Informal introduction to program structure of spectral interpolation in `nek5000`

By Azad Noorani, Adam Peplinski, and Philipp Schlatter

Linné FLOW Centre and Swedish e-Science Research Centre (SeRC),
KTH Mechanics, Royal Institute of Technology, SE-100 44 Stockholm, Sweden

Technical Report

The algorithm of the interpolation implementation in the spectral element code `nek5000` is documented informally. The original code is written by James Lottes at Argonne National Laboratories. The various steps of the operations are generally described and visualised for a typical deformed mesh. The corresponding routines and their argument lists for each stage of the interpolation are also explained. The memory structure of the implementation is briefly discussed. Finally, the code overview of the routines is presented.

1. Introduction

Ever since its introduction by Patera (1984), the spectral element method (SEM) has been evolved as one of the most reliable techniques to simulate turbulent flows in complex geometries with high accuracy. SEM is essentially a high-order weighted residual method that provides both high accuracy and geometrical flexibility simultaneously. This method can be considered as combination of classical spectral methods (see *e.g.* Canuto *et al.* 2006) and the more general, but mostly low-order finite-element method (FEM). While, originally SEM was introduced as a straight-forward combination of multiple subdomains individually discretised based on Chebyshev polynomials with suitable matching conditions, it soon evolved towards a discretisation based on Legendre polynomials. As such the computational domain is divided into a number of hexahedral local spectral elements within which the solution is represented by the previously mentioned polynomials at the Gauss–Lobatto–Legendre (GLL) quadrature points. Both Chebyshev and Legendre polynomials are taken from the family of high-order orthogonal polynomials. Such orthogonality is crucial for SEM as it leads to diagonal mass matrices and provides a set of accurate integration and derivation rules.

With SEM there is a large amount of computational work *local* to each element, and only the global communication (*i.e.* the exchange of boundary data and the pressure coupling) is required on the *global* mesh. This has led to a number of very efficient implementations of SEM in computer programs.

Among others, the code `nek5000` is standing out with the very good performance in terms of stability and scalability. The code is developed by Fischer *et al.* (2008) at the Argonne National Laboratory (ANL) and is directly originated from the initial efforts by the group that was led by Patera at MIT, at that time as a code named Nekton. The code `nek5000` employs a standardised and portable message-passing interface (MPI) platform for its parallelism based on `FORTRAN` and `C` and is proven to be scaling up to millions of processors.

In SEM, unfortunately, the GLL points are not uniformly distributed, which makes the primary mesh results somewhat unsuitable particularly for postprocessing. At the same time, having computed the coefficients of the test functions, the hard part is already paved such that with little effort any computed variable in any arbitrary position can be evaluated. In that sense the interpolation looks ideal as the accuracy will not be compromised. It turns out that the situation is not that simple and there are extensive difficulties in performing a proper interpolation for SEM mesh.

Such high-accuracy interpolation from a set of fixed GLL points becomes vital *e.g.* for immersed boundary methods or in the study of Lagrangian particles. In the latter, the particulate phase data is required to be obtained at arbitrary positions within the computational domain at each time step of the simulation. Provided that the collective analysis of the Lagrangian data is only meaningful for large number of particles, such interpolation scheme should be able to handle millions of evaluations of the field properties for arbitrary positions within the domain. Working within `nek5000`, the interpolation routine is necessary to be parallelised also, otherwise it will appear as a bottleneck when increasing the number of concurrent processors.

For those reasons, an interpolation routine with spectral accuracy is implemented in `nek5000` by James Lottes. The routine is validated and verified for the accuracy and stability to perform interpolation that preserve the spectral accuracy of the field (Eulerian) solution. The aim of the current document is to provide an informal and descriptive documentation of the implementation of the interpolation code in `nek5000`. The rigorous documentation of the operations along with the details of the memory structure, dependencies and variable naming are, however, out of the scope of this technical report. For brevity, we specifically focus on the interpolation of 2D simulations. Of course, the 3D routines are very similar but slightly more complicated due to the fact that from a topological point of view, in 2D simulations, one needs to handle vertices and faces only; however, in 3D cases also edges of the computational mesh come into play.

In the following, firstly we report the algorithm before presenting the name and the arguments of the subroutines performing the implementation. The code overview also will be shown at the end which will be followed by examining a test case and some concluding remarks.

2. Algorithm

The SEM discretisation is based on a decomposition of the global domain Ω into K non-overlapping, high-order sub-domains Ω^k (elements) building a locally structured and globally unstructured base,

$$\Omega = \bigcup_{k=1}^{K} \Omega^k.$$

Each element Ω^k is a deformed quadrilateral $\hat{\Omega} = [-1,1]^D$ in $\mathbb{R}^D$ (reference element), with defined affine mapping of the local coordinates $\boldsymbol{r} \in \hat{\Omega}$ to the physical ones $\boldsymbol{x}^k(\boldsymbol{r}) \in \Omega^k$. Within each given element in $\mathbb{R}^2$ a scalar field $f(\boldsymbol{x})$ can be represented as

$$f(\boldsymbol{x}^k(\boldsymbol{r}))|_{\Omega^k} = \sum_{i=0}^{N}\sum_{j=0}^{N} f^k_{ij} h_i(r_1) h_j(r_2), \tag{1}$$

where $h_i(r_i)$ is the Lagrange polynomial of degree $\leq N$, $f^k_{ij} = f(\boldsymbol{x}^k(\zeta_i,\zeta_j))|_{\Omega^k}$ are field nodal values taken at Gauss–Lobatto–Legendre points (ζ_i,ζ_j), and $h_i(\zeta_j) = \delta_{ij}$. In particular the (isoparametric) coordinate mapping takes the form

$$\boldsymbol{x}^k(\boldsymbol{r})|_{\Omega^k} = \sum_{i=0}^{N}\sum_{j=0}^{N} \boldsymbol{x}^k_{ij} h_i(r_1) h_j(r_2), \tag{2}$$

with $\boldsymbol{x}^k_{ij}$ being physical positions of GLL points.

