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 \( \tilde{p}_i \) 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 \( p_j \) 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

\[
\sum_j ||v_i - f_i||^2 \ast \theta(p_j - \tilde{p}_i)
\]

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 \( \tilde{p}_i \) 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

\[
a_0 x^2 + a_1 y^2 + a_2 z^2 + a_3 xy + a_4 yz + a_5 xz + a_6 x + a_7 y + a_8 z + a_9
\]

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

\[
\begin{bmatrix}
\theta(p_0 - \tilde{p}_i) & 0 & \ldots & 0 \\
0 & \ddots & 0 \\
\vdots & & \ddots & 0 \\
0 & \ldots & \theta(p_n - \tilde{p}_i)
\end{bmatrix}
\begin{bmatrix}
x_0^2 & y_0^2 & \ldots & x_0 y_0 & y_0 z_0 & x_0 z_0 & x_0 & y_0 & z_0 \\
x_1^2 & y_1^2 & \ldots & x_1 y_1 & y_1 z_1 & x_1 z_1 & x_1 & y_1 & z_1 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
x_n^2 & y_n^2 & \ldots & x_n y_n & y_n z_n & x_n z_n & x_n & y_n & z_n
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
a_4 \\
a_5 \\
a_6 \\
a_7 \\
a_8 \\
a_9
\end{bmatrix}
= \begin{bmatrix}
f_0 \\
f_1 \\
f_2 \\
f_3 \\
f_4 \\
f_5 \\
f_6 \\
f_7 \\
f_8 \\
f_9
\end{bmatrix}
\]

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 \( \tilde{p}_i \) 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.