SUPPLEMENT 2

Computer code

Supplement to Ljungström et al. (2020) — Mar Ecol Prog Ser 636: 1-18 — https://doi.org/10.3354/meps13241

1. General analysis

1. Main program

```
!***************************************************************************
!
!  PROGRAM: General_analysis
!
!***************************************************************************

    program General_analysis

    !Module for printing results to binary files
    use binary_IO

    implicit none

    ! Parameters
    integer   day,hour
    integer, parameter :: Days_In_Year = 365, dt_per_Day = 24 !240
    integer, parameter :: fminday = 91, fmaxday = 244, wminday = 245, wmaxday = 455
!365+90  Days in foraging and winter season
    !real*8,  parameter :: Latitude_Deg = 65.
    real*8,  parameter :: DayLigThresh = 0.1
    real*8,  parameter :: clouds = 0.5      !Cloud cover fraction
    !real*8, parameter :: minTemp = 4
    !Bioenergetics
    real*8, parameter:: J_Cal      = 4.184             !Joules per 1 calorie
    real*8, parameter:: alpha = 0.0033*13.560   ! Intercept of allometric weight function
= 0.0033 gO2/gWW/day * 13.560      kJ/gO2 --> kJ/(gWW*day)
    real*8, parameter :: RQ = 0.0548          ! Temperature dependence coefficient
respiration (Rudstam 88, based on alewife)
    real*8, parameter :: RB = -0.227          ! Slope of allometric weight function
(Rudstam 88, based on De Silva and Balbontin 74, 0-group Atlantic herring)
    !Gastric rates
    real*8, parameter :: kJ_gww_fish = 10   !kJ per g wet weight of herring - used in
Baulier, mean from Varpe, 2005
    real*8, parameter :: dw_ww = 0.13       !Dry weight to wet weight ratio, Mullin 1969
in Rudstan 1988
    !real*8, parameter :: GA = 1.E-4        !From Rosland & Giske, 1994
    real*8 GA                               !Calculate GA according to scaling
    real*8, parameter :: GB = 0.0693        !From Rosland & Giske, 1994
    real*8, parameter :: K = 0.5            !Digestion scaling factor

    ! Variables
```

```
    real*8 h, dh, SunHeight, SurfLig, Seasondays, day_real
    real*8 pi, rlat, radmax, ssurf, altdeg, VisualRange
    real*8, dimension(91:244) :: Seasonal_PPm3_scaler  !PreyAbund subroutine - scales prey
abundance according to seasonal curve
    real*8, dimension(2*365) :: Temp  !Temperture subroutine
    real*8 SurpE, Tot_Surplus, IngE, DaySurpE, DayNetIng, DayResp, Tot_Resp,  DayDigCost,
Dig_cost  !Bioenergetics subroutine
    real*8 WResp, Tot_WResp, Growth_pot, Tot_WinterResp !Winter_Bioenergetics subroutine
    real*8 VisRtoSDPreyinBL, VisRange, EyeSens, PreyImageArea, Prey_J_Cons  !Ingestion
subroutine
    real*8 Clearance_Rate, Day_Clearance, Tot_Clearance, Day_Prey_items, Tot_n, Prey_J,
Day_Consum, Tot_Consum, Tot_NetIng   !Ingestion subroutine
    real*8 Spec_GastricEvac, GastricEvacRate

    !! Parameters
    ! Fish characteristics
    real*8 :: min_Length = 10
    real*8 :: Length_cm(1:36)
    real*8 :: WWg(1:36)
    real*8 :: Gut(1:36)
    real*8 :: MaxGut(1:36)
    real*8 :: MaxGut_g(1:36)
    real*8 :: Consumption
    real*8 :: Digestion_s(1:36)
    real*8 :: Digestion(1:36)
    real*8 :: Pot_Digestion(1:36)
    real*8 :: DailyDigestion(1:36)
    real*8 :: Tot_Digestion(1:36)
    real*8 :: DailyPotDigestion(1:36)
    real*8 :: Tot_Pot_Digestion(1:36)
    real*8 :: Gut_gww_p1(1:36)
    real*8 :: Gut_gww_p2(1:36)
    real*8 :: Gut_gdw_p1(1:36)
    real*8 :: Gut_gdw_p2(1:36)
    real*8 :: Gut_kJ_p1(1:36)
    real*8 :: Gut_kJ_p2(1:36)
    real*8 :: MaxGut_kJ(1:36)
    ! Prey characteristics
    real*8 :: PreyContrast = 0.3
    real*8, dimension(1:3) :: PreyLength = [0.002, 0.003, 0.004]
    !real*8, dimension(1:3) :: PreyLength = [0.0033, 0.0033, 0.0033]  !Only CF
    real*8, dimension(1:3) :: PreyLength_mm = [0.002*1000, 0.003*1000, 0.004*1000]
    !real*8, dimension(1:3) :: PreyLength_mm = [0.0033*1000, 0.0033*1000, 0.0033*1000]
    real*8, dimension(1:3) :: PreyWidth = [0.002/4, 0.003/4, 0.004/4]
    !real*8, dimension(1:3) :: PreyWidth = [0.0033/4, 0.0033/4, 0.0033/4]
    real*8 :: Prey_mgdw(1:3)
    !real*8, dimension(1:2) :: PreyEnergyDensity = [3000., 6000.]
    real*8, dimension(1:3) :: PreyEnergyDensity = [5000, 6000, 7000]
    !real*8, dimension(1:2) :: PreyEnergyDensity = [6400., 6400.]
    !real*8, dimension(1:3) :: PreyAbundance = [3674, 1333, 650]         !ind/m3
    real*8, dimension(1:3) :: PreyAbundance = [1652, 600, 292]         !ind/m3
    !real*8, dimension(1:3) :: PreyAbundance = [600, 600, 600]         !ind/m3
    real*8, dimension(1:3) :: PreyAbundanceLevel = [0.5, 1., 1.5]
    !real*8, dimension(1:3) :: PreyAbundanceLevel = [1, 1, 1]
    real*8, dimension(1:3) :: PreyHandlingTime = [1.5, 1.5, 1.5]
    !Gut calibration
    real*8, dimension(1:2) :: PreyEnergy = [6400, 5200]
    ! Light
```

```
    real*8  BeamAtt
    real*8  Eb
    real*8, dimension(1:3) :: Latitude = [58., 68., 78.]        !deg
    !real*8, dimension(1:3) :: Latitude = [65., 65., 65.]        !deg
    !real*8, dimension(1:2) :: Diff_att = [0.0863, 0.2121]       !Diffuse attenuation
    real*8, dimension(1:3) :: Diff_att = [0.064, 0.0863, 0.2121]       !Diffuse
attenuation
    !real*8, dimension(1:2) :: Diff_att = [0.0863, 0.0863]       !Diffuse attenuation
    real*8, dimension(1:3) :: Beam_att = [0.0579, 0.4209, 0.6044]       !Beam attenuation
    !real*8, dimension(1:2) :: Beam_att = [0.4209, 0.4209]       !Beam attenuation
    ! Forage depth
    real*8 :: Forage_depth = 30     !m
    ! Temperature
    real*8, dimension(1:3) :: MinTemp = [2, 4, 6] !degrees C
    real*8, dimension(1:3) :: MaxTemp = [5, 7, 9]
    !real*8, dimension(1:3) :: MaxTemp = [7, 7, 7]

    ! Array index names
    real*8 L, MaxT, Da, Cal_gdw, PPm3, Lat, PSize

    ! Output matrices (L,PSize,Lat,Temp,PPm3,Cal_gdw,Da)
    real(8),  dimension(36, 3, 3, 3, 3, 3, 3) :: loop_output_summer1, loop_output_summer2,
loop_output_summer3, loop_output_summer4,&
        loop_output_summer5, loop_output_summer6, loop_output_winter, loop_output_TotCost

    ! Parameter arrays (values)
    Length_cm(:) = min_Length + 1*([1:size(Length_cm)]-1)

    ! Variable calculations
    pi = 4.0*atan(1.0)
    !rlat = -(Latitude+180)*(pi/180)
    dh = 24./real(dt_per_Day)         !Hours per timestep = 0.1 --> NOW dt_per_Day = 24 -
-> dh = 1


    ! Run simulations over parameters in sensitivity analysis

    !!! FEEDING SEASON PART !!!!

    call PreyAbundanceNWS(Seasonal_PPm3_scaler)

    do MaxT = 1,size(MaxTemp)

        call Temperature(MinTemp(MaxT),MaxTemp(MaxT),Temp)

        do Da = 1,size(Diff_att)

            BeamAtt = Beam_att(Da)  !Pick beam attenuation value that corresponds to Chla
level as defined by the diffuse attenuation

            do Cal_gdw = 1,size(PreyEnergyDensity)

                do PPm3 = 1,size(PreyAbundanceLevel)

                    do Lat =1,size(Latitude)
                        rlat = -(Latitude(Lat)+180)*(pi/180)

                        do PSize = 1,size(PreyLength)
```

```
                            ! Run simulation over range of adult herring lengths
                            do L = 1,size(Length_cm)   !Loop over size

                                WWg(L) = 0.00603 * Length_cm(L)**3.0904 ! General
length/weight relationship for Atlantic herring (from ICES)
                                MaxGut_g(L) = 0.03*WWg(L) ! Max gut in grams, 3% of
weight/size
                                Gut_gww_p1(L) = 0.6*MaxGut_g(L)
                                Gut_gww_p2(L) = 0.4*MaxGut_g(L)
                                Gut_gdw_p1(L) = 0.13*Gut_gww_p1(L)
                                Gut_gdw_p2(L) = 0.13*Gut_gww_p2(L)
                                Gut_kJ_p1(L) = PreyEnergy(1) * Gut_gdw_p1(L) * J_Cal !Prey
energy content [J]
                                Gut_kJ_p2(L) = PreyEnergy(2) * Gut_gdw_p2(L) * J_Cal
!Prey energy content [J]
                                MaxGut_kJ(L) = Gut_kJ_p1(L) + Gut_kJ_p2(L)  ! Max gut in
kJ

                                !EnergyFatInd_kJ = 100.012*exp(0.11*Length_cm(L)) !Energy
content pr ind (fat) (Slotte 1999, Slotte & Fiksen 2000)

                                ! Sums over feeding season
                                Tot_Consum = 0.
                                Tot_n = 0.
                                Tot_Surplus = 0.
                                Tot_WResp = 0.
                                Tot_Digestion(L) = 0.
                                Tot_Pot_Digestion(L) = 0.
                                Tot_NetIng = 0.
                                Tot_Resp = 0.
                                Dig_cost = 0.

                                Gut(L) = 0.

                                ! Ingestion loop over days in feeding season
                                Do day = fminday, fmaxday

                                    ! Daily sums

                                    DailyDigestion(L) = 0.
                                    DailyPotDigestion(L) = 0.
                                    Day_Prey_items = 0.
                                    Day_Consum = 0.

                                    !Ingestion loop over hours for each day of the feeding
season
                                    Do hour = 1, dt_per_Day
                                        h = dh*real(hour) ! from 0 to 24 by 0.1 - NOW by 1

                                        day_real = real(day)

                                        ! Subroutine gets max irradiance for time, day,
latitude, cloud cover (HYCOM qsw0 modified by Anders F. Opdal)
                                        call
qsw0_hr(radmax,ssurf,altdeg,clouds,rlat,day_real,hour,Days_in_year)

                                        ! Light at foraging depth (W/m2)
```

```
                                    Eb = ssurf*exp(-Diff_att(Da)*Forage_depth)
                                    Eb = Eb/4.6*0.45*3

                                    ! Subroutine gets ingestion estimates over hours
for each day of the feeding season
                                    call ingestion(Eb, BeamAtt, PreyAbundance(PSize),
PreyAbundanceLevel(PPm3), Seasonal_PPm3_scaler(day), PreyHandlingTime(PSize),
Length_cm(L)*0.01, &
                                        PreyLength(PSize), PreyWidth(PSize),
PreyContrast, PreyEnergyDensity(Cal_gdw), PreyLength_mm(PSize), Prey_J_Cons)

                                    ! Daily net consumed energy, i.e. daily digested
energy

                                    !! Digestion dependent on both size and
temperature - used K scaled to fit Marion's data
                                    ! Eats
                                    Consumption = Prey_J_Cons * 3600. * dh * 1E-3
!Consumed energy (kJ) - Prey_J_Enc(J/s)*3600s/h*1h*0.001kJ/J --> kJ
                                    Gut(L) = max(0., min(Gut(L) + Consumption,
MaxGut_kJ(L)))    !New gut content (kJ) after consumption
                                    ! Digests part of what is in gut
                                    Spec_GastricEvac = 10 * alpha * WWg(L)**RB *
exp(RQ*Temp(day))   !Weight specific evacuation rate (kJ/(gWW*day) - units of K)
                                    GastricEvacRate = Spec_GastricEvac * WWg(L) / 24
!Evacuation rate (kJ/h)
                                    Digestion(L) = min(GastricEvacRate*dh, Gut(L))
!Digestion per timestep (kJ)
                                    Pot_Digestion(L) = GastricEvacRate*dh
                                    ! Have left in gut what is not digested
                                    Gut(L) = max(0., Gut(L) - Digestion(L))
!New gut content (kJ) after digestion

                                    ! Daily sum of digestion - i.e. consumption with
digestion limitation
                                    DailyDigestion(L) = DailyDigestion(L) +
Digestion(L)     !Daily digested energy (kJ) --> input Bioenergetics subroutine
                                    DailyPotDigestion(L) = DailyPotDigestion(L) +
Pot_Digestion(L)   !Daily digested energy with no encounter limitation

                                    !! Daily sums of ingestion estimates - no
digestion limitation
                                    Day_Consum = Day_Consum + Consumption

                                enddo !End hour loop - time steps

                                Call BioEnergetics(Temp(day), Length_cm(L),
DailyDigestion(L), DayResp, DayDigCost, DayNetIng, DaySurpE)

                                ! Feeding season total surplus energy (sum of daily
sum DaySurpE)

                                Tot_Resp = Tot_Resp + DayResp
                                Dig_cost = Dig_cost + DayDigCost
                                Tot_NetIng = Tot_NetIng + DayNetIng !Total digested
energy - SDA -EX -EG
                                Tot_Surplus = Tot_Surplus + DaySurpE !Digestion and
respiration costs subtracted from ingested energy (incl. dig. lim.)(kJ)
```

```
                                    !! Feeding season sums from ingestion subroutine (sum
of daily sums - Used in previous script!)
                                    Tot_Consum = Tot_Consum + Day_Consum ! Sum of daily
consumed energy --> Total energy ingested over feeding season (kJ) - no digestion
limitation
                                    Tot_Digestion(L) = Tot_Digestion(L) +
DailyDigestion(L)
                                    Tot_Pot_Digestion(L) = Tot_Pot_Digestion(L) +
DailyPotDigestion(L)

                                enddo !End day loop - daily foraging cycle

                                loop_output_summer1(L,PSize,Lat,MaxT,PPm3,Cal_gdw,Da)  =
Tot_Consum   !(L,PSize,Lat,Temp,PPm3,Cal_gdw,Da)
                                loop_output_summer2(L,PSize,Lat,MaxT,PPm3,Cal_gdw,Da)  =
Tot_Digestion(L)
                                loop_output_summer3(L,PSize,Lat,MaxT,PPm3,Cal_gdw,Da)  =
Tot_Resp
                                loop_output_summer4(L,PSize,Lat,MaxT,PPm3,Cal_gdw,Da)  =
Tot_NetIng
                                loop_output_summer5(L,PSize,Lat,MaxT,PPm3,Cal_gdw,Da)  =
Tot_Surplus
                                loop_output_summer6(L,PSize,Lat,MaxT,PPm3,Cal_gdw,Da)   =
Tot_Pot_Digestion(L)


                                !!!! WINTER SEASON PART !!!!

                                Tot_WResp = 0.

                                do day = wminday, wmaxday

                                    Call BioEnergetics_Winter(Temp(day), Length_cm(L),
WResp)

                                    ! Total winter respiration cost (sum of daily sums
WResp)
                                    Tot_WResp = Tot_WResp + WResp ! kJ

                                enddo !End day loop

                                ! Growth potential - suplus energy left after winter =
total surplus energy at end of feeding season - total repiration energy over winter
                                Growth_pot = Tot_Surplus - Tot_WResp

                                loop_output_winter(L,PSize,Lat,MaxT,PPm3,Cal_gdw,Da) =
Growth_pot ! (kJ) changes with feeding season surplus energy

                                ! Total annual energy cost
                                loop_output_TotCost(L,PSize,Lat,MaxT,PPm3,Cal_gdw,Da) =
Tot_Resp + Dig_cost + Tot_WResp


                            enddo !End L
                        enddo !End Psize
                    enddo !End Lat
                enddo !PPm3
```

```
            enddo !Cal_gdw
        enddo !Diff_att
    enddo !MaxT


    !Save output matrices to binary files
    call array_to_binary(loop_output_summer1, 'TotCons_gen.bin')
    call array_to_binary(loop_output_summer2, 'TotDig_gen.bin')
    call array_to_binary(loop_output_summer3, 'TotResp_gen.bin')
    call array_to_binary(loop_output_summer4, 'TotNetIng_gen.bin')
    call array_to_binary(loop_output_summer5, 'TotSurp_gen.bin')
    call array_to_binary(loop_output_summer6, 'Tot_Pot_Dig_gen.bin')
    call array_to_binary(loop_output_winter, 'Growth_pot_gen.bin')
    call array_to_binary(loop_output_TotCost, 'TotCost_gen.bin')


    end program General_analysis
```

