The Markuspline Fortran Module

Francesco Calcavecchia

January 30, 2015

The `markuspline` Fortran module follows the recipe of Ref. [HB05] for representing a spline.

The module `markuspline` depends on the LAPACK library.

The following is a short user's documentation.

1 Declaration of the module

First of all one has to declare the utilization of the `markuspline` module, using the instruction
`USE markuspline`
which has to be inserted just before the `IMPLICIT NONE` command in any `PROGRAM`, `SUBROUTINE`, `FUNCTION`, or `MODULE`.

2 Spline's declaration

In the following we will use the variable name `spl` in our examples. A spline `spl` must be declared as

```
TYPE(MSPLINE) :: spl
```

3 Spline's initialization

As first step, the spline should be initialized, by using the command

```
spl=new_MSPL(M= , NKNOTS= , LA= , LB= , CUTOFF= )
```

where `M` and `NKNOTS` are two `INTEGER`s which refer to m and N_{spline} in [HB05], while `LA` and `LB` are two `REAL(KIND=8)` which represent the range on which the spline is defined (typically `LA=0.d0`). Finally, `CUTOFF` is a `LOGICAL` which specifies whether the spline should smoothly goes to zero at `LB` or not. Alternatively, one can use the subroutine

```
CALL MSPL_new(M= , NKNOTS= , LA= , LB= , SPL=spl, CUTOFF= )
```

After the initialization, the spline can be used. One has access to the following internal variables:

- `LOGICAL :: spl%flag_init`, it indicates whether the `MSPLINE` has been initialized or not (by using `new_MSPLINE` or `MSPLINE_new`);
- `INTEGER :: spl%m`;
- `INTEGER :: spl%Nknots`;
- `REAL(KIND=8) :: spl%La`;
- `REAL(KIND=8) :: spl%Lb`;
- `REAL(KIND=8) :: spl%x(-1:spl%Nknots+1)`
  The grid of the spline. The index `0` corresponds to `spl%La` and the index `spl%Nknots` to `spl%Lb`;
- `REAL(KIND=8) :: spl%delta`
  The distance between two consecutive points in the grid;
- `REAL(KIND=8) :: spl%t(0:spl%m,0:spl%Nknots)`
  The t matrix in reference [HB05] but with exchanged indexes (or, in other words, is the transposed matrix), which by default is initialized to zero by `new_MSPLINE` and `MSPLINE_new`;
- `LOGICAL :: spl%cutoff`.

4 Store and Load a spline

It is possible to store in a file a `MSPLINE` variable by using the subroutine

```
CALL MSPL_store(SPL= , FILENAME= )
```

The file is organized as following:

```
spl%m,spl%Nknots,spl%La,spl%Lb
spl%t
```

A stored spline can be loaded by means of

```
CALL MSPL_load(SPL= , FILENAME= )
```

which will also allocate the `MSPLINE` spline (in other words, it can replace the `CALL MSPL_new()` command).

5 Fit a function

At this point one can set the `t` matrix. The `markuspline` module provide a tool to fit any given function of the form:

```
FUNCTION f(i,x)
   REAL(KIND=8) :: f
   INTEGER, INTENT(IN) :: i
   REAL(KIND=8), INTENT(IN) :: x
END FUNCTION f
```

To accomplish this, it is sufficient to invoke the following subroutine

```
CALL MSPL_fit_function(SPL=spl,F=f)
```

6 Compute the spline and its derivatives

After that the spline has been set, its value in any $\mathtt{r} \in [\mathtt{spl\%La}, \mathtt{spl\%Lb}]$ can be computed by invoking the function

```
compute_MSPL(SPL=spl, DERIV=0, R=r)
```

which returns a `REAL(KIND=8)`. The derivatives of the spline can be computed simply by setting the value of `DERIV` accordingly (`DERIV=1` for the first derivative, `DERIV=2` for the second derivative, etc.). Alternatively, the value of the spline function and of its derivatives can be computed by means of the subroutine

```
CALL MSPL_compute(SPL=spl, DERIV= , R=r, VAL=val)
```

where `val` is a `REAL(KIND=8)`. If one wants to accumulate the computed value in the variable `val`, one can add the optional argument `RESET=.FALSE.`, e.g.

```
CALL MSPL_compute(SPL=spl,DERIV= ,R=r,VAL=val,RESET=.FALSE.)
```

