IBMlib

A generic tool for individual-based modelling of aquatic organisms

Global version of code: Revision: 495
SVN revision of this manual: Revision: 446
Last change of this manual Date: 2013-08-23 10:34:25 +0200 (Fri, 23 Aug 2013)

February 8, 2016

Contents

0.1 Unresolved issues

- How should we include Kritins algorithms ?

Part I

User guide

Chapter 1

Overall concepts

IBMlib is a tool box for individual-based modelling intended for research purposes by emphasizing flexibility and detailed simulation control ability. The core vision of IBMlib is to provide a light weight environment that easily allows to combine existing/new biological modules with existing/new physical data sets. IBMlib provides and supports simple core functionality and provide templates for new developments. IBMlib abstracts biological organisms as particles which may display arbitrary behavior and have complex properties describing life processes. The IBMlib implementation concept is shown in Figure 1.1 It shows a core part IBMlib_core that contains generic core functionality for particle tracking. IBMlib_core accesses the physical environment via the physical interface; this interface can be linked to any data source that is able to describe the physical environment. The biological properties is specified in particle state module that provides the particle state interface to IBMlib; this amounts to specifying e.g. how biological particles grow and behave, if relevant. The particle state module can also describe passive particles and other idealized types. Finally the task interface contrals what IBMlib is actually going to do, e.g. a forward simulation or a data dump or any customized run type. When an actual physical, biological and task module has been selected, we refer to it as an IBMlib configuration. The setup is able to run offline (i.e. reading physical and biological fields from a database) and online (i.e. running contemporarely and coupled with physical circulation and ecosystem models) using various coupler schemes.

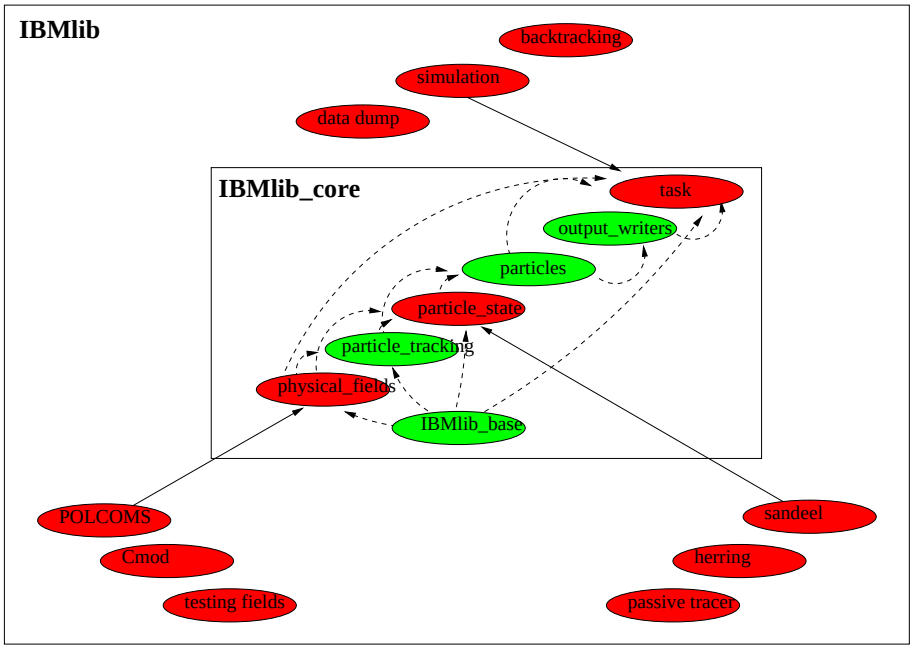

Figure 1.1: Concept diagram of the IBMlib framework.

Chapter 2

License issues

Currently unspecified, likely GPL in future

Chapter 3

Installation

3.1 System resources and requirements

IBMlib is currently set up to a Linux environment, however there should be no fundamental problems to port IBMlib to other OS environments. The minimal system resources required are:

- Fortran 90 compiler. All performance code is written in Fortran 90.
- Gmake. All objects are build from source code by makefiles. All makefiles are written for and tested with gmake; other make implementations may be used, but this may require minor adaptation to makefiles
- Python. Fortran modules are scanned for use associations with a preprocessor utility in Python. At some point we may ship dependency files along to avoid this requirement.

Depending on which sub modules for physical environment, biological properties and task are selected additional system resources may be required (e.g. NetCDF and HDF) - this should be available from the automatic documentation of the sub modules. Further the usage of postprocessing tools for visualization and data analysis may require further system resources, e.g. R and MatLab. The IBMlib test suite applies bash scripts.

3.2 Obtaining IBMlib sources

IBMlib are usualyy distributed in either two ways:

1. A tarball. Place the tarball (referring to this with filename `ibmlib.tgz`) in a desired directory (referring to this with directory name `IBMLIB_DIR`). The type on the command line:

   $$tar \;\; xvfz \;\; ibmlib.tgz$$

   to inflate the tarball. Now IBMlib is ready for configuration, see Sec. 3.3 below.

2. Via SVN source control. You need a login to obtain sources via SVN. The IBMlib repository is currently hosted by DTU. To get the latest version type on the command line in `IBMLIB_DIR`:

   $$svn \;\; co \;\; svn://svn.gbar.dtu.dk/asch/IBM\_Phoenix/trunk$$

   Now IBMlib is ready for configuration, see Sec. 3.3 below. To obtain a specific revision use the `-r` option to svn.

3.3 Building an IBMlib configuration

1. Configuring IBMlib. This basically amounts to placing/editing two files in the IBMlib base directory `IBMLIB_DIR` (where the Makefile is):

   - **config.mk**. This is the *logical* configuration of IBMlib, which is platform independent. In `config.mk` you select sub modules for physical environment, biological properties and task. These are read by the Makefile. This is done by stating the file directory, where these modules is. More specifically, you assign the the variables `PHYSICAL_FIELDS_DIR, PARTICLE_STATE_DIR, TASK_DIR` in which file directory the module is. The Makefile reads this from `config.mk` and takes care of the rest By default, the offline version builds into the executable `ibmrun`; if you want it to be called something else, set the make variable `EXECUTABLE` as desired. In

directory `setups/configurations/generic` you may find examples on files that can be copyed to `config.mk` and modified as desired.

