Using BSPLINES in Particle Codes

Trach-Minh Tran v0.1, March 2012

These notes present some practical considerations on using BSPLINES in particle codes, in particular for the charge or current assignment as well as the field interpolation. Performance measurements are done on an Intel Xeon X5570 and the more recent Xeon E5-2680.

1 Introduction

For simplicity, we assume in these notes that we are dealing with a 2D electrostatic particle code and the 2D Poisson equation is to be solved using the Finite Element Method. Starting from the *weak form* and using the *splines* for both *basis* and *test* functions, the electrostatic field potential together with its gradient and the right hand side can be computed from

$$\begin{aligned} \phi(x,y) &= \sum_{ij} c_{ij}\, \Lambda_i(x)\Lambda_j(y) \\ \frac{\partial\phi}{\partial x} &= \sum_{ij} c_{ij}\, \Lambda_i'(x)\Lambda_j(y) \\ \frac{\partial\phi}{\partial y} &= \sum_{ij} c_{ij}\, \Lambda_i(x)\Lambda_j'(y) \\ S_{ij} &= \sum_{\mu=1}^{N_p} q_\mu \Lambda_i(x_\mu)\Lambda_j(y_\mu), \end{aligned} \tag{1}$$

where c_{ij} are the solutions of the discretized Poisson equation and $\{x_\mu, y_\mu\}$ are the coordinates of the N_p simulation particles. At each time step, the calculation of both the field ϕ and its gradient (*field interpolation*) for the particle pusher and the construction of the RHS S_i (*charge assignment*) involve thus the computation of a large number of splines Λ and its derivatives Λ'.

Notice that the construction of the solver matrix requires also the calculations of the splines. This operation is however performed only once at the initial timestep in the (most common) case where the matrix is time independent and thus will not be considered in further these notes.

2 Computation of splines

Let consider the grid defined by x_i, $i = 1, \ldots, N+1$. Inside the interval $[x_i, x_{i+1}]$, the $p+1$ non-zero splines of degree p can be computed efficiently using its polynomial representation given by

$$\begin{aligned} \Lambda_{i+\alpha}(x) &= \sum_{k=0}^{p} V_{k\alpha}^i (x - x_i)^k, \qquad \alpha = 1, \ldots, p+1, \\ V_{k\alpha}^i &= \frac{1}{k!}\frac{d^k}{dx^k}\Lambda_{i+\alpha}(x)\bigg|_{x=x_i}. \end{aligned} \tag{2}$$

The $(p+1)^2 N$ coefficients $V_{k\alpha}^i$ are precalculated and stored during the spline initialization (in routine `SET_SPLINE`) by using the *recurrence relation* [1] to compute the spline and all its p derivatives. Note that for periodic splines on an equidistant mesh, only $(p+1)^2$ coefficients $V_{k\alpha}$ are required since the splines have *translational invariance*.

For a polynomial $P(x) = a_0 + a_1 x + \ldots + a_p x^p$, its value can be calculated together with it first derivative, using Horner's rule as:

```
f = a(p)
fp = f
DO i=p-1,1,-1
 f = a(i) + x*f
 fp = f + x*fp
END DO
f = a(0) + x*f
```

showing that exactly $4p - 2$ floating operations (flops) per point are required. If only the value of the polynomial is needed, only $2p$ flops per point are required.

3 Field interpolation

3.1 1D case

Let considered first the 1D case. The spline expansion of ϕ for $x_i \leq x < x_{i+1}$ are expressed as

$$\phi(x) = \sum_{\alpha=0}^{p} c_{i+\alpha} \Lambda_{i+\alpha}(x). \tag{3}$$

To calculate the field using this spline expansion, $p+1$ splines have to be first calculated followed by the sum above, which yields a total cost of $2(p+1)^2 \sim 2p^2$ flops per point. This cost can be reduced by observing that $\phi(x)$ is a *piecewise polynomial* (PP) of degree p in each interval. Its PP coefficients can be obtained from

$$\begin{aligned} \phi(x) &= \sum_{\alpha=0}^{p} c_{i+\alpha} \sum_{k=0}^{p} V^i_{k\alpha} (x - x_i)^k \\ &= \sum_{k=0}^{p} \Pi^i_k (x - x_i)^k, \qquad \Pi^i_k = \sum_{\alpha=0}^{p} c_{i+\alpha} V^i_{k\alpha} \end{aligned} \tag{4}$$

