SsTC library:
User's Guide
Version 0.3.0

Álvaro R. Puente-Uriona

October 2, 2023

Contents

1 Scope

Solid-state Task Constructor (SsTC) is a high-performance computing (HPC) oriented Fortran 2018 library which aims to aid programmers and researchers in the process of sampling and integration of functions in the first Brillouin zone (BZ) of a given crystalline system. The library is centered around the following concepts:

- The calculator: A general function representing a physical quantity of interest to the programmer or researcher. It is of the form
$$C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta}), \tag{1}$$
where $\boldsymbol{k}$ is a vector in the BZ and $\boldsymbol{\alpha}, \boldsymbol{\beta}$ encompass *all* other functional dependences of C. In the document we refer to $\boldsymbol{\alpha}$ as integer or discrete indices and seek to represent, for example, a set of Cartesian components $\{x, y, z\}$ or band indices. On the other side, $\boldsymbol{\beta}$ are referred as continuous indices and seek to represent the dependence of C on non-intrinsic variables such as an externally controlled frequency ω or a variable range of Fermi energies ε_F.
- The task: An object containing a complete description of the calculator C and the sampling or integration task that the programmer wants to perform.
- The system: An object containing a complete description of a crystalline system given by its tight-binding [1] representation. This includes:
  - Number of bands.
  - Fermi energy.
  - Unit cell description: the projection of lattice vectors $\boldsymbol{a}_i$ in the $\{x, y, z\}$ axes in units of Å.
  - Resolution of the Hamiltonian operator $\hat{H}$ in the basis of the Wannier functions (WFs) [1] in units of eV, i.e., the on-site and tunnelling amplitudes.
  - Resolution of the position operator $\hat{\boldsymbol{r}}$ in the basis of the WFs in units of Å.

The sampling and integration utilities contained in SsTC are the following.

- "Kpath" sampling. 1-dimensional sampling of C with specification of the path to sample.
- "Kslice" sampling. 2-dimensional regular sampling of C with specification of the slice to sample.
- "Regular" sampling. 3-dimensional regular sampling of C with specification of the number of samples in each dimension.
- Integration. 3-dimensional regular sampling of C followed by an integration process. The integration process as for v0.3.0 amounts to the rectangle approximation in each dimension. There exists also an experimental integration method based on extrapolation methods.

All the concepts covered in this section are treated with greater detail in the specific sections in the document.

2 Prerequisites, installation & linking to application

SsTC is a Fortran 2018 standard complying code and the prerequisites for its compilation are,

- Intel oneAPI Math Kernel Libraries (MKL), OpenMP library and MPI library.
- Intel Fortran oneAPI compilers `mpiifort` or `mpiifx`.
- Make software.
- Python 3 (> v3.9) software with 're' and 'glob' libraries.

As for v0.3.0, SsTC is only guaranteed to compile on Linux systems using the `mpiifort` and `mpiifx` (with

limitations) compilers contained in the Intel®oneAPI HPC toolkit. As such, the main application to be linked with must be compiled with `mpiifort` or `mpiifx`.

The SsTC project is hosted on GitHub and can be downloaded by running the command

Download

```
git clone --recurse-submodules https://github.com/irukoa/SsTC.git
```

or by downloading a compressed version of the source code on the tag history.

To install the source code, move the `SsTC` folder to a path of your choice and run `make`.

Installation

```
bash:/current/dir$ mv SsTC/ /path/of/your/choice/
bash:/path/of/your/choice$ cd SsTC/
bash:/path/of/your/choice/SsTC$ make
```

These commands will compile the source code and install the library `libSsTC.a` and the module header file `sstc.mod` in the directory `/path/of/your/choice/SsTC/bin/`.

The default behaviour will compile the library with `mpiifort`. This is the most tested and recommended compiler, but the user can also employ `mpiifx` to compile. This can be achieved by running,

Installation using ifx

```
bash:/path/of/your/choice/SsTC$ cp config/ifx Makefile
bash:/path/of/your/choice/SsTC$ make
```

which will replace the Makefile for compilation with `mpiifort` by the one for compilation with `mpiifx` and build the library.

To link to your Fortran application `appl.F90`, add the line `use SsTC` in your application preamble and compile with

Compilation & Linking

```
bash:/path/to/application/$ $(F90) $(F90FLAGS) appl.F90
-I/path/of/your/choice/SsTC/bin
/path/of/your/choice/SsTC/bin/libSsTC.a -o "appl.x"
```

where `F90 = mpiifort/mpiifx` and `F90FLAGS` contains, at least,

Compilation flags

```
F90FLAGS = -qopenmp -lmkl_intel_lp64 -lmkl_core -lmkl_gnu_thread -pthread
```

To make use of SsTC, the application preamble of `appl.F90` should also contain the lines

Use statements

```
use MPI_F08
use OMP_LIB
```

The application must also contain a call to the MPI [2] initialization routine `call MPI_INIT(ierror)` and to the SsTC initialization routine `call SsTC_init()` *before* any call to other SsTC routines. The programmer also needs to make sure that the MPI finalizing routine `call MPI_FINALIZE(ierror)` has not been called before using SsTC routines.

Note: By default, SsTC uses double precision `dp, kind=8` numbers for real and complex valued scalars and arrays.

The application `appl.x` can be run by the commands

Running

```
/path/to/application/appl.x
mpirun -np $N /path/to/application/appl.x
```

To uninstall the library run

Uninstalling

```
bash:/path/of/your/choice/SsTC$ make uninstall
```

3 Calculators & tasks

The calculator Eq. (1) is the generic name for Fortran function with interface

```
abstract interface
  function SsTC_global_calculator(task, system, k, error) &
           result(u)
    class(SsTC_global_k_data), intent(in) :: task
    type(SsTC_sys), intent(in)            :: system
    real(kind=dp), intent(in)             :: k(3)
    !In coords. relative to recip. lattice vectors.
    logical, intent(inout)                :: error

    complex(kind=dp) :: u(product(task%integer_indices), &
                          product(task%continuous_indices))
  end function SsTC_global_calculator
end interface
```

Listing 1: Interface of a global calculator.

given by external SsTC modules, see Sec. 10, or otherwise provided by the user in the scope of the main application using SsTC. The calculator must be interface conforming.

Additionally to the general "global calculator", there exists a "local calculator" with interface

```
abstract interface
  function SsTC_local_calculator(k_data, system, k, error) &
           result(u)
    class(SsTC_local_k_data), intent(in) :: k_data
    type(SsTC_sys), intent(in)           :: system
    real(kind=dp), intent(in)            :: k(3)
    !In coords. relative to recip. lattice vectors.
    logical, intent(inout)               :: error

    complex(kind=dp) :: u(product(k_data%integer_indices))
  end function SsTC_local_calculator
end interface
```

Listing 2: Interface of a local calculator.

with reduced functionality and meant for internal computations regarding quantities local for each $\boldsymbol{k}$ point, as described in Sec. 8.4.

The task is a Fortran object (derived type) of `class(SsTC_local_k_data)`, of which `type(SsTC_global_k_data)` is an extension, specifying the sampling or integration task to perform and contains a full description of the properties of the calculator $\boldsymbol{\alpha}, \boldsymbol{\beta}$. The generic properties which apply to every task `task` are the following,

- `character(len=*) :: task%name`: The name given to the task.
- `procedure(SsTC_global_calculator), pointer :: task%global_calculator`: Interface conforming procedure pointer to the selected calculator. The library's sampling routines will use the provided calculator to sample.
- `procedure(SsTC_local_calculator), pointer :: task%local_calculator`: Interface conforming procedure pointer to the selected calculator. The library's sampling routines will use the provided calculator to sample. Note: only one of `task%local_calculator` or `task%global_calculator` shall be associated when sampling.
- `integer :: task%integer_indices(N_int_ind)`: Each entry `i` of the array contains the number of values the integer index α_i in Eq. (1) can have. `N_int_ind` is the total number of integer indices encompassed by $\boldsymbol{\alpha}$ and the total number of integer index combinations is given by `product(task%integer_indices)`.
- `integer :: task%continuous_indices(N_ext_vars)`: Each entry `i` of the array contains the number `ext_vars_steps(i)` of values the continuous index β_i in Eq. (1) can have. `N_ext_vars` is the total number of external variables encompassed by $\boldsymbol{\beta}$ and the total number of continuous index combinations is given by `product(task%continuous_indices)`. Not applicable to `type(SsTC_local_k_data)`.
- `real(kind=dp) :: task%ext_var_data(N_ext_vars)%data(ext_vars_steps(N_ext_vars))`: Each entry `i, j` corresponding to `task%ext_var_data(i)%data(j)` is a real number λ_{ij} containing the particular value the continuous index β_i has, as given by

$$\lambda_{ij} = \lambda_{i1} + \left(\lambda_{i\,\texttt{ext\_vars\_steps(i)}} - \lambda_{i1}\right) \times (j-1) \,/\, (\texttt{ext\_vars\_steps(i)} - 1)\,. \tag{2}$$

  where λ_{i1} and $\lambda_{i\,\texttt{ext\_vars\_steps(i)}}$ are the starting and ending points of the values given to β_i, $j \in [1, \texttt{ext\_vars\_steps(i)}]$. Not applicable to `type(SsTC_local_k_data)`.

Particular tasks can be created by means of a specific sampling or integrator task constructor as described in Sec. 7 or by the programmer with complete freedom. Extension of the task members is also possible by means of derived type extension.

In the following we provide the type declarations of `SsTC_local_k_data` and `SsTC_global_k_data`,

Global k data

```
type SsTC_local_k_data
  character(len=120)                                  :: name
  integer, allocatable                                :: integer_indices(:)
  !Each entry contains the range of each of the integer indices.
  complex(kind=dp), allocatable                       :: k_data(:)
  !Data local for each k with integer index in memory array.
  procedure(SsTC_local_calculator), pointer, nopass :: local_calculator => null()
  !Pointer to the local calculator.
  integer                                             :: particular_integer_component &
                                                         = 0
  !Specification of some integer component.
end type SsTC_local_k_data
```

Listing 3: Derived type corresponding to "local k data".

Local k data

```
type, extends(SsTC_local_k_data) :: SsTC_global_k_data
  integer, allocatable                                :: continuous_indices(:)
  !Each entry contains the range of each continuous indices.
  type(SsTC_external_vars), allocatable               :: ext_var_data(:)
  !External variable data.
```

```
  procedure(SsTC_global_calculator), pointer, nopass :: global_calculator => null()
  !Pointer to the global calculator.
  integer, allocatable                               :: iterables(:, :)
  !Iterable dictionary.
end type SsTC_global_k_data
```

Listing 4: Derived type corresponding to "global k data".

4 SsTC initialization

SsTC can be initialized in the application by using the routine `SsTC_init()`.

SsTC initializer

```
subroutine SsTC_init(nThreads, nNested, exec_label)

  integer, intent(in), optional :: nThreads
  !Default are max available threads per MPI process.
  integer, intent(in), optional :: nNested !Default = 1.
  character(len=*), intent(in), optional :: exec_label !Default "SsTC_exec".

end subroutine SsTC_init
```

Listing 5: Interface of "SsTC initialization".

The routine will check whether the MPI library has been initialized and stop the program on execution if it was not. It will also open the output and error log files `file=trim(exec_label//".out")` and `file=trim(exec_label//".err")`. The user can also control the number of parallel nested regions with the variable `nNested`. Changing this variable from the default value of 1 is not recommended unless the number of $\boldsymbol{k}$ points being sampled by a particular task is smaller than the total number of threads available. Lastly, `nThreads` sets the number of threads per MPI process, with the default value of `OMP_GET_MAX_THREADS()`.

5 Notation: Memory layout and array layout

These concepts apply to N-dimensional arrays such as `array(:, :, ..., :)`, where each dimension i has size s_i. The total size of `array` is

$$\texttt{size(array)} = \prod_{i=1}^{N} s_i. \tag{3}$$

An array with such a shape is said to be in "array layout". In tandem this layout, we consider now the 1-dimensional array `mem(:)`, with the same size as `array`. The array `mem(:)` is defined as the "memory layout" counterpart of the array `array` if

$$\texttt{mem(r)} = \texttt{array(n\_1, n\_2, ..., n\_N)}, \tag{4}$$

with

$$r = n_1 + s_1\left(n_2 + s_2\left(n_3 + \cdots\right)\cdots\right) = \sum_{i=1}^{N} n_i \times \left(\prod_{j=1}^{i-1} s_j\right). \tag{5}$$

This mapping is called the column mayor array to memory index mapping. It provides an invertible relation

$$r \Leftrightarrow \{n_1, n_2, \cdots, n_N\}, \tag{6}$$

which makes it possible to keep track of the elements of an array in both layouts, provided that the sizes of each dimension, s_i, are known. The output values of the calculator interfaces, corresponding to the sampling $C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta})$ for a $\boldsymbol{k}$ vector, are always arrays in memory layout, both for integer and continuous indices, which makes the interface of the calculator flexible for any number of integer or continuous indices.

