

SUPPLEMENT 1

1. ASSUMPTIONS FOR PARAMETER k_D

The parameter k_D (scaling stomach evacuation rate) was used to calibrate the model to fit with adult surplus energy requirements inferred from data. In our modelling framework, surplus energy needs to cover costs associated with both growth and reproduction. Slotte, 1999, shows that for NSS herring a 28 cm and 38 cm fish spends about 15 g and 25 g fat, respectively, on their spawning migration. Using the energy value for fat commonly used in the literature of 39.75 kJ g^{-1} , this represents approximately 596.25 and 993.75 kJ. We also used the mean (for 1995 and 1996) from equations of length-specific ovary weights (1995: $W_o = 0.805e^{0.13L}$ for 1995; 1996: $W_o = 1.79e^{0.104L}$ for 1996; with the energy density of ovaries of 8 kJ g^{-1}) in Slotte, 1999, to estimate the energy of spawning products for a herring of 28 and 38 cm. To account for the energy required to build gonads, we assumed the total energy expenditure for gonad production to be twice the energy of gonads (i.e. 508.71 kJ for 28 cm, and 1645.30 kJ for 38 cm). From the data on NSS herring used in this study, growth for individuals above 33 cm (all individuals mature) is approximately 16 g yr^{-1} . We assume that a mixed diet gives 10 kJ g^{-1} and that the cost of growing equals the cost of the actual growth, i.e. 320 kJ. Altogether, this means that our estimate of the surplus energy for a herring of 28 cm should around 1425 kJ, and that for a 38 cm herring should be around 2959 kJ and we scaled k_D to match our predictions to these values. For herring of body lengths 28 and 38 cm our model predicts surplus energy values of 1708 and 3131 kJ, respectively.

2. DIETS OF NORWEGIAN-SPRING SPAWNING ANF NORTH SEA HERRING

We assumed the diet of NSS herring to consist of 60% *Calanus finmarchicus* and 40% euphausiids and amphipods, as this is the approximate wet weight ratio observed in stomach content data from summer samples (Dalpadado et al. 2000, Gislason & Astthorsson 2002, Dommasnes et al. 2004, Bachiller et al. 2016). Summer stomach content data for North Sea herring are scarce, but their diet appears to consist primarily of *C. finmarchicus* and *Calanus helgolandicus*, with the remainder being mainly post-larval stages of fish (Last, 1989). To simplify interpretation of the results and to account for some feeding outside of the main

foraging season, we assumed a wet weight ratio in the diet of North Sea herring of 60% *C. finmarchicus* and *C. helgolandicus*, and 40% larger prey (Last, 1989). For the default scenarios these ratios were obtained by scaling the predicted stomach content so that it corresponds to stomach data, and the same scaling factors were used in the sensitivity analyses. According to observations, we characterized *C. finmarchicus* in the North Sea simulation by a slightly smaller prosome length than in the Norwegian Sea simulation (Boxshall & Schminke, 1988; Skjoldal, 2004; Jónasdóttir *et al.*, 2005; Jonasdottir & Koski, 2011). Furthermore, due to the lack of discernible size differences between *C. finmarchicus* and *helgolandicus* in the North Sea (Wilson *et al.*, 2015), these species were modelled as one prey type. For generality, the ‘larger prey’ fraction in the diet of North Sea herring was parameterized as for euphausiids and amphipods in the Norwegian Sea simulation. NSS herring mainly forage in the upper 50 m during the main feeding season (Nøttestad *et al.* 2004). We defined one foraging depth for each prey type in the Norwegian Sea (30 m for the ‘smaller prey’ and 60 m for the ‘larger prey’), and set the foraging depth to 20 m in the North Sea to account for shallower and less clear waters. Seasonal prey abundance distributions were modelled and parameterized to fit with observations (see section 4.2; North Sea, Colebrook, 1979; Norwegian Sea, Varpe & Fiksen, 2010). Section 4 contains values and references used to parameterize prey characteristics.