Once the $N(p+1)$ PP coefficients Π^i_k have been calculated from the spline expansion coefficients $c_{i+\alpha}$, only $2p$ flops per point are required to obtain the field value, usingthe Horner's rule described previously.

3.2 2D case

Extension for the spline expansion and the PP representation for $\phi(x, y)$ is straightforwards and yields, for $x_i \leq x < x_{i+1}$, $y_j \leq y < y_{j+1}$:

$$\begin{aligned} \phi(x,y) &= \sum_{\alpha=0}^{p1} \sum_{\beta=0}^{p2} c_{i+\alpha,j+\beta} \Lambda_{i+\alpha}(x) \Lambda_{j+\beta}(y) \\ \phi(x,y) &= \sum_{k=0}^{p1} \sum_{l=0}^{p2} \Pi^{ij}_{kl} (x - x_i)^k (y - y_j)^l, \qquad \Pi^{ij}_{kl} = \sum_{\alpha=0}^{p1} \sum_{\beta=0}^{p2} c_{i+\alpha,j+\beta} V^i_{k\alpha} V^j_{l\beta}, \end{aligned} \tag{5}$$

where $V^i_{k\alpha}$ and $V^j_{l\beta}$ are the PP coefficients of the splines $\Lambda_{i+\alpha}(x)$ and $\Lambda_{j+\beta}(y)$ respectively. Assuming the same spline order p in both x and y, the flop counts per point for the 2 representations are respectively $2(3p+2)(p+1) \sim 6p^2$ and $2p(p+2) \sim 2p^2$, while the storages required for the spline coefficients c and the PP coefficients Π are $(N+p)^2 \sim N^2$ and $N^2(p+1)^2$ respectively.

3.3 Implementation in BSPLINES

The PP representation is selected by default in BSPLINES, *unless* the logical keyword `NLPPFORM` is set to `.FALSE.` when calling the spline initialization routine `SET_SPLINE`. The flop counts per point for both methods are summarized in the table below

	1D	2D
Spline expansion	$2(p+1)^2$	$2(3p+2)(p+1)$
PP representation	$2p$	$2p(p+2)$

The routine `GRIDVAL` computes the value of the field or one of its derivatives. The first call to this routine computes the PP coefficients Π if `NLPPFORM=.TRUE.` is selected or just store the spline coefficients c in the spline internal data otherwise. In the following calls to `GRIDVAL`, c should not be passed to the routines.

Notice that the PP representation requires to store the $N^2(p+1)^2$ PP coefficients in the 2D case, which is still acceptable. In the 3D case, this storage requirement becomes $N^3(p+1)^3$ which can be prohibitive! In this case the less efficient *Spline expansion* formulation should be selected.

In the *particle loop*, the routine `GETGRAD` which computes the function and all its first partial derivatives at once should be called instead of `GRIDVAL`.

4 Particle localization(locintv)

In both charge assignment and field interpolation, finding in which interval of the spatial grid the particle is localized should be first performed. This operation is trivial for the case of an equidistant mesh. For non-equidistant mesh, an *equidistant fine* mesh and its mapping to the actual mesh are first constructed in the spline initialization routine `SET_SPLINE` and used to localize the particles in the routine `LOCINTV`.

5 Performances

From the considerations above, using BSPLINES to perform the charge assignment and field interpolation in 2D and 3D particle codes might result in large overheads because of the large number of calls to the routines `BASFUN` to compute the splines or `GETGRAD` to perform the field interpolation at a *single* particle position. In the following, the performances the 2D linearized gyrokinetic code GYGLES which has been adapted to use BSPLINES are analyzed. Vectorization by grouping the particles for both charge assignment and field interpolation is then proposed as a way to speed up these two operations when using BSPLINES.