For every calculated variable `nek5000` stores its nodal values, which allows to reconstruct $f(\boldsymbol{x})|_{\Omega^k}$ and perform interpolation to any particular position $\boldsymbol{x}_i \in \Omega^k$. As interpolation is performed in the reference element, it requires first, identification of the proper element k and next, finding local coordinates $\boldsymbol{r}_i$ corresponding to particular physical position $\boldsymbol{x}_i$. This could in principle be done by inverting the coordinate mapping. However, inverting equation (2) is not a straight-forward task for `nek5000`, as the code can be run on a distributed memory computers with the elements scattered among a set of processes. In this case each process keeps detailed information (variables nodal values, *e.g.* $\boldsymbol{x}^k_{ij}$) about local elements only, and has only restricted information about element to processor mapping with no physical coordinates of non-local elements. On the other hand, memory constraints do not allow to duplicate $\boldsymbol{x}^k_{ij}$ on all the processes and build the global coordinate mapping, that would be local to every process. That is why the $\boldsymbol{x}_i \rightarrow \boldsymbol{r}_i$ mapping requires first finding the process or P owning the given point.

The `findpts` module performs the point $\rightarrow$ process mapping by the use of an uniform, rectangular mesh (global hash mesh) covering the whole domain Ω (see figure 1 *b*). For this mesh we consider mesh cells instead of mesh nodes. Each processor maps its sub-domain on this hash mesh by marking all mesh cells entirely or partially covered by the local sub-domain as belonging to given process P. This allows to build a global list of processes at least partially owning to a given hash cell. The efficiency of this approach depends strongly on the resolution of the hash mesh. In the current implementation, the extent of the hash mesh is defined by the minimum ($\boldsymbol{x}^{min}$) and maximum-values ($\boldsymbol{x}^{max}$) of the physical coordinates $\boldsymbol{x} \in \Omega$, and its resolution is defined by the number of processes NP and the *global_hash_size* parameter, which sets the maximum number of cells for a given processor that it can store. The number of rows (nr) and columns (nc) of the hash mesh is $nr = nc \sim (NP * global\_hash\_size)^{1/D}$, giving relatively high resolution of the hash mesh which makes it impossible for local storage of the whole array. With the hash cells ordered by rows and columns, the global data distribution is based on the hash cell index $hi = i_1 + nr*(i_2+nc*i_3)$ (with i_j being integer coordinates of the cell in the mesh along X and Y directions), and the data of hi cell are stored at process $mod(hi, NP)$ with local index hi/NP. To get the list of possible process owners of a given point $\boldsymbol{x}_i$ one has to calculate first its global index $pi = i_1 + nr * (i_2 + nc * i_3)$ with $i_j = int(nr * (x_j - x_j^{min})/(x_j^{max} - x_j^{min}))$, and next retrieve the process list from node $mod(pi, NP)$ using local point index pi/NP.

A similar operation is performed on the local level, when the initial point $\rightarrow$ element mapping is generated. In the local case a uniform, rectangular mesh (local hash mesh) is used to keep the information about all the local elements owning, at least partially, a given local hash cell (see figure 1 *b*). The general work-flow is analogous to the point $\rightarrow$ process mapping starting with the calculation of the local point index pi, except the fact that there is no global communication step, as the local hash mesh covers the local sub-domain only and is stored locally. In this case $\boldsymbol{x}^{min}$ and $\boldsymbol{x}^{max}$ give the extent of the local sub-domain, and the mesh resolution is defined by $local\_hash\_size^{1/D}$. With *global_hash_size* and *local_hash_size* equal, the local hash mesh would be of higher resolution than the global one, as it covers the local sub-domain only.

After performing the point $\rightarrow$ process and the initial point $\rightarrow$ element mappings there should be a set of processes with non-empty list of possible element owners. At this stage the ownership information is very crude and has to be refined to get a single processor/element pair.

As usually the volume assigned by the hash mesh to given element is significantly larger than the element itself, the element bounding box (*obbox*) information is compared with the point position $\boldsymbol{x}_i$. *obbox* contains the element range (extent of the smallest rectangular box covering the element; analogous to $\boldsymbol{x}^{min}$ and $\boldsymbol{x}^{max}$ for the hash meshes), centre $\boldsymbol{x}_c^k$ and inverse Jacobian ${J_c^k}^{-1}$ of the coordinate mapping (2) taken at the element centre. Calculation of the element ranges takes care of its possible deformation and is performed on the higher resolution mesh using Chebyshev–Lobatto nodes for polynomial order $M > N$ using piecewise D-linear bounds. In addition, to avoid the possibility of non-matching element faces and points falling through "cracks" between elements, every element range is expanded by the *bbox_tol* parameter, giving the extent of bounding box:

$$\triangle \boldsymbol{x}_{box} = (\boldsymbol{x}_{box}^{max} - \boldsymbol{x}_{box}^{min}) * (1 + bbox\_tol).$$

