```fortran
#include "fabm_driver.h"
#include "fabm_private.h"

! This module define standard expressions that can be used by biogeochemical models.

module fabm_expressions

   use fabm_types
   use fabm_driver

   implicit none

   private

   public temporal_mean, temporal_maximum, vertical_mean, vertical_integral
   public type_interior_temporal_mean, type_horizontal_temporal_mean, type_horizontal_temporal_maximum, type_vertical_integral

   type, extends(type_interior_expression) :: type_interior_temporal_mean
      real(rk) :: period   ! Time period to average over (s)
      integer  :: n
      real(rk) :: missing_value = -2.e20_rk
      type (type_link), pointer :: link => null()
      integer :: in = -1

      real(rk), private :: previous_time, bin_end_time
      integer,  private :: ioldest  = -1
      integer,  private :: icurrent = -1
      logical,  private :: complete = .false.
      real(rke), allocatable _DIMENSION_GLOBAL_PLUS_1_ :: history
#if _FABM_DIMENSION_COUNT_>0
      real(rke), allocatable _DIMENSION_GLOBAL_ :: previous_value, last_exact_mean, mean
#else
      real(rke) :: previous_value, last_exact_mean, mean
#endif
   contains
      procedure :: update => interior_temporal_mean_update
   end type

   type, extends(type_horizontal_expression) :: type_horizontal_temporal_mean
      real(rk) :: period   ! Time period to average over (s)
      integer  :: n
      real(rk) :: missing_value = -2.e20_rk
      real(rk) :: last_time, next_save_time

      type (type_link), pointer :: link => null()
      integer :: in = -1

      real(rk), private :: previous_time, bin_end_time
      integer :: ioldest  = -1
      integer :: icurrent = -1
      logical,  private :: complete = .false.
      real(rke), allocatable _DIMENSION_GLOBAL_HORIZONTAL_PLUS_1_ :: history
#if _HORIZONTAL_DIMENSION_COUNT_>0
      real(rke), allocatable _DIMENSION_GLOBAL_HORIZONTAL_ :: previous_value, last_exact_mean, mean
#else
      real(rke) :: previous_value, last_exact_mean, mean
#endif
   end type

   type, extends(type_horizontal_expression) :: type_horizontal_temporal_maximum
      real(rk) :: period                     ! Time window to compute running maximum over (s)
      integer  :: n                          ! Number of bins to use to cover the period
      real(rk) :: missing_value = -2.e20_rk  ! Missing value to use until the simulation has covered the window size [period]

      type (type_link), pointer :: link => null()
      integer :: in = -1

      real(rk), private :: previous_time, bin_end_time
      integer :: icurrent = -1
      logical,  private :: complete = .false.
      real(rke), allocatable _DIMENSION_GLOBAL_HORIZONTAL_PLUS_1_ :: history
#if _HORIZONTAL_DIMENSION_COUNT_>0
      real(rke), allocatable _DIMENSION_GLOBAL_HORIZONTAL_ :: previous_value, maximum
#else
      real(rke) :: previous_value, maximum
#endif
   contains
      procedure :: update => horizontal_temporal_maximum_update
   end type

   type, extends(type_horizontal_expression) :: type_vertical_integral
      real(rk) :: minimum_depth = 0.0_rk        ! Depth below surface in m (positive)
      real(rk) :: maximum_depth = huge(1.0_rk)  ! Depth below surface in m (positive)
      logical  :: average       = .false.       ! Whether to divide the depth integral by water depth, thus computing the vertical average
      character(len=attribute_length) :: input_name = ''

      type (type_link), pointer :: link => null()
   end type

   interface temporal_mean
      module procedure interior_temporal_mean
      module procedure horizontal_temporal_mean
   end interface

   interface temporal_maximum
```

```
      module procedure horizontal_temporal_maximum
   end interface

   interface vertical_mean
      module procedure vertical_dependency_mean
      module procedure vertical_state_mean
   end interface

   interface vertical_integral
      module procedure vertical_dependency_integral
      module procedure vertical_state_integral
   end interface

contains

   function vertical_dependency_mean(input, minimum_depth, maximum_depth) result(expression)
      type (type_dependency_id), intent(inout), target :: input
      real(rk), optional,        intent(in)            :: minimum_depth,maximum_depth
      type (type_vertical_integral)                    :: expression
      expression = vertical_integral(input,minimum_depth,maximum_depth,average=.true.)