- **compiler_defaults.mk**. Here you specify the F90 compiler you wish to use to compile IBMlib, as well as the compilation flag that should be applied. In directory `setups/compilers` you may find examples on files that can be copyed to `compiler_defaults.mk` and modified as desired. For each compiler, there are different compilation flag sets corresponding to what you may want to do, e.g. debugging (slower,safer) or production runs (faster, optimized, with no unnecessary runtime checks)

Additionally, you may augment **common_rules.mk** if you add customized components requiring special make rules to build. To ease the configuration step, there are configuration scripts in `setups/combo_scripts` that does the two steps for you. Just type

setups/combo_scripts/script_name

in `IBMLIB_DIR` to configure IBMlib corresponding to *script_name*. This is identical to the usual `./configure` in Linux source packages. In this way you can quickly setup your favorite configuration without typing arguments to `./configure`.

2. Building IBMlib. When IBMlib is configured as above, you simply type

make

on the command line in `IBMLIB_DIR`. Then the selected IBMlib configuration is build into the executable name given as `EXECUTABLE` in `config.mk`. This executable may be copied to somewhere in the standard executable search PATH, if it should be installed as an application.

3. Testing IBMlib. IBMlib comes along with a automatized test suite that basically test the integrity of the IBMlib version you are trying to compile. To invoke it, you simply type

make test

on the command line in in `IBMLIB_DIR`. If you are developing modules to IBMlib, you will find additional auxillary tests useful in the development phase, as well as auxillary analysis components. `IBMLIB_DIR/test_suite`.

3.4 Running an IBMlib configuration

When the executable is build, you simply type

$$ibmrun\ [\langle arguments \rangle]$$

on the command line. Often a file with simulation parameters is required as argument after ibmrun; the task sub modules should document which arguments and options they require and support.

3.5 Simulation files

When IBMlib is used in a standard task configuration, like a basic simulation, it expects you to provide an input file on the command line like

$$ibmrun\ \langle inputfile \rangle\ [\ >\ \langle outputfile \rangle]$$

All parts of the compiled IBMlib configuration will try to read most input data from this file in a standard task configuration. File reading is distributed, meaning that different parts of IBMlib reads this file independently of the other parts. The simulation input file is based on a tagged input ASCII format containing lines as

$$tag = value\ [value*] \tag{3.1}$$

tag is a character identifier terminated at first occurence of the separator =. value of the tag is what follows the separator to the end of that line and anything can appear here. If the value is a vector, each item in the vector is separated by spaces (but stay on same line). IBMlib does not check that all parameters are read. The same tag may appear many several times - e.g. a particle emission box, in other cases the tag should be unique, e.g. the particle time step. In this case the first

occurence is picked, the rest is ignored. The following rules also apply for the input file markup:

- Comments: everything after (and including) " !" is skipped
- Empty lines and spaces are just ignored
- Malformed lines are just ignored (e.g. if you forgot "=")
- Everything after "=" is considered part of the value(s) for tag. Values need not to be of the same elementary data type, you may e.g. mix letters and numbers. The documentation should specify what is expected for tag.
- Required tag (or value) is missing. This will generate a runtime error, IBMlib will stop, unless a default applies to the tag.
- The order of the tags does not matter.

The input format is convenient for scripting, where upper-level scripts generates input files. The input is read once and cached when the simulation is started. The following are mandatory input parameters:

3.5.1 Task = basic_simulation tags

- *start_time*. The beginning of the simulation, specified as (year month day second_of_day)
- *end_time*. The end time of the simulation, specified as (year month day second_of_day)
- *particle_time_step*. Nominel time step for the integration algortihm in seconds.
- *emitbox*. Gives a window in space and time, where sandeel larvae/eggs are released. The may be specified as many emitbox entries as desired, in this way they may act in parallel and quite complex release patterns can be set up. The first 4 integers are (year month day second_of_day) where the release begins (of that

box); the next 4 integers are (year month day second_of_day) where the release stops (of that box). The next six numbers specify a spatial box (in latitude and longitude) where sandeel larvae/eggs are released. Dry (land-locked) sectors of the spatial box are omitted when releasing larvae/eggs. The first three numbers are the spatial lower SW corner of the release box given as (longitude,latitude,vertical position), the next three numbers are the spatial upper NE corner of the release box given as (longitude,latitude,vertical position). The vertical position can be spcified as either:

- absolute depth: specify the depth below the water surface as a negative number.
- relative depth$z$:$0 < z < 1$. $z = 0$ corresponds to the sea surface, $z = 1$ corresponds to the sea bed.

Biological particles are released uniformly in time and space with in space-time window specified, so that the total number of particles released adds up to the integer given as number 15. All other parameters following number 15 in emit box are passed to the biological module.

3.5.2 Particle_tracking module

- *advec_intg_method*. The integration algortihm for time forward integration of advected particles (options are euler (Euler forward) or rk2/rk4 (Runge-Kutta 2 or 4)).

