Selalib Notes - DRAFT -

Edwin Chacon-Golcher

April, 2012

Abstract

This is a draft for a guide to the facilities offered by Selalib, the Semi-Lagrangian Library. It serves as a record for module descriptions, design decision, pending issues, etc., which are relevant for the development and improvement of the library. The main focus is the exposed API, thus the document also serves as a working document for the architectural specifications. More specific implementation details should be found in the comments within the source code. We may decide later to extract those either with an automatic system or in a developer's manual. At the time of writing this version of the draft, the library contains only a few modules in the lower levels and aims at having a minimalist structure. Of those capabilities that could be deemed interesting enough by the managers of GYSELA, the intent would be to incorporate them that code. In this sense, this library prototype can also be a testbed for ideas and suggestions that could eventually be absorbed into production code.

Contents

Chapter 1

Introduction

Selalib is the *Se*mi-*La*grangian *Lib*rary, a collection of types and its associated methods that are useful for creating parallel plasma physics simulations that use this specific methodology for the solution of the *Vlasov* equation. In its design, we have attempted to expose to the users an interface that expresses as naturally as possible the problems that arise when using the semi-lagrangian approach. The present version of the top-level types and interfaces is the same as discussed in Selalib's project meeting in January 2011.

Selalib is structured in layers. One can think about these layers as libraries in their own right. A given layer can use the capabilities offered by a lower layer through the exposed interface, but never from a higher level. The layer with the lowest level of abstraction presently contains basic utilities like memory allocators, assertions and basic numeric types. The second layer is composed of numerical and parallel utilities. The third and highest level of the library contains the semi-lagrangian methodology tools, types and methods. This manual will ultimately describe each of these layers.

Most of the native types and operations provided by Selalib are prefixed by `sll_`. In this way you can at least have an expectation of finding documentation (if a user) and a starting point of where to start looking if you wish to dive into some particular aspect of an implementation (if a developer). This is an early implementation, so the prefix to designate Selalib's features could change. It is also worth considering to eliminate the prefix for the lower layers of the library, as these are general/reusable components that could in principle be used in different projects. In such case, the `sll_` prefix would be reserved for the top-level layer that implements the specific semi-lagrangian functionality.

The Selalib prototype is written in Fortran 95. There are some aspects of its implementation that may warrant some commentary as these aspects have an impact on how the library is used. For instance, the full library would be imported by the declaration:

```
#include "selalib.h"
```

instead of the more Fortran-natural form:

```
use selalib
```

The reason for this is that some features of the library are implemented as *macros*. To the user of the library, it makes no difference whether some functionality is implemented in the form of a procedure or a macro, with the exception that presently, macro names are written in `ALL-CAPS` (but we could change this). The use of the macro is required to offer certain capabilities, like informative error messages. For a developer, the use of the macros is needed in many cases to reduce code redundancy. The need for the use of macros will hopefully be more understandable when the reader sees the behavior of calls to simple macros like `SLL_ALLOCATE()` or `SLL_ASSERT()`.

A macro is a pre-processor directive. Selalib uses only very simple macros that are handled by *fpp*, the Fortran Pre-Processor. *fpp*'s capabilities are very limited and one peculiarity of its output is that macros are expanded into a single long line, which can easily surpass the 132 character limit that Fortran systems have. For this reason alone, the compilation of Selalib requires the use of a compilation flag: `-ffree-line-length-none` (in **gfortran**) or its equivalent in another compiler. On a similar vein, some compilers require the extension `.F90` (as opposed to `.f90`) in order to apply a preprocessing step. Thus, all files in Selalib use the `.F90` extension, and this is another reason that clients of this version of the library should as well.

Chapter 2

Low-Level Layer: Basic Utilities

2.1 Numeric Types

2.1.1 Description

As a *convenience*, Selalib offers aliases to some of Fortran's basic numeric types. This is intended to:

1. concisely and uniformly make clear the intended precision of a given variable,
2. permit mixed precision representations when needed (for example, a developer could wish to represent a particular real number with a combination of a 32-bit integer and a 32-bit real instead of a single 64-bit real as is sometimes done in ultra-high performance software),
3. provide a centralized location to change the precision of a numerical representation program-wide, and
4. save from the typing of a few characters.

2.1.2 Exposed Interface

Selalib's numeric precision features are accessed through the following aliases:

alias	shorthand for...
`sll_int32`	`integer(kind=i32)`
`sll_int64`	`integer(kind=i64)`
`sll_real32`	`real(kind=f32)`
`sll_real64`	`real(kind=f64)`
`sll_int`	`integer(kind=user)`
`sll_real`	`real(kind=user)`
`sll_comp32`	`complex(kind=f32)`
`sll_comp64`	`complex(kind=f64)`

Where the kind type parameters `i32`, `i64`, `f32` and `f64` have been defined to give a representation that is at least the denoted size for a given number. `user` is available for a flexible kind type parameter. These kind type parameters can also be used to specify the precision of numerical constants in the usual Fortran way, i.e. `3.14159265_f32`.

2.1.3 Usage

To use the module in a stand-alone way, use the line:

```
#include "sll_working_precision.h"
```

The aliases are to be used like the native Fortran types that they are aliasing:

```
sll_real64 :: my_pi
sll_real64 :: theta
sll_int32  :: i
sll_int32  :: N
sll_real64, dimension(:), allocatable :: w
! allocate w ...
my_pi = 3.1415926535897932384626433_f64
theta = 2.0*my_pi/real(N,f64)
do i=1,N/2
     w(i) = exp((0.0,1.0_f64)*theta*real(i,f64))
end do
```

2.1.4 Status

Unit-tested.

2.2 Memory Allocator

2.2.1 Description

Selalib's memory allocators are simple wrappers around Fortran's native allocators. We ask very little from these allocators:

1. to allocate memory,
2. to initialize it if requested (only for Fortran-native types),
3. to deallocate memory, and
4. to fail with as descriptive a message as possible.

2.2.2 Exposed Interface

The interface to these allocators follows very closely the interface of Fortran's `allocate()` and `deallocate()` functions. The user may decide to allocate arrays or pointers of any type, and up to the same number of dimensions as permitted by a given Fortran implementation. The exposed macros are:

```
SLL_ALLOCATE( array_and_lims, ierr )
```

This is the basic memory allocator with the same syntax as Fortran's native `allocate()` but with a required integer error parameter. Any type and dimension can be given as an argument.

```
SLL_CLEAR_ALLOCATE( array_and_lims, ierr )
```

Same behavior as `SLL_ALLOCATE()` but also initializes the allocated memory to zero. This works for Fortran native types only or for derived types for which an assignment operator $(= 0)$ has been defined.

```
SLL_DEALLOCATE( array_and_lims, ierr )
```

Basic deallocator. It is a wrapper around Fortran's native `deallocate()` function but also nullifies the pointer given as an argument after deallocation.

```
SLL_DEALLOCATE_ARRAY( array, ierr )
```

The array deallocator differs from the previous in that it does not attempt to nullify the array name after the deallocation is complete. This and the previous macro could in principle be merged into one for simplification.

```
SLL_INIT_ARRAY( array, val )
```

While not really an allocator/deallocator, we expose this macro for convenience in initializing an array with a given constant value. For consistency, it may be decided to eliminate this from the interface.

In contrast with the native `allocate()`, Selalib's allocators require that the user provide an integer variable for error checking (`ierr`).

2.2.3 Usage

To use the memory module in stand-alone fashion, use the line:

```
#include "sll_memory.h"
```

An example of the use of the module is:

```
    integer :: err
    real, dimension(:), allocatable :: a
    real, dimension(:,:,:), pointer :: b=>null()

    SLL_ALLOCATE( a(5000), err )
    SLL_CLEAR_ALLOCATE( b(1:4,1:3,1:2), err)
    SLL_INIT_ARRAY(b,0)
```

When finding an error condition, the user is informed about the location of the failing call:

```
Memory allocation Failure. Triggered in FILE tester.F90,
in LINE:   35
STOP ERROR: test_error_code(): exiting program
```

2.2.4 Status

Unit-tested.

2.3 Assertions

2.3.1 Description

This is a very small but very useful capability that permits the developer to devise many sorts of defensive programming techniques. The simple idea is that of declaring a condition that is expected to be true, thus triggering a descriptive error if the condition is violated.

2.3.2 Exposed Interface

Wherever a specific condition needs to be asserted, simply write:

```
    SLL_ASSERT( logical_condition )
```

Thus the condition to be checked must be cast in the form of a logical statement. Falseness of such statement would trigger an assertion error.

