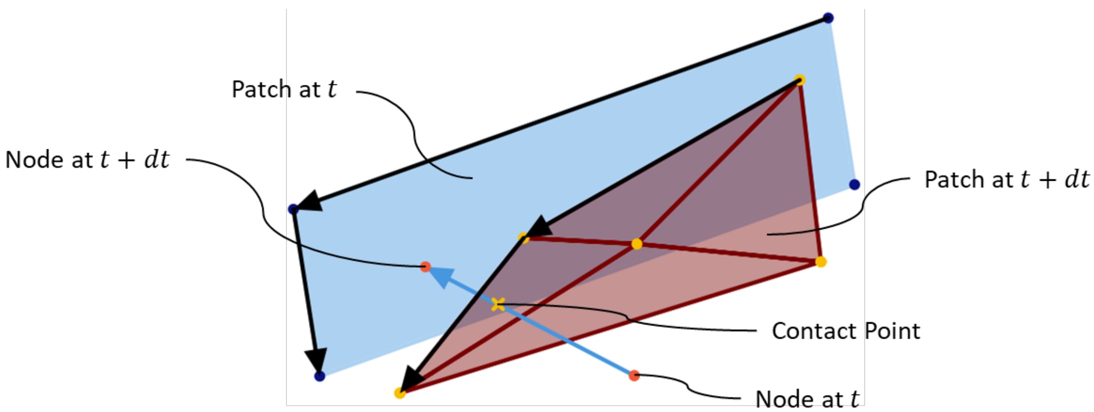
Contact Point to Reference

March 7, 2024

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

x0, x1, x2, x3 = sp.symbols('x0:4')  # x values of quadrilateral points
y0, y1, y2, y3 = sp.symbols('y0:4')  # y values of quadrilateral points
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
xi_c, eta_c = sp.symbols(r'xi_c eta_c')  # reference coordinates of contact␣
↪point
```



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.

[2]:
```
# With the order of the quadrilateral points starting from the bottom left␣
↪going counterclockwise, the basis functions are:
phi_p = [(sp.Rational(1, 4)*(1 + xi_p*xi)*(1 + eta_p*eta)).simplify() for xi_p,␣
↪eta_p in ((-1, -1), (1, -1), (1, 1), (-1, 1))]

# Construct the matrix A
A = np.array([
    [1, x0, y0],
    [1, x1, y1],
    [1, x2, y2],
    [1, x3, y3]
```

```
]).T

# Show the matrix equation
phi_p = sp.Matrix(phi_p)
A = sp.Matrix(A)
b = sp.Matrix([1, xc, yc])

eq = sp.Eq(b, sp.MatMul(A, phi_p))
display(eq)

eq = sp.Eq(eq.lhs, eq.rhs.doit())
display(eq)

# We are not interested in the first equation
eq = sp.Eq(sp.Matrix(eq.lhs[1:]), eq.rhs[1:, :])
eq
```

$$\begin{bmatrix}1\\x_c\\y_c\end{bmatrix} = \begin{bmatrix}1 & 1 & 1 & 1\\x_0 & x_1 & x_2 & x_3\\y_0 & y_1 & y_2 & y_3\end{bmatrix}\begin{bmatrix}\frac{(\eta-1)(\xi-1)}{4}\\-\frac{(\eta-1)(\xi+1)}{4}\\\frac{(\eta+1)(\xi+1)}{4}\\-\frac{(\eta+1)(\xi-1)}{4}\end{bmatrix}$$

$$\begin{bmatrix}1\\x_c\\y_c\end{bmatrix} = \begin{bmatrix}\frac{(\eta-1)(\xi-1)}{4} - \frac{(\eta-1)(\xi+1)}{4} - \frac{(\eta+1)(\xi-1)}{4} + \frac{(\eta+1)(\xi+1)}{4}\\\frac{x_0(\eta-1)(\xi-1)}{4} - \frac{x_1(\eta-1)(\xi+1)}{4} + \frac{x_2(\eta+1)(\xi+1)}{4} - \frac{x_3(\eta+1)(\xi-1)}{4}\\\frac{y_0(\eta-1)(\xi-1)}{4} - \frac{y_1(\eta-1)(\xi+1)}{4} + \frac{y_2(\eta+1)(\xi+1)}{4} - \frac{y_3(\eta+1)(\xi-1)}{4}\end{bmatrix}$$

[2]:

$$\begin{bmatrix}x_c\\y_c\end{bmatrix} = \begin{bmatrix}\frac{x_0(\eta-1)(\xi-1)}{4} - \frac{x_1(\eta-1)(\xi+1)}{4} + \frac{x_2(\eta+1)(\xi+1)}{4} - \frac{x_3(\eta+1)(\xi-1)}{4}\\\frac{y_0(\eta-1)(\xi-1)}{4} - \frac{y_1(\eta-1)(\xi+1)}{4} + \frac{y_2(\eta+1)(\xi+1)}{4} - \frac{y_3(\eta+1)(\xi-1)}{4}\end{bmatrix}$$

With two equations and two unknowns, the Newton-Raphson method can be implemented.

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

[3]:

