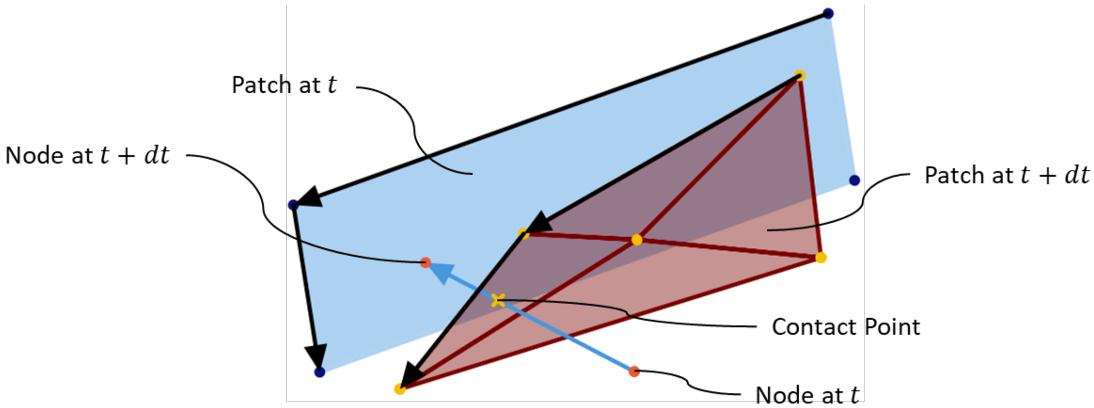
Contact Point to Reference

March 14, 2024

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

xc, yc = sp.symbols('x_c y_c')  # x and y values of contact point
xi, eta = sp.symbols(r'xi eta')  # reference coordinate variables
p, n = sp.symbols('p n')  # p is the index of an array and n is the total␣
↪number of points
xp, yp = sp.Function('x'), sp.Function('y')  # x and y components of a vector
xi_p, eta_p = sp.Function(r'\xi'), sp.Function(r'\eta')  # xi and eta points␣
↪that make up the boundary of the surface (either 1 or -1 always)
```



This demo is provided for constructing the set of non-linear functions to solve for the ξ and η reference coordinates of the contact point and construct a Newton-Raphson scheme.

For mapping a reference point (ξ, η) to the global/actual position point $(\vec{s})$, we use the following

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

where $\phi_p(\xi, \eta) = \frac{1}{4}(1 + \xi_p\xi)(1 + \eta_p\eta)$ is the basis/shape function for 2D corresponding to a known reference point $\vec{s}_p$. The position point has components

$$\vec{s}_p = \begin{bmatrix} x_p \\ y_p \end{bmatrix}$$

At some contact point (ξ_c, η_c), we can set up the following equation below to be analyzed.

[2]:
```python
phi_p = (sp.Rational(1, 4)*(1 + xi*xi_p(p))*(1 + eta*eta_p(p))).simplify()  # shape function in 2D space
phi_p_func = sp.Function(r'\phi_p')(xi, eta, p)
s_p = sp.Matrix([xp(p), yp(p)])  # Position vector of surface point
b = sp.Matrix([xc, yc])