1.2 Subroutines

1.2.1 ingestion

```
subroutine ingestion(Eb, BeamAtt, PreyPer_m3, PreyPer_m3_level,
Seasonal_PreyPer_m3_scaler, PreyHandlingTime, FishLength_m, PreyLength, PreyWidth,
PreyContrast, Prey_energy_Cal_gdw, PreyLength_mm, Prey_J_Consumed)

    implicit none

    ! Parameters
    real*8, parameter:: EyeSensPreyLength = 0.004   !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: EyeSensPreyWidth = 0.001    !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: EyeSensPreyContrast = 0.3   !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: VisRtoSDPreyinBL = 1.       !Detection distance in BL of small
prey at satiating light and clear water
    real*8, parameter:: Ke = 1.                                 !Fish light satiation
(umol p m-2 s-1)
    real*8, parameter:: VisFieldShape = 0.5             !Fraction of visual field
effectively scanned
    real*8, parameter:: MinVisRange = 0.01              !Minimum detection range non-
visual cues - assume herring detect
    real*8, parameter:: Scale_ing  = 0.3
    !Scaling for overlapping search fields, capture efficiency..
    ! Prey parameters
    real*8, parameter:: J_Cal      = 4.184             !Joules per 1 calorie

    real*8 Eb, BeamAtt, PreyPer_m3, PreyPer_m3_level, Seasonal_PreyPer_m3_scaler,
PreyHandlingTime, FishLength_m, PreyLength, PreyWidth,&
        PreyContrast, Prey_energy_Cal_gdw, PreyLength_mm    !in
    real*8 pi, PreyImageArea, Prey_abundance, Swim_vel, EyeSens, VisualRange,
Clearance_Rate, Prey_J, n, Prey_mgdw  !in subroutine
    real*8 Prey_J_Consumed  !out
```

```
    ! Variables
    integer   IER, no
    pi=acos(-1.)

    ! Prey properties
    Prey_mgdw = (10**(2.50*log10(1000*PreyLength_mm)-6.51))/1E3  !Prey body mass (mg dw) -
from Uye, 1982 (used in van Deurs, 2015, but other version)
    PreyImageArea = PreyLength * PreyWidth * 0.75     !Elongated prey 0.75
    Prey_J = Prey_energy_Cal_gdw * Prey_mgdw*0.001 * J_Cal      !Prey energy content [J]
    ! Prey abundance
    Prey_abundance = PreyPer_m3_level * (Seasonal_PreyPer_m3_scaler*PreyPer_m3 +
0.1*PreyPer_m3)  !PreyPer_m3_scaler has been reduced by abundance (prop of max PPm3)
assumed at start of feeding season, here 10%

    !Predator properties
    Swim_vel = FishLength_m * 1.5                !Blaxter 1966

    ! Eye sensitivity - assumes fish detect prey (VisRtoSDPreyinBL x Larval_m) away in
clear water - Here eye sensitivity is a constant for each fish length:
    EyeSens = ((FishLength_m*VisRtoSDPreyinBL)**2.) /
(EyeSensPreyContrast*EyeSensPreyLength*EyeSensPreyWidth*0.75)

    ! Visual range in m (from AU97)
    VisualRange = sqrt(EyeSens * PreyContrast * PreyImageArea * (Eb/(Ke+Eb)))
!Approximation
    VisualRange = max(VisualRange, MinVisRange)

    if(VisualRange > 0.05)then    !Exact visual range (above ca 5 cm)
        call getr(VisualRange, BeamAtt, PreyContrast, PreyImageArea, EyeSens, Ke, Eb, IER)
    endif

    Clearance_Rate = VisFieldShape * pi * (VisualRange**2) * Swim_vel   !In m3/s

    ! Ingestion rates in number of prey items/s and energy/s (including handling time
limitation - Holling disc)
    n = Clearance_Rate * Prey_abundance / (1. + PreyHandlingTime * Clearance_Rate *
Prey_abundance)  !Total number of prey items ingested/s (items/s)
    n = n * Scale_ing                                    !Calibration of feeding
limitations - overlapping search fields, capture efficiency..
    Prey_J_Consumed = n * Prey_J               !Total energy ingested/s (J/s) -
n(items/s)*Prey_J(joules/item)

    return
    end subroutine ingestion
```

1.2.2 BioEnergetics

```
Subroutine
BioEnergetics(Temperature,FishLength_cm,DailyDig,DayResp,DayDigCost,DayNetIng,DaySurpE)
implicit none

! Parameters
!real*8, parameter:: Temp = 7      ! Degrees celcius
real*8, parameter:: alpha = 0.0033*13.560 ! Intercept of allometric weight function =
0.0033 gO2/gWW/day * 13.560 kJ/gO2 --> kJ/(gWW*day)
```

```
real*8, parameter:: RB = -0.227    ! Slope of allometric weight function (Rudstam 88,
based on De Silva and Balbontin 74, 0-group Atlantic herring)
real*8, parameter:: RQ = 0.0548    ! Water temperature dependence coefficient (Rudstam 88,
based on alewife)
real*8, parameter:: RTO = 0.03     ! Swimming speed dependence coefficient (Rudstam
88,based on aholehole Muir and Niimi 72, the same is used in alewife models)
real*8, parameter:: SDA = 0.175    ! Coefficient for specific dynamic action (Rudstam 88)
real*8, parameter:: ACT = 3.9      ! Intercept of relationship for swimming speed vs.
weight and temperature (Rudstam 88, for temp < 9)
real*8, parameter:: K4 = 0.1       ! Slope for weight dependence of swimming speed at all
water temperatures (Rudstam 88, based on Ware 78)
real*8, parameter:: BACT = 0.149   ! Coefficient for temperature dependence of swimming
speed at temp below 9 (Rudstam 88, based on alewife)
real*8, parameter:: FA = 0.16      ! Proportion of consumed food egested (Rudstam 88)
real*8, parameter:: UA = 0.10      ! Proportion of assimilated food excreted (Rudstam 88)
real*8, parameter:: beta = 0.4     ! Calibrated beta to yield output in realistic range
real*8, parameter:: SW = 0.77      ! Slope for weight dependence of swimming velociy and
cost ('The ecological implications of body size' Robert Henry Peters)

! Variables
real*8 Temperature,FishLength_cm,DailyDig !in
real*8 Weight,SwimVelcms,DayRespRate,Temp_SMR,Swim_cost_Js,Swim_cost !in routine
real*8 DayResp,DayDigCost,DayNetIng,DaySurpE !out

! General length/weight relationship for Atlantic herring (from ICES)
Weight = 0.00603 * FishLength_cm**3.0904

! Respiration cost for day - depends on body weight (g), temperature (degrees C), and
weight-dependent swimmig cost (J/s)
Temp_SMR = (alpha*Weight**RB) * exp(RQ*Temperature) * Weight !Temp-dependent SMR (kJ/day)
Swim_cost = (alpha*Weight**RB) * Weight * 0.75
DayResp = Temp_SMR + Swim_cost                                !Respiration cost/day (kJ)

! Energy ingested for day, assimilation costs subtracted
DayDigCost = ((FA*DailyDig) + UA*(DailyDig-(FA*DailyDig)) + SDA*(DailyDig-(FA*DailyDig)))
DayNetIng = DailyDig - DayDigCost

! Surplus energy for day, digestion and respiration costs subtracted from ingested energy
(incl. dig. lim.) (kJ)
DaySurpE = DayNetIng - DayResp

return
end subroutine BioEnergetics
```

1.2.3 BioEnergetics_Winter

```
subroutine BioEnergetics_Winter(Temperature,FishLength_cm,WResp)
implicit none

! Parameters
real*8, parameter:: alpha = 0.0033*13.560 ! Intercept of allometric weight function in
gO2/gWW^beta/day (Rudstam 88, alewife) X kJ/gO2
real*8, parameter:: RB = -0.227    ! Slope of allometric weight function (Rudstam 88,
based on De Silva and Balbontin 74, 0-group Atlantic herring)
real*8, parameter:: RQ = 0.0548    ! Water temperature dependence coefficient (Rudstam 88,
based on alewife)
```

```
real*8, parameter:: RTO = 0.03     ! Swimming speed dependence coefficient (Rudstam
88,based on aholehole Muir and Niimi 72, the same is used in alewife models)
real*8, parameter:: SDA = 0.175    ! Coefficient for specific dynamic action (Rudstam 88)
real*8, parameter:: ACT = 3.9      ! Intercept of relationship for swimming speed vs.
weight and temperature (Rudstam 88, for temp < 9)
real*8, parameter:: K4 = 0.13      ! Slope for weight dependence of swimming speed at all
water temperatures (Rudstam 88, based on Ware 78)
real*8, parameter:: BACT = 0.149   ! Coefficient for temperature dependence of swimming
speed at temp below 9 (Rudstam 88, based on alewife)
real*8, parameter:: FA = 0.16      ! Proportion of consumed food egested (Rudstam 88)
real*8, parameter:: UA = 0.10      ! Proportion of assimilated food excreted (Rudstam 88)
real*8, parameter:: beta = 0.4
real*8, parameter:: SW = 0.77      ! Slope for weight dependence of swimming velocity and
cost ('The ecological implications of body size' Robert Henry Peters)

! Variables
real*8 Temperature,FishLength_cm!in
real*8 Weight,SwimVelcms,Resp,Temp_SMR,Swim_cost_Js,Swim_cost !inside
real*8 WResp !out

! General length/weight relationship for Atlantic herring (from ICES)
Weight = 0.00603 * FishLength_cm**3.0904

! Respiration cost for day - depends on body weight (g), temperature (degrees C), and
weight-dependent swimmig cost (J/s)
Temp_SMR = (alpha*Weight**RB) * exp(RQ*Temperature) * Weight !Temp-dependent SMR (kJ/day)
Swim_cost = (alpha*Weight**RB) * Weight * 0.1
WResp = Temp_SMR + Swim_cost                                    !Respiration cost/day (kJ)

return
end subroutine BioEnergetics_Winter
```

1.2.4 PreyAbundanceNWS

```
    !Gives prey adundance scaler for day of the year
    subroutine PreyAbundanceNWS(PPm3_scaler)
    implicit none

    ! Parameters
    real*8, parameter:: sigma = 0.4
    real*8, parameter::  mu = 2
    real*8, parameter:: feed_day_min = 91 !1 April
    real*8, parameter:: prey_day_peak = 166 !15 June
    real*8, parameter:: feed_day_max = 244 !1 Sept

    !Variables
    integer i, j
    integer index_max_x(1)
    real*8 days_season, pi, max_scaled, pdf_max, pdfscaler, min_to_peak, xscaler
    !Arrays
    real*8 :: x(1:201)
    real*8 :: pdf(1:201)
    real*8 :: pdf_scaled(1:201)
    real*8 :: new_x(1:201)
    real*8 :: PPm3_scaler(91:244) !out

    pi = 4.0*atan(1.0)
```

```
    days_season = feed_day_max - feed_day_min
    x(:) = (/0:400:2/)
    x(:) = x(:)*0.01

    ! Create pdf curve
    do i = 1,size(x)
        pdf(i) = (1/sqrt(2*pi*sigma**2)) * exp(-((x(i)-mu)**2)/(2*sigma**2))
        write(10,'(E15.5,E15.5,18E15.5)') x(i), pdf(i)
    enddo

    ! Location of max value of x for scaling
    index_max_x(:) = maxloc(pdf)

    ! Scale x-axis according to peak day and season days
    min_to_peak = feed_day_max - feed_day_min
    xscaler = min_to_peak/(maxval(x)-minval(x))

    ! Scale y-axis to max value
    max_scaled = 1 - 0.1*1      !Max value of curve - start season with 10% of max food
abundance
    pdf_max = (1/sqrt(2*pi*sigma**2)) * exp(-((x(index_max_x(1))-mu)**2)/(2*sigma**2))
    pdfscaler = max_scaled/pdf_max    ! scaler for p, such that pscaler*pdf(mode) =
pdensity

    do j = 1,size(x)
        pdf_scaled(j) = pdfscaler * ((1/sqrt(2*pi*sigma**2)) * exp(-((x(j)-
mu)**2)/(2*sigma**2)))
        new_x(j) = x(j)*xscaler
    enddo

    !Output values for feeding season day 244 - 91 = 153 days long
    PPm3_scaler(91:244) = pdf_scaled(1:154)

    return
end subroutine PreyAbundanceNWS
```

