PSCF+: Polymer Self-Consistent Field Theory (C++/CUDA) with Improved and Extended Capabilities

Contents:

Generated on Tue Mar 5 2024 11:14:01 for PSCF+ by doxygen 1.8.17

Overview

Main Page (Up) **Installation** (Next)

PSCF+ is a software package for solving the polymer self-consistent field (SCF) theory in continuum. It is based on the nice GPU framework of PSCF (which is only for the "standard" or known as the Edwards-Helfand model, *i.e.*, incompressible melts of continuous Gaussian chains with the Dirac *δ*-function interactions, commonly used in polymer field theories) originally developed by Prof. David Morse and co-workers, but is improved with better numerical methods, less GPU memory usage and more flexible algorithms, and is extended to various discrete-chain models. Similar to the C++/CUDA version of PSCF, PSCF+ described here is written primarily in C++ with GPU accelerated code in CUDA.

Same as the C++/CUDA version of PSCF, PSCF+ is applicable to mixtures containing arbitrary acyclic copolymers, and preserves all of the nice features already implemented in the former, including the use of cuFFT on GPU, the use of Anderson mixing (which is performed on GPU) combined with a variable-cell scheme to simultaneously solve the nonlinear SCF equations and find the bulk periodicity for the ordered phases formed by block copolymer self-assembly (which speeds-up the calculation by about one order of magnitude), and the documentation produced by Doxygen. Their differences and expected advantages of the latter include:

- PSCF is only applicable to the "standard" model, while PSCF+ can also be applied to various discrete-chain models with finite-range non-bonded interactions commonly used in molecular simulations, thus providing the mean-field reference results for such simulations; see Models.pdf for more details.
- For the continuous-Gaussian-chain models, PSCF+ uses the Richardson-extrapolated pseudo-spectral methods (denoted by REPS-*K* with *K*=0,1,2,3,4) to solve the modified diffusion equations (which is the crux of SCF calculations of such models), while PSCF uses only REPS-1. A larger *K*-value gives more accurate result at larger computational cost; see REPS.pdf for more details.
- For 3D spatially periodic ordered phases such as those formed by block copolymer self-assembly, while PSCF uses fast Fourier transforms (FFTs) between a uniform grid in the real space and that in the reciprocal space, for the Pmmm supergroup PSCF+ uses discrete cosine transforms instead of FFTs to take advantage of the (partial) symmetry of an ordered phase to reduce the number of grid points, thus both speeding up the calculation and reducing the memory usage; see *Qiang and Li*, **Macromolecules 53**, 9943 (2020) for more details.
- In SCF calculations the (one-end-integrated) forward and backward propagators q and $q^\dagger$ of each block usually take the largest memory usage, but the GPU memory is rather limited. While in PSCF the size of these propagators is MN_s, where M denotes the number of grid points in real space and N_s the number of contour discretization points on a continuous Gaussian chain (or the number of segments on a discrete chain), in PSCF+ the "slice" algorithm proposed by Li and Qiang can be used to reduce the size of q to $M\sqrt{N_s}$ and that of $q^\dagger$ to just M, thus greatly reducing the GPU memory usage at the cost of computing q twice; see SavMem.pdf for more details.
- Since SCF equations are highly nonlinear, having a good initial guess is very important in practice as it determines not only which final solution (corresponding to a phase in block copolymer self-assembly) can be obtained but also how many iteration steps the solver (*e.g.*, the Anderson mixing) takes to converge these equations. PSCF+ uses automated calculation along a path (ACAP), where the converged solution at a

neighboring point is taken as the initial guess at the current point in the parameter space. While this is similar to the "SWEEP" command in PSCF, the key for ACAP to be successful and efficient is that it automatically adjusts the step size along the path connecting the two points. In PSCF+, ACAP is further combined with the phase-boundary calculation between two specified phases, making the construction of phase diagrams very efficient. See ACAP.pdf for more details.

- The approach used by PSCF to solve the SCF equations (for an incompressible system) does not allow any athermal species in the system, which has no Flory-Huggins-type interactions with all other species. This problem is solved in PSCF+; see SlvSCF.pdf for more details.

Main Page (Up) **Installation** (Next)

Installation

The following pages give instructions for obtaining, configuring and compiling PSCF+.