3. INPUT DATA

3.1 Prey size, energy density and concentration

Table S1. Parameter values and references for prey characteristics

Prey type	Parameter	Value	Reference
<i>Norwegian Sea</i>			
<i>Calanus finmarchicus</i>	Length	3.0 mm	Castellani & Edwards, 2017
	Weight	0.1523 mg dw	Using length-weight formula adapted from Uye (1982) ¹ .
	Energy density	6400 cal/g dw	Laurence, 1976
	Max concentration	600 prey/m ³	Varpe & Fiksen, 2010
Amphipods and euphasiids ²	Length	14.4 mm	Lindley, 1978, 1982; Lindley et al. 1999; Kraft et al. 2012
	Weight	7.6896 mg dw	Using length-weight formula adapted from Uye (1982) ¹ .
	Energy density	5200 cal/g dw	Percy & Fife, 1981; Kulka & Corey, 1982
	Max concentration	3 prey/m ³	See 4.1.1
<i>North Sea</i>			
<i>Calanus finmarchicus</i> and <i>helgolandicus</i>	Length	2.6 mm	Jónasdóttir et al. 2005; Jonasdottir & Koski, 2011
	Weight	0.1065 mg dw	Using length-weight formula adapted from Uye (1982) ¹ .

	Energy density	6400 cal/g dw	Laurence, 1976; Kattner & Krause, 1989
	Max concentration	473 prey/m ³	See 4.1.1
Larger prey	Length	14.4 mm	Assume same value as for amphipods and euphausiids in the Norwegian Sea.
	Weight	7.6896 mg dw	Using length-weight formula adapted from Uye (1982) ¹ .
	Energy density	5200 cal/g dw	See text above.
	Max concentration	2 prey/m ³	See 4.1.1

¹Prey weight (mg dry weight) = $10^{2.5 \log(1000l_p) - 6.51} 10^{-3}$; Adapted from Uye (1982)

²Mean for *Themisto abyssorum* and *compressa*, *Meganycitiphanes norvegica*, and *Thysanoessa inermis* and *longicaudata*.

