Python Interface of PicoScenes

Zhiping Jiang About 1281 words About 4 min

PyPicoScenes

PyPicoScenes is a Python binding library for PicoScenes, a C++-based Integrated Sensing and Communication (ISAC) research framework PicoScenes. By leveraging Cppyy's dynamic binding technology, PyPicoScenes seamlessly encapsulates the underlying C++ APIs, providing researchers with a Python programming interface that combines high performance with development efficiency. This Python version fully inherits PicoScenes' original hardware compatibility and algorithmic innovations, while deeply integrating with the Python ecosystem. This significantly lowers the development barriers for Wi-Fi sensing and communication synergy research.

Installation

PyPicoScenes relies on Python's cppyy library along with PicoScenes header files and dynamic libraries. Currently supported platforms include Ubuntu, macOS, and Windows.

Installing PyPicoScenes on Ubuntu

1. Obtain PyPicoScenes
   PyPicoScenes can only be obtained by `git clone` from it's git repo.
2. PicoScenes Installation
   Refer to the PicoScenes installation guide here.
3. Anaconda Installation
   Refer to the Anaconda installation guide here.
4. Update Anaconda Environment
   Activate your conda environment using conda activate ENV_NAME. If the libstdc++ dynamic library in Conda is outdated, run:

```
conda install -c conda-forge libstdcxx-ng=13 -y
```

5. Install cppyy && Dependencies
   cppyy is a Cling/LLVM-based dynamic binding tool that enables seamless Python-C++ interaction

through runtime C++ code parsing. Its key advantages include no precompiled bindings, support for C++98 to C++20 standards, and compatibility with both PyPy and CPython interpreters. We recommend installing cppyy and its dependencies via Anaconda:

```
conda create -n ENV_NAME python=3.10
conda activate ENV_NAME
pip install -r requirements
```

6. Verify Installation
   Navigate to the PyPicoScenes directory and run `python parse_frame.py` . Successful cppyy installation will output:

```
<cppyy.gbl.std.optional<ModularPicoScenesRxFrame> object at 0xeb54fa0>
```

Installing PyPicoScenes on Windows

1. Obtain PyPicoScenes
   PyPicoScenes can only be `git clone` from it's git repo.
2. PicoScenes Installation
   Refer to the PicoScenes installation guide here.
3. Install MSVC Build Tools
   Both VS2019 and VS2022 can compile cppyy. For VS2022 users, additional installations are required:
   - Windows 10 SDK
   - MSVC v142 - VS2019 C++ x64/x86 build tools
4. Install cppyy
   Use venv to avoid polluting system directories and enable full cleanup by simply deleting the virtual environment directory (e.g., "WORK" in this example). Open Visual Studio `x64 Native Tools Command Prompt` , and for VS2022 users, specify the `VS2019 v142` toolchain:

```
    # Set build environment to VS2019
    # Set VCToolsInstallDir to your actual Visual Studio installation directory.
    set VCToolsInstallDir=C:\Program Files\Microsoft Visual
Studio\2019\Community\VC\Tools\MSVC\14.29.30133\
    set PATH=%VCToolsInstallDir%\bin\Hostx64\x64;%PATH%

    # Create Python virtual environment
    python -m venv WORK
    WORK\Scripts\activate

    # Install cppyy v3.5.0
    python -m pip install cppyy-cling==6.32.8 --no-deps --no-build-isolation --force-
reinstall
    python -m pip install cppyy-backend==1.15.3 --no-deps --no-build-isolation --force-
reinstall
    python -m pip install CPyCppyy==1.13.0 --no-deps --no-build-isolation --force-reinstall
    python -m pip install cppyy==3.5.0 --no-deps --no-build-isolation --force-reinstall
```

5. Verify Installation
   Execute WORK\Scripts\activate to activate the Python environment created in Step 3. Navigate to the PyPicoScenes directory and run `python parse_frame.py` . Successful cppyy installation will output:

```
    <cppyy.gbl.std.optional<ModularPicoScenesRxFrame> object at 0xeb54fa0>
```

Cppyy Wrapping for PicoScenes

Cppyy, built upon the Cling interpreter, is a dynamic runtime Python-C++ bidirectional binding tool that generates efficient interfaces through real-time parsing of C++ code, enabling deep interoperability between the two languages. Its core value lies in zero manual wrapping, high performance, low memory overhead, and support for complex scenarios like cross-language inheritance, template instantiation, and exception mapping. It significantly simplifies the process of calling C++ libraries from Python, making it particularly suitable for large-scale projects and interactive development.

```
    """
    The following example uses STL to demonstrate how to use cppyy for interaction between
Python and C++.
    """
    # Using C++ vector
    import cppyy
    cppyy.include("vector")
    # C++ symbols reside in cppyy.gbl namespace, access via cppyy.gbl
    vec = cppyy.gbl.std.vector[int](3, 1)
    print(vec)  # Outputs { 1, 1, 1 }
    # Using size() method of vector<int> in Python
    print(vec.size())
    # Using push_back() method of vector<int> in Python
    vec.push_back(5)
    print(vec)  # Outputs { 1, 1, 1, 5 }
    print(vec.size())
```

The following explains how to use cppyy to wrap PicoScenes.