Contents:

System Requirements

The PSCF+ package provides programs that are designed to run on a desktop, laptop or cluster with an NVIDIA GPU. PSCF+ is distributed only as source code, and must be compiled by the user. All source code is written in ANSI 2011 C++ language standard with CUDA. Compilation of PSCF+ is controlled by a system of Unix makefiles and a series of shell scripts. In order to compile all of the programs in the PSCF+ package, the system on which the code is compiled must have:

- a "git" version control client
- a C++ compiler
- a python interpreter
- the GNU Scientific Library (GSL)
- a NVIDIA graphics card
- a CUDA compiler (nvcc)
- the cuFFT GPU-accelerated fast Fourier transform library
- JsonCpp, the C++ library that allows manipulating JSON values.

A git client is needed to obtain (clone) the source code, which is maintained in a git repository on the github.com server. A python interpreter is needed during compilation (but not during execution) because the build system that compiles the PSCF+ source code uses a few python scripts that are provided with the package. The GNU scientific library is used by several programs within the package for linear algebra operations. The cuFFT library, which is used extensively in SCF calculations here, is provided with recent versions of the CUDA development environment.

Source Code

The source code for PSCF+ is hosted on the github server project qwcsu/pscfplus, and can be obtained by using a git version-control manager to clone the public git repository. Instructions below assume that a "git" client has been installed on your computer.

To obtain a working copy of the PSCF+ git repository, you should first change directory (`cd`) to the directory you want to contain the `pscfplus/` directory. From there, then enter the command

```
> git clone --recursive https://github.com/qwcsu/pscfplus.git
```

This should create a complete working copy of the PSCF+ source code in a new subdirectory named `pscfplus/` of the directory from which you invoked the above command.

Hereafter, we assume that the root directory of the PSCF+ working copy is named `pscfplus/`. References to paths that do not start explicitly with a prefix `pscfplus/` should be understood as relative paths, relative to this directory.

Environment Variables

To compile PSCF+ in a Unix environment, before compiling any code, the user should modify the following Unix environment variables:

- Add the `pscfplus/bin/` directory to the Unix `$PATH` shell environment variable (the shell command search path). By default, executable file created by the PSCF+ build system is installed in the `pscfplus/bin/` directory. The directory in which these files are located must be added to the user's `$PATH` variable in order to allow the Unix shell to find the executable file when it is invoked by name in a command executed from any other directory.
- Add the `pscfplus/lib/python` directory to the `$PYTHONPATH` environment variable (the python module search path). The `pscfplus/scripts/python` directory contains a python script that is used by the build system during compilation to generate information about dependencies among C++ files. This directory must be added to the `$PYTHONPATH` variable in order to allow the python interpreter to find this file.

To make these changes using a bash shell, add some variant of the following lines to the `.profile` or `.bash_profile` file in your user home directory:

```
PSCFPLUS_DIR=${HOME}/pscfplus
export PATH=${PATH}:/${PSCFPLUS_DIR}/bin
export PYTHONPATH=${PYTHONPATH}:/${PSCFPLUS_DIR}/scripts/python
```

The value of PSCFPLUS_DIR should be set to the path to the PSCF+ root directory (*i.e.*, the root of the directory tree created by cloning the PSCF+ git repository). In the above fragment, as an example, it is assumed that this is a subdirectory named `pscfplus/` within the user's home directory.

After adding an appropriate variant of these lines to `.profile` or `.bash_profile`, re-login, and then enter `echo $PATH` and `echo $PYTHONPATH` to make sure that these variables have been set correctly.

Compilation

Environment Variables (Next) **Installation** (Up) **User Guide** (Next)

Below are the instructions for compiling the PSCF+ program with examples. It is assumed that you have cloned the PSCF+ repository and installed all required dependencies, and that the root directory of the repository is named `pscfplus/`.

Instructions:

- **Set environment variables:** Modify the user's $PATH and $PYTHONPATH Unix environment variables, as discussed **here**.
- **Navigate to root directory:** Change directory (`cd`) to the `pscfplus/` root directory.
- **Setup:** Invoke the "setup" script from the `pscfplus/` root directory. Enter the command

  ```
  > ./setup
  ```

  to setup the build system with default compiler options. Alternatively, invoke the setup with a filename argument for non-default compiler options.
- **Change directory to the build directory:** Change directory (`cd`) to the `pscfplus/bld` subdirectory, by entering `cd bld` from the root directory.
- **Compile the PSCF+ program for a given model system:** From `pscflus/bld`, enter

  ```
  > bash compile.sh [-B CHN] [-N NBP] [-C] [-D] [-K K]
  ```

  This will generate a large number of intermediate object (`*.o`), dependency (`*.d`) and library (`*.a`) files in subdirectories of the `pscfplus/bld` directory, and install the executables in the `pscfplus/bin` directory. The options in the above command are as follows:
  - CHN: Specifying the model of chain connectivity (by default it is the continuous Gaussian chain); see Models.pdf for details.

