5ELEN018W - Robotic Principles
Lecture 5: Inverse Kinematics

Dr Dimitris C. Dracopoulos

The problem of Forward kinematics

The problem of Forward kinematics

Kinematic chain: a series of rigid bodies (e.g. links of a robotic arm) connected together by joints.

The problem of Forward kinematics

Kinematic chain: a series of rigid bodies (e.g. links of a robotic arm) connected together by joints.
The joint angles of a kinematic chain determine the position and orientation of the end effector.

- A coordinate frame i relative to coordinate frame $i-1$ is denoted by ${}^{i-1}\boldsymbol{T}_i$:

The problem of Forward kinematics

Kinematic chain: a series of rigid bodies (e.g. links of a robotic arm) connected together by joints.
The joint angles of a kinematic chain determine the position and orientation of the end effector.

- A coordinate frame i relative to coordinate frame $i-1$ is denoted by ${}^{i-1}\boldsymbol{T}_i$:

$$\begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & r_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & r_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

The problem of Forward kinematics

Kinematic chain: a series of rigid bodies (e.g. links of a robotic arm) connected together by joints.
The joint angles of a kinematic chain determine the position and orientation of the end effector.

- A coordinate frame i relative to coordinate frame $i-1$ is denoted by ${}^{i-1}\boldsymbol{T}_i$:

$$\begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & r_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & r_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

where $\theta_i, \alpha_i, r_i, d_i$ are the DH parameters.

The Problem of Forward Kinematics (cont'd)

The problem of forward kinematics is expressed as the calculation of the transformation between a coordinate frame fixed in the *end-effector* and another coordinate frame fixed in the base.

The Problem of Forward Kinematics (cont'd)

The problem of forward kinematics is expressed as the calculation of the transformation between a coordinate frame fixed in the *end-effector* and another coordinate frame fixed in the base. For example, for 6-joint manipulator:

$$ {}^{0}\boldsymbol{T}_6 = {}^{0}\boldsymbol{T}_1 \cdot {}^{1}\boldsymbol{T}_2 \cdot {}^{2}\boldsymbol{T}_3 \cdot {}^{3}\boldsymbol{T}_4 \cdot {}^{4}\boldsymbol{T}_5 \cdot {}^{5}\boldsymbol{T}_6 \qquad (2) $$

The Problem of Inverse Kinematics

The Problem of Inverse Kinematics

Given a position and orientation of a robot's end-effector, calculate the angles θ of the joints.

The Problem of Inverse Kinematics

Given a position and orientation of a robot's end-effector, calculate the angles θ of the joints.

- Solving the kinematics equations of a manipulator robot is a nonlinear problem.

The Problem of Inverse Kinematics

Given a position and orientation of a robot's end-effector, calculate the angles θ of the joints.

- Solving the kinematics equations of a manipulator robot is a nonlinear problem.
- Given the homogeneous matrix of the end-effector with respect to the base frame, solve for all the joint angles $\theta_1, \theta_2, \ldots, \theta_n$.

The Problem of Inverse Kinematics

Given a position and orientation of a robot's end-effector, calculate the angles θ of the joints.

- Solving the kinematics equations of a manipulator robot is a nonlinear problem.
- Given the homogeneous matrix of the end-effector with respect to the base frame, solve for all the joint angles $\theta_1, \theta_2, \ldots, \theta_n$.

Challenging mathematical problem due to:

The Problem of Inverse Kinematics

Given a position and orientation of a robot's end-effector, calculate the angles θ of the joints.

- Solving the kinematics equations of a manipulator robot is a nonlinear problem.
- Given the homogeneous matrix of the end-effector with respect to the base frame, solve for all the joint angles $\theta_1, \theta_2, \ldots, \theta_n$.

Challenging mathematical problem due to:

1. nature of the nonlinear equations. Often no analytic solutions (closed form) can be calculated and numerical methods are required.

The Problem of Inverse Kinematics

Given a position and orientation of a robot's end-effector, calculate the angles θ of the joints.

- Solving the kinematics equations of a manipulator robot is a nonlinear problem.
- Given the homogeneous matrix of the end-effector with respect to the base frame, solve for all the joint angles $\theta_1, \theta_2, \ldots, \theta_n$.