5.1 Scalar performances

Optimization of the scalar versions of `BASFUN` and `GETGRAD` (when these routines are called with a *single* particle) is performed essentially by

- Minimizing the flop counts and reducing redundant operations.
- Unrolling small loops, for example the loop over the $p+1$ splines that are non-zero at a given position, for small p.
- Define all routines called by `BASFUN` and `GETGRAD` as *internal procedures*.
- Rearranging the memory layout of the multi-dimension array containing the PP coefficients of the spline.

The timings of the charge and current assignment (assign), the particle pusher (push) and the main time loop for a 5 time step run of GYGLES, on an Intel Xeon X5570 (hpcff.fz-juelich.de), using 4 MPI processes and Intel Fortran 12.1.2 are summarized in the following table

	T_0(s)	T_1(s)	T_2(s)	T_1/T_0	T_2/T_1
assign	1.454E+01	2.126E+01	2.259E+01	1.46	1.06
push	2.536E+01	3.080E+01	3.144E+01	1.21	1.02
mainloop	4.197E+01	5.955E+01	6.149E+01	1.42	1.03

where T_0 is the time in seconds obtained with the original code while T_1 and T_2 are the times obtained with BSPLINES, respectively using an *equidistant* and *non-equidistant* radial mesh. In all the 3 runs, a quadratic splines were used. The small difference between *equidistant* and *non-equidistant* mesh comes mainly from the particle localization.

The same run on an Intel Xeon E5-2680 (helios.iferc-csc.org), using the same Intel compiler (with AVX instructions) yields

	T_0(s)	T_1(s)	T_2(s)	T_1/T_0	T_2/T_1
assign	1.093E+01	1.987E+01	2.086E+01	1.82	1.05
push	2.385E+01	2.868E+01	2.994E+01	1.20	1.04
mainloop	3.656E+01	5.411E+01	5.598E+01	1.48	1.03

