Automatic Differentiation of Moving Least Squares

Tim Langlois

In one of our projects, we compressed the mode shapes used for modal sound synthesis by fitting moving least squares approximations to them. Basically, instead of storing the mode values at each vertex of the mesh, we stored a small set of control points, and used MLS to interpolate between them. Part of the compression process involved non-linear optimization, which required the jacobian of the error function. This would have been very complicated to write out analytically, but was easy to do automatically.

The optimization problem was as follows. We were given a mesh where each vertex i had a position $\tilde{\mathbf{p}}_i$ and a value $\tilde{f_i}$ (in reality these values were vectors, but for simplicity just assume they are scalars). We also had a set of control points. Each control point j had a position $\mathbf{p}_j$ and a value f_j. We use MLS to interpolate the values at vertices. We'll use v_i to refer to the interpolated value at position i. Specifically, at a vertex i, we find a polynomial approximation which minimizes

$$\sum_j \|v_i - f_i\|^2 * \theta(\mathbf{p}_j - \tilde{\mathbf{p}}_i)$$

Theta is a weighting function which gives more weight to control points that are closer to vertex i, i.e., we want the approximation to be more accurate near vertex i. Then we can evaluate the polynomial at position $\tilde{\mathbf{p}}_\mathbf{i}$ to get v_i.

The polynomial can be found by solving a linear system. For example, suppose we are using quadratic polynomials. That means they will be of the form

$$a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4yz + a_5xz + a_6x + a_7y + a_8z + a_9$$

We want to solve for the coefficients a_i. Assume that we have n control points. We build the system

$$\begin{bmatrix} \theta(\mathbf{p}_0 - \tilde{\mathbf{p}}_i) & 0 & \dots \\ 0 & \ddots & 0 \\ \vdots & & \\ 0 & \dots & \theta(\mathbf{p}_n - \tilde{\mathbf{p}}_i) \end{bmatrix} \begin{bmatrix} x_0^2 & y_0^2 & z_0^2 & x_0y_0 & y_0z_0 & x_0z_0 & x_0 & y_0 & z_0 & 1 \\ x_1^2 & y_1^2 & z_1^2 & x_1y_1 & y_1z_1 & x_1z_1 & x_1 & y_1 & z_1 & 1 \\ \vdots & & & & & & & & & \\ x_n^2 & y_n^2 & z_n^2 & x_ny_n & y_nz_n & x_nz_n & x_n & y_n & z_n & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \\ a_9 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix}$$

We use a QR solver to solve for the unknown **a** vector. Then we can evaluate this polynomial with the a_i coefficients at position $\tilde{\mathbf{p}}_i$ to get the approximation v_i.

Remember that all of this was just for getting one v_i value. We need to optimize all the values. We want to minimize the error

$$err = \sum_i \|v_i - \tilde{f_i}\|^2$$

i.e., we want the approximation the match the original values as well as possible. This is a non-linear least squares problem, which can be solved with the Levenberg-Marquardt algorithm. However, the LM algorithm requires the jacobian of the error function (the derivative of the error with respect to each control point's position and value). This would be complicated: we would need to take the derivative of a matrix inverse. But automatic differentiation does it automatically for us.