In SsTC, the sizes s_i of integer or continuous indices are specified by the components of `task%integer_indices` and `task%continuous_indices` (if applicable) of the task `class(SsTC_local_k_data) :: task`, respectively. The library provides the utilities

- `SsTC_integer_array_element_to_memory_element`: Returns $r = f(\{n_1, n_2, \cdots, n_N\})$ for integer indices.
- `SsTC_integer_memory_element_to_array_element`: Returns $\{n_1, n_2, \cdots, n_N\} = f^{-1}(r)$ for integer indices.
- `SsTC_continuous_array_element_to_memory_element`: Returns $r = f(\{n_1, n_2, \cdots, n_N\})$ for continuous indices.
- `SsTC_continuous_memory_element_to_array_element`: Returns $\{n_1, n_2, \cdots, n_N\} = f^{-1}(r)$ for continuous indices.

where f is a function representing the mapping Eq. (5) and f^{-1} represents its inverse mapping. The four utilities are described in detail in Sec. 8.3.

6 Systems

A system is a Fortran derived type

Crystalline systems

```
type SsTC_sys
  character(len=120)            :: name
  integer                       :: num_bands
  real(kind=dp)                 :: direct_lattice_basis(3, 3)
  !1st index is vector label, 2nd index is vector component.
  real(kind=dp)                 :: metric_tensor(3, 3)
  !Metric tensor of the direct lattice basis.
  real(kind=dp)                 :: cell_volume
  integer                       :: num_R_points
  !Number of R points (unit cells).
  integer, allocatable          :: R_point(:, :)
  !Id of the R-point (1st index) and R-vector coords.
  !relative to the direct lattice basis vectors (2nd index).
  integer, allocatable          :: deg_R_point(:)
  !Degeneracy of the R-point specified by its memory layout id.
  complex(kind=dp), allocatable :: real_space_hamiltonian_elements(:, :, :)
  !Hamiltonian matrix elements (1st and 2nd indexes) and
  !id of the R-point (3rd index) in eV.
  complex(kind=dp), allocatable :: real_space_position_elements(:, :, :, :)
  !Position operator matrix elements (1st and 2nd indexes),
  !Cartesian coordinate (3rd index) and id of the R-point (4th index) in A.
  real(kind=dp)                 :: e_fermi = 0.0_dp
  !Fermi energy.
  real(kind=dp)                 :: deg_thr = 1.0E-4_dp
  !Degeneracy threshold in eV.
  real(kind=dp)                 :: deg_offset = 0.04_dp
  !Offset for regularization in case of degeneracies, in eV.
end type SsTC_sys
```

Listing 6: Derived type corresponding to a system.

representing a crystalline system by its tight-binding [1] representation. The components

$$\texttt{real\_space\_hamiltonian\_elements(m, n, id)} = \langle m\mathbf{0}|\hat{H}|n\boldsymbol{R}\rangle\,, \quad \texttt{id} \text{ corresponds to } \boldsymbol{R}. \tag{7}$$

and

$$\texttt{real\_space\_position\_elements(m, n, i, id)} = \langle m\mathbf{0}|\hat{r}_i|n\boldsymbol{R}\rangle\,, \quad \texttt{id} \text{ corresponds to } \boldsymbol{R}. \tag{8}$$

are identified. The recommended way to create system is by using the function `SsTC_sys_constructor`,

System constructor

```
function SsTC_sys_constructor(name, path_to_tb_file, &
                              efermi, deg_thr, deg_offset) &
  result(system)

  character(len=*), intent(in)         :: name
  character(len=*), intent(in)         :: path_to_tb_file
  real(kind=dp), optional, intent(in) :: efermi, &
                                         deg_thr, deg_offset

  type(SsTC_sys) :: system
end function SsTC_sys_constructor
```

Listing 7: Interface of the system constructor.

where the optional arguments can be specified. The function will try to read a file named `trim(path_to_tb_file)//trim(name)//"_tb.dat"` in the path relative to the main application directory. The files have the format of a Wannier90 [3] tight-binding (`*_tb.dat` file, see Sec. 8.21 of the Wannier90 user's guide for v3.1.0) which can be written by the user for toy tight-binding models or can be generated by the code Wannier90 from postprocessing *ab-initio* calculations. As for v3.1.0 of Wannier90, the file is automatically generated when running `wannier90.x` if the option `write_tb=.TRUE.` is specified in the Wannier90 input card (`*.win` file).

7 Sampling and integration routines

In this section we describe the main routines of the SsTC library, encompassing task creation, task sampling or integration and printing to files. Note that the sampling or integration subroutines take as inputs tasks corresponding to the same `class` as the tasks generated by the respective task constructors, which are, at the same time, extensions of `type(SsTC_global_k_data)`. As such, much flexibility in the definition of calculators can be achieved by means of type extension.

7.1 Kpath module

This module is centered around creating, sampling, and printing tasks which involve a path in reciprocal space. The "kpath" task is a derived type

Kpath task

```
type, extends(SsTC_global_k_data) :: SsTC_kpath_task
  !1st index is the id of the vector in the path.
  !2nd index corresponds to the component of the vector
  !in the path in coordinates relative to the reciprocal lattice.
  real(kind=dp), allocatable :: vectors(:, :)
  !number_of_pts(i) contains the number of k-points between vector i and vector i+1.
  integer, allocatable :: number_of_pts(:)
  !Array to store data with integer index,
  !continuous index and kpt index respectively.
  complex(kind=dp), allocatable :: kpath_data(:, :, :)
end type SsTC_kpath_task
```

Listing 8: Derived type corresponding to a "kpath" task.

where we consider a set of N reciprocal space vectors $\{\boldsymbol{q}_i, i \in [1, N]\}$ by their components $\{a_{ij}\}$ relative to the reciprocal space basis vectors $\boldsymbol{b}_{\{1,2,3\}}$,

$$\boldsymbol{q}_i = \sum_{j=1}^{3} a_{ij} \times \boldsymbol{b}_j, \quad a_{ij} \in [-0.5, 0.5]\,, \tag{9}$$

and identify

$$\texttt{vectors(i, j)} = a_{ij}. \tag{10}$$

The array `number_of_pts(i)` contains the number of points between vector $\boldsymbol{q}_i$ and vector $\boldsymbol{q}_{i+1}$. For the general calculator Eq. (1), we identify

$$\texttt{kpath\_data(alpha, beta, ik)} = C^{\alpha}(\boldsymbol{k};\boldsymbol{\beta}), \quad \boldsymbol{k} \text{ is identified with } \texttt{ik}, \tag{11}$$

with

$$\boldsymbol{k} = \boldsymbol{q}_i + (\boldsymbol{q}_{i+1} - \boldsymbol{q}_i) \times (\texttt{ik} - 1)/(\texttt{number\_of\_pts(i)} - 1). \tag{12}$$

A kpath task can be constructed by the function `SsTC_kpath_constructor`,

Kpath task constructor

```
subroutine SsTC_kpath_constructor(task, name, &
                                  l_calculator, g_calculator, &
                                  Nvec, vec_coord, nkpts, &
                                  N_int_ind, int_ind_range, &
                                  N_ext_vars, ext_vars_start, ext_vars_end, &
                                  ext_vars_steps, &
                                  part_int_comp)

  character(len=*) :: name

  procedure(SsTC_local_calculator), optional  :: l_calculator
  procedure(SsTC_global_calculator), optional :: g_calculator

  integer, intent(in)       :: Nvec
  real(kind=dp), intent(in) :: vec_coord(Nvec, 3)
  integer, intent(in)       :: nkpts(Nvec - 1)

  integer, intent(in)           :: N_int_ind
  integer, optional, intent(in) :: int_ind_range(N_int_ind)

  integer, intent(in)                 :: N_ext_vars
  real(kind=dp), optional, intent(in) :: ext_vars_start(N_ext_vars), &
                                         ext_vars_end(N_ext_vars)
  integer, optional, intent(in)       :: ext_vars_steps(N_ext_vars)

  integer, optional, intent(in) :: part_int_comp(N_int_ind)

  class(SsTC_kpath_task), intent(out) :: task
end subroutine SsTC_kpath_constructor
```

Listing 9: Interface of the kpath task constructor.

where

- `name`: Name given to the task.
- `l_calculator`: Pointer to a function that wants to be sampled, with interface 2. Only one of `l_calculator` or `g_calculator` can be specified.
- `g_calculator`: Pointer to a function that wants to be sampled, with interface 1. Only one of `l_calculator` or `g_calculator` can be specified.
- `Nvec`: Number of vectors in the path.
- `vec_coord(i, j)`: Vector coordinates a_{ij} in Eq. (9).
- `nkpts(i)`: Number of points between vectors $\boldsymbol{q}_i$ and $\boldsymbol{q}_{i+1}$.
- `N_int_ind`: Number of integer indices.

- `int_ind_range(i)`: Number of values the integer index α_i can have.
- `N_ext_vars`: Number of continuous variables.
- `ext_vars_start(i)`: Starting point λ_{i1} in Eq. (2) for the variable β_i.
- `ext_vars_end(i)`: Ending point $\lambda_{i\,\texttt{ext\_vars\_steps(i)}}$ in Eq. (2) for the variable β_i.
- `ext_vars_steps(i)`: Number of points into which to discretize the variable β_i.
- `part_int_comp(N_int_ind)`: Array corresponding to a selection of a particular integer component in array layout.

Sampling, and thus writing the values of C in Eq. (1) given by `l_calculator` or `g_calculator` to the array `kpath_data` can be made with the subroutine `SsTC_kpath_sampler`.

Kpath task sampler

```
subroutine SsTC_kpath_sampler(task, system)
  class(SsTC_kpath_task), intent(inout) :: task
  type(SsTC_sys), intent(in)            :: system
end subroutine SsTC_kpath_sampler
```

Listing 10: Interface of the kpath task sampler.

All the `allocatable` components of both `task` and `system` should be allocated before the subroutine call either by the corresponding constructors or by the user.

Writing to files can be done by means of the subroutine `SsTC_print_kpath`.

Kpath task printer

```
subroutine SsTC_print_kpath(task, system)
  class(SsTC_kpath_task), intent(in) :: task
  type(SsTC_sys), intent(in)         :: system
end subroutine SsTC_print_kpath
```

Listing 11: Interface of the kpath task printer.

The routine will write a file for each integer index with name `trim(system%name)//'-'//trim(task%name)//'_'trim(num_label)//'.dat'` with `num_label` being an `N_int_ind`-dimensional array with the corresponding integer index in array layout (see Sec. 5). Each file will contain, column by column, the following,

- An `id` corresponding to the particular $\boldsymbol{k}$ point being sampled (1 column).
- The components $a_{i\{1,2,3\}}$ in Eq. (9) of the vector $\boldsymbol{k}$ corresponding to `id` (3 columns).
- For each continuous index i, the particular values of the data λ_{ij} as given by Eq. (2) (`size(task%continuous_indices)` columns).
- The real and imaginary part of the calculator $C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta})$ (2 columns).

If further use of the sampled data is intended within the execution of the application, making a copy of `task%kpath_data` is suggested.