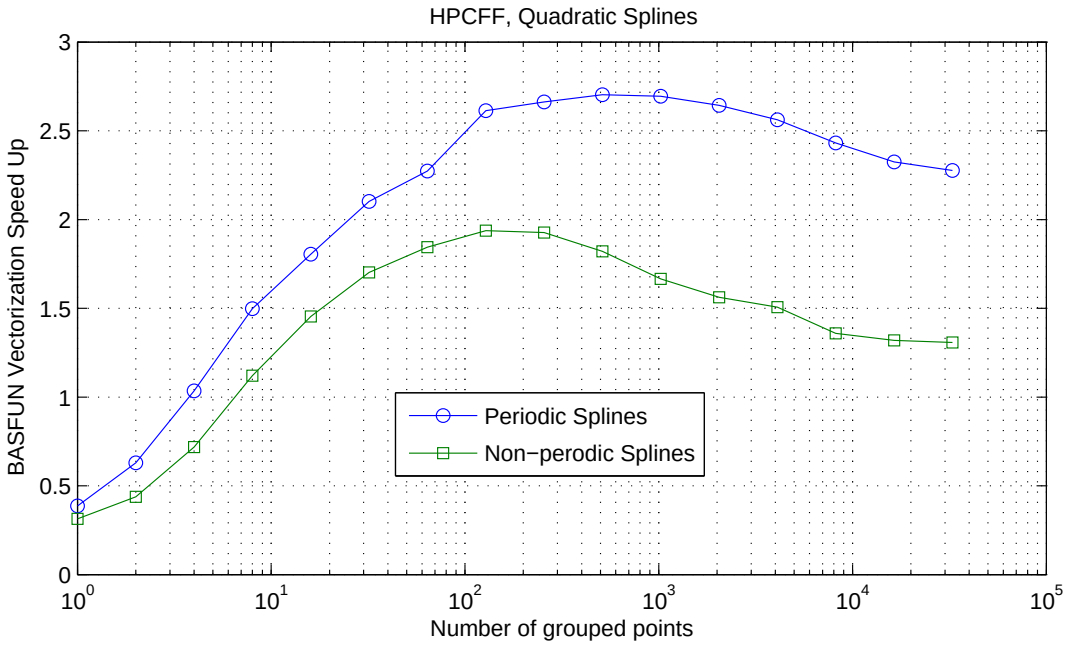
5.2 Speed up by vectorization

As found in the last section, using external routines from BSPLINES instead of *hard coding* the spline computations results in a slowing down of 40–50% for the main time loop. As will shown later, this problem could be solved by *grouping* the particles and using the vectorized `BASFUN` and `GETGRAD` routines. Such particle grouping can be done for example, by replacing the usual particle loop by the following Fortran code fragment

```
nset = npart/ngroup
IF(MODULO(npt, ngroup).NE.0) nset = nset+1
i2 = 0
DO is=1,nset
  i1 = i2+1
  i2 = MIN(i2+ngroup,npart)
  CALL basfun(x(i1:i2), ...)
END DO
```

where `npart` particles are partitioned into `nset` groups, each containing at most `ngroup` particles. Vectorization of the routines `BASFUN` and `GETGRAD` is achieved by moving whenever is possible the loop over the `ngroup` particles into the innermost loop.

The vectorization performances shown in Fig .1 and Fig .2, respectively for `BASFUN` and `GETGRAD` are obtained using version 12.1.2 of Intel compiler on an Intel Xeon X5570 (hpcff.fz-juelich.de). With a speedup of at least 2 for quadratic splines, the slowing down found previously in the scalar version could be likely compensated. The new AVX instructions present in the recent Intel Xeon E5-2680 (helios.iferc-csc.org) seems to improve somewhat the vectorization performance as shown in Fig .3 and Fig .4.



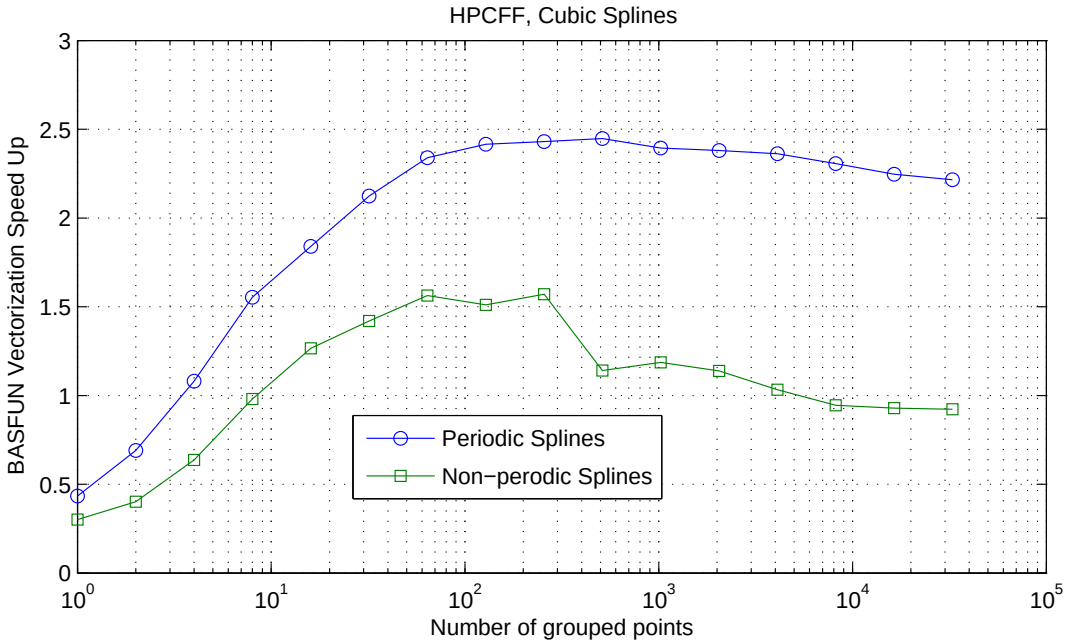


Figure 1: In this test, 10^5 particles are distributed randomly on an equidistant mesh of 64 intervals. On each point, all the $p+1$ splines are computed. The particle localization routine `locintv` is included in the timing. In order to have a good statistics in the measurements, $1'000$ iterations of the particle loop are considered.

