Implementation of biologically-inspired dynamical systems for movement generation: automatic real-time goal adaptation and obstacle avoidance

Alan Gomez, Samuel Parra, and Brennan Penfold

***Abstract*—**

I. Introduction

Humanoid robots and arm like manipulators have a great many current and potential, useful applications. Currently robotic manipulators are used in a range of manufacturing processes like painting and spot welding [1]. Another great application of humanoid robotics is to help people who cannot help themselves, as a separate system, or as a prosthetic. Robotic prosthetics have been implemented previously, like Dean Kamen's Luke Arm [2], but the control of such device is limited. In human applications the control system used in areas like manufacturing cannot be just dropped into place, there are many considerations that are required from both the software and hardware perspectives. In manufacturing applications manipulators typically use precomputed trajectories for robustness and efficiency, but this is unsuitable in a human environment, where obstacles and goals are constantly on the move [3]. The current approach in prosthetics is to use a human controller to solve this problem, by adding a joystick in a shoe [2] or another such interface, so the operator can directly control the movement of the arm. While this addresses the adaptable control issue, it can be quite taxing for the user, requiring a lot of concentration just to perform basic tasks. This solution is also not very viable for stand-alone assistant robots, the cost of training an operator would far outweigh the benefit of using a robot and operator instead of a single human nurse or assistant.

One solution to the control problem is to use dynamic movement primitives, which can be generated during the movement itself, allowing for real-time obstacle avoidance and goal changes. Previous approaches [4] used were initially not able to handle moving goals, but improvements made by H. Hoffmann et al. [3] by applying a potential field method to the process overcame this limitation. The use of dynamic movement primitives in the potential field method also overcame the local minimum problem, that is typically inherent with potential fields. Hence this paper is the basis of the project. There are some limitations to this method the most significant being the representation of the obstacles and manipulator. The method presented by H. Hoffman et al.[3] only considers points in cartesian space, where other methods, like those discussed by O. Khatib [5], represent objects as basic primitives with volume and shape.

For objects to be avoided first they must be located relative to the manipulator, and that is the core concept of the library built for this project. Biologically-inspired dynamical systems for movement generation [3] does not tackle the problem of distances between objects, instead a precursor paper was used, Real-Time Obstacle Avoidance for Manipulators and Mobile Robots [5]. This paper goes into details on how to find the distance between two obstacles, which was modified to find the shortest vector between the two obstacles, allowing for it to be applied to the dynamic movements primitives control algorithm discussed in Biologically-inspired dynamical systems for movement generation [3]. The these two techniques, along with the addition of distance monitoring with all links, has the potential to produce a better algorithm. The combination of the two techniques means that both the volume of the object is considered, and the trajectory planning can occur online allowing it to be adapted to any changes in a fast, efficient manner. These properties along with the addition of obstacle distance calculations for all links means that humanoid robots and manipulators would be more robust and safer to use in dynamic, human environments.

Biolog inspir dy- nam- i- cal sys- tems for move- ment gen- er- a- tion is re- peated twice in the last para- graph of the in- tro- duc- tion

II. Approach

A. Distance to objects

To compute the closest distance between two objects we first obtain the closest points between the objects and then the distance between these two points.

1) Sphere - Sphere: To compute the closest distance between two spheres, one can easily calculate the distance between the center of both spheres and subtract the radius of both spheres.

2) Sphere - Capsule: To compute the closest distance between a capsule C and a sphere S, one can compute the distance between the center of $S = p_S$ and the axis of symmetry of $C = l_C$ and then subtract the radius of the sphere and the capsule.