The assertions can be used liberally since they are controlled by a `-DEBUG` flag at compilation time. Absence of this flag will delete all calls to `SLL_ASSERT()` from the code. Hence, assertion conditions can be used during a debugging or testing phase without increasing the overhead in production code.

2.3.3 Usage

To use the *assertions* module in a stand-alone way, use the line:

```
#include "sll_assert.h"
```

However, all the following steps are necessary:

1. to make sure that the macros expand properly when using **gfortran**, include the compilation flag `-DGFORTRAN`. This is to ensure that the assertion conditions can be properly converted into strings to be returned in case that an assertion is triggered,
2. when using **gfortran**, use pass along the flag `-ffree-line-length-none` to prevent compilation error as some of the included macros may go beyond Fortrans 132 character limit,
3. to activate the assertions, pass along the flag `-DDEBUG`. If this flag is not present, the assertions will compile into nothing, which may be desirable after the code has been debugged.

An example may be the checking of in-range indices on a protected array:

```
function get_val( a, i )
     sll_int32 :: get_val
     sll_int32, intent(in) :: i
     sll_int32, dimension(:), intent(in) :: a
     SLL_ASSERT( (i .ge. 1) .and. (i .le. size(a)) )
     get_val = a(i)
end function get_val
```

Which could produce a behavior such as:

```
$./unit_test
     The size of a is: 1000
     a(1) =    0
     a(117) =    0
     Here we ask for the value of a(1001):

     (i .ge. 1) .and. (i .le. size(a)) : Assertion
     error triggered in file unit_test.F90 in line       25
     STOP :  ASSERTION FAILURE
```

Such way of stopping a program is much less uncomfortable than the sinking feeling one has when the program stops as in:

```
$ ./unit_test
     Array values:
     The size of a is: 1000
     a(1) =    0
     a(117) =    0
     Incident de segmentation (core dumped)
```

which of course doesn't even need to happen at the moment of the first error.

2.3.4 Status

Unit tested.

2.4 Timing Utility

2.4.1 Description

Selalib has its own timing facility, capable of resolving time periods slightly shorter than 2 μs.

2.4.2 Exposed Interface

In a tentative way, presently we hide the fact that the timing utility is based on external C functions and the `iso_c_binding` features of $f2003$. The module's capabilities are brought into the client code by the line:

```
#include "sll_timer.h"
```

which defines the type `time_mark`. Instances of this type can be used to store clock readings. The timing utility also allows to measure the time elapsed since a particular time mark was set. The available functions are:

```
set_time_mark()
reset_time_mark( time_mark )
time_elapsed_since( time_mark )
```

`set_time_mark()` is a function with no arguments that returns an instance of the `time_mark` type, set to a clock reading done at the moment of the function call.

`reset_time_mark( time_mark )` is also a function, and has essentially a redundant role to `set_time_mark()`. It will take a time mark and return it updated with a new clock reading.

`time_elapsed_since( time_mark )` is a function that takes a time mark as an argument and returns the time, *in seconds* since the clock reading contained in the passed argument.

2.4.3 Usage

Here is an example of the use of this module:

```
program timer_test
#include "timer.h"
implicit none
integer(8) :: i, j, k
```

```
5     real(c_double), dimension(:), allocatable :: dt
6     time_mark      :: tmark
7
8     #define ITERATIONS 1000
9
10    allocate(dt(ITERATIONS))
11
12    do i=1,ITERATIONS
13       tmark = reset_time_mark(tmark)
14       do k=1,100000    ! Just a delay
15          j = i * i - i
16       end do
17       dt(i) = time_elapsed_since(tmark)
18    end do
19    ! print diagnostics here, averages, min, max, etc.
```

In brief:

Line 2: Imports the timer module.

Line 5: Here we define an array to store the results of the timing test. This array stores the return values of the function `time_elapsed_since()`, which returns a double precision value (`sll_real64`).

Line 6: Declaration of the time marker.

Lines 13: Sets the time marker with a current clock reading.

Line 17: We store the result of the timing operation.

Any number of time markers can be declared and set at different points. So this minimal interface should be sufficient for most needs. An important point is that this module requires linking with the *real time* system library, thus `-lrt` should appear as an instruction to the linker during the build.

2.4.4 Status

Unit-tested.

2.5 Low-Level File IO

2.5.1 Description

Presently, Selalib offers facilities to output data in the XDMF file format, readable by the VisIt program. To use, import the module `sll_low_level_file_io`. The interface allows to open a file, write multi-dimensional data in it and then close it. The main purpose of this module is to provide the low-level building blocks that can be used by higher-level functions to conveniently write data to files.

2.5.2 Exposed Interface

The current interface is a collection of generic functions:

```
sll_xdmf_open( file_name,
               mesh_name,
               nnodes_x1,nnodes_x2,[nnodes_x3,]
               file_id,
               error )
sll_xdmf_write_array( mesh_name,
                      array,
                      array_name,
                      error,
                      xmffile_id,
                      center )
sll_xdmf_close(file_id,error)
```

Where the parameters in brackets are used depending of the dimensionality of the data.

`sll_xdmf_open( )` opens a file with a given name and returns a file identifier on `file_id`.

`sll_xdmf_write_array( )` writes a given multi-dimensional array on a file identified by `xmffile_id`.

`sll_sdmf_close( )` closes the given file.

2.5.3 Critical Commentary

ECG: This interface can be improved. To consider:

The job of `sll_xdmf_open( )` should be to open a file, with a given filename; it should take as parameters some flags to indicate what behavior to exercise in case of error or if the file exists (overwrite? append?) and nothing else. The nnodes parameters are used to write XDMF-related preambles, but this should be sent to the other functions that actually write into the file.

The mesh_name parameter is not clear. Why mesh?

What is the 'center' parameter in the write_array() function?

Chapter 3

Mid-Level Layer: Numerical and Parallel Utilities

3.1 Tridiagonal System Solver

3.1.1 Description

To solve systems of the form $Ax = b$, where A is a tridiagonal matrix, Selalib offers a native, robust tridiagonal system solver. The present implementation contains only a serial version. The algorithm is based on an LU factorization of a given matrix, with row pivoting. The tridiagonal matrix must be given as a single array, with a memory layout shown next.

$$\begin{bmatrix} a(2) & a(3) & & & & & a(1) \\ a(4) & a(5) & a(6) & & & & \\ & a(7) & a(8) & a(9) & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & a(3n-5) & a(3n-4) & a(3n-3) \\ a(3n) & & & & & a(3n-2) & a(3n-1) \end{bmatrix}$$

3.1.2 Exposed Interface

Factorization of the matrix A is obtained through a call to the subroutine

```
sll_setup_cyclic_tridiag( a, n, lu, ipiv )
```

where `a` is the matrix to be factorized, `n` is the problem size (the number of unknowns), `lu` is a real array of size $7n$ where factorization information will be returned and `ipiv` is an integer array of length `n` on which pivoting information will be returned. From the perspective of the user, `lu` and `ipiv` are only arrays that `sll_setup_cyclic_tridiag` requires and do not need any further consideration.

The solution of a tridiagonal system, once the original array A has been factorized, is obtained through a call to

```
sll_solve_cyclic_tridiag( lu, ipiv, b, n, x )
```

where `lu`, `ipiv` are the arrays returned by `sll_setup_cyclic_tridiag()`, `b` is the independent term in the original matrix equation, `n` is the system size and `x` is the array where the solution will be returned.

3.1.3 Usage

To use the module in a stand-alone way, include the line:

```
use sll_tridiagonal
```

The following code snippet is an example of the use of the tridiagonal solver.

```
sll_int32 :: n = 1000
sll_int32 :: ierr
sll_real64, allocatable, dimension(:) :: lu
sll_int32,  allocatable, dimension(:) :: ipiv
sll_real64, allocatable, dimension(:) :: x

SLL_ALLOCATE( lu(7*n), ierr )
SLL_ALLOCATE( ipiv(n), ierr )
SLL_ALLOCATE( x(n), ierr )

! initialize a(:) with the proper coefficients here...
and then:

sll_setup_cyclic_tridiag( a, n, lu, ipiv )
sll_solve_cyclic_tridiag( lu, ipiv, b, n, x )

 SLL_DEALLOCATE_ARRAY( lu, ierr )
 SLL_DEALLOCATE_ARRAY( ipiv, ierr )
 SLL_DEALLOCATE_ARRAY( x, ierr )
```

Note that if the last call had been made as in

```
sll_solve_cyclic_tridiag( lu, ipiv, b, n, b )
```

the system would have been solved in-place.