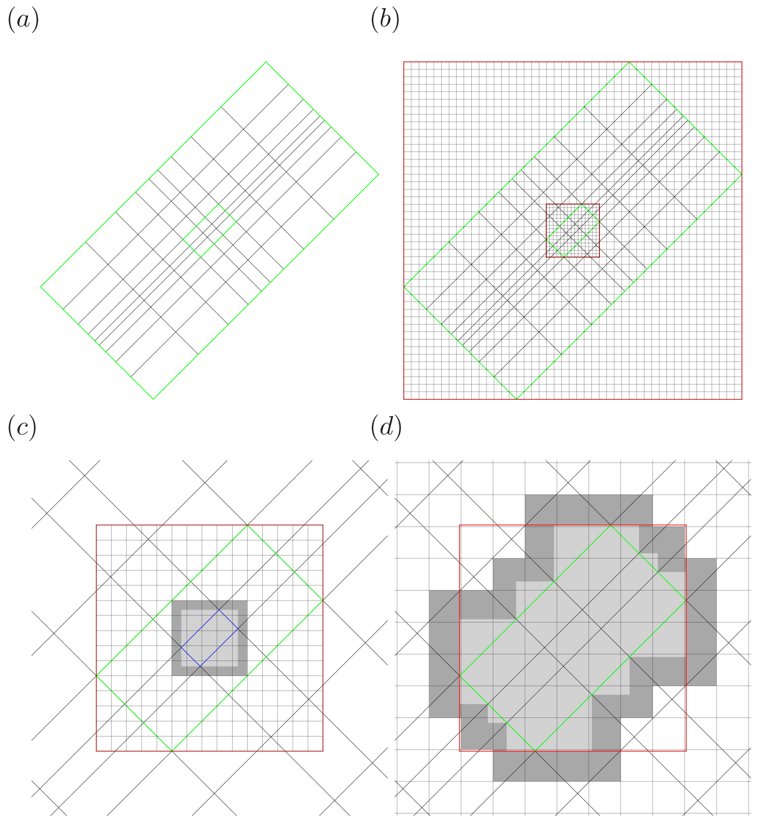

FIGURE 1. *(a)* A typical computational mesh (only the spectral elements are shown). The total 81 elements of the domain is mapped onto 9 processors. A local (3 × 3 elements) subdomain, that belongs to a certain processor, is marked in the middle. *(b)* The red bounding box outside the full domain marks the extent of global hash mesh. The border of the local hash mesh for the subdomain in *(a)* is also marked as red but with higher resolution mesh inside. *(c)* Zoomed view of the local subdomain with local hash mesh. An element in the middle of the subdomain is marked with blue. Light grey indicates the bounding box for that specific element. Dark grey shows the extended mapping of the bounding box to the local hash mesh for the element marked in blue. *(d)* Overview of the global hash mesh for the subdomain. The bounding box of the subdomain is marked in red. Note the local hash mesh is not shown in this plot. Light grey is the sum of all the element bounding boxes belonging to the given subdomain. Dark grey shows the global mapping of such bounding boxes.

The effect of curved element faces and boundary box expansion are taken into account for element centre and inverse Jacobian as well.

Refining of the ownership information is done in three steps. First, all the elements with $\boldsymbol{x}_i$ lying outside their ranges are discarded. Next, an approximate value of the element local coordinates (in the reference element) $\boldsymbol{r}_p$ are calculated using linear approximation

$$\triangle \boldsymbol{r}_p = J^{-1}(\boldsymbol{r}_p)\triangle \boldsymbol{x}_p, \tag{3}$$

where $\triangle \boldsymbol{x}_p$ and $\triangle \boldsymbol{r}_p$ are updates of the physical and local coordinates. As first estimate $\triangle \boldsymbol{x}_p = \boldsymbol{x}_i - \boldsymbol{x}_c^k$ (a distance of the point from the element centre), $J^{-1}(\boldsymbol{r}_p = \boldsymbol{0}) = J_c^{k^{-1}}$ and finally $\boldsymbol{r}_p = \boldsymbol{0} + \triangle \boldsymbol{r}_p$ are taken. At this stage all elements with initial $\boldsymbol{r}$ outside the reference box $[-1, 1]^D$ are discarded.

On the remaining elements a more exact calculation of $\boldsymbol{r}$ is performed using Newton iterations of (3), with $\triangle \boldsymbol{x}_p = \boldsymbol{x}_i - \boldsymbol{x}(\boldsymbol{r}_p + \triangle \boldsymbol{r}_p)$ (according to equation (2)). A starting point for iterations is a GLL point $\boldsymbol{x}_{ij}^k$ closest to $\boldsymbol{x}_i$. In addition an exact procedure for Newton iterations depend on the position $\boldsymbol{r}_p$ of the point within the reference element, and for $\boldsymbol{r}_p$ close to element borders (faces, edges and vertices) part of the Hessian matrix (second derivatives) is taken into account. In the current implementation maximal number of Newton iterations is set to 50. Convergence is tested both on $\triangle \boldsymbol{r}_p$ and $\triangle \boldsymbol{x}_p$ using *newt_tol* parameter. For local coordinates the L_1 norm is checked with the convergence criterion

$$\sum_{i=0}^{D} |\triangle r_{pi}| < newt\_tol,$$

and in the physical coordinates space the square of the L_2 norm of the residual from the previous ($\triangle \boldsymbol{x}_p^{pr}$; taking into account round-off errors) and current ($\triangle \boldsymbol{x}_p^{cr}$) iterations are compared

$$\|\triangle \boldsymbol{x}_p^{pr}\|^2 - \|\triangle \boldsymbol{x}_p^{pr} - J(\boldsymbol{r}_p^{pr})\triangle \boldsymbol{r}_p^{pr}\|^2 < newt\_tol * \|\triangle \boldsymbol{x}_p^{cr}\|^2.$$