7.2 Kslice module

This module is centered around creating, sampling, and printing tasks which involve a 2-dimensional reciprocal space "slice". The "kslice" task is a derived type

Kslice task

```
type, extends(SsTC_global_k_data) :: SsTC_kslice_task
  !Default: sample k_z = 0 slice in a 100x100 mesh.
  real(kind=dp) :: corner(3) = (/-0.5_dp, -0.5_dp, 0.0_dp/), &
                   vector(2, 3) = reshape((/1.0_dp, 0.0_dp, 0.0_dp, &
                   1.0_dp, 0.0_dp, 0.0_dp/), (/2, 3/))
  integer       :: samples(2) = (/100, 100/)
  !Integer index, continuous index and kpt index 1 and 2 respectively.
  complex(kind=dp), allocatable :: kslice_data(:, :, :, :)
end type SsTC_kslice_task
```

Listing 12: Derived type corresponding to a kslice task.

where we consider 3 reciprocal space vectors $\{\boldsymbol{q}_i, i \in [1,3]\}$ by their components $\{a_{ij}\}$ relative to the reciprocal space basis vectors $\boldsymbol{b}_{\{1,2,3\}}$,

$$\boldsymbol{q}_i = \sum_{j=1}^{3} a_{ij} \times \boldsymbol{b}_j, \quad a_{ij} \in [-0.5, 0.5], \tag{13}$$

and identify the two vectors spanning a plane

$$\texttt{vector(i, j)} = a_{ij}, \quad i = [1,2]. \tag{14}$$

We also identify the "sampling corner" or starting point

$$\texttt{corner(j)} = a_{3,j}. \tag{15}$$

For the general calculator Eq. (1), we identify

$$\texttt{kslice\_data(alpha, beta, ik1, ik2)} = C^{\boldsymbol{\alpha}}(\boldsymbol{k}; \boldsymbol{\beta}), \quad \boldsymbol{k} \text{ is identified with } \texttt{ik1, ik2}, \tag{16}$$

and given by

$$\boldsymbol{k} = \boldsymbol{q}_3 + \boldsymbol{q}_1 \times (\texttt{ik1} - 1)/(\texttt{samples(1)} - 1) + \boldsymbol{q}_2 \times (\texttt{ik2} - 1)/(\texttt{samples(2)} - 1). \tag{17}$$

A kslice task can be constructed by the function `SsTC_kslice_task_constructor`,

Kslice task constructor

```
subroutine SsTC_kslice_task_constructor(task, name, &
                                        l_calculator, g_calculator, &
                                        corner, vector_a, vector_b, samples, &
                                        N_int_ind, int_ind_range, &
                                        N_ext_vars, ext_vars_start, ext_vars_end, &
                                        ext_vars_steps, &
                                        part_int_comp)

  character(len=*) :: name

  procedure(SsTC_local_calculator), optional  :: l_calculator
  procedure(SsTC_global_calculator), optional :: g_calculator

  real(kind=dp), optional, intent(in) :: corner(3), vector_a(3), vector_b(3)
  integer, optional, intent(in)       :: samples(2)

  integer, intent(in)           :: N_int_ind
  integer, optional, intent(in) :: int_ind_range(N_int_ind)

  integer, intent(in)                 :: N_ext_vars
```

```
  real(kind=dp), optional, intent(in) :: ext_vars_start(N_ext_vars), &
                                         ext_vars_end(N_ext_vars)
  integer, optional, intent(in)       :: ext_vars_steps(N_ext_vars)

  integer, optional, intent(in) :: part_int_comp(N_int_ind)

  class(SsTC_kslice_task), intent(out) :: task
end subroutine SsTC_kpath_constructor
```

Listing 13: Interface of the kslice task constructor.

where

- `name`: Name given to the task.
- `l_calculator`: Pointer to a function that wants to be sampled, with interface 2. Only one of `l_calculator` or `g_calculator` can be specified.
- `g_calculator`: Pointer to a function that wants to be sampled, with interface 1. Only one of `l_calculator` or `g_calculator` can be specified.
- `corner(3)`: Starting point of the sampling.
- `vector_a(3)`: Vector coordinates a_{1j} in Eq. (13).
- `vector_b(3)`: Vector coordinates a_{2j} in Eq. (13).
- `samples(2)`: Each entry i contains the number into which $\boldsymbol{q}_i$ has been discretized.
- `N_int_ind`: Number of integer indices.
- `int_ind_range(i)`: Number of values the integer index α_i can have.
- `N_ext_vars`: Number of continuous variables.
- `ext_vars_start(i)`: Starting point λ_{i1} in Eq. (2) for the variable β_i.
- `ext_vars_end(i)`: Ending point $\lambda_{i\,\texttt{ext\_vars\_steps(i)}}$ in Eq. (2) for the variable β_i.
- `ext_vars_steps(i)`: Number of points into which to discretize the variable β_i.
- `part_int_comp(N_int_ind)`: Array corresponding to a selection of a particular integer component in array layout.

Sampling, and thus writing the values of C in Eq. (1) given by `l_calculator` or `g_calculator` to the array `kslice_data` can be made with the subroutine `SsTC_sample_kslice_task`.

Kslice task sampler

```
subroutine SsTC_sample_kslice_task(task, system)
  class(SsTC_kslice_task), intent(inout) :: task
  type(SsTC_sys), intent(in)             :: system
end subroutine SsTC_sample_kslice_task
```

Listing 14: Interface of the kslice task sampler.

All the `allocatable` components of both `task` and `system` should be allocated before the subroutine call either by the corresponding constructors or by the user.

Writing to files can be done by means of the subroutine `SsTC_print_kslice`.

Kslice task printer

```
subroutine SsTC_print_kslice(task, system)
  class(SsTC_kpath_task), intent(in) :: task
  type(SsTC_sys), intent(in)         :: system
end subroutine SsTC_print_kslice
```

Listing 15: Interface of the kslice task printer.

The routine will write a file for each integer index with name `trim(system%name)//'-'//trim(task%name)//'_'trim(num_label)//'.dat'` with `num_label` being an `N_int_ind`-dimensional array with the corresponding integer index in array layout (see Sec. 5). Each file will contain, column by column, the following,

- The components $a_{i\{1,2,3\}}$ in Eq. (13) of the vector $\boldsymbol{k}$ corresponding to `id` (3 columns).
- For each continuous index i, the particular values of the data λ_{ij} as given by Eq. (2) (`size(task%continuous_indices)` columns).
- The real and imaginary part of the calculator $C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta})$ (2 columns).

If further use of the sampled data is intended within the execution of the application, making a copy of `task%kslice_data` is suggested.

7.3 Sampler module

This module is centered around creating, sampling, and printing tasks which involve a regular 3-dimensional BZ sampling. The "sampler" task is a derived type

Sampler task

```
type, extends(SsTC_global_k_data) :: SsTC_sampling_task
  integer                         :: samples(3) = (/100, 100, 100/)
  !Integer index, continuous index and kpt index 1, 2 and 3 respectively.
  complex(kind=dp), allocatable :: BZ_data(:, :, :, :, :)
end type SsTC_sampling_task
```

Listing 16: Derived type corresponding to a sampler task.

For the general calculator Eq. (1), and the reciprocal space basis vectors $\boldsymbol{b}_{\{1,2,3\}}$, we identify

$$\texttt{BZ\_data(alpha, beta, ik1, ik2, ik3)} = C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta}), \quad \boldsymbol{k} \text{ is identified with } \texttt{ik1, ik2, ik3}, \tag{18}$$

and given by

$$\boldsymbol{k} = \sum_{i=1}^{3} \boldsymbol{b}_i \times (\texttt{ik}i - 1)/(\texttt{samples(i)} - 1). \tag{19}$$

A sampling task can be constructed by the function `SsTC_sampling_task_constructor`,

Sampler task constructor

```
subroutine SsTC_sampling_task_constructor(task, name, &
                                          l_calculator, g_calculator, &
                                          samples, &
                                          N_int_ind, int_ind_range, &
                                          N_ext_vars, &
                                          ext_vars_start, ext_vars_end, &
                                          ext_vars_steps, &
                                          part_int_comp)
```

```
  character(len=*) :: name

  procedure(SsTC_local_calculator), optional  :: l_calculator
  procedure(SsTC_global_calculator), optional :: g_calculator

  integer, optional, intent(in) :: samples(3)

  integer, intent(in)           :: N_int_ind
  integer, optional, intent(in) :: int_ind_range(N_int_ind)

  integer, intent(in)                  :: N_ext_vars
  real(kind=dp), optional, intent(in) :: ext_vars_start(N_ext_vars), &
                                         ext_vars_end(N_ext_vars)
  integer, optional, intent(in)        :: ext_vars_steps(N_ext_vars)

  integer, optional, intent(in) :: part_int_comp(N_int_ind)

  class(SsTC_sampling_task), intent(out) :: task
end subroutine SsTC_sampling_task_constructor
```

Listing 17: Interface of the sampling task constructor.

where

- `name`: Name given to the task.
- `l_calculator`: Pointer to a function that wants to be sampled, with interface 2. Only one of `l_calculator` or `g_calculator` can be specified.
- `g_calculator`: Pointer to a function that wants to be sampled, with interface 1. Only one of `l_calculator` or `g_calculator` can be specified.
- `samples(3)`: Each entry i contains the number into which $\boldsymbol{b}_i$ has been discretized.
- `N_int_ind`: Number of integer indices.
- `int_ind_range(i)`: Number of values the integer index α_i can have.
- `N_ext_vars`: Number of continuous variables.
- `ext_vars_start(i)`: Starting point λ_{i1} in Eq. (2) for the variable β_i.
- `ext_vars_end(i)`: Ending point $\lambda_{i\,\texttt{ext\_vars\_steps(i)}}$ in Eq. (2) for the variable β_i.
- `ext_vars_steps(i)`: Number of points into which to discretize the variable β_i.
- `part_int_comp(N_int_ind)`: Array corresponding to a selection of a particular integer component in array layout.

Sampling, and thus writing the values of C in Eq. (1) given by `l_calculator` or `g_calculator` to the array `BZ_data` can be made with the subroutine `SsTC_sample_sampling_task`.

Sampler task sampler

```
subroutine SsTC_sample_sampling_task(task, system)
  class(SsTC_kslice_task), intent(inout) :: task
  type(SsTC_sys), intent(in)             :: system
end subroutine SsTC_sample_sampling_task
```

Listing 18: Interface of the sampling task sampler.

All the `allocatable` components of both `task` and `system` should be allocated before the subroutine call either by the corresponding constructors or by the user.

Writing to files can be done by means of the subroutine `SsTC_print_sampling`.

Sampler task printer

```
subroutine SsTC_print_sampling(task, system)
  class(SsTC_kpath_task), intent(in) :: task
  type(SsTC_sys), intent(in)         :: system
end subroutine SsTC_print_sampling
```

Listing 19: Interface of the sampling task printer.

The routine will write a file for each integer index with name `trim(system%name)//'-'//trim(task%name)//'_'trim(num_label)//'.dat'` with `num_label` being an `N_int_ind`-dimensional array with the corresponding integer index in array layout (see Sec. 5). Each file will contain, column by column, the following,

- The components `ik`$i = \{1, 2, 3\}$ of the vector $\boldsymbol{k}$ in Eq. (19) corresponding to `id` (3 columns).
- For each continuous index i, the particular values of the data λ_{ij} as given by Eq. (2) (`size(task%continuous_indices)` columns).
- The real and imaginary part of the calculator $C^{\boldsymbol{\alpha}}(\boldsymbol{k}; \boldsymbol{\beta})$ (2 columns).

If further use of the sampled data is intended within the execution of the application, making a copy of `task%BZ_data` is suggested.

7.4 Integrator module

This module is centered around creating, sampling, integrating, and printing tasks which involve a BZ integral. The "integrator" task is a derived type

Integration task

```
type, extends(SsTC_global_k_data) :: SsTC_BZ_integral_task
  !Integration method.
  character(len=120)                              :: method
  !Integration samples.
  integer                                         :: samples(3)
  !Result of the integration, contains the integer
  !index and the continuous index in memory layout, respectively.
  complex(kind=dp), allocatable                   :: result(:, :)
end type SsTC_BZ_integral_task
```

Listing 20: Derived type corresponding to an integrator task.