3.5.3 Example on minimal input file

```
!-----------------------------------------
!       Main simulation control file
!-----------------------------------------

start_time  = 2005 03 01 0    !  year  month  day  second_of_day
end_time    = 2005 03 18 0    !  year  month  day  second_of_day
```

```
advec_intg_method  = euler    !  advection scheme: euler/rk2/rk4
particle_time_step = 1800     !  in seconds for time integration of motion

! --------------- biology spatial control ---------------
! r(1:4) start:  year month day sec_of_day
! r(5:8) end:    year month day sec_of_day
! r(9:11)        lon_min lat_min z_min  (z=0 -> surface)
! r(12:14)       lon_max lat_max z_max  (z=1 -> bottom)
! r(15)          max_number_of_tracers
! r(16:)         other input item to particle state

emitbox = 2005 03 02 0     2005 03 02 3600  2 54 1   3 55 1  100 e
emitbox = 2005 03 02 3600  2 03 02 7200  4 54 1   5 56 1  100 l 9.66
! -----------------------------------------------------
```

There may be other mandatory entries, depending on which oceanography provider and biology provider you are applying in your configuration - consult these to find out which input fields are mandatory.

Part II

Programmers guide

Chapter 4

Makefiles and build protocols

IBMlib has seven dependency levels

1. TASK
2. output providers
3. particles.mod
4. PARTICLE_STATE
5. particle_tracking.mod
6. PHYSICAL_FIELDS
7. IBMLIB_BASE (incl. included external tools)

which gives the allowed use associations (higher to lower or same level) and the appropriate build order (from below and up). One level is only allowed to build objects at its own level (distributed makefiles) To build an IBMlib configuration, four make files are required:

1. Makefile:
2. $(PHYSICAL_FIELDS_DIR)/Makefile
3. $(PARTICLE_STATE_DIR)/Makefile
4. $(TASK_DIR)/Makefile

The first is generic and takes care of the overall build syncronization. The remaining make files are independent and invoked from the first makefile. They are not included, because this breaks encapsulation and creates name clashes. The specification of the last three makefiles are given below. Each makefile of may PHYSICAL_FIELDS/PARTICLE_STATE/TASK - or may not - include `common.mk`

4.1 $(PHYSICAL_FIELDS_DIR)/Makefiles

Module PHYSICAL_FIELDS, rooted in PHYSICAL_FIELDS_DIR. In this directory, there should be a makefile updating the targets:

- physical_fields.mod (F90 module interface, in directory PHYSICAL_FIELDS_DIR)
- physical_fields.a (all compiled objects of module, in directory PHYSICAL_FIELDS_DIR)
- clean

An (optional) makefile link_opt.mk in PHYSICAL_FIELDS_DIR may define the following variables for link options to be used for the final stage linking:

- LINKFLAGS_PHYSICAL
- LINKLIBS_PHYSICAL

4.2 $(PARTICLE_STATE_DIR)/Makefiles

Module PARTICLE_STATE rooted in PARTICLE_STATE_DIR. In this directory, there should be a makefile updating the targets:

- particle_state.mod (F90 module interface, in directory PARTICLE_STATE_DIR)
- particle_state.a (all compiled objects of module, in directory PARTICLE_STATE_DIR)

- clean

An (optional) makefile link_opt.mk in PARTICLE_STATE_DIR may define the following variables for link options to be used for the final stage linking:

- LINKFLAGS_STATE
- LINKLIBS_STATE

4.3 $(TASK_DIR)/Makefiles

Module TASK rooted in TASK_DIR In this directory, there should be a makefile updating the targets:

- task.a (all compiled objects INCLUDING the main program, in directory PARTICLE_STATE_DIR)
- clean

An (optional) makefile link_opt.mk in PHYSICAL_FIELDS_DIR may define the following variables for link options to be used for the final stage linking:

- LINKFLAGS_TASK
- LINKLIBS_TASK

4.4 Miscellaneous building notes

About order of objects at linking: "The traditional behavior of linkers is to search for external functions from left to right in the libraries specified on the command line. This means that a library containing the definition of a function should appear after any source files or object files which use it ...When several libraries are being used, the same convention should be followed for the libraries themselves."

ifort note: apparently this problem with ifort can just be handled by duplicating link objects once ...

Chapter 5

Data stractures, variables and units

5.1 Space

Space in IBMlib is continuous and *grid free* and data is accessed by query functions; thereby grid details (grid type, resoultion, nested grids, vertical layer structure, layout etc) are hidden behind the physical interface and the particle ware code becomes independent of particular data sets.

5.1.1 Space units

At the particle side horizontal coordinates (x, y) are longitude (degrees East) and latitude (degrees North) and vertical coordinate z is depth below actual sea surface in meters (i.e. positive down, zero at actual sea surface).

5.1.2 Vector orientation and units

At the particle side vectors are oriented along tangent space vectors in appropriate units. This means that vectors pointing East and North are positive and the positive vertical direction is toward the bottom. Note that this means that unit vectors for (longitude,latitude,vertical) (in this order) forms a left hand screw; we think it is more convenient

to have water depth positive with zero at surface than forming a right hand screw. Behind the interface(s), other internal conventions may be used as appropriate

Chapter 6

Interfaces

The subroutines and data structures constituting the major interfaces are specified below. An argument like `r5` means a real array of (minimal) length 5 etc; `r` means a scalar real.

6.1 The task interface

6.2 The physical-biological interface

To make particle ware code becomes independent of particular data sets, physical (and biogeochemical data) are acccessed by interpolation functions, thereby hiding the actual structure of the physical-biological data sets. Further a set of auxillary query and transformation functions are available to service particle dynamics, as well as module operators. [] indicates an optional field that need not to be provided by the interface. A physical-biological interface must be associated with a separate directory under directory `oceanography_providers`

6.2.1 Public module operators

- `init_physical_fields()`
- `close_physical_fields()`
- `update_physical_fields(time)`

6.2.2 Public data interpolation

Generally, data interpolation functions are named `interpolate_P(xy|xyz, ..., P, ..., status)` to interpolate property P at space position xyz. xyz is a vector with (longitude,latitude,depth) as specified in 5.1.1. xy is a lateral vector with (longitude,latitude). Generally, a length 3 vector may be supplied for xy, but not a length 2 vector for xyz. status is a return flag for the interpolation. status == 0 is the result of a normal interpolation. Non zero values of status means an exception has occured at interpolating P at xyz, e.g. a domain violation. A specified value will be assigned to P on exit, i.e. interpolation should always exit gracefully.