Challenging mathematical problem due to:

1. nature of the nonlinear equations. Often no analytic solutions (closed form) can be calculated and numerical methods are required.
2. often, there are multiple solutions (i.e. multiple sets of joint angles) that can place the end effector at the desired position.

The Problem of Inverse Kinematics

Given a position and orientation of a robot's end-effector, calculate the angles θ of the joints.

- Solving the kinematics equations of a manipulator robot is a nonlinear problem.
- Given the homogeneous matrix of the end-effector with respect to the base frame, solve for all the joint angles $\theta_1, \theta_2, \ldots, \theta_n$.

Challenging mathematical problem due to:

1. nature of the nonlinear equations. Often no analytic solutions (closed form) can be calculated and numerical methods are required.
2. often, there are multiple solutions (i.e. multiple sets of joint angles) that can place the end effector at the desired position.

   $\longrightarrow$ The algorithm must choose the solution that results in the most natural and efficient motion of the robot.

The Problem of Inverse Kinematics

Given a position and orientation of a robot's end-effector, calculate the angles θ of the joints.

- Solving the kinematics equations of a manipulator robot is a nonlinear problem.
- Given the homogeneous matrix of the end-effector with respect to the base frame, solve for all the joint angles $\theta_1, \theta_2, \ldots, \theta_n$.

Challenging mathematical problem due to:

1. nature of the nonlinear equations. Often no analytic solutions (closed form) can be calculated and numerical methods are required.
2. often, there are multiple solutions (i.e. multiple sets of joint angles) that can place the end effector at the desired position.

   $\longrightarrow$ The algorithm must choose the solution that results in the most natural and efficient motion of the robot.
3. It is possible that no solutions exist

Applications of Inverse Kinematics

Applications of Inverse Kinematics

- Manufacturing and assembly

Applications of Inverse Kinematics

- Manufacturing and assembly
- Surgery

Applications of Inverse Kinematics

- Manufacturing and assembly
- Surgery
- Search and rescue

Methods for Solving Inverse Kinematics

Methods for Solving Inverse Kinematics

- Closed-form methods

Methods for Solving Inverse Kinematics

- Closed-form methods
- Iterative methods

Methods for Solving Inverse Kinematics

- Closed-form methods
- Iterative methods

Closed-form solutions are desirable because they are faster than numerical solutions and identify all possible solutions.

Methods for Solving Inverse Kinematics

- Closed-form methods
- Iterative methods

Closed-form solutions are desirable because they are faster than numerical solutions and identify all possible solutions.

- They are not general, but <u>robot dependent</u>.

Methods for Solving Inverse Kinematics

- Closed-form methods
- Iterative methods

Closed-form solutions are desirable because they are faster than numerical solutions and identify all possible solutions.

- They are not general, but robot dependent.
- To calculate, they take advantage of particular geometric features of specific robot mechanisms.

Methods for Solving Inverse Kinematics

- Closed-form methods
- Iterative methods

Closed-form solutions are desirable because they are faster than numerical solutions and identify all possible solutions.

- They are not general, but <u>robot dependent</u>.
- To calculate, they take advantage of particular geometric features of specific robot mechanisms.
- As the number of joints increases, this becomes increasingly difficult.

Methods for Solving Inverse Kinematics

- Closed-form methods
- Iterative methods

Closed-form solutions are desirable because they are faster than numerical solutions and identify all possible solutions.

- They are not general, but robot dependent.
- To calculate, they take advantage of particular geometric features of specific robot mechanisms.
- As the number of joints increases, this becomes increasingly difficult.
- For some serial-link robot manipulators, no analytical (closed form) solution exists!

Closed-form Methods

Closed-form Methods

- Algebraic methods

Closed-form Methods

- Algebraic methods
- Geometric methods

Algebraic Methods for Solving Inverse Kinematics

Algebraic Methods for Solving Inverse Kinematics

1. Identify the significant equations containing the joint variables.

Algebraic Methods for Solving Inverse Kinematics

1. Identify the significant equations containing the joint variables.
2. Manipulate them into a soluble form.

Algebraic Methods for Solving Inverse Kinematics

1. Identify the significant equations containing the joint variables.
2. Manipulate them into a soluble form.

- A common strategy is reduction to a equation in a single variable,

Algebraic Methods for Solving Inverse Kinematics

1. Identify the significant equations containing the joint variables.
2. Manipulate them into a soluble form.

- A common strategy is reduction to a equation in a single variable, e.g.

$$C_1 \cos \theta_i + C_2 \sin \theta_i + C_3 = 0$$

where C_1, C_2, C_3 are constants.

