リアルタイム音分析

～チューナー開発と既存実機との比較～

関西大学　環境都市工学部　建築学科

建築環境工学第Ⅰ研究室

建 20－0071　樋富　かな子

指導教官　豊田　政弘　教授

目次

第1章　序論 1
1.1　研究背景 1
1.2　研究目的 2

第2章　研究方法 3
2.1　プログラムの構成 3
2.2　研究方法 14

第3章　研究結果 15
3.1　反応速度についての比較検討 15
3.1.1　音叉における反応速度 15
3.1.2　ファゴットにおける反応速度 18
3.1.3　反応速度についての考察 20
3.2　周波数についての比較検討 21
3.2.1　音叉における周波数 21
3.2.2　ファゴットにおける周波数 22
3.2.3　追加実験　安定した音における周波数 25
3.2.4　ピークの周波数における考察 26
3.3　考察 27

第4章　結論 28

参考文献 29

第 1 章　序論

1.1　研究背景

音の分析方法のひとつに、リアルタイム周波数分析がある。これは入力された波形を高速でフーリエ変換し、周波数成分に分解することで、入力波形にどの周波数成分が多く含まれているかをリアルタイムに分析するものである。この分析方法を利用した身近な機器の一つにチューナーがある。ここで言うチューナーとは、楽器が出す対象音の音程が正しいかを判断するための機器である。チューナーを使用する上で、音程の評価が安定するまでに時間がかかり、チューナーのメーター上での音程の評価が合っていたとしても、合奏などで周りと合わせたときに合わない場合があるという問題点がある。

1.2 研究目的

本研究の目的は、リアルタイムで正確に音を処理・評価するための周波数分析プログラムを作成することで、リアルタイム分析の基礎フレームワークを築くことである。1.1 で前述した既存のチューナーの問題点を解決するため、チューナーの根幹となるリアルタイムでの音の処理を行うプログラムについての理解を深め、その精度を高める。また、作成したプログラムを使用し、周波数毎に既存機器と比較を行うことで、プログラムの実用性を検討する。

第 2 章 研究方法

2.1 プログラムの構成

本研究では、Visual Studio 2022[1]、Intel Integrated Performance Primitives (Intel IPP)[2]を用いて、VST と呼ばれる、音楽制作ソフト上で動くエフェクタやデジタル楽器等のソフトウェア規格に基づいてプログラムを作成した[3]。作成したプログラムは主に、音声処理を行う「processor」、パラメーターの制御を行う「controller」、処理後の波形をグラフで表すといった画像処理を行う「guieditor」の 3 つのクラスから構成されている。以下にそれぞれのクラスが持つ関数について説明する。

入力された波形データは「processor」クラスで”process 関数“によって音声の処理が行われる。また、画像・信号処理を最適化したソフトウェアライブラリ Intel IPP に含まれる高速フーリエ変換関数によって、入力された波形の周波数成分を解析し、ピークとなる周波数を検出する。

