IBMlib concept description

July 15, 2010

Contents

0.1 Unresolved issues

- How should we include Kritins algorithms ?

Part I

User guide

Chapter 1

Overall concepts

IBMlib is a particle simulation toolbox intended research purposes. 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 provides templates for developments. 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 access 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 tracer grow and swim, 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. Alltogether, IBMlib_core and optional modules for physical environment, biological properties and tasks constitutes IBMlib; when an actual physical, biological and task has been selected, we refer to it as an IBMlib configuration.

online/offline mode

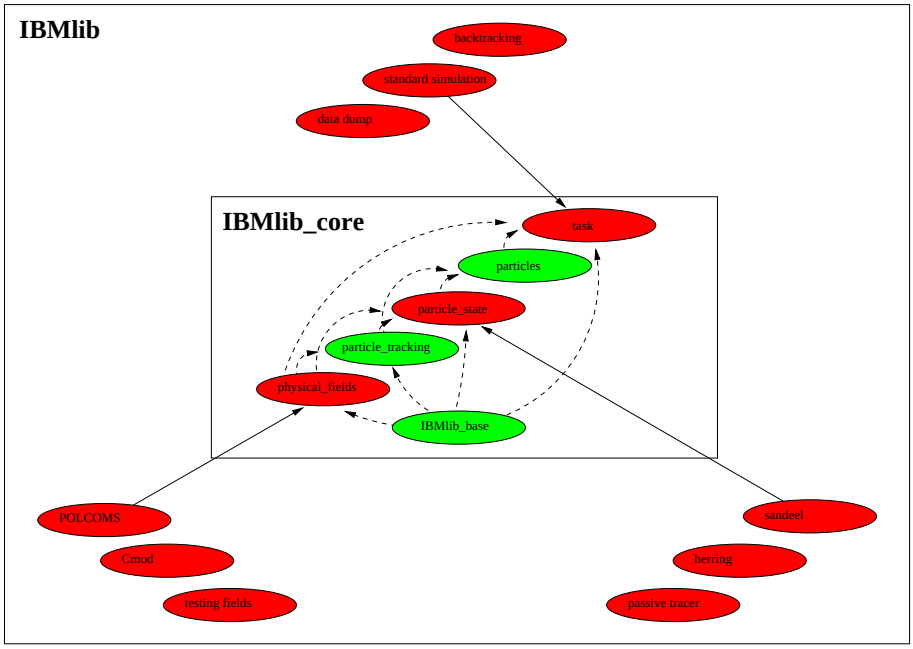

Figure 1.1: Concept diagram of the IBMlib framework.

Chapter 2

License issues

currently none, 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.

3.2 System resources and requirements

- Fortran 90 compiler. All performance code is written in Fortran 90.

- Gmake. All makefiles are written for and tested on 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 tasks constitutes 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.

3.3 Configuration

You configure IBMlib by selecting sub modules for physical environment, biological properties and task. In the Makefile, look for make variables `PHYSICAL_FIELDS_DIR, PARTICLE_STATE_DIR, TASK_DIR` and point these to the directories corresponding to the desired sub modules.

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

3.4 Building an IBMlib configuration

When the makefile is configured, you simply type [make] on the command line, which produces and executable `ibmrun` (or another target name specified in make variable `EXECUTABLE`). This executable may be copied to somewhere in the standard executable search PATH, if it should be installed as an application.

3.5 Running an IBMlib configuration

When the executable is build, you simply type [ibmrun [¡arguments¿]] 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.

3.6 Simulation files

When IBMlib is used in a standard task configuration, it expects you to provide an input file on the command line like [ibmrun ¡inputfile¿ [¿ ¡outputfile¿]] All parts of the compiled IBMlib configuration will try to read most input data from this file in a standard task configuration. 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 patricle time step. In this case the first occurence is picked, the rest is ignored. The following rules also apply for the markup:

- Comments: everything after (and including) " !" is skipped
- Empty lines and spaces: just ignored
- Malformed lines: just ignored (e.g. if you forgot "=")
- Everything after "=" is considered part of the value for tag
- Required tag (or value) is missing. This will generate a runtime error, SLAM will stop, unless a default applied to the tag.
- Order of tags does not matter.

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

3.6.1 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 absolute depth (counted negative below the water surface) or as 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. There can be an "e", which means sandeel eggs are released by this emitbox; the can be an "l" which means sandeel larvae are released by this emitbox - in the latter case, a number giving the inital length (in mm) of the released sandeel larvae should be provided.

3.6.2 Particle_tracking module

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

3.6.3 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  2005 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 six dependency levels

1. TASK
2. particles.mod
3. PARTICLE_STATE
4. particle_tracking.mod
5. PHYSICAL_FIELDS
6. 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 syncrinization. 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.

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_salty(xyz, r, status)` PSU

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

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

- `[interpolate_wind(xy, r2, status)]` m/s

  surface wind vector with standard orientation in m/s

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.

- `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.

- `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)

- `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 → 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.

6.2.4 Public transformation functions

- `d_cart2d_xy(xy, r2)` Convert a small Cartesian displacement vector r2 (meters) inplace in tangent space at xy to a displacement vector in (longitude,latitude,depth).

- `d_xy2d_cart(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 d_xy2d_cart(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

6.2.5 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.

- `init_particle_state()`

  Module start-up method

- `close_particle_state()`

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

- `init_state_attributes(state_stack, space_stack, time_dir, nstart, npar, initdata, emitboxID)`

  Initialize a subarray state_stack(nstart:nstart+npar-1) of particle_state instances (associated with space attributes space_attr(nstart:nstart+np 1). 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 paramemers 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 The task interface

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

6.5 The general module interface

Chapter 7

Code design principles and standards

maximal privatization of module/derived type content always implicit
none fixed format layout (start at col 7, end at max 72)
meaningfull names but not too long
extensive in-code documantation light weight
self test prefer subroutines over functions (easier to trace) cvs
Each module should have a private declaration "integer, parameter :: verbose = 0"
allowing for debugging when verbose ¿ 0
1a) one-module-one-file: only one module in each file 1c) a simple
consistent case convention for module names (so `module_dependence_scan.py`
can express dependency rules)

Chapter 8

Development time line

particle classes

Chapter 9

Functionality

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

9.1 task_providers

9.2 oceanography_providers

synthetic fields

9.3 biology_providers

particle

9.4 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

$$tag = value[value*] \tag{9.1}$$

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.

9.5 Other service providers