2. Norwegian Sea sensitivity analysis

1. Main program

```
!**************************************************************************
!
!  PROGRAM: NWS_sensitivity
!
! Sensitivity analysis for NWS herring with NWS parameter values as default and scaling of
! these values over loops (0.75, 1, 1.25).
! In default scenario the intake is scaled according to proportions by weight in diet from
! data using parameter 'Scale_prey_diet'.
!
!**************************************************************************

    program NWS_sensitivity

    !Module for printing results to binary files
    use binary_IO

    implicit none

    ! Parameters
    integer   day,hour
    real*8 day_real
    integer, parameter :: Days_In_Year = 365, dt_per_Day = 24 !240
    integer, parameter :: fminday = 91, fmaxday = 244, wminday = 245, wmaxday = 455
!365+90  Days in foraging and winter season
    real*8,  parameter :: DayLigThresh = 0.1
    real*8,  parameter :: clouds = 0.5    !Cloud cover fraction
    !real*8, parameter :: minTemp = 4
    !Bioenergetics
    real*8, parameter:: J_Cal      = 4.184             !Joules per 1 calorie
    real*8, parameter:: alpha = 0.0033*13.560   ! Intercept of allometric weight function
= 0.0033 gO2/gWW/day * 13.560      kJ/gO2 --> kJ/(gWW*day)
    real*8, parameter :: RQ = 0.0548          ! Temperature dependence coefficient
respiration (Rudstam 88, based on alewife)
    real*8, parameter :: RB = -0.227          ! Slope of allometric weight function
(Rudstam 88, based on De Silva and Balbontin 74, 0-group Atlantic herring)
    !Gastric rates
    real*8, parameter :: kJ_gww_fish = 10   !kJ per g wet weight of herring - used in
Baulier, mean from Varpe, 2005
    real*8, parameter :: dw_ww = 0.13       !Dry weight to wet weight ratio, Mullin 1969
in Rudstan 1988
    real*8, parameter :: K = 0.5            !Digestion scaling factor

    ! Variables
    real*8 h, dh, SunHeight, SurfLig, Seasondays, BeamAtt
    real*8 pi, rlat, radmax, ssurf, altdeg, VisualRange
    real*8, dimension(91:244) :: Seasonal_PPm3_scaler  !PreyAbund subroutine - scales prey
abundance according to seasonal curve
    real*8, dimension(2*365) :: Temp  !Temperture subroutine
    real*8 SurpE, Tot_Surplus, IngE, DaySurpE, DayNetIng, DayResp, Tot_Resp !Bioenergetics
subroutine
    real*8 WResp, Tot_WResp, Growth_pot, Tot_WinterResp !Winter_Bioenergetics subroutine
```

```
    real*8 VisRtoSDPreyinBL, VisRange, EyeSens, PreyImageArea, Prey_J_Enc,
Tot_prey_weight_cons, Day_Prey_weight, Tot_Prey_weight  !Ingestion subroutine
    real*8 Clearance_Rate, Day_Clearance, Tot_Clearance, Day_Prey_items, Tot_n, Prey_J,
Day_Consum, Tot_Consum, Tot_NetIng   !Ingestion subroutine
    real*8 Spec_GastricEvac, GastricEvacRate

    !! Parameters (size array)
    ! Fish characteristics
    integer :: r = 36
    real*8, dimension(1:36) :: Length_cm = [(r, r=10,45,1)]
    real*8 :: WWg(1:36)
    real*8 :: Gut(1:36)
    real*8 :: MaxGut(1:36)
    real*8 :: MaxGut_g(1:36)
    real*8 :: Consumption
    real*8 :: Digestion_s(1:36)
    real*8 :: Digestion(1:36)
    real*8 :: DailyDigestion(1:36)
    real*8 :: Tot_Digestion(1:36)
    real*8 :: Gut_gww_p1(1:36)
    real*8 :: Gut_gww_p2(1:36)
    real*8 :: Gut_gdw_p1(1:36)
    real*8 :: Gut_gdw_p2(1:36)
    real*8 :: Gut_kJ_p1(1:36)
    real*8 :: Gut_kJ_p2(1:36)
    real*8 :: MaxGut_kJ(1:36)
    ! Prey characteristics; 1 = Calanus finmarchicus, 2 = Euphasiids and Amphipods
    real*8, dimension(1:2) :: PreyLength =  [0.003, 0.0144] !m
    real*8, dimension(1:2) :: PreyLength_mm = [0.003*1000, 0.0144*1000] !mm
    real*8, dimension(1:2) :: PreyWidth = [0.003/4, 0.0144/4] !m
    real*8 :: Prey_mgdw(1:2) !Calculated
    real*8, dimension(1:2) :: PreyEnergyDensity = [6400, 5200] !cal/g dry weight
    real*8, dimension(1:3) :: Prey_Abundance1 = [750,600,500]
    real*8, dimension(1:3) :: Prey_Abundance2 = [3.75,3.,2.5]

    real*8, dimension(1:2) :: PreyHandlingTime = [1.5, 5.]
    ! Scale_prey_diet = Scaling factor for diet such that Cf constitues 60 % of consumed
dry weight and A&E 40 %...
    !...Values stored in array loop_output_summer7(L,1:2)  = Tot_Prey_g_Enc(1:2) and
loop_output_summer6(L)  = Tot_Prey_weight
    real*8, dimension(1:2) :: Scale_prey_diet = [0.75, 1.]
    ! Output
    real*8 :: Day_Prey_g_Enc(1:2)
    real*8 :: Tot_Prey_g_Enc(1:2)
    real*8 :: Prey_g_Enc(1:2)

    ! Parameters for sensitivity analysis
    real*8, dimension(1:3) :: Preysize_scaler1 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Preysize_scaler2 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Energy_density_scaler = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Prey_Abund_scaler1 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Prey_Abund_scaler2 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Turb_scaler = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Handling_scaler1 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Handling_scaler2 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Ingestion_scaler = [0.8, 1., 1.2]

    !! Parameters (values)
```

```
    ! Prey characteristics
    real*8 :: PreyContrast = 0.3
    ! Light
    real*8, dimension(1:3) :: Latitude_Deg = [58, 68, 78]
    real*8 :: Diff_att = 0.0863         !Diffuse attenuation
    real*8 :: Beam_att = 0.4209         !Beam attenuation
    real*8 :: Eb(1:2)                   !Calculated
    ! Forage depth
    real*8, dimension(1:2) :: Forage_depth = [30, 60] !m
    ! Temperature
    !real*8 MaxTemp
    real*8, dimension(1:3) :: MinTemp = [2, 4, 6] !degrees C - summer and winter increase
   ! real*8, dimension(1:3) :: MinTemp = [4, 4, 4] !degrees C
    !real*8, dimension(1:3) :: MaxTemp = [7, 7, 7] !degrees C
    real*8, dimension(1:3) :: MaxTemp = [5, 7, 9] !degrees C
    !real*8 :: MinTemp = 4
    !real*8 :: MaxTemp = 7

    ! Array index names
    real*8 L, Lat, Cal_gdw, PSize1, PSize2, PPm3_1, PPm3_2, MaxT, Da, Hand1, Hand2, Ing

    ! Output matrices (L,PSize1,PSize2,Lat,Temp,PPm3,Cal_gdw,Da,Hand,Ing)
    real*8,  dimension(36, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3) :: loop_output_summer1,
loop_output_summer2, loop_output_summer3, loop_output_summer4,&
        loop_output_summer5, loop_output_winter


    ! Variable calculations
    pi = 4.0*atan(1.0)
    !rlat = -(Latitude+180)*(pi/180)
    dh = 24./real(dt_per_Day)      !Hours per timestep = 0.1 --> NOW dt_per_Day = 24 -->
dh = 1


    ! Run simulations over parameters in sensitivity analysis

    !!! FEEDING SEASON PART !!!!

    ! Get temperature and prey abundance for each day of the year
    call PreyAbundanceNWS_NEW(Seasonal_PPm3_scaler)

    do Hand1 = 1,size(Handling_scaler1)
        do Hand2 = 1,size(Handling_scaler2)

            do Ing = 1,size(Ingestion_scaler)

                do MaxT = 1,size(MaxTemp)

                    call Temperature(MinTemp(MaxT),MaxTemp(MaxT),Temp)

                    do Da = 1,size(Turb_scaler)

                        BeamAtt = Turb_scaler(Da)*Beam_att  !Calculate scaled beam
attenuation value

                        do Cal_gdw = 1,size(Energy_density_scaler)

                            do PPm3_1 = 1,size(Prey_Abund_scaler1)
```

```
                                do PPm3_2 = 1,size(Prey_Abund_scaler2)

                                    do Lat =1,size(Latitude_Deg)
                                        rlat = -(Latitude_Deg(Lat)+180)*(pi/180)

                                        do PSize1 = 1,size(Preysize_scaler1)
                                            do PSize2 = 1,size(Preysize_scaler2)

                                                ! Run simulation over range of adult
herring lengths
                                                do L = 1,size(Length_cm)   !Loop over size


                                                    ! General length/weight relationship
for Atlantic herring (from ICES)
                                                    WWg(L) = 0.00603 *
Length_cm(L)**3.0904
                                                    MaxGut_g(L) = 0.03*WWg(L)           !
Max gut in grams, 3% of weight/size
                                                    Gut_gww_p1(L) = 0.6*MaxGut_g(L)
                                                    Gut_gww_p2(L) = 0.4*MaxGut_g(L)
                                                    Gut_gdw_p1(L) = 0.13*Gut_gww_p1(L)
                                                    Gut_gdw_p2(L) = 0.13*Gut_gww_p2(L)
                                                    Gut_kJ_p1(L) = PreyEnergyDensity(1) *
Gut_gdw_p1(L) * J_Cal      !Prey energy content [J]
                                                    Gut_kJ_p2(L) = PreyEnergyDensity(2) *
Gut_gdw_p2(L) * J_Cal      !Prey energy content [J]
                                                    MaxGut_kJ(L) = Gut_kJ_p1(L) +
Gut_kJ_p2(L)  ! Max gut in kJ

                                                    !EnergyFatInd_kJ =
100.012*exp(0.11*Length_cm(L)) !Energy content pr ind (fat) (Slotte 1999, Slotte & Fiksen
2000)


                                                    ! Sums over feeding season
                                                    Tot_Consum = 0.
                                                    Tot_n = 0.
                                                    Tot_Surplus = 0.
                                                    Tot_WResp = 0.
                                                    Tot_Digestion(L) = 0.
                                                    Tot_NetIng = 0.
                                                    Tot_Resp = 0.
                                                    Tot_Prey_g_Enc(:) = 0.
                                                    Tot_Prey_weight = 0.

                                                     Gut(L) = 0.

                                                    ! Ingestion loop over days in feeding
season
                                                    Do day = fminday, fmaxday

                                                        ! Daily sums

                                                        DailyDigestion(L) = 0.
                                                        Day_Prey_items = 0.
                                                        Day_Consum = 0.
                                                        Day_Prey_g_Enc(:) = 0.
                                                        Day_Prey_weight = 0.
```

```
                                                    !Ingestion loop over hours for
each day of the feeding season
                                                    Do hour = 1, dt_per_Day
                                                        h = dh*real(hour) ! from 0 to
24 by 0.1 - NOW by 1

                                                        day_real = real(day)

                                                        ! Subroutine gets max
irradiance for time, day, latitude, cloud cover (HYCOM qsw0 modified by Anders F. Opdal)
                                                        call
qsw0_hr(radmax,ssurf,altdeg,clouds,rlat,day_real,hour,Days_in_year)

                                                        ! Light at foraging depth
                                                        Eb(1) = ssurf*exp(-
Diff_att*Turb_scaler(Da)*Forage_depth(1))
                                                        Eb(2) = ssurf*exp(-
Diff_att*Turb_scaler(Da)*Forage_depth(2))
                                                        Eb(1) = Eb(1)/4.6*0.45*3
                                                        Eb(2) = Eb(2)/4.6*0.45*3

                                                        ! Subroutine gets ingestion
estimates over hours for each day of the feeding season
                                                        call ingestion(Eb(:),
Beam_att, Prey_Abundance1(PSize1),Prey_Abundance2(PSize2),Prey_Abund_scaler1(PPm3_1),
Prey_Abund_scaler2(PPm3_2), Seasonal_PPm3_scaler(day), &
                                                            PreyHandlingTime(:),
Scale_prey_diet(:), Length_cm(L)*0.01, PreyLength(:), PreyLength_mm(:), PreyWidth(:),
Preysize_scaler1(PSize1), &
                                                            Preysize_scaler2(PSize2),
PreyContrast, PreyEnergyDensity(:), Energy_density_scaler(Cal_gdw),
Handling_scaler1(Hand1), Handling_scaler2(Hand2), &
                                                            Ingestion_scaler(Ing),
Prey_J_Enc, Prey_g_Enc, Tot_prey_weight_cons)

                                                        ! Daily net consumed energy,
i.e. daily digested energy

                                                        !! Digestion dependent on both
size and temperature - used K scaled to fit Marion's data
                                                        ! Eats
                                                        Consumption = Prey_J_Enc *
3600. * dh * 1E-3            !Consumed energy (kJ) - Prey_J_Enc(J/s)*3600s/h*1h*0.001kJ/J
--> kJ
                                                        Gut(L) = max(0., min(Gut(L) +
Consumption, MaxGut_kJ(L)))    !New gut content (kJ) after consumption
                                                        ! Digests part of what is in
gut
                                                        Spec_GastricEvac = 10 * alpha
* WWg(L)**RB * exp(RQ*Temp(day))     !Weight specific evacuation rate (kJ/(gWW*day) -
units of K)
                                                        GastricEvacRate =
Spec_GastricEvac * WWg(L) / 24        !Evacuation rate (kJ/h)
                                                        Digestion(L) =
min(GastricEvacRate*dh, Gut(L))          !Digestion per timestep (kJ)
                                                        ! Have left in gut what is not
digested
```

```
                                                    Gut(L) = max(0., Gut(L) -
Digestion(L))                   !New gut content (kJ) after digestion

                                                    ! Daily sum of digestion -
i.e. consumption with digestion limitation
                                                    DailyDigestion(L) =
DailyDigestion(L) + Digestion(L)     !Daily digested energy (kJ) --> input Bioenergetics
subroutine

                                                    !! Daily sums of ingestion
estimates - no digestion limitation
                                                    !Day_Prey_items =
Day_Prey_items + n*dh*3600.
                                                    Day_Prey_g_Enc(1) =
Day_Prey_g_Enc(1) + Prey_g_Enc(1)
                                                    Day_Prey_g_Enc(2) =
Day_Prey_g_Enc(2) + Prey_g_Enc(2)
                                                    Day_Prey_weight =
Day_Prey_weight + Tot_prey_weight_cons
                                                    Day_Consum = Day_Consum +
Consumption

                                                enddo !End hour loop - time steps

                                                Call
BioEnergetics(Temp(day),Length_cm(L),DailyDigestion(L),DayResp,DayNetIng,DaySurpE)

                                                ! Feeding season total surplus
energy (sum of daily sum DaySurpE)

                                                Tot_Resp = Tot_Resp + DayResp
                                                Tot_NetIng = Tot_NetIng +
DayNetIng !Total digested energy - SDA -EX -EG
                                                Tot_Surplus = Tot_Surplus +
DaySurpE !Digestion and respiration costs subtracted from ingested energy (incl. dig.
lim.)(kJ)

                                                !! Feeding season sums from
ingestion subroutine (sum of daily sums - Used in previous script!)
                                                !Tot_n = Tot_n + Day_Prey_items  !
Total prey items ingested (before digestion limitation has been imposed)
                                                Tot_Prey_g_Enc(1) =
Tot_Prey_g_Enc(1) + Day_Prey_g_Enc(1)
                                                Tot_Prey_g_Enc(2) =
Tot_Prey_g_Enc(2) + Day_Prey_g_Enc(2)
                                                Tot_Prey_weight = Tot_Prey_weight
+ Day_Prey_weight
                                                Tot_Consum = Tot_Consum +
Day_Consum ! Sum of daily consumed energy --> Total energy ingested over feeding season
(kJ) - no digestion limitation
                                                Tot_Digestion(L) =
Tot_Digestion(L) + DailyDigestion(L)

                                            enddo !End day loop - daily foraging
cycle
```

```
loop_output_summer1(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_Digestion(L)  !(L,PSize,Lat,Temp,PPm3,Cal_gdw,Da)

loop_output_summer2(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_Resp

loop_output_summer3(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_Consum

loop_output_summer4(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_NetIng

loop_output_summer5(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_Surplus


                                                    !!!! WINTER SEASON PART !!!!

                                                    Tot_WResp = 0.

                                                    do day = wminday, wmaxday

                                                        Call
BioEnergetics_Winter(Temp(day),Length_cm(L),WResp)

                                                        ! Total winter respiration cost
(sum of daily sums WResp)
                                                        Tot_WResp = Tot_WResp + WResp ! kJ

                                                    enddo !End day loop

                                                    ! Growth potential - suplus energy
left after winter = total surplus energy at end of feeding season - total repiration
energy over winter
                                                    Growth_pot = Tot_Surplus - Tot_WResp


loop_output_winter(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing) =
Growth_pot ! (kJ) changes with feeding season surplus energy


                                                enddo !End L
                                            enddo !End Psize1
                                        enddo !End Psize2
                                    enddo !End Lat
                                enddo !PPm3_1
                            enddo !PPm3_2
                        enddo !Cal_gdw
                    enddo !Diff_att
                enddo !MaxT
            enddo !Hand1
        enddo !Hand2
    enddo !Ing


    !Save output matrices to binary files
    !Save output matrices to binary files
```

```
    call array_to_binary(loop_output_summer1, 'TotDig_annual_Temp_NWS_V2.bin')
    call array_to_binary(loop_output_summer2, 'TotResp_annual_Temp_NWS_V2.bin')
    call array_to_binary(loop_output_summer3, 'Tot_Consum_annual_Temp_NWS_V2.bin')
    call array_to_binary(loop_output_summer4, 'Tot_NetIng_annual_Temp_NWS_V2.bin')
    call array_to_binary(loop_output_summer5, 'TotSurp_annual_Temp_NWS_V2.bin')
    call array_to_binary(loop_output_winter, 'Growth_pot_annual_Temp_NWS_V2.bin')

end program NWS_sensitivity
```