   end function

   function vertical_state_mean(input, minimum_depth, maximum_depth) result(expression)
      type (type_state_variable_id), intent(inout), target :: input
      real(rk), optional,            intent(in)            :: minimum_depth, maximum_depth
      type (type_vertical_integral)                        :: expression
      expression = vertical_integral_generic(input,minimum_depth,maximum_depth,average=.true.)
   end function

   function vertical_dependency_integral(input, minimum_depth, maximum_depth, average) result(expression)
      type (type_dependency_id), intent(inout),target :: input
      real(rk), optional,        intent(in)           :: minimum_depth, maximum_depth
      logical,  optional,        intent(in)           :: average
      type (type_vertical_integral)                     :: expression
      expression = vertical_integral_generic(input, minimum_depth, maximum_depth, average)
   end function

   function vertical_state_integral(input, minimum_depth, maximum_depth, average) result(expression)
      type (type_state_variable_id), intent(inout), target :: input
      real(rk), optional,            intent(in)            :: minimum_depth, maximum_depth
      logical,  optional,            intent(in)            :: average
      type (type_vertical_integral)                          :: expression
      expression = vertical_integral_generic(input, minimum_depth, maximum_depth, average)
   end function

   function vertical_integral_generic(input, minimum_depth, maximum_depth, average) result(expression)
      class (type_variable_id), intent(inout), target :: input
      real(rk), optional,       intent(in)            :: minimum_depth, maximum_depth
      logical,  optional,       intent(in)            :: average
      type (type_vertical_integral)                   :: expression

      character(len=attribute_length) :: postfix

      if (.not. associated(input%link)) call fatal_error('fabm_expressions::vertical_mean', &
         'Input variable has not been registered yet.')
      expression%input_name = input%link%target%name

      ! Create a name for the expression
      postfix = ''
      if (present(minimum_depth) .and. present(maximum_depth)) then
         if (minimum_depth > maximum_depth) call fatal_error('fabm_expressions::vertical_mean', &
            'Minimum depth exceeds maximum depth.')
         write (postfix,'(a,i0,a,i0,a)') '_between_', int(minimum_depth), '_m_and_', int(maximum_depth), '_m'
      elseif (present(minimum_depth)) then
         write (postfix,'(a,i0,a)') '_below_', int(minimum_depth), '_m'
      elseif (present(maximum_depth)) then
         write (postfix,'(a,i0,a)') '_above_', int(maximum_depth), '_m'
      end if
      if (present(average)) expression%average = average

      if (expression%average) then
         expression%output_name = 'vertical_mean_' // trim(input%link%name) // trim(postfix)
      else
         expression%output_name = 'integral_of_' // trim(input%link%name) // '_wrt_depth' // trim(postfix)
      end if

      expression%link => input%link
      if (present(minimum_depth)) expression%minimum_depth = minimum_depth
      if (present(maximum_depth)) expression%maximum_depth = maximum_depth
   end function

   function interior_temporal_mean(input, period, resolution, missing_value) result(expression)
      type (type_dependency_id), intent(inout), target :: input
      real(rk),                  intent(in)            :: period, resolution
      real(rk), optional,        intent(in)            :: missing_value
      type (type_interior_temporal_mean)               :: expression

      character(len=attribute_length) :: prefix, postfix

      if (.not. associated(input%link)) call fatal_error('fabm_expressions::interior_temporal_mean', &
         'Input variable has not been registered yet.')

