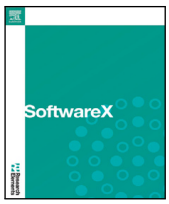
SoftwareX 24 (2023) 101532

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

Taming the Panda with Python: A powerful duo for seamless robotics programming and integration

Jean Elsner

Munich Institute of Robotics and Machine Intelligence (MIRMI), Technical University of Munich (TUM), Munich, Germany

ARTICLE INFO

Keywords:
Robotics Python software
Franka Emika robots
Real-time control
Panda

ABSTRACT

Franka Emika robots have gained significant popularity in research and education due to their exceptional versatility and advanced capabilities. This work introduces panda-py — a Python interface and framework designed to empower Franka Emika robotics with accessible and efficient programming. The panda-py interface enhances the usability of Franka Emika robots, enabling researchers and educators to interact with them more effectively. By leveraging Python's simplicity and readability, users can quickly grasp the necessary programming concepts for robot control and manipulation. Moreover, integrating panda-py with other widely used Python packages in domains such as computer vision and machine learning amplifies the robot's capabilities. Researchers can seamlessly leverage the vast ecosystem of Python libraries, thereby enabling advanced perception, decision-making, and control functionalities. This compatibility facilitates the efficient development of sophisticated robotic applications, integrating state-of-the-art techniques from diverse domains.

Code metadata

Current code version	v0.6.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00483
Permanent link to Reproducible Capsule	
Legal Code License	Apache-2.0
Code versioning system used	git
Software code languages, tools, and services used	C++, Python
Compilation requirements, operating environments & dependencies	Python ≥ 3.7, Eigen, libfranka
If available Link to developer documentation/manual	https://jeanelsner.github.io/panda-py/
Support email for questions	jean.elsner@tum.de

1. Introduction

In recent years, Python has emerged as a dominant language in the machine learning community, thanks to its extensive libraries and frameworks such as TensorFlow, PyTorch, and scikit-learn [1–3]. However, its popularity is not limited to machine learning alone. Python is gaining significant traction in the robotics community as well [4]. While there have been occasional voices of concern from the robotics community regarding Python's performance for real-time and resource-intensive robotics tasks, it is worth noting that performance-critical components can be implemented in languages like C or C++ and seamlessly integrated with Python [5]. This combination allows developers to harness the high-level features and ease of use provided by Python while still achieving the desired performance [6]. Additionally, Python's cross-platform compatibility, portability, and extensive ecosystem of libraries make it an attractive choice for robotics. The language's ease of use and rapid prototyping capabilities further contribute to its growing adoption in the robotics community, enabling researchers and developers to quickly iterate, experiment, and deploy robotics systems with ease. Finally, the programming language's immense popularity and ease of use make it an excellent choice for robotics education, enabling students to quickly learn and experiment with robotics concepts.

Franka Emika robot manipulators have gained significant popularity in research and education due to their exceptional capabilities and versatility. These robots are highly sought after for their industrial

E-mail address: jean.elsner@tum.de.

https://doi.org/10.1016/j.softx.2023.101532
Received 26 July 2023; Received in revised form 16 September 2023; Accepted 17 September 2023
Available online 26 September 2023

repeatability, force sensitivity, and torque control interface, making them well-suited for various applications. The ability to precisely repeat tasks with high accuracy and their sensitive force feedback capabilities, enables researchers to explore areas such as human–robot interaction, collaborative robotics, and intricate manipulation tasks [7]. Efforts are already underway to integrate Franka Emika robots into educational settings, including schools in Germany, where the user-friendly browser-based interface called "Desk" enables users to program the robots graphically using drag and drop for simple tasks, facilitating robotics education at various levels [8]. However, for more advanced applications, users will have to use the provided torque control interface through an open-source C++ client library called libfranka. The C++ library has rather strict real-time requirements, and setting it up and effectively programming it can be a daunting task for novice programmers. On the other hand, panda-py provides a safe and beginner-friendly interface that allows for rapid prototyping. While there have been efforts to provide Python software for Franka Emika robots [9,10], panda-py is the only software known to the author that provides direct control, packaged installation, a Desk client, and fully wraps the libfranka API.