    ```
    DGC: discrete Gaussian chain
    FJC: freely jointed chain
    ```
  - NBP: Specifying the form of non-bonded pair potential (by default it is the Dirac δ-function potential); see Models.pdf for details.

    ```
    G: Gaussian potential
    DPD: dissipative particle dynamics potential
    SS: soft-sphere potential
    ```
  - `-C`: Specifying a compressible system (by default the system is incompressible); see Models.pdf for details.
  - `-D`: Specifying the use of discrete cosine transforms between the real and reciprocal space (by default the fast Fourier transforms are used.)
  - `-K`: Specifying the *K*-value of the REPS-*K* method (by default the REPS-1 method is used); this is used only for the continuous-Gaussian-chain models (see REPS.pdf for details.)

Examples:

- **Compilation for the "standard" model:** To compile PSCF+ for calculations of the “standard” model (*i.e.*, incompressible melts of continuous Gaussian chains with the Dirac δ-function repulsion) using the REPS-1

method and fast Fourier transforms (same as used in PSCF), simply use the following command:

```
bash compile.sh
```

- **Compilation for the DPDC model:** To compile PSCF+ for calculations of the DPDC model (*i.e.*, compressible melts of discrete Gaussian chains with the dissipative particle dynamics potential) using fast Fourier transforms, users can use the following command:

```
bash compile.sh -B DGC -C -N DPD
```

- To get a list of the aboved options, use the following command:

```
bash compile.sh -h
```

User Guide

Installation (Prev)

Contents:

Installation (Prev) **Main Page** (Up)

Generated on Tue Mar 5 2024 11:13:58 for PSCF+ by doxygen 1.8.17

Invoking an Executable

Calculation of a single (ordered) phase

Here is an example of command-line usage of PSCF+ program for calculation of a single ordered phase:

```
pg [-e] -d D
```

In the above, `pg` is the name of executable, `-e` activates echoing of the parameter file to standard output (which is optional), dimensionality `D` of the system is passed to the program as argument of the `-d` command-line option (such implementation is due to Prof. David Morse).

Single-phase SCF calculation requires two input files:

- a parameter file: `param`
- a command file: `command`

under the working directory, and their names have to be `param` and `command`, respectively.

When the program is executed, the parameter file is read first, which is used to initialize the state of the program and allocate memory. The command file is read and interpreted after the parameter file. The command file is in JSON format and contains a list of commands that are interpreted and executed in sequence, which controls the program flow after initialization. The contents and formats of these two types of file are explained in detail elsewhere (see **Parameter Files**, **Command Files**).

Calculation of the boundary between two phases

Here is an example of command-line usage of PSCF+ to calculate the boundry between two phases (where they have the same Helmholtz free-energy density) using the Ridders' method.

```
pg [-e] -d D1,D2
```

In the above, dimensionalities of the two phases, `D1` and `D2`, are passed to the program as arguments of the `-d` command-line option; use 0 for dimensionality of the disordered phase.

Two-phase SCF calculation requires three input files:

- two parameter files: `param1` and `param2`
- a command file: `command`

under the working directory, and their names have to be `param1`, `param2` (for the two phases having dimensionalities `D1` and `D2`, respectively) and `command`.