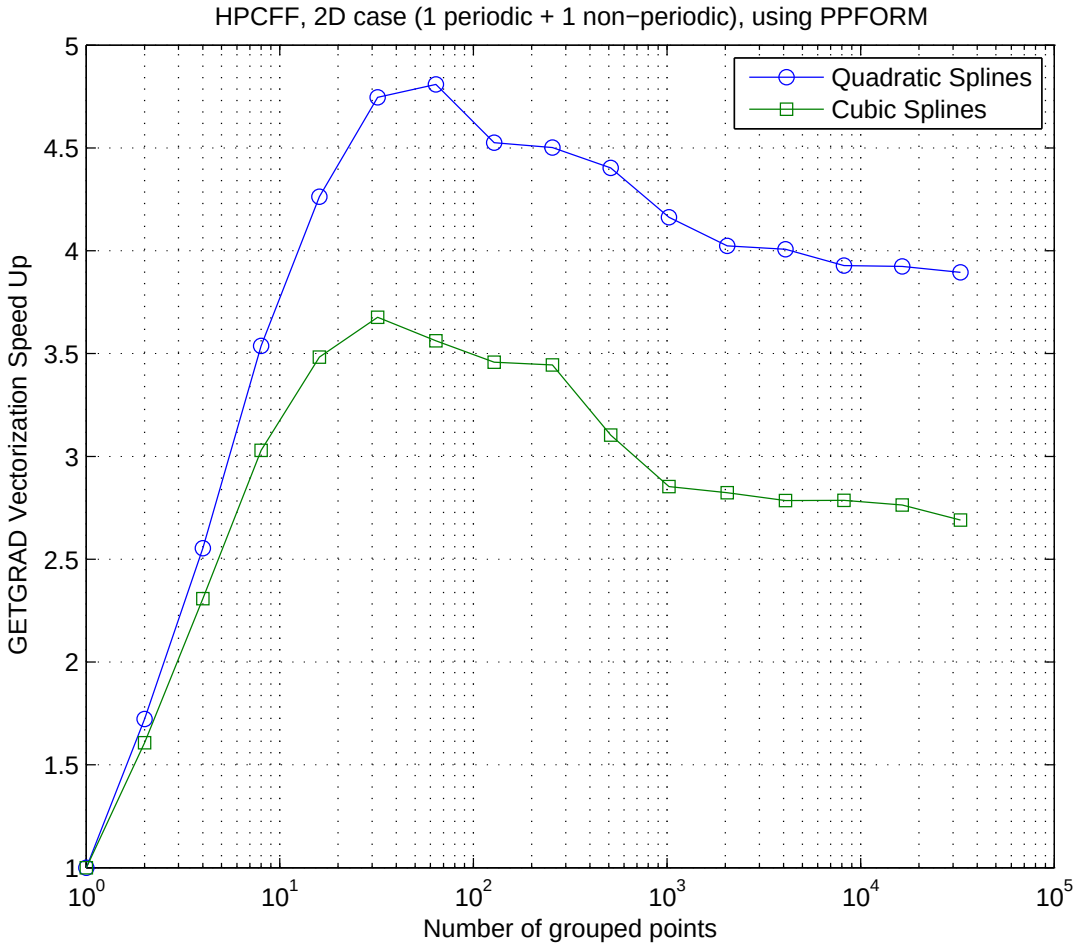


Figure 2: In this test, 10^5 particles are distributed randomly on an equidistant 2D (x, y) mesh of 64×64 intervals, where the coordinate y is periodic. On each point, the function together with its gradient are computed, using the PP representation. The particle localization routine `locintv` is included in the timing. In order to have a good statistics in the measurements, 100 iterations of the particle loop are considered.

