Use of the Weighted Histogram Analysis Method for the Analysis of Simulated and Parallel Tempering Simulations. Supplementary material: Sample Implementation in Fortran 95

John D. Chodera[*†], **William C. Swope**[‡], **Jed W. Pitera**[‡], **Chaok Seok**[§], **and Ken A. Dill**[¶]

[†] *Graduate Group in Biophysics and* [¶] *Department of Pharmaceutical Chemistry, University of California at San Francisco, San Francisco, CA 94143*

[‡] *IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120*

[§] *Department of Chemistry, College of Natural Sciences, Seoul National University, Gwanak-gu, Shillim-dong, san 56-1 Seoul 151-747, Republic of Korea*

Draft date July 18, 2006

1 Listing 1. Implementation for inequivalent replicas.

This listing presents code for an efficient implementation of the WHAM algorithm described in the text for estimating the expectation of observables and potentials of mean force for coordinates that have been discretized into a discrete set of states. Correlation functions are computed separately for individual replicas. This code was used to generate all the figures in the text.

```
!=============================================================================================
! A module to apply the weighted histogram analysis method (WHAM) to a number of independent
! canonical simulations, independent simulated tempering simulations, or parallel tempering
! simulations.  All simulations must be of the same length.
! NOTE: All 'real' variables should be compiled to double precision.
!=============================================================================================
module WHAM
  implicit none

  private
  public :: WHAM_initialize, WHAM_finalize, WHAM_computeWeights, WHAM_computeExpectation, &
       WHAM_computePMF, WHAM_getDimensionlessFreeEnergies

  !=============================================================================================
  ! Constants.
  !=============================================================================================
  integer, parameter :: M = 100          ! the number of energy bins
  real, parameter :: TOLERANCE = 1.0e-5 ! termination criterion for calculation of free energies
  integer, parameter :: MAXITS = 1000   ! maximum number of iterations to attempt

  !=============================================================================================
  ! Private module data.
  !=============================================================================================
  integer :: K ! the number of replicas
  integer :: L ! the number of temperatures
  integer :: N ! the number of snapshots per replica
  real, dimension(:), pointer :: beta_l
               ! beta_l(l) is the inverse temperature of temperature index l
  real, dimension(:), pointer :: U_m
               ! U_m(m) is the potential energy at the midpoint of energy bin m
  integer, dimension(:,:), pointer :: m_kn
               ! m_kn(k,n) is the potential energy bin of snapshot n of replica k
  real, dimension(:,:), pointer :: H_km
               ! H(k,m) is the histogram (number of counts) of energy bin m for replica k
  real, dimension(:,:), pointer :: g_km
               ! g_km(k,m) is the statistical inefficiency of energy bin m
  integer, dimension(:,:), pointer :: N_kl
               ! N_kl(k,l) is the number of visits to temperature index l by replica k
  real, dimension(:), pointer :: log_Heff_m
               ! log_Heff_m(m) is the log the effective number of independent counts of energy
               ! bin m across all replicas
  real, dimension(:,:), pointer :: log_Neff_lm
               ! log_Neff_lm(l,m) is the log the effective number of independent visits to
               ! temperature level l
  real, dimension(:), pointer :: log_Omega_m
               ! logOmega_m(m) is the log density of states of energy bin m
  real, dimension(:), pointer :: f_l
```

[*]Author to whom correspondence is to be addressed: John D. Chodera <jchodera@gmail.com>

```
                ! f_l(l) is the dimensionless free energy of temperature index l
  real, dimension(:,:), pointer :: w_kn
                ! Storage for weights.  w_kn(k,n) is the weight of snapshot n of replica k.
  !========================================================================================

contains

  !========================================================================================
  ! Initialize the WHAM module.
  !========================================================================================
  subroutine WHAM_initialize(U_kn, beta_l_arg, temp_index_kn, given_f_l)
    ! Arguments
    real, dimension(:,:), intent(in) :: U_kn
            ! U_kn(k,n) is the potential energy of snapshot n of replica k
    real, dimension(:), intent(in) :: beta_l_arg
            ! beta_l(l) is the inverse temperature of temperature index l
    integer, dimension(:,:), intent(in) :: temp_index_kn
            ! temp_index_kn(k,n) is the temperature index (from 1..L) of snapshot n of replica k
    real, dimension(:), intent(in), optional :: given_f_l
            ! given_f_l(l) is the (optional) given dimensionless free energy.
            ! If provided, this bypasses the calculation of f_l by self-consistent iteration.

    ! Determine the number of replicas, snapshots/replica, and temperatures from the input arguments.
    K = size(U_kn, 1)
    N = size(U_kn, 2)
    L = size(beta_l_arg, 1)
    write(*,*) 'K = ', K, ', N = ', N, ', L = ', L

    ! Allocate storage.
    allocate(beta_l(L), U_m(M), m_kn(K,N), H_km(K,M), g_km(K,M), N_kl(K,L), log_Heff_m(M))
    allocate(log_Neff_lm(L,M), log_Omega_m(M), f_l(L), w_kn(K,N))

    ! Make local copy of the inverse temperatures.
    beta_l(1:L) = beta_l_arg(1:L)

    ! Set up histograms.
    call constructHistograms(U_kn)

    ! Compute energy bin statistical inefficiencies.
    call computeBinStatInefficiencies(m_kn, M, g_km)

    ! Compute effective number of bin counts.
    call computeEffectiveCounts(temp_index_kn)

    ! Compute dimensionless free energies.
    if(present(given_f_l)) then
       write(*,*) 'Bypassing computation of dimensionless free energies to use provided f_l.'
       f_l = given_f_l
       write(*,*) f_l

       ! Compute log density of states using the current estimate of f.
       call computeLogDensityOfStates()
    else
       call computeFreeEnergies()
    end if

  end subroutine WHAM_initialize

  !========================================================================================
  ! Finalize the WHAM module and delete all data.
  !========================================================================================
  subroutine WHAM_finalize()
    deallocate(beta_l, U_m, m_kn, H_km, g_km, N_kl, log_Heff_m, log_Neff_lm, log_Omega_m, f_l, w_kn)
  end subroutine WHAM_finalize

  !========================================================================================
  ! Retrieve dimensionless free energies.
  !========================================================================================
  subroutine WHAM_getDimensionlessFreeEnergies(fout_l)

    ! Arguments.
    real, dimension(K), intent(out) :: fout_l ! f_l(l) is the dimensionless free energy of temperature l

    ! Assign output.
    fout_l = f_l

  end subroutine WHAM_getDimensionlessFreeEnergies

  !========================================================================================
```

```
! Construct energy bins, bin assignments, and histogram counts.
!=============================================================================================
subroutine constructHistograms(U_kn)
  ! Arguments
  real, dimension(:,:), intent(in) :: U_kn
                       ! U_kn(k,n) is the potential energy of snapshot n of replica k

  ! Constants
  real, parameter :: EPSILON = 1.0e-3
                       ! epsilon is a value larger than the machine precision

  ! Local variables
  real :: U_min, U_max ! the minimum and maximum potential energies
  real :: DeltaU       ! the energy bin width
  integer :: mIndex    ! energy bin index
  integer :: kIndex    ! replica index
  integer :: nIndex    ! snapshot index

  ! Compute bin spacing.
  U_min = minval(U_kn)
  U_max = maxval(U_kn)
  DeltaU = (U_max - U_min + EPSILON) / M

  ! Compute the energies at midpoints of the energy bins.
  forall(mIndex = 1:M)
     U_m(mIndex) = U_min + DeltaU * (mIndex - 0.5)
  end forall

  ! Assign snapshots to energy bins.
  m_kn = floor((U_kn - U_min) / DeltaU) + 1

  ! Compute histogram counts.
  forall(kIndex = 1:K)
     H_km(kIndex,1:M) = histogram(m_kn(kIndex,1:N), M)
  end forall

end subroutine constructHistograms

!=============================================================================================
! Compute effective numbers of statistically independent counts to make subsequent estimation
! of the density of states very rapid.
!=============================================================================================
subroutine computeEffectiveCounts(temp_index_kn)

  ! Arguments
  integer, dimension(K,N), intent(in) :: temp_index_kn
         ! temp_index_kn(k,n) is the temperature index (from 1..L) of snapshot n of replica k

  ! Constants
  real, parameter :: LOG_ZERO = -7 ! the log of some number much less than unity

  ! Local variables
  integer :: kIndex ! replica index
  integer :: nIndex ! snapshot index
  integer :: mIndex ! energy bin index
  integer :: lIndex ! temperature index
  real, dimension(M) :: Heff_m
                    ! Heff_m(m) is the effective histogram count for energy bin m
  real, dimension(L,M) :: Neff_lm
                    ! Neff_lm(l,m) is the effective number of independent visits to
                    ! temperature l using the correlation time from energy bin m

  ! Compute number of visits to temperature l by replica k.