■ Processor.cpp

```
// 自作VST用のインクルードファイル
#include "fuid.h"
#include "processor.h"

// VST3作成に必要なの名前空間を使用
namespace Steinberg{
namespace Vst {

		// ====================================================================================
		// コンストラクタ
		// ====================================================================================
		MyVSTProcessor::MyVSTProcessor()
		{
			// コントローラーのFUIDを設定する
			setControllerClass(ControllerUID);
		}

		// ====================================================================================
		// デストラクタ
		// ====================================================================================
		MyVSTProcessor::~MyVSTProcessor()
		{
			th_run = false;
			th.join();
		}

		// ====================================================================================
		// クラスを初期化する関数
		// ====================================================================================
		tresult PLUGIN_API MyVSTProcessor::initialize(FUnknown* context)
		{
			// まず継承元クラスの初期化を実施
			tresult result = AudioEffect::initialize(context);
			if (result == kResultTrue)
			{
				// 入力と出力を設定
				addAudioInput(STR16("AudioInput"), SpeakerArr::kMono);
				addAudioOutput(STR16("AudioOutput"), SpeakerArr::kMono);

				// 以下固有の初期化を実施。
				th = std::thread(&MyVSTProcessor::sendMessageThread, this);

				// FFT用
				order = (int)(log((double)wavlen) / log(2.0));
				ippsFFTGetSize_R_32f(order, IPP_FFT_NODIV_BY_ANY, ippAlgHintAccurate,
&sizeFFTSpec, &sizeFFTInitBuf, &sizeFFTWorkBuf);
				pFFTSpecBuf = ippsMalloc_8u(sizeFFTSpec);
				pFFTInitBuf = ippsMalloc_8u(sizeFFTInitBuf);
				pFFTWorkBuf = ippsMalloc_8u(sizeFFTWorkBuf);
				ippsFFTInit_R_32f(&pFFTSpec, order, IPP_FFT_NODIV_BY_ANY, ippAlgHintNone,
pFFTSpecBuf, pFFTInitBuf);
				if (pFFTInitBuf) ippFree(pFFTInitBuf);
				ippsSet_32f(1, win, wavlen);
```

```
                ippsWinHann_32f_I(win, wavlen);
        }

        // 初期化が成功すればkResultTrueを返す。
        return result;
}

tresult PLUGIN_API MyVSTProcessor::setBusArrangements(SpeakerArrangement* inputs, int32 numIns,
SpeakerArrangement* outputs, int32 numOuts)
{
        // inputとoutputのバスが1つずつで、かつinputとoutputの構成がモノラルの場合
        if (numIns == 1 && numOuts == 1 && inputs[0] == SpeakerArr::kMono && outputs[0] ==
SpeakerArr::kMono)
        {
                return AudioEffect::setBusArrangements(inputs, numIns, outputs, numOuts);
        }

        // 対応していないバス構成の場合、kResultFalseを返す。
        return kResultFalse;
}

// ================================================================================
// 音声信号を処理する関数
// ================================================================================
tresult PLUGIN_API MyVSTProcessor::process(ProcessData& data)
{
        // 入力・出力バッファのポインタをわかりやすい変数に格納
        // inputs[]、outputs[]はAudioBusの数だけある(addAudioInput()、addAudioOutput()で追加
した分だけ)
        // 今回はAudioBusは1つだけなので 0 のみとなる
        // channelBuffers32は32bit浮動小数点型のバッファで音声信号のチャンネル数分ある
        // モノラル(kMono)なら 0 のみで、ステレオ(kStereo)なら 0(Left) と 1(Right) となる
        Sample32* in = data.inputs[0].channelBuffers32[0];
        Sample32* out = data.outputs[0].channelBuffers32[0];

        // numSamplesで示されるサンプル分、音声を処理する
        for (int32 i = 0; i < data.numSamples; i++)
        {
                out[i] = in[i];
        }

        // リングバッファに書き込み
        rbL.bwrite(in, data.numSamples);
        rbL.bmove(data.numSamples);

        // 描画速度に合わせて約0.1秒ごとに波形を更新する
        int sendTiming = (int)processSetup.sampleRate / 10;
        samplecount += data.numSamples;

        if (samplecount > sendTiming)
        {
                if (mtx.try_lock()) // wavdata[]にアクセスするのでロックする。(process()関
数がロック待ちにならないよう、try_lock()にしている)
                {
                        // Fourier変換(cf. P.143, 262, 277, 333, 270)
                        //
https://www.intel.com/content/www/us/en/developer/articles/training/how-to-use-intel-ipp-s-1d-fourier-
transform-functions.html
                        rbL.bread(x, wavlen);
                        ippsMul_32f_I(win, x, wavlen);
                        ippsFFTFwd_RToPack_32f(x, X, pFFTSpec, pFFTWorkBuf);
                        ippsConjPack_32fc(X, Xc, wavlen);
                        ippsAbs_32fc_A11(Xc, Xabs, wavlen);
                        for (int32 i = 0; i < wavlen / 2; i++)
                        {
```

```
                                        Xabs[i] = 20.0 * log10(Xabs[i]) / 100.0;
                                }

                                // ピーク検出
                                Xmax = 0.0f;
                                for (int32 i = (int)((float)flow / df); i < (int)((float)fhigh /
df); i++)
                                {
                                        if (Xabs[i] > Xmax)
                                        {
                                                Xmax = Xabs[i];
                                                Xmaxindex = i;
                                        }
                                }

                                // wavdataに周波数特性をコピー
                                memcpy(wavdata, Xabs, sizeof(float) * wavlen / 2);

                                samplecount -= sendTiming;

                                mtx.unlock();
                        }
                }

                // 問題なければkResultTrueを返す(おそらく必ずkResultTrueを返す)
                return kResultTrue;
        }

        void MyVSTProcessor::sendMessageThread()
        {
                while (th_run)
                {
                        // メッセージの送信では、ポインタを直接パラメーター操作クラス
(EditController側)に渡す。
                        // メンバー変数wavdataのポインタを渡すことも可能だが、複数スレッドからのア
クセスは
                        // 予期しない不具合の可能性があるので、ローカル変数にコピーしている。
                        float data[wavlen / 2];
                        int64 idata = 0;

                        if (mtx.try_lock()) // wavdata[]にアクセスするのでロックを試す。
                        {
                                memcpy(data, wavdata, sizeof(float) * wavlen / 2);
                                idata = Xmaxindex;
                                mtx.unlock();
                        }

                        // パラメーター操作クラスにメッセージを送信するため、
                        // allocateMessage()でホストアプリ側からメッセージクラスを取得する
                        IMessage* msg = allocateMessage();

                        // メッセージクラスにメッセージIDを設定する
                        msg->setMessageID("WaveData");

                        // アトリビュートIDを「data」としてバイナリデータをメッセージクラスに追加
する
                        msg->getAttributes()->setBinary("data", (void*)data, sizeof(float) *
wavlen / 2);

                        // アトリビュートIDを「idata」としてバイナリデータをメッセージクラスに追加
する
                        msg->getAttributes()->setInt("idata", idata);

                        // メッセージをパラメーター操作クラス(EditController側)に送信する
                        // パラメーター操作クラスではnotify()関数が実行される
```

```
                                sendMessage(msg);

                                // 確保したメッセージクラスを解放する
                                msg->release();

                                // 0.1秒待つ
                                std::this_thread::sleep_for(std::chrono::milliseconds(100));
                        }
                }

}
} // namespace Steinberg と Vst の終わり
```