Chang be- tween two sphere and so on

The closest point, on l_C, to p_S is given by (1),

$$\begin{aligned} x &= x_1 + \lambda(x_2 - x_1) \\ y &= y_1 + \lambda(y_2 - y_1) \\ z &= z_1 + \lambda(z_2 - z_1) \end{aligned} \tag{1}$$

where the start point of l_C is $m1(x_1, y_1, z_1)$, the end point is $m2(x_2, y_2, z_2)$, and λ is given by (2).

$$\lambda = \frac{(c-m_1)\cdot(m_2-m_1)}{l^2} \quad (2)$$

3) Capsule - Capsule: To compute the closest distance between two capsules (capsule A and capsule B), with axis of symmetry $l_A = \overline{m1_A m2_A}$ and $l_B = \overline{m1_B m2_B}$, where A is the longest capsule, one can consider four different cases:

1) $m1_B$ and $m2_B$ can be perpendicularly projected onto l_A
2) $m1_B$ and $m2_B$ cannot be perpendicularly projected onto l_A
3) only $m1_B$ can be perpendicularly projected onto l_A
4) only $m2_B$ can be perpendicularly projected onto l_A

Depending on these cases, the following steps are going to be calculated with different objects.

In case 1, capsule M will be capsule A, and capsule O will be capsule B.

In case 2, if $m1_B$ is closer to l_A than $m2_A$ is to l_B, then capsule M will be capsule A, and capsule O will be capsule B. Else capsule M will be capsule B, and capsule O will be capsule A.

In case 3, if $m2_B$ is closer to l_A than $m1_A$ is to l_B, then capsule M will be capsule A, and capsule O will be capsule B. Else capsule M will be capsule B, and capsule O will be capsule A.

In case 4, if if $m1_B$ is closer to l_A than $m2_B$ is to l_A, then capsule M will be capsule A, and capsule O will be capsule B. Else capsule M will be capsule B, and capsule O will be capsule A.

With capsule M and capsule P, one can follow the next steps. The first step is to place the reference frame of M in its center of mass, with the Z-axis in the direction of l_M. The second step is to project l_O onto the XY-plane given by the reference frame of M (Figure 1), the result of this projection would be the line l_P. The third step is to calculate the closest point on l_P to the origin of the reference frame of P by using (1). The final step is to compute the distance and subtract the radius of the capsule and the sphere. If the true closest points on the original axis of symmetry of the capsules are needed, one can project back the obtained points.

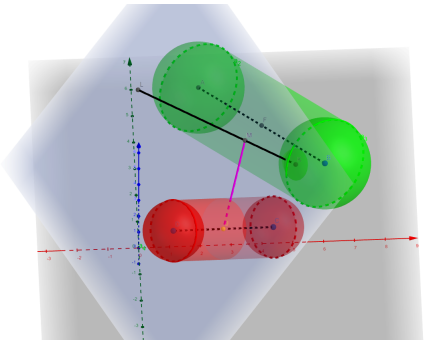


Fig. 1: Capsule - capsule

B. Potential field

To avoid collisions we decided to implement the method presented by Hoffmann et al. [3]. This method is based on dynamic movement primitives (DMP), which are used to generate a trajectory $x(t)$ with a velocity $v(t)$. These DMP are motivated from the dynamic of damped spring and can generate trajectories in many dimensions. These can be used to describe accelerations in 3D space that move towards a goal:

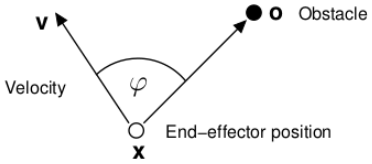


Fig. 2: Graphical representation of steering angle [**?**]

$$\dot{v} = K(g-x) - Dv - K(g-x_0)s + Kf(s) + p(x,v) \quad (3)$$

Where g is the goal, x is the current position, x_0 is the starting position, v is the current velocity, s is the phase variable, D is the damping constant, and K is the spring constant. The function $f(s)$ is a non-linear function used to define a learned path for the manipulator to take. However, the terms dependent on s are not relevant for our application; thus we decided to not include it in the final equation.