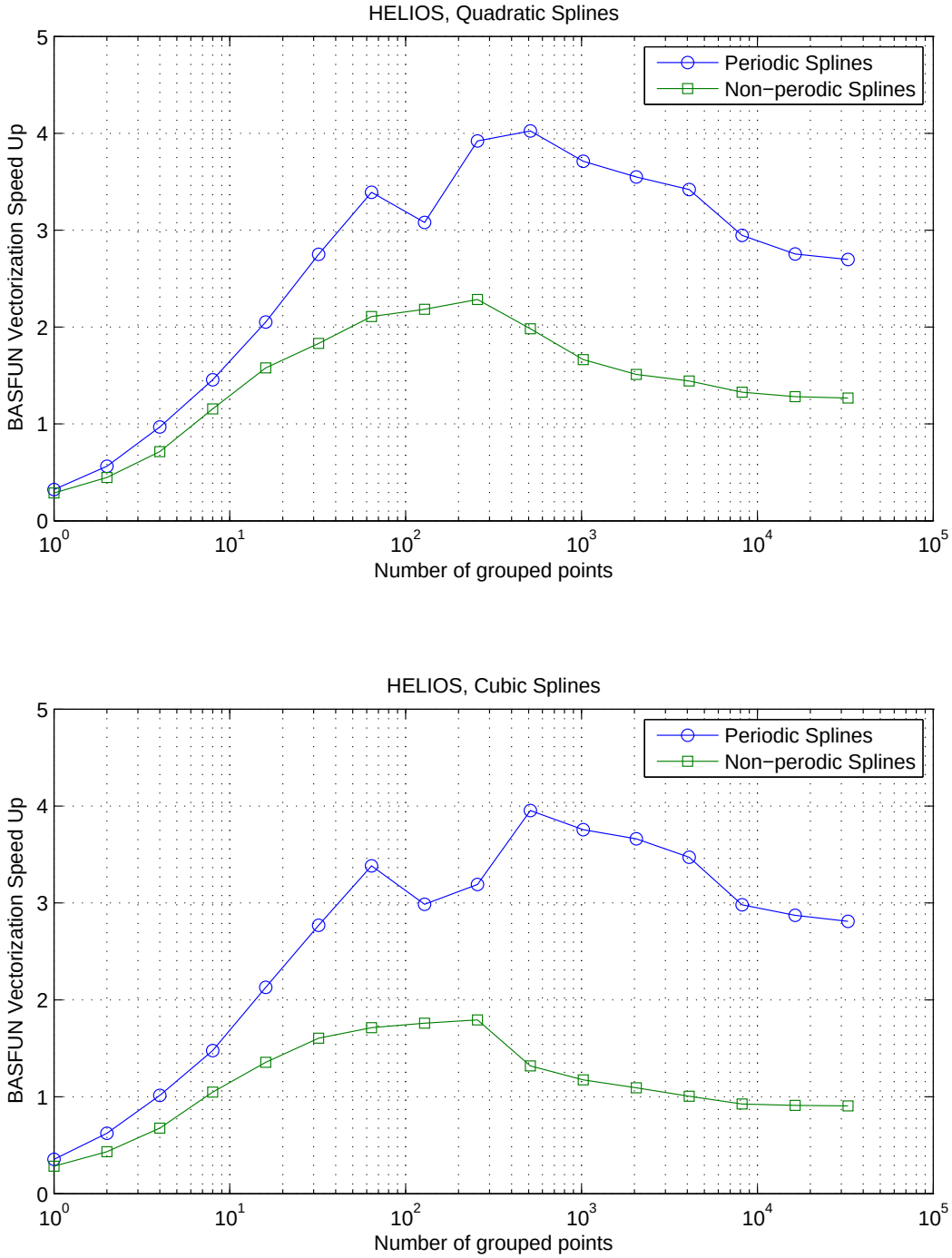


Figure 3: In this test, 10^5 particles are distributed randomly on an equidistant mesh of 64 intervals. On each point, all the $p+1$ splines are computed. The particle localization routine `locintv` is included in the timing. In order to have a good statistics in the measurements, $1'000$ iterations of the particle loop are considered.

