人工知能学会第二種研究会資料
AIチャレンジ研究会
SIG-Challenge-061-04

PyHARK: HARKのオンライン・オフライン処理用 Pythonパッケージ

PyHARK: HARK Python package supporting online and offline processing

中臺 一博 [1*]　瀧ヶ平 将行 [2]　糸山 克寿 [1,2]
Kazuhiro NAKADAI[1]　Masayuki TAKIGAHIRA[1]　Katsutoshi ITOYAMA[2]

[1] 東京工業大学 工学院システム制御系
[1] Dept. Sys. & Contr. Eng., School of Engineering, Tokyo Institute of Technology
[2] （株）ホンダ・リサーチ・インスティチュート・ジャパン
[2] Honda Research Institute Japan Co., Ltd.

Abstract: 本稿では，ロボット聴覚オープンソースソフトウェア HARK 3.4 で新規に導入される PyHARK を HARK 講習会に先立ち紹介する。PyHARK は HARK の Python インタフェースを提供するパッケージであり，Python から HARK の機能のオンライン・オフライン呼び出しを可能にする実装である。そのアーキテクチャ，既存の HARK との違い，使い方を中心に解説する。

1　はじめに

筆者らは 2008 年のリリース以降，ロボット聴覚オープンソースソフトウェア HARK [1, 2, 3, 4, 5, 6, 7, 8, 9][1] の研究開発を続けている [10, 11, 12, 13, 14, 15]．本稿では， 2022 年 11 月 23 日にリリースを行う HARK の最新版 3.4 の新機能である PyHARK を紹介する．端的に言えば，PyHARK は，HARK の Python パッケージである．これまでにも HARK の開発環境上で Python プログラムとの連携を実現する HARK-Python が存在していたが，あくまでも HARK 上で Python プログラムを呼び出すための仕組みであった．また，HARK のプログラム開発では，HARK 独自のプログラミング環境 HARKDesigner を利用する必要があった．HARKDesigner は GUI プログラミング環境であるため，初心者には直感的でわかりやすいものの，Node.js[2] ベースであり，ブラウザを立ち上げて作業しなければならないため，プログラミングに慣れた人にとっては，必ずしも効率的な開発環境ではなかった．PyHARK では，HARK を Python プログラムから直接呼び出すことができるため，Python 開発によく利用される Jupiter Notebook[3] や Visual Studio Code[4] を使い，効率的なプログラミングが可能である．また，HARK の特長である逐次処理実行機能をそのまま継承しているので，Python で，センサやファイルからデータを逐次的に取得し処理を行うプログラムを作成することができる．ファイルをまとめて読み込んで処理をするオフライン・バッチ処理も実装可能である．

*東京工業大学 工学院システム制御系
〒 152-8552 東京都目黒区大岡山 2-12-1
E-mail: nakadai@ra.sc.e.titech.ac.jp

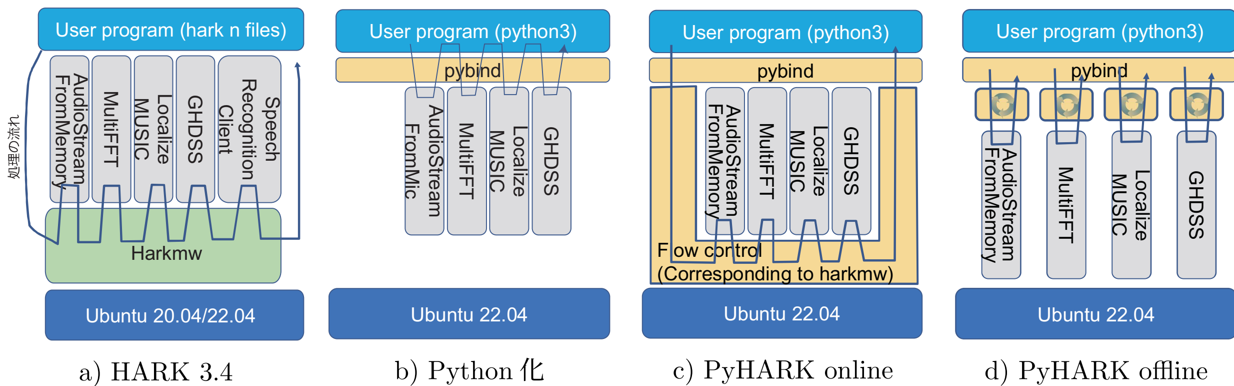
[1] https://hark.jp/
[2] https://nodejs.org/ja/
[3] https://jupyter.org/
[4] https://azure.microsoft.com/ja-jp/products/visual-studio-code/