      write (expression%output_name,'(i0,a,a,a,i0,a)') int(period), '_s_mean_', trim(input%link%name), '_at_', int(resolution), '_s_resolution'
      expression%link => input%link
      expression%n = nint(period / resolution)
      expression%period = period
      if (present(missing_value)) expression%missing_value = missing_value
```

```fortran
   end function

   function horizontal_temporal_mean(input, period, resolution, missing_value) result(expression)
      class (type_horizontal_dependency_id), intent(inout), target :: input
      real(rk),                              intent(in)            :: period, resolution
      real(rk), optional,                    intent(in)            :: missing_value
      type (type_horizontal_temporal_mean)                         :: expression

      character(len=attribute_length) :: prefix, postfix

      if (.not. associated(input%link)) call fatal_error('fabm_expressions::horizontal_temporal_mean', &
         'Input variable has not been registered yet.')

      write (expression%output_name,'(i0,a,a,a,i0,a)') int(period), '_s_mean_', trim(input%link%name), '_at_', int(resolution), '_s_resolution'
      expression%link => input%link
      expression%n = nint(period / resolution)
      expression%period = period
      if (present(missing_value)) expression%missing_value = missing_value
   end function

   function horizontal_temporal_maximum(input, period, resolution, missing_value) result(expression)
      class (type_horizontal_dependency_id), intent(inout), target :: input
      real(rk),                              intent(in)            :: period, resolution
      real(rk), optional,                    intent(in)            :: missing_value
      type (type_horizontal_temporal_maximum)                      :: expression

      if (.not. associated(input%link)) call fatal_error('fabm_expressions::horizontal_temporal_max', &
         'Input variable has not been registered yet.')

      write (expression%output_name,'(i0,a,a,a,i0,a)') int(period), '_s_max_', trim(input%link%name), '_at_', int(resolution), '_s_resolution'
      expression%link => input%link
      expression%n = nint(period / resolution)
      expression%period = period
      if (present(missing_value)) expression%missing_value = missing_value
   end function

   subroutine interior_temporal_mean_update(self, time, value _POSTARG_LOCATION_RANGE_)
      class (type_interior_temporal_mean), intent(inout) :: self
      real(rke),                           intent(in)    :: time
      real(rke) _ATTRIBUTES_GLOBAL_,       intent(in)    :: value
      _DECLARE_ARGUMENTS_LOCATION_RANGE_

      integer  :: ibin
      real(rke) :: dt, w, dt_bin
      _DECLARE_LOCATION_

      ! Note that all array processing below uses explicit loops in order to respect
      ! any limits on the active domain given by the _LOCATION_RANGE_ argument.

      dt_bin = self%period / self%n

      if (self%ioldest == -1) then
         ! Start of simulation
         self%previous_time = time
         self%bin_end_time = time + dt_bin
         self%icurrent = 1
         self%ioldest = 2
         self%previous_value = 0.0_rke
      end if

      do while (time >= self%bin_end_time)
         ! Linearly interpolate to value at end-of-bin time and add that to the current bin
         dt = self%bin_end_time - self%previous_time
         w = dt / (time - self%previous_time)   ! weight for current time (leaving 1-w for previous time)
         _BEGIN_GLOBAL_LOOP_
            self%history(_PREARG_LOCATION_ self%icurrent) = self%history(_PREARG_LOCATION_ self%icurrent) &
               + ((1._rke - 0.5_rke * w) * self%previous_value _INDEX_LOCATION_ + 0.5_rke * w * value _INDEX_LOCATION_) &
               * dt / self%period
         _END_GLOBAL_LOOP_

         if (self%complete) then
            ! We already had a complete history (bins covering the full window size). Add the newly full bin, subtract the oldest bin
            _BEGIN_GLOBAL_LOOP_
               self%last_exact_mean _INDEX_LOCATION_ = self%last_exact_mean _INDEX_LOCATION_ &
                  - self%history(_PREARG_LOCATION_ self%ioldest) + self%history(_PREARG_LOCATION_ self%icurrent)
            _END_GLOBAL_LOOP_
         elseif (self%icurrent == self%n) then
            ! We just completed our history. Create the mean by summing all filled bins.