Parameter Files

Invoking an Executable (Prev) **Command Files** (Next)

The structure of parameter file is adapted from the C++/CUDA version of PSCF, and contain one `System` block as shown below.

```
System{
  Mixture{
    nMonomer   ...
    monomers   ...  ...
               ...  ...
    nPolymer   ...
    Polymer{
      nBlock  ...
      nVertex ...
      blocks  ...  ... ... ...  ...
              ...  ... ... ...  ...
      phi     ...
    }
    DPolymer{
      nBlock  ...
      nVertex ...
      bonds   ...  ... ... ...  ...  ...
              ...  ... ... ...  ...  ...
      phi     ...
    }
    [ns        ...]
  }
  Interaction{
     chi ... ... ...
         ... ... ...
     [kappa    ...]
     sigma     ...
  }
  unitCell ... ... ...
  mesh ....
  groupName ...
  AmIterator{
     maxItr  ...
     epsilon ...
     maxHist ...
     isMinimized ...
  }
}
```

Each sub-block and required parameter (represented by ...) is explained as follows:

- **Mixture**: Description of molecular components (each is considered as a block copolymer in general with each block being a linear homopolymer) and composition in the system (which is considered as a mixture in general).
    - **nMonomer**: Number of monomer (segment) types in the system; this includes solvent molecules.
    - **monomers**: Description of each segment type in a seperate line (thus a total of `nMonomer` lines). The first parameter in each line is a unique integer index starting from 0 for the segment type, and the second parameter specifies its statistical segment length.
    - **nPolymer**: Number of molecular components in the system.
    - **Polymer** (only used for continuous-Gaussian-chain models): Description of each molecular component in a seperate sub-block (thus a total of `nPolymer` sub-blocks), which includes its chain architecture (specified by `nBlock`, `nVertex`, and `blocks` as explained below) and its overall volume fraction `phi` in the system.

        - **nBlock**: Number of blocks of this molecular component.
        - **nVertex**: Number of vertices of this molecular component. A vertex is either a joint (where at least two blocks meet) or a free end.
        - **blocks**: Description of each block in a seperate line (thus a total of `nBlock` lines). The first parameter in each line is a unique integer index starting from 0 for the block, the second parameter specifies its segment type, the next two parameters are the indices of the two vertices it connects, and the last parameter specifies its length.
    - **DPolymer** (only used for discrete-chain models): Description of each molecular component in a seperate sub-block (thus a total of `nPolymer` sub-blocks), which includes its chain architecture (specified by `nBond`, `nVertex`, and `bonds` as explained below; see Models.pdf for details.) and its overall volume fraction `phi` in the system.
        - **nBond**: Number of v-bonds (including both block bonds and joint bonds) of this molecular component.
        - **nVertex**: Number of vertices of this molecular component. A vertex here is either a joint (which is connected by at least two v-bonds) or a free end (which is connected by one v-bond).
        - **bonds**: Description of each v-bond in a seperate line (thus a total of `nBond` lines). The first parameter in each line is a unique integer index starting from 0 for the bond, the second and the third parameters are the indices of the two vertices it connects, the next two parameters specify the types of these vertices (segments), and the last parameter is its number of segments (0 for a joint bond).
    - **ns**: Total number of discretization steps along the chain contour of length 1. This line is used only for continuous-Gaussian-chain models, and is omitted for discrete-chain models.
- **Interaction**: Description of non-bonded interactions in the system.
    - **chi**: Value of the (generalized) Flory-Huggins χ parameter for each pair of different segment types in a seperate line. The first two parameters in each line are the segment-type indices, and the third one is the corresponding value of χ. By default, the value between segments of the same type is 0, and thus not needed.
    - **kappa**: Compressibility parameter κ, used only for compressible systems and omitted for incompressible systems.
    - **sigma**: Interaction range of the non-bounded potential, which is 0 for Dirac δ-function interaction.
- **unitCell**: The first parameter in this line is the lattice system of the unit cell and the following is a list of real numbers needed to describe the unit cell.
- **mesh**: Description of the mesh size used for spatial discretization, given by D integer numbers with D being the dimensionality of the system.
- **groupName**: Name of the crystallographic space group.
- **AmIterator**: Parameters required by Anderson mixing for iteratively solving the SCF equations; see *Matsen*, **Eur. Phys. J. E 53**, 361 (2009) for details.
    - **maxItr**: Maximum number of iterations.
    - **epsilon**: Criterion of convergence for SCF equations.
    - **maxHist**: A positive integer for the maximum size of the history matrix used in Anderson mixing.
    - **isMinimized**: 1 for finding the bulk period of the ordered phase, and 0 otherwise.

