

Notes based on H. Wang's meshless method presentation for the FSI team

Juha Jeronen

September 13, 2011

Abstract

This technical note explains how to approximate derivatives of a known function, defined as a set of values on a point cloud, where each point may have arbitrary Cartesian coordinates. This is a meshless method based on Taylor series expansion in a local set of nearest neighbors. It can be used for, e.g., integration of initial boundary value problems using explicit methods (e.g. RK4).

Also, a simple $O(dN \log N)$ time algorithm for finding the nearest neighbors in d dimensions is presented for the sake of completeness.

Derivative approximation — the weighted least squares meshless method

We will present the *weighted least squares meshless method* (WLSQ). It belongs to the class of finite point methods (collocation methods), so in spirit it is similar to finite differences. Because the method only differentiates known quantities, it is best suited for time evolution problems (initial boundary value problems; IBVP), which are solved with explicit time integration methods such as RK4. Dirichlet boundary conditions are very easy to enforce; Neumann and Robin are much harder.

To start with, consider a point cloud of N points in \mathbb{R}^d . Let i denote the index of the current node under consideration, and k the index of one of its nearest neighbors. (For finding the m nearest neighbors of a point in a point cloud, refer to the final section of this document.)

Let $f = f(x_k)$, $k = 1, \dots, N$ be a function defined on the point cloud. Here we will only consider the two-dimensional case ($d = 2$) for simplicity. Let us shorten the notation by defining $f_k := f(x_k)$.

We would like to be able to approximate the derivatives of f at the point x_i , using only the point cloud data. This has applications in e.g. explicit time integration of PDEs with given initial data.

Below, we will only consider the problem for one node x_i . Trivially, the same procedure can be repeated for each node.

Using multivariate Taylor expansion up to the second order, we can write f_k (value of f at one of the nearest neighbors) in terms of f_i as

$$f_k = f_i + h_k a_1 + \ell_k a_2 + \frac{h_k^2}{2} a_3 + h_k \ell_k a_4 + \frac{\ell_k^2}{2} a_5 + O(h_k^3, \ell_k^3), \quad (1)$$

where $h_k = (x_k)_1 - (x_i)_1$ (i.e. the x component of the vector from x_i to x_k) and $\ell_k = (x_k)_2 - (x_i)_2$ (respectively, the y component).

Note that generally, we must expand up to as many orders as is the highest derivative we wish to approximate. We will assume here for simplicity that we are building the approximation for a second-order problem.

If we drop the asymptotic term, we get the approximation

$$\bar{f}_k = f_i + h_k a_1 + \ell_k a_2 + \frac{h_k^2}{2} a_3 + h_k \ell_k a_4 + \frac{\ell_k^2}{2} a_5. \quad (2)$$

By the Taylor expansion, we would expect to have

$$\begin{aligned}
a_1 &= \frac{\partial f_k}{\partial x} \Big|_{x=x_i} \\
a_2 &= \frac{\partial f_k}{\partial y} \Big|_{x=x_i} \\
a_3 &= \frac{\partial^2 f_k}{\partial x^2} \Big|_{x=x_i} \\
a_4 &= \frac{\partial^2 f_k}{\partial x \partial y} \Big|_{x=x_i} \\
a_5 &= \frac{\partial^2 f_k}{\partial y^2} \Big|_{x=x_i} ,
\end{aligned} \tag{3}$$

if f was defined on all of \mathbb{R}^2 . Our problem is thus to find a good approximation for the values of the a_j .

Let us denote

$$\begin{aligned}
c_k^{(1)} &:= h_k \\
c_k^{(2)} &:= \ell_k \\
c_k^{(3)} &:= \frac{h_k^2}{2} \\
c_k^{(4)} &:= h_k \ell_k \\
c_k^{(5)} &:= \frac{\ell_k^2}{2} .
\end{aligned} \tag{4}$$

We would like to minimize the approximation error. Let us denote the error as

$$e_k := f_k - \bar{f}_k . \tag{5}$$

We proceed by making a least squares approximation. Let

$$G := \frac{1}{2} \sum_k e_k^2 \tag{6}$$

where the sum is taken over the nearest-neighbor set of x_i . The least-squares approximation is given by the minimum

$$\min_{a_j} G ,$$

i.e. such values for the a_j that they minimize the squared error G .

The minimum of the function $G = G(a_1, \dots, a_5)$ is necessarily at an extremum point. Thus, we set all its partial derivatives to zero (w.r.t the a_j):

$$\frac{\partial G}{\partial a_j} = 0 \quad \forall j = 1, \dots, 5 . \tag{7}$$

Because $G \geq 0$ for any values of the a_j and it is a quadratic function, this point is also necessarily the minimum. Thus, solving equation (7) gives us the optimal a_j .