3.2 Seasonal prey abundance distributions

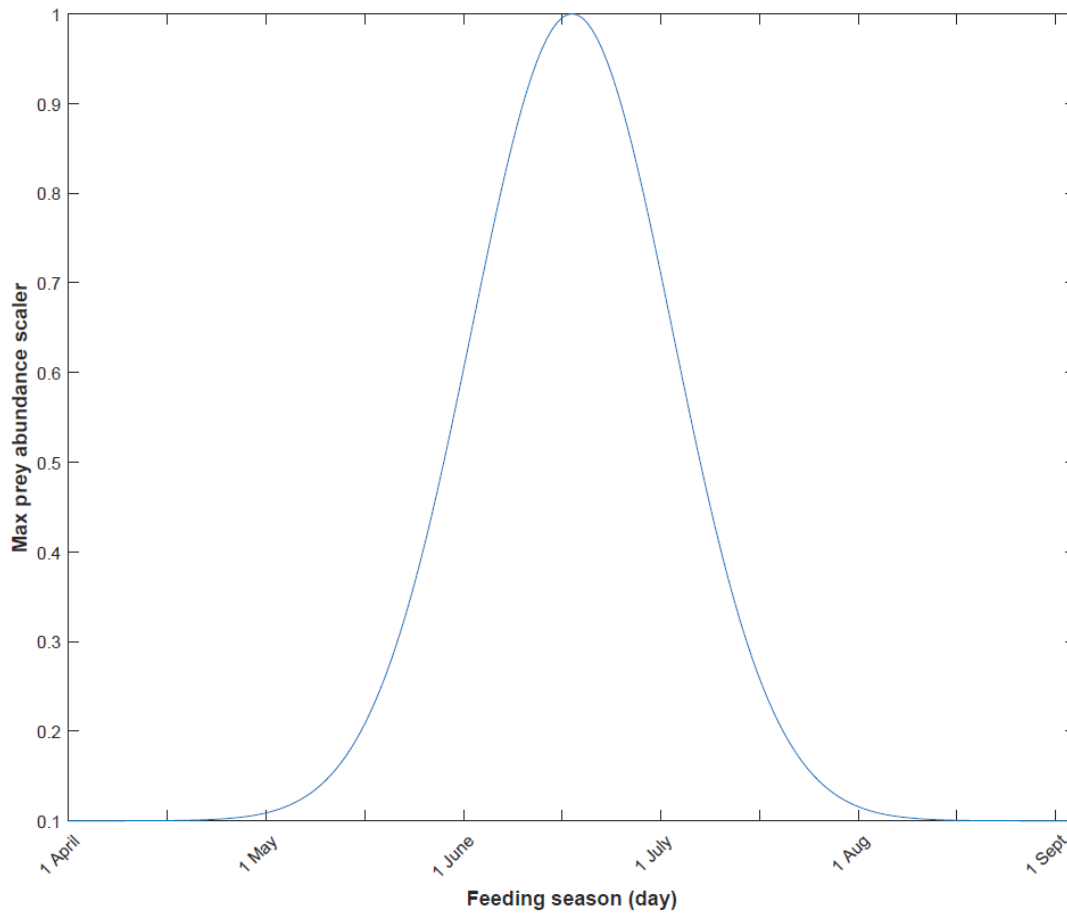


Figure S1. Seasonal prey biomass distribution s_d used to scale the maximum prey abundance $N_{\max,p}$ for the Norwegian Sea (see Equation 10 in Table 1)

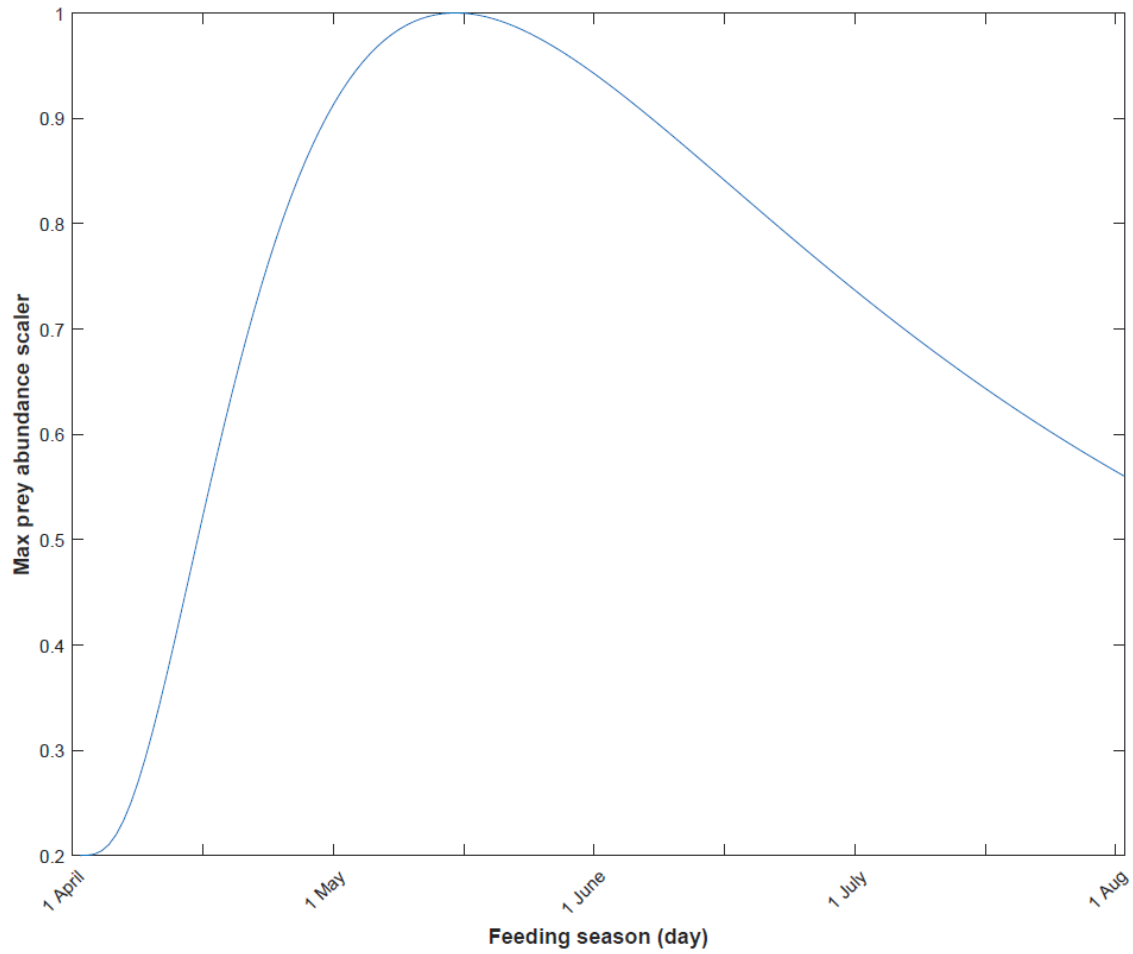


Figure S2. Seasonal prey biomass distribution s_d used to scale the maximum prey abundance $N_{\max,p}$ for the North Sea (see Equation 10 in Table 1)