1.2 Subroutines

1.2.1 ingestion

```
subroutine ingestion(Eb, BeamAtt, PreyPer_m3_1, PreyPer_m3_2, PreyPer_m3_scaler1,
PreyPer_m3_scaler2, Seasonal_PreyPer_m3_scaler, PreyHandlingTime, Scale_prey_diet,
FishLength_m, &
        PreyLength, PreyLength_mm, PreyWidth, Psize_scaler1, Psize_scaler2, PreyContrast,
Prey_energy_Cal_gdw, Prey_energy_scaler,  Handling_scaler1, Handling_scaler2,
Ingestion_scaler, &
        Prey_J_Consumed, Prey_g_Enc, Tot_prey_weight_cons)

    implicit none

    ! Parameters
    real*8, parameter:: EyeSensPreyLength = 0.004 !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: EyeSensPreyWidth = 0.001  !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: EyeSensPreyContrast = 0.3 !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: VisRtoSDPreyinBL = 1.       !Detection distance in BL of small
prey at satiating light and clear water
    real*8, parameter:: Ke = 1.                             !Fish light satiation (umol p
m-2 s-1)
    real*8, parameter:: VisFieldShape = 0.5         !Fraction of visual field effectively
scanned
    real*8, parameter:: MinVisRange = 0.01          !Minimum detection range non-visual
cues - assume herring detect
    !real*8, parameter:: Scale_ing = 0.5            !Scaling for overlapping search
fields, capture efficiency..
    ! Prey parameters
    real*8, parameter:: J_Cal     = 4.184             !Joules per 1 calorie

    real*8 BeamAtt, PreyPer_m3_1, PreyPer_m3_2, PreyPer_m3_scaler1, PreyPer_m3_scaler2,
Seasonal_PreyPer_m3_scaler, FishLength_m, Psize_scaler1, Psize_scaler2, PreyContrast,
Prey_energy_scaler, &
    Handling_scaler1, Handling_scaler2, Ingestion_scaler  !  !in
    real*8 pi, Prey_mm, Swim_vel, EyeSens  !in subroutine
    real*8 Prey_J_Consumed,Tot_prey_weight_cons !out

    ! In
    real*8 :: Eb(1:2)
    real*8 :: PreyLength(1:2)
    real*8 :: PreyLength_mm(1:2)
```

```
    real*8 :: PreyWidth(1:2)
    real*8 :: Prey_energy_Cal_gdw(1:2)
    !real*8 :: PreyPer_m3(1:2)
    real*8 :: PreyHandlingTime(1:2)
    real*8 :: Scale_prey_diet(1:2)
    ! Inside
    real*8 :: Scale_ing(1:2)
    real*8 :: PreyPer_m3(1:2)
    real*8 :: Psize_scaler(1:2)
    real*8 :: PreyPer_m3_scaler(1:2)
    real*8 :: Prey_mgdw(1:2)
    real*8 :: PreyImageArea(1:2)
    real*8 :: Prey_J(1:2)
    real*8 :: Prey_abundance(1:2)
    real*8 :: VisualRange(1:2)
    real*8 :: Clearance_Rate(1:2)
    real*8 :: n(1:2)
    real*8 :: Prey_J_Enc(1:2)
    ! Out
    real*8 :: Prey_g_Enc(1:2)

    ! Variables
    integer   IER, no
    pi=acos(-1.)

    ! Scaling for overlapping search fields, capture efficiency, schooling etc.
    Scale_ing(1) = 0.3
    Scale_ing(2) = 0.1

    ! Prey abundance
    PreyPer_m3(1)= PreyPer_m3_1
    PreyPer_m3(2) = PreyPer_m3_2

    ! Scalers
    Psize_scaler(1) = Psize_scaler1
    Psize_scaler(2) = Psize_scaler2

    PreyPer_m3_scaler(1) = PreyPer_m3_scaler1
    PreyPer_m3_scaler(2) = PreyPer_m3_scaler2

    Prey_g_Enc(:) = 0.
    Prey_J_Enc(:) = 0.

    ! Loop over different prey types
    do no = 1,size(PreyLength)

        ! Prey properties
        Prey_mm = Psize_scaler(no) * PreyLength_mm(no)
        Prey_mgdw(no) = (10**(2.50*log10(1000*Prey_mm)-6.51))/1E3  !Prey body mass (mg dw)
- from Uye, 1982 (used in van Deurs, 2015, but other version)
        PreyImageArea(no) = Psize_scaler(no) * PreyLength(no) * PreyWidth(no) * 0.75
!Elongated prey 0.75
        Prey_J(no) = Prey_energy_scaler * Prey_energy_Cal_gdw(no) * Prey_mgdw(no)*0.001 *
J_Cal      !Prey energy content [J]
        ! Prey abundance
        Prey_abundance(no) = PreyPer_m3_scaler(no) * Scale_prey_diet(no) *
(Seasonal_PreyPer_m3_scaler*PreyPer_m3(no) + 0.1*PreyPer_m3(no))   !PreyPer_m3_scaler has
been reduced by prop of max PPm3 assumed at start of feeding season
```

```
        !Predator properties
        Swim_vel = FishLength_m * 1.5                !Blaxter 1966

        ! Eye sensitivity - assumes fish detect prey (VisRtoSDPreyinBL x Larval_m) away in
clear water - Here eye sensitivity is a constant for each fish length:
        EyeSens = ((FishLength_m*VisRtoSDPreyinBL)**2.) /
(EyeSensPreyContrast*EyeSensPreyLength*EyeSensPreyWidth*0.75)

        ! Visual range in m (from AU97)
        VisualRange(no) = sqrt(EyeSens * PreyContrast * PreyImageArea(no) *
(Eb(no)/(Ke+Eb(no))))     !Approximation
        VisualRange(no) = max(VisualRange(no), MinVisRange)

        if(VisualRange(no) > 0.05)then    !Exact visual range (above ca 5 cm)
            call getr(VisualRange(no), BeamAtt, PreyContrast, PreyImageArea(no), EyeSens,
Ke, Eb(no), IER)
        endif

        Clearance_Rate(no) = VisFieldShape * pi * (VisualRange(no)**2) * Swim_vel   !In
m3/s

        enddo ! no loop over number of prey items


        ! Ingestion rates in number of prey items/s and energy/s (including handling time
limitation - Holling disc 2 prey types)
        n(1) = Clearance_Rate(1) * Prey_abundance(1) / (1. +
(PreyHandlingTime(1)*Handling_scaler1)*Clearance_Rate(1)*Prey_abundance(1) &
        + (PreyHandlingTime(2)*Handling_scaler2)*Clearance_Rate(2)*Prey_abundance(2))
!Total number of prey items ingested/s (items/s)
        n(2) = Clearance_Rate(2) * Prey_abundance(2) / (1. +
(PreyHandlingTime(1)*Handling_scaler1)*Clearance_Rate(1)*Prey_abundance(1) &
        + (PreyHandlingTime(2)*Handling_scaler2)*Clearance_Rate(2)*Prey_abundance(2))
!Total number of prey items ingested/s (items/s)

        do no = 1,size(PreyLength)

        n(no) = n(no) * (Scale_ing(no) * Ingestion_scaler)    !Calibration of feeding
limitations - overlapping search fields, capture efficiency, schooling etc.
        Prey_g_Enc(no) = n(no) * (Prey_mgdw(no)/0.13) * 1000  !Total grams prey ingested
(g ww)
        Prey_J_Enc(no) = n(no) * Prey_J(no)                        !Total energy
ingested/s (J/s) - n(items/s)*Prey_J(joules/item)

        enddo ! no loop over number of prey items


    ! Total prey energy consumed for all prey types
    Prey_J_Consumed = Prey_J_Enc(1) + Prey_J_Enc(2)

    !Total weight consumed - use for comparison of estimated intake proportion by prey
with real estimates
    Tot_prey_weight_cons = Prey_g_Enc(1) + Prey_g_Enc(2)

    return
    end subroutine ingestion
```

1.2.2 BioEnergetics

```
subroutine BioEnergetics(Temperature,FishLength_cm,DailyDig,DayResp,DayNetIng,DaySurpE)
implicit none

! Parameters
real*8, parameter:: alpha = 0.0033*13.560 ! Intercept of allometric weight function =
0.0033 gO2/gWW/day * 13.560 kJ/gO2 --> kJ/(gWW*day)
real*8, parameter:: RB = -0.227           ! Slope of allometric weight function (Rudstam
88, based on De Silva and Balbontin 74, 0-group Atlantic herring)
real*8, parameter:: RQ = 0.0548           ! Water temperature dependence coefficient
(Rudstam 88, based on alewife)
real*8, parameter:: RTO = 0.03            ! Swimming speed dependence coefficient (Rudstam
88,based on aholehole Muir and Niimi 72, the same is used in alewife models)
real*8, parameter:: SDA = 0.175           ! Coefficient for specific dynamic action
(Rudstam 88)
real*8, parameter:: ACT = 3.9             ! Intercept of relationship for swimming speed
vs. weight and temperature (Rudstam 88, for temp < 9)
real*8, parameter:: K4 = 0.13             ! Slope for weight dependence of swimming speed
at all water temperatures (Rudstam 88, based on Ware 78)
real*8, parameter:: BACT = 0.149   ! Coefficient for temperature dependence of swimming
speed at temp below 9 (Rudstam 88, based on alewife)
real*8, parameter:: FA = 0.16             ! Proportion of consumed food egested (Rudstam
88)
real*8, parameter:: UA = 0.10             ! Proportion of assimilated food excreted
(Rudstam 88)
real*8, parameter:: beta = 0.4            ! Calibrated beta to yield output in realistic
range
real*8, parameter:: SW = 0.77             ! Slope for weight dependence of swimming
velociy and cost ('The ecological implications of body size' Robert Henry Peters)

! Variables
real*8 Temperature,FishLength_cm,DailyDig !in
real*8 Weight,SwimVelcms,DayRespRate,Temp_SMR,Swim_cost_Js,Swim_cost !in routine
real*8 DayResp,DayNetIng,DaySurpE !out

! General length/weight relationship for Atlantic herring (from ICES)
Weight = 0.00603 * FishLength_cm**3.0904

! Respiration cost for day - depends on body weight, temperature, and weight-dependent
swimmig cost
Temp_SMR = (alpha*Weight**RB) * exp(RQ*Temperature) * Weight !Temp-dependent SMR (kJ/day)
Swim_cost = (alpha*Weight**RB) * Weight * 0.75
DayResp = Temp_SMR + Swim_cost  !Respiration cost per day (kJ)

! Energy ingested for day, assimilation costs subtracted
DayNetIng = DailyDig - ((FA*DailyDig) + UA*(DailyDig-(FA*DailyDig)) + SDA*(DailyDig-
(FA*DailyDig)))

! Surplus energy for day, digestion and respiration costs subtracted from ingested energy
(incl. dig. lim.) (kJ)
DaySurpE = DayNetIng - DayResp

return
end subroutine BioEnergetics
```

1.2.3 BioEnergetics_Winter

```
subroutine BioEnergetics_Winter(Temperature,FishLength_cm,WResp)
implicit none

! Parameters
real*8, parameter:: alpha = 0.0033*13.560 ! Intercept of allometric weight function in
gO2/gWW^beta/day (Rudstam 88, alewife) X kJ/gO2
real*8, parameter:: RB = -0.227    ! Slope of allometric weight function (Rudstam 88,
based on De Silva and Balbontin 74, 0-group Atlantic herring)
real*8, parameter:: RQ = 0.0548    ! Water temperature dependence coefficient (Rudstam 88,
based on alewife)
real*8, parameter:: RTO = 0.03     ! Swimming speed dependence coefficient (Rudstam
88,based on aholehole Muir and Niimi 72, the same is used in alewife models)
real*8, parameter:: SDA = 0.175    ! Coefficient for specific dynamic action (Rudstam 88)
real*8, parameter:: ACT = 3.9      ! Intercept of relationship for swimming speed vs.
weight and temperature (Rudstam 88, for temp < 9)
real*8, parameter:: K4 = 0.13      ! Slope for weight dependence of swimming speed at all
water temperatures (Rudstam 88, based on Ware 78)
real*8, parameter:: BACT = 0.149   ! Coefficient for temperature dependence of swimming
speed at temp below 9 (Rudstam 88, based on alewife)
real*8, parameter:: FA = 0.16      ! Proportion of consumed food egested (Rudstam 88)
real*8, parameter:: UA = 0.10      ! Proportion of assimilated food excreted (Rudstam 88)
real*8, parameter:: beta = 0.4
real*8, parameter:: SW = 0.77      ! Slope for weight dependence of swimming velociy and
cost ('The ecological implications of body size' Robert Henry Peters)

! Variables
real*8 Temperature,FishLength_cm!in
real*8 Weight,SwimVelcms,Resp,Temp_SMR,Swim_cost_Js,Swim_cost !inside
real*8 WResp !out

! General length/weight relationship for Atlantic herring (from ICES)
Weight = 0.00603 * FishLength_cm**3.0904

! Respiration cost for day - depends on body weight, temperature, and weight-dependent
swimmig cost
Temp_SMR = (alpha*Weight**RB) * exp(RQ*Temperature) * Weight !Temp-dependent SMR (kJ/day)
Swim_cost = (alpha*Weight**RB) * Weight * 0.1
WResp = Temp_SMR + Swim_cost                                    !Respiration cost per
day (kJ)

return
end subroutine BioEnergetics_Winter
```

1.2.4 PreyAbundanceNWS

```
    !Gives prey adundance scaler for day of the year
    subroutine PreyAbundanceNWS(PPm3_scaler)
    implicit none

    ! Parameters
    real*8, parameter:: sigma = 0.4
    real*8, parameter::  mu = 2
    real*8, parameter:: feed_day_min = 91 !1 April
    real*8, parameter:: prey_day_peak = 166 !15 June
    real*8, parameter:: feed_day_max = 244 !1 Sept
```

```
    !Variables
    integer i, j
    integer index_max_x(1)
    real*8 days_season, pi, max_scaled, pdf_max, pdfscaler, min_to_peak, xscaler
    !Arrays
    real*8 :: x(1:201)
    real*8 :: pdf(1:201)
    real*8 :: pdf_scaled(1:201)
    real*8 :: new_x(1:201)
    real*8 :: PPm3_scaler(91:244) !out

    pi = 4.0*atan(1.0)
    days_season = feed_day_max - feed_day_min
    x(:) = (/0:400:2/)
    x(:) = x(:)*0.01

    ! Create pdf curve
    do i = 1,size(x)
        pdf(i) = (1/sqrt(2*pi*sigma**2)) * exp(-((x(i)-mu)**2)/(2*sigma**2))
        write(10,'(E15.5,E15.5,18E15.5)') x(i), pdf(i)
    enddo

    ! Location of max value of x for scaling
    index_max_x(:) = maxloc(pdf)

    ! Scale x-axis according to peak day and season days
    min_to_peak = feed_day_max - feed_day_min
    xscaler = min_to_peak/(maxval(x)-minval(x))

    ! Scale y-axis to max value
    max_scaled = 1 - 0.1*1      !Max value of curve - start season with 10% of max food
abundance
    pdf_max = (1/sqrt(2*pi*sigma**2)) * exp(-((x(index_max_x(1))-mu)**2)/(2*sigma**2))
    pdfscaler = max_scaled/pdf_max    ! scaler for p, such that pscaler*pdf(mode) =
pdensity

    do j = 1,size(x)
        pdf_scaled(j) = pdfscaler * ((1/sqrt(2*pi*sigma**2)) * exp(-((x(j)-
mu)**2)/(2*sigma**2)))
        new_x(j) = x(j)*xscaler
    enddo

    !Output values for feeding season day 244 - 91 = 153 days long
    PPm3_scaler(91:244) = pdf_scaled(1:154)

    return
end subroutine PreyAbundanceNWS
```

