Finding the Contact Point

March 13, 2024

[1]:
```
# Imports
import numpy as np
import sympy as sp
```

1 Introduction of the Problem

Consider a patch and node that moves with time. Our goal is to determine the contact point in the reference space of the patch/element. The mapping of a reference point (ξ, η, ζ) to the global space is given by

$$\vec{s} = \sum_{p=0}^{n-1} \phi_p(\xi, \eta, \zeta)\vec{s}_p$$

where $\vec{s}$ is the position in the global space and $\vec{s}_p$ is a basis vector of the patch in the global space. If we consider the contact point to be of interest ($\vec{s}_c$) and the fact that its position moves with time as well as the basis vectors, we can write

$$\vec{s}_c + \dot{\vec{s}}_c \Delta t = \sum_{p=0}^{n-1} \phi_p(\xi, \eta, \zeta)(\vec{s}_p + \dot{\vec{s}}_p \Delta t)$$

With $\vec{s}_c = \langle x_s, y_s, z_s \rangle$ (the changing positions of the slave node) and $\vec{s}_p = \langle x_p, y_p, z_p \rangle$, we can write the above equation as

$$\begin{bmatrix} x_s \\ y_s \\ z_s \end{bmatrix} + \Delta t \begin{bmatrix} \dot{x}_s \\ \dot{y}_s \\ \dot{z}_s \end{bmatrix} = \sum_{p=0}^{n-1} \phi_p(\xi, \eta, \zeta) \left(\begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} + \Delta t \begin{bmatrix} \dot{x}_p \\ \dot{y}_p \\ \dot{z}_p \end{bmatrix} \right)$$

The above results in a system of three equations and three unknowns - Δt and any two of ξ, η, and ζ. The one of ξ, η, and ζ that is known is either 1 or -1. The newton-raphson scheme presented here will assume that ζ is known, but the numerical method will need to take into account the other two cases by adjusting the order of the nodal ξ, η, and ζ array prior to the analysis. It's as simple as moving the known value to the end of the array.

After the contact point is found as well as the Δt, the node is considered to be in contact with the patch if the following conditions are met:

1. All reference coordinates ξ, η, and ζ are between -1 and 1.
2. The solution for Δt is between 0 and dt (the time step of the explicit analysis).

2 Sympy Solution

[2]:
```
# Define the variables
p, n = sp.symbols('p n')
xi, eta, zeta = sp.symbols('xi eta zeta')
xs, ys, zs = sp.symbols('x_s y_s z_s')
xs_dot, ys_dot, zs_dot = sp.symbols(r'\dot{x}_s \dot{y}_s \dot{z}_s')
xp, yp, zp, xp_dot, yp_dot, zp_dot = [sp.Function(v) for v in ['x', 'y', 'z',␣
 ↪r'\dot{x}', r'\dot{y}', r'\dot{z}']]
del_t = sp.Symbol('\Delta t')
xi_p, eta_p, zeta_p = sp.symbols('xi_p eta_p zeta_p')
phi_p = sp.Function(r'\phi_p')(xi, eta, zeta, p)
```

```
# Define the matrix equation
lhs = sp.Matrix([xs, ys, zs]) + del_t*sp.Matrix([xs_dot, ys_dot, zs_dot])
rhs = sp.Sum(phi_p*(sp.Matrix([xp(p), yp(p), zp(p)]) + del_t*sp.
 ↪Matrix([xp_dot(p), yp_dot(p), zp_dot(p)])), (p, 0, n-1))
eq1 = sp.Eq(lhs, rhs, evaluate=False)
eq1
```

[2]:
$$\begin{bmatrix} \Delta t\dot{x}_s + x_s \\ \Delta t\dot{y}_s + y_s \\ \Delta t\dot{z}_s + z_s \end{bmatrix} = \sum_{p=0}^{n-1} \begin{bmatrix} (\Delta t\dot{x}(p) + x(p))\,\phi_p(\xi,\eta,\zeta,p) \\ (\Delta t\dot{y}(p) + y(p))\,\phi_p(\xi,\eta,\zeta,p) \\ (\Delta t\dot{z}(p) + z(p))\,\phi_p(\xi,\eta,\zeta,p) \end{bmatrix}$$

The $x(p)$, $y(p)$, $z(p)$, and so on should be interpreted as x_p, y_p, z_p, and so on. They are not actually functions of p, but this was implemented so that the `sympy` sum operation won't interpret the variable as a constant.