Below are two examples of the parameter file:

- ***Example for SCF calculations of the BCC phase formed by the "Standard" model of compositionally asymmetric A-B diblock copolymer***

```
System{
  Mixture{
    nMonomer  2
    monomers  0   A   1.0
              1   B   1.0
    nPolymer  1
    Polymer{
      nBlock  2
      nVertex 3
      blocks  0  0  0  1  2.50000000E-01
              1  1  1  2  7.50000000E-01
      phi     1.0
    }
    ns        128
  }
  Interaction{
    chi   1   0   20.0
    sigma 0.0
  }
  unitCell cubic      4.6662857614e+00
  mesh      64   64  64
  groupName         I_m_-3_m
  AmIterator{
    maxItr 5000
    epsilon 1e-9
    maxHist 20
    isMinimized 1
  }
}
```

- ***Example for SCF calculations of the σ phase formed by the DPDC model of conformationally asymmetric A-B diblock copolymer***

```
System{
  Mixture{
    nMonomer  2
    monomers  0   A   3.0
              1   B   1.0
    nPolymer  1
    DPolymer{
      nBond   3
      nVertex 4
      bonds   0  0  1  0  0   3
              1  2  3  1  1   7
              2  1  2  0  1   0
      phi     1.0
    }
  }
  Interaction{
    chi   1   0   2.0
    kappa 0.06366197723676
    sigma 0.89442719099992
  }
  unitCell tetragonal      2.8767371691e+01      1.5168759856e+01
  mesh    128  128  64
  groupName         P_42%m_n_m
  AmIterator{
    maxItr 5000
    epsilon 1e-8
    maxHist 20
    isMinimized 1
  }
}
```

Command Files

Parameter Files (Prev)

The command file contains a sequence of commands that are read and executed in serial. The commands are organized into a JSON file. Below is an example of a command file for single-phase calculation:

```
[
    { "CaseId": "1" },
    {
        "FieldIO": {
            "IO" : "read",
            "Type": "omega",
            "Format": "basis",
            "Directory": "in/"
        }
    },
    {
        "SinglePhaseSCF": {
            "OutputDirectory": "out/"
        }
    },
    {
        "FieldIO": {
            "IO" : "write",
            "Type": "omega",
            "Format": "basis",
            "Directory": "out/omega/"
        }
    },
    {
        "FieldIO": {
            "IO" : "write",
            "Type": "phi",
            "Format": "real",
            "Directory": "out/phi/"
        }
    }
]
```

All commands are put in a pair of square brackets, and they are divided into different blocks. (Here gives an introduction to JSON)

The following explain the usage of each command block.

- The first block must be the `"CaseId"` block.

  ```
  { "CaseId": "your_case_id" }
  ```

  This command specifies the case ID of the calculation, `your_case_id`, which is part of the names of output files. The case ID can be anything, even an empty string.
- To read or write a field (*e.g.*, volume-fraction or conjugate field) file in a specified format, use `"FieldIO"` block.

  ```
  {
      "FieldIO": {
          "IO" : "read",
          "Type": "omega",
          "Format": "basis",
          "Directory": "in/"
      }
  }
  ```

`"IO"` is either `"read"` or `"write"` for reading from or writing to a file, respectively. `"Type"` specifies the field, which can be either `"omega"` for conjugate field or `"phi"` for volume-fraction field. `"Format"` specifies the format of the field, which can be either `"basis"` for the basis format, `"real"` for the real-space-grid format, or `"reciprocal"` for the reciprocal-space-grid format; see PSCF documentation for the explaination of these formats. `"Directory"` specifies the directory of the file. Finally, the file name is specified by the case ID, field type and format as `your_case_id_type.format`; for example, to read a conjugate field in the basis format as input of your SCF calculation with the case ID 1234, the file name must be `1234_omega.basis`.

- To perform SCF calculation of a single phase with given initial guess (the conjugate field of which should be read before), use the `"SinglePhaseSCF"` block.

```
{
    "SinglePhaseSCF": {
        "OutputDirectory": "out/"
    }
}
```

`"OutputDirectory"` specifies the directory of the output file for the system free energy and its components. This output file name is `your_case_id_out.json`. For example, with the case ID 1234, the name of the output file is `1234_out.json`.