3. North Sea sensitivity analysis

1. Main program

```
!***************************************************************************
!
!  PROGRAM: NS_sensitivity
!
! Sensitivity analysis for NS herring with NS parameter values as default and scaling of
! these values over loops (0.75, 1, 1.25).
! In default scenario the intake is scaled according to proportions by weight in diet from
! data using parameter 'Scale_prey_diet'.
!
!***************************************************************************

    program NS_sensitivity

    !Module for printing results to binary files
    use binary_IO

    implicit none


    ! Parameters
    integer   day,hour
    real*8 day_real
    integer, parameter :: Days_In_Year = 365, dt_per_Day = 24 !240
    integer, parameter :: fminday = 91, fmaxday = 213, wminday = 214, wmaxday = 455
!365+90  Days in foraging and winter season
    real*8,  parameter :: DayLigThresh = 0.1
    real*8,  parameter :: clouds = 0.5    !Cloud cover fraction
    !real*8, parameter :: minTemp = 4
    !Bioenergetics
        real*8, parameter:: J_Cal  = 4.184             !Joules per 1 calorie
    real*8, parameter:: alpha = 0.0033*13.560   ! Intercept of allometric weight function
= 0.0033 gO2/gWW/day * 13.560      kJ/gO2 --> kJ/(gWW*day)
    real*8, parameter :: RQ = 0.0548          ! Temperature dependence coefficient
respiration (Rudstam 88, based on alewife)
    real*8, parameter :: RB = -0.227          ! Slope of allometric weight function
(Rudstam 88, based on De Silva and Balbontin 74, 0-group Atlantic herring)
    !Gastric rates
    real*8, parameter :: kJ_gww_fish = 10   !kJ per g wet weight of herring - used in
Baulier, mean from Varpe, 2005
    real*8, parameter :: dw_ww = 0.13       !Dry weight to wet weight ratio, Mullin 1969
in Rudstan 1988
    !real*8, parameter :: GA = 1.E-4        !From Rosland & Giske, 1994
    !real*8 GA                               !Calculate GA according to scaling
    !real*8, parameter :: GB = 0.0693        !From Rosland & Giske, 1994
    real*8, parameter :: K = 0.5            !Digestion scaling factor

    ! Variables
    real*8 h, dh, SunHeight, Eb, SurfLig, Seasondays, BeamAtt
    real*8 pi, rlat, radmax, ssurf, altdeg, VisualRange
    real*8, dimension(91:244) :: Seasonal_PPm3_scaler  !PreyAbund subroutine - scales prey
abundance according to seasonal curve
    real*8, dimension(2*365) :: Temp  !Temperture subroutine
```

```
    real*8 SurpE, Tot_Surplus, IngE, DaySurpE, DayNetIng, DayResp, Tot_Resp !Bioenergetics
subroutine
    real*8 WResp, Tot_WResp, Growth_pot, Tot_WinterResp !Winter_Bioenergetics subroutine
    real*8 VisRtoSDPreyinBL, VisRange, EyeSens, PreyImageArea, Prey_J_Enc,
Tot_prey_weight_cons, Day_Prey_weight, Tot_Prey_weight  !Ingestion subroutine
    real*8 Clearance_Rate, Day_Clearance, Tot_Clearance, Day_Prey_items, Tot_n, Prey_J,
Day_Consum, Tot_Consum, Tot_NetIng   !Ingestion subroutine
    real*8 Spec_GastricEvac, GastricEvacRate

    !! Parameters (size array)
    ! Fish characteristics
    integer :: r = 36
    real*8, dimension(1:36) :: Length_cm = [(r, r=10,45,1)]
    real*8 :: WWg(1:36)
    real*8 :: Gut(1:36)
    real*8 :: MaxGut(1:36)
    real*8 :: MaxGut_g(1:36)
    real*8 :: Consumption
    real*8 :: Digestion_s(1:36)
    real*8 :: Digestion(1:36)
    real*8 :: DailyDigestion(1:36)
    real*8 :: Tot_Digestion(1:36)
    real*8 :: Gut_gww_p1(1:36)
    real*8 :: Gut_gww_p2(1:36)
    real*8 :: Gut_gdw_p1(1:36)
    real*8 :: Gut_gdw_p2(1:36)
    real*8 :: Gut_kJ_p1(1:36)
    real*8 :: Gut_kJ_p2(1:36)
    real*8 :: MaxGut_kJ(1:36)
    ! Prey characteristics; 1 = Calanus finmarchicus & helgolandicus, 2 = Temora & Acartia
    real*8, dimension(1:2) :: PreyLength =  [0.0026, 0.0144] !m
    real*8, dimension(1:2) :: PreyLength_mm = [0.0026*1000, 0.0144*1000] !mm
    real*8, dimension(1:2) :: PreyWidth = [0.0026/4, 0.0144/4] !m
    real*8 :: Prey_mgdw(1:2) !Calculated
    real*8, dimension(1:2) :: PreyEnergyDensity = [6400, 5200] !cal/g dry weight
    !real*8, dimension(1:2) :: Prey_Abundance = [473, 2]
    real*8, dimension(1:3) :: Prey_Abundance1 = [591,473,394]
    real*8, dimension(1:3) :: Prey_Abundance2 = [2.5,2.,1.67]
    real*8, dimension(1:2) :: PreyHandlingTime = [1.5, 5.]
    ! Scale_prey_diet = Scaling factor for diet such that Cf and Ch constitue 90 % of
consumed dry weight and A&T 10 %...
    !...Values stored in array loop_output_summer7(L,1:2)  = Tot_Prey_g_Enc(1:2) and
loop_output_summer6(L)  = Tot_Prey_weight
    real*8, dimension(1:2) :: Scale_prey_diet = [1., 0.16]
    ! Output
    real*8 :: Day_Prey_g_Enc(1:2)
    real*8 :: Tot_Prey_g_Enc(1:2)
    real*8 :: Prey_g_Enc(1:2)

    ! Parameters for sensitivity analysis
    real*8, dimension(1:3) :: Preysize_scaler1 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Preysize_scaler2 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Energy_density_scaler = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Prey_Abund_scaler1 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Prey_Abund_scaler2 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Turb_scaler = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Handling_scaler1 = [0.8, 1., 1.2]
    real*8, dimension(1:3) :: Handling_scaler2 = [0.8, 1., 1.2]
```

```
    real*8, dimension(1:3) :: Ingestion_scaler = [0.8, 1., 1.2]

    !! Parameters (values)
    ! Prey characteristics
    real*8 :: PreyContrast = 0.3
    ! Light
    real*8, dimension(1:3) :: Latitude_Deg = [48, 58, 68]
    real*8 :: Diff_att = 0.2121         !Diffuse attenuation
    real*8 :: Beam_att = 0.6044          !Beam attenuation
    ! Forage depth
    real*8 :: Forage_depth = 20 !m
    ! Temperature
    !real*8 MaxTemp
    !real*8, dimension(1:3) :: MinTemp = [4, 4, 4] !degrees C
    real*8, dimension(1:3) :: MinTemp = [2, 4, 6] !degrees C
    real*8, dimension(1:3) :: MaxTemp = [12, 14, 16] !degrees C
    !real*8, dimension(1:3) :: MaxTemp = [14, 14, 14] !degrees C
    !real*8 :: MinTemp = 4
    !real*8 :: MaxTemp = 7

    ! Array index names
    real*8 L, Lat, Cal_gdw, PSize1, PSize2, PPm3_1, PPm3_2, MaxT, Da, Hand1, Hand2, Ing

    ! Output matrices (L,PSize1,PSize2,Lat,Temp,PPm3,Cal_gdw,Da,Hand,Ing)
    real*8,  dimension(36, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3) :: loop_output_summer1,
loop_output_summer2, loop_output_summer3, loop_output_summer4,&
        loop_output_summer5, loop_output_summer6, loop_output_winter


    ! Variable calculations
    pi = 4.0*atan(1.0)
    !rlat = -(Latitude+180)*(pi/180)
    dh = 24./real(dt_per_Day)      !Hours per timestep = 0.1 --> NOW dt_per_Day = 24 -->
dh = 1


    ! Run simulations over parameters in sensitivity analysis

    !!! FEEDING SEASON PART !!!!

    ! Get temperature and prey abundance for each day of the year
    call PreyAbundanceNS(Seasonal_PPm3_scaler)

    do Hand1 = 1,size(Handling_scaler1)
    do Hand2 = 1,size(Handling_scaler2)

        do Ing = 1,size(Ingestion_scaler)

            do MaxT = 1,size(MaxTemp)

                call Temperature(MinTemp(MaxT),MaxTemp(MaxT),Temp)

                do Da = 1,size(Turb_scaler)

                    BeamAtt = Turb_scaler(Da)*Beam_att  !Calculate scaled beam attenuation
value

                    do Cal_gdw = 1,size(Energy_density_scaler)
```

```
                        do PPm3_1 = 1,size(Prey_Abund_scaler1)
                        do PPm3_2 = 1,size(Prey_Abund_scaler2)

                            do Lat =1,size(Latitude_Deg)
                                rlat = -(Latitude_Deg(Lat)+180)*(pi/180)

                                do PSize1 = 1,size(Preysize_scaler1)
                                do PSize2 = 1,size(Preysize_scaler2)

                                    ! Run simulation over range of adult herring lengths
                                    do L = 1,size(Length_cm)   !Loop over size

                                                                ! General length/weight relationship
for Atlantic herring (from ICES)
                                                                WWg(L) = 0.00603 *
Length_cm(L)**3.0904
                                                                MaxGut_g(L) = 0.03*WWg(L)           !
Max gut in grams, 3% of weight/size
                                                                Gut_gww_p1(L) = 0.6*MaxGut_g(L)
                                                                Gut_gww_p2(L) = 0.4*MaxGut_g(L)
                                                                Gut_gdw_p1(L) = 0.13*Gut_gww_p1(L)
                                                                Gut_gdw_p2(L) = 0.13*Gut_gww_p2(L)
                                                                Gut_kJ_p1(L) = PreyEnergyDensity(1) *
Gut_gdw_p1(L) * J_Cal      !Prey energy content [J]
                                                                Gut_kJ_p2(L) = PreyEnergyDensity(2) *
Gut_gdw_p2(L) * J_Cal      !Prey energy content [J]
                                                                MaxGut_kJ(L) = Gut_kJ_p1(L) +
Gut_kJ_p2(L)  ! Max gut in kJ

                                                !EnergyFatInd_kJ = 100.012*exp(0.11*Length_cm(L))
!Energy content pr ind (fat) (Slotte 1999, Slotte & Fiksen 2000)

                                                ! Sums over feeding season
                                                Tot_Consum = 0.
                                                Tot_n = 0.
                                                Tot_Surplus = 0.
                                                Tot_WResp = 0.
                                                Tot_Digestion(L) = 0.
                                                Tot_NetIng = 0.
                                                Tot_Resp = 0.
                                                Tot_Prey_g_Enc(:) = 0.
                                                Tot_Prey_weight = 0.

                                                Gut(L) = 0.

                                                ! Ingestion loop over days in feeding season
                                                Do day = fminday, fmaxday

                                                    ! Daily sums

                                                    DailyDigestion(L) = 0.
                                                    Day_Prey_items = 0.
                                                    Day_Consum = 0.
                                                    Day_Prey_g_Enc(:) = 0.
                                                    Day_Prey_weight = 0.
```

```
                                        !Ingestion loop over hours for each day of the
feeding season
                                        Do hour = 1, dt_per_Day
                                            h = dh*real(hour) ! from 0 to 24 by 0.1 -
NOW by 1

                                            day_real = real(day)

                                            ! Subroutine gets max irradiance for time,
day, latitude, cloud cover (HYCOM qsw0 modified by Anders F. Opdal)
                                            call
qsw0_hr(radmax,ssurf,altdeg,clouds,rlat,day_real,hour,Days_in_year)

                                            ! Light at foraging depth
                                            Eb = ssurf*exp(-
Diff_att*Turb_scaler(Da)*Forage_depth)
                                            Eb = Eb/4.6*0.45*3

                                            ! Subroutine gets ingestion estimates over
hours for each day of the feeding season
                                            call
ingestion(Eb,Beam_att,Prey_Abundance1(PSize1),Prey_Abundance2(PSize2),Prey_Abund_scaler1(P
Pm3_1),Prey_Abund_scaler2(PPm3_2),Seasonal_PPm3_scaler(day),PreyHandlingTime(:),&

Scale_prey_diet(:),Length_cm(L)*0.01,PreyLength(:),PreyLength_mm(:),PreyWidth(:),Preysize_
scaler1(PSize1),Preysize_scaler2(PSize2),PreyContrast,&

PreyEnergyDensity(:),Energy_density_scaler(Cal_gdw),Handling_scaler1(Hand1),Handling_scale
r2(Hand2),Ingestion_scaler(Ing),Prey_J_Enc,Prey_g_Enc,Tot_prey_weight_cons)

                                            ! Daily net consumed energy, i.e. daily
digested energy

                                            !! Digestion dependent on both size and
temperature - used K scaled to fit Marion's data
                                            ! Eats
                                            Consumption = Prey_J_Enc * 3600. * dh *
1E-3            !Consumed energy (kJ) - Prey_J_Enc(J/s)*3600s/h*1h*0.001kJ/J --> kJ
                                            Gut(L) = max(0., min(Gut(L) + Consumption,
MaxGut_kJ(L)))    !New gut content (kJ) after consumption
                                            ! Digests part of what is in gut
                                            Spec_GastricEvac = 10 * alpha * WWg(L)**RB
* exp(RQ*Temp(day))    !Weight specific evacuation rate (kJ/(gWW*day) - units of K)
                                            GastricEvacRate = Spec_GastricEvac *
WWg(L) / 24        !Evacuation rate (kJ/h)
                                            Digestion(L) = min(GastricEvacRate*dh,
Gut(L))          !Digestion per timestep (kJ)
                                            ! Have left in gut what is not digested
                                            Gut(L) = max(0., Gut(L) - Digestion(L))
!New gut content (kJ) after digestion

                                            ! Daily sum of digestion - i.e.
consumption with digestion limitation
                                            DailyDigestion(L) = DailyDigestion(L) +
Digestion(L)     !Daily digested energy (kJ) --> input Bioenergetics subroutine

                                            !! Daily sums of ingestion estimates - no
digestion limitation
```

```
                                        !Day_Prey_items =  Day_Prey_items +
n*dh*3600.
                                        Day_Prey_g_Enc(1) = Day_Prey_g_Enc(1) +
Prey_g_Enc(1)
                                        Day_Prey_g_Enc(2) = Day_Prey_g_Enc(2) +
Prey_g_Enc(2)
                                        Day_Prey_weight = Day_Prey_weight +
Tot_prey_weight_cons
                                        Day_Consum = Day_Consum + Consumption

                                    enddo !End hour loop - time steps

                                    Call
BioEnergetics(Temp(day),Length_cm(L),DailyDigestion(L),DayResp,DayNetIng,DaySurpE)

                                    ! Feeding season total surplus energy (sum of
daily sum DaySurpE)

                                    Tot_Resp = Tot_Resp + DayResp
                                    Tot_NetIng = Tot_NetIng + DayNetIng !Total
digested energy - SDA -EX -EG
                                    Tot_Surplus = Tot_Surplus + DaySurpE
!Digestion and respiration costs subtracted from ingested energy (incl. dig. lim.)(kJ)

                                    !! Feeding season sums from ingestion
subroutine (sum of daily sums - Used in previous script!)
                                    !Tot_n = Tot_n + Day_Prey_items  ! Total prey
items ingested (before digestion limitation has been imposed)
                                    Tot_Prey_g_Enc(1) = Tot_Prey_g_Enc(1) +
Day_Prey_g_Enc(1)
                                    Tot_Prey_g_Enc(2) = Tot_Prey_g_Enc(2) +
Day_Prey_g_Enc(2)
                                    Tot_Prey_weight = Tot_Prey_weight +
Day_Prey_weight
                                    Tot_Consum = Tot_Consum + Day_Consum ! Sum of
daily consumed energy --> Total energy ingested over feeding season (kJ) - no digestion
limitation
                                    Tot_Digestion(L) = Tot_Digestion(L) +
DailyDigestion(L)

                                enddo !End day loop - daily foraging cycle


loop_output_summer1(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_Digestion(L)  !(L,PSize,Lat,Temp,PPm3,Cal_gdw,Da)

loop_output_summer2(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_Resp

loop_output_summer3(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_Consum

loop_output_summer4(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_NetIng

loop_output_summer5(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_Surplus
```

```
loop_output_summer6(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing)  =
Tot_n



                                        !!!! WINTER SEASON PART !!!!

                                        Tot_WResp = 0.

                                        do day = wminday, wmaxday

                                            Call
BioEnergetics_Winter(Temp(day),Length_cm(L),WResp)

                                            ! Total winter respiration cost (sum of daily
sums WResp)
                                            Tot_WResp = Tot_WResp + WResp ! kJ

                                        enddo !End day loop

                                        ! Growth potential - suplus energy left after
winter = total surplus energy at end of feeding season - total repiration energy over
winter
                                        Growth_pot = Tot_Surplus - Tot_WResp


loop_output_winter(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Cal_gdw,Da,Hand1,Hand2,Ing) =
Growth_pot ! (kJ) changes with feeding season surplus energy


                                    enddo !End L
                                enddo !End Psize1
                                enddo !End Psize2
                            enddo !End Lat
                        enddo !PPm3_1
                        enddo !PPm3_2
                    enddo !Cal_gdw
                enddo !Diff_att
            enddo !MaxT
        enddo !Hand1
        enddo !Hand2
    enddo !Ing

    !Save output matrices to binary files
    !Save output matrices to binary files
    call array_to_binary(loop_output_summer1, 'TotDig_annual_Temp_NS_NEW.bin')
    call array_to_binary(loop_output_summer2, 'TotResp_annual_Temp_NS_NEW.bin')
    call array_to_binary(loop_output_summer3, 'Tot_Consum_annual_Temp_NS_NEW.bin')
    call array_to_binary(loop_output_summer4, 'Tot_NetIng_annual_Temp_NS_NEW.bin')
    call array_to_binary(loop_output_summer4, 'Tot_NetIng_annual_Temp_NS_NEW.bin')
    call array_to_binary(loop_output_summer5, 'TotSurp_annual_Temp_NS_NEW.bin')
    call array_to_binary(loop_output_summer6, 'Tot_n_annual_Temp_NS_NEW.bin')
    call array_to_binary(loop_output_winter, 'Growth_pot_annual_Temp_NS_NEW.bin')

    end program NS_sensitivity
```