The panda-py[1] framework simplifies the programming, deployment, and installation process for Franka Emika robot systems. It offers an all-in-one solution with pre-packaged dependencies, ensuring a seamless experience out of the box. With panda-py, researchers and developers can focus on their work without the hassle of manual setup, benefiting from its user-friendly interface, extensive Python ecosystem, and effortless integration. It streamlines the process, making programming and experimentation with Franka Emika robots more accessible and efficient for research and education purposes. In this paper, the author aims to demonstrate the utility of panda-py in a tutorial style.[2]

2. Installation and setup

The panda-py software is implemented as a Python package. Specifically, it is distributed as a Python wheel, i.e., a pre-built binary that can be installed using the package manager pip. The wheel includes all the needed dependencies to connect to and control the robot, while the package manager will install the appropriate versions for the local platform. To install the package, execute

```
pip install panda-python
```

from a terminal.[3] Franka Emika robots come with a browser-based interface called Desk [7] which is used to perform system-level tasks such as locking/unlocking brakes, activating the libfranka interface, and change various settings. This interface can only be accessed from the computer connected to the robot using a browser. For convenience, panda-py includes a client that enables users to access the Desk from Python code instead. You can use `panda_py.Desk` for example to connect to the Desk running on the control unit to unlock the brakes and activate the Franka Research Interface (FCI) for robot torque control

```
import panda_py

desk = panda_py.Desk(hostname, username,
 ↪ password)
desk.unlock()
desk.activate_fci()
```

[1] Earlier models of the Franka Emika robot were known as Panda, hence the name. There is no relation to the similarly named Python data analysis library pandas.

[2] All of the provided examples are also available online and are ready to run on real hardware.

[3] Visit the online repository at http://github.com/JeanElsner/panda-py for more information on how to build from source or install specific versions.

Table 1
Average runtimes of common panda-py API calls executed in Python 3.10 and C++17. Each call was executed and timed 10 000 times with random state-space samples (5 waypoints for motion generation) on an Intel® Core™ i7-8565U CPU @ 1.80 GHz.

API call	Python runtime (s)	C++ runtime (s)
fk	2.49×10^{-6}	1.84×10^{-6}
ik	2.34×10^{-6}	7.65×10^{-7}
JointMotion	1.73×10^{-2}	1.70×10^{-2}
CartesianMotion	9.66×10^{-3}	8.10×10^{-3}

where the variable hostname holds the configured IP address or hostname of the robot and username and password refer to the login information of a user with access to the Desk. Once the robot is unlocked, a connection to the hardware can be established. Connect to the robot using the Panda class. The default gripper from Franka Emika[4] does not support real-time control and can be controlled using the libfranka bindings directly.

```
from panda_py import libfranka

panda = panda_py.Panda(hostname)
gripper = libfranka.Gripper(hostname)
```

The Panda class is a high-level wrapper with various convenience functions over libfranka's robot class. However, panda-py also includes bindings for all the low-level libfranka types and functions as part of a subpackage that may be used directly. This feature is used above to connect to the gripper.

For the remainder of the tutorial, we will assume that a connection to the robot hardware was established, and the Panda and Gripper instances were assigned to the variables panda and gripper, respectively.

3. Basic Robot control

The panda-py package offers its users powerful features out of the box. There are modules for time-optimal motion generation [11], analytical inverse kinematics [12], a library of proven robust standard controllers, integrated state logging, and more. These components are implemented as CPython modules in C++ and can seamlessly be used in Python code. Table 1 compares the runtimes of common API calls between the Python bindings and native C++.

Motion generation can be accessed through the methods of the Panda class. The robot's neutral or starting pose can be reached with a single call to `panda.move_to_start()`. The workspace around this pose is characterized by high manipulability, reachability, and distance to joint limits. The class can be used to quickly generate point-to-point motions in joint-space

```
panda.move_to_start()
pose = panda.get_pose()
pose[2,3] -= .1
q = panda_py.ik(pose)
panda.move_to_joint_position(q)
```

The call to get_pose produces a 4×4 matrix representing the homogeneous transform from robot base to end-effector. The indices 2, 3 refer to the third row and fourth column, respectively, i.e., the z-coordinate. The position in z is lowered by 0.1 m and passed to the inverse kinematics function to produce joint positions. Finally, the call

[4] As of the time of this writing, the flange connector and libfranka gripper API of Franka Emika robots support only the official gripper known as the "Franka Hand".

to `panda.move_to_joint_position()` generates a motion from the current to the desired joint potions. Calls to the move functions are blocking by default until the goal is reached or a timeout or other error occurs.