2　HARK とその Python 化における課題

本節では，HARK から PyHARK に至るアーキテクチャを紹介し，PyHARK 化における課題，およびその解決法について議論する．

2.1　HARK のアーキテクチャ

図 1a) に従来版の HARK の代表として，HARK 3.4 のソフトウェアスタックを示す．従来版の HARK では，OS レイヤ (Ubuntu OS) の上に HARK の機能ノードの制御を司るミドルウェア harkmw [16] (緑色のボックス）が載っている．その上のレイヤは，harkmw を通じて動作する HARK の機能ノード（灰色のボックス群）で構成されている．最上位のレイヤはユーザプログラムレイヤ（水色のボックス）であり，このレイヤには，ユーザが，HARK の機能ノードを自由に配置し，それらの関係を記述することで作成した HARK のプログラムが置かれる．この HARK のプログラムは XML で記述されており，拡張子が “n” であるため n ファイルと呼ばれる．XML を直接手作業で記述することは現実的ではないため，HARKDesigner と呼ばれる独自の GUI プログラミング環境を用いて n ファイルを作成することで，プログラミングを行う．プログ



a) HARK 3.4　b) Python 化　c) PyHARK online　d) PyHARK offline

図 1: HARK のソフトウェアスタック

ラム実行時は n ファイルを harkmw に引数として与えて実行する．すると，harkmw は，n ファイルを読込み，XML の Parsing を行い，n ファイルの記述に沿って，フロー制御を行う．つまり，HARK の機能ノードは ユーザプログラムから直接呼び出されるのではなく，図 1 の矢印で示されているように harkmw から pybind11[5] を通じて，C/C++レベル の関数コールにより実行されることになる．このような仕組みを構築することにより，以下の 2 つの利点が得られる．

- ノード間統合のオーバーヘッドの低減：C/C++レベル の関数コールで機能ノード間が接続されるため，統合のオーバヘッドが小さく，10 ms オーダの逐次処理が求められる音響信号処理でも十分な性能を発揮できる．
- 逐次処理記述の簡素化：ユーザが意識しなくても harkmw が裏で逐次処理を行うため，ユーザは逐次処理の 1 サイクル分の処理を記述するだけでよい[6]．この仕組みは，逐次処理が必要な時系列信号の処理全般に有効といえるので，機能ノードさえあれば，音響信号処理に限定せず利用することが可能である．

一方で，n ファイルを作成するためには独自の GUI 環境をわざわざ立ち上げてプログラミングしなければならないため，ある程度プログラミングに熟練したユーザには，プログラミングの作業効率が悪いという課題があった．

2.2　PyHARK に向けた課題

HARK の問題を解決するもっとも簡単な方法は，熟練プログラマでもストレスなく利用できるプログラミング言語・環境で HARK を利用できるようにすることである．近年は，信号処理や機械学習では，Python が一般的に利用されるようになってきている．また，Jupiter notebook や Visual Studio Code など Python をサポートし，かつインタラクティブにプログラミングやデバッグができる高機能なプログラミング環境が登場している．そこで，HARK を Python から利用できるよう HARK の Python パッケージ化を図ったのが PyHARK である．PyHARK をもっとも簡単に実現するには，図 1b) に示すように，HARK の機能ノードを，pybind11 でラップし，ユーザには，Python の関数として見えるようにすることである．このようにすれば．Python プログラム内で，”import hark” を記述することで，Python 内で HARK の機能が利用できるようになる．これだけ考えると，もはや，harkmw のようなミドルウェアは不要に見えるが，実際には，このままでは，以下の 2 つの問題が発生してしまう．

