**Code explaining**

The code defines a **robot\_arm** class with methods for generating a random configuration of joint angles, performing direct and inverse kinematics to find the position of the end-effector and corresponding joint angles, finding the minimum distance between two points in free space, and generating a collision-free path between two points.

The **robot\_arm** class has several attributes that define the physical properties of the arm, including the lengths of the three arm segments (**l1**, **l2**, **l3**), the coordinates of the center of the third circle (**center**), the radius of the circles (**ro**), the height of the arm (**height**), and the joint angle limits (**lim\_min** and **lim\_max**).

The **gen\_rand\_conf()** method generates a random configuration of joint angles within the limits of the robot. The **direct\_kinematics()** method takes a vector of joint angles as input and returns the position of the end-effector. The **inverse\_kinematics()** method takes a position vector as input and returns a vector of corresponding joint angles.

The **get\_indix\_of\_min\_distance\_point()** method finds the index of the closest point to a given point in a set of free space points. The **find\_path()** method uses a modified version of Dijkstra's algorithm to generate a path between two points in free space. The **collision\_checker()** method checks if a given point is outside the bounds of free space or inside one of the three circles in the workspace.

The code imports several modules including **numpy**, **matplotlib**, and **scipy**. The **matplotlib.animation** and **IPython.display** modules are used to animate the robot arm's movement in 3D space.

The algorithm uses a sampling-based approach to motion planning called the Rapidly exploring Random Tree (RRT) algorithm. The RRT algorithm builds a tree of configurations that connect the start and goal points. At each step, the algorithm generates a random configuration, and it tries to extend the tree towards that configuration. If the new configuration is feasible (i.e., collision-free), it adds it to the tree. The algorithm continues this process until it finds a path connecting the start and goal configurations or reaches a certain time or iteration limit.

The code uses a variant of RRT called RRT\* (RRT star), which tries to optimize the tree by biasing the expansion towards configurations that have not been explored yet or configurations that are closer to the goal. The implementation also uses a collision checker to avoid sampling and expanding configurations that lead to collisions with obstacles in the environment.

Once the RRT\* algorithm finds a collision-free path, the implementation uses inverse kinematics to find the joint angles for each point in the path. It then uses forward kinematics to compute the Cartesian position of the end effector (the tip of the robot arm) for each joint angle. Finally, the implementation uses Matplotlib to visualize the robot arm moving along the collision-free path.

The algorithm is based on the idea of supervised learning, where the model is trained using labeled data to make predictions on new, unseen data.

The model is typically trained using an iterative process where the weights of the model are updated based on the errors made on the training data.

One of the key challenges in building a good model is finding the right balance between overfitting and underfitting. Overfitting occurs when the model becomes too complex and fits the noise in the training data, while underfitting occurs when the model is too simple and cannot capture the patterns in the data.

To prevent overfitting, various techniques such as regularization, early stopping, and dropout are commonly used in deep learning.

The success of the algorithm often depends on the quality and quantity of the training data, as well as the architecture and hyperparameters of the model.