The interpolations (`interpolate_X`) offered by module physical_fields are really divided into layers of core functions and extension sets. Some interpolation services are also required by more specialized applications, e.g. zooplankton concentration for energy budget growth models of early life stages. These interpolation services are not required by core particle tracking modules and therefore these interpolation services need only be provided in case the more specialized applications utilized in the current setup. Trying to link a task requiring extension sets to a implementation of module physical_fields offering only core physical fields functions will generate a compile time error. If this is not desired, dummy functions can be provided for extensions (`interpolate_X`). The core functions are the basic Physical interpolations are

Core physical fields

- `interpolate_turbulence(xyz, r3, status)` m^2/s

  hdiffus_x, hdiffus_y → r3(1:2)
  vdiffus → r3(3)

- `interpolate_turbulence_deriv(xyz, r3, status)` m/s

  Cartesian derivative, on axes along normal vector orientation
  (d/dX) hdiffus_x, (d/dY) hdiffus_y) → r3(1:2)

(d/dZ) vdiffus → r3(3)

- `interpolate_currents(xyz, r3, status)` m/s

  current (u, v, w) in the neutral frame (horizontally stationary, zero at sea surface) in m/s oriented as standard vectors (5.1.2)

- `interpolate_temp (xyz, r, status)` Degree Celcius

- `interpolate_dslm(xyz, r, status)` m

  sea surface elevation above a reference level. Rigid lid data sets should return 0 for this field.

6.2.3 Public query subroutines/functions

- `interpolate_wdepth(xy, r, status)` m
  Below actual sea surface, in meters.

- `LOGICAL is_wet(xyz)` (function) Return TRUE if continuous position xyz is a (currently) wet point, else FALSE. Sea surface/bottom points must be accepted as wet. Normally this means that a parameter htol is given a very small value and that the accepted horizontal range is -htol ¡ z ¡ wdepth+htol where wdepth comes from interpolate_wdepth(xy,wdepth,status).

- `LOGICAL is_land(xy)` (function) Return TRUE if continuous horizontal position xy is a (currently) dry column, else FALSE.

- `LOGICAL horizontal_range_check(xy)` (function) Return TRUE if continuous horizontal position xy is inside horizontal domain covered by the physical data set - else FALSE. Notice that this is not a wet point check.

- `coast_line_intersection(xyz1, xyz2, anycross, xyzref, xyzhit)`

This function is a service function assisting in enforcing coastal boundary conditions on particle steps. xyz1 is the current valid (wet, inside domain) position. xyz2 is the next position that may or may not be in water. The assumption is that the particle moves in a straight line between xyz1 and xyz2 in (lon,lat,z) coordinates. Output: If the straight line xyz1 $\rightarrow$ xyz2 has crossed a coast line, anycross is returned true else false (i.e. the line between xyz1 and xyz2 is all in water). If anycross is true, the following vectors are computed: xyzref is the modified final position, if step from xyz1 to xyz2 is reflected in the coast line. xyzhit is the *first* position where the line from xyz1 to xyz2 crosses a coast line. If anycross is false xyzref and xyzhit is not computed but assigned an interface dependent dummy value. The particle tracking algorithm uses the elemental subroutine `coast_line_intersection` to generate multiple-reflection paths, if xyzref is a position on land.

Optional extensions of physical field

These extensions are not available as out from all circulation models, and therefore these interpolation services are optional. Currently, there are no defined subsets in extensions of physical properties

- `interpolate_windstress(xy, r2, status)]` N/m^2

  surface wind stress vector with oriented as standard horizontal vectors (longitunal, latitunal) (5.1.2)

- `interpolate_salty(xyz, r, status)` PSU

  Standard salinity

Optional Biogeochemistry

Biogeochemistry are increasingly becoming available from coupled circulation-biogeochemistry models. Below is the current name space for the biogeochemistry subinterface of Currently, there are no defined subsets in the biogeochemistry subinterface.

- `interpolate_zooplankton(xyz, rvec, status)]` kg dry weight/m^3

  Zooplankton concentration. Currently bulk zooplankton concentration is returned in element `rvec(1)`. The protocol for this interpolation service is under development.

- `interpolate_oxygen(xyz, r, istat)` mmol/m^3
  O_2 concentration

- `interpolate_nh4(xyz, r, istat)` mmol/m^3
  NH_4 concentration

- `interpolate_no3(xyz, r, istat)` mmol/m^3
  NO_3^- concentration

- `interpolate_po4(xyz, r, istat)` mmol/m^3
  PO_4^{3-} concentration

- `interpolate_diatoms(xyz, r, istat)` mmol/m^3
  Diatoms concentration

- `interpolate_flagellates(xyz, r, istat)` mmol/m^3
  Flagellate concentration

- `interpolate_cyanobacteria(xyz, r, istat)` mmol/m^3
  Cyanobacteria concentration

- `interpolate_organic_detritus(xyz, extract_data, istat)` mmol/m^3
  Organic detritus concentration

- `interpolate_part_org_matter(xyz, extract_data, istat)` mmol/m^3
  Particular organic matter concentration

- `interpolate_DIC(xyz, extract_data, istat)` mmol/m^3
  Dissolved inorganic carbon concentration

- `interpolate_alkalinity(xyz, extract_data, istat)` mmol/m^3
  Alkalinity

- `interpolate_DIN(xyz, extract_data, istat)` mmol/m^3
  Dissolved inorganic nitrogen concentration

- `interpolate_chlorophyl(xyz, r, istat)` mmol/m^3
  Chlorophyl concentration

.

6.2.4 Access exceptions

There are cases of invalid data access attempts; interpolation should exit gracefully, but with status integer set according to encountered exception.