For the general calculator Eq. (1), and the reciprocal space basis vectors $\boldsymbol{b}_{\{1,2,3\}}$, we identify

$$\texttt{result(alpha, beta)} = \int_{\mathrm{BZ}} [d\boldsymbol{k}]\, C^{\boldsymbol{\alpha}}(\boldsymbol{k}; \boldsymbol{\beta}), \quad [d\boldsymbol{k}] = \frac{dk_1 dk_2 dk_3}{(2\pi)^3}, \tag{20}$$

where $\boldsymbol{k}$ is given by

$$\boldsymbol{k} = \sum_{i=1}^{3} \boldsymbol{b}_i \times (\texttt{ik}i - 1)/(\texttt{samples(i)} - 1). \tag{21}$$

An integrator task can be constructed by the function `SsTC_BZ_integral_task_constructor`,

Integration task constructor

```
subroutine SsTC_BZ_integral_task_constructor(task, name, &
                                             l_calculator, g_calculator, &
                                             method, samples, &
                                             N_int_ind, int_ind_range, &
                                             N_ext_vars, &
                                             ext_vars_start, ext_vars_end, &
                                             ext_vars_steps, &
                                             part_int_comp)

  character(len=*) :: name

  procedure(SsTC_local_calculator), optional  :: l_calculator
  procedure(SsTC_global_calculator), optional :: g_calculator

  character(len=*), optional, intent(in) :: method
  integer, optional, intent(in)          :: samples(3)

  integer, intent(in)            :: N_int_ind
  integer, optional, intent(in) :: int_ind_range(N_int_ind)

  integer, intent(in)                  :: N_ext_vars
  real(kind=dp), optional, intent(in) :: ext_vars_start(N_ext_vars), &
                                         ext_vars_end(N_ext_vars)
  integer, optional, intent(in)        :: ext_vars_steps(N_ext_vars)

  integer, optional, intent(in) :: part_int_comp(N_int_ind)

  class(SsTC_BZ_integral_task), intent(out) :: task
end subroutine SsTC_BZ_integral_task_constructor
```

Listing 21: Interface of the integrator task constructor.

where

- `name`: Name given to the task.
- `l_calculator`: Pointer to a function that wants to be sampled, with interface 2. Only one of `l_calculator` or `g_calculator` can be specified.
- `g_calculator`: Pointer to a function that wants to be sampled, with interface 1. Only one of `l_calculator` or `g_calculator` can be specified.
- `method`: is either `"rectangle"` or `"extrapolation"`, default is `rectangle`.
- `samples(3)`: Each entry i contains the number into which $\boldsymbol{b}_i$ has been discretized.
- `N_int_ind`: Number of integer indices.
- `int_ind_range(i)`: Number of values the integer index α_i can have.
- `N_ext_vars`: Number of continuous variables.
- `ext_vars_start(i)`: Starting point λ_{i1} in Eq. (2) for the variable β_i.
- `ext_vars_end(i)`: Ending point $\lambda_{i\,\texttt{ext\_vars\_steps(i)}}$ in Eq. (2) for the variable β_i.
- `ext_vars_steps(i)`: Number of points into which to discretize the variable β_i.
- `part_int_comp(N_int_ind)`: Array corresponding to a selection of a particular integer component in array layout.

Sampling, and thus writing the values of C in Eq. (1) given by `l_calculator` or `g_calculator` to the array `result` can be made with the subroutine `SsTC_sample_and_integrate_BZ_integral_task`.

Integration task sampler and integrator

```
subroutine SsTC_sample_and_integrate_BZ_integral_task(task, system)
  class(SsTC_kslice_task), intent(inout) :: task
  type(SsTC_sys), intent(in)             :: system
end subroutine SsTC_sample_and_integrate_BZ_integral_task
```

Listing 22: Interface of the integrator task sampler and integrator.

All the `allocatable` components of both `task` and `system` should be allocated before the subroutine call either by the corresponding constructors or by the user.

If `method = "rectangle"` is selected, the integral in Eq. (20) will be done performed within the rectangle approximation,

$$\int_{\mathrm{BZ}} [d\boldsymbol{k}]\, C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta}) \approx \left(\prod_{i=1}^{3} \texttt{samples(i)}\right)^{-1} \sum_{\boldsymbol{k}\in\mathrm{BZ}} C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta}). \tag{22}$$

If `method = "extrapolation"` is selected, the integral in Eq. (20) will be done performed by means of extrapolation methods, requiring large memory usage, but capable of greater precision per sampled $\boldsymbol{k}$ point. To employ extrapolation, each of the 3 elements of the array `samples(3)` must be either 1, or expressible as 2^n+1, $n = 0, 1, \cdots$, or else extrapolation will fail. If this is the case, the rectangle approximation will be employed. The extrapolation method and its implementation are powered by the `F90-Extrapolation-Integration` project, hosted on GitHub.

Writing to files can be done by means of the subroutine `SsTC_print_BZ_integral_task`.

Integration task printer

```
subroutine SsTC_print_BZ_integral_task(task, system)
  class(SsTC_kpath_task), intent(in) :: task
  type(SsTC_sys), intent(in)         :: system
end subroutine SsTC_print_BZ_integral_task
```

Listing 23: Interface of the integrator task printer.

The routine will write a file for each integer index with name `trim(system%name)//'-'//trim(task%name)//'_'trim(num_label)//'.dat'` with `num_label` being an `N_int_ind`-dimensional array with the corresponding integer index in array layout (see Sec. 5). Each file will contain, column by column, the following,

- For each continuous index i, the particular values of the data λ_{ij} as given by Eq. (2) (`size(task%continuous_indices)` columns).
- The real and imaginary part of the calculator $C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta})$ (2 columns).

If further use of the sampled data is intended within the execution of the application, making a copy of `task%result` is suggested.

8 Other routines

In this section we describe all other routines public to the user when using SsTC.

8.1 Utility module

The utility module contains various routines and definitions which are of use in the creation of calculators and are widely employed in most of SsTC routines.

8.1.1 Parameters: Symmetrization and antisymmetrization utilities

We provide the 4 arrays,

Symmetrizing and antisymmetrizing arrays

```
!Symmetric.
integer, dimension(6), parameter :: SsTC_alpha_S = (/1, 2, 3, 1, 1, 2/)
integer, dimension(6), parameter :: SsTC_beta_S = (/1, 2, 3, 2, 3, 3/)
!Antisymmetric.
integer, dimension(3), parameter :: SsTC_alpha_A = (/2, 3, 1/)
integer, dimension(3), parameter :: SsTC_beta_A = (/3, 1, 2/)
```

Listing 24: Symmetrization and antisymmetrization arrays.

which provide linearly independent combinations of (anti)symmetric pairs of 3-dimensional indices. Considering, a symmetric pair of indices $\{i, j\}$, the code

```
do ij = 1, 6
  i = SsTC_alpha_S(ij)
  j = SsTC_beta_S(ij)
  .
  .
  .
enddo
```

Listing 25: Loop over symmetric pair of indices.

will loop over the 6 linearly independent combination of indices. In the same way and now considering a pair of antisymmetric indices $\{i, j\}$,

```
do ij = 1, 3
  i = SsTC_alpha_A(ij)
  j = SsTC_beta_A(ij)
  .
  .
  .
enddo
```

Listing 26: Loop over antisymmetric pair of indices.

will loop over the 3 linearly independent combination of indices.

8.1.2 Utilities

The function `SsTC_utility_delta`,

Utility delta

```
function SsTC_utility_delta(x) result(res)

  real(kind=dp), intent(in) :: x
  real(kind=dp) :: res

end function SsTC_utility_delta
```

Listing 27: Interface of "utility delta".

and the function `SsTC_utility_delta_vec`,

Utility vector delta

```
function SsTC_utility_delta_vec(x) result(res)

  real(kind=dp), intent(in) :: x(:)
  real(kind=dp) :: res(size(x))

end function SsTC_utility_delta_vec
```

Listing 28: Interface of "utility delta vector".

will approximate the Dirac delta distribution $\delta(x)$ and $\delta(\boldsymbol{x})$ respectively.
The function `SsTC_utility_get_degen`,

Utility get degeneracy

```
function SsTC_utility_get_degen(eig, degen_thr) result(deg)

  real(kind=dp), intent(in)  :: degen_thr
  real(kind=dp), intent(in)  :: eig(:)

  integer :: deg(size(eig))

end function SsTC_utility_get_degen
```

Listing 29: Interface of "utility get degeneracy of lists".

will take as an input an ascending ordered list `eig` of real elements and return an integer valued list `deg` of the same size of `eig`, with the degree of degeneracy of each element in `eig` stored in the respective element of `deg` as determined by the threshold `degen_thr`. The values each element of `deg` can have, are the following,

- `deg(i)` $= 1$: Nondegenerate subspace.
- `deg(i)` $= N > 1$: Degenerate subspace with degree of degeneracy N, the next $N - 1$ elements of the list have the value 0.
- `deg(i)` $= 0$: Level belongs to a degenerate subspace.

The routine `SsTC_utility_diagonalize`,

Utility diagonalize

```
subroutine SsTC_utility_diagonalize(mat, dim, eig, rot, error)

  integer, intent(in)           :: dim
  complex(kind=dp), intent(in)  :: mat(dim, dim)
  real(kind=dp), intent(out)    :: eig(dim)      !Eigenvalues.
  complex(kind=dp), intent(out) :: rot(dim, dim) !Eigenvectors.
  logical, intent(inout)        :: error

end subroutine SsTC_utility_diagonalize
```

Listing 30: Interface of "utility diagonalize".

computes the diagonalization of the Hermitian `dim` $\times$ `dim` matrix `mat`. It returns the eigenvalues `eig` sorted in ascending order and the unitary rotation `rot`. If an error occurs in the calculation, `error=.true.` is set.
The routine `SsTC_utility_schur`,

Utility Schur

```
subroutine SsTC_utility_schur(mat, dim, T, Z, error, S)

  integer, intent(in)                    :: dim
  complex(kind=dp), intent(in)           :: mat(dim, dim)
  complex(kind=dp), intent(out)          :: T(dim)       !Eigenvalues.
  complex(kind=dp), intent(out)          :: Z(dim, dim) !Eigenvectors.
  logical, intent(inout)                 :: error
  complex(kind=dp), intent(out), optional :: S(dim, dim) !Schur Form.

end subroutine SsTC_utility_schur
```

Listing 31: Interface of "utility Schur".

computes the Schur decomposition of the complex `dim` × `dim` matrix `mat`. It returns the eigenvalues `T` and the unitary rotation `Z`. If an error occurs in the calculation, `error=.true.` is set. Optionally, it returns the Schur form `S`, obeying $\texttt{mat} = \texttt{Z} * \texttt{S} * \texttt{Z}^{\dagger}$.

The routine `SsTC_utility_SVD`,

Utility SVD

```
subroutine SsTC_utility_SVD(mat, sigma, error, U, V)

  complex(kind=dp), intent(in)           :: mat(:, :)
  real(kind=dp), intent(out)             :: sigma(size(mat(:, 1)), &
                                            size(mat(1, :)))
  logical, intent(inout)                 :: error
  complex(kind=dp), intent(out), optional :: U(size(mat(:, 1)), &
                                            size(mat(:, 1))), &
                                            V(size(mat(1, :)), &
                                            size(mat(1, :)))

end subroutine SsTC_utility_SVD
```

Listing 32: Interface of "utility SVD".

computes the singular value decomposition of the complex rectangular matrix `mat`. It returns the eigenvalues `sigma`. If an error occurs in the calculation, `error=.true.` is set. Optionally, it returns and the unitary rotations `U`, `V` obeying $\texttt{mat} = \texttt{U} * \texttt{sigma} * \texttt{V}^{\dagger}$.

The function `SsTC_utility_exphs`,

Utility (anti)Hermitian matrix exponential

```
function SsTC_utility_exphs(mat, dim, skew, error) result(exphs)

  integer, intent(in)          :: dim
  complex(kind=dp), intent(in) :: mat(dim, dim)
  logical, intent(in)          :: skew
  logical, intent(inout)       :: error

  complex(kind=dp) :: exphs(dim, dim)

end function SsTC_utility_exphs
```

Listing 33: Interface of "utility exponential of matrix".

computes the matrix exponential of the (skew)Hermitian `dim` × `dim` matrix `mat`. The (skew)Hermiticity of

`mat` is specified by `skew`, which is `.false.` for a Hermitian `mat` and `.true.` for a skew Hermitian `mat`. It returns `exphs` $= e^{\texttt{mat}}$. If an error occurs in the calculation, `error=.true.` is set.

The function `SsTC_utility_logu`,

Utility unitary matrix logarithm

```
function SsTC_utility_logu(mat, dim, error) result(logu)

  integer, intent(in)          :: dim
  complex(kind=dp), intent(in) :: mat(dim, dim)
  logical, intent(inout)       :: error

  complex(kind=dp) :: logu(dim, dim)

end function SsTC_utility_logu
```

Listing 34: Interface of "utility logarithm of matrix".

computes the matrix logarithm of the unitary `dim` × `dim` matrix `mat`. It returns `logu` = log (`mat`). If an error occurs in the calculation, `error=.true.` is set.