3.1.4 Status

Unit-tested.

3.2 Splines

3.2.1 Description

The splines module provides capabilities for 1D data interpolation with cubic B-splines and different boundary conditions (at the time of this writing: periodic, hermite). The data to be interpolated is represented by a simple array. The spline coefficients and other information are stored in a spline object, which is also used to interpolate the fitted data.

3.2.2 Exposed Interface

Fundamental type:

```
sll_spline_1d
```

Like all the other fundamental types in Selalib, this type is declared as a pointer and only manipulated through the functions and subroutines described below. For more explicit examples, see the usage section.

Available functions:

```
new_spline_1D( num_points,
               xmin,
               xmax,
               bc_type,
               slope_L,
               slope_R )
set_slope_left( spline_object, value )
set_slope_right(spline_object, value )
compute_spline_1D( data, bc_type, spline_object )
compute_spline_1D_periodic( data, spline_object )
compute_spline_1D_hermite ( data, spline_object )
delete( spline_object )
delete_spline_1D( spline_object )
interpolate_value( x, spline_object )
interpolate_array_values( a_in, a_out, np, spline_object )
```

`new_spline_1D()` is responsible for allocating all the necessary storage for the spline object and doing some partial initialization of the object. Hence the full creation of a spline is a three-step process: the first one is the declaration of the spline pointer, the second is its assignment through a call to the `new_spline_1D()` function. The last step, which fully initializes the spline by computing all the coefficients is carried out by calling `compute_spline_1D()`. This last call uses information inside the partially initialized spline object. For details, see the usage section below.

`set_slope_left()` and `set_slope_right()`: These subroutines are offered in case that the user wants to modify the values of the slopes in a spline object before a call to any of the `compute_spline_1D()` subroutines. These subroutines would extract the boundary types and slopes (if any) from the spline object, so in case that the user wants to change these values after the initialization step, these are the subroutines to use.

`compute_spline_1D_xxx()`: Note the presence of redundant subroutines. For instance, the subroutine `compute_spline_1D()`, which computes the spline coefficients for a given data, is just a wrapper around the other specialized subroutines that specify the type of boundary conditions in their names. We leave both options until we decide which interface is preferable. This involves a tradeoff between a uniform call that specifies the type of boundary condition through an argument, and a specific subroutine call, more efficient, but less uniform[1]. The `compute_spline_1D()` functions have to be called again any time that the underlying data changes or if the spline object is to be used with new data.

`interpolate_value()` returns the value of *f(x)* where x is a floating-point value between `xmin` and `xmax`, and f is a continuous function built with cubic B-splines and the user-defined boundary conditions contained in the spline object used to interpolate. Essentially, the spline object, together with the `interpolate_value()` function create the illusion of having available a continuous function when originally, only discrete data were available.

Finally, `interpolate_array_values()` has an analogous functionality than the previous interpolation function, but is capable of processing whole arrays. This is useful to avoid the overhead associated with a call to `interpolate_value()` inside a loop.

In the functions above:

num_points: Size of the array that contains the data that must be fitted with the spline. One should think of this as the full size of the array, that is, the user should not be concerned about whether the last point is to be taken into account in the periodic case or anything of the sort. Thus for a data array indexed $1 : NP$, the size is simply NP.

xmin: Lower bound of the grid in which the data array is defined. In other words, if we think of the data array (indexed 1:NP) as the values of a discrete function f defined over a sequence of x_i's, then $x_1 = xmin$.

xmax: Similarly to `xmin`, `xmax` represents the maximum extent of the grid. Following the same example of the data indexed $1 : NP$: $xmax = x_{NP}$.

bc_type: Descriptor of the type of boundary condition to be imposed in the spline. Presently, one of `PERIODIC_SPLINE` or `HERMITE_SPLINE`. These are really aliases to integer flags, but in Selalib we avoid the use of non-descriptive flags. The boundary conditions are enforced at the endpoints.

data: The double precision floating point data array to be fitted.

[1]Presently, both interfaces coexist. We may decide to ultimately leave them both. The same happens in the case of the destructor functions.

spline_object: The spline entity that is allocated and returned by the function `new_spline_1D()`, initialized by the `compute_spline()` functions and deleted by the `delete()` functions.

slope_L, slope_R: optional arguments, in the case of the hermitian boundary conditions, the user can define the value of the slope at either end of the domain. If not present, these values default to zero. As an example, a spline to fit the data representing the function $sin(x)$ between 0 and π would be called with slopes 1 and -1.

a_in: Input array to be interpolated.

a_out: Array that returns the interpolated values.

np: Number of points to be interpolated.

3.2.3 Usage

To use the module in a stand-alone way, include the line:

```
use sll_splines
```

The following example is an extract from the module's unit test.

```
program spline_tester
#include "sll_working_precision.h"
#include "sll_assert.h"
#include "sll_memory.h"
  use sll_splines
  use numeric_constants
  implicit none

#define NP 5000

  sll_int32 :: err
  sll_int32 :: i
  type(sll_spline_1d), pointer :: sp1
  type(sll_spline_1d), pointer :: sp2
  sll_real64, allocatable, dimension(:) :: data
  sll_real64 :: accumulator1, accumulator2
      sll_real64 :: val

  accumulator1 = 0.0_f64
  accumulator2 = 0.0_f64

  SLL_ALLOCATE(data(NP), err)

  print *, 'initialize data array'
```

```
    do i=1,NP
      data(i) = sin((i-1)*sll_pi/real(NP-1,f64))
    end do

    sp1 =>  new_spline_1D( NP,          &
                           0.0_f64,     &
                           sll_pi,      &
                           PERIODIC_SPLINE )
    call compute_spline_1D( data, PERIODIC_SPLINE, sp1 )
    sp2 =>  new_spline_1D( NP, 0.0_f64,     &
                           sll_pi,          &
                           HERMITE_SPLINE,
                           1.0_f64,
                          -1.0_f64 )
    call compute_spline_1D_hermite( data, sp2 )

    print *, 'cumulative errors at nodes: '
    do i=1, NP-1
       val = real(i-1,f64)*sll_pi/real(NP-1,f64)
       accumulator1 = accumulator1 + abs(data(i) - &
                      interpolate_value(val, sp1))
    end do

    print *, 'hermite case: '
    do i=1, NP
       val = real(i-1,f64)*sll_pi/real(NP-1,f64)
       accumulator2 = accumulator2 + abs(data(i) - &
                      interpolate_value(val, sp2))
    end do
    print *, 'Periodic case: '
    print *, 'average error at the nodes = '
    print *, accumulator1/real(NP,f64)
    call delete_spline_1D(sp1)
    if( accumulator1/real(NP,f64) < 1.0e-15 ) then
       print *, 'PASSED TEST'
    else
       print *, 'FAILED TEST'
    end if
    print *, '**************************** '
    print *, 'Hermite case: '
    print *, 'average error at the nodes = '
    print *, accumulator2/real(NP,f64)
    call delete_spline_1D(sp2)
    if( accumulator2/real(NP,f64) < 1.0e-15 ) then
       print *, 'PASSED TEST'
    else
```

```
71        print *, 'FAILED TEST'
72     end if
73   end program spline_tester
```

Here we do not go in detail over every line but only highlight those lines in which we interact with the splines module

Line 5: Imports the spline module. The intent is to eventually not require this but to import a single module, say 'selalib' which will include all modules itself. For now, this is the way to include these individual capabilities.

Lines 13 - 14: Declaration of the spline pointers.

Lines 29, 34: Allocation and partial initialization of the splines. Note the `=>` pointer assignment syntax.

Lines 33, 39: Calculation of the spline coefficients. After this call the spline object becomes fully usable. Note that in the example we used both existing interfaces to call the initialization subroutine.

Lines 45, 52: Value interpolation using the existing splines.

Lines 57, 67: Destruction of spline objects.

3.2.4 Status

Unit-tested.

3.3 Gauss-Legendre Integrator

3.3.1 Description

This is a low-level mathematical utility that applies the Gauss-Legendre method to compute numeric integrals. This module aims at providing a single interface to the process of integrating a function on a given interval.

3.3.2 Exposed Interface

To integrate the function `f(x)` (real-valued and of a single, real-valued argument `x`) over the interval $[a, b]$, the simplest way is through a function call such as:

```
gauss_legendre_integrate_1D(f, a, b, n)
```

In the function above, `n` represents the desired number of *Gauss* points used in the calculation:

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{k=1}^{n} w_k f(x_k) \tag{3.1}$$

Presently, the implementation accepts values of `degree` between 2 and 10 inclusively. The function `gauss_legendre_integrate_1D` internally does the proper scaling of the points to adjust the integral over the desired interval.