- ノード間統合のオーバーヘッドの増大
- オフライン・バッチ的な使い方は実装が大変

一つ目の問題は，harkmw がない場合，機能ノード間に直接のつながりはないため，機能ノード間のデータの受け渡しをユーザプログラム（ Python）レイヤで行わざるを得ないことに起因している．これは，音響信号のような時系列信号を扱う場合は特に大きな問題となる．音響信号処理では，一般的に，一連のデータをある時間単位で分割（この単位を一般にフレームという，音響信号の場合は 10〜50 ms とすることが多い）し，フレーム単位で逐次的に処理を行う．例えば，図 1b) のように，AudioStreamFromMic, MultiFFT, Localize-MUSIC, GHDSS という機能ノードを順番に実行する処理を考える．あるフレームのデータをある機能ノードから次の機能ノードに渡す際（例えば，MultiFFT から LocalizeMUSIC に渡す場合）に，そのデータをいちいちユーザプログラムレイヤまで引き上げてから機能ノードレイヤに戻すというレイヤをまたいだ通信が必要になる．このレイヤをまたぐ通信が一度行われるたびに，データのシリアライズ・デシリアライズが発生する．図 1b) の矢印に示すように，このレイヤをまたぎの通信は AudioStreamFromMic から GHDSS まで

[5] https://github.com/pybind/pybind11
[6] Perl で暗黙のループを実現する ”-n” スイッチと似たイメージ

の一連の処理で機能ノードにデータが渡されるたびに実行されることになる．さらに，この一連の処理がすべてのフレームについて行われるため，必然的にオーバヘッドが大きくなる．

二つ目の問題は，機能ノードが，もともとフレーム単位で処理を行うように作られていることに由来する．従来の HARK では，これはフレーム単位の逐次処理を可能とする利点であったが，図 1b) に示すような構成をとった場合，従来の HARK では，harkmw が隠ぺいしてくれていた逐次処理に相当する，入力信号をフレーム単位に分割し，1 フレームごとに機能ブロックを呼び出すコードをユーザ側で用意しなければならなくなってしまう．このため，従来の HARK に比べて使い勝手が悪くなってしまう．

3 PyHARK アーキテクチャ

PyHARK では, 逐次処理, オフライン・バッチ処理の二種類に処理を分けて考えることで HARK の Python 化における 2 つの課題の解決を図っている．また，詳細は割愛するが，HARK の C++ 部分の実装を見直し，マルチスレッド化，高速な行列演算を提供するライブラリである Eigen を導入によることによって，オリジナルの OSS 版と比較して，高速化を行っている．

3.1 逐次版 PyHARK

図 1c) に逐次版の PyHARK の構成図を示す．逐次版では，従来の HARK の利点である逐次処理を継承できるよう，機能ノードのフロー制御を導入し，ユーザの Python プログラムから呼び出しができるようになっている．フロー制御部分は，基本的には，harkmw から n ファイルの読込み，Parsing 機能を取り除いたものに相当している．また，従来版では，harkmw は実行ファイルとなっていたが，PyHARK ではユーザのプログラム上で，Publisher, Subscriber を呼び出すことで実現している．このような違いはあるものの，端的に言えば，これまでの n ファイルに相当する部分を python プログラムとして記述できるようにしたパッケージと捉えることができる．図 1c) の矢印で示すように実行時のフロー制御は，フロー制御部にまかせて，内部的には従来版 HARK と同様，C/C++レベルの関数コールでの機能ノード接続となっているため，オーバヘッドは従来版と同様に低く抑えることができる．もちろん，Python でコーディングした自前の機能ノードを用いる場合や機能ノードの途中経過を probe したい時などは pybind11 を経由した機能ノードへのアクセスも可能であるが，この場合は，シリアライズ・デシリアライズに伴う通信のオーバヘッドが発生する．