eq = sp.Eq(b, sp.Sum(phi_p_func*s_p, (p, 0, n - 1)), evaluate=False)
display(eq)
eq1 = sp.Eq(b, sp.Sum(phi_p*s_p, (p, 0, n - 1)), evaluate=False)
display(eq1)
```

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \sum_{p=0}^{n-1} \begin{bmatrix} \phi_p(\xi,\eta,p)x(p) \\ \phi_p(\xi,\eta,p)y(p) \end{bmatrix}$$

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \sum_{p=0}^{n-1} \begin{bmatrix} \frac{(\eta\eta(p)+1)(\xi\xi(p)+1)x(p)}{4} \\ \frac{(\eta\eta(p)+1)(\xi\xi(p)+1)y(p)}{4} \end{bmatrix}$$

The $x(p)$, $\xi(p)$, and so on should be interpreted as x_p, ξ_p, and so on. This is how we can use `sympy` to symbolically construct the Newton-Raphson scheme in terms of reference points. For the Newton-Raphson scheme, we have

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

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

[3]:
$$\begin{bmatrix} -x_c + \sum_{p=0}^{n-1} \phi_p(\xi,\eta,p)x(p) \\ -y_c + \sum_{p=0}^{n-1} \phi_p(\xi,\eta,p)y(p) \end{bmatrix}$$

[4]:
```python
# Constructing the jacobian J
jac = F.jacobian([xi, eta])
jac
```

[4]:
$$\begin{bmatrix} \sum_{p=0}^{n-1} x(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,p) & \sum_{p=0}^{n-1} x(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,p) \\ \sum_{p=0}^{n-1} y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,p) & \sum_{p=0}^{n-1} y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,p) \end{bmatrix}$$

[5]:
```python
# Construct the inverse jacobian J^{-1}
jac_inv = jac.inv()
jac_inv
```

[5]:
$$\begin{bmatrix} \frac{\sum_{p=0}^{n-1} y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,p)}{-\left(\sum_{p=0}^{n-1} x(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,p)\right)\sum_{p=0}^{n-1} y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,p)+\left(\sum_{p=0}^{n-1} x(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,p)\right)\sum_{p=0}^{n-1} y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,p)} & \cdots \\ -\frac{\sum_{p=0}^{n-1} y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,p)}{-\left(\sum_{p=0}^{n-1} x(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,p)\right)\sum_{p=0}^{n-1} y(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,p)+\left(\sum_{p=0}^{n-1} x(p)\frac{\partial}{\partial\xi}\phi_p(\xi,\eta,p)\right)\sum_{p=0}^{n-1} y(p)\frac{\partial}{\partial\eta}\phi_p(\xi,\eta,p)} & \cdots \end{bmatrix}$$

[6]:
```python
# The denominator of each item is messy
# To clean it up, I'm replacing it with a variable "d"
a11 = jac_inv[0, 0]
_, d = sp.fraction(a11)
jac_inv.subs(d, sp.Symbol("d"))
```

[6]:
$$\left[\begin{matrix}\frac{\sum_{p=0}^{n-1} y{\left(p \right)} \frac{\partial}{\partial \eta} \phi_{p}{\left(\xi,\eta,p \right)}}{d} & - \frac{\sum_{p=0}^{n-1} x{\left(p \right)} \frac{\partial}{\partial \eta} \phi_{p}{\left(\xi,\eta,p \right)}}{d}\\- \frac{\sum_{p=0}^{n-1} y{\left(p \right)} \frac{\partial}{\partial \xi} \phi_{p}{\left(\xi,\eta,p \right)}}{d} & \frac{\sum_{p=0}^{n-1} x{\left(p \right)} \frac{\partial}{\partial \xi} \phi_{p}{\left(\xi,\eta,p \right)}}{d}\end{matrix}\right]$$

In summary, we have

$$F = \begin{bmatrix} -x_c + \sum_{p=0}^{n-1} \phi_p(\xi, \eta) x_p \\ -y_c + \sum_{p=0}^{n-1} \phi_p(\xi, \eta) y_p \end{bmatrix}$$

$$J = \begin{bmatrix} \sum_{p=0}^{n-1} x_p \frac{\partial}{\partial \xi} \phi_p(\xi, \eta) & \sum_{p=0}^{n-1} x_p \frac{\partial}{\partial \eta} \phi_p(\xi, \eta) \\ \sum_{p=0}^{n-1} y_p \frac{\partial}{\partial \xi} \phi_p(\xi, \eta) & \sum_{p=0}^{n-1} y_p \frac{\partial}{\partial \eta} \phi_p(\xi, \eta) \end{bmatrix}$$

$$J^{-1} = \begin{bmatrix} \frac{\sum_{p=0}^{n-1} y_p \frac{\partial}{\partial \eta} \phi_p(\xi, \eta)}{d} & -\frac{\sum_{p=0}^{n-1} x_p \frac{\partial}{\partial \eta} \phi_p(\xi, \eta)}{d} \\ -\frac{\sum_{p=0}^{n-1} y_p \frac{\partial}{\partial \xi} \phi_p(\xi, \eta)}{d} & \frac{\sum_{p=0}^{n-1} x_p \frac{\partial}{\partial \xi} \phi_p(\xi, \eta)}{d} \end{bmatrix}$$

where

$$d = -\left(\sum_{p=0}^{n-1} x_p \frac{\partial}{\partial \eta} \phi_p(\xi, \eta)\right) \sum_{p=0}^{n-1} y_p \frac{\partial}{\partial \xi} \phi_p(\xi, \eta) + \left(\sum_{p=0}^{n-1} x_p \frac{\partial}{\partial \xi} \phi_p(\xi, \eta)\right) \sum_{p=0}^{n-1} y_p \frac{\partial}{\partial \eta} \phi_p(\xi, \eta)$$

For a linear hex element, the shape function and its derivatives are shown below.

[7]:
```python
phi_p
```

[7]:
$$\frac{\left(\eta \eta{\left(p \right)} + 1\right) \left(\xi \xi{\left(p \right)} + 1\right)}{4}$$

[8]:
```python
# Derivative of phi_p with respect to xi
phi_p.diff(xi)
```

[8]:
$$\frac{\left(\eta \eta{\left(p \right)} + 1\right) \xi{\left(p \right)}}{4}$$

[9]:
```python
# Derivative of phi_p with respect to eta
phi_p.diff(eta)
```

[9]:
$$\frac{\left(\xi \xi{\left(p \right)} + 1\right) \eta{\left(p \right)}}{4}$$

The code below is the numerical implementation of this scheme.

[10]:
```python
class Node:
    def __init__(self, pos, ref_pos):
        """
```