The function `gauss_legendre_integrate_1D` is a generic interface and as such, it hides some alternative integrators, which are selected depending on the type of the passed arguments. For instance, we have available the function

```
gauss_legendre_integrate_interpolated_1D(f, spline, a, b, n)
```

which integrates a function represented by a spline object. The function `f` in this case is the spline interpolation function. It looks like this interface could be simplified and we could eliminate the first parameter and pass only the spline object. The only reason to leave the interpolation function as an argument is if we find some compelling reason to parametrize the interpolation function as well.

It will be necessary to implement other integrators for functions with a different signature, such as two- or three-parameter functions. It might also be necessary to distinguish between one-dimensional and two- or 3-dimensional integration. While this is not yet implemented, here we lay out some suggestions on how to proceed in such cases.

For the class of integrals that are done in one-dimension, such as the above, the cleanest but somewhat more laborious approach appears to be to write a different integrator for every function signature that is needed. For instance, one may need to write specialized integrators for $f(x_1, x_2)$, $f(x_1, x_2, x_3)$ and so on, with the convention that the integral is carried out over, say, the first of the variables. The variables which are not integrated can be used as parameters. The alternative to this approach could be to write a single integrator that is able to receive multiple parameters, through the use of arrays or derived types, but the ugliness of this approach, and the need to basically write glue-code (pack/unpack the arrays or derived types with variables and parameters) every time one wants to integrate something are reasons to reject this approach.

3.3.3 Usage

As mentioned above, the name of the generic function that hides the specialized functions is `gauss_legendre_integrate_1D`. The specialized functions can be individually called to avoid the overhead of the generic function call if desired. A one-dimensional function (user or Fortran) can be integrated by a call like:

```
gauss_legendre_integral_1D( test_function, &
                            0.0_f64,       &
                            sll_pi/2.0,    &
                            4 )
```

A function that is represented by an underlying spline object can be called like:

```
gauss_legendre_integral_interpolated_1D( interpolate_value,&
                                         sp1,              &
```

```
                                                 0.0_f64,          &
                                                 sll_pi,           &
                                                 4)
```

where sp1 is a spline object. It should be decided if this last case is indeed that interface that is wished, or if something more simplified should be implemented intstead.

3.3.4 Status

Unit-tested.

3.4 FFT

3.4.1 Description

The FFT module intends to provide an unified interface to native or external library FFT functions. This module plays a role analogous to the *Collective* module, explained below, but in this case applied to FFT capabilities instead of a parallelization library like MPI. In this sense, the `sll_fft` module provides a set of types, functions and subroutines that can in principle be implemented with a choice of external libraries.

3.4.2 Exposed Interface

The interface to the FFT functions is inspired by the FFTW interface in which the FFT operation is carried out by the creation of a plan followed by the execution of this plan. The "plan" stores all relevant information (twiddle factor arrays, auxiliary arrays, etc.) that may be needed by a given type of FFT operation. The application of the plan executes the operation itself on a given data. Like all other native types in Selalib, the FFT plan is declared through a pointer, which must be allocated and initialized. To declare, call:

```
type(sll_fft_plan), pointer :: fft_plan
```

Followed by an initialization step:

```
fft_plan => sll_new_fft( sample_number,
                         data_type,
                         fft_flags )
```

Where the `data_type` parameter can take either of the values: `FFT_REAL` or `FFT_COMPLEX`. Presently we only provide double precision transforms (i.e.: 64-bit floating precision numbers). The `fft_flags` can take the values: `FFT_NORMALIZE_FORWARD` and/or `FFT_NORMALIZE_INVERSE`. If no flags are present, an unnormalized FFT will be executed. The user may request that the FFT be normalized in both directions by combining the flags with a + sign.

To execute the FFT plan:

sample_number:

spline: A pointer to the spline object to be filled or updated.

3.4.3 Usage

3.4.4 Status

3.5 Collective Communications

3.5.1 Description

Selalib applies the principle of modularization throughout all levels of abstraction of the library and aims at keeping third-party library modules as what they are: separate library modules. Therefore, in its current design, even a library like MPI has a single point of entry to Selalib. The collective communications module is such point of entry. We focus thus on the functionality offered by MPI, assign wrappers to its most desirable functionalities and write wrappers around them. These are the functions that are actually used throughout the program. This allows to adjust the exposed interfaces, do additional error-checking and would even permit to completely change the means to parallelize a code, by being able to replace MPI in a single file if this were ever needed.

3.5.2 Exposed Interface

Fundamental type:

```
sll_collective_t
```

Constructors, destructors and access functions:

```
sll_new_collective( parent_col )
sll_delete_collective( col )
```

When the Selalib environment is activated, there exists, in exact analogy with `MPI_COMM_WORLD`, a global named `sll_world_collective`. At the beginning of a program execution, this is the only collective in existence. Further collectives can be created down the road. The above functions are responsible for the creation and destruction of such collectives. The following functions are used to access the values that a particular collective knows about.

```
sll_get_collective_rank( col )
sll_get_collective_size( col )
sll_get_collective_color( col )
sll_get_collective_comm( col )
```

Since the wrapped library requires initialization, so does `sll_collective`. To start and end the parallel environment, the user needs to call the functions:

```
sll_boot_collective( )
sll_halt_collective( )
```

These functions would not be exposed at the top level, and would be hidden by a further call to something akin to `boot_selalib` and `halt_selalib`. Finally, the wrappers around the standard `MPI` capabilities are presently exposed through the following generic functions:

```
sll_collective_bcast( col, buffer, size, root )
sll_collective_gather( col, send_buf, send_sz, root,
                       rec_buf )
sll_collective_gatherv( col, send_buf, send_sz, recvcnts,
                        displs, root, recv_buf )
sll_collective_allgatherv( col, send_buf, send_sz, sizes,
                           displs, rec_buf )
sll_collective_scatter( col, send_buf, size, root,
                        rec_buf )
sll_collective_scatterv( col, send_buf, sizes, displs,
                         rec_szs, root, rec_buf )
sll_collective_all_reduce( col, send_buf, count, op,
                           rec_buf )
```

which presently stand for specialized versions that operate on specific types. For instance:

```
sll_collective_all_to_allv_real( send_buf,
                                 send_cnts,
                                 send_displs,
                                 recv_buf,
                                 recv_cnts,
                                 recv_displs,
                                 col )
```

3.5.3 Usage

To use the module as stand-alone, include the line:

```
use sll_collective
```

Any use of the module's functionalities must be preceeded by calling

```
call sll_boot_collective()
```

and to "turn off" the parallel capabilities, one should finish by a call to:

```
call sll_halt_collective()
```

This *booting* of the parallel environment needs to be done only once in a program.

Some more specific examples are needed here...

3.5.4 Status

Several core functionalities tested, but no comprehensive unit test done yet

3.6 Remapper

3.6.1 Description

Written on top of `sll_collective`, the remapper is a powerful facility that is capable of rearranging data in flexible and convenient ways in a parallel machine. It is meant to be a generalization of the 'transposition', which users/developers of *CALVI* team codes know and love. The main difference is is generality, as here we extend the idea to encompass something beyond a data transposition in 2D, to an operation that can be carried out in any number of dimensions. For instance, suppose that you start with a multidimensional array that has been domain decomposed and distributed among N_p processors. The layout of the data (that is, the description of what ranges of the data are contained in each processor) is specified by an instance of the type `layout_XD_t`, (where `X` is the dimension of the data). The layout contains a notion of an N_p-sized collection of boxes, each box representing a contiguous chunk of the multidimensional array stored in each node. If in the course of a computation, you wish to reconfigure the layout of the data (for example, if you wished to re-arrange data in a way that would permit launching serial algorithms locally in each node), then you would create and initialize a new layout descriptor with the target configuration (i.e.: you to define the box to be stored in each node). This is a conceptually simple but perhaps slightly verbose task. Then a call to the appropriate choice among:

```
NEW_REMAPPER_PLAN_3D( initial_layout,
                      target_layout,
                      data_size_in_integer_sizes )
NEW_REMAPPER_PLAN_4D( initial_layout,
                      target_layout,
                      data_size_in_integer_sizes )
NEW_REMAPPER_PLAN_5D( initial_layout,
                      target_layout,
                      data_size_in_integer_sizes )
```

will yield an instance of the type `remap_plan_3D_t`, or `remap_plan_4D_t` or `remap_plan_5D` , respectively, that will contain all the information necessary to actually carry out the data re-distribution. Finally, a call to

```
apply_remap_3D( plan, data_in, data_out )
apply_remap_4D( plan, data_in, data_out )
apply_remap_5D( plan, data_in, data_out )
```

will actually redistribute `data` (as an out-of-place operation) according to `plan` in an optimized way[2].