3.2 オフライン・バッチ処理版 PyHARK

図 1d) にオフライン・バッチ処理版の PyHARK の構成図を示す．逐次版では，逐次処理が可能な代わりに，n ファイルに相当する機能ノードや機能ノード間の接続ネットワークの定義を Python プログラムとして記載する必要があった．HARKDesigner を使わずに記述できる反面，これまで XML として記述していたものと同様の内容をプログラム化しなければならないので煩雑である．また，実際に Python でプログラミングする場合, 大抵はファイルもしくはファイルセット単位で順番に処理を行うオフライン・バッチ処理として記述することが多い．オフライン・バッチ処理を念頭に置いている場合にも逐次処理 のプログラムを書かなければならいのでは使い勝手が悪い．深層学習のパッケージとしてよく用いられる Keras では，すべてのデータを一度に読込んで学習できる場合は，fit() を使って簡単に学習コードを記述できる．しかし，データ量が多くなり，メモリ上に一度に読み込めない場合は，逐次的にデータを読みながら学習を実行する fit_generator() を別途実装する必要があり，実装が煩雑になる[7]．誤解を招くことを恐れずに書けば，PyHark の逐次版は，常に fit_generator() に相当する実装を強いられるのと近いイメージといえる．そこで，PyHARK では，逐次処理を一切意識せず，オフライン・バッチ処理的に Python のプログラミング作成することも可能である．これを実現するため，HARK の Python 化における課題の 2 つ目として挙げている機能ノードが，もともとフレーム単位で処理を行うように作られているということをユーザが意識せずに済むようにする作りになっている．具体的には，pybind11 経由で機能ノードを呼び出す際に，フレームごとに回して処理をする部分を自動的に行うような実装を行っている．当然，シリアライズ・デシリアライズは発生するものの，そもそもオフライン・バッチ処理用の仕組みであるため，通信のオーバヘッドが問題にはならない場合の構成である（問題になる場合は逐次版を使えばよい）．

4 PyHARK を用いた実装例

PyHARK を用いて音源定位を行うプログラムの例を Listing 1（オンライン版）と 2（オフライン版）に示す．これらは，短時間フーリエ変換，MUSIC 法による音源定位，音源追跡処理を行うプログラムで，実装的には，主に，HARK の MultiFFT, LocalizeMUSIC, SourceTracker ノードを用いている．

Listing 1: pyhark-online-sample.py

```
1 #! /usr/bin/env python
2 # -*- coding: utf-8 -*-
```