Similarly, Cartesian motions can be executed directly by providing a Cartesian end-effector goal pose resulting in a straight line in Cartesian space

```
panda.move_to_start()
pose = panda.get_pose()
pose[2,3] -= .1
panda.move_to_pose(pose)
```

Note that the interface can also compute motions with multiple waypoints. The integrated trajectory motion generators generate time-optimal trajectories following piecewise linear paths between the given points within the robot's bounds on accelerations and velocities [11]. In the case of Cartesian motion, inverse kinematics are used to compute the corresponding joint-space path. Internally this joint-space trajectory will then be traced by a joint impedance controller. Additionally, the user can set various advanced parameters, such as control gains and allowed path deviation from via-points, when generating trajectory motion.

While the software features numerous shorthands and convenience methods,[5] users can always access the full breadth of the libfranka library. For instance, a call to

```
panda.get_state()
```

will retrieve the latest `libfranka.RobotState` received from the robot. Similarly, the Panda class also provides a reference to the libfranka Model associated with the running instance.

```
panda.get_model()
```

These Python wrappers offer the entire C++ API, i.e., class member variables and functions. The same is true for the previously initialized gripper. The gripper can be controlled using the class member functions grasp and move.

```
gripper.grasp(width, speed, force,
 ↪ epsilon_inner, epsilon_outer)
gripper.move(width, speed)
```

Function calls in panda-py allow users to use native Python types as arguments. More than that, the backend uses the powerful Eigen [13] library for linear algebra and will transparently and efficiently convert Eigen matrices to numpy [14] arrays without modifying the underlying memory structure.

Logging robot states is a ubiquitous requirement in robotics experiments, yet it can be challenging to set up, particularly when capturing states at high frequencies. However, panda-py offers a convenient solution with its integrated mechanism to write the whole state of the robot into a circular buffer at 1 kHz when activated. This feature simplifies the logging process, allowing users to easily capture and store data for subsequent evaluation, plotting, and other signal-processing tasks. Simply enable logging and store the resulting buffer (cf. Listing 1).

Listing 1: Using the integrated state logger.

```
from panda_py import constants

T_0 = panda_py.fk(constants.
 ↪ JOINT_POSITION_START)
T_0[1,3] = 0.25
T_1 = T_0.copy()
T_1[1,3] = -0.25

panda.move_to_pose(T_0)
panda.enable_logging(2000)
panda.move_to_pose(T_1)
panda.disable_logging()
log = panda.get_log()
```

Using the integrated logging mechanism, the libfranka RobotState can be logged at a frequency of 1 kHz. Based on the starting pose, Listing 1 creates two end-effector poses, `T_0` and `T_1`, displaced 0.25 m to the left and right, respectively. Logging is enabled for this Panda instance before a motion is generated between these poses. The `panda.enable_logging()` function takes the buffer size in the number of steps as an argument. As such, 2000 steps at 1 kHz correspond to a buffer holding the state of the past 2 s. After the motion is finished, logging is disabled, and the buffer is retrieved. Fig. 1 shows plots generated from the logged end-effector pose.

Finally, for more advanced applications, there is a library of standard controllers.[6] Controllers are classes instantiated by users and passed to the `Panda` class for execution. The controllers will run independently of the Python Global Interpreter Lock in the background and meet all of libfranka's real-time requirements. The user can provide input signals to the controller by calling `Controller.set_control` as demonstrated in Listing 2.