One important thing to notice here is that we of course do not have the value of the asymptotic term $O(h_k^3, \ell_k^3)$ in (1). However, we do not need equation (1) for computing the error (5). This is because we already have the value of f_k directly, since it is one of the points in the data! Thus, for any set of values for the a_j , the error (5) can be computed (by replacing f_k with the data point in question and computing \bar{f}_k from (2)).

Let us write out (7). We have

$$\begin{aligned}\frac{\partial G}{\partial a_j} &= \sum_k e_k \frac{\partial e_k}{\partial a_j} \\ &= \sum_k [f_k - \bar{f}_k(a_1, \dots, a_5)] \left[-\frac{\partial \bar{f}_k}{\partial a_j} \right] = 0 \quad \forall j = 1, \dots, 5,\end{aligned}\tag{8}$$

where we have replaced e_k by the difference of data f_k and the interpolate \bar{f}_k , as noted above.

Now the rest is essentially technique. Expanding the first \bar{f}_k in (8) and taking the minus sign in front, we have

$$-\sum_k \left([f_k - f_i - c_k^{(1)} a_1 - c_k^{(2)} a_2 - c_k^{(3)} a_3 - c_k^{(4)} a_4 - c_k^{(5)} a_5] \left[\frac{\partial \bar{f}_k}{\partial a_j} \right] \right) = 0 \quad \forall j.$$

This can be rewritten as a standard linear equation system

$$A\mathbf{a} = \mathbf{b},\tag{9}$$

where

$$\mathbf{a} = (a_1, \dots, a_5)^T$$

are the unknowns, and the j th component of the load vector \mathbf{b} is

$$b_j = \sum_k [f_k - f_i] \left[\frac{\partial \bar{f}_k}{\partial a_j} \right] = \sum_k [f_k - f_i] c_k^{(j)},\tag{10}$$

where in the last form we have used (2) and the definition (4). The sum, like above, is taken over the set of nearest neighbors. Especially note that, as required, all the quantities on the right-hand side of (10) are known.

The element A_{jn} of the coefficient matrix A is

$$A_{jn} = \sum_k c_k^{(n)} c_k^{(j)}.\tag{11}$$

This sum, too, is taken over the set of nearest neighbors. The matrix A is symmetric, $A = A^T$.

Solving (9), by e.g. pivoted Gaussian elimination (routine DGEVS in LAPACK, operator \ in MATLAB, scipy.linalg.solve() in Python, ...), produces the derivative approximations a_j , up to the second order.

Note that both A and \mathbf{b} depend on the node index i ! That is, each node comes with its own A and \mathbf{b} , and thus (10) and (11) must be re-evaluated for each node where we wish to obtain the derivative approximation.

Sensitivity of the solution

It is possible to also obtain the sensitivity of the solution \mathbf{a} in terms of small changes in the values of the data points f_k . Consider, formally, manipulating (9) into

$$\mathbf{a}(f_k) = A^{-1} \cdot \mathbf{b}(f_k).$$

Differentiation on both sides, and writing the equation in component form, gives (the matrix A is constant w.r.t. f_k)

$$\begin{aligned}\frac{\partial a_j}{\partial f_k} &= \sum_n (A^{-1})_{jn} \frac{\partial b_n}{\partial f_k} \\ &= \sum_n (A^{-1})_{jn} c_k^{(n)}, \quad \forall j = 1, \dots, 5,\end{aligned}$$

which can be rewritten as

$$A \frac{\partial \mathbf{a}}{\partial f_k} = (c_k^{(1)}, c_k^{(2)}, c_k^{(3)}, c_k^{(4)}, c_k^{(5)})^T. \quad (12)$$

Thus we have a linear equation system, from which the sensitivities of each of the a_j in terms of the node value f_k can be solved. By changing k on the right-hand side and solving again for each k , we obtain the sensitivity with respect to each of the neighbors. (Note that there is **no** sum over k , except inside the matrix A .)

This sensitivity result may be useful for forcing Neumann boundary conditions to hold during IBVP integration (at each timestep, changing the values at the nodes belonging to the boundary until the BC is satisfied).

Again, it should be noted that equation (12) is valid for the node i , and in principle must be solved separately for each node.

However, we observe that the sensitivities depend on the (local) geometry of the point cloud only. Recall the definitions of A and $c_k^{(n)}$, equations (11) and (4); the only quantities that appear are the pairwise node distances. This observation holds for any point cloud.