  N_kl = 0
  forall(kIndex = 1:K)
     N_kl(kIndex,1:L) = histogram(temp_index_kn(kIndex,1:N), L)
  end forall

  ! Compute effective number of independent energy bin counts summed across all replicas.
  forall(mIndex = 1:M)
     Heff_m(mIndex) = sum( H_km(1:K,mIndex) / g_km(1:K,mIndex) )
  end forall
  ! Store the log.
  log_Heff_m = LOG_ZERO
  where(Heff_m > 0) log_Heff_m = log(Heff_m)

  ! Compute effective number of snapshots at each temperature from each replica.
  forall(lIndex = 1:L, mIndex = 1:M)
     Neff_lm(lIndex,mIndex) = sum( N_kl(1:K,lIndex) / g_km(1:K,mIndex) )
```

```
  end forall
  ! Store the log.
  log_Neff_lm = LOG_ZERO
  where(Neff_lm > 0) log_Neff_lm = log(Neff_lm)

end subroutine computeEffectiveCounts

!=======================================================================================================
! Return the histogram count.
!=======================================================================================================
pure function histogram(vector, nbins)
  ! Arguments.
  integer, dimension(:), intent(in) :: vector
             ! the vector of entries to tally the histogram for, which must be in 1..nbins
  integer, intent(in) :: nbins           ! the number of bins in the histogram
  integer, dimension(nbins) :: histogram ! the resulting histogram

  ! Local variables.
  integer :: index
  integer :: bin_index

  ! Construct histogram count.
  histogram = 0
  do index = 1,size(vector)
     bin_index = vector(index)
     histogram(bin_index) = histogram(bin_index) + 1
  end do

end function histogram

!=======================================================================================================
! Compute statistical inefficiencies for a binned observable, such as energy.
!=======================================================================================================
subroutine computeBinStatInefficiencies(m_kn, M, g_km)

  ! Arguments.
  integer, dimension(:,:), intent(in) :: m_kn ! m_kn(k,n) is the value of a binned observable
                       ! for snapshot n of replica k, and can only assume a value of 1..M.
  integer :: M         ! the number of bins
  real, dimension(:,:), intent(out) :: g_km ! g_km(k,m) is the statistical inefficiency for
                       ! bin m from replica k

  ! Local variables.
  real, dimension(K,M) :: C_km
                       ! C_km(k,m) is the correlation function for energy bin m of replica k
                       ! at a single lag time
  integer :: t         ! lag time
  integer :: t0        ! time origin index
  real, dimension(K,M) :: Epsi_km
                       ! Epsi_km(k,m) is the expectation of the indicator function psi_m for
                       ! energy bin m of replica k
  logical, dimension(K,M) :: has_crossed_zero_km
                       ! has_crossed_zero_km(k,m) is true if the correlation function for bin
                       ! m from replica k has crossed zero
  integer :: kIndex    ! replica index
  integer :: nIndex    ! snapshot index
  integer :: m1Index, m2Index ! energy bin indices
  integer :: increment ! the amount by which t is incremented each iteration

  write(*,*) 'Computing energy bin statistical inefficiencies...'

  ! Compute expectation of psi_m for each replica and energy bin.
  Epsi_km = 0
  forall(kIndex = 1:K)
     Epsi_km(kIndex,1:M) = histogram(m_kn(kIndex,1:N), M) / real(N)
  end forall

  ! Accumulate the integrated correlation time by computing the normalized correlation time at
  ! increasing values of t.  Stop accumulating if the correlation function goes negative, since
  ! this is unlikely to occur unless the correlation function has decayed to the point where it
  ! is dominated by noise and indistinguishable from zero.
  g_km = 1
  has_crossed_zero_km = .false.
  t = 1
  increment = 1
  do while(t < N-1)
     ! Compute unnormalized correlation function for time t.
     C_km = 0
     do kIndex = 1,K
```

```
        do t0 = 1,(N-t)
           ! a contribution is only made if the replica is in a bin at time t0 and t0+t
           m1Index = m_kn(kIndex,t0)
           m2Index = m_kn(kIndex,t0+t)
           if( m1Index == m2Index ) then
              C_km(kIndex,m1Index) = C_km(kIndex,m1Index) + 1
           end if
        end do
     end do
     C_km = C_km / real(N-t)

     ! Compute normalized fluctuation correlation functions from unnormalized correlation functions.
     where (Epsi_km /= 0.0) C_km = (C_km - Epsi_km**2) / (Epsi_km - Epsi_km**2)

     ! Note those correlation functions that have crossed zero.
     where (C_km < 0) has_crossed_zero_km = .true.

     ! Accumulate statistical inefficiencies for those correlation functions that have not yet
     ! crossed zero.
     where (.not. has_crossed_zero_km) g_km = g_km + 2.0 * C_km * (1.0 - real(t)/real(N)) * increment

     ! Terminate if all correlation functions have crossed zero.
     if (all(has_crossed_zero_km)) exit

     ! Increment t and the amount by which we increment t.
     t = t + increment
     increment = increment + 1
  end do

end subroutine computeBinStatInefficiencies

!=======================================================================================================
! Compute the log of a sum of terms whose logarithms are provided.
!=======================================================================================================
function logSum(arg_i)

  ! Arguments
  real, dimension(:), intent(in) :: arg_i ! the logs of the terms to be summed
  real :: logSum                          ! the log of the sum of the exponentials of arg_i

  ! Local variables
  real :: max_arg

  ! Compute the maximum argument.
  max_arg = maxval(arg_i)

  ! Compute the log sum
  logSum = log( sum( exp(arg_i - max_arg) ) ) + max_arg

end function logSum

!=======================================================================================================
! Compute the log density of states from the current dimensionless free energies.
! The log potential energy of states is computed instead of the density of states directly
! because it is more numerically stable.
!=======================================================================================================
subroutine computeLogDensityOfStates()

  ! Local variables
  integer :: mIndex       ! energy bin index
  real :: log_numerator   ! log of the numerator
  real :: log_denominator ! log of the denominator

  ! For each energy bin, compute the log density of states.
  do mIndex = 1,M
     ! Compute log numerator.
     log_numerator = log_Heff_m(mIndex);

     ! Compute log denominator.
     log_denominator = logSum(log_Neff_lm(1:L,mIndex) + f_l(1:L) - beta_l(1:L) * U_m(mIndex))

     ! Compute the log density of states.
     log_Omega_m(mIndex) = log_numerator - log_denominator
  end do

end subroutine ComputeLogDensityOfStates

!=======================================================================================================
! Compute dimensionless free energies by self-consistent iteration.
```

```
!=======================================================================================================
subroutine computeFreeEnergies()

  ! Local variables
  integer :: iteration            ! iteration counter
  real, dimension(L) :: f_l_old   ! old copy of dimensionless free energies f_l
  real, dimension(L) :: Delta_f_l ! change in dimensionless free energies from last iteration
  real :: max_delta               ! maximum relative change in this iteration
  integer :: lIndex               ! temperature index

  ! Set the initial estimate of dimensionless free energies to zero.
  f_l = 0

  ! Iterate until maximum number of allowed iterations has been exceeded.
  write(*,*) 'Computing dimensionless free energies...'
  do iteration = 1,MAXITS
     ! Store a copy of the dimensionless free en
     f_l_old = f_l

     ! Compute log density of states using the current estimate of f.
     call computeLogDensityOfStates()

     ! For each temperature, compute a new estimate of the dimensionless free energy.
     do lIndex = 1,L
        f_l(lIndex) = - logSum( log_Omega_m(1:M) - beta_l(lIndex) * U_m(1:M) )
     end do

     ! Shift free energies such that f(1) = 0.
     f_l = f_l - f_l(1)

     ! Check convergence of the dimensionless free energies, and terminate loop if achieved.
     ! Terminate when max((f - fold) / f) < tolerance for all nonzero f.
     Delta_f_l = f_l - f_l_old
     max_delta = abs(maxval(delta_f_l(2:L) / f_l(2:L)))
     if(max_delta < TOLERANCE) exit
  end do

  ! Report convergence, or warn user if not achieved.
  if(iteration <= MAXITS) then
     write(*,'(A,es9.2,A,i8,A)') 'Converged to tolerance of ', max_delta, ' in ', &
          iteration, ' iterations.'
  else
     write(*,*) 'WARNING: Did not converge to within specified tolerance.'
     write(*,'(A,es9.2,A,es9.2,A,i8)') 'max_delta = ', max_delta, ', TOLERANCE = ', &
          TOLERANCE, ', MAXITS = ', MAXITS
  end if
  write(*,*) 'f_l = '
  write(*,*) f_l

end subroutine computeFreeEnergies

!=======================================================================================================
! Compute snapshot weights for inverse temperature beta.
