Pseudocode for Using Supplemental C Code

(from "Practical Parameterization of Rotations Using the Exponential Map" by F. Sebastian Grassia)

Postscript and Acrobat versions of this document, along with sample C source code for computing the rotation matrix $\mathbf{R}$, its partial derivatives $\partial\mathbf{R}/\partial v$, and $\dot{v}(v,\omega)$ for the two and three DOF versions of exponential map rotations can be found at http://www.acm.org/jgt/papers/Grassia98. The code can be used to start computing with the exponential map immediately, but as it stands, it is not very efficient. We have purposely omitted the caching necessary to make the implementation efficient, because the form of cache storage will depend on the modularization strategy.

In particular, the values `theta, cosp, sinp`, and the quaternion `q` should be cached and stored with `v` when the rotation matrix is calculated from `v`, to be used later when calculating derivatives. Also, as noted in the code, when computing the derivatives of the two DOF rotation, the intermediate derivatives of `q` with respect to a three DOF exponential map rotation should be computed only once and cached.

The code and functionality pertaining to the exponential map is typically a very small (but important!) part of much larger systems. It is beyond the scope of this paper to provide any kind of meaningful working system for example purposes. But to help give a feel for where computing $\partial\mathbf{R}/\partial\mathbf{v}$ fits into the overall scheme of things, we provide pseudocode for computing the Jacobian contribution of a node in a transformation hierarchy, with respect to all of the end effectors below it in the hierarchy. `Node_Jacobian_Contribution` makes the following assumptions:

- The function that maps an end effector *ee* from its local coordinate system (*i.e.* in a finger's frame of reference) to world coordinates is of the form

$$\mathrm{F}(ee) = \mathbf{T}_0\mathbf{R}_0\mathbf{T}_1\mathbf{R}_1\cdots\mathbf{T}_n\mathbf{R}_n ee$$

where *ee* is a column 3-vector, $\mathbf{T}_0$ is the root translation of the entire hierarchy (in the form of a 4x4 matrix), and $\mathbf{T}_i\mathbf{R}_i$ represent the i^{th} joint between the root and the end effector - $\mathbf{T}_i$ is the constant translation that positions the base of a limb in its parent's frame, and $\mathbf{R}_i$ is the joint's current rotation, a function of exponential map parameters.

- `Node_Jacobian_Contribution` only works for a hypothetical joint type that has a fixed center of rotation and a free, unconstrained three DOF rotation. Thus it will not work for the root of the hierarchy, since the root also has translational degrees of freedom.
- The type `Jacobian` is essentially an *m* by *n* matrix, where *m* is (3 * number of end effectors), and *n* is the number of DOFs in the system. Upon exit of `Node_Jacobian_Contribution`, *jac* will have the entries pertaining to node *i*'s degrees of freedom filled in.
- The type `TransformStack` is an object-oriented stack supporting the same types of operations as the OpenGL stack used to traverse hierarchies. Upon entry to `Node_Jacobian_Contribution`, *stack* should contain the combined transformation of all nodes above *i* in the hierarchy. It will be restored to this state upon exit.
- *nodeTrans* is the structural translation for the node ($\mathbf{T}_i$) as a 3-vector.
- *nodeOrient* is the vector representing the current state of the joint, *i.e.* its DOFs. It has parameter class **inout** because the act of computing its derivatives may cause it to be dynamically reparameterized. Therefore its value upon exit may differ from its value on entry, and this must be propagated into the global state vector (if your system gathers all DOFs into a global state vector).
- *startDOF* is the column index in *jac* of node *i*'s first degree-of-freedom; it is generally the same as *nodeOrient*'s position in the global state vector.
- *effectors* is a list of all of the end effectors below node *i* in the hierarchy. Each element has a member *jacIndex* that gives the first of three consecutive rows in the jacobian that correspond to the end effector. Each element has a member *coords* that is a 3-vector containing the *i-local* position of the end effector. "i-local" means the original end effector *ee* transformed by the combined transformation $\mathbf{T}_{i+1}\mathbf{R}_{i+1}\cdots\mathbf{T}_n\mathbf{R}_n$, *i.e.* the transformation from the originating frame of the end effector up to (but not including) node *i* in the hierarchy.

```
void Node_Jacobian_Contribution(inout Jacobian jac, inout TransformStack stack,
                                in double nodeTrans[3], inout double nodeOrient[3],
                                in int startDOF, in EffectorList effectors)
{
    int              i,j;
    EffectorList     currEff;
    double           dRdv[4][4];
    double           column[3];

    stack.Push();
    stack.Translate(nodeTrans);  /* put the constant translation on the stack */

    for j = 0 to 2 do {
      stack.Push();
      Partial_R_Partial_EM3(nodeOrient, j, dRdv);   /* provided function */
      stack.Concat_Matrix(dRdv);

      /* In the terminology used above, the partial derivative of the position of
       * the end effector with respect to the j'th rotational parameter of node
       * i is:
       *    (d F)/(d nodeOrient_j) =
       *      T_0 R_0 … T_i (d R_i)/(d nodeOrient_j)… T_n R_n  ee =
       *      stack.Current * i-local(ee)
       *
       *  which is what we do below */
      for currEff = each element of effectors do {
          stack.Vector_Transform(currEff.coords, column);
          for i = 0 to 2 do
            jac[currEff.jacIndex+i][startDOF+j] = column[i];
      }

      stack.Pop();
    }

    stack.Pop();
}
```