Listing 2: Using a real-time controller.

```
import numpy as np

panda.move_to_start()
ctrl = controllers.CartesianImpedance()
x0 = panda.get_position()
q0 = panda.get_orientation()
runtime = np.pi*4.0
panda.start_controller(ctrl)

with panda.create_context(frequency=1e3,
 ↪ max_runtime=runtime) as ctx:
  while ctx.ok():
    x_d = x0.copy()
    x_d[1] += 0.1*np.sin(ctrl.get_time())
    ctrl.set_control(x_d, q0)
```

After initializing the controller, the current position and orientation are stored in `x0` and `q0`, respectively, where `q0` is a quaternion representation of the end-effector orientation. After starting the controller, a `PandaContext` is created from the `Panda` object. `PandaContext` is a convenient context manager that executes a loop at a fixed frequency. The call to `PandaContext.ok()` throttles the loop and raises any control exceptions that may have been raised by libfranka. The use of `PandaContext` is optional, and users are free to manage the control flow how they wish. This example adds a sinusoidal linear displacement along the y-axis to the initial pose. This results in the end-effector moving sinusoidally from left to right in straight lines. The controller `CartesianImpedance` is instantiated and passed to the `Panda` class for execution. While the controller is running, the user can use `CartesianImpedance.set_control()` to provide an input signal. The panda-py controllers provide many configuration options specific to the individual controller, such as control gains or filter

[5] Shorthands such as `panda.get_pose()` or `panda.q` provide a convenient representation of fields provided by the RobotState struct in libfranka.

[6] While the included controllers cover many common applications, users may provide their own custom controllers in C++. Please refer to the official documentation at https://jeanelsner.github.io/panda-py for more details.

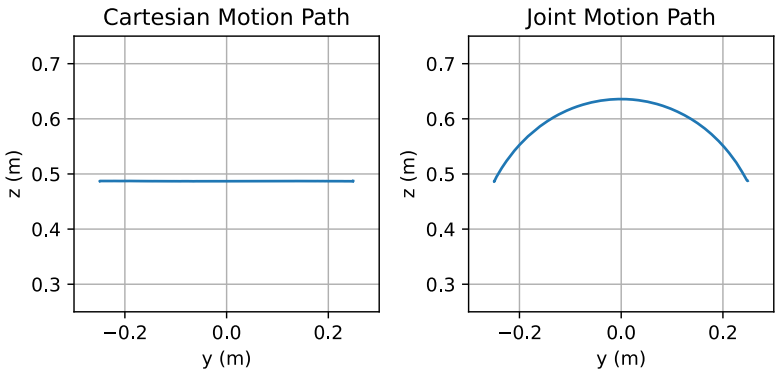


Fig. 1. Plotting the robot's internal state during motion recorded with the integrated logging mechanism. The plots show the traced path of the end-effector in the robot base frame projected onto the yz-plane. The motion is generated as seen in Listing 1, i.e., using two waypoints displaced 0.5 m along the y-axis. The motion of the left plot was generated using `panda.move_to_pose()` and resulted in a linear path in Cartesian space. In contrast, the motion of the right plot is the result of a call to `panda.move_to_joint_position()` with the same waypoints, resulting in a non-linear path in Cartesian space.

settings. These options may be changed at run-time as well. In this example, the input signal is provided in a high-frequency loop without filtering for more fine-grained control. All controllers include virtual joint walls, so the input signal need not explicitly consider the joint limits. While various methods exist to control aspects such as the robot's reflex behavior and error recovery, the content covered in this chapter already provides a foundational level of control over the robot.

4. Integration with Python packages

A notable aspect of panda-py is its ability to seamlessly integrate with popular Python packages, such as the Robotics Toolbox for Python [4] or MuJoCo [15]. Using Python bindings to integrate the robot hardware makes for singularly lightweight and straightforward integration. The middleware layer can be avoided entirely while all necessary hardware setup and preparation can be centralized.

By leveraging the capabilities of these existing packages, researchers can easily extend the functionalities of Franka Emika robots, as is demonstrated in this final example. Specifically, a resolved rate motion controller utilizing reactive control based on quadratic programming is integrated with panda-py, leveraging the Robotics Toolbox for Python. This integration exemplifies the ease of extending panda-py to incorporate more complex functionalities, such as reactive collision avoidance and mobile manipulation. Listing 3 provides a clear representation of the integrated implementation. Notably, the solution to the controller's quadratic program yields joint velocities that seamlessly interface with the `IntegratedVelocity` controller.