!=======================================================================================================
subroutine WHAM_computeWeights(beta, w_kn)

  ! Arguments.
  real, intent(in) :: beta ! the inverse temperature at which the weights are desired
  real, dimension(:,:), intent(out) :: w_kn ! w_kn(k,n) is the weight for snapshot n of replica k

  ! Local variables.
  integer :: kIndex ! replica index
  integer :: nIndex ! snapshot index
  integer :: mIndex ! energy bin index

  ! First, compute the unnormalized log weights from the density of states.
  do kIndex = 1,K
     do nIndex = 1,N
        ! Get energy bin that this snapshot falls in.
        mIndex = m_kn(kIndex,nIndex)
        ! Compute unnormalized log weight.
        w_kn(kIndex,nIndex) = log_Omega_m(mIndex) - beta * U_m(mIndex) - log(sum(H_km(1:K,mIndex)))
     end do
  end do

  ! Compute unnormalized weights from log weights, making sure to avoid overflow.
  w_kn = exp(w_kn - maxval(w_kn))

  ! Normalize.
```

```
  w_kn = w_kn / sum(w_kn)

end subroutine WHAM_computeWeights

!=============================================================================================
! Compute the (cross) statistical inefficiency of two timeseries.
!=============================================================================================
function statisticalInefficiency(A_n, B_n)

  ! Arguments.
  real, dimension(:), intent(in) :: A_n ! timeseries of length N
  real, dimension(:), intent(in) :: B_n ! timeseries of length N
  real :: statisticalInefficiency ! the statistical inefficiency

  ! Local variables.
  integer :: N ! length of timeseries
  integer :: t ! lag time
  integer :: t0 ! time origin
  integer :: increment ! amount by which lag time is to be incremented each step
  real :: mu_A ! mean of timeseries A
  real :: mu_B ! mean of timeseries B
  real :: sigma2_AB ! covariance
  real :: C ! value of the correlation function at current lag time

  ! Get the length of the timeseries.
  N = size(A_n)

  ! Initialize statistical inefficiency estimate with uncorrelated value.
  statisticalInefficiency = 1

  ! Compute means and variance.
  mu_A = sum(A_n) / N
  mu_B = sum(B_n) / N
  sigma2_AB = sum((A_n-mu_A) * (B_n-mu_B)) / (N-1.0)
  if(sigma2_AB == 0) return ! We cannot compute the statstical inefficiency if the variance is zero.

  ! Accumulate the integrated correlation time by computing the normalized correlation time at
  ! increasing values of t.  Stop accumulating if the correlation function goes negative, since
  ! this is unlikely to occur unless the correlation function has decayed to the point where it
  ! is dominated by noise and indistinguishable from zero.
  t = 1
  increment = 1
  do while(t < N-1)
     ! Compute unnormalized correlation function for time t.
     C = sum( A_n(1:(N-t))*B_n((1+t):N) + B_n(1:(N-t))*A_n((1+t):N) ) / (2.0 * (N-t))

     ! Compute normalized fluctuation correlation functions from unnormalized correlation functions.
     C = (C - mu_A*mu_B) / sigma2_AB

     ! Terminate if the correlation function has crossed zero.
     if(C <= 0) exit

     ! Accumulate contribution to the statistical inefficiency.
     statisticalInefficiency = statisticalInefficiency + 2.0 * C * (1.0 - real(t)/real(N)) * increment

     ! Increment t and the amount by which we increment t.
     t = t + increment
     increment = increment + 1
  end do

end function statisticalInefficiency

!=============================================================================================
! Compute expectation and uncertainty.
!=============================================================================================
subroutine WHAM_computeExpectation(beta, A_kn, A, dA)

  ! Arguments.
  real, intent(in) :: beta   ! the inverse temperature of interest
  real, dimension(:,:), intent(in) :: A_kn
       ! A_kn(k,n) is the value of the observable of interest A for snapshot n of replica k
  real, intent(out) :: A  ! the expectation of the observable A at inverse temperature beta
  real, intent(out) :: dA ! the uncertainty in the expectation of A

  ! Local variables.
  real :: g_Aw_Aw, g_Aw_w, g_w_w ! statistical inefficiencies
  real :: max_g_Aw_w ! maximum allowed value of g_Aw_w
  real :: mu_Aw, mu_w ! means
  real :: sigma2_Aw_Aw, sigma2_Aw_w, sigma2_w_w ! variances
```

```
  integer :: kIndex ! replica index
  real :: X ! expectation of numerator
  real :: Y ! expectation of denominator
  real :: d2X ! square uncertainty in numerator
  real :: d2Y ! square uncertainty in denominator
  real :: dXdY ! cross-uncertainty between numerator and denominator
  real, dimension(:), allocatable :: Aw_n, w_n

  ! Compute weights.
  call WHAM_computeWeights(beta, w_kn)

  ! Allocate storage.
  allocate(Aw_n(N), w_n(N))

  ! Accumulate contributions to expectation and uncertainty from each replica.
  X = 0; Y = 0; d2X = 0; d2Y = 0; dXdY = 0
  do kIndex = 1,K
     ! Compute modified replica timeseries.
     Aw_n = A_kn(kIndex,1:N) * w_kn(kIndex,1:N)
     w_n = w_kn(kIndex,1:N)

     ! Compute means and (co)variances of Aw and w timeseries.
     mu_Aw = sum(Aw_n) / N
     mu_w  = sum(w_n) / N
     sigma2_Aw_Aw = sum((Aw_n-mu_Aw)**2) / (N-1)
     sigma2_Aw_w = sum((Aw_n-mu_Aw)*(w_n-mu_w)) / (N-1)
     sigma2_w_w = sum((w_n-mu_w)**2) / (N-1)

     ! Compute statistical inefficiencies.
     g_Aw_Aw = statisticalInefficiency(Aw_n, Aw_n)
     g_Aw_w  = statisticalInefficiency(Aw_n, w_n)
     g_w_w   = statisticalInefficiency(w_n, w_n)
     ! Ensure that estimate of cross-uncertainty is reasonable (see Appendix).
     max_g_Aw_w = sqrt(g_Aw_Aw * g_w_w) * abs( sqrt(sigma2_Aw_Aw * sigma2_w_w) / sigma2_Aw_w )
     if( (g_Aw_w > max_g_Aw_w) .and. (max_g_Aw_w > 1) ) g_Aw_w = max_g_Aw_w

     ! Accumulate contributions to numerator and denominator.
     X = X + mu_Aw
     Y = Y + mu_w

     ! Accumulate contributions to uncertainty.
     d2X  = d2X  + sigma2_Aw_Aw / (N / g_Aw_Aw);
     d2Y  = d2Y  + sigma2_w_w   / (N / g_w_w);
     dXdY = dXdY + sigma2_Aw_w  / (N / g_Aw_w);
  end do

  ! Compute expectation and uncertainty.
  A = X/Y
  dA = sqrt( A**2 * (d2X/(X**2) + d2Y/(Y**2) - 2*dXdY/(X*Y)) )

  ! Clean up.
  deallocate(Aw_n, w_n)

end subroutine WHAM_computeExpectation

!=============================================================================================
! Compute free energies of an observable that only assumes discrete values.
! Useful for computing a potential of mean force.
!=============================================================================================
subroutine WHAM_computePMF(beta, state_kn, nstates, F_i, dF_i)

  ! Arguments.
  real, intent(in) :: beta   ! the inverse temperature of interest
  integer, dimension(:,:), intent(in) :: state_kn
       ! state_kn(k,n) is the state (1..nstates) occupied by snapshot n of replica k
  integer, intent(in) :: nstates ! the number of states
  real, dimension(:), intent(out) :: F_i  ! estimate of the free energy of state i
  real, dimension(:), intent(out) :: dF_i ! the uncertainty in the free energy of state i

  ! Local variables.
  integer :: kIndex ! replica index
  integer :: nIndex ! snapshot index
  integer :: iIndex, jIndex ! state index
  real, dimension(:), allocatable :: P_i, dP_i
       ! P_i(i) is the probability that state i is occupied, and dP_i(i) is uncertainty
  real, dimension(:), allocatable :: X_i, Y_i, d2X_i, d2Y_i, dXdY_i
  real, dimension(:), allocatable :: mu_Aw_i, sigma2_Aw_Aw_i, sigma2_Aw_w_i
  real, dimension(:), allocatable :: g_Aw_Aw_i, g_Aw_w_i, max_g_Aw_w_i
  real :: mu_w, sigma2_w_w, g_w_w, w
```

```
real, dimension(:), allocatable :: C_Aw_Aw_i, C_Aw_w_i
logical, dimension(:), allocatable :: has_crossed_zero_Aw_Aw_i, has_crossed_zero_Aw_w_i
integer :: t, t0, increment

! Compute weights.