8.2 Extrapolation integration module

Extrapolation integration is a sub-module of the SsTC project, powered by the `F90-Extrapolation-Integration` project, hosted on GitHub. We refer the reader to the user's guide provided by the repository for a full review of the routines provided by the module.

In SsTC, the routines are made public to the user with the following renaming.

```
SsTC_integral_extrapolation => integral_extrapolation, &
          SsTC_shrink_array => shrink_array, &
          SsTC_expand_array => expand_array
```

Listing 35: Renaming of extrapolation routines.

8.3 Data structures module

The module data structures provides utility, array transformation, and index tracking routines motivated by the comfort of storing data local for each $\boldsymbol{k}$ in array layout and the flexibility of memory layout in array passing, as discussed in Sec. 5.

8.3.1 Utility

The derived type `SsTC_external_vars`,

External variable data type

```
type SsTC_external_vars
  real(kind=dp), allocatable :: data(:) !External variable data array.
end type SsTC_external_vars
```

Listing 36: Derived type "external variables".

stores the external variable data λ_{ij} in Eq. (2) for each external variable β_i. The data is created by the constructor,

External variable data constructor

```
function SsTC_external_variable_constructor(start, end, steps) &
```

```
  result(vars)
    !Function to set external variable data.
    real(kind=dp), intent(in) :: start, end
    integer, intent(in)       :: steps

    type(SsTC_external_vars) :: vars

  end function SsTC_external_variable_constructor
```

Listing 37: Interface of "external variable constructor".

which implements Eq. (2) for an index β_i.

8.3.2 Array transformation and index tracking

The function `SsTC_integer_array_element_to_memory_element`,

Array element to memory element (integer indices)

```
function SsTC_integer_array_element_to_memory_element(data_k, i_arr) &
         result(i_mem)

  !Get integer indices from array layout to memory layout.
  class(SsTC_local_k_data), intent(in) :: data_k
  integer, intent(in)                  :: i_arr(size(data_k%integer_indices))

  integer :: i_mem

end function SsTC_integer_array_element_to_memory_element
```

Listing 38: Interface of "integer array element to memory element".

computes the integer index $r = f(\{n_1, n_2, \cdots, n_N\})$ in Eq. (5) from the indices $\{n_1, \cdots, n_N\}$ specified by `i_arr`. The sizes s_i are given by `data_k%integer_indices(i)`.

The function `SsTC_integer_memory_element_to_array_element`,

Memory element to array element (integer indices)

```
function SsTC_integer_memory_element_to_array_element(data_k, i_mem) &
         result(i_arr)

  !Get integer indices from memory layout to array layout.
  class(SsTC_local_k_data), intent(in) :: data_k
  integer, intent(in)                  :: i_mem

  integer :: i_arr(size(data_k%integer_indices))

end function SsTC_integer_memory_element_to_array_element
```

Listing 39: Interface of "integer memory element to array element".

computes the integer indexes $\{n_1, n_2, \cdots, n_N\} = f^{-1}(r)$ in Eq. (5) from the index r. The sizes s_i are given by `data_k%integer_indices(i)`.

The function `SsTC_continuous_array_element_to_memory_element`,

Array element to memory element (continuous indices)

```
function SsTC_continuous_array_element_to_memory_element(task, r_arr) &
         result(r_mem)

  !Get continuous indices from array layout to memory layout.
  class(SsTC_global_k_data), intent(in) :: task
  integer, intent(in)                   :: r_arr(size(task%continuous_indices))

  integer :: r_mem

end function SsTC_continuous_memory_element_to_array_element
```

Listing 40: Interface of "continuous array element to memory element".

computes the continuous index $r = f(\{n_1, n_2, \cdots, n_N\})$ in Eq. (5) from the indices $\{n_1, \cdots, n_N\}$ specified by `r_arr`. The sizes s_i are given by `task%continuous_indices(i)`.

The function `SsTC_continuous_memory_element_to_array_element`,

Memory element to array element (continuous indices)

```
function SsTC_continuous_memory_element_to_array_element(task, r_mem) &
         result(r_arr)

  !Get continuous indices from memory layout to array layout.
  class(SsTC_global_k_data), intent(in) :: task
  integer, intent(in)                   :: r_mem

  integer :: r_arr(size(task%continuous_indices))

end function SsTC_continuous_memory_element_to_array_element
```

Listing 41: Interface of "continuous array element to memory element".

computes the continuous indexes $\{n_1, n_2, \cdots, n_N\} = f^{-1}(r)$ in Eq. (5) from the index r. The sizes s_i are given by `task%continuous_indices(i)`.

The routine `SsTC_construct_iterable`

Construct iterable

```
subroutine SsTC_construct_iterable(global, vars)
  class(SsTC_global_k_data), intent(inout) :: global
  integer, intent(in)                      :: vars(:)
end subroutine SsTC_construct_iterable
```

Listing 42: Interface of "construct iterable".

creates an "iterable" dictionary which is stored in `global%iterables(:,:)`. The dictionary holds all possible combinations from the variation of the β_i-indexed coordinate labels which are specified in the array `vars`, i.e., `vars` is an array which each entry specifying the i label of β_i to permute. The first index of the dictionary holds the "permutation label" associated with variation of the variables `vars` and the second the particular permutation of the variables, so the set of arrays $\texttt{array}_n$ = `global%iterables(n,:)` hold a list of size `size(global%continuous_indices)` with a particular permutation in the indices `vars(j)` for each $j \in$ `size(vars)`. The variables not permutated are stored with the constant value of 1 in the corresponding entry of `global%iterables(:,:)`.

8.4 Local k-quantities module

The local $\boldsymbol{k}$ quantities module provides utilities which implement Wannier interpolation [1] of several common physically meaningful quantities by computing the Fourier transformation of a `type(SsTC_sys)` system's Hamiltonian and position matrix elements from direct space to reciprocal space.

8.4.1 Definitions: fundamentals of Wannier interpolation

The matrix elements of an operator $\hat{O}$ in the Wannier basis $\{|n\boldsymbol{R}\rangle\}$ are

$$O_{nm}(\boldsymbol{R}) = \langle n\boldsymbol{0}|\hat{O}|m\boldsymbol{R}\rangle\,. \tag{23}$$

The Fourier transform of those matrix elements

$$O^{(W)}_{nm}(\boldsymbol{k}) = \sum_{\boldsymbol{R}} e^{i\boldsymbol{k}\cdot\boldsymbol{R}} O_{nm}(\boldsymbol{R}) \tag{24}$$

are called the matrix elements of an operator $\hat{O}$ in the Bloch basis and in the Wannier gauge [1, 4, 5]. The Bloch basis in the Wannier gauge is a reciprocal space basis $\{|n\boldsymbol{k}\rangle^{(W)}\}$ which obeys

$$|n\boldsymbol{k}\rangle^{(W)} = \sum_{\boldsymbol{R}} e^{i\boldsymbol{k}\cdot\boldsymbol{R}}\,|n\boldsymbol{R}\rangle \tag{25}$$

It is important to note that the Bloch basis in the Wannier gauge does not, in general, make any operator, such as the Hamiltonian $\hat{H}$, diagonal. It rather provides a localized basis which makes it possible to interpolate most physical quantities with precision. The Bloch basis which diagonalizes the Hamiltonian $\hat{H}$ is called the Bloch basis in the Hamiltonian gauge, $\{|n\boldsymbol{k}\rangle^{(H)}\}$ which obeys

$$\hat{H}\,|n\boldsymbol{k}\rangle^{(H)} = \varepsilon_n(\boldsymbol{k})\,|n\boldsymbol{k}\rangle^{(H)}\,, \quad |n\boldsymbol{k}\rangle^{(H)} = \sum_m U_{mn}(\boldsymbol{k})\,|n\boldsymbol{k}\rangle^{(W)} \tag{26}$$

for some unitary operator $\hat{U}$ with matrix elements $U_{mn}(\boldsymbol{k})$ and we call $\varepsilon_n(\boldsymbol{k})$ the eigenvalues of $\hat{H}$. Both reciprocal space bases W and H are related by unitary transformations local for each $\boldsymbol{k}$ [4].

The Fourier transform of the Berry connection in the Bloch basis and in the Wannier gauge is given by [4]

$$\boldsymbol{A}^{(W)}_{nm}(\boldsymbol{k}) = \sum_{\boldsymbol{R}} e^{i\boldsymbol{k}\cdot\boldsymbol{R}} \boldsymbol{r}_{nm}(\boldsymbol{R}) =^{(W)} \langle n\boldsymbol{k}|\boldsymbol{\nabla}|m\boldsymbol{k}\rangle^{(W)}\,, \tag{27}$$

which is called the Berry connection in the Bloch basis and in the Wannier gauge. The matrix representing the Berry connection in the Bloch basis and in the Hamiltonian gauge is related to the matrix representing the Berry connection in the Wannier gauge by [4],

$$\boldsymbol{A}^{(H)}(\boldsymbol{k}) = U^{\dagger}(\boldsymbol{k})\boldsymbol{A}^{(W)}(\boldsymbol{k})U(\boldsymbol{k}) + i\boldsymbol{D}^{(H)}(\boldsymbol{k}), \tag{28}$$

where $U(\boldsymbol{k})$ is the matrix representing the unitary operator $\hat{U}$ with matrix elements $U_{mn}(\boldsymbol{k})$ in Eq. (26) and $\boldsymbol{D}^{(H)}$ has matrix elements

$$D^{a(H)}_{nm}(\boldsymbol{k}) = \begin{cases} \frac{\left[U^{\dagger}(\boldsymbol{k})\left(\nabla^a H^{(W)}(\boldsymbol{k})\right)U(\boldsymbol{k})\right]_{nm}}{\varepsilon_m(\boldsymbol{k})-\varepsilon_n(\boldsymbol{k})}, & \varepsilon_m(\boldsymbol{k}) \neq \varepsilon_n(\boldsymbol{k}), \\ 0, & \varepsilon_m(\boldsymbol{k}) = \varepsilon_n(\boldsymbol{k}). \end{cases} \tag{29}$$

8.4.2 Procedures

The module contains 2 core procedures, which Fourier transform the elements of the Hamiltonian and Berry connection read from `real_space_hamiltonian_elements(:, :, :)` and `real_space_position_elements(:, :, :, :)` respectively to the Bloch basis in the Wannier gauge. This implements Eq. (24) for both operators and makes it possible to calculate also its $\boldsymbol{k}$ derivatives.

The routine `SsTC_get_hamiltonian`,

Get Hamiltonian

```
subroutine SsTC_get_hamiltonian(system, k, H, Nder_i, only_i)

  type(SsTC_sys), intent(in)                          :: system
```

```
  real(kind=dp), intent(in)                              :: k(3)
  type(SsTC_local_k_data), allocatable, intent(out) :: H(:)
  integer, optional                                      :: Nder_i
  !Order of the derivative. Nder = 0 means normal Hamiltonian,
  !Nder = 1, \partial H/\partial k^i...
  logical, optional                                      :: only_i
  !Determine, in the case of Nder > 0,
  !if all derivatives i with i < Nder are requested.

end subroutine SsTC_get_hamiltonian
```

Listing 43: Interface of "get Hamiltonian".

computes the `Nder_i` $= nD$-th Hamiltonian derivative, retrieving the Bloch basis in the Wannier gauge matrix elements

$$\texttt{H(i)\%k\_data(r)} = \frac{\partial^{nD}}{\partial k^{a_1}\cdots\partial k^{a_{nD}}} H^{(W)}_{nm}(\boldsymbol{k}), \tag{30}$$

where the set of band and derivative indices $\{n, m, a_1, \cdots, a_{nD}\}$ are stored in memory layout in the index `r`, for a system `system` and $\boldsymbol{k}$ point `k`. If $nD = 0$, the regular Hamiltonian is stored. The index `i` in Eq. (30) references the $i+1$-th derivative of the Hamiltonian in the case of `only_i = .false.` and $nD > 1$. If `only_i = .true.`, the routine will only compute the nD-th Hamiltonian derivative and store it in `H(1)`. The units of the nD-th Hamiltonian derivative are $\text{eV}\text{Å}^{nD}$.