```
# Construct vector function
F = eq.rhs - eq.lhs
F
```

[3]:

$$\begin{bmatrix}\frac{x_0(\eta-1)(\xi-1)}{4} - \frac{x_1(\eta-1)(\xi+1)}{4} + \frac{x_2(\eta+1)(\xi+1)}{4} - \frac{x_3(\eta+1)(\xi-1)}{4} - x_c\\\frac{y_0(\eta-1)(\xi-1)}{4} - \frac{y_1(\eta-1)(\xi+1)}{4} + \frac{y_2(\eta+1)(\xi+1)}{4} - \frac{y_3(\eta+1)(\xi-1)}{4} - y_c\end{bmatrix}$$

[4]:

```
# Find the jacobian
jac = F.jacobian([xi, eta])
jac
```

[4]:

$$\begin{bmatrix}\frac{x_0(\eta-1)}{4} - \frac{x_1(\eta-1)}{4} + \frac{x_2(\eta+1)}{4} - \frac{x_3(\eta+1)}{4} & \frac{x_0(\xi-1)}{4} - \frac{x_1(\xi+1)}{4} + \frac{x_2(\xi+1)}{4} - \frac{x_3(\xi-1)}{4}\\\frac{y_0(\eta-1)}{4} - \frac{y_1(\eta-1)}{4} + \frac{y_2(\eta+1)}{4} - \frac{y_3(\eta+1)}{4} & \frac{y_0(\xi-1)}{4} - \frac{y_1(\xi+1)}{4} + \frac{y_2(\xi+1)}{4} - \frac{y_3(\xi-1)}{4}\end{bmatrix}$$

[5]:
```python
# Find the inverse of the jacobian
jac_inv = jac.inv()
jac_inv
```

[5]:

$$\left[\begin{matrix}\frac{-2\xi y_0+2\xi y_1-2\xi y_2+2\xi y_3+2y_0+2y_1-2y_2-2y_3}{\eta x_0 y_1-\eta x_0 y_2-\eta x_1 y_0+\eta x_1 y_3+\eta x_2 y_0-\eta x_2 y_3-\eta x_3 y_1+\eta x_3 y_2+x_0\xi y_2-x_0\xi y_3-x_0 y_1+x_0 y_3-x_1\xi y_2+x_1\xi y_3+x_1 y_0-x_1 y_2-x_2\xi y_0+x_2\xi y_1+x_2 y_1-x_2 y_3-} \\ \frac{2\eta y_0-2\eta y_1+2\eta y_2-2\eta y_3-2y_0+2y_1+2y_2-2y_3}{\eta x_0 y_1-\eta x_0 y_2-\eta x_1 y_0+\eta x_1 y_3+\eta x_2 y_0-\eta x_2 y_3-\eta x_3 y_1+\eta x_3 y_2+x_0\xi y_2-x_0\xi y_3-x_0 y_1+x_0 y_3-x_1\xi y_2+x_1\xi y_3+x_1 y_0-x_1 y_2-x_2\xi y_0+x_2\xi y_1+x_2 y_1-x_2 y_3-}\end{matrix}\right.$$

[6]:
```python
# Jacobian constructor
# noinspection PyShadowingNames
def jac_inv(ref_point, *surface_points):
    x0, y0, x1, y1, x2, y2, x3, y3 = surface_points
    xi, eta = ref_point
    den = eta*x0*y1 - eta*x0*y2 - eta*x1*y0 + eta*x1*y3 + eta*x2*y0 - eta*x2*y3␣
 ↪- eta*x3*y1 + eta*x3*y2 + xi*x0*y2 - xi*x0*y3 - xi*x1*y2 + xi*x1*y3 -␣
 ↪xi*x2*y0 + xi*x2*y1 + xi*x3*y0 - xi*x3*y1 - x0*y1 + x0*y3 + x1*y0 - x1*y2 +␣
 ↪x2*y1 - x2*y3 - x3*y0 + x3*y2

    return np.array([
        [-2*xi*y0 + 2*xi*y1 - 2*xi*y2 + 2*xi*y3 + 2*y0 + 2*y1 - 2*y2 - 2*y3,␣
 ↪2*xi*x0 - 2*xi*x1 + 2*xi*x2 - 2*xi*x3 - 2*x0 - 2*x1 + 2*x2 + 2*x3],
        [2*eta*y0 - 2*eta*y1 + 2*eta*y2 - 2*eta*y3 - 2*y0 + 2*y1 + 2*y2 - 2*y3,␣
 ↪-2*eta*x0 + 2*eta*x1 - 2*eta*x2 + 2*eta*x3 + 2*x0 - 2*x1 - 2*x2 + 2*x3]
    ])/den