call WHAM_computeWeights(beta, w_kn)

! Accumulate contributions to expectation and uncertainty from each replica.
allocate(P_i(nstates), dP_i(nstates))
allocate(X_i(nstates),Y_i(nstates),d2X_i(nstates),d2Y_i(nstates),dXdY_i(nstates))
allocate(mu_Aw_i(nstates),sigma2_Aw_Aw_i(nstates),sigma2_Aw_w_i(nstates))
allocate(g_Aw_Aw_i(nstates),g_Aw_w_i(nstates),max_g_Aw_w_i(nstates))
allocate(C_Aw_Aw_i(nstates),C_Aw_w_i(nstates))
allocate(has_crossed_zero_Aw_Aw_i(nstates),has_crossed_zero_Aw_w_i(nstates))
X_i = 0; Y_i = 0; d2X_i = 0 ; d2Y_i = 0; dXdY_i = 0
do kIndex = 1,K
   ! Compute means and (co)variances.
   mu_w = sum(w_kn(kIndex,1:N)) / N
   sigma2_w_w = sum(w_kn(kIndex,1:N)**2)/N - mu_w**2

   mu_Aw_i = 0; sigma2_Aw_Aw_i = 0; sigma2_Aw_w_i = 0
   do nIndex = 1,N
      iIndex = state_kn(kIndex,nIndex)
      w = w_kn(kIndex,nIndex)
      mu_Aw_i(iIndex) = mu_Aw_i(iIndex) + w
      sigma2_Aw_Aw_i(iIndex) = sigma2_Aw_Aw_i(iIndex) + w**2
      sigma2_Aw_w_i(iIndex) = sigma2_Aw_w_i(iIndex) + w**2
   end do
   mu_Aw_i = mu_Aw_i / N
   sigma2_Aw_Aw_i = sigma2_Aw_Aw_i/N - mu_Aw_i*mu_Aw_i
   sigma2_Aw_w_i = sigma2_Aw_w_i/N - mu_Aw_i*mu_w

   ! Compute statistical inefficiencies.
   g_w_w = statisticalInefficiency(w_kn(kIndex,1:N),w_kn(kIndex,1:N))

   ! Accumulate the integrated correlation time by computing the normalized correlation time at
   ! increasing values of t.  Stop accumulating if the correlation function goes negative, since
   ! this is unlikely to occur unless the correlation function has decayed to the point where it
   ! is dominated by noise and indistinguishable from zero.
   g_Aw_Aw_i = 1 ; g_Aw_w_i = 1
   t = 1; increment = 1
   has_crossed_zero_Aw_Aw_i = .false.; has_crossed_zero_Aw_w_i = .false.
   do while(t < N-1)
      ! Compute unnormalized correlation function for time t.
      C_Aw_Aw_i = 0; C_Aw_w_i = 0
      do t0 = 1,(N-t)
         ! a contribution is only made if the replica is in a bin at time t0 and t0+t
         iIndex = state_kn(kIndex,t0)
         jIndex = state_kn(kIndex,t0+t)
         w = w_kn(kIndex,t0)*w_kn(kIndex,t0+t)
         if( iIndex == jIndex ) then
            C_Aw_Aw_i(iIndex) = C_Aw_Aw_i(iIndex) + w
         end if
         C_Aw_w_i(iIndex) = C_Aw_w_i(iIndex) + 0.5 * w
         C_Aw_w_i(jIndex) = C_Aw_w_i(jIndex) + 0.5 * w

      end do
      C_Aw_Aw_i = C_Aw_Aw_i / (N-t)
      C_Aw_w_i = C_Aw_w_i / (N-t)

      ! Compute normalized fluctuation correlation functions from unnormalized correlation functions.
      where (sigma2_Aw_Aw_i /= 0) C_Aw_Aw_i = (C_Aw_Aw_i - mu_Aw_i**2) / sigma2_Aw_Aw_i
      where (sigma2_Aw_w_i /= 0) C_Aw_w_i = (C_Aw_w_i - mu_Aw_i*mu_w) / sigma2_Aw_w_i

      ! Note those correlation functions that have crossed zero.
      where (C_Aw_Aw_i <= 0) has_crossed_zero_Aw_Aw_i = .true.
      where (C_Aw_w_i <= 0) has_crossed_zero_Aw_w_i = .true.

      ! Accumulate statistical inefficiencies for those correlation functions that have not yet
      ! crossed zero.
      where (.not. has_crossed_zero_Aw_Aw_i) &
           g_Aw_Aw_i = g_Aw_Aw_i + 2 * C_Aw_Aw_i * (1.0 - real(t)/real(N)) * increment
      where (.not. has_crossed_zero_Aw_w_i) &
           g_Aw_w_i = g_Aw_w_i + 2 * C_Aw_w_i * (1.0 - real(t)/real(N)) * increment

      ! Terminate if all correlation functions have crossed zero.
      if (all(has_crossed_zero_Aw_Aw_i) .and. all(has_crossed_zero_Aw_w_i)) exit

      ! Increment t and the amount by which we increment t.
```

```
            t = t + increment
            increment = increment + 1
         end do

         ! Ensure that estimate of cross-uncertainty is reasonable (see Appendix).
         max_g_Aw_w_i = sqrt(g_Aw_Aw_i * g_w_w) * abs( sqrt(sigma2_Aw_Aw_i * sigma2_w_w) / sigma2_Aw_w_i )
         where( (g_Aw_w_i > max_g_Aw_w_i) .and. (max_g_Aw_w_i > 1) ) g_Aw_w_i = max_g_Aw_w_i

         ! Accumulate contributions to numerator and denominator of expectation.
         X_i = X_i + mu_Aw_i
         Y_i = Y_i + mu_w

         ! Accumulate contributions to uncertainties in numerator and denominator.
         d2X_i  = d2X_i + sigma2_Aw_Aw_i / (N / g_Aw_Aw_i);
         d2Y_i  = d2Y_i + sigma2_w_w / (N / g_w_w);
         dXdY_i = dXdY_i + sigma2_Aw_w_i / (N / g_Aw_w_i);
      end do

      ! Compute expectation and uncertainties.
      P_i = X_i / Y_i
      dP_i = sqrt( P_i**2 * (d2X_i/(X_i**2) + d2Y_i/(Y_i**2) - 2*dXdY_i/(X_i*Y_i)) )

      ! Convert probabilities to free energies.
      F_i = - (1/beta) * log(P_i)
      dF_i = (1/beta) * (dP_i / P_i)

      ! Shift F_i so that min_i F_i = 0.
      F_i = F_i - minval(F_i)

      ! Clean up.
      deallocate(X_i,Y_i,d2X_i,d2Y_i,dXdY_i)
      deallocate(mu_Aw_i,sigma2_Aw_Aw_i,sigma2_Aw_w_i)
      deallocate(g_Aw_Aw_i,g_Aw_w_i,max_g_Aw_w_i)
      deallocate(C_Aw_Aw_i,C_Aw_w_i,has_crossed_zero_Aw_Aw_i,has_crossed_zero_Aw_w_i)
      deallocate(P_i, dP_i)

   end subroutine WHAM_computePMF

end module WHAM
```

2 Listing 2. Implementation for equivalent replicas.

As discussed in the text, in the limit of long simulations, each replica should execute an equivalent random walk. In this case, the estimates for means, variances, and correlation functions can be improved by averaging over all replicas. This code was not used to generate figures in the text, and is provided for reference only.

```
!=============================================================================================
! A module to apply the weighted histogram analysis method (WHAM) to a number of independent
! canonical simulations, independent simulated tempering simulations, or parallel tempering
! simulations.  All simulations must be of the same length.
! NOTE: reals should be compiled to double precision.
! NOTE: This version assumes all replicas have been run long enough so as to be equivalent
! for the purposes of computing correlation functions.
!=============================================================================================
module WHAM
   implicit none

   private
   public :: WHAM_initialize, WHAM_finalize, WHAM_computeWeights, WHAM_computeExpectation, &
        WHAM_computePMF

   !=============================================================================================
   ! Constants.
   !=============================================================================================
   integer, parameter :: M = 100         ! the number of energy bins
   real, parameter :: TOLERANCE = 1.0e-5 ! termination criterion for calculation of free energies
   integer, parameter :: MAXITS = 1000   ! maximum number of iterations to attempt

   !=============================================================================================
   ! Private module data.