To appreciate the power of such facility, note that in principle, the construction of a (communications latency-limited) parallel quasi-neutral solver can be based exclusively on remapping operations. This is an important tool in any problem that would require global rearrangements of data. The remapper thus is able to present a single powerful abstraction that is general, reusable and completely hides most of the complications introduced by the data distribution.

3.6.2 Exposed Interface

The remapper offers the following descriptor types for parallel data layout, differing from one another only in the dimensionality of the data described:

```
layout_3D_t
layout_4D_t
layout_5D_t
```

(Note that for the remapper, we have forgone the use of the `sll_` prefix. This is as an example of the likely policy that the low- and mid-level reusable utilities should not be prefixed, thus being instantly available for any other development. Eventually a decision needs to be made and the choice implemented uniformly throughout the library.) These types are each accompanied by their own constructors, destructors and accessors. Specifically, the constructors are:

```
new_layout_3D( collective )
new_layout_4D( collective )
new_layout_5D( collective )
```

Note that each layout descriptor needs to be allocated by providing an instance of `sll_collective_t`. This can be understood by thinking of the data layout as being associated with a given group of processors (the collective) and a specification of the data boxes contained in each one. After calling any of the `new_layout` functions, the returned instance becomes associated to the given collective and enough memory is allocated (size of the collective) to hold the boxes specification.

The destructors are:

```
delete_layout_3D( layout )
delete_layout_4D( layout )
delete_layout_5D( layout )
```

[2]This is a very loaded comment. Some of the optimizations are carried out by the remapper, like the identification of the minimally-sized communicators to launch the exchanges, or the selection of the lower-level communications functions (alltoall vs. alltoallv, for instance). Other optimizations would need to be triggered externally, by passing proper compilation flags to the MPI facilities. This would be problem-dependent.

The access functions for the `layout` types are are always prefixed with the corresponding `get_layout_XD`/`set_layout_XD` (where the 'X' denotes the dimensionality of the data), and they presuppose knowledge of the convention for ordering the indices as in `i, j, k, l, m`, for the dimensions. Specifically, to get/set values inside the `layout` types we have available for 3D layouts:

```
get_layout_3D_num_nodes( layout )
get_layout_3D_box( layout, rank )

get_layout_3D_i_min( layout, rank )
get_layout_3D_i_max( layout, rank )
get_layout_3D_j_min( layout, rank )
get_layout_3D_j_max( layout, rank )
get_layout_3D_k_min( layout, rank )
get_layout_3D_k_max( layout, rank )

set_layout_3D_i_min( layout, rank, val )
set_layout_3D_i_max( layout, rank, val )
set_layout_3D_j_min( layout, rank, val )
set_layout_3D_j_max( layout, rank, val )
set_layout_3D_k_min( layout, rank, val )
set_layout_3D_k_max( layout, rank, val )
```

As a very inelegant convenience, the layout type allows direct access to its collective reference. For 4D layouts:

```
get_layout_4D_num_nodes( layout )
get_layout_4D_box( layout, rank )

get_layout_4D_i_min( layout, rank )
get_layout_4D_i_max( layout, rank )
get_layout_4D_j_min( layout, rank )
get_layout_4D_j_max( layout, rank )
get_layout_4D_k_min( layout, rank )
get_layout_4D_k_max( layout, rank )
get_layout_4D_l_min( layout, rank )
get_layout_4D_l_max( layout, rank )

set_layout_4D_i_min( layout, rank, val )
set_layout_4D_i_max( layout, rank, val )
set_layout_4D_j_min( layout, rank, val )
set_layout_4D_j_max( layout, rank, val )
set_layout_4D_k_min( layout, rank, val )
set_layout_4D_k_max( layout, rank, val )
set_layout_4D_l_min( layout, rank, val )
set_layout_4D_l_max( layout, rank, val )
```

And for 5D layouts:

```
get_layout_5D_num_nodes( layout )
get_layout_5D_box( layout, rank )

get_layout_5D_i_min( layout, rank )
get_layout_5D_i_max( layout, rank )
get_layout_5D_j_min( layout, rank )
get_layout_5D_j_max( layout, rank )
get_layout_5D_k_min( layout, rank )
get_layout_5D_k_max( layout, rank )
get_layout_5D_l_min( layout, rank )
get_layout_5D_l_max( layout, rank )
get_layout_5D_m_min( layout, rank )
get_layout_5D_m_max( layout, rank )

set_layout_5D_i_min( layout, rank, val )
set_layout_5D_i_max( layout, rank, val )
set_layout_5D_j_min( layout, rank, val )
set_layout_5D_j_max( layout, rank, val )
set_layout_5D_k_min( layout, rank, val )
set_layout_5D_k_max( layout, rank, val )
set_layout_5D_l_min( layout, rank, val )
set_layout_5D_l_max( layout, rank, val )
set_layout_5D_m_min( layout, rank, val )
set_layout_5D_m_max( layout, rank, val )
```

The above functions define the interface that will allow you to declare and initialize the `layout` types as desired. This is where the work lies when using this module. Note that all the above functions could be coalesced into a set of functions of the type `set_layout_X_XXX(layout, rank, val)` if we choose to hide all the above functions behind a generic interface. The selection would be done automatically depending on the type of layout passed as an argument.

The type `remap_plan` exists also in multiple flavors, depending on the dimensionality of the data to be remapped:

```
remap_plan_3D_t
remap_plan_4D_t
remap_plan_5D_t
```

The `remap_plan_t` type stores the locations of the memory buffers that will be involved in the communications, the specification of the data that will be sent and received, as well as the collective within which the communications will take place. There are, however, declaration functions available. The choice depends on the dimensionality of the data:

```
NEW_REMAPPER_PLAN_3D( initial_layout,
                          final_layout,
                          array_name )
     NEW_REMAPPER_PLAN_4D( initial_layout,
                          final_layout,
                          array_name )
     NEW_REMAPPER_PLAN_5D( initial_layout,
                          final_layout,
                          array_name )
```

Finally, the way to execute the plan on a particular data set is through a call of the appropriate subroutine (here presented as generic interfaces)

```
     apply_remap_3D( plan, data_in, data_out )
     apply_remap_4D( plan, data_in, data_out )
     apply_remap_5D( plan, data_in, data_out )
```