# noinspection PyShadowingNames
def F_vec(ref_point, contact_point, *surface_points):
    x0, y0, x1, y1, x2, y2, x3, y3 = surface_points
    xi, eta = ref_point
    xc, yc = contact_point
    return np.array([
        x0*(eta - 1)*(xi - 1)/4 - x1*(eta - 1)*(xi + 1)/4 + x2*(eta + 1)*(xi +␣
 ↪1)/4 - x3*(eta + 1)*(xi - 1)/4 - xc,
        y0*(eta - 1)*(xi - 1)/4 - y1*(eta - 1)*(xi + 1)/4 + y2*(eta + 1)*(xi +␣
 ↪1)/4 - y3*(eta + 1)*(xi - 1)/4 - yc
    ])

# Construct the Newton-Raphson solver function
# noinspection PyShadowingNames
def find_reference(guess, contact_point, surface_points, tol=1e-8):
    x0, y0, x1, y1, x2, y2, x3, y3 = surface_points.flatten()
    xc, yc = contact_point
    xi, eta = guess

    i = 0
    while not -tol <= np.linalg.norm(F_vec([xi, eta], contact_point, x0, y0,␣
 ↪x1, y1, x2, y2, x3, y3)) <= tol:
```

```
        xi, eta = np.array([xi, eta]) - np.matmul(jac_inv([xi, eta], x0, y0,␣
↪x1, y1, x2, y2, x3, y3), F_vec([xi, eta], [xc, yc], x0, y0, x1, y1, x2, y2,␣
↪x3, y3))
        i += 1
        if i == 100:
            return None
    return np.array([xi, eta])
```

The Sandia paper also claims that the following is a valid Newton-Raphson scheme:

$$\left\{ \begin{array}{c} \xi_{i+1} \\ \eta_{i+1} \end{array} \right\} = \left[\begin{array}{cc} \sum_{j=1}^{4} x_j \xi_j & \sum_{j=1}^{4} x_j \eta_j \\ \sum_{j=1}^{4} y_j \xi_j & \sum_{j=1}^{4} y_j \eta_j \end{array} \right] \left\{ \begin{array}{c} 4x - \sum_{j=1}^{4} \left(1 + \xi_j \eta_j \xi_i \eta_i\right) x_j \\ 4y - \sum_{j=1}^{4} \left(1 + \xi_j \eta_j \xi_i \eta_i\right) y_j \end{array} \right\}$$

Both methods will be tested in the following code cells.

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.

[7]:
```
# Set up surface points
points = np.array([
    [0.51025339, 0.50683559, 0.99572776],
    [1.17943427, 0.69225101, 1.93591633],
    [0.99487331, 0.99743665, 2.97094874],
    [0.49444608, 0.99700943, 1.96411315]
])

contact_point = np.array([0.92088978, 0.74145551, 1.89717136])  # Contact point␣
↪should be in quadrant 4 on the reference plane (+xi, -eta)
```

Before testing the `find_reference` function, the solution can be found using `sympy` for verification.

[8]:
```
x0_, y0_, x1_, y1_, x2_, y2_, x3_, y3_ = points[:, :2].flatten()
xc_, yc_ = contact_point[:2]
eq_sub = eq.subs([
    (x0, x0_),
    (y0, y0_),
    (x1, x1_),
```

```
    (y1, y1_),
    (x2, x2_),
    (y2, y2_),
    (x3, x3_),
    (y3, y3_),
    (xc, xc_),
    (yc, yc_)
])
eqs = [
    sp.Eq(eq_sub.lhs[0], eq_sub.rhs[0]),
    sp.Eq(eq_sub.lhs[1], eq_sub.rhs[1])
]
sp.nsolve(eqs, (xi, eta), [0.5, -0.5])
```

[8]:

$$\begin{bmatrix} 0.34340496965211 \\ -0.398355474595736 \end{bmatrix}$$

Here is the result using the `find_reference` function.

[9]:

```
find_reference([0.5, -0.5], contact_point[:2], points[:, :2])
```

[9]:

```
array([ 0.34340497, -0.39835547])
```

The same result here indicates that the solving method works.

For the Sandia scheme:

[10]:

```
# Reference point map
ref_map = np.array([
    [-1, -1, -1],
    [1, -1, -1],
    [1, 1, -1],
    [-1, 1, -1]
])

# noinspection PyShadowingNames
def sandia_calc(guess):
    x = points[:, 0]
    xi_ref = ref_map[:, 0]
    y = points[:, 1]
    eta_ref = ref_map[:, 1]

    xc, yc = contact_point[:2]
    xi, eta = guess

    J = np.array([
        [sum(x*xi_ref), sum(x*eta_ref)],
        [sum(y*xi_ref), sum(y*eta_ref)]
    ])
```

```
    for _ in range(30):
        p = np.array([
            4*xc - sum((1 + xi*eta*xi_ref*eta_ref)*x),
            4*yc - sum((1 + xi*eta*xi_ref*eta_ref)*y)
        ])
        xi, eta = np.matmul(J, p)

    return xi, eta

sandia_calc([0.34, -0.39])
```

[10]: (0.6258881729365896, -0.09832963091766227)

The Sandia scheme is not producing correct results.