Listing 3: Integration with the Robotics Toolbox for Python.

```
import qpsolvers as qp
import roboticstoolbox as rtb
import spatialmath as sm

ctrl = controllers.IntegratedVelocity()
panda.move_to_start()
panda.start_controller(ctrl)

# Initialize roboticstoolbox model
panda_rtb = rtb.models.Panda()

# Set the desired end-effector pose
Tep = panda_rtb.fkine(panda.q) * sm.SE3
 ↪ (0.3, 0.2, 0.3)

# Number of joints in the panda which we
 ↪ are controlling
n = 7

arrived = False

# The original example runs in simulation
 ↪ with a control frequency of 20Hz
with panda.create_context(frequency=20) as
 ↪ ctx:
  while ctx.ok() and not arrived:

    # The pose of the Panda's end-effector
    Te = panda_rtb.fkine(panda.q)

    # Transform from the end-effector to
     ↪ desired pose
    eTep = Te.inv() * Tep

    # Calulate the required end-effector
     ↪ spatial velocity for the robot
    # to approach the goal. Gain is set to
     ↪ 1.0
    v, arrived = rtb.p_servo(Te, Tep, 1.0)

    # Gain term (lambda) for control
     ↪ minimisation
    Y = 0.01

    # Quadratic component of objective
     ↪ function
    Q = np.eye(n + 6)

    # Joint velocity component of Q
    Q[:n, :n] *= Y

    # Slack component of Q
    # The slack is inversely proportional
     ↪ to goal distance with the angular
     ↪ component weighted down
    e = np.sum(np.abs(np.r_[eTep.t, eTep.
     ↪ rpy() * np.pi / 180]))
    Q[n:, n:] = (1 / e) * np.eye(6)

    # The equality contraints
    Aeq = np.c_[panda_rtb.jacobe(panda.q),
     ↪ np.eye(6)]
    beq = v.reshape((6,))
```

```
    # Linear component of objective
     ↪ function: the manipulability
     ↪ Jacobian
    c = np.r_[ -panda_rtb.jacobm().reshape
     ↪ ((n,)), np.zeros(6)]

    # The lower and upper bounds on the
     ↪ joint velocity and slack variable
    lb = -np.r_[panda_rtb.qdlim[:n], 10 *
     ↪ np.ones(6)]
    ub = np.r_[panda_rtb.qdlim[:n], 10 * np
     ↪ .ones(6)]

    # Solve for the joint velocities dq
    qd = qp.solve_qp(Q, c, None, None, Aeq,
     ↪  beq, lb=lb, ub=ub, solver='daqp')

    # Apply the joint velocities to the
     ↪ Panda

    ctrl.set_control(qd[:n])
```

This example is from the Robotics Toolbox for Python [16]. To run it on the real hardware with panda-py requires only connecting the inputs and outputs of the control loop to panda-py, i.e., using the joint positions `panda.q` and providing the control signal to `IntegrateVelocity.set_control()`. Additionally, the inequality constraints to avoid the joint limits were removed, as panda-py controllers already have integrated joint limit avoidance using impedance control.

5. Impact

By simplifying the programming, deployment, and installation processes, panda-py enhances the accessibility and efficiency of working with Franka Emika robot systems. This impact extends beyond the research community, as it addresses the need for user-friendly robotics solutions in educational settings. The availability of panda-py not only empowers researchers to explore advanced robotics concepts but also opens doors for students to engage in hands-on learning experiences. In our previous work [17], we used panda-py to achieve real-time hardware-in-the-loop integration between complex robotic systems and the MuJoCo physics engine for haptic interaction and predictive modeling. This underscores panda-py's utility in complex scenarios, where its seamless integration with Python's extensive ecosystem facilitates interdisciplinary collaborations.

6. Conclusion

In conclusion, this software paper has introduced panda-py as a Python interface and framework that facilitates the programming of Franka Emika robots. The inclusion of concise and approachable code examples throughout the paper highlights panda-py's user-friendly nature and effectiveness in controlling the robots. It is worth noting that this paper provides only a glimpse into the extensive capabilities of panda-py, as it represents a dynamic and evolving ecosystem. Online resources, including additional examples, documentation, and the help of the robotics community, contribute to the continual maintenance and expansion of panda-py. Researchers and users are encouraged to explore these resources for a comprehensive understanding of panda-py's potential. For future work, the author aims to integrate the software with ROS2 to facilitate the use of motion planning and other high-level features provided by the ecosystem. Additionally, integrating panda-py with reinforcement learning environments opens up exciting opportunities to explore robot learning.