4. ASSUMPTIONS AND CALCULATIONS

4.1 Prey biomass fractions

4.1.1 Predicting optimal body size in two herring populations

NORWEGAIN SEA

Approximation of biomass fractions by mesh size (dry weight; data obtained from Cecilie Broms, IMR):

<1 mm = 43 %

1-2 mm = 45 %

>2 mm = 12 %

Assumptions:

- *Calanus finmarchicus* constitute 45 % of the prey field
- Peak *C. finmarchicus* biomass concentration is 0.70 g/m³ (600 prey/m³; Varpe & Fiksen 2010) using prey dry weight from Table S1 and a dry weight to wet weight ratio of 0.13 (Rudstam 1988)
- Amphipods and euphausiids constitute 12 % of the prey field
- This yields a peak biomass concentration for amphipods and euphausiids of 0.18 g/m³ (3 prey/m³)

NORTH SEA

Approximation of biomass fractions by mesh size (dry weight; Falkenhaus, 2016):

<1 mm = 50 %

1-2 mm = 40 %

>2 mm = 10 %

Assumptions:

- Biomass ratio North Sea and Norwegain Sea = 5.7/9.2 (Broms 2016, Falkenhaus 2016), assume the same ratio between *C. finmarchicus* and *helgolandicus*
- *C. finmarchicus* and *helgolandicus* constitute 40 % of the prey field

- This yields a peak biomass concentration for *C. finmarchicus* and *helgolandicus* of 0.39 g/m³ (473 prey/m³)
- Larger prey constitute 10 % of the prey field
- This yields a peak biomass concentration for larger prey of 0.12 g/m³ (2 prey/m³)

4.1.2 Drivers of optimal body size in Norwegian spring-spawning and North Sea herring

NORWEGIAN SEA

- For this analysis default prey size (3 mm and 14.4 mm) is varied by ±20%
- Assuming a constant biomass concentration (see 4.1.1; Norwegian Sea) for all prey sizes yields peak prey concentrations for 3 mm ±20% prey of 750 prey/m³ and 500 prey/m³ for smaller and larger size, respectively
- Assuming constant biomass concentration (see 4.1.1; Norwegian Sea) for all prey sizes yields peak prey concentrations for 14.4 mm ±20% prey of 3.75 prey/m³ and 2.5 prey/m³ for smaller and larger size, respectively

NORTH SEA

- For this analysis default prey size (2.6 mm and 14.4 mm) is varied by ±20%
- Assuming constant biomass concentration (see 4.1.1; North Sea) for all prey sizes yields a peak concentration for 2.6 mm ±20% prey of 591 prey/m³ and 394 prey/m³ for smaller and larger size, respectively
- Assuming constant biomass concentration (see 4.1.1; North Sea) for all prey sizes yields a peak concentration for 14.4 mm ±20% prey of 2.5 prey/m³ and 1.67 prey/m³ for smaller and larger size, respectively

4.1.3 Drivers of optimal body size in planktivores

Assumptions:

- Peak *C. finmarchicus* (3 mm prey) biomass concentration is 0.70 g/m³ (600 prey/m³; Varpe & Fiksen 2010)
- Assuming constant biomass concentration for all prey sizes yields a peak prey concentration for 2 mm prey of 1652 prey/m³, and for 4 mm prey of 292 prey/m³ (using the length-weight formula adapted from Uye (1982) and a dry weight to wet weight ratio of 0.13 (Rudstam 1988))