[7]現在は fit_generator() は fit() に統合されている

```
3
4   import sys
5   import threading
6   import time
7   import numpy
8   import soundfile
9   import hark
10  import plotQuickSourceKivy
11
12  # マイク数等設定
13  nch    = 8
14  winlen = 512
15  advance = 160
16
17  # ネットワークの定義HARK(Localize)
18  class HARK_Localize(hark.NetworkDef):
19      def build(self,
20                network: hark.Network,
21                input:   hark.DataSourceMap,
22                output:  hark.DataSinkMap):
23
24          # 機能ノードの生成
25          node_audio_stream_from_memory = network.create(hark.node.AudioStreamFromMemory)
26          node_multi_fft = network.create(hark.node.MultiFFT)
27          node_localize_music = network.create(hark.node.LocalizeMUSIC)
28          node_cm_identity_matrix = network.create(hark.node.CMIdentityMatrix, dispatch=hark.RepeatDispatcher)
29          node_constant = network.create(hark.node.Constant, dispatch=hark.RepeatDispatcher)
30          node_source_tracker = network.create(hark.node.SourceTracker)
31          node_plotsource_kivy = network.create(plotQuickSourceKivy.plotQuickSourceKivy)
32
33          try:
34              # 機能ノードのプロパティ設定とノード間接続
35              r = [
36                  node_audio_stream_from_memory
37                      .add_input("INPUT", input["INPUT"])
38                      .add_input("CHANNEL_COUNT", nch)
39                  ,
40                  node_multi_fft
41                      .add_input("INPUT", node_audio_stream_from_memory["AUDIO"])
42                  ,
43                  node_cm_identity_matrix
44                      .add_input("NB_CHANNELS", nch)
45                      .add_input("LENGTH", winlen)
46                  ,
47                  node_constant
48                      .add_input("VALUE", True)
49                  ,
50                  node_localize_music
51                      .add_input("INPUT", node_multi_fft["OUTPUT"])
52                      .add_input("NOISECM", node_cm_identity_matrix["OUTPUT"])
53                      .add_input("OPERATION_FLAG", node_constant["OUTPUT"])
54                      .add_input("A_MATRIX", "tf.zip")
55                      .add_input("MUSIC_ALGORITHM", "SEVD")
56                      .add_input("WINDOW_TYPE", "PAST")
57                      .add_input("LOWER_BOUND_FREQUENCY", 500)
58                      .add_input("UPPER_BOUND_FREQUENCY", 2800)
59                      .add_input("WINDOW", 50)
60                      .add_input("PERIOD", 1)
61                      .add_input("NUM_SOURCE", 2)
62                  ,
63                  node_source_tracker
64                      .add_input("INPUT", node_localize_music["OUTPUT"])
65                      .add_input("THRESH", 25)
66                      .add_input("PAUSE_LENGTH", 1200.0)
67                      .add_input("MIN_SRC_INTERVAL", 20.0)
68                      .add_input("MIN_ID", 0)
69                  ,
70                  node_plotsource_kivy
71                      .add_input("SOURCES", node_source_tracker["OUTPUT"])
72                  ,
73              ]
74
75              # 出力設定
76              output.add_input("OUTPUT", node_source_tracker["OUTPUT"])
77
78          except BaseException as ex:
79              print('error: {}'.format(ex))
80
81          return r
82
83  # ネットワークの定義HARK(ループMAIN)
84  class HARK_Main(hark.NetworkDef):
85      def __init__(self):
86          hark.NetworkDef.__init__(self)
87
88      def build(self,
89                network: hark.Network,
90                input:   hark.DataSourceMap,
91                output:  hark.DataSinkMap):
92
93          try:
94              # フロー制御用
95              node_publisher = network.create(hark.node.PublishData, dispatch=hark.RepeatDispatcher, name="Publisher")
96              node_subscriber = network.create(hark.node.SubscribeData, name="Subscriber")
97              # フレーム毎音源定位処理
98              loop = network.create(HARK_Localize, name="HARK_Localize")
99          except BaseException as ex:
100             print(ex)
101
102         # フロー制御との接続
103         r = [
104             loop.add_input("INPUT", node_publisher["OUTPUT"]),
105             node_subscriber.add_input("INPUT", loop["OUTPUT"])
106         ]
107
108         return r
109
110 # 結果取得用
111 def received(data):
112     print('>>>> received: {}'.format(data))
113
114 def main(args=sys.argv[1:]):
115
116     # ネットワーク読込みHARK
117     network = hark.Network.from_networkdef(HARK_Main, name="HARK_Main1")
118
119     # フロー制御用
120     publisher = network.query_nodedef("Publisher")
121     subscriber = network.query_nodedef("Subscriber")
122
123     # 結果取得用
124     subscriber.receive = received
125
126     # 読込んだネットワークの実行
127     try:
128         def target():
129             network.execute()
130
131         th = threading.Thread(target=network.execute)
132         th.start()
133     except BaseException as ex:
134         print(ex)
135
136     # 音響信号読込み (8 ch, 16bit integer)
137     audio, rate = soundfile.read('input.wav', dtype=numpy.int16)
138
139     # シフト長 advance(160) でフレーム化
140     frames = numpy.lib.stride_tricks.sliding_window_view(audio.T, (nch, advance))[0, ::advance]
141
142     # フレーム毎実行
143     try:
144         for t in range(frames.shape[0]):
```

```
145                 if not th.is_alive():
146                     break
147                 print('<<<<␣send:␣count={}'.format(t))
148                 publisher.push(frames[t,:,:])
149                 time.sleep(0.01)
150         finally:
151             publisher.close()
152             network.stop()
153             th.join()
154 
155 if __name__ == '__main__':
156     main(sys.argv[1:])
```