「processor」クラスで処理された波形データは”send Message Thread 関数“によって「controller」クラスに送信され、”notify 関数“において受け取られる。

■ Controller.cpp

```
// 自作VST用のインクルードファイル
//#include "myvst3def.h"
#include "myvst3fuid.h"
#include "controller.h"


// VST3作成に必要なの名前空間を使用
namespace Steinberg{
namespace Vst {


// クラスを初期化する関数(必須)
tresult PLUGIN_API MyVSTController::initialize(FUnknown* context)
{
	// まず継承元クラスの初期化を実施
	tresult result = EditController::initialize(context);
	if (result == kResultTrue)
	{

		// 以下固有の初期化を実施。

	}

	// 初期化が成功すればkResultTrueを返す。
	result = kResultTrue;
	return result;
}

// 自作VST GUIEditorを作成する関数
IPlugView* PLUGIN_API MyVSTController::createView(const char* name)
{
	// editorを指定された場合
	if (strcmp(name, "editor") == 0)
	{
		// 自作GUIクラスのインスタンスを作成し返す
		MyVSTGUIEditor* view = new MyVSTGUIEditor(this);
		return view;
	}
	return 0;
}

// 音声処理クラスでメッセージを受け取った時の関数
tresult MyVSTController::notify(IMessage* message)
{
	if (!message) { return kInvalidArgument; }

	// メッセージのIDをチェックする
	if (strcmp(message->getMessageID(), "WaveData") == 0)
	{
		// メッセージデータ読込用の一時変数
		const void* data;
		uint32 datasize;
		int64 idata;

		// アトリビュートID「data」を指定して、バイナリデータを読み込む
		message->getAttributes()->getBinary("data", data, datasize);
```

```
                    // 読み込んだバイナリデータをパラメーター操作クラス側のメンバー変数wavdataにコピーす
る
                    datasize = std::min(datasize, (uint32)(sizeof(float) * wavlen/2));
                    memcpy(wavdata, data, datasize);

                    // アトリビュートID「data」を指定して、バイナリデータを読み込む
                    message->getAttributes()->getInt("idata", idata);

                    //読み込んだバイナリデータをパラメーター操作クラス側のメンバー変数maxindexにコピーす
る
                    maxindex = (int)idata;

                    return kResultOk;
            }

            return EditController::notify(message);
}

} } // namespace SteinbergとVstの終わり
```