To perform automated calculation along a path (ACAP; see ACAP.pdf for details.), use the block `"ACAP"`.

```
[
    { "CaseId": "1" },
    {
        "FieldIO": {
            "IO" : "read",
            "Type": "omega",
            "Format": "basis",
            "Directory": "in/"
        }
    },
    {
        "ACAP":{
            "Variable": ["chi", 0, 1],
            "InitialValue": 16,
            "FinalValue": 15.5,
            "InitialStep": 0.1,
            "SmallestStep": 0.001,
            "LargestStep": 0.5,
            "StepScale": 1.1,
            "OutputDirectory": "out/",
            "IntermediateOuput":
            [
                {
                    "OutputPoints": [15.4, 15.6, 15.8]
                },
                {
                    "Field" : "omega",
                    "Format": "basis",
                    "OutputDirectory": "out/omega/"
                },
                {
                    "Field" : "phi",
                    "Format": "real",
                    "OutputDirectory": "out/phi/"
                }
            ]
        }
    },
    {
        "FieldIO": {
            "IO" : "write",
            "Type": "omega",
            "Format": "basis",
```

```
            "Directory": "out/omega/"
        }
    },
    {
        "FieldIO": {
            "IO" : "write",
            "Type": "phi",
            "Format": "real",
            "Directory": "out/phi/"
        }
    }
]
```

`"Variable"` specifies the paramter whose value is varied along the path; this is so far either `"chi"`, the Flory-Huggins parameter bewteen two segments of different types, or `"b"`, the statistical segment length of a segment type. If the varing parameter is `"chi"`, user needs to specify the two segment types as shown in the above example. If the varing parameter is `"b"`, user needs to specify the corresponding segment type (*e.g.*, `["b", 0]`). `"InitialValue"` and `"FinalValue"` give the starting and ending parameter values of the path, respectively. `"InitialStep"`, `"SmallestStep"`, and `"LargestStep"` specifies the initial, smallest and largest absolute values of the stepsize, respectively, used for varing the parameter along the path. `"StepScale"` specifies the scaling factor used to vary the stepsize. `"OutputDirectory"` specifies the directory of the output file for the system free energy and its components along the path. `"IntermediateOuput"` is needed when user wants to output field files during ACAP. The first block in `"IntermediateOuput"` specifies the parameter values at which the fields are output along the path (the order of these values does not matter, which means `[1.1, 1.2, 1.3]` and `[1.2, 1.3, 1.1]` result in the same intermediate output files). Each of the following blocks specifies the type of the field, its format, and the directory of the output files via `"Field"`, `"Format"`, and `"IntermediateDirectory"`, respectively.

To find a boundary point between two specified phases, where they have the same Helmholtz free-energy density, use the `"PhaseBoundaryPoints"` block as in the following example:

```
[
    {   "CaseId": "1"   },
    {
        "FieldIO": {
            "PhaseId": 1,
            "IO" : "read",
            "Type": "omega",
            "Format": "basis",
            "Directory": "in/1/"
        }
    },
    {
        "FieldIO": {
            "PhaseId": 2,
            "IO" : "read",
            "Type": "omega",
            "Format": "basis",
            "Directory": "in/2/"
        }
    },
    {
        "PhaseBoundaryPoints": {
            "epsilon": 1e-5,
            "b": [1, 1.0],
            "InitialGuess(chi)": [0, 1, 19.1, 19.3]
        }
    }
    {
        "FieldIO": {
            "PhaseId": 1,
            "IO" : "write",
            "Type": "omega",
```

```
            "Format": "basis",
            "Directory": "out/1/omega/"
        }
    },
    {
        "FieldIO": {
            "PhaseId": 2,
            "IO" : "write",
            "Type": "phi",
            "Format": "real",
            "Directory": "out/2/phi/"
        }
    }
]
```

Here, the initial guess of each phase is read first by the two `"FieldIO"` blocks; different from the above single-phase calculation, `"PhaseId"` is needed in each `"FieldIO"` block, which takes the value of `1` or `2` in accordance to the command-line arguments of `-d`, `D1` and `D2`, respectively (see **Invoking an Executable**). In the `"PhaseBoundaryPoints"` block, `"epsilon"` specifies the criterion of convergence, which is the absolute difference in the Helmholtz free-energy density between the two phases; the next line specifies that the calculation is performed at the constant value for the statistical segment length (*i.e.*, `"b"`) of segment type 1, which is 1.0; in this case, the calculation solves for the χ value between segment types `0` and `1`, which falls in the interval of [19.1, 19.3] as shown in third line. Note that this interval is required by the Ridders' method used for the phase-boundary calculation.