The last term is the potential field generated by the obstacles. This term has a repel effect on the end effector, making it avoid obstacles. The potential field is calculated as follow: We first determine the steering angle φ using equation 4, which is the angle between the velocity vector and a vector from the end effector to the obstacle as seen in Fig 2. In our implementation, this vector is the shortest distance between the end-effector and the obstacle, as the original method considered points instead of volumes.

$$\varphi = cos^{-1}((o-v)^T v/(\|o-x\| \cdot \|v\|)); \quad (4)$$

This steering angle determines how sharp the end effector steers away from the obstacle. The closer the angle is to 0, the more abrupt is the change of direction. The potential field is given by:

$$p(x,v) = \gamma \sum_i R_i v \varphi_i exp(-\beta \varphi_i) \quad (5)$$

Where γ and β are constants, and R is a rotation matrix which rotates by the axis $r = (o-x) \times v$ with an angle of rotation of $\pi/2$. This rotation matrix can be calculated with the Rodrigues' rotation formula [6]. The final potential field is the sum of the potential fields generated by all the obstacles. Our final equation converges towards the goal avoiding the obstacles in its way.

$$\dot{v} = K(g-x) - Dv + p(x,v) \quad (6)$$

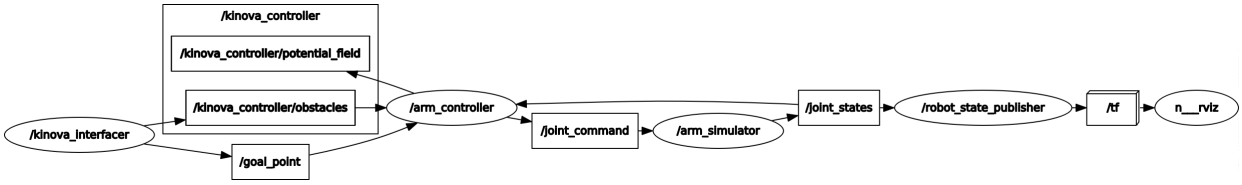


Fig. 3: ROS node graph for single arm

C. Collision monitoring library

Our library was designed to be platform and framework independent. To achieve this, the library user must implement the Arm interface. This provides the flexibility of choosing how to describe mathematically the manipulator and update its position. Consider the case of two developers, one wants to use KDL for the forward and backwards kinematics, while the other one wants to derive the equations manually for a closed form solution. Both developers are supported by this library.

In order to monitor the distances from the links of the arm to the obstacles in the workspace we represent the geometry of the links with the use of capsules. This representation enables us to calculate the distances between the links themselves in order to monitor self-collisions along with regular obstacles. This requires that the developer updates the geometric representation alongside the arms movement in the Arm implementation.

D. Controller

The manipulator is controlled with a control loop that is executed at a specified rate. This loop is responsible for updating the geometric representation to match the real life manipulator and calculate the new velocity with the collision avoidance algorithm. In the loop the developer must keep the current position, velocity, positions of obstacles, and goal updated for the algorithm to produce an accurate output.

III. Implementation

The final product that combines all the obstacle monitoring and avoidance techniques was implemented using the Robotic operating system (ROS) framework, specifically ROS Kinetic. ROS has a few advantages in this area, the first and foremost being the Kinova Kortex library [7], which is a high and low level control API for Kinova robotic manipulators, including the one chosen for demonstration. This library has a lot of useful features from the MoveIt control algorithm to the fully fledged gazebo simulation of the arm. The final software produced does actually rely on any of the Kortex libraries, but was instrumental in testing and development.

The ROS package can be broken down into 3 sections, the controller, simulator and visualisation. The control section is the implementation of Biologically-inspired dynamical systems for movement generation [3], the collision avoidance library is implemented inside a control loop that uses the monitor class to obtain the shortest distance vectors from the obstacles to the end-effector. This shortest distance is then passed into the potential field method along with the current velocity of the manipulator. From this the new velocity is derived and then passed into the simulator via an inverse kinematics function of the kinova_arm class, mapping the velocity from cartesian space to the manipulators joint space.