また、処理したデータを視覚的に確認するために、”create View 関数”を用いて GUI を作成し、「guieditor」クラスで周波数分析結果の描画を行う。これにより、処理された波形は図 1 のように表示される。

■ Guieditorr.cpp

```
// 自作VST用のインクルードファイル
#include "guieditor.h"
#include "controller.h"

// VST3作成に必要なの名前空間を使用
namespace Steinberg {
namespace Vst {

// ======================================================================================
// コンストラクタ
// ======================================================================================
MyVSTGUIEditor::MyVSTGUIEditor(void* controller)
	: VSTGUIEditor(controller)
{
	// コンストラクタでウィンドウサイズを設定する
	// 設定しなければ、ウィンドウが開かない
	ViewRect viewRect(0, 0, 600, 400);
	setRect(viewRect);
}



// ======================================================================================
// GUIウィンドウを開いたとき、閉じたときに呼び出される関数
// ======================================================================================
bool PLUGIN_API MyVSTGUIEditor::open(void* parent, const PlatformType& platformType)
{
	// GUIウィンドウが開かれたときに、UIを作成する

	// まずはフレーム(配置領域)がすでに作成・設定されているか確認。
	// すでに作成・設定されている場合(frameがNULLでない場合)は終了
	// frameは継承元クラスで定義されている。
	if (frame) { return false; }

	// 作成するフレームのサイズを設定
	CRect size(0, 0, 600, 400);

	// フレームを作成。作成に失敗したら(NULLなら)終了。
	// 引数には、フレームサイズ、自作GUIクラスのポインタを指定する
	frame = new CFrame(size, this);
	if (frame == NULL) { return false; }

	// 作成したフレームに背景画像を設定
	//CBitmap* cbmp = new CBitmap("background.png"); // リソースから背景画像を読み込む
	//frame->setBackground(cbmp); // フレームに背景画像を設定
	//cbmp->forget();	// フレームに設定後は背景画像はforgetで解放しておく

	// 作成したフレームを開く
	frame->open(parent);

	// CWaveDrawViewを作成する
	waveView = new CWaveDrawView(size);

	// CWaveDrawViewに設定する波形情報
```

```
		// とりあえず適当な周波数のサイン波を設定
		float wav[wavlen/2];
		for (int i = 0; i < wavlen/2; i++)
		{
			wav[i] = sin(2.0 * 3.14159265 * (double)(i * 880) / 44100.0);
		}

		// 波形情報を設定
		waveView->setWave(wav, wavlen/2);

		// フレームに追加する
		frame->addView(waveView);

		CRect size2(0, 0, 100, 30);
		size2.offset(500, 0);
		textLabel = new CTextLabel(size2, "");
		textLabel->setFont(kNormalFontBig);
		textLabel->setFontColor(kBlackCColor);
		textLabel->setBackColor(kWhiteCColor);
		frame->addView(textLabel);

		// GUIウィンドウのオープンに成功した場合はtrueを返す
		return true;
}


// GUIウィンドウを閉じたときに呼び出される関数
void PLUGIN_API MyVSTGUIEditor::close()
{
		// GUIウィンドウが閉じたときに、UIを削除する

		// フレームを解放
		// 背景画像や追加したつまみ(ノブ) やスライダーなどもあわせて解放される
		// (個別で解放する必要はない)
		if (frame)
		{
			frame->forget();
			frame = 0;
		}
}

// ==============================================================================================
// GUIウィンドウのコントローラを操作したときに呼び出される関数
// ==============================================================================================
void MyVSTGUIEditor::valueChanged(CControl* pControl)
{
		// どのパラメーターが操作されたかを取得する。
		int32 index = pControl->getTag();

		// パラメーターの値を取得する。
		float value = pControl->getValueNormalized();

		// 取得した値をパラメーターに反映させる