In the vector form, the Newton-Raphson scheme is

$$\begin{bmatrix} \xi_{i+1} \\ \eta_{i+1} \\ \Delta t_{i+1} \end{bmatrix} = \begin{bmatrix} \xi_i \\ \eta_i \\ \Delta t_i \end{bmatrix} - \mathbf{J}^{-1}\mathbf{F}$$

[3]:
```
# Constructing the vector function F
F = eq1.rhs.doit() - eq1.lhs
F
```

[3]:
$$\begin{bmatrix} -\Delta t\dot{x}_s - x_s + \sum_{p=0}^{n-1}\left(\Delta t\dot{x}(p)\phi_p(\xi,\eta,\zeta,p) + \phi_p(\xi,\eta,\zeta,p)x(p)\right) \\ -\Delta t\dot{y}_s - y_s + \sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\phi_p(\xi,\eta,\zeta,p) + \phi_p(\xi,\eta,\zeta,p)y(p)\right) \\ -\Delta t\dot{z}_s - z_s + \sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\phi_p(\xi,\eta,\zeta,p) + \phi_p(\xi,\eta,\zeta,p)z(p)\right) \end{bmatrix}$$

[4]:
```
# Constructing the Jacobian matrix J
J = F.jacobian([xi, eta, del_t])
J
```

[4]:
$$\begin{bmatrix} \sum_{p=0}^{n-1}\left(\Delta t\dot{x}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p) + x(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right) & \sum_{p=0}^{n-1}\left(\Delta t\dot{x}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p) + x(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right) & -\dot{x}_s \\ \sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p) + y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right) & \sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p) + y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right) & -\dot{y}_s \\ \sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p) + z(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right) & \sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p) + z(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right) & -\dot{z}_s \end{bmatrix}$$

It's a bit hard to see in the PDF, but here's a better look:

$$\begin{bmatrix} \sum_{p=0}^{n-1}\left(\Delta t\dot{x}_p\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta)+x_p\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta)\right) & \sum_{p=0}^{n-1}\left(\Delta t\dot{x}_p\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta)+x_p\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta)\right) & -\dot{x}_s+\sum_{p=0}^{n-1}\dot{x}_p\phi_p(\xi,\eta,\zeta) \\ \sum_{p=0}^{n-1}\left(\Delta t\dot{y}_p\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta)+y_p\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta)\right) & \sum_{p=0}^{n-1}\left(\Delta t\dot{y}_p\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta)+y_p\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta)\right) & -\dot{y}_s+\sum_{p=0}^{n-1}\dot{y}_p\phi_p(\xi,\eta,\zeta) \\ \sum_{p=0}^{n-1}\left(\Delta t\dot{z}_p\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta)+z_p\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta)\right) & \sum_{p=0}^{n-1}\left(\Delta t\dot{z}_p\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta)+z_p\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta)\right) & -\dot{z}_s+\sum_{p=0}^{n-1}\dot{z}_p\phi_p(\xi,\eta,\zeta) \end{bmatrix}$$

[5]:
```
# Constructing the inverse of the Jacobian matrix
J_inv = J.inv()
J_inv  # Takes about two minutes to complete
```

[5]:

$$\begin{bmatrix} \frac{\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)-\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)}{\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)-\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)} \\ \frac{-\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)+\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)}{\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)-\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)} \\ \frac{-\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)+\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)}{\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)-\dot{x}_s\left(\sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta,p)\right)\right)\sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)} \end{bmatrix}$$

Further observation of the above inverse matrix shows that there are more common terms that can be re-used in other elements of the matrix.