This is equivalent to the instruction

```
val=val+compute_MSPL(SPL=spl,DERIV=0,R=r)
```

7 Compute the gradient of the spline

Suppose that the spline is computed on the distance between two particles `i` and `j`, whose coordinates are represented in $\mathbb{R}^{\mathtt{d}}$:

$$\mathtt{r} = ||\mathbf{r}_j - \mathbf{r}_i||$$

It is possible to compute the gradient

$$\mathtt{grad(1:d)} = \nabla_{\mathtt{j}}\mathtt{spl}(\mathtt{r})$$

by invoking

```
CALL MSPL_grad(SPL=spl,R_VEC=dist,NDIM=d,GRAD=grad[,RESET= ])
```

where `dist(0:d)` is of type `REAL(KIND=8)` and `RESET` is the usual `LOGICAL` optional argument. It is important to know that `dist(1:d)` contains the vector $\mathbf{r}_j - \mathbf{r}_i$, whereas `dist(0)` contains the distance (i.e. `r`).

8 Compute the laplacian of the spline

Suppose that the spline is computed on the distance between two particles `i` and `j`, whose coordinates are represented in $\mathbb{R}^{\mathtt{d}}$:

$$\mathtt{r} = ||\mathbf{r}_j - \mathbf{r}_i||$$

It is possible to compute the laplacian

$$\mathtt{lapl} = \nabla^2_{\mathtt{j}}\mathtt{spl}(\mathtt{r})$$

by invoking

```
CALL MSPL_laplacian(SPL=spl,R=r,NDIM=d,LAPL=lapl[,RESET= ])
```

where `r` is of type `REAL(KIND=8)` and `RESET` is the usual `LOGICAL` optional argument. It is important to know that `r` contains the scalar $||\mathbf{r}_j - \mathbf{r}_i||$.

9 Print on file

It is possible to print on file the values of the spline or its derivatives. This can be useful in order to visualize it. To do so, use

```
CALL MSPL_print_on_file(SPL= , DERIV= , FILENAME= , NPOINTS= )
```

where `DERIV` and `NPOINTS` are `INTEGER`s, and `FILENAME` is a string containing the name of the file where the spline values should be stored. As a result, a file with `NPOINTS` rows containing the position $\mathtt{r} \in [\mathtt{SPL\%La}, \mathtt{SPL\%Lb}]$ and the value of the spline $\mathtt{SPL}(\mathtt{r})$.

10 Parameter derivative of the spline

It is also possible to compute the parameter (or variational) derivatives with respect to the t matrix. In particular, the subroutine

```
CALL MSPL_t_deriv(SPL=spl,R=r,T_DERIV=td)
```

assigns to the matrix `td`, defined as

```
REAL(KIND=8) :: td(0:spl%m,0:spl%Nknots)
```

the derivatives' matrix

$$\texttt{td(i,j)} = \frac{\partial}{\partial t_{ij}} \texttt{spl(r)}$$

If one wants to accumulate the derivatives on top of the pre-existing `td` matrix, it is sufficient to add the optional variable `RESET=.FALSE.` to the arguments of the subroutine, e.g

```
CALL MSPL_t_deriv(SPL=spl,R=r,T_DERIV=td,RESET=.FALSE.)
```

In this way

$$\texttt{td(i,j)} := \texttt{td(i,j)} + \frac{\partial}{\partial t_{ij}} \texttt{spl(r)}$$

10.1 Parameter derivative for particles in a simulation box

The `markuspline` module contains a subroutine which computes the variational derivatives for all the distances between `n` particles in a `d` dimensional space simulation box with sizes `L=(\Lx,Ly, ... \)`. If we label the particles coordinates as `X`, the invocation reads

```
CALL MSPL_boxNpart_t_deriv(SPL=spl,R=X,NDIM=d,NPART=n,LBOX=L,
                                                    T_DERIV=td)
```

where `n` and `d` are `INTEGER`s, whereas `X(1:d,1:n)` and `L(1:d)` are of type `REAL(KIND=8)`. Also in this case it is possible to accumulate the values of the variational derivatives by adding the optional argument `RESET=.FALSE.`