- Dry point access attempts; some fields makes no sense at dry points, like currents turbulence (however e.g. temperature, wind does). Return DRY_POINT_VALUE (module constant) in these cases.
- Outside domain access attempts. Return OUTSIDE_DOMAIN_VALUE (module constant) in these cases.

6.3 The particle state interface

A particle state interface must be associated with a separate directory under directory `biology_providers`. This is the minimal public interface of a module providing the particle_state interface:

- `type state_attributes`

  This is an F90 derived type of arbitrary content describing/logging all aspects of the particle beyond generic spatio-temporal properties, e.g. all biological properties goes here. For syntactical reasons Fortran 90 does not allow a user-defined type to be empty.

- `init_particle_state()`

  Module start-up method

- `close_particle_state()`

  Civilized module close down. Remember to deallocate pointers and allocatable arrays.

- `subroutine init_state_attributes(state, space, time_dir, initdata, emitboxID)`

  Initialize an instances of particle_state (associated with space attributes space) and possibly also set certain space attributes (like boundary conditions and mobility). time_dir is a signed real; time_dir ¿ 0 is forward simulation, time_dir ¡ 0 is reverse time simulation (initialization generally depends on time direction). If reverse time simulation is not implemented for this state type, a run time error should be thrown. initdata is all parameemers after mandatory spatio-temporal parameters specifying an emission box. initdata is a plain unparsed string. emitboxID is an integer (associated with a release box) that the state type may choose to store (intended for history analysis)

- `get_active_velocity(state, space, v_active)`

  Get the active velocity component of the particle (i.e. the velocity relative to the water masses). Passive particles should return zero here. v_active is oriented as a standard vector, i.e. see Sec. 5.1.2.

- `update_particle_state(state, space, time_step)`

  Update particle_state instance state (associated with space attributes space) corresponding to time_step (in seconds). Notes that time_step is signed and that time_step ¡ 0 corresponds to backtracking. This subroutine may also probe the particle motion state in the space argument - should we have a protocol here, e.g. a "delete-me" return variable?

- `delete_state_attributes(state)`

  If the derived type state_attributes contains any allocated pointers, they should be deallocated here. When particles are deleted from an ensemble, this method is invoked to avoid memory leakage from non-referenced memory space.

- `write_state_attributes(state)`

  Mainly for debugging

6.4 Generic bioenergetics

generic_bioenergetics is a frame work to

1. make multi stage modelling, based on independent sub stages
2. model growth energy budgets for early life stages

generic_bioenergetics delivers the particle_state interface for IBMlib. The main idea is to localize species specific properties in particular modules to ensure portability to new species. generic_bioenergetics focuses on early life stages where the natural internal units are μg for weight, mm for length and seconds for time. Currently, internal classes within generic_bioenergetics have public scope (for the time being)

6.4.1 Multi stage modelling

Multi stage modelling is handled by a light weight bundling module that combines independent stages described by classes in other modules. The multi stage module offers a class has independent stage class instances as components. The multi stage module manages which stage is currently active and handles overall attributes, like the survival chance of the organism, place and time of origin. The multi stage module interacts with independent stages via the particle_sub_state interface described below

Componets

- Decorators: implement interface particle_sub_state by loading specific biological models/data from other contexts and compiling it into a behavioral model (like optimal feeding larvae, optimal_forager.f). These module are generic, i.e. does not contain reference to a particular species.
  - module feeding_larval_stage
  - module yolksac_larval_stage
  - module released_egg_stage

Decorators may also by themselves provide the full particle_state interface in parallel with particle_sub_state interface. To use this, a module name conversion dummy should be used. Currently this is only available for the implementation optimal_forager.f implementating module feeding_larval_stage

---

- Specific biological models:

  These module contain data for a particular species, so that these modules should be provided for each species. These modules are the only ones containing parameters for a particular species, so that all species specific information is contained here.

  - module egg_properties
  - module yolksac_properties
  - module larval_properties

- Auxillary modules/components:

  Provides generic auxillary functionality for generic_bioenergetics

  - particle_state_base
  - prey_community
  - numerical_1d_integrals
  - feeding_larval_stage_2_particle_state.f (name change wrapper module)

particle_sub_state interface

The particle_sub_state interface is implemented by the decorator modules. If X designates the stage (e.g. X=feeding_larvae) then the particle_sub_state interface has the following components:

- type X. This class is embedded in the multi stage class representing a multi staged organism.

early_life_stages.f : multi stage manager

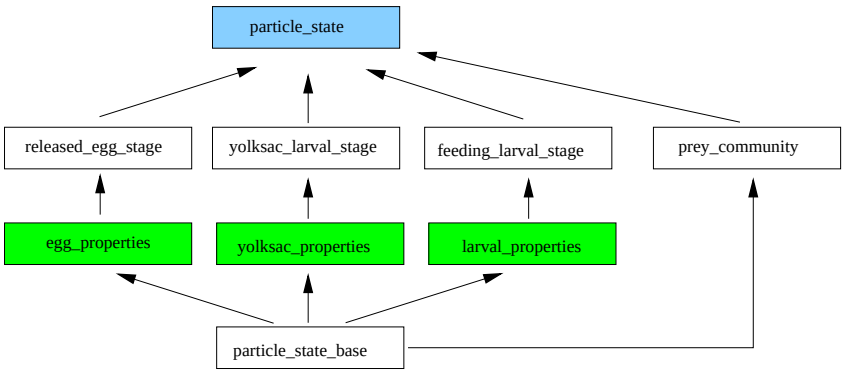

Figure 6.1: Inheritance diagram for multi stage manager

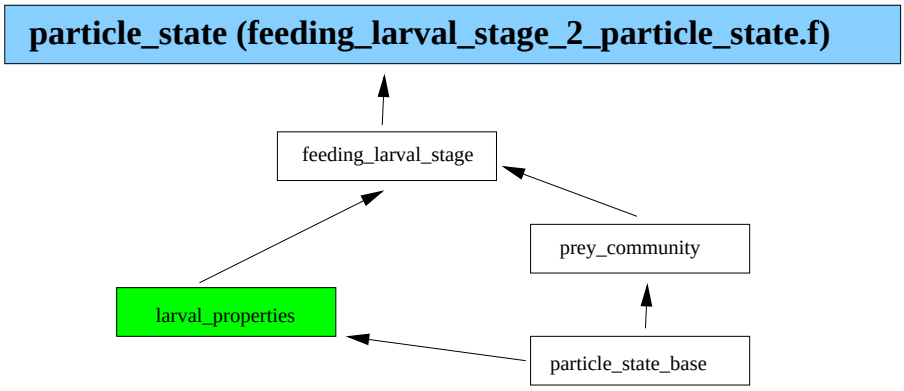

Figure 6.2: Inheritance diagram for optimal foraging theory implementation of feeding_larvae