   !=============================================================================================
   integer :: K ! the number of replicas
   integer :: L ! the number of temperatures
   integer :: N ! the number of snapshots per replica
   real, dimension(:), pointer :: beta_l
                ! beta_l(l) is the inverse temperature of temperature index l
   real, dimension(:), pointer :: U_m
```

```
                ! U_m(m) is the potential energy at the midpoint of energy bin m
  integer, dimension(:,:), pointer :: m_kn
                ! m_kn(k,n) is the potential energy bin of snapshot n of replica k
  real, dimension(:,:), pointer :: H_km
                ! H(k,m) is the histogram (number of counts) of energy bin m for replica k
  real, dimension(:), pointer :: g_m
                ! g_m(m) is the statistical inefficiency of energy bin m
  integer, dimension(:,:), pointer :: N_kl
                ! N_kl(k,l) is the number of visits to temperature index l by replica k
  real, dimension(:), pointer :: log_Heff_m
                ! log_Heff_m(m) is the log the effective number of independent counts of energy
                ! bin m across all replicas
  real, dimension(:,:), pointer :: log_Neff_lm
                ! log_Neff_lm(l,m) is the log the effective number of independent visits to
                ! temperature level l
  real, dimension(:), pointer :: log_Omega_m
                ! logOmega_m(m) is the log density of states of energy bin m
  real, dimension(:), pointer :: f_l
                ! f_l(l) is the dimensionless free energy of temperature index l
  real, dimension(:,:), pointer :: w_kn
                ! Storage for weights.  w_kn(k,n) is the weight of snapshot n of replica k.
  !======================================================================================================

contains

  !======================================================================================================
  ! Initialize the WHAM module.
  !======================================================================================================
  subroutine WHAM_initialize(U_kn, beta_l_arg, temp_index_kn)
    ! Arguments
    real, dimension(:,:), intent(in) :: U_kn
          ! U_kn(k,n) is the potential energy of snapshot n of replica k
    real, dimension(:), intent(in) :: beta_l_arg
          ! beta_l(l) is the inverse temperature of temperature index l
    integer, dimension(:,:), intent(in) :: temp_index_kn
          ! temp_index_kn(k,n) is the temperature index (from 1..L) of snapshot n of replica k

    ! Determine the number of replicas, snapshots/replica, and temperatures from the input arguments.
    K = size(U_kn, 1)
    N = size(U_kn, 2)
    L = size(beta_l_arg, 1)

    ! Allocate storage.
    allocate(beta_l(L), U_m(M), m_kn(K,N), H_km(K,M), g_m(M), N_kl(K,L), log_Heff_m(M))
    allocate(log_Neff_lm(L,M), log_Omega_m(M), f_l(L), w_kn(K,N))

    ! Make local copy of the inverse temperatures.
    beta_l(1:L) = beta_l_arg(1:L)

    ! Set up histograms.
    call constructHistograms(U_kn)

    ! Compute energy bin statistical inefficiencies.
    call computeBinStatInefficiencies(m_kn, M, g_m)

    ! Compute effective number of bin counts.
    call computeEffectiveCounts(temp_index_kn)

    ! Compute dimensionless free energies.
    call computeFreeEnergies()

  end subroutine WHAM_initialize

  !======================================================================================================
  ! Finalize the WHAM module and delete all data.
  !======================================================================================================
  subroutine WHAM_finalize()
    deallocate(beta_l, U_m, m_kn, H_km, g_m, N_kl, log_Heff_m, log_Neff_lm, log_Omega_m, f_l, w_kn)
  end subroutine WHAM_finalize

  !======================================================================================================
  ! Construct energy bins, bin assignments, and histogram counts.
  !======================================================================================================
  subroutine constructHistograms(U_kn)
    ! Arguments
    real, dimension(:,:), intent(in) :: U_kn
                          ! U_kn(k,n) is the potential energy of snapshot n of replica k

    ! Constants
```

```
  real, parameter :: EPSILON = 1.0e-3
                       ! epsilon is a value larger than the machine precision

  ! Local variables
  real :: U_min, U_max ! the minimum and maximum potential energies
  real :: DeltaU       ! the energy bin width
  integer :: mIndex    ! energy bin index
  integer :: kIndex    ! replica index
  integer :: nIndex    ! snapshot index

  ! Compute bin spacing.
  U_min = minval(U_kn)
  U_max = maxval(U_kn)
  DeltaU = (U_max - U_min + EPSILON) / M

  ! Compute the energies at midpoints of the energy bins.
  forall(mIndex = 1:M)
     U_m(mIndex) = U_min + DeltaU * (mIndex - 0.5)
  end forall

  ! Assign snapshots to energy bins.
  m_kn = floor((U_kn - U_min) / DeltaU) + 1

  ! Compute histogram counts.
  forall(kIndex = 1:K)
     H_km(kIndex,1:M) = histogram(m_kn(kIndex,1:N), M)
  end forall

end subroutine constructHistograms

!=============================================================================================
! Compute effective numbers of statistically independent counts to make subsequent estimation
! of the density of states very rapid.
!=============================================================================================
subroutine computeEffectiveCounts(temp_index_kn)

  ! Arguments
  integer, dimension(K,N), intent(in) :: temp_index_kn
         ! temp_index_kn(k,n) is the temperature index (from 1..L) of snapshot n of replica k

  ! Constants
  real, parameter :: LOG_ZERO = -7 ! the log of some number much less than unity

  ! Local variables
  integer :: kIndex ! replica index
  integer :: nIndex ! snapshot index
  integer :: mIndex ! energy bin index
  integer :: lIndex ! temperature index
  real, dimension(M) :: Heff_m
                    ! Heff_m(m) is the effective histogram count for energy bin m
  real, dimension(L,M) :: Neff_lm
                    ! Neff_lm(l,m) is the effective number of independent visits to
                    ! temperature l using the correlation time from energy bin m

  ! Compute number of visits to temperature l by replica k.
  N_kl = 0
  forall(kIndex = 1:K)
     N_kl(kIndex,1:L) = histogram(temp_index_kn(kIndex,1:N), L)
  end forall

  ! Compute effective number of independent energy bin counts summed across all replicas.
  forall(mIndex = 1:M)
     Heff_m(mIndex) = sum( H_km(1:K,mIndex) / g_m(mIndex) )
  end forall
  ! Store the log.
  log_Heff_m = LOG_ZERO
  where(Heff_m > 0) log_Heff_m = log(Heff_m)

  ! Compute effective number of snapshots at each temperature from each replica.
  forall(lIndex = 1:L, mIndex = 1:M)
     Neff_lm(lIndex,mIndex) = sum( N_kl(1:K,lIndex) / g_m(mIndex) )
  end forall
  ! Store the log.
  log_Neff_lm = LOG_ZERO
  where(Neff_lm > 0) log_Neff_lm = log(Neff_lm)

end subroutine computeEffectiveCounts

!=============================================================================================
```

```
! Return the histogram count.
!=============================================================================================
pure function histogram(vector, nbins)
  ! Arguments.
  integer, dimension(:), intent(in) :: vector
             ! the vector of entries to tally the histogram for, which must be in 1..nbins
  integer, intent(in) :: nbins           ! the number of bins in the histogram
  integer, dimension(nbins) :: histogram ! the resulting histogram

  ! Local variables.
  integer :: index
  integer :: bin_index

  ! Construct histogram count.
  histogram = 0
  do index = 1,size(vector)
     bin_index = vector(index)
     histogram(bin_index) = histogram(bin_index) + 1
  end do

end function histogram

!=============================================================================================
! Compute statistical inefficiencies for a binned observable, such as energy.
!=============================================================================================
subroutine computeBinStatInefficiencies(m_kn, M, g_m)

  ! Arguments.
  integer, dimension(:,:), intent(in) :: m_kn ! m_kn(k,n) is the value of a binned observable
                       ! for snapshot n of replica k, and can only assume a value of 1..M.
  integer :: M         ! the number of bins
  real, dimension(:), intent(out) :: g_m ! g_m(m) is the statistical inefficiency for
                       ! bin m from replica k

  ! Local variables.
  real, dimension(M) :: C_m
                       ! C_m(m) is the correlation function for energy bin m at a single lag time
  integer :: t         ! lag time
  integer :: t0        ! time origin index
  real, dimension(M) :: Epsi_m
                       ! Epsi_km(k,m) is the expectation of the indicator function psi_m for
                       ! energy bin m
  logical, dimension(M) :: has_crossed_zero_m
                       ! has_crossed_zero_km(k,m) is true if the correlation function for bin
                       ! m from replica k has crossed zero
  integer :: kIndex    ! replica index
  integer :: nIndex    ! snapshot index
  integer :: m1Index, m2Index ! energy bin indices
  integer :: increment ! the amount by which t is incremented each iteration

  write(*,*) 'Computing energy bin statistical inefficiencies...'

  ! Compute expectation of psi_m for each energy bin, assuming replicas are equivalent.