The routine `SsTC_get_position`,

Get position

```
subroutine SsTC_get_position(system, k, A, Nder_i, only_i)

  type(SsTC_sys), intent(in)                              :: system
  real(kind=dp), intent(in)                               :: k(3)
  type(SsTC_local_k_data), allocatable, intent(out) :: A(:)
  integer, optional                                       :: Nder_i
  !Order of the derivative. Nder = 0 means normal Berry connection,
  !Nder = 1, \partial A^i/\partial k^j...
  logical, optional                                       :: only_i
  !Determine, in the case of Nder > 0,
  !if all derivatives i with i < Nder are requested.

end subroutine SsTC_get_position
```

Listing 44: Interface of "get position".

computes the `Nder_i` $= nD$-th Berry connection's derivative, retrieving the Bloch basis in the Wannier gauge matrix elements

$$\texttt{A(i)\%k\_data(r)} = \frac{\partial^{nD}}{\partial k^{a_1}\cdots\partial k^{a_{nD}}} A^{a(W)}_{nm}(\boldsymbol{k}), \tag{31}$$

where the set of band and derivative indices $\{n, m, a, a_1, \cdots, a_{nD}\}$ are stored in memory layout in the index `r`, for a system `system` and $\boldsymbol{k}$ point `k`. If $nD = 0$, the regular Berry connection is stored. The index `i` in Eq. (31) references the $i+1$-th derivative of the Berry connection in the case of `only_i = .false.` and $nD > 1$. If `only_i = .true.`, the routine will only compute the nD-th Berry connection derivative and store it in `A(1)`. The units of the nD-th Berry connection derivative are Å^{nD+1}.

Next, we describe some routines which offer short cuts to calculate the firsts derivatives of both the Hamiltonian and position operator matrix elements in the Bloch basis and in the Wannier gauge.

The function `SsTC_wannier_hamiltonian`,

Wannier Hamiltonian

```
function SsTC_wannier_hamiltonian(system, k) result(HW)

  !Output: Wannier basis Hamiltonian.
  !1st and 2nd indexes: bands.
  type(SsTC_sys), intent(in) :: system
  real(kind=dp), intent(in)  :: k(3)

  complex(kind=dp) :: HW(system%num_bands, system%num_bands)

end function SsTC_wannier_hamiltonian
```

Listing 45: Interface of "Wannier Hamiltonian".

computes

$$\texttt{HW(n, m)} = H^{(W)}_{nm}(\boldsymbol{k}) = \sum_{\boldsymbol{R}} e^{i\boldsymbol{k}\cdot\boldsymbol{R}} H_{nm}(\boldsymbol{R}), \tag{32}$$

for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of eV.

The function `SsTC_wannier_berry_connection`,

Wannier Berry connection

```
function SsTC_wannier_berry_connection(system, k) result(AW)

  !Output: Wannier basis Berry connection.
  !1st and 2nd indexes: bands, 3rd index: cartesian comp.
  type(SsTC_sys), intent(in) :: system
  real(kind=dp), intent(in)  :: k(3)

  complex(kind=dp) :: AW(system%num_bands, system%num_bands, 3)

end function SsTC_wannier_berry_connection
```

Listing 46: Interface of "Wannier Berry connection".

computes

$$\texttt{AW(n, m, a)} = A^{a(W)}_{nm}(\boldsymbol{k}) = \sum_{\boldsymbol{R}} e^{i\boldsymbol{k}\cdot\boldsymbol{R}} r^{a}_{nm}(\boldsymbol{R}), \tag{33}$$

for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of Å.

The function `SsTC_wannier_dhamiltonian_dk`,

Wannier Hamiltonian derivative

```
function SsTC_wannier_dhamiltonian_dk(system, k) result(DHW)

  !Output: 1st k-derivative of the Wannier Hamiltonian.
  !1st and 2nd indexes: bands, 3rd index: derivative comp.
  type(SsTC_sys), intent(in) :: system
  real(kind=dp), intent(in)  :: k(3)

  complex(kind=dp) :: DHW(system%num_bands, system%num_bands, 3)

end function SsTC_wannier_dhamiltonian_dk
```

Listing 47: Interface of "Wannier Hamiltonian's derivative".

computes

$$\texttt{HWA(n, m, a)} = \frac{\partial H^{(W)}_{nm}(\boldsymbol{k})}{\partial k^a} = \sum_{\boldsymbol{R}} (iR^a)\, e^{i\boldsymbol{k}\cdot\boldsymbol{R}} H_{nm}(\boldsymbol{R}), \tag{34}$$

for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of eVÅ.

The function `SsTC_wannier_d2hamiltonian_dk2`,

Wannier Hamiltonian second derivative

```
function SsTC_wannier_d2hamiltonian_dk2(system, k) result(DDHW)

  !Output: 2nd k-derivative of the Wannier Hamiltonian.
  !1st and 2nd indexes: bands,
  !3rd and 4th indexes: derivative directions.
  type(SsTC_sys), intent(in) :: system
  real(kind=dp), intent(in)  :: k(3)

  complex(kind=dp) :: DDHW(system%num_bands, system%num_bands, 3, 3)

end function SsTC_wannier_d2hamiltonian_dk2
```

Listing 48: Interface of "Wannier Hamiltonian's 2nd derivative".

computes

$$\texttt{HWA(n, m, a, b)} = \frac{\partial^2 H^{(W)}_{nm}(\boldsymbol{k})}{\partial k^a \partial k^b} = \sum_{\boldsymbol{R}} \left(-R^a R^b\right) e^{i\boldsymbol{k}\cdot\boldsymbol{R}} H_{nm}(\boldsymbol{R}), \tag{35}$$

for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of eVÅ^2.

The function `SsTC_wannier_dberry_connection_dk`,

Wannier Berry connection derivative

```
function SsTC_wannier_dberry_connection_dk(system, k) result(DAW)

  !Output: 1st k-derivative of the Wannier Berry connection.
  !1st and 2nd indexes: bands, 3rd index: cartesian comp.
  !4th index : derivative direction
  type(SsTC_sys), intent(in) :: system
  real(kind=dp), intent(in)  :: k(3)

  complex(kind=dp) :: DAW(system%num_bands, system%num_bands, 3, 3)

end function SsTC_wannier_dhamiltonian_dk
```

Listing 49: Interface of "Wannier Berry connection's derivative".

computes

$$\texttt{DAW(n, m, a, b)} = \frac{\partial A^{a(W)}_{nm}(\boldsymbol{k})}{\partial k^b} = \sum_{\boldsymbol{R}} \left(iR^b\right) e^{i\boldsymbol{k}\cdot\boldsymbol{R}} r^a_{nm}(\boldsymbol{R}), \tag{36}$$

for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of Å^2.

Finally, we describe some functions involving the implementation of quantities which are commonly employed in many physical quantities.

The function `SsTC_wannier_momentum`,

Wannier momentum

```
function SsTC_wannier_momentum(system, k) result(PW)

  type(SsTC_sys), intent(in) :: system
```

```
  real(kind=dp), intent(in)  :: k(3)

  complex(kind=dp) :: PW(system%num_bands, system%num_bands, 3)

end function SsTC_wannier_momentum
```

Listing 50: Interface of "Wannier momentum".

computes [6, 7]

$$\texttt{PW(n, m, a)} = p_{nm}^{a(W)} = \frac{m}{i\hbar}\left[\hat{r}_i, \hat{H}\right]_{nm}^{(W)} = \frac{m}{\hbar}\left(\frac{\partial H_{nm}^{(W)}(\boldsymbol{k})}{\partial k^a} - i\left[A^{a(W)}, H^{(W)}\right]_{nm}\right), \tag{37}$$

where $A^{a(W)}$ and $H^{(W)}$ are the matrices with Berry connection and Hamiltonian matrix elements, respectively, for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of eVÅ^2.

The function `SsTC_hamiltonian_occ_matrix`,

Hamiltonian occupation matrix

```
function SsTC_hamiltonian_occ_matrix(system, eig) result(rho)

  !Output: Fermi occupation matrix in the Hamiltonian basis.
  !1st and 2nd indexes: bands.
  type(SsTC_sys), intent(in) :: system
  real(kind=dp), intent(in)  :: eig(system%num_bands)

  complex(kind=dp) :: rho(system%num_bands, system%num_bands)

end function SsTC_hamiltonian_occ_matrix
```

Listing 51: Interface of "Hamiltonian occupation matrix".

computes

$$\texttt{rho(n, m)} = f_{nm}^{(H)}(\boldsymbol{k}) = \begin{cases} 1, & \text{if } n = m \text{ and } \varepsilon_n(\boldsymbol{k}) < \varepsilon_F, \\ 0, & \text{else,} \end{cases} \tag{38}$$

where ε_F if the Fermi energy of the system, given by `system%e_fermi`, for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$.

The function `SsTC_non_abelian_d`,

Non-abelian D matrix

```
function SsTC_non_abelian_d(system, eig, rot, HW_a) result(DH)

  !Output: Vector valued matrix D in Eq. (32) in
  !10.1103/PhysRevB.75.195121 .
  !1st and 2nd indexes: bands, 3rd index: cartesian comp.
  type(SsTC_sys), intent(in)    :: system
  real(kind=dp), intent(in)     :: eig(system%num_bands)
  complex(kind=dp), intent(in) :: rot(system%num_bands, system%num_bands)
  complex(kind=dp), intent(in) :: HW_a(system%num_bands, system%num_bands, 3)

  complex(kind=dp) :: DH(system%num_bands, system%num_bands, 3)

end function SsTC_non_abelian_d
```

Listing 52: Interface of "D matrix".

computes Eq. (29), for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of Å.

The function `SsTC_deleig`,

Eigenvalue derivative

```
function SsTC_deleig(system, HW_a, eig, rot, error) result(v)

  !Output: Velocities v_{nm, a} in Eq. (23) in
  !10.1103/PhysRevB.75.195121 .
  !1st and 2nd indexes: bands, 3rd index: cartesian comp.
  type(SsTC_sys), intent(in)    :: system
  real(kind=dp), intent(in)     :: eig(system%num_bands)
  complex(kind=dp), intent(in) :: rot(system%num_bands, system%num_bands)
  complex(kind=dp), intent(in) :: HW_a(system%num_bands, system%num_bands, 3)
  logical, intent(inout)        :: error

  complex(kind=dp) :: v(system%num_bands, system%num_bands, 3)

end function SsTC_deleig
```

Listing 53: Interface of "Derivative of eigenvalues".

computes

$$\texttt{v(n, m, a)} = \frac{\partial H^{(H)}_{nm}(\boldsymbol{k})}{\partial k^a}, \tag{39}$$

as in Eq. (26) of Ref. [5] for multi-band case with degeneracy correction, for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of eVÅ. The diagonal elements correspond to the band gradients

$$\texttt{v(n, n, a)} = \frac{\partial \varepsilon_n(\boldsymbol{k})}{\partial k^a}. \tag{40}$$

If some error is encountered during diagonalization, `error = .true.` is set.

The function `SsTC_inverse_effective_mass`,

Eigenvalue second derivative or inverse effective mass

```
function SsTC_inverse_effective_mass(system, HW_a_b, HW_a, eig, rot, error) result(mu)

  !Output: Inverse effective mass \mu_{nm, ab} in Eq. (24) in
  !10.1103/PhysRevB.75.195121 .
  !1st and 2nd indexes: bands, 3rd and 4th index: cartesian comp.
  type(SsTC_sys), intent(in)    :: system
  real(kind=dp), intent(in)     :: eig(system%num_bands)
  complex(kind=dp), intent(in) :: rot(system%num_bands, system%num_bands)
  complex(kind=dp), intent(in) :: HW_a_b(system%num_bands, system%num_bands, 3, 3)
  complex(kind=dp), intent(in) :: HW_a(system%num_bands, system%num_bands, 3)
  logical, intent(inout)        :: error

  complex(kind=dp) :: mu(system%num_bands, system%num_bands, 3, 3)

end function SsTC_inverse_effective_mass
```

Listing 54: Interface of "Inverse effective mass".

computes

$$\texttt{mu(n, m, a, b)} = \frac{\partial H^{(H)}_{nm}(\boldsymbol{k})}{\partial k^a \partial k^b}, \tag{41}$$

as in Eq. (28) of Ref. [5] for multi-band case with degeneracy correction, for system `system` and $\boldsymbol{k}$-point $\texttt{k} = \boldsymbol{k}$ in units of eVÅ^2. The diagonal elements correspond to the inverse effective masses,

$$\texttt{mu(n, n, a, b)} = \frac{\partial^2 \varepsilon_n(\boldsymbol{k})}{\partial k^a \partial k^b}. \tag{42}$$

If some error is encountered during diagonalization, `error = .true.` is set.