The simulation software takes the joint velocities from the controller and updates the current state of the simulated manipulator, and then passes the current state back to the controller as well as forwards to the visualisation. The simulation algorithm it's self is very basic and is based around a discrete time integration technique:

$$\theta_i = \theta_{i-1} + \dot{\theta}_{i-1} \times (t_i - t_{i-1})$$

While rudimentary, as it doesn't consider inertia or any other complex parameters, it functions as desired for the testing and demonstration of a robotic manipulators trajectory control algorithm.

The basic ROS node graph can be seen in 3 where the interfacer is a basic terminal interface to send goals to the arm controller and the visualisation is performed by the built-in robot_state_publisher node and Rviz.

IV. Experiments

A. Collision Monitoring

The collision monitoring library has no real room for errors, the distance between objects was confirmed using Geogebra models, and the rest of the software relies on those calculations. The core performance measure of the collision monitoring library can be considered speed, hence the use of C++ instead of a higher level interpreted language like python. The performance test was performed on an Ubuntu 16.04 machine with a Intel Core i5-7200U 2.5GHz CPU. The following operations were timed using the C++ chrono library:

- The joint initialisation of one KinovaArm and Monitor class.
- The combination of initialising a Sphere and adding it to the Monitor class.
- The updatePose function, which updates the internal kinematic representation of the arm.
- The distanceToObjects function, calculating the distance from each of the manipulators links to the obstacle.
- The distanceBetweenArmLinks function, measuring the distance from each of the links to the other links in the same manipulator.

1) Results: Each operation was performed 100 times with a single spherical obstacle and averaged to produce the final results given in Table I below.

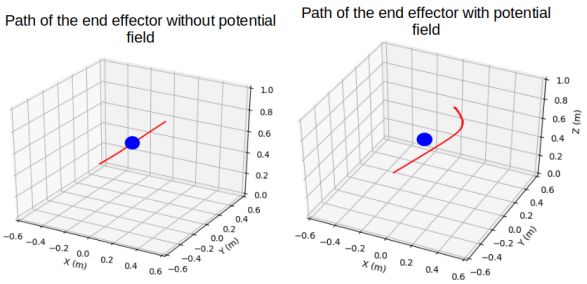
Operation	Time (seconds)
Init arm and monitor	0.00103831
Init and add object	0.00000276
Update the arm position	0.00012871
Distance to obstacle	0.00026022
Distance to self links	0.00199209

TABLE I: Execution times for collision monitoring tasks

From a cursory glance it is clear that the functions are quite fast all operations take less than two milliseconds to complete, but some of the operations are not as fast as desired. The initialisation operations are not that important when is comes to overall speed, since they tend to only be run at the start of the program, and potentially at random intervals the program for the obstacle initialisation. The distance to obstacles and between arm links along with the update function are the main operations to be concerned with, these run every loop and determine the final update rate of the software. The update rate of the arm is very quick, and the distance to obstacles only taking twice as much. It is important to note though that the time for distance to obstacles to execute relies on the number of obstacles, as the number of obstacles increases so does the time to calculate, in a linear fashion. This means that with 40 obstacles the software will be restricted to 10Hz. Assuming only one obstacle the other large impact to overall performance in the distance to self links operation, taking up over four times the time used by the other two core functions combined. This was somewhat expected, if not to this extent, since the software was designed for portability in mind, which limits the possible optimisations available, still further investigation into the limiting factors to should be performed. The final core loop time for the collision monitoring library is a little over 4.7 milliseconds, which would give a update rate of 200hz, but this would benefit significantly by the exclusion of self monitoring functions.

