisen acts first
```

Here, `dqil` is a change to cloud liquid water `qcl`, which is limited in the calculation to `qcl*area_mix/cfliq`, where `area_mix` is the fraction of the gridbox with both liquid and ice cloud, and `cfliq` is the fraction with liquid cloud. Basically, the change to cloud liquid water is being limited by the amount of liquid cloud which overlaps with ice cloud it can deposit onto.

In the special case that all the liquid cloud coincides with ice cloud, we have `area_mix = cfliq`, implying `area_mix/cfliq = 1.0`. In this case, the limit for `dqil` should be exactly `qcl`, but is coded as `qcl*area_mix/cfliq`. In tests on the IBM, it seems that the compiler decides that the multiplication should precede the division, so the outcome of the calculation is not necessarily `qcl`. Thus, the update to `qcl` on the last line does not necessarily lead to `qcl = 0.0` when the limit is hit.

One solution to this problem is to supply `area_mix/cfliq` directly as a ratio:

```
        If (cfliq(i) /= 0.0) Then
          areamix_over_cfliq(i)=area_mix(i)/cfliq(i)
        End if
...
          ! Deposition removes some liquid water content
          ! First estimate of the liquid water removed is explicit
          dqil(i) = max (min ( dqi_dep(i)*area_mix(i)                 &
     &                /(area_mix(i)+area_ice_1(i)),                   &
     &                qcl(i)*areamix_over_cfliq(i)) ,0.0)
```

This is the solution we have adopted in the large-scale precipitation code.

Example 2: Changing units when applying limits

At UM vn7.4, the routine `LSP_TIDY` contains the following calculation:

```
          ! Calculate transfer rate
          dpr(i) = temp7(i) / lfrcp ! Rate based on Tw excess

          ! Limit to the amount of snow available
          dpr(i) = min(dpr(i) , snow_agg(i)                             &
     &                        * dhi(i)*iterations*rhor(i) )
...
          ! Update values of snow and rain
          If (l_seq) Then
            snow_agg(i) = snow_agg(i) - dpr(i)*rho(i)*dhilsiterr(i)
            qrain(i)    = qrain(i)    + dpr(i)
```

where

```
dhilsiterr(i) = 1.0/(dhi(i)*iterations)
rhor(i)       = 1.0/rho(i)
```

Here, `dpr` is a conversion rate from snow into rain, and the second statement limits this rate to that required to melt all of the snow within the timestep. Thus, the intention is that if this limit is hit the final snow amount will come out to exactly 0.0. However, the outcome in this case is effectively as follows:

```
  dpr(i)      = snow_agg(i) * dhi(i)*iterations*rhor(i)
  snow_agg(i) = snow_agg(i) - dpr(i)*rho(i)*dhilsiterr(i)
            ( = snow_agg(i) &
              - snow_agg(i) &
              * dhi(i)*iterations*rhor(i)*rho(i)*1.0/(dhi(i)*iterations) )
```

In normal arithmetic, the multiplier on the final line comes out to exactly one, but this is not necessarily the case in floating-point arithmetic. Whether the expression comes out to exactly 1.0 or not will be highly sensitive to the values going into the calculation. If the result is slightly different to 1.0, the outcome is likely to be a tiny but non-zero snow amount.

The basic problem here is that the limit comes from a particular quantity, but is being applied indirectly via its rate of change. Thus when the limiting quantity is updated a change of units is required. The solution here is to apply the limit to the quantity itself, shifting the change of units to calculations involving rates:

```
          ! Calculate transfer
          dp(i) = rho(i)*dhilsiterr(i)*temp7(i) / lfrcp

          ! Limit to the amount of snow available
          dp(i) = min(dp(i), snow_agg(i))
...
          ! Update values of snow and rain
          If (l_seq) Then
            snow_agg(i) = snow_agg(i) - dp(i)
            qrain(i)    = qrain(i)    + dp(i)*dhi(i)*iterations*rhor(i)
```

Example 3: Dealing with special cases

At UM vn7.4, the routine `LS_CLD` contains the following calculation to update the total cloud fraction `CF` given the liquid and frozen cloud fractions `CFL` and `CFF`:

```
            TEMP0=OVERLAP_RANDOM
            TEMP1=0.5*(OVERLAP_MAX-OVERLAP_MIN)
            TEMP2=0.5*(OVERLAP_MAX+OVERLAP_MIN)-OVERLAP_RANDOM
            CF(I,J,K)=CFL(I,J,K)+CFF(I,J,K)                             &
     &              -(TEMP0+TEMP1*OVERLAP_ICE_LIQUID                    &
     &              +TEMP2*OVERLAP_ICE_LIQUID*OVERLAP_ICE_LIQUID)
! Check that the overlap wasnt negative
            CF(I,J,K)=MIN(CF(I,J,K),CFL(I,J,K)+CFF(I,J,K))
```

During testing, it was observed that `CF` was often coming out to 0.9999999999999....; i.e., almost but not quite 1.0, and that whether this occured was highly sensitive to the input data. This sensitivity was then being passed down to branches testing on, for example, whether `CF` was equal to `CFF`.

If the above calculations are followed through algebraically, it can be shown that if `CFL`+`CFF` ≥ 1, then `CF` must be exactly one. In the floating-point case, however, this no longer follows, so we often get cases where there is a slight deviation from unity. The simplest solution in this example is to deal with the special case separately:

```
            TEMP0=OVERLAP_RANDOM
            TEMP1=0.5*(OVERLAP_MAX-OVERLAP_MIN)
            TEMP2=0.5*(OVERLAP_MAX+OVERLAP_MIN)-OVERLAP_RANDOM
! CFF + CFL >= 1 implies CF = 1
            IF (CFL(I,J,K)+CFF(I,J,K) >= 1.0) THEN
              CF(I,J,K) = 1.0
            ELSE
              CF(I,J,K)=CFL(I,J,K)+CFF(I,J,K)                             &
     &                -(TEMP0+TEMP1*OVERLAP_ICE_LIQUID                    &
     &                +TEMP2*OVERLAP_ICE_LIQUID*OVERLAP_ICE_LIQUID)
! Check that the overlap wasnt negative
              CF(I,J,K)=MIN(CF(I,J,K),CFL(I,J,K)+CFF(I,J,K))
            END IF
```