1.2 Subroutines

1.2.1 ingestion

```
    subroutine ingestion(Eb, BeamAtt, PreyPer_m3_1, PreyPer_m3_2, PreyPer_m3_scaler1,
PreyPer_m3_scaler2, Seasonal_PreyPer_m3_scaler, PreyHandlingTime, Scale_prey_diet,
FishLength_m, &
        PreyLength, PreyLength_mm, PreyWidth, Psize_scaler1, Psize_scaler2, PreyContrast,
Prey_energy_Cal_gdw, Prey_energy_scaler, Handling_scaler1, Handling_scaler2, &
        Ingestion_scaler, Prey_J_Consumed, Prey_g_Enc, Tot_prey_weight_cons)

    implicit none

    ! Parameters
    real*8, parameter:: EyeSensPreyLength = 0.004 !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: EyeSensPreyWidth = 0.001  !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: EyeSensPreyContrast = 0.3 !Constant for eye sensitivity
calculation  (Blaxter)
    real*8, parameter:: VisRtoSDPreyinBL = 1.       !Detection distance in BL of small
prey at satiating light and clear water
    real*8, parameter:: Ke = 1.                             !Fish light satiation (umol p
m-2 s-1)
    real*8, parameter:: VisFieldShape = 0.5         !Fraction of visual field effectively
scanned
    real*8, parameter:: MinVisRange = 0.01          !Minimum detection range non-visual
cues - assume herring detect
    !real*8, parameter:: Scale_ing = 0.5            !Scaling for overlapping search
fields, capture efficiency..
    ! Prey parameters
    real*8, parameter:: J_Cal     = 4.184             !Joules per 1 calorie

    real*8 Eb, BeamAtt, PreyPer_m3_1, PreyPer_m3_2, PreyPer_m3_scaler1,
PreyPer_m3_scaler2, Seasonal_PreyPer_m3_scaler, FishLength_m, Psize_scaler1,
Psize_scaler2, PreyContrast, Prey_energy_scaler, Handling_scaler1, &
    Handling_scaler2, Ingestion_scaler !  !in
    real*8 pi, Prey_mm, Swim_vel, EyeSens  !in subroutine
    real*8 Prey_J_Consumed,Tot_prey_weight_cons !out

    ! In
    real*8 :: PreyLength(1:2)
    real*8 :: PreyLength_mm(1:2)
    real*8 :: PreyWidth(1:2)
    real*8 :: Prey_energy_Cal_gdw(1:2)
    !real*8 :: PreyPer_m3(1:2)
    real*8 :: PreyHandlingTime(1:2)
    real*8 :: Scale_prey_diet(1:2)
    ! Inside
    real*8 :: Scale_ing(1:2)
    real*8 :: PreyPer_m3(1:2)
    real*8 :: Psize_scaler(1:2)
    real*8 :: PreyPer_m3_scaler(1:2)
    real*8 :: Prey_mgdw(1:2)
    real*8 :: PreyImageArea(1:2)
```

```
    real*8 :: Prey_J(1:2)
    real*8 :: Prey_abundance(1:2)
    real*8 :: VisualRange(1:2)
    real*8 :: Clearance_Rate(1:2)
    real*8 :: n(1:2)
    real*8 :: Prey_J_Enc(1:2)
    ! Out
    real*8 :: Prey_g_Enc(1:2)

    ! Variables
    integer   IER, no
    pi=acos(-1.)

    ! Scaling for overlapping search fields, capture efficiency, schooling etc.
    Scale_ing(1) = 0.5
    Scale_ing(2) = 0.3

    ! Prey abundance
    PreyPer_m3(1)= PreyPer_m3_1
    PreyPer_m3(2) = PreyPer_m3_2

    ! Scalers
    Psize_scaler(1) = Psize_scaler1
    Psize_scaler(2) = Psize_scaler2

    PreyPer_m3_scaler(1) = PreyPer_m3_scaler1
    PreyPer_m3_scaler(2) = PreyPer_m3_scaler2

    Prey_g_Enc(:) = 0.
    Prey_J_Enc(:) = 0.

    ! Loop over different prey types
    do no = 1,size(PreyLength)

        ! Prey properties
        Prey_mm = Psize_scaler(no) * PreyLength_mm(no)
        Prey_mgdw(no) = (10**(2.50*log10(1000*Prey_mm)-6.51))/1E3  !Prey body mass (mg dw)
- from Uye, 1982 (used in van Deurs, 2015, but other version)
        PreyImageArea(no) = Psize_scaler(no) * PreyLength(no) * PreyWidth(no) * 0.75
!Elongated prey 0.75
        Prey_J(no) = Prey_energy_scaler * Prey_energy_Cal_gdw(no) * Prey_mgdw(no)*0.001 *
J_Cal      !Prey energy content [J]
        ! Prey abundance
        Prey_abundance(no) = PreyPer_m3_scaler(no) * Scale_prey_diet(no) *
(Seasonal_PreyPer_m3_scaler*PreyPer_m3(no) + 0.2*PreyPer_m3(no))   !PreyPer_m3_scaler has
been reduced by prop of max PPm3 assumed at start of feeding season

        !Predator properties
        Swim_vel = FishLength_m * 1.5                !Blaxter 1966

        ! Eye sensitivity - assumes fish detect prey (VisRtoSDPreyinBL x Larval_m) away in
clear water - Here eye sensitivity is a constant for each fish length:
        EyeSens = ((FishLength_m*VisRtoSDPreyinBL)**2.) /
(EyeSensPreyContrast*EyeSensPreyLength*EyeSensPreyWidth*0.75)

        ! Visual range in m (from AU97)
        VisualRange(no) = sqrt(EyeSens * PreyContrast * PreyImageArea(no) * (Eb/(Ke+Eb)))
!Approximation
```

```
        VisualRange(no) = max(VisualRange(no), MinVisRange)

        if(VisualRange(no) > 0.05)then    !Exact visual range (above ca 5 cm)
            call getr(VisualRange(no), BeamAtt, PreyContrast, PreyImageArea(no), EyeSens,
Ke, Eb, IER)
        endif

        Clearance_Rate(no) = VisFieldShape * pi * (VisualRange(no)**2) * Swim_vel   !In
m3/s

        enddo ! no loop over number of prey items

        ! Ingestion rates in number of prey items/s and energy/s (including handling time
limitation - Holling disc 2 prey types)
        n(1) = Clearance_Rate(1) * Prey_abundance(1) / (1. +
(PreyHandlingTime(1)*Handling_scaler1)*Clearance_Rate(1)*Prey_abundance(1) &
        + (PreyHandlingTime(2)*Handling_scaler2)*Clearance_Rate(2)*Prey_abundance(2))
!Total number of prey items ingested/s (items/s)
        n(2) = Clearance_Rate(2) * Prey_abundance(2) / (1. +
(PreyHandlingTime(1)*Handling_scaler1)*Clearance_Rate(1)*Prey_abundance(1) &
        + (PreyHandlingTime(2)*Handling_scaler2)*Clearance_Rate(2)*Prey_abundance(2))
!Total number of prey items ingested/s (items/s)

        do no = 1,size(PreyLength)

        n(no) = n(no) * (Scale_ing(no) * Ingestion_scaler)    !Calibration of feeding
limitations - overlapping search fields, capture efficiency, schooling etc.
        Prey_g_Enc(no) = n(no) * (Prey_mgdw(no)/0.13) * 1000  !Total grams prey ingested
(g ww)
        Prey_J_Enc(no) = n(no) * Prey_J(no)                        !Total energy
ingested/s (J/s) - n(items/s)*Prey_J(joules/item)

        enddo ! no loop over number of prey items


    ! Total prey energy consumed for all prey types
    Prey_J_Consumed = Prey_J_Enc(1) + Prey_J_Enc(2)

    !Total weight consumed - use for comparison of estimated intake proportion by prey
with real estimates
    Tot_prey_weight_cons = Prey_g_Enc(1) + Prey_g_Enc(2)


    return
    end subroutine ingestion
```

1.2.2 BioEnergetics

```
subroutine BioEnergetics(Temperature,FishLength_cm,DailyDig,DayResp,DayNetIng,DaySurpE)
implicit none

! Parameters
real*8, parameter:: alpha = 0.0033*13.560 ! Intercept of allometric weight function =
0.0033 gO2/gWW/day * 13.560 kJ/gO2 --> kJ/(gWW*day)
real*8, parameter:: RB = -0.227          ! Slope of allometric weight function (Rudstam
88, based on De Silva and Balbontin 74, 0-group Atlantic herring)
```

```
real*8, parameter:: RQ = 0.0548          ! Water temperature dependence coefficient
(Rudstam 88, based on alewife)
real*8, parameter:: RTO = 0.03           ! Swimming speed dependence coefficient (Rudstam
88,based on aholehole Muir and Niimi 72, the same is used in alewife models)
real*8, parameter:: SDA = 0.175          ! Coefficient for specific dynamic action
(Rudstam 88)
real*8, parameter:: ACT = 3.9            ! Intercept of relationship for swimming speed
vs. weight and temperature (Rudstam 88, for temp < 9)
real*8, parameter:: K4 = 0.13            ! Slope for weight dependence of swimming speed
at all water temperatures (Rudstam 88, based on Ware 78)
real*8, parameter:: BACT = 0.149   ! Coefficient for temperature dependence of swimming
speed at temp below 9 (Rudstam 88, based on alewife)
real*8, parameter:: FA = 0.16            ! Proportion of consumed food egested (Rudstam
88)
real*8, parameter:: UA = 0.10            ! Proportion of assimilated food excreted
(Rudstam 88)
real*8, parameter:: beta = 0.4           ! Calibrated beta to yield output in realistic
range
real*8, parameter:: SW = 0.77            ! Slope for weight dependence of swimming
velociy and cost ('The ecological implications of body size' Robert Henry Peters)

! Variables
real*8 Temperature,FishLength_cm,DailyDig !in
real*8 Weight,SwimVelcms,DayRespRate,Temp_SMR,Swim_cost_Js,Swim_cost !in routine
real*8 DayResp,DayNetIng,DaySurpE !out

! General length/weight relationship for Atlantic herring (from ICES)
Weight = 0.00603 * FishLength_cm**3.0904

! Respiration cost for day - depends on body weight, temperature, and weight-dependent
swimmig cost
Temp_SMR = (alpha*Weight**RB) * exp(RQ*Temperature) * Weight !Temp-dependent SMR (kJ/day)
Swim_cost = (alpha*Weight**RB) * Weight * 0.75
DayResp = Temp_SMR + Swim_cost  !Respiration cost per day (kJ)

! Energy ingested for day, assimilation costs subtracted
DayNetIng = DailyDig - ((FA*DailyDig) + UA*(DailyDig-(FA*DailyDig)) + SDA*(DailyDig-
(FA*DailyDig)))

! Surplus energy for day, digestion and respiration costs subtracted from ingested energy
(incl. dig. lim.) (kJ)
DaySurpE = DayNetIng - DayResp

return
end subroutine BioEnergetics
```

1.2.3 BioEnergetics_Winter

```
subroutine BioEnergetics_Winter(Temperature,FishLength_cm,WResp)
implicit none

! Parameters
real*8, parameter:: alpha = 0.0033*13.560 ! Intercept of allometric weight function in
gO2/gWW^beta/day (Rudstam 88, alewife) X kJ/gO2
real*8, parameter:: RB = -0.227    ! Slope of allometric weight function (Rudstam 88,
based on De Silva and Balbontin 74, 0-group Atlantic herring)
```

```
real*8, parameter:: RQ = 0.0548    ! Water temperature dependence coefficient (Rudstam 88,
based on alewife)
real*8, parameter:: RTO = 0.03     ! Swimming speed dependence coefficient (Rudstam
88,based on aholehole Muir and Niimi 72, the same is used in alewife models)
real*8, parameter:: SDA = 0.175    ! Coefficient for specific dynamic action (Rudstam 88)
real*8, parameter:: ACT = 3.9      ! Intercept of relationship for swimming speed vs.
weight and temperature (Rudstam 88, for temp < 9)
real*8, parameter:: K4 = 0.13      ! Slope for weight dependence of swimming speed at all
water temperatures (Rudstam 88, based on Ware 78)
real*8, parameter:: BACT = 0.149   ! Coefficient for temperature dependence of swimming
speed at temp below 9 (Rudstam 88, based on alewife)
real*8, parameter:: FA = 0.16      ! Proportion of consumed food egested (Rudstam 88)
real*8, parameter:: UA = 0.10      ! Proportion of assimilated food excreted (Rudstam 88)
real*8, parameter:: beta = 0.4
real*8, parameter:: SW = 0.77      ! Slope for weight dependence of swimming velociy and
cost ('The ecological implications of body size' Robert Henry Peters)

! Variables
real*8 Temperature,FishLength_cm!in
real*8 Weight,SwimVelcms,Resp,Temp_SMR,Swim_cost_Js,Swim_cost !inside
real*8 WResp !out

! General length/weight relationship for Atlantic herring (from ICES)
Weight = 0.00603 * FishLength_cm**3.0904

! Respiration cost for day - depends on body weight, temperature, and weight-dependent
swimmig cost
Temp_SMR = (alpha*Weight**RB) * exp(RQ*Temperature) * Weight !Temp-dependent SMR (kJ/day)
Swim_cost = (alpha*Weight**RB) * Weight * 0.1
WResp = Temp_SMR + Swim_cost                                    !Respiration cost per
day (kJ)

return
end subroutine BioEnergetics_Winter
```

1.2.4 PreyAbundanceNWS

```
    !Gives prey adundance scaler for day of the year
    subroutine PreyAbundanceNWS(PPm3_scaler)
    implicit none

    ! Parameters
    real*8, parameter:: sigma = 0.4
    real*8, parameter::  mu = 2
    real*8, parameter:: feed_day_min = 91 !1 April
    real*8, parameter:: prey_day_peak = 166 !15 June
    real*8, parameter:: feed_day_max = 244 !1 Sept

    !Variables
    integer i, j
    integer index_max_x(1)
    real*8 days_season, pi, max_scaled, pdf_max, pdfscaler, min_to_peak, xscaler
    !Arrays
    real*8 :: x(1:201)
    real*8 :: pdf(1:201)
    real*8 :: pdf_scaled(1:201)
    real*8 :: new_x(1:201)
```

```
    real*8 :: PPm3_scaler(91:244) !out

    pi = 4.0*atan(1.0)
    days_season = feed_day_max - feed_day_min
    x(:) = (/0:400:2/)
    x(:) = x(:)*0.01

    ! Create pdf curve
    do i = 1,size(x)
        pdf(i) = (1/sqrt(2*pi*sigma**2)) * exp(-((x(i)-mu)**2)/(2*sigma**2))
        write(10,'(E15.5,E15.5,18E15.5)') x(i), pdf(i)
    enddo

    ! Location of max value of x for scaling
    index_max_x(:) = maxloc(pdf)

    ! Scale x-axis according to peak day and season days
    min_to_peak = feed_day_max - feed_day_min
    xscaler = min_to_peak/(maxval(x)-minval(x))

    ! Scale y-axis to max value
    max_scaled = 1 - 0.1*1      !Max value of curve - start season with 10% of max food
abundance
    pdf_max = (1/sqrt(2*pi*sigma**2)) * exp(-((x(index_max_x(1))-mu)**2)/(2*sigma**2))
    pdfscaler = max_scaled/pdf_max    ! scaler for p, such that pscaler*pdf(mode) =
pdensity

    do j = 1,size(x)
        pdf_scaled(j) = pdfscaler * ((1/sqrt(2*pi*sigma**2)) * exp(-((x(j)-
mu)**2)/(2*sigma**2)))
        new_x(j) = x(j)*xscaler
    enddo

    !Output values for feeding season day 244 - 91 = 153 days long
    PPm3_scaler(91:244) = pdf_scaled(1:154)

    return
end subroutine PreyAbundanceNWS
```

4. General subroutines

1. Temperature

```
subroutine Temperature(WTemp,STemp,Temp)
implicit none

! Parameters
integer       t
real*8, parameter ::  peak_day = 212  !Day peak water temp - 31 July
real*8, dimension(2*365) :: timespan_days,timespan_radians,Temp
real*8 STemp,WTemp,day_STemp,loop_day,dag

! Variables
real*8 min_y,max_y,peak_radians,phase_shift,pi  !in subroutine

! Max and min annual temperature
min_y = WTemp
max_y = STemp

pi = 4.0*atan(1.0)

peak_radians = (peak_day/365)*2*pi
phase_shift = (pi/2-peak_radians)  !unit radians

do t = 1,2*365,1
timespan_days(t) = t
timespan_radians(t) = (timespan_days(t)/365)*2*pi
Temp(t) = sin(timespan_radians(t)+phase_shift)
end do

Temp = Temp-minval(Temp)  !Calculate values and make sure all values are positive
Temp = Temp/maxval(Temp)  !all values between 0 and 1
Temp = min_y + (max_y-min_y)*Temp  !rescale between min and max

return
end subroutine Temperature
```