Funding

The author gratefully acknowledges the funding support provided by the Lighthouse Initiative Geriatronics by StMWi Bayern (Project X, grant no. IUK-1807-0007 / IUK582/001).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al. TensorFlow, large-scale machine learning on heterogeneous systems. 2015, http://dx.doi.org/10.5281/zenodo.4724125.

[2] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. PyTorch: An imperative style, high-performance deep learning library. In: Wallach H, Larochelle H, Beygelzimer A, d'Alché Buc F, Fox E, Garnett R, editors. Advances in neural information processing systems, vol. 32. Curran Associates, Inc.; 2019, p. 8024–35.

[3] Buitinck L, Louppe G, Blondel M, Pedregosa F, Mueller A, Grisel O, et al. API design for machine learning software: Experiences from the scikit-learn project. In: ECML PKDD workshop: languages for data mining and machine learning. 2013, p. 108–22.

[4] Corke P, Haviland J. Not your grandmother's toolbox–the Robotics Toolbox reinvented for Python. In: 2021 IEEE international conference on robotics and automation. IEEE; 2021, p. 11357–63.

[5] Abrahams D, Grosse-Kunstleve RW. Building hybrid systems with Boost. Python. C/C++ Users J 2003;21(LBNL-53142).

[6] Jakob W, Rhinelander J, Moldovan D. pybind11–seamless operability between C++ 11 and Python. 2017, URL https://github.com/pybind/pybind11.

[7] Haddadin S, Parusel S, Johannsmeier L, Golz S, Gabl S, Walch F, et al. The Franka Emika Robot: A reference platform for robotics research and education. IEEE Robot Autom Mag 2022;29(2):46–64. http://dx.doi.org/10.1109/MRA.2021.3138382.

[8] Haddadin S, Johannsmeier L, Schmid J, Ende T, Parusel S, Haddadin S, et al. Roboterfabrik: A pilot to link and unify german robotics education to match industrial and societal demands. In: Lepuschitz W, Merdan M, Koppensteiner G, Balogh R, Obdržálek D, editors. Robotics in education. Cham: Springer International Publishing; 2019, p. 3–17.

[9] frankx: High-Level motion library for the Franka Emika Robot. 2022, URL https://github.com/pantor/frankx.

[10] Zhang K, Sharma M, Liang J, Kroemer O. A modular robotic arm control stack for research: Franka-interface and frankapy. 2020, arXiv preprint arXiv:2011.02398.

[11] Kunz T, Stilman M. Time-optimal trajectory generation for path following with bounded acceleration and velocity. In: Robotics: science and systems. 2012, p. 09–13.

[12] He Y, Liu S. Analytical inverse kinematics for Franka Emika Panda – A geometrical solver for 7-DOF manipulators with unconventional design. In: 2021 9th international conference on control, mechatronics and automation. IEEE; 2021, http://dx.doi.org/10.1109/ICCMA54375.2021.9646185.

[13] Guennebaud G, Jacob B, et al. Eigen v3. 2010, http://eigen.tuxfamily.org.

[14] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature 2020;585(7825):357–62. http://dx.doi.org/10.1038/s41586-020-2649-2.

[15] Todorov E, Erez T, Tassa Y. Mujoco: A physics engine for model-based control. In: 2012 IEEE/RSJ international conference on intelligent robots and systems. IEEE; 2012, p. 5026–33. http://dx.doi.org/10.1109/IROS.2012.6386109.

[16] Haviland J, Corke P. Maximising manipulability during resolved-rate motion control. 2020, CoRR abs/2002.11901, arXiv:2002.11901.

[17] Elsner J, Reinerth G, Figueredo L, Naceri A, Walter U, Haddadin S. PARTI-A haptic virtual reality control station for model-mediated robotic applications. Front Virtual Real 2022;3. http://dx.doi.org/10.3389/frvir.2022.925794, URL https://www.frontiersin.org/articles/10.3389/frvir.2022.925794.