            do ibin = 1, self%n
               _BEGIN_GLOBAL_LOOP_
                  self%last_exact_mean _INDEX_LOCATION_ = self%last_exact_mean _INDEX_LOCATION_  &
                     + self%history(_PREARG_LOCATION_ ibin)
               _END_GLOBAL_LOOP_
            end do
            self%complete = .true.
         end if

         ! Update previous time and value to match end of current bin
         self%previous_time = self%bin_end_time
         _BEGIN_GLOBAL_LOOP_
            self%previous_value _INDEX_LOCATION_ = (1._rke - w) * self%previous_value _INDEX_LOCATION_ + w * value _INDEX_LOCATION_
         _END_GLOBAL_LOOP_

         ! Move to next bin: update indices, end time and empty newly current bin
         self%icurrent = self%ioldest
         self%ioldest = self%ioldest + 1
         if (self%ioldest > self%n + 1) self%ioldest = 1
         self%bin_end_time = self%bin_end_time + dt_bin
         _BEGIN_GLOBAL_LOOP_
            self%history(_PREARG_LOCATION_ self%icurrent) = 0
         _END_GLOBAL_LOOP_
      end do

      ! Compute average of previous and current value, multiply by time difference, pre-divide by window size, and add to current bin.
      _BEGIN_GLOBAL_LOOP_
         self%history(_PREARG_LOCATION_ self%icurrent) = self%history(_PREARG_LOCATION_ self%icurrent) &
            + 0.5_rke * (self%previous_value _INDEX_LOCATION_ + value _INDEX_LOCATION_) &
            * (time - self%previous_time) / self%period
      _END_GLOBAL_LOOP_

      if (self%complete) then
         ! We have a full history covering at least one window size. Update the running mean.
         ! The result is an approximation that assumes linear change over the period covered by the oldest bin.
         _BEGIN_GLOBAL_LOOP_
            self%mean _INDEX_LOCATION_ = self%last_exact_mean _INDEX_LOCATION_ &
               + self%history(_PREARG_LOCATION_ self%icurrent) &
               - self%history(_PREARG_LOCATION_ self%ioldest) * (time - self%bin_end_time + dt_bin) / dt_bin
         _END_GLOBAL_LOOP_
      end if

      ! Store current time and value to enable linear interpolation in subsequent call.
      self%previous_time = time
      _BEGIN_GLOBAL_LOOP_
         self%previous_value _INDEX_LOCATION_ = value _INDEX_LOCATION_
      _END_GLOBAL_LOOP_
   end subroutine

   subroutine horizontal_temporal_mean_update(self, time, value _POSTARG_HORIZONTAL_LOCATION_RANGE_)
      class (type_horizontal_temporal_mean),    intent(inout) :: self
      real(rke),                                intent(in)    :: time
      real(rke) _ATTRIBUTES_GLOBAL_HORIZONTAL_, intent(in)    :: value
      _DECLARE_ARGUMENTS_HORIZONTAL_LOCATION_RANGE_

      integer  :: ibin
      real(rke) :: dt, w, dt_bin
      _DECLARE_HORIZONTAL_LOCATION_

      ! Note that all array processing below uses explicit loops in order to respect
      ! any limits on the active domain given by the _HORIZONTAL_LOCATION_RANGE_ argument.