3.6.3 Usage

For use in stand-alone way, use the line:

```
#include "sll_remap.h"
```

While verbose, the best way to demonstrate the usage of the remapper is with a complete program. Below it, we examine the different statements.

```
1   program remap_test
2     use sll_collective
3   #include "sll_remap.h"
4   #include "sll_memory.h"
5   #include "sll_working_precision.h"
6   #include "misc_utils.h"
7     implicit none
8
9     ! Test of the 3D remapper takes a 3D array whose global
10    ! size Nx*Ny*Nz, distributed among pi*pj*pk processors.
11    integer, dimension(:,:,:), allocatable :: a3
12    integer, dimension(:,:,:), allocatable :: b3
13
14    ! Take a 3D array of dimensions 8X8X1
15    integer, parameter                 :: total_sz_i = 8
16    integer, parameter                 :: total_sz_j = 8
17    integer, parameter                 :: total_sz_k = 1
18
19    ! the process mesh
20    integer, parameter                 :: pi = 4
21    integer, parameter                 :: pj = 4
22    integer, parameter                 :: pk = 1
```

```

! Split into 16 processes, each with a local chunk 2X2X1
integer                            :: local_sz_i
integer                            :: local_sz_j
integer                            :: local_sz_k
integer                            :: ierr
integer                            :: myrank
integer                            :: colsz
integer                            :: i,j,k
integer                            :: i_min, i_max
integer                            :: j_min, j_max
integer                            :: k_min, k_max
integer                            :: node
integer, dimension(1:3)            :: gcoords

! Remap variables
type(layout_3D_t), pointer         :: conf3_init
type(layout_3D_t), pointer         :: conf3_final
type(remap_plan_3D_t), pointer     :: rmp3

! Boot parallel layer
call sll_boot_collective()

! Initialize and allocate the variables.
local_sz_i = total_sz_i/pi
local_sz_j = total_sz_j/pj
local_sz_k = total_sz_k/pk
SLL_ALLOCATE(a3(1:local_sz_i,1:local_sz_j,1:local_sz_k), ierr)
SLL_ALLOCATE(b3(1:local_sz_i,1:local_sz_j,1:local_sz_k), ierr)
myrank    = sll_get_collective_rank(sll_world_collective)
colsz     = sll_get_collective_size(sll_world_collective)

conf3_init     => new_layout_3D( sll_world_collective )
conf3_final    => new_layout_3D( sll_world_collective )
random_layout1 => new_layout_3D( sll_world_collective )

! Initialize the layout
do k=0, pk-1
   do j=0, pj-1
      do i=0, pi-1
         node = i+pi*(j+pj*k) ! linear index of node
         i_min = i*local_sz_i + 1
         i_max = i*local_sz_i + local_sz_i
         j_min = j*local_sz_j + 1
         j_max = j*local_sz_j + local_sz_j
         k_min = k*local_sz_k + 1
```

```
            k_max = k*local_sz_k + local_sz_k
            call set_layout_i_min( conf3_init, node, i_min )
            call set_layout_i_max( conf3_init, node, i_max )
            call set_layout_j_min( conf3_init, node, j_min )
            call set_layout_j_max( conf3_init, node, j_max )
            call set_layout_k_min( conf3_init, node, k_min )
            call set_layout_k_max( conf3_init, node, k_max )
         end do
      end do
   end do

   ! Initialize the data using layout information.
   do k=1, local_sz_k
      do j=1, local_sz_j
         do i=1, local_sz_i
            gcoords= local_to_global_3D(conf3_init,(/i,j,k/))
            a3(i,j,k) = gcoords(1) + &
                 total_sz_i*((gcoords(2)-1) + &
                 total_sz_j*(gcoords(3)-1))
         end do
      end do
   end do

   ! Initialize the final layout, in this case, just a
   ! transposition
   do k=0, pk-1
      do j=0, pj-1
         do i=0, pi-1
            node = i+pi*(j+pj*k) ! linear index of node
            i_min = i*local_sz_i + 1
            i_max = i*local_sz_i + local_sz_i
            j_min = j*local_sz_j + 1
            j_max = j*local_sz_j + local_sz_j
            k_min = k*local_sz_k + 1
            k_max = k*local_sz_k + local_sz_k
            call set_layout_i_min( conf3_final, node, j_min )
            call set_layout_i_max( conf3_final, node, j_max )
            call set_layout_j_min( conf3_final, node, i_min )
            call set_layout_j_max( conf3_final, node, i_max )
            call set_layout_k_min( conf3_final, node, k_min )
            call set_layout_k_max( conf3_final, node, k_max )
         end do
      end do
   end do

   rmp3 => NEW_REMAPPER_PLAN_3D( conf3_init, conf3_final, a3 )
```

```
115   call apply_remap_3D( rmp3, a3, b3 )
116
117   ! At this moment, b3 contains the expected output
118   ! from the remap operation.
119
120   ! Delete the layouts
121   call delete_layout_3D( conf3_init )
122   call delete_layout_3D( conf3_final )
123
124    call sll_halt_collective()
125
126  end program remap_test
```

Lines 1 - 5: Required preamble at the time of this writing. Eventually this will be replaced by a single statement to include the whole library. Presently, we include various headers individually, so bear in mind that this is not the way this will end up being. Line 3 specifically loads the remapper facility. Here it is brought as a header file as the NEW_REMAPPER_PLAN_XD() is implemented as a macro.

Lines 9 - 12: For this example we allocate two 3D arrays for the input and output of the remap operation.

Lines 14 - 22: Definition of the array size from a global perspective. In other words, the array to be remapped is a $8 * 8 * 1$ array, to be distributed on a processor mesh of dimensions $4 * 4 * 1$.

Lines 24 - 36 Miscellaneous integer variables that we will use.

Lines 38 - 41 Pointers to the initial and final layouts and the remap plan.

Line 44 Presently we boot from collective. Eventually this will be replaced by a call to something like `boot_selalib()` or something similar, where we declare and initialize anything we need in a single call.

Lines 46 - 57 Initialization of the variables.

Lines 59 - 78 This is where the actual work is when using the remapper. We need to initialize a layout, in this case the initial configuration. We use the access functions `set_layout_x_xxx( )` to populate the fields. Here we obviously take into account the geometry of our 'process mesh' to find out the rank of the process that we are initializing.

Lines 80 - 90 We need to initialize the data, here we choose simply to assign the index of the array, considered as a 1D array. Note the use of the helper function `local_to_global_3D( layout, triplet )`. We exploit the knowledge of the global layout of the data to find out the global indices of a local 3-tuple.

Lines 92 - 112 The other main part of the work, the initialization of the target layout. In this case, we chose a simple transposition, which is achieved by switching `i` and `j`.

Lines 114 - 115 Here we allocate and initialize the remap plan, using the initial and final configurations as input. The third argument is passed to inform the remapper of the type of data to be passed. The call to `apply_remap_3D()` is a call to a generic function, hence, a type-dependent sub-function must have been defined to be able to successfully make this call. At the time of this writing, only single precision integers and double precision floats have been implemented.

Line 116 Here we apply the plan. This function is type-dependent due to the input/output arrays. Please refer to the implementation notes for some commentary on our options with this interface.

Lines 122 - 125 Cleanup. The layouts need to be deleted to prevent memory leaks.

3.6.4 Implementation Notes

The biggest challenge with the remapper is to attain a desired level of genericity and to preserve the modularity of the library. These two problems are intimately related. Ideally, we should be able to apply a remap operation on data of any type, including user-derived types. Another requirement has been to confine a library like MPI to a single entry point into Selalib. This means that we do not want the MPI derived types to pollute the higher abstraction levels of the library: especially at the top level, we want to express our programs with the capabilities of the Fortran language alone.

These requirements were solved in the prototype version of the remapper through the use of a single datatype to represent all other types of data at the moment of assembling the exchange buffers and launching the MPI calls. In our case, we have chosen to represent all data as 'integers'. This means that the exchange buffers that are stored in the remap plans are integer arrays. Thus, the design decision in the prototype has been to choose flexibility and ease of change over execution speed. In contrast with the C language, the constant call to the `transfer()` function to store and retrieve data from the exchange buffers carries with it a possibly significant execution time penalty.

The function `NEW_REMAPPER_PLAN_XD()` is by nature type-independent, as the design of the plan only depends on the layouts. However, it is also convenient to store the send/receive buffers in the plan, and the allocation of these buffers requires knowledge of the amount of memory required. This information is passed in the third argument. The macro will internally select an element of this array and determine its size in terms of the fundamental datatype being exchanged (i.e.: `integer`). This way we now how much memory to allocate in the buffers.

Another means to achieve the illusion of genericity are Fortran's built-in features in this regard. For example, we can have specialized `apply_remap_3D()` functions for the most commonly used datatypes, all hidden behind the same generic name. These specialized functions would not depend on the current choice of using a single type for the exchange buffers, eliminating any penalty that we are definitively paying at present, with the calls to the `transfer()` intrinsic function. This solution would mean writing redundant code, something that could be addressed with preprocessor macros, but this would not be a solution for eliminating the penalizations of the `transfer()` intrinsic when we are exchanging derived types. A solution that can exchange these arbitrary data while not requiring the use of the MPI derived types at the higher levels is yet to be found. It could be that the Fortran way to solve this problem would be to accept the invasion of MPI at the higher levels...

3.6.5 Status

In testing.

Chapter 4

Top-Level Layer: Semi-Lagrangian Toolbox

4.1 Logical Meshes

4.1.1 Description

The types of mathematical problems which we aim at studying are usually most easily solved on regular cartesian grids. Because of reasons which will become clear when we discuss the coordinate transformations module, in Selalib we follow the convention of calling the simplest cartesian grids by the name *logical meshes*. We use the terms *grid* and *mesh* interchangeably in this text. In the code we use exclusively the term *mesh*. Also by our convention, these grids will represent a uniform meshing of a multidimensional domain on the logical coordinates $\eta_1, \eta_2, ..., \eta_n$.