[6]:
```
# Cleaning up the inverse of the Jacobian matrix
# We can loop through each term of each numerator and define a new symbol for␣
 ↪each summation entity
J_inv_cleaned = J_inv
r, c = J_inv_cleaned.shape
sym_counter = 0
a = []
for i in range(r):
    for j in range(c):
        num, _ = sp.fraction(J_inv_cleaned[i, j])  # Get the numerator
        for term in num.args:  # Loop through each term of the numerator
            for entity in term.args:  # For each term, find the summation symbol
                if isinstance(entity, sp.Sum):
                    new_sym = sp.Symbol(f'a_{sym_counter}')
                    J_inv_cleaned = J_inv_cleaned.subs(entity, new_sym)
                    sym_counter += 1
                    a.append(entity)
                    display(sp.Eq(new_sym, entity))

# Cleaning up the denominator
J_inv_cleaned = J_inv_cleaned.simplify()
_, d = sp.fraction(J_inv_cleaned[0, 0])
J_inv_cleaned = J_inv_cleaned.subs(d, sp.Symbol('d'))
display(sp.Eq(sp.Symbol('d'), d))
display(J_inv_cleaned)
```

$$a_0 = \sum_{p=0}^{n-1}\left(\Delta t\dot{z}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+z(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)$$

$$a_1 = \sum_{p=0}^{n-1}\dot{z}(p)\phi_p(\xi,\eta,\zeta,p)$$

$$a_2 = \sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)$$

$$a_3 = \sum_{p=0}^{n-1}\left(\Delta t\dot{y}(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)+y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta,p)\right)$$

$$a_4 = \sum_{p=0}^{n-1} \dot{y}(p)\phi_p(\xi,\eta,\zeta,p)$$

$$a_5 = \sum_{p=0}^{n-1} \left(\Delta t \dot{z}(p) \frac{\partial}{\partial \eta}\phi_p(\xi,\eta,\zeta,p) + z(p)\frac{\partial}{\partial \eta}\phi_p(\xi,\eta,\zeta,p) \right)$$

$$a_6 = \sum_{p=0}^{n-1} \left(\Delta t \dot{x}(p) \frac{\partial}{\partial \eta}\phi_p(\xi,\eta,\zeta,p) + x(p)\frac{\partial}{\partial \eta}\phi_p(\xi,\eta,\zeta,p) \right)$$

$$a_7 = \sum_{p=0}^{n-1} \dot{x}(p)\phi_p(\xi,\eta,\zeta,p)$$

$$a_8 = \sum_{p=0}^{n-1} \left(\Delta t \dot{x}(p) \frac{\partial}{\partial \eta}\phi_p(\xi,\eta,\zeta,p) + x(p)\frac{\partial}{\partial \eta}\phi_p(\xi,\eta,\zeta,p) \right)$$

$$a_9 = \sum_{p=0}^{n-1} \left(\Delta t \dot{y}(p) \frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) + y(p)\frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) \right)$$

$$a_{10} = \sum_{p=0}^{n-1} \left(\Delta t \dot{z}(p) \frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) + z(p)\frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) \right)$$

$$a_{11} = \sum_{p=0}^{n-1} \left(\Delta t \dot{z}(p) \frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) + z(p)\frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) \right)$$

$$a_{12} = \sum_{p=0}^{n-1} \left(\Delta t \dot{y}(p) \frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) + y(p)\frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) \right)$$

$$a_{13} = \sum_{p=0}^{n-1} \left(\Delta t \dot{x}(p) \frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) + x(p)\frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) \right)$$

$$a_{14} = \sum_{p=0}^{n-1} \left(\Delta t \dot{x}(p) \frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) + x(p)\frac{\partial}{\partial \xi}\phi_p(\xi,\eta,\zeta,p) \right)$$