Algebraic Methods for Solving Inverse Kinematics

1. Identify the significant equations containing the joint variables.
2. Manipulate them into a soluble form.

- A common strategy is reduction to a equation in a single variable, e.g.

$$C_1 \cos\theta_i + C_2 \sin\theta_i + C_3 = 0$$

where C_1, C_2, C_3 are constants.

Solution:

$$\theta_i = 2\tan^{-1}\left(\frac{C_2 \pm \sqrt{C_2^2 - C_3^2 + C_1^2}}{C_1 - C_3}\right)$$

Algebraic Methods for Solving Inverse Kinematics (cont'd)

- Another useful strategy is the reduction to a pair of equations having the form:

Algebraic Methods for Solving Inverse Kinematics (cont'd)

- Another useful strategy is the reduction to a pair of equations having the form:

$$
\begin{aligned}
C_1 \cos\theta_i + C_2 \sin\theta_i + C_3 &= 0 \\
C_1 \sin\theta_i - C_2 \cos\theta_i + C_4 &= 0
\end{aligned}
$$

Algebraic Methods for Solving Inverse Kinematics (cont'd)

- Another useful strategy is the reduction to a pair of equations having the form:

$$
\begin{aligned}
C_1 \cos\theta_i + C_2 \sin\theta_i + C_3 &= 0 \\
C_1 \sin\theta_i - C_2 \cos\theta_i + C_4 &= 0
\end{aligned}
$$

only one solution:

$$\theta_i = \operatorname{atan2}(-C_1 C_4 - C_2 C_3, C_2 C_4 - C_1 C_3)$$

Geometric Methods for Solving Inverse Kinematics

Geometric Methods for Solving Inverse Kinematics

Such methods involve identifying points on the manipulator relative to which position and/or orientation can be expressed as a function of a reduced set of the joint variables using trigonometric relationships.

Geometric Methods for Solving Inverse Kinematics

Such methods involve identifying points on the manipulator relative to which position and/or orientation can be expressed as a function of a reduced set of the joint variables using trigonometric relationships.
$\longrightarrow$ often results to the decomposition of the spatial problem into separate planar problems.

Geometric Methods for Solving Inverse Kinematics

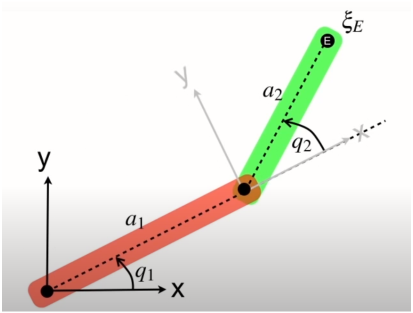
Such methods involve identifying points on the manipulator relative to which position and/or orientation can be expressed as a function of a reduced set of the joint variables using trigonometric relationships.
$\longrightarrow$ often results to the decomposition of the spatial problem into separate planar problems.

- Decomposition of the full problem into inverse position kinematics and inverse orientation kinematics.
- The solution is derived by rewriting equation (2) as:

$$^0\boldsymbol{T}_6 \cdot {}^6\boldsymbol{T}_5 \cdot {}^5\boldsymbol{T}_4 \cdot {}^4\boldsymbol{T}_3 = {}^0\boldsymbol{T}_1 \cdot {}^1\boldsymbol{T}_2 \cdot {}^2\boldsymbol{T}_3 \tag{3}$$

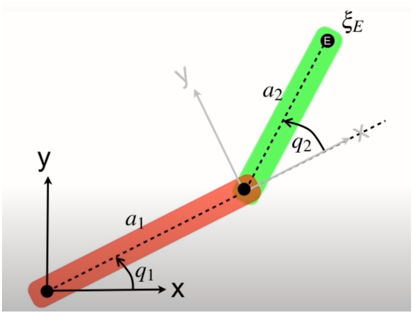
Calculating Analytical Solutions in the Python Robotics Toolbox

Consider a 2-joint Planar (2D) Robot



Calculating Analytical Solutions in the Python Robotics Toolbox

Consider a 2-joint Planar (2D) Robot



Given the position of the end-effector (x_E, y_E) calculate the required joint angles to achieve this position.