9 Usage in high performance computing

SsTC uses a hybrid parallelization model, using the MPI [2] and the OpenMP [8] libraries. The intended use of hybrid parallelization is to allow the MPI library to distribute (coarse grained) chunks of $\boldsymbol{k}$ points across different MPI ranks in sampling tasks. Then, each rank can make use of multi-threading to further distribute chunks of $\boldsymbol{k}$ points across threads and SIMD units in a shared memory scheme.

In the following, we consider an application `appl.x` making use of SsTC, which has been compiled and linked as described in Sec. 2.

9.1 Example of a SLURM job

SLURM is a widely used workload manager in computational clusters. The following file `run.sh` located in the same directory as `appl.x`,

SLURM script

```
#!/bin/bash -x
#SBATCH -J APPL
###Number of nodes.
#SBATCH --nodes=4
###Number of tasks.
#SBATCH --ntasks=4
###Number of tasks per node.
#SBATCH --ntasks-per-node=1
###Number of threads per task.
#SBATCH --cpus-per-task=48
#SBATCH -e error-%j.err
#SBATCH -o output-%j.out
#SBATCH --mail-type=all
#SBATCH --partition=128gb
#SBATCH --time=480:00:00
#SBATCH --mem=126000

ml purge
ml intel/2022a

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

mpirun -np $SLURM_NTASKS ./appl.x
```

Listing 55: SLURM script "run".

will run `appl.x` in a total of 4×48 threads, distributed in 4 nodes.

10 Modularity

SsTC features a modularity capability for users to extend the library by their own modifications. These modifications, or "mods", allow users to create and maintain modules featuring further derived type extension and creation of custom calculators.

Let's consider a SsTC user who implemented, by using the basic SsTC library's routines, a set of calculators and task constructors involving various physically meaningful quantities by extending the basic integrator/sampler/kslice/kpath tasks to satisfy his/her own needs. For example, a set of optical photocurrent tensor calculations contained in a module. In order to distribute the implemented set of utilities, the user would have to create and maintain a repository containing SsTC within its dependencies or create pull requests to the SsTC repository. However, with the modularity feature, this is unnecessary: the user is only responsible for maintaining his/her own calculator implementing module, and in compile time, the utilities and declarations provided in the module get added to the SsTC library's scope and made public to the user. This creates a modified version of SsTC, a "mod". The SsTC user who created the mod has only to instruct other users to use SsTC (perhaps some particular version) and add his/hers "mod" with the aid of the "mod loader", whose

working principle is described in Sec. 10.2. An example of this capability is shown in Example 2 in Sec. 11.2, where we instruct on how to add capabilities to SsTC for the calculation of an optical response.

10.1 Structure of a mod

A modification is a user made and maintained set of files contained in a folder, containing at least the following,

- One or more source code Fortran module files with the extension `*.F90`, containing the set of mentioned user defined declarations, utilities, or calculators. These files should have the structure,

  User made module file

```
module mymod

  use SsTC_utility
  use SsTC_...      !All the SsTC dependencies required by this module.

  use my_other_mods !Any other modules.

  implicit none

  private

  public :: some_thing_I_implemented

  public :: ...

  ... !Declarations

contains

  ... !Procedures

end module mymod
```

  Listing 56: Structure of user made "mod" files.

  where the user is free to modify everything provided by the basic SsTC library or add any other capabilities.

- A header file named `headers.inc` containing the Fortran "use" statement required to load the provided Fortran module, for example, in the case of a module file like the one above,

  Module header file

```
  use mymod
```

  Listing 57: Module header file.

  will be the header file.

- A procedure header file named `procedures.inc` containing the Fortran "public" statement required to make a set of declarations public. In the case of a module file like the one above,

Module procedure header file

```
public :: some_thing_I_implemented

public :: ...
```

Listing 58: Module procedure header file.

will be the header file.

- A Makefile with any name, but with extension `*.mk`. It is recommended that the name is something related with the mod theme. The Makefile needs to contain the set of instructions needed to compile the source code contained in the `*.F90` files in a standard make language fashion. The name of the dependencies is the following,

  - `utility.o` if the module file `*.F90` has `use SsTC_utility`.
  - `extrapolation_integration.o` if the module file `*.F90` has `use extrapolation_integration`.
  - `data_structures.o` if the module file `*.F90` has `use SsTC_data_structures`.
  - `kpath.o` if the module file `*.F90` has `use SsTC_kpath`.
  - `kslice.o` if the module file `*.F90` has `use SsTC_kslice`.
  - `sampler.o` if the module file `*.F90` has `use SsTC_sampler`.
  - `integrator.o` if the module file `*.F90` has `use SsTC_integrator`.
  - `local_k_quantities.o` if the module file `*.F90` has `use SsTC_local_k_quantities`.

  In the case of a module file like the one above the Makefile should have the structure,

```
mymod.o: $(CALC)/mymod/mymod.F90 utility.o ... #All other deps
         $(F90) $(F90FLAGS) -c $(CALC)/mymod/mymod.F90 -o "$(OBJ)/mymod.o"

DEPS += mymod.o
```

Listing 59: Makefile.

  where we are assuming a single source code file `mymod.F90` and the variables `CALC`, `F90`, `F90FLAGS` and `OBJ` are given in the main Makefile at `bash:/path/of/your/choice/SsTC`.

10.2 The mod loader

The "mod loader" works by searching for `*.mk` files in the subdirectories of the variable `CALC`, which is given by `bash:/path/of/your/choice/SsTC/src/calculators`. Once a Makefile is found, its contents get included in the main Makefile at `bash:/path/of/your/choice/SsTC`. When running make for building SsTC, after the new dependencies are included, a `python` script is executed, which finds the `headers.inc` and `procedures.inc` files associated with each found Makefile, creates a copy of the source code `SsTC.F90`, `SsTC_mod.F90` and includes the contents of the `*.inc` files in the new source code. Then, the source code is compiled to a library `libSsTC.a`, thus creating a "mod" of SsTC.

For the mod loader to load the mod, the folder containing the files listed in Sec. 10.1 needs to be copied to `bash:/path/of/your/choice/SsTC/src/calculators` and then run

```
bash:/path/of/your/choice/SsTC$ make
```

If mod Makefiles are detected, the make log will show the lines

```
Detected mod Makefiles are [./src/calculators/mymod/makefile.mk ...].
...
Including header file: src/calculators/mymod/headers.inc
Including procedure list file: src/calculators/mymod/procedures.inc
...
```

for every detected Makefile.

11 Examples

In this section, we consider two examples of creation and sampling of tasks involving the integrator module. The first one involves the formal example of the integral of a user defined function in the BZ. The second one, involves the calculation of the jerk current of GaAs by using a custom modification for SsTC.

11.1 Example 1: Integral of user defined functions

This example is intended to show the capabilities of SsTC when considering the toy problem of a calculator depending on a user defined number of integer and continuous indices. The code is contained in the directory

```
bash:/path/of/your/choice/SsTC/doc/examples/example1$
```

In the directory are 3 files,

```
bash:/path/of/your/choice/SsTC/doc/examples/example1$ ls
compile.sh  dummy_tb.dat  example1.F90
```

`compile.sh` contains a compilation script for the source code `example1.F90`. The file `dummy_tb.dat` contains the tight-binding model of a dummy system without any interactions.

The source code declares the dummy system, initializes MPI and SsTC and loads the dummy system into the program. Then, an integration task is declared and constructed by using the integrator task constructor.

We specify the name, the function implementing the calculator $C^{\boldsymbol{\alpha}}(\boldsymbol{k};\boldsymbol{\beta})$, the integration method, the number of samples, the number of integer indices encompassed by $\boldsymbol{\alpha}$, the ranges of each of them, the number of external variables encompassed by $\boldsymbol{\beta}$ and the starting, ending and number of samples for each of the external variables.

The function implementing the calculator is given by `test_calculator`, defined within the `contains` section of the source code, it represents the calculator

$$C^{ij}(\boldsymbol{k},a) = (i+j)e^{ak_1}, i,j \in [1,3] \subset \mathbb{Z}, a \in [1,10] \subset \mathbb{R}. \tag{43}$$

the sampling and integration task states that we want to calculate

$$R^{ij}(a) = \int_{\mathrm{BZ}} [d\boldsymbol{k}]\, C^{ij}(\boldsymbol{k},a) = \int_{-0.5}^{0.5} dk_1\ (i+j)e^{ak_1} = (i+j)\frac{e^{a/2}-e^{-a/2}}{a}. \tag{44}$$

by using the rectangle approximation and 33 samples in the direction of the coordinate k_1.

The sampling and integration are then performed by the sampling and integration subroutine. This writes the values of $R^{ij}(a)$ to `test_integral%result(:, :)` in memory layout. Lastly the results are printed by the printing subroutine and stored in the 3×3 files `dummy-example01-test_*.dat`.

Compiling & running gives

```
bash:/path/of/your/choice/SsTC/doc/examples/example1$ ./compile.sh
bash:/path/of/your/choice/SsTC/doc/examples/example1$ mpirun -np 1 ./test.x
bash:/path/of/your/choice/SsTC/doc/examples/example1$ cat dummy-example01-test_11.dat
   0.10000000E+001   0.20897236E+001   0.00000000E+000
   0.20000000E+001   0.23734400E+001   0.00000000E+000
   0.30000000E+001   0.28975941E+001   0.00000000E+000
   0.40000000E+001   0.37495456E+001   0.00000000E+000
   0.50000000E+001   0.50746909E+001   0.00000000E+000
   0.60000000E+001   0.71053255E+001   0.00000000E+000
```

```
   0.70000000E+001   0.10207415E+002   0.00000000E+000
   0.80000000E+001   0.14955362E+002   0.00000000E+000
   0.90000000E+001   0.22251046E+002   0.00000000E+000
   0.10000000E+002   0.33513272E+002   0.00000000E+000
bash:/path/of/your/choice/SsTC/doc/examples/example1$
```

As an example, the true result for $R^{11}(7)$ is 9.45292988. By modifying `samples = (/33, 1, 1/)` to `samples = (/100, 1, 1/)`, compiling and running again gives

```
bash:/path/of/your/choice/SsTC/doc/examples/example1$ cat dummy-example01-test_11.dat
   0.10000000E+001   0.20861075E+001   0.00000000E+000
   0.20000000E+001   0.23578391E+001   0.00000000E+000
   0.30000000E+001   0.28579122E+001   0.00000000E+000
   0.40000000E+001   0.36663242E+001   0.00000000E+000
   0.50000000E+001   0.49154263E+001   0.00000000E+000
   0.60000000E+001   0.68151744E+001   0.00000000E+000
   0.70000000E+001   0.96937557E+001   0.00000000E+000
   0.80000000E+001   0.14062024E+002   0.00000000E+000
   0.90000000E+001   0.20715243E+002   0.00000000E+000
   0.10000000E+002   0.30893650E+002   0.00000000E+000
bash:/path/of/your/choice/SsTC/doc/examples/example1$
```

By now choosing `method = "extrapolation"` and returning to `samples = (/33, 1, 1/)`, gives

```
bash:/path/of/your/choice/SsTC/doc/examples/example1$ cat dummy-example01-test_11.dat
   0.10000000E+001   0.20843812E+001   0.00000000E+000
   0.20000000E+001   0.23504024E+001   0.00000000E+000
   0.30000000E+001   0.28390393E+001   0.00000000E+000
   0.40000000E+001   0.36268604E+001   0.00000000E+000
   0.50000000E+001   0.48401636E+001   0.00000000E+000
   0.60000000E+001   0.66785833E+001   0.00000000E+000
   0.70000000E+001   0.94529300E+001   0.00000000E+000
   0.80000000E+001   0.13644959E+002   0.00000000E+000
   0.90000000E+001   0.20001339E+002   0.00000000E+000
   0.10000000E+002   0.29681288E+002   0.00000000E+000
bash:/path/of/your/choice/SsTC/doc/examples/example1$
```

which shows faster convergence than the rectangle method. The user is now encouraged to modify the parameters given to the task constructor and also modify the calculator accordingly. We recommend considering more external variables or other functions to be integrated. The user can also check the output and error logs `SsTC_exec.out/err`. More complex calculators can also be implemented with the help of the routines in the "local k quantities" module.