B. *Single arm obstacle avoidance*

To assess the effectiveness of the algorithm we place several obstacles in the vicinity of the manipulator, and then command it to move between points in the workspace. First, we used the same set of obstacles, initial point and goal point for two experiments with two different values for γ as seen in Fig. 4. When $\gamma = 0$ there is no influence of the potential field in the velocity, hence the end effector does not avoid the obstacle. Whereas when $\gamma = 35$ the effect of the potential field steer away the end effector from the obstacle in its path. From these results we can assess that the DMP will converge to the goal point, and the value of γ has an impact in the effectiveness of the algorithm.



(a) $\gamma = 0$ (b) $\gamma = 35$

Fig. 4: Movement with different values for γ

In Fig. 5 we can observe the behavior of the potential field when several obstacles are nearby. Note that the potential field's magnitude is larger when the end effector is approaching the obstacles and gets smaller after it passes the obstacles. This is the effect of the steering angle, as when the end effector passes the obstacles the angle between the current velocity and the obstacles is large. It should also be noted that the direction of the potential field effect is not uniform between small steps.

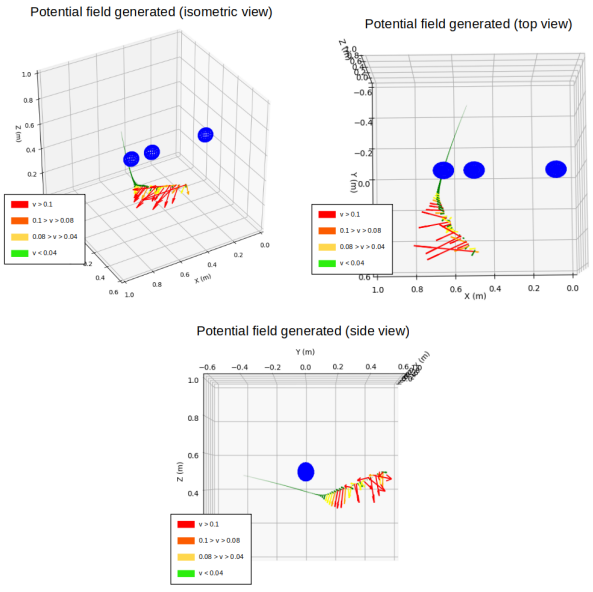


Fig. 5: Potential field generated during the movement

For one of the experiments we placed several obstacles in the workspace and command it to move between points. The results of the collision avoidance algorithm can be seen from several perspectives in fig 6. The red line is the path of the end effector from the initial point to the goal. In fig 7 the velocity vectors in their respective point in time can be observed.