Listing 2: pyhark-offline-sample.py

```
1  #! /usr/bin/env python
2  # -*- coding: utf-8 -*-
3  
4  import hark
5  import numpy
6  
7  # マイク数等設定
8  nch    = 8
9  winlen = 512
10 advance = 160
11 
12 # 音響信号読込み (8 ch, 32 bit float)
13 audio, rate = soundfile.read('input.wav', dtype=
       numpy.float32)
14 
15 # フレーム化フレーム長( 512 シフト長 160)
16 frames = numpy.lib.stride_tricks.
       sliding_window_view(audio, winlen, axis=0)[::
       advance, :, :]
17 
18 multi_fft = hark.node.MultiFFT()
19 multifft_out = multi_fft(INPUT=frames)
20 
21 noisecm = numpy.eye(nch, dtype=numpy.complex64).
       flatten()
22 noisecm_bins = numpy.broadcast_to(noisecm, (
       multifft_out.OUTPUT.shape[0], multifft_out.OUTPUT
       .shape[1], nch*nch))
23 localizemusic_node = hark.node.LocalizeMUSIC()
24 localizemusic_out = localizemusic_node(INPUT=
       multifft_out.OUTPUT, NOISECM=noisecm_bins,
       A_MATRIX='tf.zip', MUSIC_ALGORITHM='SEVD', PERIOD
       =1, WINDOW_TYPE="PAST", WINDOW=50, NUM_SOURCE=2,
       LOWER_BOUND_FREQUENCY=500, UPPER_BOUND_FREQUENCY
       =2800,ENABLE_OUTPUT_RXXN=True)
25 
26 sourcetracker_node = hark.node.SourceTracker()
27 sourcetracker_out = sourcetracker_node(INPUT=
       localizemusic_out.OUTPUT, THRESH=25.0,
       PAUSE_LENGTH=1200.0,MIN_SRC_INTERVAL=20.0,MIN_ID
       =0)
28 
29 print(sourcetracker_out.OUTPUT)
```

Listing 1 では，n ファイルに相当するネットワークの定義を行っているのが, HARK_Main クラス (83–108 行目)，HARK_Localize クラス (17–81 行目) である．HARK_Main クラスでフロー制御を含めたネットワークの枠組みを用意し， 1 フレーム分の音源定位処理を記述した HARK_Localize クラスがその中で展開される形になっている．これらのクラスの中では, 用いる機能ノードの宣言, 各機能ノードのプロパティ設定, 機能ノード間の接続設定が記述されている．HARK_Main では，これに加え，フロー制御とのインタフェース用に，Publisher, Subscriber に関する記述を行っている．114–153 行目の main 関数では，上記のクラスとして定義されているネットワーク定義を読込み，スレッドとしてこれを実行する．その後，フレームごとのデータを publisher を通じて push している (148 行目)．PyHARK 内部では，決められたフレーム長を単位として，逐次処理が実行される（何も指定しなければデフォルト値として 512 サンプルがフレーム長として用いられる)．このプログラムでは，137 行目で読み込んだ音響データを 140 行目でフレームシフト長である 160 サンプルごとにフレーム化を行っているが，実センサでは必ずしもフレームシフト長分のデータが毎回きっちり得られるわけではない．実際には，HARK ネットワーク内の AudioStreamFromMemory (25 行目) がバッファ処理を行い，取得データ量の揺れを吸収する仕組みになっているので，取得した分だけ push すればよい仕組みになっている．

Listing 2 は，逐次処理を意識する必要がないので，事前にネットワーク定義を行う必要もなく，逐次版に比べシンプルに記述することができる．13 行目は，入力信号を読み込む部分であり，データ型を float32 として読み込んでいることを除けば, 逐次版と同等のコードになっている[8]．逐次版では，フレームシフト長分 (センサの場合は, センサからその時までに得られたデータのみ）を publisher に push しており，512 サンプルのフレームを構成する処理は，AudioStreamFromMemory が行っていた．オフライン・バッチ版では，最初からすべてのデータが利用可能であることが前提であるので，わざわざこのような処理をするまでもなく，16 行目で直接，全データに対して，フレーム長 512 サンプル，シフト長 160 サンプルでフレーム化処理を行っている．フレーム化したデータをまとめて multi_fft に入力すれば，STFT の結果がまとめて得られ，その結果を localizemusic_node に入力すれば，定位結果を，さらにその結果を sourcetracker_node に入力すれば音源追跡結果を得ることができる．オフライン処理で記述すれば，このように，直感的に記述することが可能である．