11.2 Example 2: Modularity and jerk current

In this example we illustrate how the user can create a "mod" for SsTC, intended to calculate the jerk current of the system GaAs. The jerk current is a fourth order conductivity tensor [9, 10],

$$\iota^{abcd}(\omega) = \frac{2\pi e^4}{\hbar^3}\int_{\mathrm{BZ}}[d\boldsymbol{k}]\sum_{nm}[f_{nn}(\boldsymbol{k}) - f_{mm}(\boldsymbol{k})]\,\frac{\partial^2\left[\varepsilon_n(\boldsymbol{k}) - \varepsilon_m(\boldsymbol{k})\right]}{\partial k^a \partial k^d} r^b_{nm}(\boldsymbol{k}) r^c_{mn}(\boldsymbol{k})\delta(\varepsilon_n(\boldsymbol{k}) - \varepsilon_m(\boldsymbol{k}) - \hbar\omega), \quad (45)$$

where $r^a_{nm}(\boldsymbol{k}) = (1 - \delta_{nm})A^a_{nm}(\boldsymbol{k})$.

The code is contained in the directory

```
bash:/path/of/your/choice/SsTC/doc/examples/example2$
```

In the directory are 3 files and a folder,

```
bash:/path/of/your/choice/SsTC/doc/examples/example2$ ls
compile.sh  example2.F90  GaAs_tb.dat  mymod
```

`mymod` contains a module implementing the jerk conductivity tensor, `calculators_jerk.F90`. It provides the following

- A new type of integral task: the `optical_BZ_integral_task` an augmented version of the integrator task, with further capabilities, such as the possibility to determine a smearing in the calculation of delta functions.
- The task constructor `jerk_current_constructor`, which employs the regular integral task constructor and sets the value of the optical smearing as well as the values of integer and continuous indices for the case of the jerk current tensor .
- The jerk current calculator `jerk_current`, implementing the integrand of Eq. (45).

Aside from that, the folder also contains the necessary Makefiles and header files to be included by the SsTC "mod" loader. `GaAs_tb.dat` is an "exact" tight binding model for GaAs, constructed by post-processing a wannierization procedure of an *ab-initio* calculation for GaAs by Wannier90. `compile.sh` contains a compilation script for the source code `example1.F90`, which computes the jerk current tensor and prints the results.

The first step is to copy the `mymod` folder to the "mod" directory of SsTC and compile the library again,

```
bash:/path/of/your/choice/SsTC/doc/examples/example2$ cp -r mymod
../../../src/calculators/
bash:/path/of/your/choice/SsTC/doc/examples/example2$ cd ../../..
bash:/path/of/your/choice/SsTC$ make
```

Notice that the log has shown the messages

```
Detected mod makefiles are [./src/calculators/mymod/optical.mk ].
...
Including header file: src/calculators/mymod/headers.inc
Including procedure list file: src/calculators/mymod/procedures.inc
...
```

which states that the routines and declarations in the module `calculators_jerk.F90` have been added correctly to those provided by the unmodified SsTC. Now, any application using SsTC will also have in its scope the routines and declarations provided by the custom module.

Then, we return to the example directory and check the source code `example2.F90`. It declares the custom derived type `optical_BZ_integral_task`, which is constructed with the aid of the custom `jerk_current_constructor`. Notice that the routine already assumes that the user wants to calculate the jerk current, so there is no need to specify the calculator or the number of integer and continuous indices, which is already done by the programmer in the custom constructor routine. The only quantities to specify are the integration method, samples, and starting point, ending point and steps for ω in Eq. (45). Notice that no optical smearing is specified, so the adaptive smearing scheme [5] is employed. Then, the sampling, integration and printing is done by means of unmodified SsTC routines. In the calculation, a regular mesh of $100 \times 100 \times 100$ is employed, which requires, approximately 5000 seconds to finish in a personal computer with an Intel(R) Core(TM) i7-10700 CPU at 2.90GHz and 16 threads. Compiling & running gives

```
bash:/path/of/your/choice/SsTC/doc/examples/example2$ ./compile.sh
bash:/path/of/your/choice/SsTC/doc/examples/example2$ mpirun -np 1 ./test.x
bash:/path/of/your/choice/SsTC/doc/examples/example2$ ls
compile.sh  example2.F90  GaAs-jc_*.dat  GaAs_tb.dat  mymod  SsTC_exec.err
SsTC_exec.out  test.x
```

examining the output with and visualizing some response functions with a visualization software like `gnuplot` should reproduce Fig. 1 for the file `GaAs-jc_1111.dat`

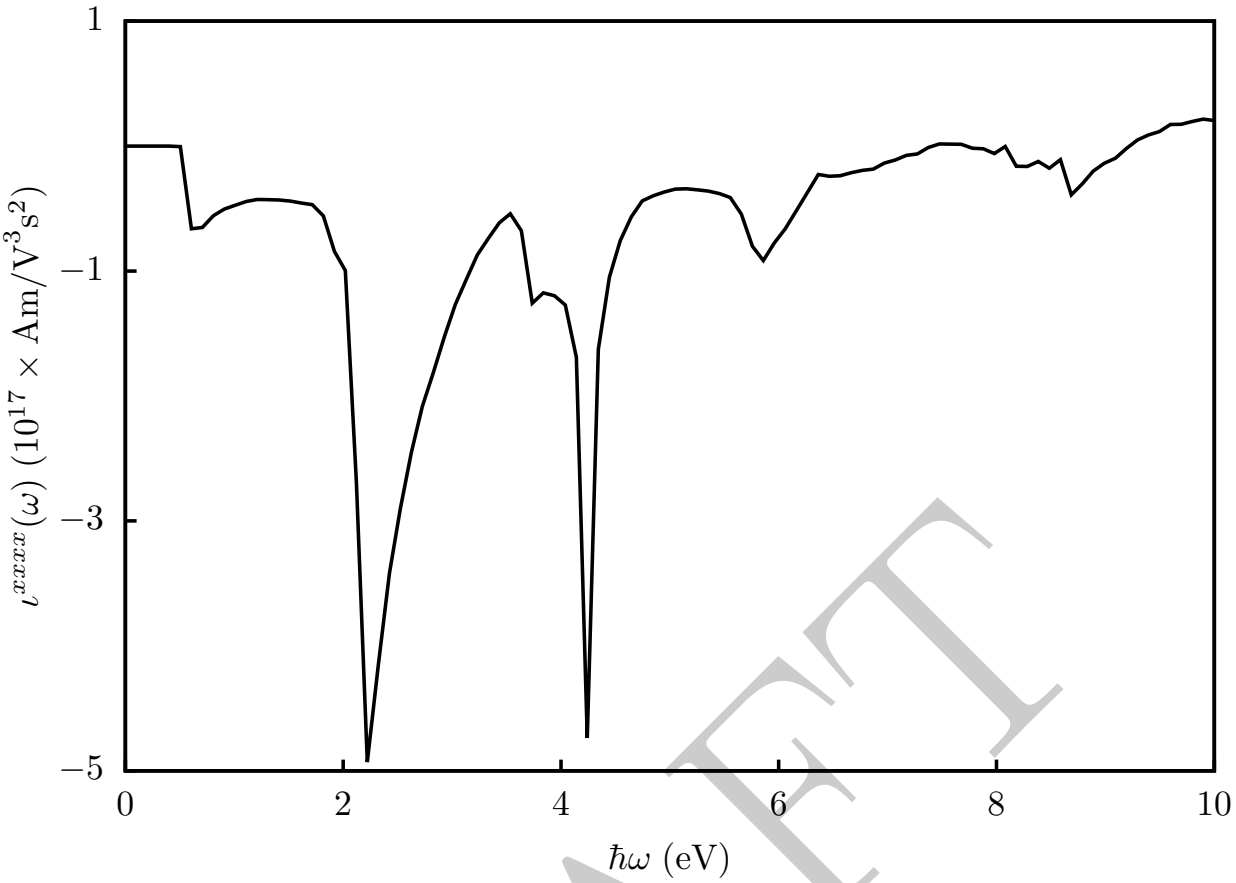

Figure 1: Jerk current tensor component ι^{xxxx} for GaAs in the $\hbar\omega \in [0, 10]$ eV range.

```
bash:/path/of/your/choice/SsTC/doc/examples/example2$ gnuplot
gnuplot> plot 'GaAs-jc_1111.dat' u ($1):(($2)/1E17) w l
```

Other components can also be visualized by the same means.

The user is encouraged to follow the lines of this example and implement his/her own calculators by using the SsTC routines as a basis.

12 Suggested practices

It is a good idea to make use of the BLOCK scoping unit in the main application when defining tasks,

```
...
block

  type(optical_BZ_integral_task) :: jerk

  call jerk_current_constructor(optical_task = jerk, method = "rectangle", &
                                samples = (/100, 100, 100/), &
                                omegastart = 0.0_dp, omegaend = 10.0_dp, &
                                omegasteps = 100)

  call SsTC_sample_and_integrate_BZ_integral_task(task = jerk, &
                                                  system = GaAs)

  call SsTC_print_BZ_integral_task(task = jerk, &
```

```
                                    system = GaAs)

end block
...
```

Listing 60: Definition of SsTC tasks in BLOCK scoping units.

these allow for efficient memory usage, especially if multiple tasks are going to be run. It also helps with derived type finalization. If any component of `task` is going to be used further in the application, creating a copy of the component to a global variable within the application scoping unit is suggested.

References

[1] Nicola Marzari, Arash A. Mostofi, Jonathan R. Yates, Ivo Souza, and David Vanderbilt. Maximally localized Wannier functions: Theory and applications. *Reviews of Modern Physics*, 84(4):1419–1475, October 2012.

[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 4.0, June 2021.

[3] Giovanni Pizzi, Valerio Vitale, Ryotaro Arita, Stefan Blügel, Frank Freimuth, Guillaume Géranton, Marco Gibertini, Dominik Gresch, Charles Johnson, Takashi Koretsune, Julen Ibañez-Azpiroz, Hyungjun Lee, Jae-Mo Lihm, Daniel Marchand, Antimo Marrazzo, Yuriy Mokrousov, Jamal I Mustafa, Yoshiro Nohara, Yusuke Nomura, Lorenzo Paulatto, Samuel Poncé, Thomas Ponweiser, Junfeng Qiao, Florian Thöle, Stepan S Tsirkin, Małgorzata Wierzbowska, Nicola Marzari, David Vanderbilt, Ivo Souza, Arash A Mostofi, and Jonathan R Yates. Wannier90 as a community code: New features and applications. *Journal of Physics: Condensed Matter*, 32(16):165902, April 2020.

[4] Xinjie Wang, Jonathan R. Yates, Ivo Souza, and David Vanderbilt. *Ab Initio* calculation of the anomalous Hall conductivity by Wannier interpolation. *Physical Review B*, 74(19):195118, November 2006.

[5] Jonathan R. Yates, Xinjie Wang, David Vanderbilt, and Ivo Souza. Spectral and Fermi surface properties from Wannier interpolation. *Physical Review B*, 75(19):195121, May 2007.

[6] J. J. Sakurai and Jim Napolitano. *Modern Quantum Mechanics*. Cambridge University Press, 2 edition, September 2017.

[7] Daniel E. Parker, Takahiro Morimoto, Joseph Orenstein, and Joel E. Moore. Diagrammatic approach to nonlinear optical response with application to Weyl semimetals. *Physical Review B*, 99(4):045121, January 2019.

[8] OpenMP Architecture Review Board, editor. *OpenMP Application Programming Interface Specification Version 5.2*. Independently Publishing, 2021.

[9] Álvaro R. Puente-Uriona, Stepan S. Tsirkin, Ivo Souza, and Julen Ibañez-Azpiroz. *Ab Initio* study of the nonlinear optical properties and dc photocurrent of the Weyl semimetal TaIrTe 4. *Physical Review B*, 107(20):205204, May 2023.

[10] Benjamin M. Fregoso, Rodrigo A. Muniz, and J. E. Sipe. Jerk Current: A Novel Bulk Photovoltaic Effect. *Physical Review Letters*, 121(17):176604, October 2018.