  Epsi_m = 0
  do kIndex = 1,K
     Epsi_m(1:M) = Epsi_m + histogram(m_kn(kIndex,1:N), M)
  end do
  Epsi_m = Epsi_m / real(K*N)

  ! Accumulate the integrated correlation time by computing the normalized correlation time at
  ! increasing values of t.  Stop accumulating if the correlation function goes negative, since
  ! this is unlikely to occur unless the correlation function has decayed to the point where it
  ! is dominated by noise and indistinguishable from zero.
  g_m = 1
  has_crossed_zero_m = .false.
  t = 1
  increment = 1
  do while(t < N-1)
     ! Compute unnormalized correlation function for time t.
     C_m = 0
     do kIndex = 1,K
        do t0 = 1,(N-t)
           ! a contribution is only made if the replica is in a bin at time t0 and t0+t
           m1Index = m_kn(kIndex,t0)
           m2Index = m_kn(kIndex,t0+t)
           if( m1Index == m2Index ) then
              C_m(m1Index) = C_m(m1Index) + 1
           end if
        end do
```

```
      end do
      C_m = C_m / real(K*(N-t))

      ! Compute normalized fluctuation correlation functions from unnormalized correlation functions.
      where (Epsi_m /= 0.0) C_m = (C_m - Epsi_m**2) / (Epsi_m - Epsi_m**2)

      ! Note those correlation functions that have crossed zero.
      where (C_m < 0) has_crossed_zero_m = .true.

      ! Accumulate statistical inefficiencies for those correlation functions that have not yet
      ! crossed zero.
      where (.not. has_crossed_zero_m) g_m = g_m + 2.0 * C_m * (1.0 - real(t)/real(N)) * increment

      ! Terminate if all correlation functions have crossed zero.
      if (all(has_crossed_zero_m)) exit

      ! Increment t and the amount by which we increment t.
      t = t + increment
      increment = increment + 1
   end do

end subroutine computeBinStatInefficiencies

!=============================================================================================
! Compute the log of a sum of terms whose logarithms are provided.
!=============================================================================================
function logSum(arg_i)

   ! Arguments
   real, dimension(:), intent(in) :: arg_i ! the logs of the terms to be summed
   real :: logSum                          ! the log of the sum of the exponentials of arg_i

   ! Local variables
   real :: max_arg

   ! Compute the maximum argument.
   max_arg = maxval(arg_i)

   ! Compute the log sum
   logSum = log( sum( exp(arg_i - max_arg) ) ) + max_arg

end function logSum

!=============================================================================================
! Compute the log density of states from the current dimensionless free energies.
! The log potential energy of states is computed instead of the density of states directly
! because it is more numerically stable.
!=============================================================================================
subroutine computeLogDensityOfStates()

   ! Local variables
   integer :: mIndex       ! energy bin index
   real :: log_numerator   ! log of the numerator
   real :: log_denominator ! log of the denominator

   ! For each energy bin, compute the log density of states.
   do mIndex = 1,M
      ! Compute log numerator.
      log_numerator = log_Heff_m(mIndex);

      ! Compute log denominator.
      log_denominator = logSum(log_Neff_lm(1:L,mIndex) + f_l(1:L) - beta_l(1:L) * U_m(mIndex))

      ! Compute the log density of states.
      log_Omega_m(mIndex) = log_numerator - log_denominator
   end do

end subroutine ComputeLogDensityOfStates

!=============================================================================================
! Compute dimensionless free energies by self-consistent iteration.
!=============================================================================================
subroutine computeFreeEnergies()

   ! Local variables
   integer :: iteration            ! iteration counter
   real, dimension(L) :: f_l_old   ! old copy of dimensionless free energies f_l
   real, dimension(L) :: Delta_f_l ! change in dimensionless free energies from last iteration
   real :: max_delta               ! maximum relative change in this iteration
```

```
  integer :: lIndex               ! temperature index

  ! Set the initial estimate of dimensionless free energies to zero.
  f_l = 0

  ! Iterate until maximum number of allowed iterations has been exceeded.
  write(*,*) 'Computing dimensionless free energies...'
  do iteration = 1,MAXITS
     ! Store a copy of the dimensionless free en
     f_l_old = f_l

     ! Compute log density of states using the current estimate of f.
     call computeLogDensityOfStates()

     ! For each temperature, compute a new estimate of the dimensionless free energy.
     do lIndex = 1,L
        f_l(lIndex) = - logSum( log_Omega_m(1:M) - beta_l(lIndex) * U_m(1:M) )
     end do

     ! Shift free energies such that f(1) = 0.
     f_l = f_l - f_l(1)

     ! Check convergence of the dimensionless free energies, and terminate loop if achieved.
     ! Terminate when max((f - fold) / f) < tolerance for all nonzero f.
     Delta_f_l = f_l - f_l_old
     max_delta = abs(maxval(delta_f_l(2:L) / f_l(2:L)))
     if(max_delta < TOLERANCE) exit
  end do

  ! Report convergence, or warn user if not achieved.
  if(iteration <= MAXITS) then
     write(*,'(A,es9.2,A,i8,A)') 'Converged to tolerance of ', max_delta, ' in ', &
          iteration, ' iterations.'
  else
     write(*,*) 'WARNING: Did not converge to within specified tolerance.'
     write(*,'(A,es9.2,A,es9.2,A,i8)') 'max_delta = ', max_delta, ', TOLERANCE = ', &
          TOLERANCE, ', MAXITS = ', MAXITS
  end if
  write(*,*) 'f_l = '
  write(*,*) f_l

end subroutine computeFreeEnergies

!=============================================================================================
! Compute snapshot weights for inverse temperature beta.
!=============================================================================================
subroutine WHAM_computeWeights(beta, w_kn)

  ! Arguments.
  real, intent(in) :: beta ! the inverse temperature at which the weights are desired
  real, dimension(:,:), intent(out) :: w_kn ! w_kn(k,n) is the weight for snapshot n of replica k

  ! Local variables.
  integer :: kIndex ! replica index
  integer :: nIndex ! snapshot index
  integer :: mIndex ! energy bin index

  ! First, compute the unnormalized log weights from the density of states.
  do kIndex = 1,K
     do nIndex = 1,N
        ! Get energy bin that this snapshot falls in.
        mIndex = m_kn(kIndex,nIndex)
        ! Compute unnormalized log weight.
        w_kn(kIndex,nIndex) = log_Omega_m(mIndex) - beta * U_m(mIndex) - log(sum(H_km(1:K,mIndex)))
     end do
  end do

  ! Compute unnormalized weights from log weights, making sure to avoid overflow.
  w_kn = exp(w_kn - maxval(w_kn))

  ! Normalize.
  w_kn = w_kn / sum(w_kn)

end subroutine WHAM_computeWeights

!=============================================================================================
! Compute the (cross) statistical inefficiency of two observables A and B evaluated for equivalent
! sets of replica trajectories.  The means of A and B, as well as the covariance, are also computed.
!=============================================================================================
```

```
subroutine computeStatisticalInefficiency(A_kn, B_kn, mu_A, mu_B, sigma2_AB, g)

  ! Arguments.
  real, dimension(:,:), intent(in) :: A_kn, B_kn
      ! A_kn(k,n) is value of the observable A for snapshot n of replica k
  real, intent(out) :: mu_A, mu_B                ! means of A and B
  real, intent(out) :: sigma2_AB                 ! the covariance
  real, intent(out) :: g                         ! the statistical inefficiency

  ! Local variables.
  integer :: K ! number of equivalent replicas
  integer :: N ! length of timeseries
  integer :: t ! lag time
  integer :: t0 ! time origin
  integer :: increment ! amount by which lag time is to be incremented each step
  real :: C ! value of the correlation function at current lag time

  ! Get the length of the timeseries.
  K = size(A_kn,1)
  N = size(A_kn,2)

  ! Initialize statistical inefficiency estimate with uncorrelated value.
  g = 1

  ! Compute means and variance.
  mu_A = sum(A_kn) / real(K*N)
  mu_B = sum(B_kn) / real(K*N)
  sigma2_AB = sum(A_kn * B_kn) / real(K*N) - mu_A * mu_B
  if(sigma2_AB == 0) return ! We cannot compute the statstical inefficiency if the variance is zero.

  ! Accumulate the integrated correlation time by computing the normalized correlation time at
  ! increasing values of t.  Stop accumulating if the correlation function goes negative, since
  ! this is unlikely to occur unless the correlation function has decayed to the point where it
  ! is dominated by noise and indistinguishable from zero.
  t = 1
  increment = 1
  do while(t < N-1)
     ! Compute unnormalized correlation function for time t.