      dt_bin = self%period / self%n

      if (self%ioldest == -1) then
         ! Start of simulation
         self%previous_time = time
         self%bin_end_time = time + dt_bin
         self%icurrent = 1
         self%ioldest = 2
         self%previous_value = 0.0_rke
      end if

      do while (time >= self%bin_end_time)
         ! Linearly interpolate to value at end-of-bin time and add that to the current bin
         dt = self%bin_end_time - self%previous_time
         w = dt / (time - self%previous_time)   ! weight for current time (leaving 1-w for previous time)
         _BEGIN_OUTER_VERTICAL_LOOP_
            self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) = self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) &
               + ((1._rke - 0.5_rke * w) * self%previous_value _INDEX_HORIZONTAL_LOCATION_ + 0.5_rke * w * value _INDEX_HORIZONTAL_LOCATION_) &
               * dt / self%period
         _END_OUTER_VERTICAL_LOOP_

         if (self%complete) then
            ! We already had a complete history (bins covering the full window size). Add the newly full bin, subtract the oldest bin
            _BEGIN_OUTER_VERTICAL_LOOP_
               self%last_exact_mean _INDEX_HORIZONTAL_LOCATION_ = self%last_exact_mean _INDEX_HORIZONTAL_LOCATION_ &
                  - self%history(_PREARG_HORIZONTAL_LOCATION_ self%ioldest) + self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent)
            _END_OUTER_VERTICAL_LOOP_
         elseif (self%icurrent == self%n) then
            ! We just completed our history. Create the mean by summing all filled bins.
            do ibin = 1, self%n
               _BEGIN_OUTER_VERTICAL_LOOP_
                  self%last_exact_mean _INDEX_HORIZONTAL_LOCATION_ = self%last_exact_mean _INDEX_HORIZONTAL_LOCATION_ &
                     + self%history(_PREARG_HORIZONTAL_LOCATION_ ibin)
               _END_OUTER_VERTICAL_LOOP_
            end do
            self%complete = .true.
         end if

         ! Update previous time and value to match end of current bin
         self%previous_time = self%bin_end_time
         _BEGIN_OUTER_VERTICAL_LOOP_
            self%previous_value _INDEX_HORIZONTAL_LOCATION_ = (1._rke - w) * self%previous_value _INDEX_HORIZONTAL_LOCATION_ + w * value _INDEX_HORIZONTAL_LOCATION_
         _END_OUTER_VERTICAL_LOOP_

```

```fortran
         ! Move to next bin: update indices, end time and empty newly current bin
         self%icurrent = self%ioldest
         self%ioldest = self%ioldest + 1
         if (self%ioldest > self%n + 1) self%ioldest = 1
         self%bin_end_time = self%bin_end_time + dt_bin
         _BEGIN_OUTER_VERTICAL_LOOP_
            self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) = 0
         _END_OUTER_VERTICAL_LOOP_
      end do

      ! Compute average of previous and current value, multiply by time difference, pre-divide by window size, and add to current bin.
      _BEGIN_OUTER_VERTICAL_LOOP_
         self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) = self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) &
            + 0.5_rke * (self%previous_value _INDEX_HORIZONTAL_LOCATION_ + value _INDEX_HORIZONTAL_LOCATION_) &
            * (time - self%previous_time) / self%period
      _END_OUTER_VERTICAL_LOOP_

      if (self%complete) then
         ! We have a full history covering at least one window size. Update the running mean.
         ! The result is an approximation that assumes linear change over the period covered by the oldest bin.
         _BEGIN_OUTER_VERTICAL_LOOP_
            self%mean _INDEX_HORIZONTAL_LOCATION_ = self%last_exact_mean _INDEX_HORIZONTAL_LOCATION_ &
               + self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) &
               - self%history(_PREARG_HORIZONTAL_LOCATION_ self%ioldest) * (time - self%bin_end_time + dt_bin) / dt_bin
         _END_OUTER_VERTICAL_LOOP_
      end if

      ! Store current time and value to enable linear interpolation in subsequent call.