5. SENSITIVITY TO ALTERED SEASONAL PREY DISTRIBUTIONS

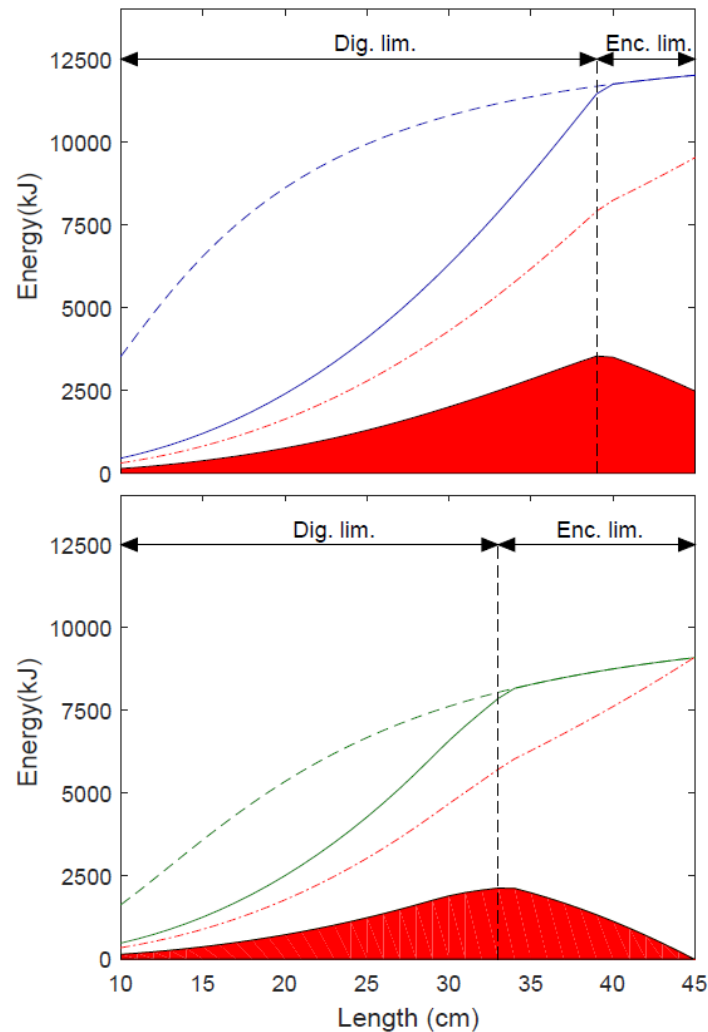


Figure S3. Predictions for the Norwegian Sea (top) using the seasonal prey abundance distribution for the North Sea (see Figure S2) and for the North Sea (bottom) using the seasonal prey abundance distribution for the Norwegian Sea (see Figure S3). Solid lines show digested food (kJ year^{-1}), dashed blue and green line show maximum potential food intake when there is no digestion limitation (sum of feeding rate; kJ year^{-1}), and red dotdash lines the metabolic cost (kJ year^{-1}). Red areas represent the difference between amount of digested food and the metabolic cost and thus represent annual surplus energy (kJ) of herring in the Norwegian Sea a) and the North Sea b). Dashed black lines show the predicted optimal size. ‘Dig. lim.’ and ‘Enc. lim.’ indicate the lengths at which digestion and prey encounters, respectively, limit energy acquisition.

LITERATURE CITED

- Bachiller E, Skaret G, Nøttestad L, Slotte A (2016) Feeding ecology of Northeast Atlantic Mackerel, Norwegian spring-spawning herring and blue whiting in the Norwegian Sea. *PLoS One* 11.
- Boxshall GA, Schminke HK (eds) (1988) *Biology of copepods: Proceedings of the third international conference on Copepoda*. Kluwer Academic Publishers, Dordrecht.
- Broms C (2016) Dyreplankton i Norskehavet. In: *Havforskningsrapporten*. p 98–99
- Castellani C, Edwards M (eds) (2017) *Marine plankton: A Practical guide to ecology, methodology, and taxonomy*. Oxford University Press, Oxford.
- Colebrook JM (1979) Continuous plankton records: seasonal cycles of phytoplankton and copepods in the northeastern North Atlantic and the North Sea. *Mar Biol* 51:23–32.
- Daan N, Rijnsdorp AD, Van Overbeeke GR (1985) Predation by North Sea Herring *Clupea harengus* on Eggs of Plaice *Pleuronectes platessa* and Cod *Gadus morhua*. *Trans Am Fish Soc* 114:499–506.
- Dalpadado P, Ellertsen B, Melle W, Dommasnes A. (2000) Food and feeding conditions of Norwegian spring-spawning herring (*Clupea harengus*) through its feeding migrations. *ICES J Mar Sci* 57:843–857.
- Dommasnes A, Melle W, Dalpadado P, Ellertsen B (2004) Herring as a major consumer in the Norwegian Sea. *ICES J Mar Sci* 61:739–751.
- Falkenhaus T (2016) Dyreplankton i Nordsjøen. In: *Havforskningsrapporten*. p 96–97
- Gislason A, Astthorsson OS (2002) The food of Norwegian spring-spawning herring in the western Norwegian Sea in relation to the annual cycle of zooplankton. *Sarsia* 87:236–247.
- Jonasdóttir SH, Koski M (2011) Biological processes in the North Sea: comparison of *Calanus helgolandicus* and *Calanus finmarchicus* vertical distribution and production. *J Plankton Res* 33:85–103.
- Jónasdóttir SH, Trung NH, Hansen F, Gärtner S (2005) Egg production and hatching success in the calanoid copepods *Calanus helgolandicus* and *Calanus finmarchicus* in the North Sea