Calculating Analytical Solutions in Python (con'd)

```
# create symbols for lengths of the 2 links
>>> a1 = Symbol('a1')
>>> a2 = Symbol('a2')
```

Calculating Analytical Solutions in Python (con'd)

```
# create symbols for lengths of the 2 links
>>> a1 = Symbol('a1')
>>> a2 = Symbol('a2')

# symbols for joints angles
>>> q1, q2 = symbols("q1:3")
```

Calculating Analytical Solutions in Python (con'd)

```
# create symbols for lengths of the 2 links
>>> a1 = Symbol('a1')
>>> a2 = Symbol('a2')

# symbols for joints angles
>>> q1, q2 = symbols("q1:3")

# transformation to calculate the position of the end-effector
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)
```

Calculating Analytical Solutions in Python (con'd)

```
# create symbols for lengths of the 2 links
>>> a1 = Symbol('a1')
>>> a2 = Symbol('a2')

# symbols for joints angles
>>> q1, q2 = symbols("q1:3")

# transformation to calculate the position of the end-effector
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# calculate forward kinematics matrix for end-effector
>>> TE = e.fkine([q1, q2])

# translation part of matrix gives the position (x_fk, y_fk) of the
# end-effector
>>> x_fk, y_fk  = TE.t
```

Calculating Analytical Solutions in Python (con'd)

```
# create symbols for lengths of the 2 links
>>> a1 = Symbol('a1')
>>> a2 = Symbol('a2')

# symbols for joints angles
>>> q1, q2 = symbols("q1:3")

# transformation to calculate the position of the end-effector
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# calculate forward kinematics matrix for end-effector
>>> TE = e.fkine([q1, q2])

# translation part of matrix gives the position (x_fk, y_fk) of the
# end-effector
>>> x_fk, y_fk  = TE.t
>>> print(x_fk)
>>> print(y_fk)

# create symbolic variables to represent the position of the end-effector
>> x, y = symbols("x, y")
```

Calculating Analytical Solutions in Python (con'd)

```
# create symbols for lengths of the 2 links
>>> a1 = Symbol('a1')
>>> a2 = Symbol('a2')

# symbols for joints angles
>>> q1, q2 = symbols("q1:3")

# transformation to calculate the position of the end-effector
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# calculate forward kinematics matrix for end-effector
>>> TE = e.fkine([q1, q2])

# translation part of matrix gives the position (x_fk, y_fk) of the
# end-effector
>>> x_fk, y_fk  = TE.t
>>> print(x_fk)
>>> print(y_fk)

# create symbolic variables to represent the position of the end-effector
>> x, y = symbols("x, y")

# x_fk = x and y_fk = y
# then x_fk**2 + y_fk**2 - x**2 - y**2 = 0
```

Calculating Analytical Solutions in Python (con'd)

```
# create symbols for lengths of the 2 links
>>> a1 = Symbol('a1')
>>> a2 = Symbol('a2')

# symbols for joints angles
>>> q1, q2 = symbols("q1:3")

# transformation to calculate the position of the end-effector
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# calculate forward kinematics matrix for end-effector
>>> TE = e.fkine([q1, q2])

# translation part of matrix gives the position (x_fk, y_fk) of the
# end-effector
>>> x_fk, y_fk  = TE.t
>>> print(x_fk)
>>> print(y_fk)

# create symbolic variables to represent the position of the end-effector
>> x, y = symbols("x, y")

# x_fk = x and y_fk = y
# then x_fk**2 + y_fk**2 - x**2 - y**2 = 0
>>> eq1 = (x_fk**2 + y_fk**2 - x**2 - y**2).trigsimp()
```

Calculating Analytical Solutions in Python (con'd)

Calculating Analytical Solutions in Python (con'd)

```
>>> print(eq1)
a1**2 + 2*a1*a2*cos(q2) + a2**2 - x**2 - y**2

>>> import sympy  # explicitly show that solve belongs to sympy
```

Calculating Analytical Solutions in Python (con'd)

```
>>> print(eq1)
a1**2 + 2*a1*a2*cos(q2) + a2**2 - x**2 - y**2

>>> import sympy  # explicitly show that solve belongs to sympy
>>> q2_sol = sympy.solve(eq1, q2)  # 2 solutions exist for q2
```