- subroutine update_particle_state_X(state, space, dt, mortality_rate, die, next)
  type(X), intent(inout) :: state
  type(spatial_attributes), intent(inout) :: space
  real,intent(in) :: dt ! in seconds
  real,intent(out) :: mortality_rate
  logical,intent(out) :: die, next

- subroutine init_state_attributes_X(state, space, time_dir, initdata)
  type(X),intent(out) :: state
  type(spatial_attributes),intent(inout) :: space
  real,intent(in) :: time_dir
  character*(*),intent(in) :: initdata

- subroutine get_active_velocity_X(state, space, v_active)
  type(X), intent(in) :: state
  type(spatial_attributes), intent(in) :: space
  real, intent(out) :: v_active(:) ! meter/second

Notes that the particle_sub_state interface is rather similar to the particle_state interface, but that update_particle_state_X features extra output parameters mortality_rate, die, next. These parameters are needed by the multi stage manager to make stage transitions and manage overall properties, like survival of the organism. The particle_sub_state interface for a particular stage will only be applied if the stage is currently active.

6.4.2 Growth energy budgets for early life stages

module feeding_larval_stage in the implementation optimal_forager.f implements foraging based on optimal foraging theory (Charnov 1975). Diet switching is controlled completely by size-based processes in capture_sucess, handling_time, search_volume provided by module larval_properties. The continuous food spectrum is size based as provided by module prey_community with prey_spectrum_density.

optimal_forager.f can determine the maximal food intake (corresponding to the optimal diet) from a continuous food spectrum either by on-the-fly diet optimization or from a pretabulated data base of optimal diets for a reference database set in the input file. The latter mode is for production runs, to avoid optimization for each organism in each time step. If the reference database is set appropriately, the results should be similar, to within numerical accuracy. To set reference database, see input parameters for module feeding_larval_stage below.

6.5 Module listing

6.5.1 Decorators

module feeding_larval_stage

implementation : optimal_forager.f. particle_state interface/sub interface for feeding larval stage This version applies optimal foraging theory in relation to a size spectrum, based on larval performance capture_sucess/handling_time/search_volume prescribed in module larval_properties

```
type feeding_larvae
      private
         type(larval_physiology) :: larvstate         ! defined in module
      end type
      public :: feeding_larvae ! make type state_attributes visible outsi

 type state_attributes



      public :: init_particle_state  ! module operator
      public :: close_particle_state ! module operator
      public :: init_state_attributes
      public :: get_active_velocity
      public :: update_particle_state
      public :: delete_state_attributes
```

```
public :: write_state_attributes
public :: get_particle_version
public :: get_property         ! currently void
public :: get_metadata_state   ! currently void

public :: init_feeding_larval_stage
public :: close_feeding_larval_stage
public :: init_state_attributes_FL
public :: get_active_velocity_FL
public :: update_particle_state_FL
public :: delete_state_attributes_FL
```

Input parameters (for simulation file):

- ingestion_integral_sampling (integer). Diet integrals over size spectra are calculated using numerical integrals. This parameter tells how many sampling points are used for numerical integrals.
- precalc_ingestion_DB (logical). Whether to generate a database of maximal intake (.true.) or calculate maximal intake on the fly for each larvae in each time step (.false.) If precalc_ingestion_DB is .true. then the following parameters specifying the reference database is generated
  - larval_length_grid_min (mm) feeding larval range, start
  - larval_length_grid_max (mm) feeding larval range, end
  - larval_length_grid_points (integer) sampling of larval range.
  - logZ_grid_min (log(kg DW/m3)) Zooplankton range, start
  - logZ_grid_max (log(kg DW/m3)) Zooplankton range, end
  - logZ_grid_points (integer) sampling of zooplankton range.
  - julian_day_grid_points (integer) sampling of seasonal cycle

module yolksac_larval_stage

implementation : yolksac_larval_stage.f

particle_state sub interface for yolksac_larval_stage for multi stage manager. Species specific properties comes from module yolksac_properties

```
type yolksac_larvae
      private
         real :: completion   ! 0 < completion < 1. Hatch at 1
      end type
      public :: yolksac_larvae ! make type state_attributes visible outsi


c.....Particle sub state interface: (for multi stage manager)
      public :: init_yolksac_larval_stage  ! module operator
      public :: close_yolksac_larval_stage ! module operator
      public :: init_state_attributes_YL
      public :: get_active_velocity_YL
      public :: update_particle_state_YL
      public :: delete_state_attributes_YL
```

module released_egg_stage

implementation : released_egg_stage.f particle_state sub interface for released_egg_stage for multi stage manager. Species specific properties comes from module yolksac_properties

```
type released_egg
      private
         real :: completion   ! 0 < completion < 1. Hatch at 1
      end type
      public :: released_egg ! make type state_attributes visible outside


c.....Particle sub state interface: (for multi stage manager)
      public :: init_released_egg_stage  ! module operator
      public :: close_released_egg_stage ! module operator
      public :: init_state_attributes_egg
      public :: get_active_velocity_egg
      public :: update_particle_state_egg
```

```
public :: delete_state_attributes_egg
```

module particle_state

implementation : early_life_stages.f multi stage manager implementing particle_state interface from merging sub interfaces
released_egg_stage + yolksac_larval_stage + feeding_larval_stage