- from March to September 2001. J Plankton Res 27:1239–1259.
- Kattner G, Krause M (1989) Seasonal variations of lipids (wax esters, fatty acids and alcohols) in calanoid copepods from the North Sea. Mar Chem 26:261–275.
- Kraft A, Bauerfeind E, Nöthig EM, Bathmann U V. (2012) Size structure and life cycle patterns of dominant pelagic amphipods collected as swimmers in sediment traps in the eastern Fram Strait. J Mar Syst 95:1–15.
- Kulka ADW, Corey S (1982) Length and weight relationships of Euphausiids and caloric values of *Meganycitiphanes norvegica* (M. Sars) in the Bay of Fundy. J Crustac Biol 2:239–247.
- Last JM (1989) The food of herring, *Clupea harengus*, in the North Sea, 1983–1986. J Fish Biol 34:489–501.
- Laurence GC (1976) Caloric values of some North Atlantic calanoid copepods. Fish Bull 74:218–220.
- Lindley J a. (1982) Population dynamics and production of euphausiids. Mar Biol 66:37–46.
- Lindley JA (1978) Population Dynamics and Production of Euphausiids. I. *Thysanoessa longicauda* in the North Atlantic Ocean. Mar Biol 46:121–130.
- Lindley JA, Robins DB, Williams R (1999) Dry weight carbon and nitrogen content of some euphausiids from the Atlantic Ocean and the Celtic Sea. J Plankton Res 21:2053–2066.
- Nøttestad L, Fernö A, Misund OA, Vabø R (2004) Understanding herring behaviour. In: *The Norwegian Sea ecosystem*. Skjoldal HR (ed) Tapir Academic Press, Trondheim, p 245
- Percy JA, Fife FJ (1981) The biochemical composition and energy content of Arctic marine macrozooplankton. Arctic 34:307–313.
- Pitois SG, Shaw M, Fox CJ, Frid CLJ (2009) A new fine-mesh zooplankton time series from the Dove sampling station (North Sea). J Plankton Res 31:337–343.
- Rudstam LG (1988) Exploring the dynamics of herring consumption in the Baltic: applications of an energetics model of fish growth. Kieler Meeresforschungen Sonderh 6:312–322.
- Segers F, Dickey-Collas M, Rijnsdorp AD (2007) Prey selection by North Sea herring (*Clupea harengus*), with special reference to fish eggs. ICES J Mar Sci 64:60–68.

Skjoldal HR (2004) The Norwegian Sea ecosystem. Sætre R, Færnø A, Misund OA, Røttingen I (eds) Tapir Academic Press, Trondheim.

Slotte A (1999) Differential utilization of energy during wintering and spawning migration in Norwegian spring-spawning herring. *J Fish Biol* 54:338–355.

Uye S-I (1982) Length-Weight Relationships of Important Zooplankton from the Inland Sea of Japan. *J Oceanogr Soc Japan* 38:149–158.

Varpe Ø, Fiksen Ø (2010) Seasonal plankton–fish interactions: light regime, prey phenology, and herring foraging. *Ecology* 91:311–318.

Wilson RJ, Speirs DC, Heath MR (2015) On the surprising lack of differences between two congeneric calanoid copepod species, *Calanus finmarchicus* and *C. helgolandicus*. *Prog Oceanogr* 134:413–431.