PyHARK を用いれば，プロトタイピングの際は，オフライン・バッチ版を用いて記述し，その後逐次版に移行することで，同じ Python 上で実センサを用いて逐次処理版のプログラムを比較的容易に構築することができる．また，現在計画を進めている組込版は逐次処理版と親和性が高い設計になっているので，IoT などの実開発への移行コストを低減することが可能と考えられる．

5　評価実験

Listing 1，Listing 2，および従来の HARK の処理速度の比較を行った．

input.wav として， 8 チャンネルの音響信号 20 秒分を用いた．この信号には，マイクロホンアレイからみて，0 度と 180 度の方向に，それぞれ白色雑音が音源として含まれている．実験は，VMWare Player 16 上の Ubuntu 22.04 OS で行った．VM には Intel i7-12700K

[8] 逐次版とのデータ型の不整合については今後改善する予定

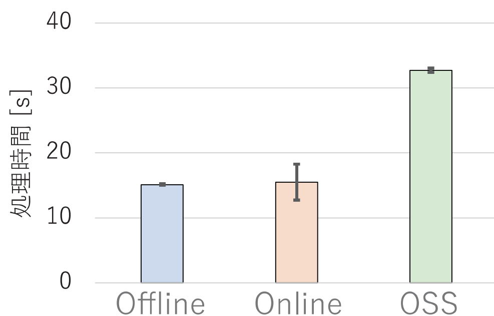


図 2: 処理時間計測結果

を 4 つ, またメモリを 32GB 割り振った．また，実験の際は，表示処理などの I/O の影響をなるべく低減するために，kivy などの描画や標準出力に書き込む処理を OFF にして実験を行った．また，従来は，実時間処理を担保するために，LocalizeMUSIC の固有値展開実行の頻度を通常は 50 フレームに一回と間引いて実行 (PERIOD=50) していたが，今回は PERIOD の値を 1 に設定することで毎フレーム固有値展開を実行する設定とした．実験は，各条件ごとに 10 回ずつ行い，平均と標準偏差をプロットした．

結果を図 2 に示す．まず，オフライン処理，オンライン処理共に，信号長である 20 秒以下で処理が終わっており，毎フレーム固有値展開を行っても，実時間処理性が保たれていることがわかる．また，オンライン処理には，実行時間にばらつきがあるものの，両者とも，平均実行時間は同程度となっており，オーバヘッドが同等であることがわかる．一方で，従来の HARK はリアルタイムファクターで 1.5 以上となっている．実際には，従来の HARK と比較すれば，PyHARK の実装のオーバヘッドは若干ではあるが，大きくなっているはずである．この結果は，PyHARK の実装の際に，同時に行った C++ 部分のマルチスレッド化，および Eigen の導入が，増加したオーバヘッド以上に効いており，高速化が実現できたと考える．

6 おわりに

本稿では，ロボット聴覚オープンソースソフトウェア HARK 3.4 について新たに加わった新機能である PyHARK に焦点を絞って紹介した．PyHARK は HARK の機能を Python から利用できるようにすると同時に，従来の HARK の利点である逐次処理も記述可能なパッケージである．設計的には，RaspberryPi などの ARM ベースの組込みシステム，FPGA，GPU への移植も考慮に入れた設計となっているため，将来的には，プロトタイプから実開発までシームレスに移行することが可能である．今後は，PyHARK の安定化，従来版 HARK とのソースの統合，ARM, FPGA, GPU サポートの検討を進めていきたい．

謝辞

本稿にかかる研究の一部は，JSPS 科研費 JP19KK0260 および JP20H00475 の助成を受けた．

参考文献

[1] K. Nakadai, T. Takahashi, H. G. Okuno, H. Nakajima, Y. Hasegawa, and H. Tsujino. Design and implementation of robot audition system "HARK". *Advanced Robotics*, Vol. 24, pp. 739–761, 2010.