Depending on the iteration result a point can be finally marked as internal (*CODE_INTERNAL*), border (*CODE_BORDER*) or not found (*CODE_NOT_-FOUND*). Element ownership is evident for internal points, as they are located relatively far away from the element borders, but uncertain for border points. That is why for the border points, and not found points the L_2 norm of the final residual in the physical coordinates $\triangle \boldsymbol{x}_p$ from different elements is compared, and the element with the smallest distance is chosen. This allows to find the best fit for given point. Finally, calculated local coordinates $\boldsymbol{r}_i$ of internal and border points are used in equation (1) to perform the actual interpolation. A visualisation of the procedure for a typical 2D domain is offered by figure 1.

3. C-Fortran interface

The `findpts` module performs in parallel interpolation of a given variable on a set of arbitrary points. It is written in `C` using `MPI` and is located under jl sub-directory. There are five routines that can be directly called from the FORTRAN code of nek5000:

- findpts_setup,
- findpts_free,
- findpts,
- findpts_eval,
- findpts_eval_local,

located in findpts.c. Most of these routines are executed by the hpts routine (postpro.f), which is the main interface for probe reading in nek5000 and is a good example of findpts usage. A short description of these routines are presented in the next sections.

3.1. *findpts_setup*

This is the first routine to be called for the findpts module. It is responsible for allocation of all the internal memory structures and initialisation of all mesh dependent variables (*e.g.* global and local hash arrays or element bounding box information). All these variables are independent on specific point positions and are later used both for $\boldsymbol{x}_i \rightarrow \boldsymbol{r}_i$ mapping and the final variable interpolation. In general findpts_setup could be called more than once for different grids generating separate setups distinguished by setup handle, which is the only output of this routine. handle is used as structure identified by the rest of findpts routines and has to be saved by calling the corresponding FORTRAN routine.

Example call:

```
call findpts_setup(h, comm,np, ndim, xm,ym,zm,
nr,ns,nt,nel,mr,ms,mt, bbox_tol, loc_hash_size,
gbl_hash_size, npt_max,newt_tol)
```

Argument list (in parenthesis nek5000 default):

- Output:
  - h - setup handle
- Input:
  - comm - MPI communicator (nekcomm)
  - np - process number (np)
  - ndim - mesh dimension (ndim)
  - xm,ym,zm - $\boldsymbol{x}_{ij}^k$ physical positions of GLL points (xm1,ym1,zm1)
  - nr,ns,nt - element dimensions (nx1, ny1, nz1)
  - nel - process local element number (nelt)
  - mr,ms,mt - finer mesh size for bounding box computation; must be larger than nr,ns,nt for correctness (2*nx1, 2*ny1, 2*nz1)
  - bbox_tol - relative size to expand bounding boxes (0.1)

  - `loc_hash_size, gbl_hash_size` - maximum number of integers locally stored in hash tables (lx1*ly1*lz1*lelt)
  - `npt_max` - number of points to iterate on simultaneously (256). This parameter allows to limit memory requirements when a large number of points is considered.
  - `newt_tol` - tolerance for Newton iteration (1E-13)

For more information see `findpts.c`.

3.2. *findpts_free*

This routine is optional. It can be used to free memory used by setup structures marked by `handle`. Should be called after all interpolation is finished.

Example call:

```
call findpts_free(h)
```

Argument list:

- Input:
  - `h` - setup handle; returned by `findpts_setup`

For more information see `findpts.c`.

3.3. *findpts*

This routine performs physical → local coordinates mapping for the given set of arbitrary points. It can be run after `findpts_setup` is called and all the mesh dependent variables are set. This routine performs all the steps with point → process, point → element and $\boldsymbol{x}_i \rightarrow \boldsymbol{r}_i^k$ mapping using hash arrays and performing Newton iterations. For every point it returns the owning processor and element, local coordinates in the reference element, together with an error code and the measure of mapping quality (distance in physical coordinates between interpolation position and its mapped counterpart $\|\boldsymbol{x}_i - \boldsymbol{x}(\boldsymbol{r}_i)\|^2$; see equation (2)). This routine has to be executed every time the set of points has changed.

Example call:

```
call findpts(h,code,  code_str, proc, proc_str,
el, el_str, r, r_str, dist2, dist2_str, x, x_str,
y, y_str, z, z_str, npt)
```

Argument list (in parenthesis `nek5000` default):

- Output:
  - `code` - error code for coordinates mapping:

        * 0 - inner point
        * 1 - border point
        * 2 - not found
    - `proc, el` - process and element ownership
    - `r` - local coordinates $\boldsymbol{r}_i$
    - `dist2` - final residual of Newton iterations $\|\boldsymbol{x}_i - \boldsymbol{x}(\boldsymbol{r}_i)\|^2$
- Input:
    - `h` - setup handle; returned by `findpts_setup`
    - `code_str, proc_str, el_str, r_str, dist2_str, x_str, y_str, z_str` - array stride (ndim for `r_str` and 1 for the rest)
    - `x, y, z` - physical coordinates for set of points
    - `npt` - number of points

For more information see `findpts.c`.

3.4. *findpts_eval*

This routine is used to interpolate a variable stored in the `input_field` array on the set of the points defined by `code, proc, el` and `r`, which are given by `findpts`. It performs equation (1) using matrix-matrix multiplication. This routine can be executed in parallel.