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 Rudstam 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

length/weight relationship for Atlantic herring (from ICES)
weight/size
Wwg(L) = 0.00603 * Length_cm(L)**3.0904 ! General
MaxGut_g(L) = 0.03*Wwg(L) ! Max gut in grams, 3% of

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]
!Prey energy content [J]
Gut_kJ_p2(L) = PreyEnergy(2) * Gut_gdw_p2(L) * J_Cal
kJ
MaxGut_kJ(L) = Gut_kJ_p1(L) + Gut_kJ_p2(L) ! Max gut in

!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 - surplus energy left after winter =
total surplus energy at end of feeding season - total reparation 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 velocity 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 wholehole 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 abundance 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 constitutes 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, 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
WWg(L) = 0.00603 *
MaxGut_g(L) = 0.03*WWg(L) !
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_kJ_p2(L) = PreyEnergyDensity(2) *
MaxGut_kJ(L) = Gut_kJ_p1(L) +

!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.

```

```

each day of the feeding season                                !Ingestion loop over hours for
                                                             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
Diff_att*Turb_scaler(Da)*Forage_depth(1))
                                                             Eb(1) = ssurf*exp(-
Diff_att*Turb_scaler(Da)*Forage_depth(2))
                                                             Eb(2) = ssurf*exp(-
                                                             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_preay_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
3600. * dh * 1E-3          !Consumed energy (kJ) - Prey_J_Enc(J/s)*3600s/h*1h*0.001kJ/J
--> kJ
                                                             Consumption = Prey_J_Enc *
                                                             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)
min(GastricEvacRate*dh, Gut(L)) !Digestion per timestep (kJ)
                                                             ! Have left in gut what is not
digested

```

```

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

i.e. consumption with digestion limitation       ! Daily sum of digestion -
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)

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

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) + Day_Prey_g_Enc(1)         Tot_Prey_g_Enc(1) =
Tot_Prey_g_Enc(2) + Day_Prey_g_Enc(2)         Tot_Prey_g_Enc(2) =
+ Day_Prey_weight                             Tot_Prey_weight = Tot_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 - surplus energy

```

```

left after winter = total surplus energy at end of feeding season - total reparation
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_preweight_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 Rudstam 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_pre_y_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 constitute 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)
          Length_cm(L)**3.0904
          Max gut in grams, 3% of weight/size

          WWg(L) = 0.00603 *
          MaxGut_g(L) = 0.03*WWg(L) !

          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_kJ_p2(L) = PreyEnergyDensity(2) *
          MaxGut_kJ(L) = Gut_kJ_p1(L) +

          Gut_gdw_p1(L) * J_Cal !Prey energy content [J]
          Gut_gdw_p2(L) * J_Cal !Prey energy content [J]
          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

```

```

n*dh*3600.
Prey_g_Enc(1)
Prey_g_Enc(2)
Tot_prey_weight_cons

!Day_Prey_items = Day_Prey_items +
Day_Prey_g_Enc(1) = Day_Prey_g_Enc(1) +
Day_Prey_g_Enc(2) = Day_Prey_g_Enc(2) +
Day_Prey_weight = Day_Prey_weight +
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) +
Tot_Prey_g_Enc(2) = Tot_Prey_g_Enc(2) +
Tot_Prey_weight = Tot_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) +

enddo !End day loop - daily foraging cycle

loop_output_summer1(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Ca1_gdw,Da,Hand1,Hand2,Ing) =
Tot_Digestion(L) !(L,PSize,Lat,Temp,PPm3,Ca1_gdw,Da)

loop_output_summer2(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Ca1_gdw,Da,Hand1,Hand2,Ing) =
Tot_Resp

loop_output_summer3(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Ca1_gdw,Da,Hand1,Hand2,Ing) =
Tot_Consum

loop_output_summer4(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Ca1_gdw,Da,Hand1,Hand2,Ing) =
Tot_NetIng

loop_output_summer5(L,PSize1,PSize2,Lat,MaxT,PPm3_1,PPm3_2,Ca1_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- surplus energy left after
```

```
winter = total surplus energy at end of feeding season - total reparation 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_preay_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/gW^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  $\pi$  radians (180°), not zero
```

```

! South pole is  $3\pi/2$  radians ( $270^\circ$ )
! North pole is  $\pi/2$  Radians ( $90^\circ$ )
!-----

implicit none
! radmax      (out) - daily maximum solar irradiance 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 :: twilight = 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 irradiance 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+eeepsil))) ! 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 = ((twilight - .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
end interface

```



```

    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 to 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))
    end if
end function binary_get_dims

```

```

!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
```

```
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

```