2. qsw0_hr

```
  subroutine qsw0_hr(radmax, ssurf, altdeg, clouds, rlat, time, npart, daysinyear)

  !-------------------------------------------------------------------------------------
------
  !HYCOM qsw0
  ! -> modified by Anders F. Opdal to accept time of day (npart) as an input variable and
respond by returning an hourly value of radiation (ssurf)
  ! -> modified by Tom Langbehn to give solar altitude angles in degrees as output
(altdeg), and modified to use calculations previously from the SURLIGHT subroutine
  !                            to calculate incremental twilight values when the sun is
6° , 6-12°  or 12-18° below the horizon
  !-------------------------------------------------------------------------------------
------

  !---------------------------------------
  ! NOTE regarding radians (rlat):
  ! Equator is at π radians (180°), not zero
```

```
  ! South pole is 3π/2 radians (270°
  ! North pole is π/2 Radians (90°)
  !----------------------------------------

  implicit none
  ! radmax     (out) - daily maximum solar irrradiance at the marine surface layer ---
(unit: w/m^2)
  ! ssurf      (out) - Anders: adding ssurf as an output variable
  ! clouds     (in)  - cloud input (fraction of cloudy sky)
  ! rlat       (in)  - latitude (radians). SEE NOTE ABOVE
  ! time        (in)  - julian day
  ! daysinyear (in)  - number of days in the year
  ! npart      (in)  - Anders: adding npart as an input variable (hour of day)

  real*8,  intent(out)     ::  radmax,ssurf,altdeg
  real*8,  intent(in)      ::  clouds,rlat
  real*8,  intent(in)      ::  time
  integer, intent(in)      ::  daysinyear, npart

  ! ---- set parameters ----
  real*8, parameter :: pi2    = 8.*atan(1.)    ! 2 times pi
  real*8, parameter :: deg    = 360./pi2       ! convert from radians to degrees
  real*8, parameter :: rad    = pi2/360.       ! convert from degrees to radians
  real*8, parameter :: eepsil = 1.e-9          ! small number

  real*8, parameter :: fraci  = 1./24.         ! split each day into 1/24 hour fractions

  real*8, parameter :: absh2o = 0.09           ! absorption of water and ozone. HYCOM
value = 0.09
  real*8, parameter :: s0     = 1365.          ! w/m^2  solar constant 1365 originally

  real*8, parameter :: twlight = 5.76          ! light at 0-degree sun (from the former
SURLIG subroutine, needs checking)

  ! ---- set variables -----
  real*8 day,dangle,decli,sundv
  real*8 cc,sin2,cos2,cosz,scosz,stot,bioday,biohr,hangle
  real*8 srad,sdir,sdif,cfac,altdeg_noon

  ! --- ------------------------------------------------------------------------
  ! --- compute hourly solar irrradiance at the marine surface layer (unit: w/m^2)
  ! --- ------------------------------------------------------------------------

  day=dmod(time,float(daysinyear))             ! 0 < day < 364
  day=floor(day)
  dangle=pi2*day/float(daysinyear)             ! day-number-angle, in radians
  if (day<0. .or. day>daysinyear+1) then
    print *,'qsw0: Error in day for day angle'
    print *,'Day angle is ', day, daysinyear
    stop
  end if

  ! --- compute astronomic quantities --
  decli = .006918+.070257*sin(dangle) - .399912*cos(dangle) &
    +.000907*sin(2.*dangle) - .006758*cos(2.*dangle) &
    +.001480*sin(3.*dangle) - .002697*cos(3.*dangle)

  sundv = 1.00011+.001280*sin(dangle)    + .034221*cos(dangle) &
```

```
    +.000077*sin(2.*dangle) + .000719*cos(2.*dangle)

  ! --- fetch cloud cover value (IN)
  cc = clouds

  ! --- compute astronomic quantities
  sin2=sin(rlat)*sin(decli)
  cos2=cos(rlat)*cos(decli)

  ! --- compute the solar radiance for each hour

  scosz  = 0.
  stot   = 0.
  radmax = 0.0

  bioday = day + (npart - .5)*fraci                                ! day + hour in decimal
number
  biohr  = bioday*86400.                                           ! hour of day in seconds
  biohr  = dmod(biohr, 86400.)                                     ! Anders: Hour of day:
biohr=12 at noon. OLD CODE: dmod(biohr+43200.,86400.) biohr=0  at noon
  hangle = pi2*biohr/86400.                                        ! hour angle, in radians

  cosz   = sin2+cos2*cos(hangle)                                   ! cosine of the zenith
angle - with negative values, meaning sun below the horizon
  scosz  = scosz + cosz                                            ! ..accumulated
  altdeg = dsin(scosz)*deg                                         ! solar altitude in
degrees (OUT)

  if (altdeg .ge. 0.) then                                         ! if cosine of the zenith
angle > 0 e.g above the horizon

    srad   = s0*sundv*cosz                                         ! extraterrestrial
radiation
    sdir   = srad*0.7**(min(100., 1./(cosz+eepsil)))               ! direct radiation
component
    sdif   = ((1. - absh2o)*srad - sdir)*.5                        ! diffusive radiation
component
    altdeg_noon = dmax1(0.,dsin(min(1.0,sin2+cos2)))*deg           ! solar noon altitude in
degrees
    cfac   = (1.-0.62*cc+0.0019*altdeg_noon)                       ! cloudiness correction

    ssurf  = (sdir + sdif)*cfac                                    ! surface light (W/m2) for
sun above the horizon (OUT)

  else if (altdeg .ge. -6.) then
    ssurf = ((twlight - .048)/6.)*(6.  + altdeg) + 0.048           ! surface light (W/m2) for
civil twilight,        sun  0 -6° below the horizon  (OUT) | adopted from SURLIG
subroutine

  else if (altdeg .ge. -12.) then
    ssurf = ((.048 - 1.15e-4)/6.)*(12. + altdeg) + 1.15e-4         ! surface light (W/m2) for
nautical twilight,     sun >6 -12° below the horizon (OUT) | adopted from SURLIG
subroutine

  else if (altdeg .ge. -18) then
    ssurf = (((1.15e-4) - 1.15e-5)/6.)*(18. + altdeg) + 1.15e-5 ! surface light (W/m2) for
astronomical twilight, sun >12-18° below the horizon (OUT) | adopted from SURLIG
subroutine
```

```
  else
    ssurf = 1.15e-5                                                ! surface light (W/m2)
accounting for e.g. stars, sun    >18° below the horizon (OUT) | adopted from SURLIG
subroutine

  endif

  radmax = max(radmax, ssurf)                                      ! maximum light, W/m2
(OUT)

  end subroutine qsw0_hr

SUBROUTINE GETR(r,c,C0,Ap,Vc,Ke,Eb,IER)
    !     ---------------------------------------------------------------

    !     Obtain visual range by solving the non-linear equation
    !     by means of Newton-Raphson iteration and derivation in
    !     subroutine DERIV. Initial value is calculated in EASYR.
    !    The calculation is based on the model described in Aksnes &
    !    Utne (1997) Sarsia 83:137-147.
    !
    !    Programmed and tested 29.01.01 Dag L Aksnes
    !
    implicit none
    REAL*8     r,c,C0,Vc,Ap,Ke,Eb
    REAL*8    EPS,RST,TOL,TOLF,F1,FDER,DX
    INTEGER    I,IEND,AS,IER

    !     Input parameters
    !         RST           : start value of r calculated by EASYR
    !         c             : beam attenuation koefficient (m-1)
    !         C0            : prey inherent contrast
    !         Ap            : prey area (m^2)
    !         Vc            : parameter characterising visual capacity (d.l.)
    !                         this parameter is denoted E' in Aksnes & Utne
    !         Ke         : saturation parameter (uE m-2 s-1)
    !      Eb     : background irradiance at depth DEPTH

    !    Output parameters
    !        F1     : function value of equation in DERIV
    !        FDER   : the derivative of the function
    !        r      : the predator's visual range (when F1=0)
    !     IER   : = 1, No convergence after IEND steps. Error return.
    !                       = 2, Return in case of zero divisor.
    !                       = 3, r out of allowed range (negative)
    !                       = 0, valid r returned




    !     Initial guess of visual range (RST)
    CALL EASYR(RST,C0,Ap,Vc,Ke,Eb)


    !     Upper boundary of allowed error of visual range
    EPS = .0000001
    !     Maximum number of iteration steps
```

```
      IEND = 100

      !     Prepare iteration
      r = RST
      TOL = r
      CALL DERIV(r,F1,FDER,c,C0,Ap,Vc,Ke,Eb)
      TOLF = 100. * EPS

      !     Start iteration loop
      DO 6 I = 1, IEND
          IF (F1) 1, 7, 1

          !        Equation is not satisfied by r
1         IF (FDER) 2, 8, 2

          !        Iteration is possible
2         DX = F1/FDER
          r = r-DX

          !        Test on allowed range
          IF (r .LT. 0.) GOTO 9

          TOL = r
          CALL DERIV(r,F1,FDER,c,C0,Ap,Vc,Ke,Eb)

          !        Test on satisfactory accuracy
          TOL = EPS
          AS = ABS(r)
          IF (AS-1.) 4, 4, 3
3         TOL = TOL*AS
4         IF (ABS(DX)-TOL) 5, 5, 6
5         IF (ABS(F1)-TOLF) 7, 7, 6
6     CONTINUE

      !     No convergence after IEND steps. Error return.
      IER = 1
7     RETURN
      !     Return in case of zero divisor
8     IER = 2
      RETURN
      !     r out of allowed range (negative)
9     IER = 3
      RETURN

      END

      !     ---------------------------------------------------------------
      SUBROUTINE EASYR(r,C0,Ap,Vc,Ke,Eb)
      !     ---------------------------------------------------------------
      !     Obtain a first estimate of visual range by using a simplified
      !     expression of visual range
      implicit none
      REAL*8       r,C0,Ap,Vc,Ke,Eb
      REAL*8       R2
      !
      !     Se calling routine for explanation of parameters
      !
      R2= ABS(C0)*Ap*Vc*(Eb/(Ke+Eb))
```

```
    r = SQRT(R2)
    RETURN
    END


    !     ------------------------------------------------------
    SUBROUTINE DERIV(r,F1,FDER,c,C0,Ap,Vc,Ke,Eb)
    !     ------------------------------------------------------
    !
    !     Derivation of equation for visual range of a predator
    implicit none
    REAL*8  r,c,C0,Ap,Vc,Ke,Eb
    REAL*8    FR1,FR2,F1,FDER

    !     Input parameters
    !        Se explanation in calling routine
    !
    !    Output parameters
    !        F1     : function value of equation in DERIV
    !        FDER   : the derivative of the function
    !        r      : the predator's visual range (when F1=0)
    !
    !     The function and the derivative is calculated on the basis of the
    !     log-transformed expression


    FR2=LOG(ABS(C0)*Ap*Vc)
    FR1=LOG(((Ke+Eb)/Eb)*r*r*EXP(c*r))
    F1 = FR1-FR2
    FDER = c + 2./r

    RETURN
    END
```