Example call:

```
call findpts_eval(h, out, out_str, code,
code_str, proc, proc_str, el, el_str, r,
r_str, npt, input_field)
```

Argument list (in parenthesis `nek5000` default):

- Output:
    - `out` - interpolated values
- Input:
    - `h` - setup handle; returned by `findpts_setup`
    - `code` - error code for coordinates mapping; returned by `findpts`
    - `proc, el` - process and element ownership; returned by `findpts`
    - `r` - local coordinates $\boldsymbol{r}_i$; returned by `findpts`
    - `out_str, code_str, proc_str, el_str, r_str` - array stride (ndim for `r_str` and 1 for the rest)
    - `npt` - number of points
    - `input_field` - interpolated variable

For more information see `findpts.c`.

3.5. *findpts_eval_local*

This is local counterpart of `findpts_eval`. It assumes all the points are local for given process. Notice, there are no error code for this routine.

Example call:

```
call findpts_eval_local(h, out, out_str, el, el_str,
r, r_str, npt, input_field)
```

Argument list (in parenthesis `nek5000` default):

- Output:
  - `out` - interpolated values
- Input:
  - `h` - setup handle; returned by `findpts_setup`
  - `el` - element ownership; returned by `findpts`
  - `r` - local coordinates $\boldsymbol{r}_i$; returned by `findpts`
  - `out_str, el_str, r_str` - array stride (ndim for `r_str` and 1 for the rest)
  - `npt` - number of points
  - `input_field` - interpolated variable

For more information see `findpts.c`.

4. Memory structures

In this section we briefly discuss the main memory structures in the `findpts` module to give a general overview of the code structure. As some names of the routines and memory structures are identical at different code levels (*e.g.* two *v.* three-dimensional or local *v.* global operations) and are adjusted by pre-processing rules, we give the source file name in parenthesis to avoid confusion.

The top level memory structure is

```
struct handle {
void *data;
unsigned ndim; };
(findpts.c)
```

which is allocated as an array and allows to identify different mesh setups with the integer `handle` parameter. It keeps information of the grid dimension D and a pointer to the global and local mesh information contained by

```
struct findpts_data {
struct crystal cr;
struct findpts_local_data local;
struct hash_data hash;};
```

(`findpts_imp.h`)

`cr`, `local` and `hash` are here a global communicator setup (crystal router; not discussed here), local elements data and global hash array respectively.

The last one contains information about the global hash mesh covering the whole domain

```
struct hash_data {
ulong hash_n;
struct dbl_range bnd[D];
double fac[D];
uint *offset;};
```

(`findpts_imp.h`)

where:

- `hash_n` - number of cells in X, Y, Z directions ($lceil((NP * glb\_hash\_size)^{1/D})$),
- `bnd[D]` - global domain ranges ($\boldsymbol{x}^{min}$ and $\boldsymbol{x}^{max}$),
- `fac[D]` - inverse cell size in X, Y, Z directions ($hash\_n/(x_j^{max} - x_j^{min})$),
- `offset` - process ownership.

`offset` is build of two parts: first `hash_n`^{D}/`NP` $+ 1$ entrances give the offset information, and the rest is reserved for the process list. The list of processes owning a given cell with index `id` is located between `offset(offset(id))` and `offset(offset(id+1)-1)`.

Information about the local set of elements is stored under

```
struct findpts_local_data {
unsigned ntot;
const double *elx[D];
struct obbox *obb;
struct hash_data hd;
struct findpts_el_data fed;
double tol;};
```

(`findpts_local_imp.h`)

Consecutive fields are:

- `ntot` - number of GLL points in element (N^D; $nx1 * ny1 * nz1$),
- `elx[D]` - pointer to physical positions of GLL points $\boldsymbol{x}_{ij}^k$ (points `xm1`, `ym1` and `zm1` arrays in `nek5000`),

- `obb` - element bounding box information,
- `hd` - local hash array,
- `fed` - local element setup,
- `tol` - tolerance of Newton iterations (*newt_tol*).

The element bounding box information is given *e.g.* by (two-dimensional version)

```
struct obbox_2 {
double c0[2], A[4];
struct dbl_range x[2]; };
```

(`obbox.h`)

with:

- `c0[D]` - element centre ($\boldsymbol{x}_c^k$),
- `A[D*D]` - inverse Jacobian ${J_c^k}^{-1}$ of the coordinate mapping (2) taken at $\boldsymbol{x}_c^k$,
- `x[D]` - element range

The local hash array (`struct hash_data` in `findpts_local_imp.h`) is similar to the global one except `hash_n` $= lceil(loc\_hash\_size^{1/D})$, and an additional integer variable giving maximum number of owning elements for given cell.

The last discussed memory structure is two-dimensional version

```
struct findpts_el_data_2 {
unsigned npt_max;
struct findpts_el_pt_2 *p;

unsigned n[2];
double *z[2];
lagrange_fun *lag[2];
double *lag_data[2];
double *wtend[2];

const double *x[2];

unsigned side_init;
double *sides;
struct findpts_el_gedge_2 edge[4];
struct findpts_el_gpt_2 pt[4];
double *work;};
(findpts_el.h)
```

It contains element-specific data like polynomial order in X, Y, Z directions (`n[D]`), the GLL points list ζ_i (`z[D]`), the Lagrange coefficients (`lag_data[D]`) or the function to calculate values and derivatives of the base functions h_i at arbitrary points (`lag[D]`), and keeps temporary arrays for the Newton iterations and other necessary calculations (e.g. `x[D], edge[D*D], pt[D*D]`).