Adding PicoScenes to cppyy's Path

Assuming the absolute installation path of PicoScenes is **your_picoscenes_path**, first add the header files and dynamic libraries to cppyy's path:

```python
import cppyy
import cppyy.ll
# Add header file path
cppyy.add_include_path("your_picoscenes_path/include")
# Add dynamic library path
cppyy.add_library_path("your_picoscenes_path/lib")
```

Importing C++ Header Files

Use `cppyy.include` to import required header files:

```python
cppyy.include("PicoScenes/SystemTools.hxx")
cppyy.include("PicoScenes/QCA9300FrontEnd.hxx")
cppyy.include("PicoScenes/IntelRateNFlag.hxx")
cppyy.include("PicoScenes/AbstractSDRFrontEnd.hxx")
cppyy.include("PicoScenes/USRPFrontEnd.hxx")
# And other required header files
```

Loading Dynamic Libraries

Use `cppyy.load_library` to load required dynamic libraries:

```python
cppyy.load_library("libDSP")
cppyy.load_library("libFrontEnd")
cppyy.load_library("libIntrinsics")
cppyy.load_library("libLicense")
cppyy.load_library("libmac80211Injection")
cppyy.load_library("libNICHAL")
cppyy.load_library("librxs_parsing")
# And other required dynamic libraries
```

Using C++ APIs

After importing header files and dynamic libraries, all C++ symbols reside in the cppyy.gbl namespace and can be accessed via Python. For example, to retrieve a USRP NIC:

```
    nicName = "usrp"
    nic = cppyy.gbl.NICPortal.getInstance().getNIC(nicName)
    print(nic)
    # nic is an AbstractNIC object, e.g.:
    # nic: <cppyy.gbl.AbstractNIC object at 0x561264e43b28 held by
std::shared_ptr<AbstractNIC> at 0x561264e25c40>

    # Start the NIC's RxService
    nic.startRxService()
    # Stop the NIC's RxService
    nic.stopRxService()
```

Quick Start

PyPicoScenes encapsulates the core APIs of the underlying PicoScenes framework, enabling developers to implement WiFi packet transceiving and CSI measurement through Python interfaces without implementing complex C++ plugins. To utilize the transceiver functionalities of NICs (Network Interface Cards) or USRP SDR devices, follow these steps:

Workflow Overview

1. Platform Initialization
   Execute `picoscenes_start()` to launch the PicoScenes runtime environment.
2. Hardware Acquisition
   Acquire the target hardware device via `getNic(nicName="SDR/NIC")` .
3. Hardware Configuration
   Configure RF front-end parameters (e.g., sampling rate, bandwidth, center frequency) via Python APIs.
4. Tx parameters set up(optional)
   When implementing frame transmission functionality, the tx parameters (e.g., packet format, MCS, STS) must be configured via Python APIs.

5. Functional Implementation
   Activate the NIC's transceiver services and execute data transmission/reception operations using the hardware-specific low-level APIs.
6. Registering Python Callbacks(optional)
   PyPicoScenes allows registering Python callback functions to process received WiFi packets. It is particularly important to note that the first formal parameter of the registered callback function must represent the WiFi packet, while all subsequent formal parameters must have default values specified. For example:

```
    def call_back(frame, arg1=1, arg2=2, arg3=3,...)
```

7. Runtime Control
   Call `picoscenes_wait` to block the main thread and maintain platform execution.
8. Platform Termination
   Deactivate the NIC's transceiver services and invoke `picoscenes_stop()` to shut down the platform (note that `picoscenes_wait()` remains in a blocking state prior to this invocation).

Key Functionalities

- CSI File Parsing
- CSI Measurement with SDR device
- WiFi Packet Transmission via SDR device
- CSI Measurement with Commercial NIC
- WiFi Packet Transmission via Commercial NIC

Important Notes

PyPicoScenes leverages `cppyy` 's dynamic binding technology to efficiently encapsulate PicoScenes' C++ APIs. Developers can directly invoke low-level APIs in Python scripts by including the relevant header files (e.g., `include("PicoScenes/SystemTools.hxx")`) and loading dynamic libraries (e.g., `load_library("libSystemTools")`), enabling core functionalities like `wireless signal transmission/reception` and `CSI file parsing` . The Python APIs are `identical` to their native C++ counterparts, with usage details documented in the PicoScenes Native API Reference. Powered by cppyy's real-time parsing mechanism, Python can directly manipulate hardware control logic (e.g., configuring USRP sampling rates or WiFi channel parameters) while maintaining strict behavioral consistency with the C++ implementation. Developers must validate dynamic library paths and environmental dependencies during cross-platform deployments.