     C = sum( A_kn(1:K,1:(N-t))*B_kn(1:K,(1+t):N) + B_kn(1:K,1:(N-t))*A_kn(1:K,(1+t):N) ) &
          / (2.0 * K * (N-t))

     ! Compute normalized fluctuation correlation functions from unnormalized correlation functions.
     C = (C - mu_A*mu_B) / sigma2_AB

     ! Terminate if the correlation function has crossed zero.
     if(C <= 0) exit

     ! Accumulate contribution to the statistical inefficiency.
     g = g + 2.0 * C * (1.0 - real(t)/real(N)) * increment

     ! Increment t and the amount by which we increment t.
     t = t + increment
     increment = increment + 1
  end do

end subroutine computeStatisticalInefficiency

!=============================================================================================
! Compute expectation and uncertainty.
!=============================================================================================
subroutine WHAM_computeExpectation(beta, A_kn, A, dA)

  ! Arguments.
  real, intent(in) :: beta   ! the inverse temperature of interest
  real, dimension(:,:), intent(in) :: A_kn
       ! A_kn(k,n) is the value of the observable of interest A for snapshot n of replica k
  real, intent(out) :: A  ! the expectation of the observable A at inverse temperature beta
  real, intent(out) :: dA ! the uncertainty in the expectation of A

  ! Local variables.
  real :: g_Aw_Aw, g_Aw_w, g_w_w ! statistical inefficiencies
  real :: max_g_Aw_w ! maximum allowed value of g_Aw_w
  real :: mu_Aw, mu_w ! means
  real :: sigma2_Aw_Aw, sigma2_Aw_w, sigma2_w_w ! variances
  integer :: kIndex ! replica index
  real :: X ! expectation of numerator
  real :: Y ! expectation of denominator
  real :: d2X ! square uncertainty in numerator
  real :: d2Y ! square uncertainty in denominator
```

```
  real :: dXdY ! cross-uncertainty between numerator and denominator
  real, dimension(:,:), allocatable :: Aw_kn ! temporary storage for the timeseries Aw_kn * w_kn

  ! Compute weights.
  call WHAM_computeWeights(beta, w_kn)

  ! Store Aw timeseries.
  allocate(Aw_kn(K,N))
  Aw_kn = A_kn * w_kn

  ! Compute means, variances, and statistical inefficiencies.
  call computeStatisticalInefficiency(Aw_kn, Aw_kn, mu_Aw, mu_Aw, sigma2_Aw_Aw, g_Aw_Aw)
  call computeStatisticalInefficiency(Aw_kn, w_kn, mu_Aw, mu_w, sigma2_Aw_w, g_Aw_w)
  call computeStatisticalInefficiency(w_kn, w_kn, mu_w, mu_w, sigma2_w_w, g_w_w)
  max_g_Aw_w = sqrt(g_Aw_Aw * g_w_w) * abs( sqrt(sigma2_Aw_Aw * sigma2_w_w) / sigma2_Aw_w )
  if( (g_Aw_w > max_g_Aw_w) .and. (max_g_Aw_w > 1) ) g_Aw_w = max_g_Aw_w

  ! Clean up.
  deallocate(Aw_kn)

  ! Compute expectation and uncertainty.
  X = K * mu_Aw
  Y = K * mu_w
  A = X / Y
  d2X = K * sigma2_Aw_Aw / (N / g_Aw_Aw);
  dXdY = K * sigma2_Aw_w / (N / g_Aw_w);
  d2Y = K * sigma2_w_w / (N / g_w_w);
  dA = sqrt( A**2 * (d2X/(X**2) + d2Y/(Y**2) - 2*dXdY/(X*Y)) )

end subroutine WHAM_computeExpectation

!=========================================================================================================
! Compute free energies of an observable that only assumes discrete values.
! Useful for computing a potential of mean force.
!=========================================================================================================
subroutine WHAM_computePMF(beta, state_kn, nstates, F_i, dF_i)

  ! Arguments.
  real, intent(in) :: beta   ! the inverse temperature of interest
  integer, dimension(:,:), intent(in) :: state_kn
       ! state_kn(k,n) is the state (1..nstates) occupied by snapshot n of replica k
  integer, intent(in) :: nstates ! the number of states
  real, dimension(:), intent(out) :: F_i  ! estimate of the free energy of state i
  real, dimension(:), intent(out) :: dF_i ! the uncertainty in the free energy of state i

  ! Local variables.
  integer :: kIndex ! replica index
  integer :: nIndex ! snapshot index
  integer :: iIndex, jIndex ! state index
  real, dimension(:), allocatable :: P_i, dP_i
       ! P_i(i) is the probability that state i is occupied, and dP_i(i) is uncertainty
  real, dimension(:), allocatable :: X_i, Y_i, d2X_i, d2Y_i, dXdY_i
  real, dimension(:), allocatable :: mu_Aw_i, sigma2_Aw_Aw_i, sigma2_Aw_w_i
  real, dimension(:), allocatable :: g_Aw_Aw_i, g_Aw_w_i, max_g_Aw_w_i
  real :: mu_w, sigma2_w_w, g_w_w, w
  real, dimension(:), allocatable :: C_Aw_Aw_i, C_Aw_w_i
  logical, dimension(:), allocatable :: has_crossed_zero_Aw_Aw_i, has_crossed_zero_Aw_w_i
  integer :: t, t0, increment

  ! Compute weights.
  call WHAM_computeWeights(beta, w_kn)

  ! Compute statistical inefficiencies.
  call computeStatisticalInefficiency(w_kn, w_kn, mu_w, mu_w, sigma2_w_w, g_w_w)

  ! Accumulate contributions to expectation and uncertainty for each bin.
  allocate(P_i(nstates), dP_i(nstates))
  allocate(X_i(nstates),Y_i(nstates),d2X_i(nstates),d2Y_i(nstates),dXdY_i(nstates))
  allocate(mu_Aw_i(nstates),sigma2_Aw_Aw_i(nstates),sigma2_Aw_w_i(nstates))
  allocate(g_Aw_Aw_i(nstates),g_Aw_w_i(nstates),max_g_Aw_w_i(nstates))
  allocate(C_Aw_Aw_i(nstates),C_Aw_w_i(nstates))
  allocate(has_crossed_zero_Aw_Aw_i(nstates),has_crossed_zero_Aw_w_i(nstates))
  X_i = 0; Y_i = 0; d2X_i = 0 ; d2Y_i = 0; dXdY_i = 0


  ! Compute means and (co)variances.
  mu_Aw_i = 0; sigma2_Aw_Aw_i = 0; sigma2_Aw_w_i = 0

  do kIndex = 1,K
```

```
   do nIndex = 1,N
      iIndex = state_kn(kIndex,nIndex)
      w = w_kn(kIndex,nIndex)
      mu_Aw_i(iIndex) = mu_Aw_i(iIndex) + w
      sigma2_Aw_Aw_i(iIndex) = sigma2_Aw_Aw_i(iIndex) + w**2
      sigma2_Aw_w_i(iIndex) = sigma2_Aw_w_i(iIndex) + w**2
   end do
end do
mu_Aw_i = mu_Aw_i / real(K*N)
sigma2_Aw_Aw_i = sigma2_Aw_Aw_i / real(K*N) - mu_Aw_i*mu_Aw_i
sigma2_Aw_w_i = sigma2_Aw_w_i / real(K*N) - mu_Aw_i*mu_w

! Accumulate the integrated correlation time by computing the normalized correlation time at
! increasing values of t.  Stop accumulating if the correlation function goes negative, since
! this is unlikely to occur unless the correlation function has decayed to the point where it
! is dominated by noise and indistinguishable from zero.
g_Aw_Aw_i = 1 ; g_Aw_w_i = 1
t = 1; increment = 1
has_crossed_zero_Aw_Aw_i = .false.; has_crossed_zero_Aw_w_i = .false.
do while(t < N-1)
   ! Compute unnormalized correlation function for time t.
   C_Aw_Aw_i = 0; C_Aw_w_i = 0
   do t0 = 1,(N-t)
      do kIndex = 1,K
         ! a contribution is only made if the replica is in a bin at time t0 and t0+t
         iIndex = state_kn(kIndex,t0)
         jIndex = state_kn(kIndex,t0+t)
         w = w_kn(kIndex,t0)*w_kn(kIndex,t0+t)
         if( iIndex == jIndex ) C_Aw_Aw_i(iIndex) = C_Aw_Aw_i(iIndex) + w
         C_Aw_w_i(iIndex) = C_Aw_w_i(iIndex) + 0.5 * w
         C_Aw_w_i(jIndex) = C_Aw_w_i(jIndex) + 0.5 * w
      end do
   end do
   C_Aw_Aw_i = C_Aw_Aw_i / real(K*(N-t))
   C_Aw_w_i = C_Aw_w_i / real(K*(N-t))

   ! Compute normalized fluctuation correlation functions from unnormalized correlation functions.
   where (sigma2_Aw_Aw_i /= 0) C_Aw_Aw_i = (C_Aw_Aw_i - mu_Aw_i**2) / sigma2_Aw_Aw_i
   where (sigma2_Aw_w_i /= 0) C_Aw_w_i = (C_Aw_w_i - mu_Aw_i*mu_w) / sigma2_Aw_w_i

   ! Note those correlation functions that have crossed zero.
   where (C_Aw_Aw_i <= 0) has_crossed_zero_Aw_Aw_i = .true.
   where (C_Aw_w_i <= 0) has_crossed_zero_Aw_w_i = .true.