Calculating Analytical Solutions in Python (con'd)

```
>>> print(eq1)
a1**2 + 2*a1*a2*cos(q2) + a2**2 - x**2 - y**2

>>> import sympy  # explicitly show that solve belongs to sympy
>>> q2_sol = sympy.solve(eq1, q2)  # 2 solutions exist for q2
>>> print(q2_sol)
```

Calculating Analytical Solutions in Python (con'd)

```
>>> print(eq1)
a1**2 + 2*a1*a2*cos(q2) + a2**2 - x**2 - y**2

>>> import sympy  # explicitly show that solve belongs to sympy
>>> q2_sol = sympy.solve(eq1, q2)  # 2 solutions exist for q2
>>> print(q2_sol)
[-acos(-(a1**2 + a2**2 - x**2 - y**2)/(2*a1*a2)) + 2*pi,
 acos((-a1**2 - a2**2 + x**2 + y**2)/(2*a1*a2))]
```

Calculating Analytical Solutions in Python (con'd)

```
>>> print(eq1)
a1**2 + 2*a1*a2*cos(q2) + a2**2 - x**2 - y**2

>>> import sympy  # explicitly show that solve belongs to sympy
>>> q2_sol = sympy.solve(eq1, q2)  # 2 solutions exist for q2
>>> print(q2_sol)
[-acos(-(a1**2 + a2**2 - x**2 - y**2)/(2*a1*a2)) + 2*pi,
 acos((-a1**2 - a2**2 + x**2 + y**2)/(2*a1*a2))]


# expand the two equations x_fk=x,  y_fk=y
>>> eq2 = tuple(map(sympy.expand_trig, [x_fk - x,  y_fk - y]))
>>> print(eq2)
```

Calculating Analytical Solutions in Python (con'd)

```
>>> print(eq1)
a1**2 + 2*a1*a2*cos(q2) + a2**2 - x**2 - y**2

>>> import sympy  # explicitly show that solve belongs to sympy
>>> q2_sol = sympy.solve(eq1, q2)  # 2 solutions exist for q2
>>> print(q2_sol)
[-acos(-(a1**2 + a2**2 - x**2 - y**2)/(2*a1*a2)) + 2*pi,
 acos((-a1**2 - a2**2 + x**2 + y**2)/(2*a1*a2))]


# expand the two equations x_fk=x,  y_fk=y
>>> eq2 = tuple(map(sympy.expand_trig, [x_fk - x,  y_fk - y]))
>>> print(eq2)

(a1*cos(q1) + a2*(-sin(q1)*sin(q2) + cos(q1)*cos(q2)) - x,
a1*sin(q1) + a2*(sin(q1)*cos(q2) + sin(q2)*cos(q1)) - y)

# solve for sin(q1), cos(q1)
>>> q1_sol = sympy.solve(eq2, [sympy.sin(q1), sympy.cos(q1)])
>>> print(q1_sol)  # dictionary containing sin(q1) and cos(q1)
```

Calculating Analytical Solutions in Python (con'd)

```
# tan(q1) = sin(q1)/cos(q1)
>>> print(q1_sol[sin(q1)]/q1_sol[cos(q1)])
```

Calculating Analytical Solutions in Python (con'd)

```
# tan(q1) = sin(q1)/cos(q1)
>>> print(q1_sol[sin(q1)]/q1_sol[cos(q1)])

# solve for q1
```

Calculating Analytical Solutions in Python (con'd)

```
# tan(q1) = sin(q1)/cos(q1)
>>> print(q1_sol[sin(q1)]/q1_sol[cos(q1)])

# solve for q1
>>> sympy.atan2(q1_sol[sin(q1)], q1_sol[cos(q1)]).simplify()
```

Iterative (Numerical) Methods for Solving Inverse Kinematics

Can be applied to any kinematic robot structure (not robot dependent).

Iterative (Numerical) Methods for Solving Inverse Kinematics

Can be applied to any kinematic robot structure (not robot dependent).

- Slower

Iterative (Numerical) Methods for Solving Inverse Kinematics

Can be applied to any kinematic robot structure (not robot dependent).

- Slower
- In some cases they do not compute all possible solutions

Iterative (Numerical) Methods for Solving Inverse Kinematics

Can be applied to any kinematic robot structure (not robot dependent).