11 Gradient of the parameter derivative

Suppose that the spline is computed on the distance between two particles `i` and `j`, whose coordinates are represented in $\mathbb{R}^{\texttt{d}}$:

$$\texttt{r} = ||\mathbf{r}_j - \mathbf{r}_i||$$

It is possible to compute the quantities

$$\texttt{grad(m,n,1:d)} = \nabla_{\texttt{j}} \frac{\partial}{\partial t_{mn}} \texttt{spl}(r)$$

for all the `m` and `n` by invoking

```
CALL MSPL_grad_t_deriv(SPL=spl,R_VEC=dist,NDIM=d,
                                  GRAD_T_DERIV=grad[,RESET= ])
```

where `dist(0:d)` is of type `REAL(KIND=8)` and `RESET` is the usual `LOGICAL` optional argument. It is important to know that `dist(1:d)` contains the vector $\mathbf{r}_j - \mathbf{r}_i$, whereas `dist(0)` contains the distance (i.e. `r`).

11.1 Gradient of the parameter derivative for particles in a simulation box

As in Sec. 10.1, also in this case it is possible to compute the gradient for a simulation box. This is done with

```
CALL MSPL_boxNpart_grad_t_deriv(SPL=spl, R=X, NDIM=d, NPART=n,
                                LBOX=L, GRAD_T_DERIV=grad [, RESET= ])
```

However, in this case the argument has different dimension:

```
grad(0:spl%m,0:spl%Nknots,1:d,1:n)
```

12 Laplacian of the parameter derivative

Suppose that, similarly to Sec. 11, we are now interested in the laplacian

$$\texttt{lapl(m,n,1:d)} = \nabla^2_{\mathbf{j}} \frac{\partial}{\partial t_{mn}} \texttt{spl}(r)$$

This can be computed with

```
CALL MSPL_lapl_t_deriv(SPL= , R_VEC= , NDIM= ,
                                   LAPL_T_DERIV= [, RESET= ])
```

where `LAPL_T_DERIV(0:spl%m,0:spl%Nknots)` is of type `REAL(KIND=8)`.

13 Gradient and laplacian of the parameter derivative

One can simultaneously obtain the results from both Sec. 11 and 12 with (i.e. both the gradient and the laplacian) with the command

```
CALL MSPL_grad_and_lapl_t_deriv(SPL= , R_VEC= , NDIM= ,
                    GRAD_T_DERIV , LAPL_T_DERIV [, RESET= ])
```

13.1 Gradient and laplacian of the parameter derivative for particles in a simulation box

As in Sec. 10.1 and 11.1, we can compute all the gradients and laplacians in a simulation box with

```
CALL MSPL_boxNpart_grad_and_lapl_t_deriv(SPL= , R= , NDIM= ,
NPART= , LBOX= , GRAD_T_DERIV= , LAPL_T_DERIV= [, RESET= ])
```

where one has to be aware of the dimensionality of the two following arrays:

```
GRAD_T_DERIV(0:SPL%m,0:SPL%Nknots,1:NDIM,1:NPART)
LAPL_T_DERIV0:SPL%m,0:SPL%Nknots,1:NPART)
```

14 Carbon-copy

A spline can be transposed to another spline with different m and N_{spline}, by doing a "carbon-copy". This can be accomplished by invoking

```
CALL MSPL_carbon_copy(ORIGINAL_SPL=spl_in,CC_SPL=spl_out)
```

where `spl_in` and `spl_out` have type `TYPE(MSPL)`.

15 Deallocate the spline's allocated memory

When the spline is of no use any more, its allocated memory can be freed by invoking

```
CALL MSPL_deallocate(SPL=spl)
```

16 Notes on the efficiency

If an instruction is called a lot of times, using the subroutine instead of the function will drastically increase the performance.

17 Debugging mode

A final note about the routine checks that are performed internally. By default, the `markuspline` module checks the input values provided by the user, in order avoid possible mistake (for example, if the user provides a `LA` $\geq$ `LB`). However, these checks can be disabled with the command

```
CALL MSPL_change_debug_mode(DEBUG_MODE=.FALSE.)
```

which changes the internal variable `MSPL_DEBUG_MODE` accordingly. This might slightly enhance the performance. The current value of `MSPL_DEBUG_MODE` can be obtained with the function

```
get_debug_mode_MSPL()
```

References

[HB05] Markus Holzmann and Bernard Bernu. Optimized periodic 1/r coulomb potential in two dimensions. *Journal of Computational Physics*, 206(1):111 – 121, 2005.