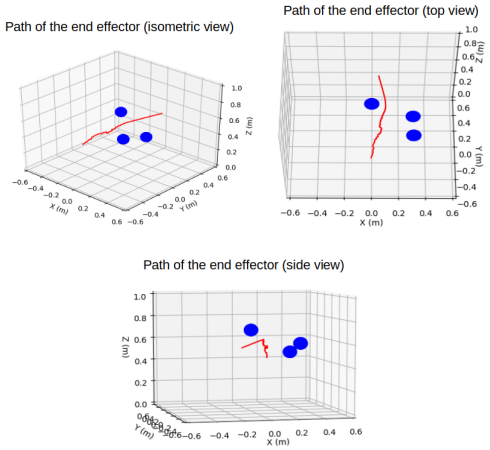


Fig. 6: Path of the end effector as it moves through obstacles

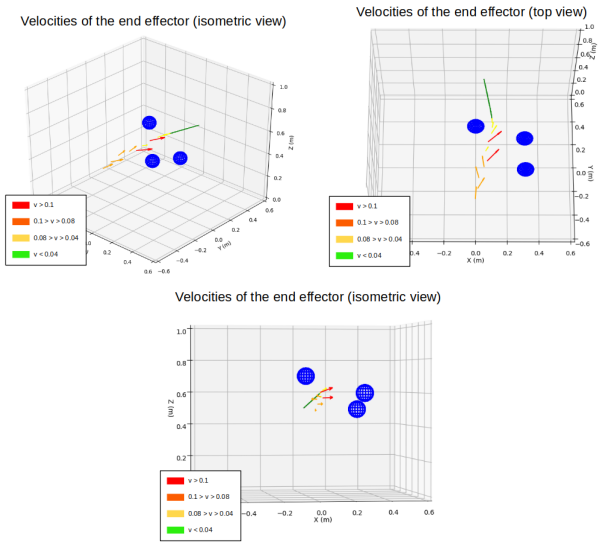


Fig. 7: Cartesian velocities generated by the algorithm

As it can be observed from Fig. 6 and Fig. 7 the collision avoidance algorithm can generate velocities that avoid several obstacles without problems.

C. *Dual arm collision avoidance*

In order for two manipulators to work together in the same obstacles it is important that these do not collide with each other. In this experiment we simulated two instances of the Kinova arm placed close to each other so their workspaces would overlap. Each arm is individually controlled, meaning we could place one arm on the path of the other to assess if the arms avoided collisions with each other, as observed in Fig 8. The objective is to assess if the arms avoid colliding with each other.

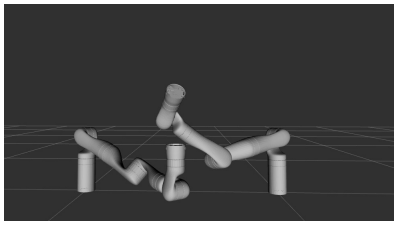


Fig. 8: Two arms sharing a space

In Fig. 9 the path of the end effector of first Kinova arm can be seen as well as the pose of the second arm. The pose of the second arm remained constant during the movement of the first arm.

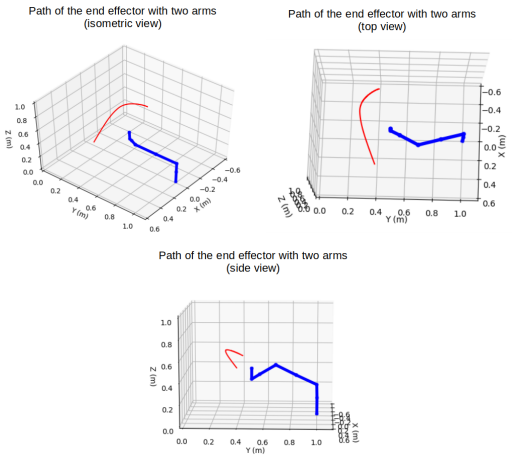


Fig. 9: Path of the end effector of one arm

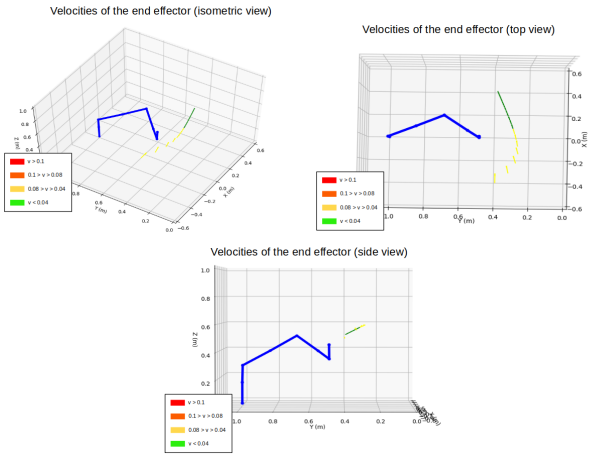


Fig. 10: Cartesian velocities generated

From these results we can see that a long as the geometric representation of the two arms is accurate, the collision avoidance algorithm can be use to avoid some collisions. the reason why some cases collisions cannot be avoided is because this algorithm only manages to avoid collisions with the end effector. This means that collisions between the body of the first arm the second arm are still plausible.