   ! Accumulate statistical inefficiencies for those correlation functions that have not yet
   ! crossed zero.
   where (.not. has_crossed_zero_Aw_Aw_i) &
        g_Aw_Aw_i = g_Aw_Aw_i + 2 * C_Aw_Aw_i * (1.0 - real(t)/real(N)) * increment
   where (.not. has_crossed_zero_Aw_w_i) &
        g_Aw_w_i = g_Aw_w_i + 2 * C_Aw_w_i * (1.0 - real(t)/real(N)) * increment

   ! Terminate if all correlation functions have crossed zero.
   if (all(has_crossed_zero_Aw_Aw_i) .and. all(has_crossed_zero_Aw_w_i)) exit

   ! Increment t and the amount by which we increment t.
   t = t + increment
   increment = increment + 1
end do

! Ensure that estimate of cross-uncertainty is reasonable (see Appendix).
max_g_Aw_w_i = sqrt(g_Aw_Aw_i * g_w_w) * abs( sqrt(sigma2_Aw_Aw_i * sigma2_w_w) / sigma2_Aw_w_i )
where( (g_Aw_w_i > max_g_Aw_w_i) .and. (max_g_Aw_w_i > 1) ) g_Aw_w_i = max_g_Aw_w_i

! Accumulate contributions to numerator and denominator of expectation.
X_i = K * mu_Aw_i
Y_i = K * mu_w

! Accumulate contributions to uncertainties in numerator and denominator.
d2X_i  = K * sigma2_Aw_Aw_i / (N / g_Aw_Aw_i);
d2Y_i  = K * sigma2_w_w / (N / g_w_w);
dXdY_i = K * sigma2_Aw_w_i / (N / g_Aw_w_i);

! Compute expectation and uncertainties.
P_i = X_i / Y_i
dP_i = sqrt( P_i**2 * (d2X_i/(X_i**2) + d2Y_i/(Y_i**2) - 2*dXdY_i/(X_i*Y_i)) )

! Convert probabilities to free energies.
F_i = - (1/beta) * log(P_i)
dF_i = (1/beta) * (dP_i / P_i)
```

```
    ! Shift F_i so that min_i F_i = 0.
    F_i = F_i - minval(F_i)

    ! Clean up.
    deallocate(X_i,Y_i,d2X_i,d2Y_i,dXdY_i)
    deallocate(mu_Aw_i,sigma2_Aw_Aw_i,sigma2_Aw_w_i)
    deallocate(g_Aw_Aw_i,g_Aw_w_i,max_g_Aw_w_i)
    deallocate(C_Aw_Aw_i,C_Aw_w_i,has_crossed_zero_Aw_Aw_i,has_crossed_zero_Aw_w_i)
    deallocate(P_i, dP_i)

  end subroutine WHAM_computePMF

end module WHAM
```

3 Listing 3. Example driver.

This example driver demonstrates use of the above modules in a simple application that reads energies, an observable, .

```
!=======================================================================================================
! EXAMPLE DRIVER PROGRAM.
! Modify parameters K, L, N, filenames, and target_temperature for use with your own data.
!=======================================================================================================
program WhamDriver
  use WHAM

  implicit none

  ! Parameters
  integer, parameter :: K = 10   ! number of replicas
  integer, parameter :: L = 10   ! number of temperatures
  integer, parameter :: N = 10000 ! number of snapshots/replica
  real, parameter :: kB = 1.381 * 6.02214 / 4184.0 ! Boltzmann's constant, in kcal/(mol K)
  real, parameter :: target_temperature = 300.0     ! target temperature, in K
  real, parameter :: beta = 1.0 / (kB * target_temperature)
  character(len=*), parameter :: betaFilename = 'test_data/by-replica/beta.dat'
      ! name of file containing inverse temperatures
  character(len=*), parameter :: energyFilename = 'test_data/by-replica/energies.dat'
      ! name of file containing energies -- each replica is a column, each snapshot a row
  character(len=*), parameter :: tempIndexFilename = 'test_data/by-replica/temp_index.dat'
      ! name of file containing indices of temperatures visited by each replica
  character(len=*), parameter :: observableFilename = 'test_data/by-replica/observable.dat'
      ! name of file containing observable -- each replica is a column, each snapshot a row
  character(len=*), parameter :: stateIndexFilename = 'test_data/by-replica/state_index.dat'
      ! name of file containing states
  integer, parameter :: nstates = 36 ! number of states for PMF
  integer, parameter :: fileUnit = 2 ! unit to use for file access

  ! Local variables
  real, dimension(:), allocatable      :: beta_l         ! inverse temperatures
  real, dimension(:,:), allocatable    :: U_kn           ! potential energies
  integer, dimension(:,:), allocatable :: temp_index_kn  ! temperature indices
  integer, dimension(:,:), allocatable :: state_index_kn ! state indices for PMF
  real, dimension(:,:), allocatable    :: A_kn
     ! A_kn(k,n) is the observable A for snapshot n of replica k
  real :: A, dA ! estimate of expectation and uncertainty of an observable
  real, dimension(:), allocatable      :: F_i, dF_i
     ! potential of mean force and associated uncertainty
  integer :: iIndex ! state index

  !=======================================================================================================
  ! Read simulation data and initialize WHAM.
  !=======================================================================================================

  ! Read inverse temperatures.
  write(*,*) 'Reading inverse temperatures from ', betaFilename, '...'
  allocate(beta_l(L))
  open(unit=fileUnit, file=betaFilename, status='old', access='sequential', form='formatted')
  read(unit=fileUnit, fmt=*) beta_l
  close(unit=fileUnit)

  ! Read snapshot energies.
  write(*,*) 'Reading energies from ', energyFilename, '...'
  allocate(U_kn(K,N))
  open(unit=fileUnit, file=energyFilename, status='old', access='sequential', form='formatted')
  read(unit=fileUnit, fmt=*) U_kn
  close(unit=fileUnit)
```

```
  ! Read temperature indices.
  write(*,*) 'Reading temperature indices from ', tempIndexFilename, '...'
  allocate(temp_index_kn(K,N))
  open(unit=fileUnit, file=tempIndexFilename, status='old', access='sequential', form='formatted')
  read(unit=fileUnit, fmt=*) temp_index_kn
  close(unit=fileUnit)

  ! Read state indices for the calculation of a PMF.
  write(*,*) 'Reading state indices from ', stateIndexFilename, '...'
  allocate(state_index_kn(K,N))
  open(unit=fileUnit, file=stateIndexFilename, status='old', access='sequential', form='formatted')
  read(unit=fileUnit, fmt=*) state_index_kn
  close(unit=fileUnit)

  ! Construct the histogram bins and counts.
  write(*,*) 'Constructing histogram counts...'
  call WHAM_initialize(U_kn, beta_l, temp_index_kn)

  !=============================================================================================
  ! Estimate the expectation and uncertainty for an observable.
  !=============================================================================================

  ! Read observable.
  write(*,*) 'Reading observable from ', observableFilename, '...'
  allocate(A_kn(K,N))
  open(unit=fileUnit, file=observableFilename, status='old', access='sequential', form='formatted')
  read(unit=fileUnit, fmt=*) A_kn
  close(unit=fileUnit)

  ! Compute expectation and uncertainty of an observable.
  write(*,*) 'Computing expectation and uncertainty of an observable...'
  call WHAM_computeExpectation(beta, A_kn, A, dA)
  write(*,*) 'A = ', A, ', dA = ', dA

  !=============================================================================================
  ! Compute a potential of mean force.
  !=============================================================================================

  ! Compute potential of mean force.
  write(*,*) 'Computing potential of mean force...'
  allocate(F_i(nstates),dF_i(nstates))
  call WHAM_computePMF(beta, state_index_kn, nstates, F_i, dF_i)
  do iIndex = 1,nstates
     write(*,'(i8,f8.4,f8.4)') iIndex, F_i(iIndex), dF_i(iIndex)
  end do

  !=============================================================================================
  ! Clean up.
  !=============================================================================================
  write(*,*) 'Cleaning up...'
  call WHAM_finalize()
  write(*,*) 'Done.'

end program WhamDriver
```