3. binary_IO

```
!*****************************************************************************
! $Id: binary_IO.f90 4710 2017-08-01 09:34:24Z cje012 $
!*****************************************************************************

!PURPOSE:
! - Write arrays to a binary files (array_to_binary)
! - Read binary files into arrays (binary_to_array)
! -

!Remember to include BINARY_IO before using the module :)

! NOTE: form='binary' is not portable and should **never be used** in any
!       new codes. The outdated 'binary' functionality is emulated in
!       F2003 standard combination of 'unformatted' and access='stream';
!       status='replace' makes sure the file is opened strictly for writing
!       and not for appending (default for stream is system dependent).

module binary_IO

  ! We get the standard error unit, which may be system-specific. It is
```

```
  ! necessary for error output. Error messages must go to the standard
  ! error device rather than standard output. This adheres to the standards
  ! and allows error redirection. In most cases, standard error goes to the
  ! terminal along with the standard output.
  use, intrinsic :: ISO_FORTRAN_ENV, only : ERROR_UNIT

  implicit none

  ! This is the portable declaration of the real type precision kind, in
  ! terms of the number of decimal digits and the maximum range of
  ! the values. This should guarantee replicable results on all systems and
  ! compilers.
  ! For example, the code below makes sure the real values have 15 digits
  ! of precision and can range from 1E-307 to 1E307. This is equivalent to
  ! real kind 8, 64 bit.
  ! and this is for 128 bit reals: selected_real_kind(33, 4931)
  integer, parameter, public :: RP  = 8

  !Global constants.
  ! used only within the module
  integer, parameter, private :: MAX_UNIT=255      ! Maximum unit number
  integer, parameter, private :: FAKE_ERR = -9999  ! Fake error array value.

  !File form for saving binary files
  character(len=*), parameter, private :: file_form = 'unformatted'

  !General declaration of folder path,
  character(100), parameter, private :: foldername =
'D:\Gabbi\P2_General_analysis\Matlab\'



  !Generic interface for printing  can use array_to_binary in the calling program.
  interface array_to_binary
    module procedure array_to_binary_real_1d
    module procedure array_to_binary_real_2d
    module procedure array_to_binary_real_3d
    module procedure array_to_binary_real_4d
    module procedure array_to_binary_real_5d
    module procedure array_to_binary_real_6d
    module procedure array_to_binary_real_7d
    module procedure array_to_binary_int_1d
    module procedure array_to_binary_int_2d
    module procedure array_to_binary_int_3d
    module procedure array_to_binary_int_4d
    module procedure array_to_binary_int_5d
  end interface array_to_binary

  !Generic interface for reading, can use binary_to_array in the calling program.
  interface binary_to_array
    module procedure binary_to_array_real_1d
    module procedure binary_to_array_real_2d
    module procedure binary_to_array_real_3d
    module procedure binary_to_array_real_4d
    module procedure binary_to_array_real_5d
    module procedure binary_to_array_int_1d
    module procedure binary_to_array_int_2d
```

```
  module procedure binary_to_array_int_3d
  module procedure binary_to_array_int_4d
  module procedure binary_to_array_int_5d
end interface binary_to_array

contains



!---------------------------------------------------------------------------
! TOOLS FOR WRITING TO FILES
!---------------------------------------------------------------------------

!-----------------------------------------------------------------------
! # GET_FREE_FUNIT #
! Purpose: returns the first free Fortran unit number (search from 1 to
! MAX_UNIT).
! ## RETURNS: ##
!  - Integer unit number
! ## CALL PARAMETERS ##
!  - optional logical execution error status (.TRUE.)
!  - optional integer max_funit to search (default MAX_UNIT defined in
!    module)
!
! Author: John Burkardt : This code is distributed under the GNU LGPL license.
! Modified by Sergey Budaev.
!
! This subroutine is a part of the *HEDTOOLS* suite. The stable SVN trunk
! address is here: https://svn.uib.no/aha-fortran/trunk/HEDTOOLS
!-----------------------------------------------------------------------
function GET_FREE_FUNIT (file_status, max_funit) result (file_unit)

  use, intrinsic :: ISO_FORTRAN_ENV     !Provides system-wide scalar constants
                                        !INPUT_UNIT OUTPUT_UNIT ERROR_UNIT
  implicit none

  !Function value
  integer :: file_unit

  !Calling parameters
  logical, optional, intent(out) :: file_status
  integer, optional, intent(in)  :: max_funit

  !Local variables: copies of optional parameters we need co copy optional
  logical  :: file_status_here    !Variables in case they are absent, so always
  integer  :: max_funit_here      !Work with copies of optionals inside
  !Other local variables
  integer :: i
  integer :: ios
  logical :: lopen

  !Subroutine name for DEBUG LOGGER
  character (len=*), parameter :: PROCNAME = "GET_FREE_FUNIT"

  file_unit = 0
  file_status_here = .FALSE.
```

```
  if (present(max_funit)) then
      max_funit_here=max_funit
    else
      max_funit_here=MAX_UNIT !Max from globals
  end if

  do i=1, max_funit_here
    if (i /= INPUT_UNIT .and. i /= OUTPUT_UNIT .and. &
          i /= ERROR_UNIT) then             !Exclude standard console units
      inquire (unit=i, opened=lopen, iostat=ios)
      if (ios == 0) then
        if (.not. lopen) then
          file_unit = i                     !First free unit found
          file_status_here=.TRUE.
          if (present(file_status)) file_status=file_status_here
          return
        end if
      end if
    end if
  end do

  if (.not. file_status_here) file_unit=-1    !If no free unit found return -1
  if (present(file_status)) file_status=file_status_here ! and error flag

end function GET_FREE_FUNIT



!-------------------------------------------------------------------------------
! DIAGNOSTIC FUNCTIONS FOR BINARY FILES
!-------------------------------------------------------------------------------

! # Diagnostic functions for binary files #
! ## Notes ##
! First, there are two diagnostic functions:
! - `binary_get_rank` determines what is the **rank** of the multidimensional
!    array saved in the binary file. If the file cannot be opened for any
!    reason, returns -1.
! - `binary_get_dims` returns the sizes of all the **dimensions** of the
!    multidimensional array saved into the binary file. If the file cannot
!    be opened for any reason, return an empty zero-size array.
! These functions should work with matrices of any dimensionality and any
! type (real and integer) because they don't read the actual data, only the
! header part of the binary file.
! These functions can be used as any other functions, e.g. in the
! `if`-statements: `if (binary_get_rank("data_file.bin")==4) call something`.

!Determine what is the rank of the array from the binary file.
function binary_get_rank(file_name, funit) result (array_rank_get)
  !Return value
  integer :: array_rank_get

  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  !Local variables
  integer :: funit_loc, error_status
```

```
  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the file.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !Check file open error. If okay, read header.
  if (error_status == 0) then
    !Get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
  else
    !If there was an error opening the file return -1 as error code.
    array_rank_get = -1
  end if
  !Finally, close the file.
  close(funit_loc)

end function binary_get_rank


!Determine what are the dimensions of the data array saved into the
! binary file. Dimensions of the array are returned as an allocatable
! array of the size equal to the rank of the array.
function binary_get_dims(file_name, funit) result (dim_length)
  !Return value is actually a 4d array
  integer, allocatable, dimension(:) :: dim_length
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  !Local variables
  integer :: i, funit_loc, error_status
  integer :: array_rank_get

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !Check file open error. If okay, read header.
  if (error_status == 0) then
    !First, get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
    !Knowing the rank, we can now allocate the output array to this size
    allocate(dim_length(array_rank_get))
```

```
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
  else
    !If there was an error opening the file return null-size output array.
    allocate(dim_length(0))
  end if
  !Finally, close the file.
  close(funit_loc)

end function binary_get_dims



!-------------------------------------------------------------------------------
! WRITE ARRAY TO BINARY FILE
!-------------------------------------------------------------------------------

!Write to binary for 1D array with real values
subroutine array_to_binary_real_1d(array, file_name, funit, error)
  integer, parameter :: array_rank = 1             !For this version it is 1
  !Calling parameters of the subroutine
  real(kind=RP), dimension(:), intent(in) :: array !1D real array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_real_1d
```

```
!Write to binary for 2D array with real values
subroutine array_to_binary_real_2d(array, file_name, funit, error)
  integer, parameter :: array_rank = 2               !For this version it is 2
  !Calling parameters of the subroutine
  real(kind=RP), dimension(:,:), intent(in) :: array !2D real array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_real_2d


!Write to binary for 3D array with real values
subroutine array_to_binary_real_3d(array, file_name, funit, error)
  integer, parameter :: array_rank = 3                 !For this version it is 3
  !Calling parameters of the subroutine
  real(kind=RP), dimension(:,:,:), intent(in) :: array !3D real array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
```

```
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_real_3d


!Write to binary for 4D array with real values
subroutine array_to_binary_real_4d(array, file_name, funit, error)
  integer, parameter :: array_rank = 4                    !For this version it is 4
  !Calling parameters of the subroutine
  real(kind=RP), dimension(:,:,:,:), intent(in) :: array !4D real array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
```

```
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_real_4d


!Write to binary for 5D array with real values
subroutine array_to_binary_real_5d(array, file_name, funit, error)
  integer, parameter :: array_rank = 5                    !For this version it is 5
  !Calling parameters of the subroutine
  real(kind=RP), dimension(:,:,:,:,:), intent(in) :: array !5D real array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_real_5d

!Write to binary for 6D array with real values
subroutine array_to_binary_real_6d(array, file_name, funit, error)
  integer, parameter :: array_rank = 6                    !For this version it is 5
  !Calling parameters of the subroutine
  real(kind=RP), dimension(:,:,:,:,:,:), intent(in) :: array !5D real array
```

```
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_real_6d

!Write to binary for 7D array with real values
subroutine array_to_binary_real_7d(array, file_name, funit, error)
  integer, parameter :: array_rank = 7                    !For this version it is 5
  !Calling parameters of the subroutine
  real(kind=RP), dimension(:,:,:,:,:,:,:), intent(in) :: array !5D real array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
```

```
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_real_7d


!Write to binary for 1D array with integer values
subroutine array_to_binary_int_1d(array, file_name, funit, error)
  integer, parameter :: array_rank = 1       !For this version it is 1
  !Calling parameters of the subroutine
  integer, dimension(:), intent(in) :: array !1D integer array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if
```

```
end subroutine array_to_binary_int_1d


!Write to binary for 2D array with integer values
subroutine array_to_binary_int_2d(array, file_name, funit, error)
  integer, parameter :: array_rank = 2         !For this version it is 2
  !Calling parameters of the subroutine
  integer, dimension(:,:), intent(in) :: array !2D integer array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_int_2d


!Write to binary for 3D array with integer values
subroutine array_to_binary_int_3d(array, file_name, funit, error)
  integer, parameter :: array_rank = 3           !For this version it is 3
  !Calling parameters of the subroutine
  integer, dimension(:,:,:), intent(in) :: array !3D integer array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message
```

```
  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_int_3d


!Write to binary for 4D array with integer values
subroutine array_to_binary_int_4d(array, file_name, funit, error)
  integer, parameter :: array_rank = 4             !For this version it is 4
  !Calling parameters of the subroutine
  integer, dimension(:,:,:,:), intent(in) :: array !4D integer array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
```

```
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_int_4d


!Write to binary for 5D array with integer values
subroutine array_to_binary_int_5d(array, file_name, funit, error)
  integer, parameter :: array_rank = 5               !For this version it is 5
  !Calling parameters of the subroutine
  integer, dimension(:,:,:,:,:), intent(in) :: array !5D array
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  character(255) :: error_message

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if
  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='replace', iostat=error_status,             &
        iomsg=error_message )
  !If there was no error opening the file, proceed to write it.
  if (error_status == 0) then
    dim_length(1:array_rank) =  shape(array)
    write(funit_loc) array_rank, (dim_length(i), i = 1, array_rank), array
  else
    write(ERROR_UNIT,*) "Error writing the file <<",file_name, ">>: ",      &
                        error_message
  end if
  close(funit_loc)
  !Report error optionally.
  if (present(error)) then
    error = error_status
  end if

end subroutine array_to_binary_int_5d

!If you want to print another array type, just add a new submodule similar
! to array_to_binary_real_5d or array_to_binary_int_5d :)
```

```
  !-------------------------------------------------------------------------------
  ! READ BINARY FILE TO ARRAY
  !-------------------------------------------------------------------------------

  ! ``fortran
  ! ! Here is a short test program that illustrates how to
  ! ! use the binary read procedures:
  ! program test
  !
  !     use binary_IO
  !
  !     integer, dimension(5,15,25,35,10) :: ZZ = -9999
  !     integer, allocatable, dimension(:,:,:,:,:) :: II
  !     integer :: ier
  !
  !     ! Write data to a binary file.
  !     call array_to_binary(ZZ, "zzz.bin")
  !
  !     ! Use the diagnostic functions to check what the files are like.
  !     print *, "RANK = ", binary_get_rank("zzz.bin")
  !     print *, "DIMS = ", binary_get_dims("zzz.bin")
  !
  !     ! Print the whole data array halved using the read function:
  !     print *, binary_read_i5d("zzz.bin") / 2
  !
  !     ! Use the read function in another way
  !     II= binary_read_i5d("zzz.bin")
  !     print *, II
  !
  !     ! Use the read subroutine:
  !     call binary_to_array(II, "zzz.bin", error=ier)
  !     print *, II
  !
  ! end program test
  ! ```


!Read binary file into a 1D array with real values.
  subroutine binary_to_array_real_1d(array, file_name, funit, error)
    integer, parameter :: array_rank = 1                !For this version it is 1
    !Return value is actually a 1D array
    real(kind=RP), dimension(:), intent(inout) :: array !1D real array
    !Calling parameters of the subroutine
    character(len=*), intent(in) :: file_name
    integer, optional, intent(in) :: funit
    integer, optional, intent(out) :: error
    !Local variables
    integer, dimension(array_rank) :: dim_length
    integer :: i, funit_loc, error_status
    integer :: array_rank_get

    !Check if optional unit name is provided, if yes, use it if not use
    ! the first free unit. This avoids possible units conflicts if many files
    ! are opened in parallel.
    if (present(funit)) then
```

```
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !If there was no error opening the file, proceed to read it.
  if (error_status == 0) then
    !First, get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
    !Third, read the actual values of the data array now.
    read(unit=funit_loc) array
  !If array read is not successful...
  else
    ! ... return a fake array with clearly unusual values, so the error
    ! is easy to notice in debugging.
    write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                         FAKE_ERR, " fake array returned."
    array = FAKE_ERR
  end if
  !Report error optionally
  if (present(error)) then
    error = error_status
  end if
  !Finally, close the file.
  close(funit_loc)

 end subroutine binary_to_array_real_1d


!Read binary file into a 2D array with real values.
 subroutine binary_to_array_real_2d(array, file_name, funit, error)
  integer, parameter :: array_rank = 2                  !For this version it is 2
  !Return value is actually a 2D array
  real(kind=RP), dimension(:,:), intent(inout) :: array !2D real array
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  integer :: array_rank_get

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
```

```
    open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
          access='stream', status='old', iostat=error_status )
    !If there was no error opening the file, proceed to read it.
    if (error_status == 0) then
      !First, get the rank of the array that is saved in the binary file.
      read(unit=funit_loc) array_rank_get
      !Second, get the shape for the dimensions rank of the array
      read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
      !Third, read the actual values of the data array now.
      read(unit=funit_loc) array
    !If array read is not successful...
    else
      ! ... return a fake array with clearly unusual values, so the error
      ! is easy to notice in debugging.
      write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                           FAKE_ERR, " fake array returned."
      array = FAKE_ERR
    end if
    !Report error optionally
    if (present(error)) then
      error = error_status
    end if
    !Finally, close the file.
    close(funit_loc)

  end subroutine binary_to_array_real_2d


!Read binary file into a 3D array with real values.
  subroutine binary_to_array_real_3d(array, file_name, funit, error)
    integer, parameter :: array_rank = 3                    !For this version it is 3
    !Return value is actually a 3D array
    real(kind=RP), dimension(:,:,:), intent(inout) :: array !3D real array
    !Calling parameters of the subroutine
    character(len=*), intent(in) :: file_name
    integer, optional, intent(in) :: funit
    integer, optional, intent(out) :: error
    !Local variables
    integer, dimension(array_rank) :: dim_length
    integer :: i, funit_loc, error_status
    integer :: array_rank_get

    !Check if optional unit name is provided, if yes, use it if not use
    ! the first free unit. This avoids possible units conflicts if many files
    ! are opened in parallel.
    if (present(funit)) then
      funit_loc=funit
    else
      funit_loc=GET_FREE_FUNIT()
    end if

    !Open the files and write. Note that local variable funit_loc keeps unit.
    open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
          access='stream', status='old', iostat=error_status )
    !If there was no error opening the file, proceed to read it.
    if (error_status == 0) then
      !First, get the rank of the array that is saved in the binary file.
      read(unit=funit_loc) array_rank_get
```

```
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
    !Third, read the actual values of the data array now.
    read(unit=funit_loc) array
  !If array read is not successful...
  else
    ! ... return a fake array with clearly unusual values, so the error
    ! is easy to notice in debugging.
    write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                         FAKE_ERR, " fake array returned."
    array = FAKE_ERR
  end if
  !Report error optionally
  if (present(error)) then
    error = error_status
  end if
  !Finally, close the file.
  close(funit_loc)

end subroutine binary_to_array_real_3d


!Read binary file into a 4D array with real values.
subroutine binary_to_array_real_4d(array, file_name, funit, error)
  integer, parameter :: array_rank = 4                    !For this version it is 4
  !Return value is actually a 4D array
  real(kind=RP), dimension(:,:,:,:), intent(inout) :: array !4D real array
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  integer :: array_rank_get

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !If there was no error opening the file, proceed to read it.
  if (error_status == 0) then
    !First, get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
    !Third, read the actual values of the data array now.
    read(unit=funit_loc) array
  !If array read is not successful...
  else
```

```
    ! ... return a fake array with clearly unusual values, so the error
    ! is easy to notice in debugging.
    write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                         FAKE_ERR, " fake array returned."
    array = FAKE_ERR
  end if
  !Report error optionally
  if (present(error)) then
    error = error_status
  end if
  !Finally, close the file.
  close(funit_loc)

end subroutine binary_to_array_real_4d


!Read binary file into a 5D array with real values.
subroutine binary_to_array_real_5d(array, file_name, funit, error)
  integer, parameter :: array_rank = 5                        !For this version it is 5
  !Return value is actually a 5D array
  real(kind=RP), dimension(:,:,:,:,:), intent(inout) :: array !5D real array
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  integer :: array_rank_get

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !If there was no error opening the file, proceed to read it.
  if (error_status == 0) then
    !First, get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
    !Third, read the actual values of the data array now.
    read(unit=funit_loc) array
  !If array read is not successful...
  else
    ! ... return a fake array with clearly unusual values, so the error
    ! is easy to notice in debugging.
    write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                         FAKE_ERR, " fake array returned."
    array = FAKE_ERR
  end if
```

```
  !Report error optionally
  if (present(error)) then
    error = error_status
  end if
  !Finally, close the file.
  close(funit_loc)

end subroutine binary_to_array_real_5d


!Read binary file into a 1D array with integer values.
subroutine binary_to_array_int_1d(array, file_name, funit, error)
  integer, parameter :: array_rank = 1          !For this version it is 1
  !Return value is actually a 1D array
  integer, dimension(:), intent(inout) :: array !1D integer array
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  integer :: array_rank_get

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !If there was no error opening the file, proceed to read it.
  if (error_status == 0) then
    !First, get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
    !Third, read the actual values of the data array now.
    read(unit=funit_loc) array
  !If array read is not successful...
  else
    ! ... return a fake array with clearly unusual values, so the error
    ! is easy to notice in debugging.
    write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                         FAKE_ERR, " fake array returned."
    array = FAKE_ERR
  end if
  !Report error optionally
  if (present(error)) then
    error = error_status
  end if
  !Finally, close the file.
  close(funit_loc)
```

```
end subroutine binary_to_array_int_1d


!Read binary file into a 2D array with integer values.
subroutine binary_to_array_int_2d(array, file_name, funit, error)
  integer, parameter :: array_rank = 2            !For this version it is 2
  !Return value is actually a 2D array
  integer, dimension(:,:), intent(inout) :: array !2D integer array
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  integer :: array_rank_get

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !If there was no error opening the file, proceed to read it.
  if (error_status == 0) then
    !First, get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
    !Third, read the actual values of the data array now.
    read(unit=funit_loc) array
  !If array read is not successful...
  else
    ! ... return a fake array with clearly unusual values, so the error
    ! is easy to notice in debugging.
    write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                         FAKE_ERR, " fake array returned."
    array = FAKE_ERR
  end if
  !Report error optionally
  if (present(error)) then
    error = error_status
  end if
  !Finally, close the file.
  close(funit_loc)

end subroutine binary_to_array_int_2d


!Read binary file into a 3D array with integer values.
subroutine binary_to_array_int_3d(array, file_name, funit, error)
```

```
  integer, parameter :: array_rank = 3              !For this version it is 3
  !Return value is actually a 3D array
  integer, dimension(:,:,:), intent(inout) :: array !3D integer array
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  integer :: array_rank_get

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !If there was no error opening the file, proceed to read it.
  if (error_status == 0) then
    !First, get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
    !Third, read the actual values of the data array now.
    read(unit=funit_loc) array
  !If array read is not successful...
  else
    ! ... return a fake array with clearly unusual values, so the error
    ! is easy to notice in debugging.
    write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                         FAKE_ERR, " fake array returned."
    array = FAKE_ERR
  end if
  !Report error optionally
  if (present(error)) then
    error = error_status
  end if
  !Finally, close the file.
  close(funit_loc)

end subroutine binary_to_array_int_3d


!Read binary file into a 4D array with integer values.
subroutine binary_to_array_int_4d(array, file_name, funit, error)
  integer, parameter :: array_rank = 4               !For this version it is 4
  !Return value is actually a 4D array
  integer, dimension(:,:,:,:), intent(inout) :: array !4D integer array
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
```

```
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  integer :: array_rank_get

  !Check if optional unit name is provided, if yes, use it if not use
  ! the first free unit. This avoids possible units conflicts if many files
  ! are opened in parallel.
  if (present(funit)) then
    funit_loc=funit
  else
    funit_loc=GET_FREE_FUNIT()
  end if

  !Open the files and write. Note that local variable funit_loc keeps unit.
  open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
        access='stream', status='old', iostat=error_status )
  !If there was no error opening the file, proceed to read it.
  if (error_status == 0) then
    !First, get the rank of the array that is saved in the binary file.
    read(unit=funit_loc) array_rank_get
    !Second, get the shape for the dimensions rank of the array
    read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
    !Third, read the actual values of the data array now.
    read(unit=funit_loc) array
  !If array read is not successful...
  else
    ! ... return a fake array with clearly unusual values, so the error
    ! is easy to notice in debugging.
    write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                         FAKE_ERR, " fake array returned."
    array = FAKE_ERR
  end if
  !Report error optionally
  if (present(error)) then
    error = error_status
  end if
  !Finally, close the file.
  close(funit_loc)

end subroutine binary_to_array_int_4d


!Read binary file into a 5D array with integer values.
subroutine binary_to_array_int_5d(array, file_name, funit, error)
  integer, parameter :: array_rank = 5                  !For this version it is 5
  !Return value is actually a 5D array
  integer, dimension(:,:,:,:,:), intent(inout) :: array !5D array
  !Calling parameters of the subroutine
  character(len=*), intent(in) :: file_name
  integer, optional, intent(in) :: funit
  integer, optional, intent(out) :: error
  !Local variables
  integer, dimension(array_rank) :: dim_length
  integer :: i, funit_loc, error_status
  integer :: array_rank_get
```

```
    !Check if optional unit name is provided, if yes, use it if not use
    ! the first free unit. This avoids possible units conflicts if many files
    ! are opened in parallel.
    if (present(funit)) then
      funit_loc=funit
    else
      funit_loc=GET_FREE_FUNIT()
    end if

    !Open the files and write. Note that local variable funit_loc keeps unit.
    open( unit=funit_loc, file=trim(foldername) // file_name, form=file_form, &
          access='stream', status='old', iostat=error_status )
    !If there was no error opening the file, proceed to read it.
    if (error_status == 0) then
      !First, get the rank of the array that is saved in the binary file.
      read(unit=funit_loc) array_rank_get
      !Second, get the shape for the dimensions rank of the array
      read(unit=funit_loc) (dim_length(i), i = 1, array_rank_get)
      !Third, read the actual values of the data array now.
      read(unit=funit_loc) array
    !If array read is not successful...
    else
      ! ... return a fake array with clearly unusual values, so the error
      ! is easy to notice in debugging.
      write(ERROR_UNIT,*), "Error reading the file <<",file_name, ">>, ",     &
                           FAKE_ERR, " fake array returned."
      array = FAKE_ERR
    end if
    !Report error optionally
    if (present(error)) then
      error = error_status
    end if
    !Finally, close the file.
    close(funit_loc)

  end subroutine binary_to_array_int_5d


end module binary_IO
```