$d = \dot{x}_s a_0 a_9 - \dot{x}_s a_{10} a_2 - \dot{y}_s a_0 a_{13} + \dot{y}_s a_{10} a_6 + \dot{z}_s a_{13} a_2 - \dot{z}_s a_6 a_9 + a_0 a_{13} a_4 - a_0 a_7 a_9 - a_1 a_{13} a_2 + a_1 a_6 a_9 + a_{10} a_2 a_7 - a_{10} a_4 a_6$

$$\begin{bmatrix} \frac{-\dot{y}_s a_0 + \dot{z}_s a_2 + a_0 a_4 - a_1 a_2}{d} & \frac{\dot{x}_s a_0 - \dot{z}_s a_6 - a_0 a_7 + a_1 a_6}{d} & \frac{-\dot{x}_s a_2 + \dot{y}_s a_6 + a_2 a_7 - a_4 a_6}{d} \\ \frac{\dot{y}_s a_{10} - \dot{z}_s a_9 + a_1 a_9 - a_{10} a_4}{d} & \frac{-\dot{x}_s a_{10} + \dot{z}_s a_{13} - a_1 a_{13} + a_{10} a_7}{d} & \frac{\dot{x}_s a_9 - \dot{y}_s a_{13} + a_{13} a_4 - a_7 a_9}{d} \\ \frac{-a_0 a_9 + a_{10} a_2}{d} & \frac{a_0 a_{13} - a_{10} a_6}{d} & \frac{-a_{13} a_2 + a_6 a_9}{d} \end{bmatrix}$$

The above cleaned inverse Jacobian matrix can be used for the case where there is no matrix inverse operation in the code base. Otherwise, just compute the Jacobian numerically, then take the inverse of the Jacobian numerically.

2.1 Linear Hex Element Shape Functions

For a linear hex element, the shape function is

$$\phi_p(\xi,\eta,\zeta) = \frac{1}{8}(1+\xi\xi_p)(1+\eta\eta_p)(1+\zeta\zeta_p)$$

The first order partial derivatives also need to be found.

[7]:
```
# Define the shape function
phi_p_sym = sp.Function(r'\phi_p')(xi, eta, zeta)
phi_p = (sp.Rational(1, 8)*(1 + xi*xi_p)*(1 + eta*eta_p)*(1 + zeta*zeta_p)).
 ↪simplify()
sp.Eq(phi_p_sym, phi_p)
```

[7]:
$$\phi_p(\xi,\eta,\zeta) = \frac{\left(\eta\eta_p+1\right)\left(\xi\xi_p+1\right)\left(\zeta\zeta_p+1\right)}{8}$$

[8]:
```
# Partial derivatives
display(sp.Eq(phi_p_sym.diff(xi), phi_p.diff(xi).simplify()))
display(sp.Eq(phi_p_sym.diff(eta), phi_p.diff(eta).simplify()))
```

$$\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,\zeta) = \frac{\xi_p\left(\eta\eta_p+1\right)\left(\zeta\zeta_p+1\right)}{8}$$

$$\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,\zeta) = \frac{\eta_p\left(\xi\xi_p+1\right)\left(\zeta\zeta_p+1\right)}{8}$$