[2] K. Nakadai, H. G. Okuno, and T. Mizumoto. Development, deployment and applications of robot audition open source software HARK. *Journal of Robotics and Mechatronics*, Vol. 29, No. 1, pp. 16–25, 2017.

[3] 中臺一博．オープンソースコミュニティーに貢献するということ．映像情報メディア学会誌, Vol. 71, No. 5, pp. 647–653, 2017.

[4] 中臺一博, 奥乃博．ロボット聴覚用オープンソースソフトウェア HARK の展開．デジタルプラクティス, Vol. 2, No. 2, pp. 133–140, 2011.

[5] K. Nakadai, H. G. Okuno, T. Takahashi, K. Nakamura, T. Mizumoto, T. Yoshida, T. Otsuka, and G. Ince. Introduction to open source robot audition software HARK. In *The 29th Annual Conference of the Robotics Society of Japan (RSJ2011)*, 2011.

[6] 奥乃博, 中臺一博．ロボット聴覚オープンソフトウエア HARK. 日本ロボット学会誌 特集「ロボット聴覚」, Vol. 28, No. 1, pp. 6–9, 2010.

[7] K. Nakadai, H. G. Okuno, H. Nakajima, Y. Hasegawa, and H. Tsujino. An open source software system for robot audition hark and its evaluation. In *2008 IEEE RAS International Conference on Humanoid Robots (Humanoids 2008)*, pp. 561–566, 2008.

[8] 中臺一博, 奥乃博, 中島弘史, 長谷川雄二, 辻野広司．ロボット聴覚オープンソースソフトウェア HARK の概要と評価. 第 26 回日本ロボット学会学術講演会予稿集 (RSJ 2008), 2008.

[9] 中臺一博, 山本俊一, 奥乃博, 中島弘史, 長谷川雄二, 辻野広司. ロボット聴覚用オープンソースソフトウェア HARK の概要. ロボティクス・メカトロニクス 講演会 2008 講演論文集, 2008.

[10] 公文誠, 若林瑞穂, 干場功太郎, 中臺一博, 奥乃博．ドローンによる地上音源の位置推定 - HARK を用いたドローン聴覚の取り組み．第 19 回計測自動制御学会システムインテグレーション部門講演会 (SI2018) 講演論文集, 2018.

[11] 鈴木麗璽, 炭谷晋司, 中臺一博, 奥乃博．ロボット聴覚技術を用いた鳥類の歌行動分析の試み - 複数のマイクロホンアレイを用いた二次元リアルタイム歌定位 -．第 18 回計測自動制御学会システムインテグレーション部門講演会 (SI2017) 講演論文集, pp. 1124–1126, 2017.

[12] 中臺一博, 坂東宜昭, 水本武志, 干場功太郎, 小島諒介, 糸山克寿, 杉山治, 公文誠, 奥乃博． HARK 2.3 の紹介とタフロボティクスチャレンジへの展開. 第 17 回計測自動制御学会システムインテグレーション部門講演会 (SI2016) 講演論文集, pp. 2175–2178, 2016.

[13] 中臺一博, 水本武志, 中村圭佑, 奥乃博．HARK 2.2 の新機能とその組込み，saas への展開. 第 16 回計測自動制御学会システムインテグレーション部門講演会 (SI2015) 講演論文集, pp. 1835–1838, 2015.

[14] 中臺一博, 奥乃博．ロボット聴覚オープンソースソフトウェア HARK の紹介．第 15 回計測自動制御学会システムインテグレーション部門講演会 (SI2014) 講演論文集, pp. 1712–1716, 2014.

[15] 中臺一博, 奥乃博. ロボット聴覚用オープンソースソフトウェア HARK 1.0.0 の概要. 第 11 回計測自動制御学会システムインテグレーション部門講演会 (SI2010) 講演論文集, pp. 1771–1774, 2010.

[16] 木下智義, 中臺一博．ロボット聴覚オープンソースソフトウェア hark 用ミドルウェア hark middleware の紹介．人工知能学会研究会資料 SIG-Challenge-057-012, pp. 73–78, 2020.