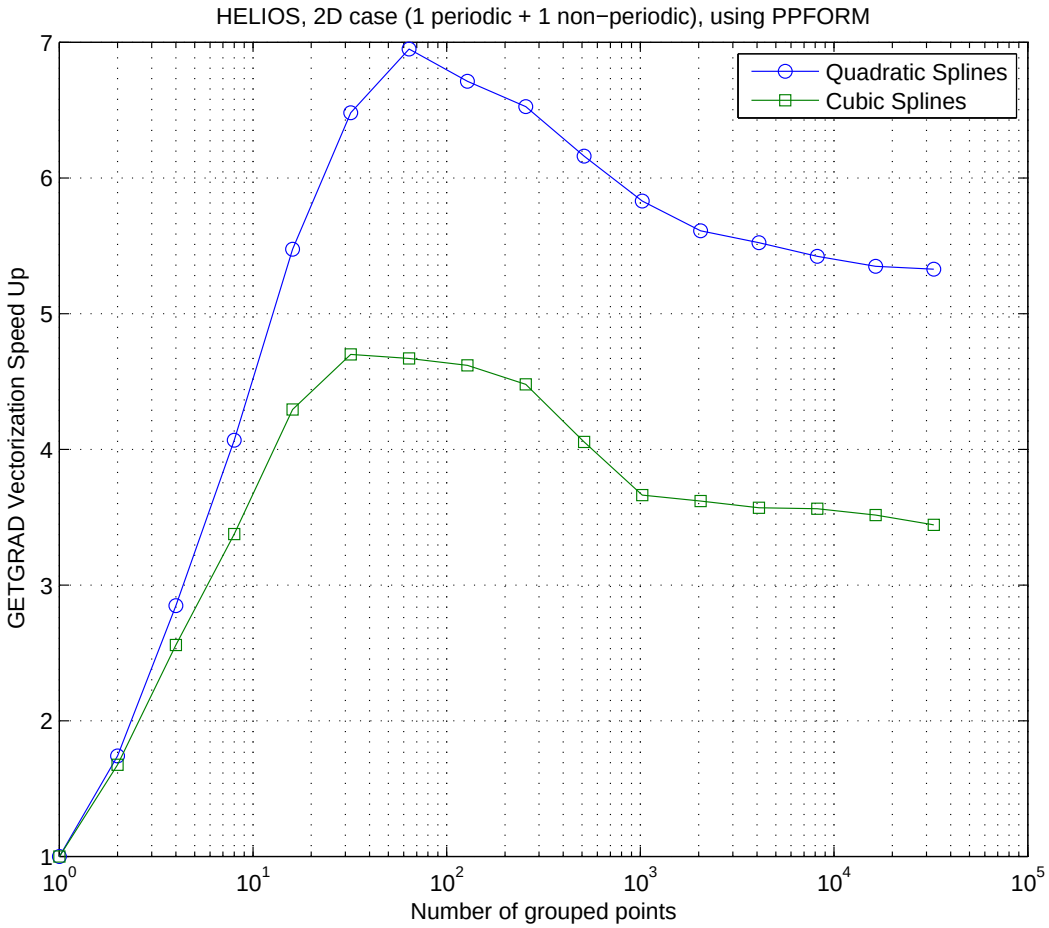


Figure 4: In this test, 10^5 particles are distributed randomly on an equidistant 2D (x, y) mesh of 64×64 intervals, where the coordinate y is periodic. On each point, the function together with its gradient are computed, using the PP representation. The particle localization routine `locintv` is included in the timing. In order to have a good statistics in the measurements, 100 iterations of the particle loop are considered.

References

[1] `BSPLINES` Reference Guide.