```
        :param pos: Physical position array
        :param ref_pos: Reference position array
        """
        self.x, self.y, self.z = pos
        self.xi, self.eta, self.zeta = ref_pos

def phi_p_lamb(xi_, eta_, xi_p_, eta_p_):
    return 0.25*(1 + xi_*xi_p_)*(1 + eta_*eta_p_)

# The derivative functions could easily be just one, but for clarity they are␣
↪two separate functions.

def d_phi_p_lamb_xi(eta_, xi_p_, eta_p_):
    return 0.25*xi_p_*(1 + eta_*eta_p_)

def d_phi_p_lamb_eta(xi_, xi_p_, eta_p_):
    return 0.25*eta_p_*(1 + xi_*xi_p_)

def get_F(reference_point, physical_point, nodes):
    xi_, eta_ = reference_point
    xc_, yc_ = physical_point
    xp_ = np.array([p_.x for p_ in nodes])
    yp_ = np.array([p_.y for p_ in nodes])
    xi_p_ = np.array([p_.xi for p_ in nodes])
    eta_p_ = np.array([p_.eta for p_ in nodes])
    phi_p_ = phi_p_lamb(xi_, eta_, xi_p_, eta_p_)
    return np.array([
        sum(phi_p_*xp_) - xc_,
        sum(phi_p_*yp_) - yc_
    ])

def get_jac(reference_point, nodes):
    xi_, eta_ = reference_point
    xp_ = np.array([p_.x for p_ in nodes])
    yp_ = np.array([p_.y for p_ in nodes])
    xi_p_ = np.array([p_.xi for p_ in nodes])
    eta_p_ = np.array([p_.eta for p_ in nodes])
    d_phi_p_xi_ = d_phi_p_lamb_xi(eta_, xi_p_, eta_p_)
    d_phi_p_eta_ = d_phi_p_lamb_eta(xi_, xi_p_, eta_p_)
    return np.array([
        [sum(xp_*d_phi_p_xi_), sum(xp_*d_phi_p_eta_)],
        [sum(yp_*d_phi_p_xi_), sum(yp_*d_phi_p_eta_)]
    ])

def newton_raphson(reference_point, physical_point, nodes, tol=1e-8,␣
↪max_iter=100):
    xi_, eta_ = reference_point
```

```
    for i in range(max_iter):
        F_ = get_F([xi_, eta_], physical_point, nodes)
        xi_, eta_ = np.array([xi_, eta_]) - np.linalg.
↪inv(get_jac(reference_point, nodes)) @ F_
        if np.linalg.norm(F_) < tol:
            break
    # noinspection PyUnboundLocalVariable
    return np.array([xi_, eta_]), i
```

Consider a quadrilateral surface bound by the following points:

Label	ξ, η, ζ	x, y, z
0	$-1, -1, -1$	$0.51025339, 0.50683559, 0.99572776$
1	$1, -1, -1$	$1.17943427, 0.69225101, 1.93591633$
2	$1, 1, -1$	$0.99487331, 0.99743665, 2.97094874$
3	$-1, 1, -1$	$0.49444608, 0.99700943, 1.96411315$

The contact point is $(0.92088978, 0.74145551, 1.89717136)$. The analysis omits ζ because we already know that the contact point is on the exterior surface. For this case, $\zeta = -1$. **Note: The implemented procedure needs to use those reference points that are changing.** For example, if contact is on the reference plane $\eta = 1$, then the process needs to solve for ξ and ζ.

[11]:
```
patch_nodes = [
    Node([0.51025339, 0.50683559, 0.99572776], [-1, -1, -1]),
    Node([1.17943427, 0.69225101, 1.93591633], [1, -1, -1]),
    Node([0.99487331, 0.99743665, 2.97094874], [1, 1, -1]),
    Node([0.49444608, 0.99700943, 1.96411315], [-1, 1, -1])
]

newton_raphson([0.5, -0.5], [0.92088978, 0.74145551], patch_nodes)
```

[11]:
```
(array([ 0.34340497, -0.39835547]), 4)
```