There are a number of other memory structures (e.g. `findpts_el_pt_D` used for Newton iterations) allocated in `findpts` and omitted in our introduction. Detailed information about them can be found in the source code.

5. Code overview

In this section we give a general overview of the workflow of the `findpts` module focusing on three routines `findpts_setup`, `findpts` and `findpts_eval`. Our intention is marking the most important steps rather than going into all implementation details. That is why we do not cover all of the important aspects of the implementation and for the more specific issues we direct the reader to the source code. To avoid confusion we give in parenthesis the name of the source file containing a given routine. Depending on the mesh dimension the capital `D` at the end of the routine or memory structure names should be replaced with `ndim` value.

5.1. *findpts_setup*

This routine allocates internal memory structures and initialises mesh dependent variables. The following graph shows the main routines called by `findpts_setup`:

- ♠ **findpts_setup** (`findpts.c`)
    - ♦ **setup_aux_D** (`findpts_imp.h`)
        - ♣ **findpts_local_setup** (`findpts_local_imp.h`)
            - ■ **obbox_calc_D** (`obbox.c`)
            - ■ **hash_build** (`findpts_local_imp.h`)
            - ■ **findpts_el_setup** (`findpts_el_D.c`)
        - ♣ **hash_build** (`findpts_imp.h`)

The `findpts_setup` (`findpts.c`) allocates `handle_array` (a top level memory structure `handle`) and sets the mesh dimension `ndim` for a given mesh setup. Depending on `ndim` it also allocates the proper `findpts_data_D` structure (`data` pointer in `handle`) and executes `setup_aux_D` (`findpts_imp.h`), which is responsible for setting all the mesh dependent data. `setup_aux_D` performs two major steps collecting information of the process-local subdomain (executing `findpts_local_setup`; `findpts_local_imp.h`) and generating process ownership using global hash array (executing `hash_build`; `findpts_imp.h`).

Operation on the local subdomain (performed by `findpts_local_setup`) starts with the generation of the bounding box information for each element. It is done with the routine `obbox_calc_D` (`obbox.c`), which (after allocation of the `obbox_D` structure) generates lower and higher resolution meshes based on the Gauss–Lobatto–Legendre and Chebyshev–Lobatto nodes, respectively. The higher resolution mesh is used to calculate linear bounds for the lower resolution base functions $h_i(r)$, which are necessary to get more accurate approximation of element faces (three-dimensional case) and edges (two-dimensional case). This information is used to calculate element ranges ($\boldsymbol{x}_{box}^{max}$, $\boldsymbol{x}_{box}^{min}$), the element centres $\boldsymbol{x}_c^k$ and inverse Jacobians ${J_c^k}^{-1}$.

In the next step the element ownership is generated by `hash_build` (`findpts_local_imp.h`) using a local hash array. Notice this routine is different from `hash_build` (`findpts_imp.h`) used for construction of the process ownership. It starts from evaluation of the subdomain extent (based on the elements bounding boxes) and calculation of the hash cell size depending on *local_hash_size*. Next the bounding box of each element is mapped on the local hash mesh giving hash cell $\rightarrow$ element correspondence, which is used to build the hash offset array.

The last operation on the local subdomain consists of the allocation of the work arrays and setting up element specific data under the `findpts_el_data_D` structure and is performed by `findpts_el_setup` (`findpts_el_D.c`) routine. It provides the polynomial order in X, Y, Z directions (`n[D]`), the GLL points list ζ_i (`z[D]`), the Lagrange coefficients (`lag_data[D]`) and the pointer to the function calculating values and derivatives of the base functions h_i at arbitrary point (`lag[D]`).

The final step in `setup_aux_D` is calling `hash_build` (`findpts_imp.h`) to build the process ownership. To some extent it is analogous to the operation performed on the local hash mesh, although the global mesh is not stored locally and the global communication step has to be added. At first the global domain extent is estimated based on the extent of the local hash meshes, and the size of the global hash cell is calculated depending on *global_hash_size*. Next the bounding box of every single element in the given subdomain (not the local hash mesh) is mapped on the global hash mesh providing more exact representation of the local subdomain on the global hash mesh. Constructed this way, the local mask of each process is later transferred to the cell index $\rightarrow$ process list and redistributed between processes to build global database.

5.2. *findpts*

This routine performs physical → local coordinates mapping for the given set of arbitrary points. The following graph shows the main routines called by `findpts`:

- ♠ **findpts** (`findpts.c`)
  - ♦ **findpts_D** (`findpts_input.h`)
    - ♣ **findpts_local** (`findpts_local_imp.h`)
      - ■ **map_points_to_els** (`findpts_local_imp.h`)
      - ■ **findpts_el_D** (`findpts_el_D.c`)
    - ♣ global communication
    - ♣ **findpts_local** (`findpts_local_imp.h`)
    - ♣ global communication

This routine does not allocate any major memory structures, but uses the work spaces and variables provided by `findpts_setup`. In addition no point-related information is permanently stored in the internal memory structures of the `findpts` module and all such data is transferred through the argument list. The `findpts` routine takes as an input a list of points (physical coordinates) and returns the mapping information for every point (processor and element ownership with local coordinates and error code). It is important to mention here that similar to the grid the total set of points is distributed between processors (with no point duplicates) giving local point subsets that are not correlated with the local grid subdomains. As a result the point → element mapping requires global communication.