6.5.2 Species specific modules

These - currently three - modules are the only ones that contain species specific information. When setting up a new species, it is these modules that should be implemented for a particular species. A good way to start is to look at an example implementation

module egg_properties

example implementation : sprat/sprat_egg.f

```
public ::  egg_devel_rate ! fraction/sec
```

module yolksac_properties

example implementation : sprat/sprat_yolksac_larv.f

```
public ::  yolksac_absorb_rate  ! fraction/sec
```

module larval_properties

example implementation : sprat/sprat_feeding_larv.f
TODO: deriva

```
type larval_physiology              ! component for state_attributes
        real    :: length           ! mandatory public entry: standard length in mm
        real    :: weight           ! mandatory public entry: standard DW weight in m
end type

public :: init_larval_properties   ! module initialization
```

```
public :: close_larval_properties  ! module close down

public :: init_larval_physiology   ! class constructor
public :: close_larval_physiology  ! class destructor
public :: set_larvae_hatched       ! alternative class constructor

public :: length_to_nominal_weight ! length-weight key
public :: weight_to_nominal_length ! corresponding weight-length key

public :: capture_sucess           ! evaluate capture success of encounte
public :: handling_time            ! evaluate handling time
public :: search_volume            ! evaluate search volume
public :: grow_larvae              ! update larval weight+length
public :: inquire_stage_change     ! inquire whether stage should change
public :: evaluate_mortality_rate
```

Dummy arguments in subroutine synopsis

- lp: prey length (mm)
- llarv: larval length (mm)
- local_env: local environment object
- space: space attribute object for current position of the larvae
- weight: larval weight (μg)

Subroutine synopsis:

- subroutine init_larval_properties()
- subroutine close_larval_properties()
- subroutine init_larval_physiology(self,llarv,weight)
  set larval length llarv (mm) and weight (μg)
- subroutine close_larval_physiology(self)

- subroutine set_larvae_hatched(self, space)

- subroutine length_to_nominal_weight(llarv, weight)

- subroutine weight_to_nominal_length(weight, llarvh)

- subroutine capture_sucess(lp, llarv, local_env, csuc, dcsuc_dlp)
  prey capture success csuc (propability) and optionally its derivative wrt. lp: dcsuc_dlp

- subroutine handling_time(lp, llarv, local_env, ht, dht_dlp)
  prey handling time ht [seconds] and optionally its derivative wrt. lp: dht_dlp

- subroutine search_volume(lp, llarv, local_env, svol, dsvol_dlp)
  Search volume svol (unit = mm3/sec) of larvae with length llarv wrt. prey of length lp and optionally its derivative wrt. lp: dsvol_dlp)

- subroutine grow_larvae(self, local_env, irate, dt)
  Update larval mass and length corresponding to interval dt subject to (maximal) ingestion rate.

- subroutine inquire_stage_change(self, next)
  Test whether to advance (or regress) ontogenetic stage Return next = .true. to advance (or regress) to ontogenetic stage (depending on sign of dt) Do not consider sign of simulation time arrow assumes weight/length are appropriately updated

- subroutine evaluate_mortality_rate(self, local_env, mortality_rate, die)
  Based on the current state and in the current environment evaluate the current mortality_rate and wheter the larvae should die (logical) absolutely (avoid setting mortality_rate = infinity)

6.5.3 Auxillary modules

module particle_state_base

implementation : particle_state_base.f generic services for particle_state implementations

```
type local_environment   ! cache local environment to avoid multiple iden

public :: probe_local_environment
public :: write_local_environment
public :: clear_local_environment
```

module prey_community

implementation : prey_community.f
reconstruct prey size spectrum

```
public :: prey_mass                ! length-weight key for prey
public :: prey_spectrum_density    ! evaluate size spectrum
```

module particle_state

implementation: feeding_larval_stage_2_particle_state.f transparent wrapper to change name of module feeding_larval_stage to particle_state

6.6 The task interface

A task interface must be associated with a separate directory under directory `task_providers`

6.7 Other modules

6.7.1 geometry

- `dcart2dxy(xy, r2)` Convert a small Cartesian displacement vector r2 (meters) inplace in tangent space at xy to a displacement vector in (longitude,latitude,depth).

- `dxy2dcart(xy, r2)` Convert a small displacement vector in (longitude,latitude,depth) inplace in tangent space at xy to a Cartesian displacement vector (meters).

  for xy an xyz may be provided (only xy is transformed) to get the Jacobian at xyz:
  jacobian = (/1,1/)
  call dxy2dcart(xyz, jacobian)

  we will add a convenience subroutine: get_jacobian ...

- `add_finite_step(xy, dR)` inplace add a finite Cartesian vector dR(2/3) in meters to xy including the effect of sphere curvature (i.e. not based on tangent space arithmetics) Make a isospheric translation of length —dR(1:2)— in the direction of dR(1:2). If dR has length 3, add the vertical component afterwards i.e. xy(3) = xy(3) + dR(3). Tangent space arithmetics is obtained by:
  call d_cart2d_xy(xyz,dR);
  xyz = xyz + dR

- `get_horizontal_distance(xy1, xy2, r)` m
  r is the sphere distance in meters between xy1, xy2 (i.e. distance when travelling at the ideal sea surface between xy1, xy2).

- `get_local_distance(xyz1, xyz2, v)` m

  Overloaded function. If v is scalar, return distance in meters between xyz1 and xyz2 in the average tangent spaces of xyz1 and xyz2 (so the function is symmetric in xyz1 and xyz2). If v is a vector, return distance vector in meters (oriented from xyz1 to xyz2)

Chapter 7

Installation notes

Even though IBMlib code and build tools should stricktly adhere to standards, there may be minor issues when setting IBMlib up in a new computing environments. Below we have logged the work arounds for these cases, organized by computing environment.