- Slower
- In some cases they do not compute all possible solutions
- Refining the solution through iterations

Iterative (Numerical) Methods for Solving Inverse Kinematics

Can be applied to any kinematic robot structure (not robot dependent).

- Slower
- In some cases they do not compute all possible solutions
- Refining the solution through iterations
- Initial starting point affects the solution time

Iterative (Numerical) Methods for Solving Inverse Kinematics

Can be applied to any kinematic robot structure (not robot dependent).

- Slower
- In some cases they do not compute all possible solutions
- Refining the solution through iterations
- Initial starting point affects the solution time

How? Minimise the error between the forward kinematics solution and the desired end-effector pose $\boldsymbol{\xi}_E$:

$$\boldsymbol{q}^* = \arg\min_{\boldsymbol{q}} (FK(\boldsymbol{q}) - \boldsymbol{\xi}_E)$$

Numerical Methods for Solving Inverse Kinematics (cont'd)

Various classical numerical methods can be applied, including among others:

Numerical Methods for Solving Inverse Kinematics (cont'd)

Various classical numerical methods can be applied, including among others:

- Newton-Raphson: first order approximation of original equations

Numerical Methods for Solving Inverse Kinematics (cont'd)

Various classical numerical methods can be applied, including among others:

- Newton-Raphson: first order approximation of original equations
- Levenberg–Marquardt optimisation: using the second order derivative for the approximation of the original system.

The Newton-Raphson Algorithm

The Newton-Raphson Algorithm

The slope (tangent) of a function $f(x)$ for $x = x_n$ is defined (calculated) by the derivative of the function at that point:

The Newton-Raphson Algorithm

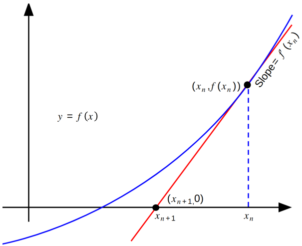
The slope (tangent) of a function $f(x)$ for $x = x_n$ is defined (calculated) by the derivative of the function at that point:

$$f'(x_n) = \frac{f(x_n)}{x_n - x_{n+1}} \tag{4}$$

The Newton-Raphson Algorithm

The slope (tangent) of a function $f(x)$ for $x = x_n$ is defined (calculated) by the derivative of the function at that point:

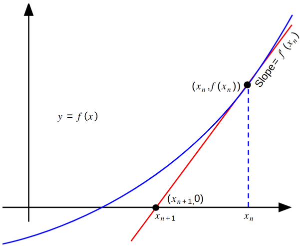
$$f'(x_n) = \frac{f(x_n)}{x_n - x_{n+1}} \tag{4}$$



The Newton-Raphson Algorithm

The slope (tangent) of a function $f(x)$ for $x = x_n$ is defined (calculated) by the derivative of the function at that point:

$$f'(x_n) = \frac{f(x_n)}{x_n - x_{n+1}} \tag{4}$$



Then the approximated solution for finding the root of f (where $f(x) = 0$) can be calculated iteratively by:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{5}$$

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
>>> def E(q):
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
>>> def E(q):
>>>     return np.linalg.norm(e.fkine(q).t - des_pos)
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
>>> def E(q):
>>>     return np.linalg.norm(e.fkine(q).t - des_pos)

# Minimise the error (E) between the forward kinematics solution and the
# desired position of the end-effector - Use optimize from SciPy
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
>>> def E(q):
>>>     return np.linalg.norm(e.fkine(q).t - des_pos)

# Minimise the error (E) between the forward kinematics solution and the
# desired position of the end-effector - Use optimize from SciPy
>>> sol = optimize.minimize(E, [0, 0])
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
>>> def E(q):
>>>     return np.linalg.norm(e.fkine(q).t - des_pos)

# Minimise the error (E) between the forward kinematics solution and the
# desired position of the end-effector - Use optimize from SciPy
>>> sol = optimize.minimize(E, [0, 0])

>>> print(sol.x)   # required q values to achieve des_pos for end-effector
[1.91964289 3.7933814 ]
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
>>> def E(q):
>>>     return np.linalg.norm(e.fkine(q).t - des_pos)

# Minimise the error (E) between the forward kinematics solution and the
# desired position of the end-effector - Use optimize from SciPy
>>> sol = optimize.minimize(E, [0, 0])

>>> print(sol.x)   # required q values to achieve des_pos for end-effector
[1.91964289 3.7933814 ]