Similar to `findpts_setup` the first step in `findpts` allows to choose between two- and three-dimensional cases by calling proper `findpts_D`, which performs all the mapping operations. To minimise global communication `findpts_D` starts with the local search by calling `findpts_local` (`findpts_local_imp.h`), which performs mapping of the given (in this case local) subset of points to the local subdomain. This process starts with the element ownership that is performed by `map_points_to_els` (`findpts_local_imp.h`). After marking all points as not found (*CODE_NOT_-FOUND*) `map_points_to_els` routine calculates for every point its local index *pi* and uses local hash array to identify the possible element owners. Next it uses the element bounding box information (*obbox*) comparing the point position with element ranges ($\boldsymbol{x}_{box}^{max}$, $\boldsymbol{x}_{box}^{min}$) and provides the first linear estimate for the local point coordinates $\boldsymbol{r}_i$ based on the element centre $\boldsymbol{x}_c^k$ and inverse Jacobian $J_c^{k^{-1}}$ (see equation (3)). This allows to shorten the list of the possible element owners.

This list is later refined by performing Newton iterations of equation (3), which is done by `findpts_el_D` (`findpts_el_D.c`). To improve performance points are sorted by elements and all points residing in a single element are grouped into sets smaller than `npt_max`. For every considered point iterations start at a GLL point $\boldsymbol{x}_{ij}^k$ closest to $\boldsymbol{x}_i$. As an exact procedure for Newton iterations depends on the position $\boldsymbol{r}_p$ of the point within the reference element, after every iteration points are sorted and grouped into bins related to the element interior, faces, edges and vertices. In the next iteration each bin is treated separately. The result of the Newton iteration is a set of internal flags, the possible point-local coordinates $\boldsymbol{r}_i$ and residuals $\boldsymbol{x}_i - \boldsymbol{x}(\boldsymbol{r}_i)$. Depending on the internal flags and residuals points are marked as element internal (*CODE_INTERNAL*), border (*CODE_BORDER*) or not found (*CODE_NOT_FOUND*).

`findpts_local` is followed by the global communication block, where all the border and non-found points are transferred to new possible process owners. The list of possible owners is provided by the global hash array using the global point hash index. After the new set of points for local mapping is generated each process executes `findpts_local` second time. At this stage given point can be duplicated on a number of processes, which independently perform local point $\rightarrow$ element mapping. To remove duplicates the results of the Newton iteration are transferred back to the source process in the second global communication block. Next each process takes care of creating the final ownership list picking up the mappings with the smallest residual.

5.3. *findpts_eval*

This routine is used to interpolate given variable on a set of points. The following graph shows main routines called by `findpts_eval`:

- ♠ **findpts_eval** (`findpts.c`)
  - ♦ **findpts_eval_D** (`findpts_input.h`)
    - ♣ $\boxed{\text{global communication}}$
    - ♣ **findpts_local_eval** (`findpts_local_imp.h`)
      - ■ **findpts_el_eval_D** (`findpts_el_D.c`)
    - ♣ $\boxed{\text{global communication}}$

Like `findpts` this routine does not allocate any memory structures and gets all required point information through an argument list. It requires also global communication to redistribute points between processors, as `findpts` does not change the initial point distribution.

As in other routines `findpts_eval` starts from choosing two- and three-dimensional setup by calling proper `findpts_eval_D` (`findpts_input.h`). `findpts_eval_D` starts with the global communication block, where internal and border points are redistributed between processors according to their ownership. Next the resulting set of points is sorted on every process according to element ownership and then the `findpts_local_eval` (`findpts_local_imp.h`) is called. Inside `findpts_local_eval` all points residing in a single element are grouped into sets smaller than `npt_max` and sent to `findpts_el_eval_D` (`findpts_el_D.c`), where the interpolation takes place. At this stage values of the base functions h_i are calculated at the local position $\boldsymbol{r}_i$ of every point in the set and a tensor product is used to get interpolated value of a given variable. Finally, in the second global communication block interpolated values are transferred back to the source processes.

6. Scaling test

A Lagrangian particle tracking (LPT) scheme is developed by Noorani (2014) in which the field data in the position of point particles are evaluated using the current spectral interpolation. As a test case, therefore, a standard particle-laden turbulent channel flow in a large box is simulated in order to quantify the impact of large number of interpolation operations in every time step on the overall performance of the code in real situations. The box size is set to be $(12\pi, 2, 6\pi)$ in (x, y, z) directions respectively with $(100, 10, 100)$ elements distributed in each of these directions. Each spectral element is resolved with 8^3 internal Gauss–Lobatto–Legendre nodes which results in 51 200 000 grid points. A total of 1 530 000 points particles are distributed uniformly in the computational domain. The simulation is continued until the particle dispersion reached a statistically stationary state in which the distribution is highly non-homogeneous and large number of particles accumulate in specific regions of the domain.

A strong parallel scaling is evaluated by enhancing the number of processors while monitoring the workload measured as the averaged wall-clock time per time step during the simulation. The values for the entire solver, the flow solver `nek5000` and the LPT module are shown in figure 2 where `nek5000` shows an excellent linear scalability for all considered number of processors. Increasing the number of processors, however, the LPT solver slowly deviates from the linear scaling (figure 2 *b*) and correspondingly the whole solver (figure 2 *c*). On the other hand, due to the fact that the total time that is spent in the LPT module is merely 10% of the total simulation time per time-step, this deficit is barely visible in figure 2 *(c)*. It is important to note that more than 95% of the wall-time spent in the LPT module is due to the interpolation. The remaining 5% are spent to advect the particles; a comparably cheap and in particular completely parallel operation.