3 Numerical Solution

[9]:
```
# Define a Node datastructure
class Node:
    def __init__(self, pos, vel, ref_pos):
        """
        :param pos: Physical position array
        :param vel: Physical velocity array
        :param ref_pos: Reference position array
        """
        self.pos = pos
        self.vel = vel
        self.ref_pos = ref_pos
        self.x, self.y, self.z = pos
        self.xi, self.eta, self.zeta = ref_pos

# Define shape functions
def phi_p_lamb(xi_, eta_, zeta_, xi_p_, eta_p_, zeta_p_):
    return (1/8)*(1 + xi_*xi_p_)*(1 + eta_*eta_p_)*(1 + zeta_*zeta_p_)

def diff_xi_phi_p(eta_, zeta_, xi_p_, eta_p_, zeta_p_):
    return (1/8)*xi_p_*(1 + eta_*eta_p_)*(1 + zeta_*zeta_p_)

def diff_eta_phi_p(xi_, zeta_, xi_p_, eta_p_, zeta_p_):
    return (1/8)*eta_p_*(1 + xi_*xi_p_)*(1 + zeta_*zeta_p_)
```

```
# Define the vector function F
def F_lamb(x, sc, sc_dot, sp_vec, sp_dot, xi_p_, eta_p_, zeta_p_):
    """
    Compute the F array

    :param x: np.array; The array of unknowns that is being solved for. If zeta␣
 ↪is known, then x = [xi, eta, t_del].
    :param sc: np.array; The position of the contact point in the global space;␣
 ↪sc = [x_s, y_s, z_s].
    :param sc_dot: np.array; The velocity of the contact point in the global␣
 ↪space; sc_dot = [x_s_dot, y_s_dot, z_s_dot].
    :param sp_vec: np.array; The position of the nodes of the patch in the␣
 ↪global space; sp_vec = [[x_0, y_0, z_0], [x_1, y_1, z_1], ...].
    :param sp_dot: np.array; The velocity of the nodes of the patch in the␣
 ↪global space; sp_dot = [[x0_dot, y0_dot, z0_dot], [x1_dot, y1_dot, z1_dot], .
 ↪..].
    :param xi_p_: np.array; The xi coordinates of the nodes of the patch in the␣
 ↪reference space. xi_p_ = [xi_0, xi_1, ...].
    :param eta_p_: np.array; The eta coordinates of the nodes of the patch in␣
 ↪the reference space. eta_p_ = [eta_0, eta_1, ...].
    :param zeta_p_: np.array; The zeta coordinates of the nodes of the patch in␣
 ↪the reference space. zeta_p_ = [zeta_0, zeta_1, ...]. These should all be␣
 ↪the same value of 1 or -1.
    :return: np.array; The F array.
    """
    xi_, eta_, t_del = x
    # zeta_p_ is repeated twice here because zeta=zeta_p. This is the known␣
 ↪reference surface.
    phi_p_loc = phi_p_lamb(xi_, eta_, zeta_p_, xi_p_, eta_p_, zeta_p_)
    rhs_ = np.sum(phi_p_loc[:, None]*(sp_vec + t_del * sp_dot), axis=0)
    lhs_ = sc + t_del * sc_dot
    return rhs_ - lhs_

# Define the Jacobian matrix
def jac(x, sc_dot, sp_vec, sp_dot, xi_p_, eta_p_, zeta_p_):
    """
    Compute the Jacobian matrix

    :param x: np.array; The array of unknowns that is being solved for. If zeta␣
 ↪is known, then x = [xi, eta, t_del].
    :param sc_dot: np.array; The velocity of the contact point in the global␣
 ↪space; sc_dot = [x_s_dot, y_s_dot, z_s_dot].
    :param sp_vec: np.array; The position of the nodes of the patch in the␣
 ↪global space; sp_vec = [[x_0, y_0, z_0], [x_1, y_1, z_1], ...].
```

```
    :param sp_dot: np.array; The velocity of the nodes of the patch in the␣
↪global space; sp_dot = [[x0_dot, y0_dot, z0_dot], [x1_dot, y1_dot, z1_dot], .
↪..].
    :param xi_p_: np.array; The xi coordinates of the nodes of the patch in the␣
↪reference space. xi_p_ = [xi_0, xi_1, ...].
    :param eta_p_: np.array; The eta coordinates of the nodes of the patch in␣