      self%previous_time = time
      _BEGIN_OUTER_VERTICAL_LOOP_
         self%previous_value _INDEX_HORIZONTAL_LOCATION_ = value _INDEX_HORIZONTAL_LOCATION_
      _END_OUTER_VERTICAL_LOOP_
   end subroutine

   subroutine horizontal_temporal_maximum_update(self, time, value _POSTARG_HORIZONTAL_LOCATION_RANGE_)
      class (type_horizontal_temporal_maximum), intent(inout) :: self
      real(rke),                                intent(in)    :: time
      real(rke) _ATTRIBUTES_GLOBAL_HORIZONTAL_, intent(in)    :: value
      _DECLARE_ARGUMENTS_HORIZONTAL_LOCATION_RANGE_

      integer  :: ibin
      real(rke) :: w
      _DECLARE_HORIZONTAL_LOCATION_

      ! Note that all array processing below uses explicit loops in order to respect
      ! any limits on the active domain given by the _HORIZONTAL_LOCATION_RANGE_ argument.

      if (self%icurrent == -1) then
         ! Start of simulation
         self%bin_end_time = time + self%period / self%n
         self%icurrent = 1
      end if

      do while (time >= self%bin_end_time)
         ! Update previous time and value to match end of current bin. For the latter, linearly interpolate to value at end-of-bin time.
         w = (self%bin_end_time - self%previous_time) / (time - self%previous_time)   ! weight for current time (leaving 1-w for previous time)
         self%previous_time = self%bin_end_time
         _BEGIN_OUTER_VERTICAL_LOOP_
            self%previous_value _INDEX_HORIZONTAL_LOCATION_ = (1._rke - w) * self%previous_value _INDEX_HORIZONTAL_LOCATION_ + w * value _INDEX_HORIZONTAL_LOCATION_
         _END_OUTER_VERTICAL_LOOP_

         ! Complete the current bin by taking the maximum over its previous value and the value at end-of-boin time.
         _BEGIN_OUTER_VERTICAL_LOOP_
            self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) = max(self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent), &
               self%previous_value _INDEX_HORIZONTAL_LOCATION_)
         _END_OUTER_VERTICAL_LOOP_

         self%complete = self%complete .or. self%icurrent == self%n
         if (self%complete) then
            ! We have a complete history - compute the maximum over all bins
            _BEGIN_OUTER_VERTICAL_LOOP_
               self%maximum _INDEX_HORIZONTAL_LOCATION_ = self%history(_PREARG_HORIZONTAL_LOCATION_ 1)
            _END_OUTER_VERTICAL_LOOP_
            do ibin = 2, self%n
               _BEGIN_OUTER_VERTICAL_LOOP_
                  self%maximum _INDEX_HORIZONTAL_LOCATION_ = max(self%maximum _INDEX_HORIZONTAL_LOCATION_,  &
                     self%history(_PREARG_HORIZONTAL_LOCATION_ ibin))
               _END_OUTER_VERTICAL_LOOP_
            end do
         end if

         ! Move to next bin: update indices, end time and set maximum of newly current bin to current value (at start of bin)
         self%icurrent = self%icurrent + 1
         if (self%icurrent > self%n) self%icurrent = 1
         self%bin_end_time = self%bin_end_time + self%period / self%n
         _BEGIN_OUTER_VERTICAL_LOOP_
            self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) = self%previous_value _INDEX_HORIZONTAL_LOCATION_
         _END_OUTER_VERTICAL_LOOP_
      end do

      ! Update the maximum of the current bin
      _BEGIN_OUTER_VERTICAL_LOOP_
         self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent) = max(self%history(_PREARG_HORIZONTAL_LOCATION_ self%icurrent), &
            value _INĐEX_HORIZONTAL_LOCATION_)
      _ENĐ_OUTER_VERTICAL_LOOP_

      ! Store current time and value to enable linear interpolation in subsequent call.
      self%previous_time = time
      _BEGIN_OUTER_VERTICAL_LOOP_
         self%previous_value _INĐEX_HORIZONTAL_LOCATION_ = value _INĐEX_HORIZONTAL_LOCATION_
      _ENĐ_OUTER_VERTICAL_LOOP_
   end subroutine

end module fabm_expressions
```