A similar test is conducted examining a turbulent channel flow with box size of $4\pi \times 2 \times 2\pi$ laden with uniformly distributed particles. In that case the number of processors is fixed and set to 256 and, the number of particles per processor is increased. As illustrated in figure 2 *(d)*, the workload increases linearly, which is expected. This, of course, only confirms that the particles are essentially independent of each other. The current scaling tests are all performed on a Cray XE-6 machine at PDC (KTH).

In the context of LPT and related to interpolation routine, a few possibilities are mentioned in the following that might help improving the efficiency of

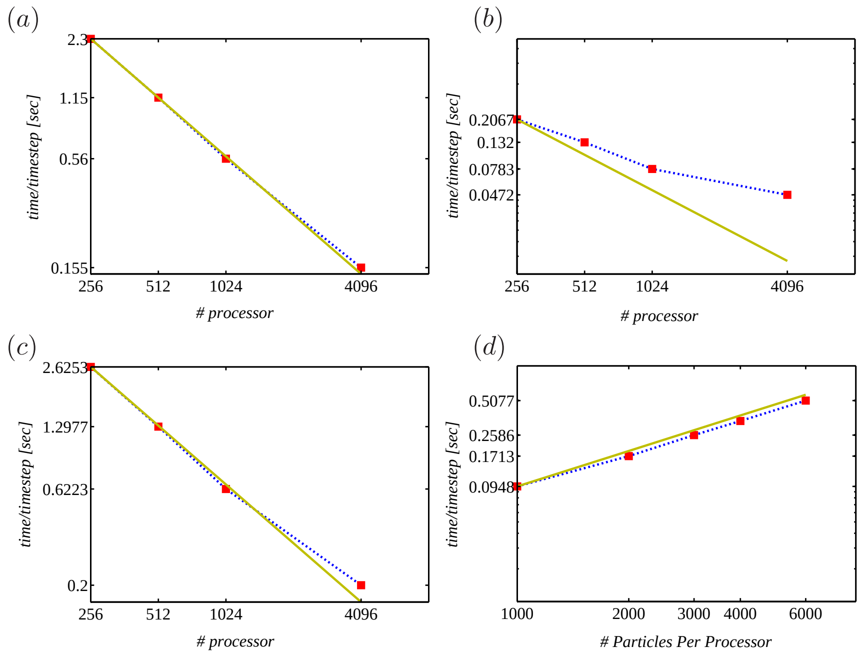

FIGURE 2. Averaged wall-time per timestep of the simulation for different number of processors (🟥) performed by *(a)* fluid solver `nek5000`, *(b)* LPT solver, *(c)* the whole code. The test is a turbulent channel with a large box (51 200 000 grid points) laden with 1 530 000 inertial particles that are reached statistically stationary state. *(d)* Simulation wall-time for different number of particles per processor at a fixed 256 processors performed in a small-box turbulent channel. The particles are distributed uniformly. The solid line illustrates linear scaling. Taken from Noorani (2014).

the interpolation scheme for particle tracking. The most straight-forward choice is to decrease the search time for the processor ID that is handling a certain particle. Assuming the time step of simulations is small, the particles might spend relatively long time (many time steps) in a specific subdomain that belongs to a specific processor before traversing to another processor subdomain. The search for the processor IDs, then can be postponed to occasional failure of processors in providing field data at particle position. This, however, might not be practical if the time steps are large or particles are traversing quickly from one sub-domain to another. On the other hand, the CFL condition applicable for the fluid put restrictions on the time steps anyway to reasonable amounts. Similar assumption can be used to apply a *history* scheme. In that, each particle possesses a unique ID, so it is possible to track down the particle IDs per processor ID. The fact that particles move into the neighbouring blocks can be taken into considerations, which makes it possible to search the particle information only on those processors which handle these neighbouring blocks. Naturally one could also allow for local particle evaluations. In other words, the particle computations are performed exactly on the same processor that handles its position. Problems arise in this scheme if the majority of particles accumulate in a small region where it needs to be handled by few processors while the rest of the processors are particle free. In this situation the amount of local workload might be large and a more sophisticated load balancing might be required. Defining a threshold after which the workloads are distributed to the other processors with less particles might be useful in this case. This *re-shuffling* could be also costly at times, but does not need to be performed at every time step. Finally, a simpler possibility to reduce the overall cost of the particles is to use different time steps for the fluid and particle phase. Usually the maximum possible time step is lower for the fluid, which means that particulate phase could be updated less frequently in comparison to the fluid phase. This update can be performed as a function of particle relaxation time; *i.e.* the heaviest particles allowing the largest time step.

7. Conclusion and outlook

In this technical report we documented the algorithm of the interpolation routines for the spectral element code `nek5000`. The code was originally developed by James Lottes at Argonne National Laboratory. Here, we described the various steps of the operations. These steps are visualised for a typical deformed 2D mesh in Cartesian coordinates. The corresponding routines and their argument lists for each stage of the operation were also explained. The memory structure pertaining to the implementation was briefly discussed. At the end, we presented the overview of the interpolation routines as they appear in the code.

References

Canuto, C., Hussaini, M.Y., Quarteroni, A. & Zang, T.A. 2006 *Spectral Methods: Fundamentals in Single Domains*. Springer.

Fischer, P. F., Lottes, J. W. & Kerkemeier, S. G. 2008 `nek5000` Web page. `http://nek5000.mcs.anl.gov`.

Noorani, A. 2014 Lagrangian particles in turbulence and complex geometries. *Tech. Rep.*. KTH, Mechanics, licenciate Thesis.

Patera, A. T. 1984 A spectral element method for fluid dynamics: laminar flow in a channel expansion. *J. Comput. Phys.* **54** (3), 468–488.