V. Use cases

A. Single arm

The library presented in this paper can be used with a single manipulator to avoid the collision of its end-effector with objects in the environment. The user can use the Arm interface to provide the necessary information about the manipulator. The user can then also add different obstacles that are present in the environment. The pose of the manipulator and the obstacles need to be updated using the methods given by the library.

B. Dual arm

One can also use the library with two manipulators by creating two instances of the implementation of the Arm interface and adding the links of the opposite manipulator as obstacles represented by capsules. This will prevent the collision of the end-effector of both robots with the opposite manipulator

C. Multiple arms

In the same way, as one can use the library with two manipulators, one can add as many manipulators as the computational resources allow it.

D. Self collision

Finally, one can also use this library to prevent the collision of the end-effector of the manipulator with its other links. This can be done by implementing the Arm interface and adding each link of the robot as an obstacle represented by a capsule.

VI. Conclusion

The first line is weird, it should be more specific about manipulators

For robots and robotic manipulators to be used in a human environment they first need to be able to adapt to a constantly changing, dynamic conditions. This can be done in a robust and safe manner using a combination of two different methods discussed by H. Hoffmann et al.[3] and O. Khatib [5]. by implementing the obstacle monitoring by O. Khatib [5] and the collision avoidance by H. Hoffmann et al.[3], along with monitoring collisions for all links, robotic manipulators are more resilient to changes in obstacle and goal placement. The final collision monitoring library can have an update rate of 200Hz with all features enabled, this is a viable update rate, but this would become drastically impacted if more that 10 obstacles were to be added to the environment. From the experiments it can be concluded that the DMPs used in this algorithm converge to the goal, and that the potential field term influences the end effector's path to steer away from obstacles. It can also be observed that the effectiveness of the algorithm is dependent on the parameters of the DMP equation, hence it is important for a developer to fine tune this parameters until a certain level of performance is reached.

While the algorithm produced gives some significant benefits over previous methods, there is still a lot of potential improvements. The collision monitoring library could benefit greatly from some optimisation of the self collision functions, along with the addition of more shapes. The current shapes are a basic sphere and capsule, with the addition of an n-ellipsoid primitive a large majority of basic shapes and structures could be constructed by the combination of the primitives. Finally, in order to avoid more serious collisions, the algorithm should consider the entire body of the manipulator rather than just the end effector.

Appendices

Acknowledgments

References

[1] M. S. Fadali and A. Visioli, Introduction to Digital Control, in Digital Control Engineering, Elsevier, 2013, pp. 18.

[2] S. Adee, Dean Kamens Luke Arm Prosthesis Readies for Clinical Trials - IEEE Spectrum, IEEE Spectrum: Technology, Engineering, and Science News, Feb. 01, 2008. https://spectrum.ieee.org/biomedical/bionics/dean-kamens-luke-arm-prosthesis-readies-for-clinical-trials (accessed Jun. 25, 2020).

[3] H. Hoffmann, P. Pastor, D.-H. Park, and S. Schaal, Biologically-inspired dynamical systems for movement generation: Automatic real-time goal adaptation and obstacle avoidance, in 2009 IEEE International Conference on Robotics and Automation, Kobe, May 2009, pp. 25872592, doi: 10.1109/ROBOT.2009.5152423.

[4] F. Janabi-Sharifi and D. Vinke, Integration of the artificial potential field approach with simulated annealing for robot path planning, in Proceedings of 8th IEEE International Symposium on Intelligent Control, Aug. 1993, pp. 536541, doi: 10.1109/ISIC.1993.397640.

[5] O. Khatib, Real-Time Obstacle Avoidance for Manipulators and Mobile Robots The International Journal of Robotics Research, vol. 5, no. 1, pp. 9098, Mar. 1986, doi: 10.1177/027836498600500106.

[6] Belongie, Serge. "Rodrigues' Rotation Formula." From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein. https://mathworld.wolfram.com/RodriguesRotationFormula.html

[7] Kinovarobotics/ros_kortex. Kinova Robotics, 2020.