# Computing the forward kinematics confirms that the
# solution is correct - Recall that we started the
# calculation for des_pos = np.array([0.5, 0.4])
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
>>> def E(q):
>>>     return np.linalg.norm(e.fkine(q).t - des_pos)

# Minimise the error (E) between the forward kinematics solution and the
# desired position of the end-effector - Use optimize from SciPy
>>> sol = optimize.minimize(E, [0, 0])

>>> print(sol.x)   # required q values to achieve des_pos for end-effector
[1.91964289 3.7933814 ]

# Computing the forward kinematics confirms that the
# solution is correct - Recall that we started the
# calculation for des_pos = np.array([0.5, 0.4])
>>> e.fkine(sol.x).printline()
```

Calculating Numerical Solutions in Python

```
>>> a1 = 1; a2 = 1
>>> q1, q2 = symbols("q1:3")
>>> e = ET2.R()*ET2.tx(a1)*ET2.R()*ET2.tx(a2)

# Desired position of the end-effector
>>> des_pos = np.array([0.5, 0.4])

# define the error (E) function
>>> def E(q):
>>>     return np.linalg.norm(e.fkine(q).t - des_pos)

# Minimise the error (E) between the forward kinematics solution and the
# desired position of the end-effector - Use optimize from SciPy
>>> sol = optimize.minimize(E, [0, 0])

>>> print(sol.x)   # required q values to achieve des_pos for end-effector
[1.91964289 3.7933814 ]

# Computing the forward kinematics confirms that the
# solution is correct - Recall that we started the
# calculation for des_pos = np.array([0.5, 0.4])
>>> e.fkine(sol.x).printline()

t = 0.5, 0.4; -32.7°
```

Other topics/issues in Robotics

Other topics/issues in Robotics

1. **Forward Instantaneous Kinematics**

Other topics/issues in Robotics

1. **Forward Instantaneous Kinematics**
   → Given all members of the kinematic chain and the rates of motion about all joints, find the total velocity of the end-effector.

Other topics/issues in Robotics

1. **Forward Instantaneous Kinematics**
   - $\rightarrow$ Given all members of the kinematic chain and the rates of motion about all joints, find the total velocity of the end-effector.
   - $\rightarrow$ Usage of the Jacobian matrix $\boldsymbol{J}(\boldsymbol{q})$

Other topics/issues in Robotics

1. **Forward Instantaneous Kinematics**
   - → Given all members of the kinematic chain and the rates of motion about all joints, find the total velocity of the end-effector.
   - → Usage of the Jacobian matrix $\boldsymbol{J}(\boldsymbol{q})$

   $$^{k}\boldsymbol{v}_N = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{6}$$

   where $^{k}\boldsymbol{v}_N$ is the velocity of the end-effector expressed in any frame k
2. **Inverse Instantaneous Kinematics**

Other topics/issues in Robotics

1. **Forward Instantaneous Kinematics**
   → Given all members of the kinematic chain and the rates of motion about all joints, find the total velocity of the end-effector.
   → Usage of the Jacobian matrix $\boldsymbol{J}(\boldsymbol{q})$

   $$^{k}\boldsymbol{v}_N = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{6}$$

   where $^{k}\boldsymbol{v}_N$ is the velocity of the end-effector expressed in any frame k
2. **Inverse Instantaneous Kinematics**
   → Given the positions of all the members of the kinematic chain and the total velocity of the end-effector, find the rates of the motion of all joints.

Other topics/issues in Robotics

1. **Forward Instantaneous Kinematics**
   - → Given all members of the kinematic chain and the rates of motion about all joints, find the total velocity of the end-effector.
   - → Usage of the Jacobian matrix $\boldsymbol{J}(\boldsymbol{q})$

   $$ {}^{k}\boldsymbol{v}_N = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{6} $$

   where ${}^{k}\boldsymbol{v}_N$ is the velocity of the end-effector expressed in any frame k

2. **Inverse Instantaneous Kinematics**
   - → Given the positions of all the members of the kinematic chain and the total velocity of the end-effector, find the rates of the motion of all joints.
   - → Usage of the inverse of the Jacobian matrix

   $$ \dot{\boldsymbol{q}} = \boldsymbol{J}^{-1}(\boldsymbol{q})\boldsymbol{v}_n \tag{7} $$