If there is some regularity in the geometry, it may be possible to reuse (some of) the results. As a special case, if we have a regular Cartesian grid, the $c_k^{(n)}$ are constant with respect to k , and thus in this special case only, the sensitivities at each node follow the same pattern. This extends easily to other regular geometries; e.g. for a grid based on the nodes of a hexagonal tiling, there will be only two kinds of nodes with regard to the sensitivity. The strength of the method, however, lies in being able to handle irregular geometries: in the general case, one does not need to assume anything about the distribution of the points.

Finding nearest neighbors — a simple algorithm

In this section, we look into the problem of searching a given point cloud for nearest neighbors. We consider finding the neighbors within a given distance R from a given point, and finding the m nearest neighbors of a given point, with m given.

An example MATLAB/Octave implementation of the ideas presented in this section is provided in `find_neighbors.m` (in the SAVU project git repository).

Finding all neighbors within distance R For a static point cloud (in the sense of not changing during the simulation), the nearest neighbor search problem can be solved in $O(d N \log N)$ time (where N is the number of points in the whole cloud, and d is the dimensionality of the space \mathbb{R}^d where the points live in) using an indexed search procedure. For a moving point cloud, the “expensive” $O(d N \log N)$ step must be re-performed at each timestep.

Initially, we create a sorted index of the data based on the coordinates on each axis. This gives us d sorted vectors of (coordinate along j th axis, point ID) pairs. This enables us to search for the set of points, which belong to a given interval on, say, the x axis ($j = 1$; correspondingly for the other axes). Each sort finishes in $O(N \log N)$ time, and only needs to be done once (or until the point cloud changes; then we must re-index). Then, indexed search on this data can be done using the binary search procedure in $O(\log N)$ time for each dimension.

To find the neighbors within distance R of a point with given coordinates in \mathbb{R}^d (allowed to be a point belonging to the cloud, but not necessary), we first search along each axis, producing d filtered index sets in each of which the coordinates on the k th axis match the desired interval $[(x_0)_k - R/2, (x_0)_k + R/2]$. Taking the set intersection of the result sets gives us the neighbor set within distance R in the sense of the ℓ^∞ metric. The next step is to filter the result further.

An important property here is that because $\|x\|_{\ell^\infty} \leq \|x\|_{\ell^p}$ for all $1 \leq p < \infty$, the ℓ^∞ neighbor set encloses all other ℓ^p neighbor sets, including the Euclidean neighbor set (with $p = 2$). Thus, all these other neighbor sets can be produced by filtering the ℓ^∞ neighbor set.

The ℓ^∞ neighbor set, with M points, is for any practically interesting R much smaller than the whole cloud ($M \ll N$). Thus, linear filtering of the result set, which takes $O(M)$ time, is not a major cost.

To find the ℓ^2 (Euclidean) neighbor set, we simply construct a new result set, including in it only those points in the ℓ^∞ neighbor set that also satisfy the ℓ^2 distance requirement $\|x_j - x_0\|_{\ell^2} \leq R$.

Finding the m nearest neighbors Finally, consider the question of finding R such that within this radius, there are exactly m neighbors (where m is user-specified). This provides us a nearest-neighbor search procedure for user-definable m , which is what we need in the meshless method.

We start from some $R = r_0$ (this can be e.g. some function of the size of the bounding box of the data, which can be trivially found in $O(N)$ time, and the number of points in the data (e.g. assuming them to have uniform density and estimating average R from that)). We then do a logarithmic search, counting the neighbors within radius R and, based on the result, we either double or halve R at each step.

By this logarithmic search, we may get lucky and hit an R where there are exactly m neighbors. In this case, we stop and return the current neighbor set.

But most often, we will find an interval $R \in [R_1, R_2]$ where R_1 has less than m neighbors, and $R_2 = 2R_1$ has more than m . This interval can be refined using binary search on the variable R . This produces a sequence of shrinking intervals $[R_a, R_b]$, which converges onto (some) correct R . This works, because the number of neighbors as a function of distance is a monotonic (although discontinuous and piecewise constant) function. We stop the search once we find an R which has exactly m neighbors.

The final pitfall is that in an arbitrary point cloud, for any given point, the cloud may contain exactly two (or more) points at the exact same distance from it. In these cases, there might not exist a distance with exactly m neighbors for the given point! To protect against this possibility, we set a tolerance $\varepsilon > 0$ for the length of the search interval $[R_a, R_b]$ in the above procedure. If no matching R has been found, and $R_b - R_a < \varepsilon$, we stop the search and return the neighbor set at R_b (along with e.g. an error code or some other signal, so that the calling end knows that extra neighbors have been returned).