7.1 DTU HPC (gbar)

7.1.1 Installation

Mark Payne, Thursday, August 22, 2013 6:14 PM (committed to version 445):

```
#login to HPC front end
ssh <username>@hpc-fe.gbar.dtu.dk
#load  an interactive node
qrsh
#Check out IBMlib
#Load modules required. PGI 2011 gives the newinclude problem, but not 2012.
module load pgi/2012
#Also need to load netcdf built with pgi
module load netcdf/pgi-4.2
#Configure IBMlib as usual but use the compiler defaults I have already setup for por
 ln -sf setups/compilers/pg/pgi_2012_compiler_DTU_HPC.mk compiler_defaults.mk
#Use the PGI makefile that I have already modified
```

```
rm Makefile
ln -sf Makefile_PGI Makefile
#And build.
make all
```

Chapter 8

FAQ (programmers)

8.1 Compilation

8.1.1 My XXX module requires a special compilation flag

In the sub-makefile write tt $(FCFLAGS) += special_compiler_flag(s)". This adds the special compilation flag to the other flags applied to the module. When recommitting source code to SVN such cases must be cleaned up to ensure portability. If the compilation flag are very special (only provided by a few compilers) we recommend you to find a programming solution to the problem. If it is a generic compilation flag write tt $(FCFLAGS) += GENERIC_DESCRIPTION in the sub-makefile and then define tt GENERIC_DESCRIPTION in compiler_defaults.mk. This allows to specify the special compilation flag for other compilers as well to maintain portability of IBMlib.

8.2 Execution error/warnings

8.2.1 IMBlib writes "checkstat: ... error= ..." and dies

Some data sets/interpolators display problems, usually related to boundaries, where interpolate_X raises some exceptions If the problem is

being analyzed or is understood, you can set `checkstat_action = warn_and_continue` (or `checkstat_action = ignore_exceptions`) to prevent IMBlib from stopping the simulation. Remember to recompile. This does not solve the underlying problem but only hide the symptoms. The underlying problem may be (but not lilited to):

- The hydrographic data set may contain inconsistencies (e.g. imply negative water depths)
- The interpolate_X in the actual PBI has not been carefully set up to handle positions close to boundaries
- coast_line_intersection in the actual PBI has subtle problems letting particles displace onto land

8.3 Debugging

8.3.1 Can I stop the code at some specific conditions ?

Old school programmer like to put in write statements strategic places in the code and thereby narrow in the problem. But does your problem happen only for particle number 7545345 first after 73 hours simulation ? Surely you will kill your hard disk if you write out all diagnostic information. There is a general low tech mechanism enabled in IBMlib to handle this at programming level. In module run_context there is a global public integer flag `debug_baton` that is initialized to 0 at compile time. It allows any including level to signal and another to capture a stop/write request. Simply when the problem appears, set `debug_baton` to a non-zero value and let other parts of IBMlib listen to (and respond to) `debug_baton /= 0`. Especially, when you have written the diagnostic information you want, put the line `if (debug_baton /= 0) stop` somewhere strategic in the code and recompile. (Remember to delete it again, when the problem is solved)

Chapter 9

Code development guide lines

9.1 Design principles and standards

- Core code is in strict standard conforming Fortran 90.
- Code design should adhere to object-oriented standards to the extend that it is supported by Fortran 90.
- Maximal privatization of module/derived type content. Each module should have "private" as default scope
- Always use "implicit none"
- Fixed format layout (start at col 7, end at max 72)
- Meaningfull names for variables, but not too long
- Extensive in-code documantation and in-code change log should be provided at addition/changes.
- Light weight subroutines: do not bundle too much functionality together, keep it simple.
- Prefer subroutines over functions (easier to trace)

- Each module should have a private declaration "integer, parameter :: verbose = 0" allowing for debugging when verbose > 0
- One-module-one-file: only one module in each file fortran file. Otherwise it is more complex to handle with make.
- Module name and file name must correspond for all modules (i.e. amodule.f $\rightarrow$ amodule.mod + amodule.o). However, task, oceanography and biology providers are excepted from this rule as only exceptions

9.2 Version tracking with SVN

- After changes and before SVN commit:
  - Modifications should be tested before SVN commit
  - IBMlib must be able to complete

    make test

    before SVN commit (otherwise your changes indirectly broke something).
- Each commit should be accompanied by sufficient documentation to trace and understand changes.

Chapter 10

Functionality

The code package contains several templates that implements each of the IBMlib interfaces

10.1 task_providers

10.2 oceanography_providers

synthetic fields

10.3 biology_providers

particle

10.4 Reading simulation input

Reading input can be done in any desired standard/non standard way in user porvided modules. Currently, the simulation input is based on a tagged input format (implemented in input_parser.f and its template). Input is ASCII formatted as

$$tag = value[value*] \tag{10.1}$$

tag is a character identifier terminated at first occurence of the separator =. value is what follows the separator which is cached as a string buffer at first reading. The format supports (and ignores) comments, extra spaces, spaces, empty lines non-matching lines. Tags are case sensitive. Order of tags does not matter (but if a tag appears multiple times, order of appearence input file affects the caching order of the input parser This input format is convenient for scripting, where upper-level scripts generates input files. The input parser reads the input file once and caches input, when the simulation input file is opened. The input parser offers and (overloaded) query function

- `read_control_data(filehandler, tag, value [, next])`

so that the value of *tag* is interpreted at query time by the data type of value. Value may be a single variable or a vector of any standard Fortran data type. If the value is a mixed data type (e.g both strings and numbers), it should be read with value as a raw string buffer and post parsed after reading with `read_control_data()`. The optional argument next is a stepping control (technically referring to line number where scanning for tag starts). This allows to handle multiple occurences of tag.

10.5 Other service providers