↪the reference space. eta_p_ = [eta_0, eta_1, ...].
    :param zeta_p_: np.array; The zeta coordinates of the nodes of the patch in␣
↪the reference space. zeta_p_ = [zeta_0, zeta_1, ...]. These should all be␣
↪the same value of 1 or -1.
    :return: np.array; The Jacobian matrix.
    """
    xi_, eta_, t_del = x
    # zeta_p_ is repeated twice here because zeta=zeta_p. This is the known␣
↪reference surface.
    phi_p_loc = phi_p_lamb(xi_, eta_, zeta_p_, xi_p_, eta_p_, zeta_p_)
    d_phi_p_d_xi = diff_xi_phi_p(eta_, zeta_p_, xi_p_, eta_p_, zeta_p_)
    d_phi_p_d_eta = diff_eta_phi_p(xi_, zeta_p_, xi_p_, eta_p_, zeta_p_)
    xp_d, yp_d, zp_d = sp_dot[:, 0], sp_dot[:, 1], sp_dot[:, 2]
    xp_, yp_, zp_ = sp_vec[:, 0], sp_vec[:, 1], sp_vec[:, 2]
    xs_d, ys_d, zs_d = sc_dot
    jac_ = np.array([
        [sum(d_phi_p_d_xi*(xp_ + t_del*xp_d)), sum(d_phi_p_d_eta*(xp_ +␣
↪t_del*xp_d)), sum(phi_p_loc*xp_d) - xs_d],
        [sum(d_phi_p_d_xi*(yp_ + t_del*yp_d)), sum(d_phi_p_d_eta*(yp_ +␣
↪t_del*yp_d)), sum(phi_p_loc*yp_d) - ys_d],
        [sum(d_phi_p_d_xi*(zp_ + t_del*zp_d)), sum(d_phi_p_d_eta*(zp_ +␣
↪t_del*zp_d)), sum(phi_p_loc*zp_d) - zs_d]
    ])
    return jac_

# Define the Newton-Raphson scheme
def newton_raphson(guess, patch_nodes, slave_node, tol=1e-10, max_iter=100):
    """
    Compute the Newton-Raphson scheme

    :param guess: np.array; The initial guess for the unknowns. If zeta is␣
↪known, then guess = [xi, eta, t_del].
    :param patch_nodes: list[Node]; The list of Node objects that define a␣
↪patch.
    :param slave_node: Node; The Node object that defines the slave node.
    :param tol: float; The tolerance for the solution.
    :param max_iter: int; The maximum number of iterations.
    :return: np.array; The solution for [xi, eta, t_del].
    """
    sol = guess
```

```
    sc = slave_node.pos
    sc_dot = slave_node.vel
    sp_vec = np.array([node.pos for node in patch_nodes])
    sp_dot = np.array([node.vel for node in patch_nodes])
    xi_p_ = np.array([node.xi for node in patch_nodes])
    eta_p_ = np.array([node.eta for node in patch_nodes])
    zeta_p_ = np.array([node.zeta for node in patch_nodes])

    for k in range(max_iter):
        f = F_lamb(sol, sc, sc_dot, sp_vec, sp_dot, xi_p_, eta_p_, zeta_p_)
        jac_ = jac(sol, sc_dot, sp_vec, sp_dot, xi_p_, eta_p_, zeta_p_)
        sol = sol - np.linalg.inv(jac_) @ f

        if np.linalg.norm(f) < tol:
            break

    # noinspection PyUnboundLocalVariable
    return sol, k
```

4 Example Problem

[10]:
```
# Making an example problem
# Define points that make up the patch and the reference point associated with␣
 ↪it
points = np.array([
    [0.5, 0.5, 1, -1, -1, 1],
    [1, 0.5, 2, 1, -1, 1],
    [1, 1, 3, 1, 1, 1],
    [0.5, 1, 2, -1, 1, 1]
])

# Define velocity for points
vels = np.array([
    [0.12, 0.08, -0.05],
    [0.7*3, 0.75*3, -0.25*3],
    [-0.06, -0.03, -0.34],
    [-0.065, -0.035, -0.42]
])

slave_node = Node(np.array([0.75, 0.75, 1]), np.array([2, -0.1, 10.5]), [None,␣
 ↪None, None])
nodes = [Node(point[:3], vel, point[3:]) for point, vel in zip(points, vels)]
newton_raphson([0.5, -0.5, 0.8], nodes, slave_node)
```

[10]:
```
(array([ 0.34774981, -0.41631963,  0.08798188]), 4)
```

This is the correct result.