4.2 Meshes

4.2.1 Description

The mesh types aim at storing array data, but including additional information needed to put this data in context. There are several types of meshes:

cartesian mesh : an N-D fully regular mesh defined by an N-D data array and additional parameters that define the domain (i.e.: xmin, xmax) and other parameters like the number of cells in each dimension and the spacing of the data (i.e.: delta). The services provided should be like (with corresponding name changes, indicating the dimensionality of the data):

- allocation(new) and initialization functions,
- deletion,

  - a copy constructor,
  - computation of data interpolants (cubic splines for example),
  - `get_value(mesh,x1,x2,...)`: where `x1`, `x2`,... belong to the corresponding interval `(x1_min, x1_max)`, `(x2_min, x2_max)`, etc., and which returns the interpolated value at the desired point. This operation launches spline interpolations under the hood.
  - `get_node_value(mesh,i,j,...)`: analogous to `get_value()` but does not need to launch any interpolation as the indices are integers and the requested value falls on a mesh node. This is implemented by a macro.
  - `set_node_value(mesh,i,j,...,val)`: sets the node datum at i,j,... with value. Implemented with a macro.

There are some pitfalls with the suggested interface:

- the naming convention could get a little complicated depending on the type of data stored in the array (scalar, multiple-valued, etc.), as we had originally intended with the field/vec naming convention.

- The interface may need to directly expose its underlying data, or pointers to sections of it for certain operations, like FFT. The whole data field of a mesh could need to be set to a whole data array in one step, during the initialization, such as after a remap operation. At least in these cases, the access to the data might be safer given the nature of the operations.

structured mesh : a structured mesh is a mapped cartesian mesh in which the coordinates are decoupled. An ND mesh data is stored as an ND array. In addition, however, we need N 1D arrays to store the actual coordinates of the node locations in each dimension. As an alternative, we could use N functions $x_1 = f(\eta_1)$, $x_2 = f(\eta_2)$, ... , $x_n = f(\eta_n)$, to represent each transformation. We follow the convention that η_1, η_2, etc. are values in the $[0, 1]$. The offered services ought to be:

  - allocation, initialization, deletion and copy.
  - `get_node_value(mesh, i, j, ... )`: which reads directly from the data array.
  - `set_node_value(mesh, i, j, ... )`: which writes directly to the data array.
  - In an analogous way to the `get_value()` functions described above, we could provide something like `get_value(mesh,` η_1`,` η_2`)`. This would also trigger an interpolation step using uniform splines generated with the ND data. This would require the user to be *thinking* in terms of the logical variables η_i.

- Similar functions could be provided such that the user could also request values at points x_i. For this, we could launch non-uniform splines, with the spacing determined by the x_i arrays and the ND data.
- This type may also need to grant direct access to the data array for use in operations like FFT and similar.

tensor product mesh : This is a mapped cartesian mesh in which the coordinates are coupled. That is, the mappings have the form: $x_1 = f(\eta_1, \eta_2, ..., \eta_n)$, $x_2 = f(\eta_1, \eta_2, ..., \eta_n)$, ... , $x_n = f(\eta_1, \eta_2, ..., \eta_n)$. The specification of this type of mesh requires an ND data array, N ND arrays that specify the transformations numerically, or N functions of arity N to specify the transformations. The services provided should be:

- allocation, initialization, deletion and copy.
- `get_node_value(mesh, i, j, ... )`: which reads directly from the data array.
- `set_node_value(mesh, i, j, ... )`: which writes directly to the data array.
- `get_value(mesh,` η_1`,` η_2`, ... )` which would also trigger uniform spline interpolations, and
- `get_value(mesh, x1, x2, ... )`, which would trigger nonuniform spline interpolations.

We may need to include other operations here which would entail inverse mappings (maybe also with splines), or NURBS or something else. Need to fill in the details more here.

4.2.2 Exposed Interface

4.2.3 Usage

4.2.4 Status

4.3 Scalar Fields

4.3.1 Description

The physical quantities of interest are normally defined in a physical space. For instance, the electric potential $\phi(\vec{x})$ is specified by a function on the $\vec{x}$ variables (x, y, z). Even in simple problems the borders of the physical domain may have relatively complicated shapes which make the solution of differential equations more difficult. Great flexibility (possibly at the expense of efficiency) can be gained by transferring the data from the physical space to a representation in a logical space in which the underlying grid that we use for the numerical solution is always uniform. In such uniform grid, the borders have a much simpler (straight line) representation that may aid the solution of the equations. This means that it is convenient to permit the alternative representation of the data on variables on a physical space ($\vec{x}$) or on a logical space ($\vec{\eta}$). For these reasons, the scalar field should be thought of data plus a coordinate transformation.

(place picture of an example coordinate transformation here.)

Consider a scalar field and the services that it should offer to permit us to use it in our logical grid:

1. The scalar field is a derived type.
2. The type contains a simple array to store the data, i.e.: the values of the fields on a collection of points. Both, logical mesh values or physical mesh values can be stored in the same array.
3. The type contains an object that fully specifies the associated coordinate transformation (a mapped mesh). The transformation should provide all the needed services to permit moving the representation of the data from the physical space to the logical space and vice-versa:
   (a) something
   (b) something else

4.4 Quasi-Neutral Equation Solver

4.4.1 Description

Here we present a simplified but hopefully reasonably clear step-by-step derivation of the quasi-neutral equation (the gyrokinetic Poisson equation) which this module solves. The intent is to make clear some of the assumptions that are built into this model. We follow the general argument given by Krommes (cite reference here for "Nonlinear gyrokinetics: a powerful tool for the description of microturbulence in magnetized plasmas").

The model is based on the idea that the fast gyrations of particles around the magnetic field lines can be averaged away while still preserving the most important long-term physics (hence the name of this type of approach: gyrokinetics). The simplest model that we will look into first deals directly in the distribution function of the gyrocenters, instead of the particles'.

We start with the Poisson equation:

$$-\nabla^2\phi = \frac{1}{\epsilon_0}\rho, \tag{4.1}$$

where ϕ is the electric potential, ρ is the volumetric charge density and ϵ_0 is the permittivity of free space. The main assumptions built into the model are introduced through the treatment of ρ. We decompose the charge density in its two main constituents: the contribution by the ions (ρ_i) and the electrons' (ρ_e). As mentioned earlier, the main idea behind gyrokinetics is the averaging of the fast particle motions around the field lines, hence some of the effects that occur on longer time-scales are explicitly introduced into the model, for example by particle drifts (e.g.: $\vec{E} \times \vec{B}$, polarization drift, etc.). In doing so, we allow ourselves to further separate the charge density into a polarization charge and a charge at the particles' gyrocenters (gyrocenters do not polarize). Thus, Poisson's equation becomes

$$-\nabla^2 \phi = \frac{1}{\epsilon_0}(\rho_i^G + \rho_i^{pol} - \rho_e^G - \rho_e^{pol}), \tag{4.2}$$

where the indices G, pol, i and e indicate *gyrocenters*, *polarization*, *ions* and *electrons* respectively. The polarization drift velocity, whose derivation can be found in introductory plasma physics texts is given by

$$\vec{v}^{pol} = \pm \frac{1}{\omega_{ci} B} \frac{\partial \vec{E}_\perp}{\partial t}. \tag{4.3}$$

Here ω_{ci} is the ion cyclotron frequency, B is the magnitude of the local magnetic field and $\vec{E}_\perp$ is the electric field perpendicular to the magnetic field. The plus/minus sign in the equation applies to positive and negative particles respectively. Heuristically, the polarization charge obeys a continuity equation:

$$\frac{\partial \rho^{pol}}{\partial t} = -\nabla \cdot \vec{j}^{pol} = -\nabla \cdot (nZe\vec{v}^{pol}) = -\nabla \cdot \left(nZe \frac{1}{\omega_{ci} B} \frac{\partial \vec{E}_\perp}{\partial t} \right). \tag{4.4}$$

Here, Z is the charge state of the ions under consideration (obviously 1 for a hydrogen-burning fusion plasma), e is the electron charge and n is the particle density per unit volume. As long as none of the factors of the time derivative depends itself on time, we can integrate immediately and arrive at an expression for ρ^{pol}:

$$\rho^{pol}(\vec{x}) = \nabla \cdot \left(\frac{n_i(\vec{x}) Ze}{\omega_{ci}(\vec{x}) B(\vec{x})} \nabla_\perp \phi(\vec{x}, t) \right). \tag{4.5}$$

Here we have introduced the assumption of n_i, the ion particle density, being independent of time. In equation (4.5), we can consider that the differential operators act only in the directions perpendicular to the magnetic field, as these are the only terms that will survive the taking of the divergence. Finally, by multiplying by the proper unit factors and using the relation

$$\omega_p = \left(\frac{ne^2}{\epsilon_0 m} \right)^{\frac{1}{2}} \tag{4.6}$$

we can recast equation (4.5) into

$$\rho^{pol}(\vec{x}) = \epsilon_0 \nabla_\perp \cdot \left(\varepsilon^G(\vec{x}) \nabla_\perp \phi(\vec{x}, t) \right). \tag{4.7}$$

where

$$\varepsilon^G(\vec{x}) \equiv \frac{\omega_{pi}^2(\vec{x})}{\omega_{ci}^2(\vec{x})} \tag{4.8}$$

is called the *dielectric constant of the gyrokinetic vacuum.* With equation (4.7), and neglecting the polarization of the electrons, the modified Poisson equation (4.2) can be written as:

$$-\nabla^2\phi(\vec{x},t) - \nabla_\perp \cdot \big(\varepsilon^G(\vec{x},t)\nabla_\perp\phi(\vec{x},t)\big) = \frac{1}{\epsilon_0}(\rho_i^G - \rho_e^G). \tag{4.9}$$

In fusion applications, $\varepsilon^G >> 1$ thus we neglect the ordinary laplacian term on the left-hand side.

The charge density of the electrons also receives a special treatment. The basic assumption here is that the electrons can move very quickly along a magnetic field line and thus are able to rapidly react to electric potential variations through changes in the electron particle density. Thus, it is assumed that along a field line, the electrons obey the Boltzmann relation:

$$n_e(\vec{x},t) = \bar{n}_e(\vec{x},0)\exp\left(\frac{e}{k_BT_e}(\phi(\vec{x},t) - <\phi(\vec{x},t)>_\ell)\right). \tag{4.10}$$

In equation (4.10), $n_e(\vec{x},t)$ is the instantaneous electron particle density, $\bar{n}_e(\vec{x})$ is the average electron density along the magnetic field line, k_B is Boltzmann's constant, T_e is the electron temperature and the $<\cdot>_\ell$ average is taken along the magnetic field line. By making yet another assumption of very small deviations from the average electric potential, the previous equation can be linearized:

$$n_e(\vec{x},t) = \bar{n}_e(\vec{x},0)\bigg(1 + \frac{e}{k_BT_e}(\phi(\vec{x},t) - <\phi(\vec{x},t)>_\ell)\bigg). \tag{4.11}$$

At the risk of being too sloppy, we will equate the electron particle density with the electron gyrocenter density. A more careful step would involve gyroaveraging directly the original electron distribution function. With this assumption, our modified poisson becomes:

$$-\nabla_\perp \cdot \big(\varepsilon^G(\vec{x},t)\nabla_\perp\phi(\vec{x},t)\big) = \frac{1}{\epsilon_0}\left(\rho_i^G - \bar{\rho}_{e0}^G\Big(1 + \frac{e}{k_BT_e}(\phi(\vec{x},t) - <\phi(\vec{x},t)>_\ell)\Big)\right). \tag{4.12}$$

Rearranging terms and using the relation:

$$\lambda_D = \left(\frac{\epsilon_0 k_B T_e}{ne^2}\right)^{\frac{1}{2}} = \left(\frac{\epsilon_0 k_B T_e}{\rho e}\right)^{\frac{1}{2}}, \tag{4.13}$$

we arrive at:

$$-\nabla_\perp\cdot\big(\varepsilon^G(\vec{x},t)\nabla_\perp\phi(\vec{x},t)\big) + \frac{1}{\lambda_{D0}}(\phi(\vec{x},t) - <\phi(\vec{x},t)>_\ell) = \frac{1}{\epsilon_0}\Big(\rho_i^G - \bar{\rho}_{e0}^G\Big). \tag{4.14}$$

But for a normalization of the variables, equation (4.14) is virtually the same as the one stated in the report by Latu (list here reference for "Scalable Quasineutral solver for gyrokinetic simultion"). Equation (4.14) is not yet ready to solve due to the presence of the $< \phi(\vec{x},t) >$ term. We need to average the solution we seek, after all. To obtain an additional equation that will help us find this term, we take the average of both sides of the equation in the same sense that we have been averaging before: along a magnetic field line (note that in practice, due to ergodicity, this line may extend and fill a surface or even worse.) The barred quantities have already been averaged so we can get them out of the averaging operator when necessary. Averages of averages remain unchanged and thus we arrive at:

$$-\nabla_\perp \cdot \big(< \varepsilon^G(\vec{x},t)\nabla_\perp\phi(\vec{x},t) >_\ell \big) = \frac{1}{\epsilon_0}(< \rho_i^G >_\ell - < \bar{\rho}_{e0}^G >_\ell). \qquad (4.15)$$

By making two final assumptions: that the quantities involved in the calculation of $\varepsilon^G(\vec{x},t)$ are not time-dependent but given by the initial ion distribution, and that this quantity does not vary along the domain of the average procedure (the magnetic field line), we can extract $\varepsilon^G(\vec{x},t)$ from the average operator, yielding the auxiliary equation that we need to compute $< \phi(\vec{x},t) >_\ell$:

$$-\nabla_\perp \cdot \big(\varepsilon^G\nabla_\perp < \phi(\vec{x},t) >_\ell \big) = \frac{1}{\epsilon_0}(< \rho_i^G >_\ell - < \bar{\rho}_{e0}^G >_\ell). \qquad (4.16)$$

The average ion charge density can be computed from $\rho_i^G(\vec{x},t)$ which is input data for the solver. To compute the average of the electron density we need to explicitly invoke the initial quasineutrality condition, i.e.: $\rho_i = \rho_e$. With this, $< \bar{\rho}_{e0}^G >_\ell = < \bar{\rho}_{i0}^G >_\ell$ and we are able to compute all the quantities involved based on the initial ion distribution profile.

Once we calculate $< \phi(\vec{x},t) >_\ell$ with equation (4.16) we can use this to solve the final version of our quasineutral equation, after introducing all the assumptions:

$$-\nabla_\perp \cdot \big(\bar{\varepsilon}^G(\vec{x},0)\nabla_\perp\phi(\vec{x},t)\big) + \frac{1}{\bar{\lambda}_D}(\phi(\vec{x},t) - < \phi(\vec{x},t) >_\ell) = \frac{1}{\epsilon_0}(\rho_i^G - \bar{\rho}_{i0}^G). \quad (4.17)$$

4.4.2 Exposed Interface

Fundamental type: None. It is a function that operates on other top-level types.
Function:

```
sll_solve_quasi_neutral_equation( electron_T_profile_2D,
                                  initial_rho_ion_profile_2D,
                                  charge_density,
                                  phi )
```

4.4.3 Usage

4.4.4 Status

4.5 Particle Distribution Function

4.5.1 Description

4.5.2 Exposed Interface

Fundamental type:

```
sll_distribution_function_t
```

All the fundamental types in the library are implemented as pointers. This choice has been made to ease the addition of Python bindings, in case that an even higher-level interface is desired some day.

Constructor and destructor:

```
sll_new_distribution_function( nr, ntheta, nphi, nvpar, mu )
sll_delete_distribution_function( f )
```

The constructor essentially limits itself to allocating the memory for the type. An initialization step is required afterwards:

```
sll_initialize_df( boundary_type_r,
                   boundary_type_vpar,
                   species_charge )
```

The accessors that get/set a particular value of a distribution function on a node are defined as macros to be able to keep a single interface while not risking a penalization when used in critical loops. Other queries on this type are implemented as ordinary functions. (Need to define this better, for instance, if some query functions would operate on integer arguments and/or real coordinates.)

```
SLL_GET_DF_VAL( i, j, k, l, df )
SLL_SET_DF_VAL( val, i, j, k, l, df )
sll_interpolate_df( r, theta, phi, vpar, mu )
sll_compute_derivative( f, r, theta, phi, vpar, mu )
sll_get_df_nr( df )
sll_get_df_nphi( df )
sll_get_df_ntheta( df )
sll_get_df_nvpar( df )
sll_get_df_mu( df )
```

The type also offers the services:

```
sll_compute_moments( df, ... )
```

4.5.3 Usage

4.5.4 Status

4.6 Advection Field

4.6.1 Description

4.6.2 Exposed Interface

Fundamental type:

```
sll_advection_field_3D_t
```

This implies that one of the options is to have multiple representations, for 3D, 2D, 1D.

4.6.3 Usage

4.6.4 Status

4.7 Advection

4.7.1 Description

4.7.2 Exposed Interface

Fundamental type: None. This is a function that operates on multiple top-level types. Function:

```
sll_advect( distribution_function,
            advection_field,
            dt,
            space_mesh
            scheme )
```

Above, `scheme` is the functional parametrization of the various methods in use (PSM, BSL, ...) and for which we need a standardized interface. The above assumes that we can devise a standard functional interface.

4.7.3 Usage

4.7.4 Status