		controller->setParamNormalized(index, value);

		// 音声処理クラスに反映した値を通知する
		controller->performEdit(index, value);
}
CWaveDrawView::CWaveDrawView(const CRect& size)
		: CView(size) // 継承元のコンストラクタを呼び出す。
{
		// ここでは特に何もすることはない。
};

void CWaveDrawView::draw(CDrawContext* pContext)
```

```
{
	// まずは背景を描画する。
	// setBackGround()関数で登録された画像ファイルがあるか確認し、描画する。
	if (getDrawBackground())
	{
		getDrawBackground()->draw(pContext, getViewSize());
	}

	// ここから波形の描画
	// まずは描画する線の色と見た目(スタイル)を設定する。
	pContext->setFrameColor(kWhiteCColor); // 線の色を設定
	pContext->setLineStyle(kLineSolid);    // 線のスタイル(終端の丸み、点線かどうか等)を設定
	pContext->setLineWidth(1.0);  // 線の太さを設定

	// メンバー変数のlinesを使用して、複数の直線を一気に描画する。
	pContext->drawLines(lines);

	// 描画後は必ずsetDirtyにfalseを設定して更新済みとする。
	setDirty(false);
}

void CWaveDrawView::setWave(float buf[], int size)
{
	lines.clear(); // まずは直線のリストをクリアしておく。

	// buf[]の内容をもとに、描画する直線のリスト(配列)を作成する。

	// 描画する直線には始点座標(x,y)と終点座標(x,y)が必要となる。
	// なお、座標はウィンドウの左上を座標(0,0)とした絶対座標で指定する必要がある。

	// まずはCWaveDrawViewクラスの座標情報(CRect型)を取得する。
	CRect viewSize = getViewSize();

	// CWaveDrawViewクラスの左・真ん中を最初の始点座標とする。
	CCoord leftX = viewSize.left;            // viewSizeから左端の座標を取得する
	CCoord centerY = viewSize.getCenter().y; // viewSizeから中央の座標(xとy)を取得し、そのy座標のみ
をcenterYとする。
	CPoint from(leftX, centerY); // 直線の始点座標の変数

	for (int i = 0; i < size; i++)
	{
		// 描画する直線用にCWaveDrawViewクラスの幅と高さを取得する
		CCoord width = viewSize.getWidth();
		CCoord height = viewSize.getHeight();

		// 終点座標を計算する。
		// x座標は左端から始まって、最終的には右端(leftX + width)になるようにする。
		// y座標はcenterYを中心に、buf[i]の振幅に合わせて上端から下端の間になるようにする。
		CPoint to(leftX + width * ((double)(i + 1) / (double)size), centerY - height / 2.0 *
buf[i]);

		// 始点座標と終点座標から直線を作成し、直線リスト(配列)に加える。
		CDrawContext::LinePair line(from, to);
		lines.push_back(line);

		// 今回の終点座標を次の直線の始点座標とする。
		from = to;
	}
}


CMessageResult MyVSTGUIEditor::notify(CBaseObject* sender, const char* message)
{
	if (message == CVSTGUITimer::kMsgTimer)
	{
```

```
                    if (waveView != nullptr)
                    {
                              // IPtrからEditControllerのポインタを取得する
                              EditController* tmpctrl = controller.get();

                              float* tmp = ((MyVSTController*)tmpctrl)->wavdata;
                              int size = wavlen / 2;
                              waveView->setWave(tmp, size);
                              waveView->setDirty();
                    }
                    if(textLabel!=nullptr)
                    {

                              EditController* tmpctrl = controller.get();
                              int tmp= ((MyVSTController*)tmpctrl)->maxindex;
                              ((CTextLabel*)textLabel)->setText(std::to_string(df * (float)tmp));
                    }
          }

          return VSTGUIEditor::notify(sender, message);
}

} } // namespace SteinbergとVstの終わり
```

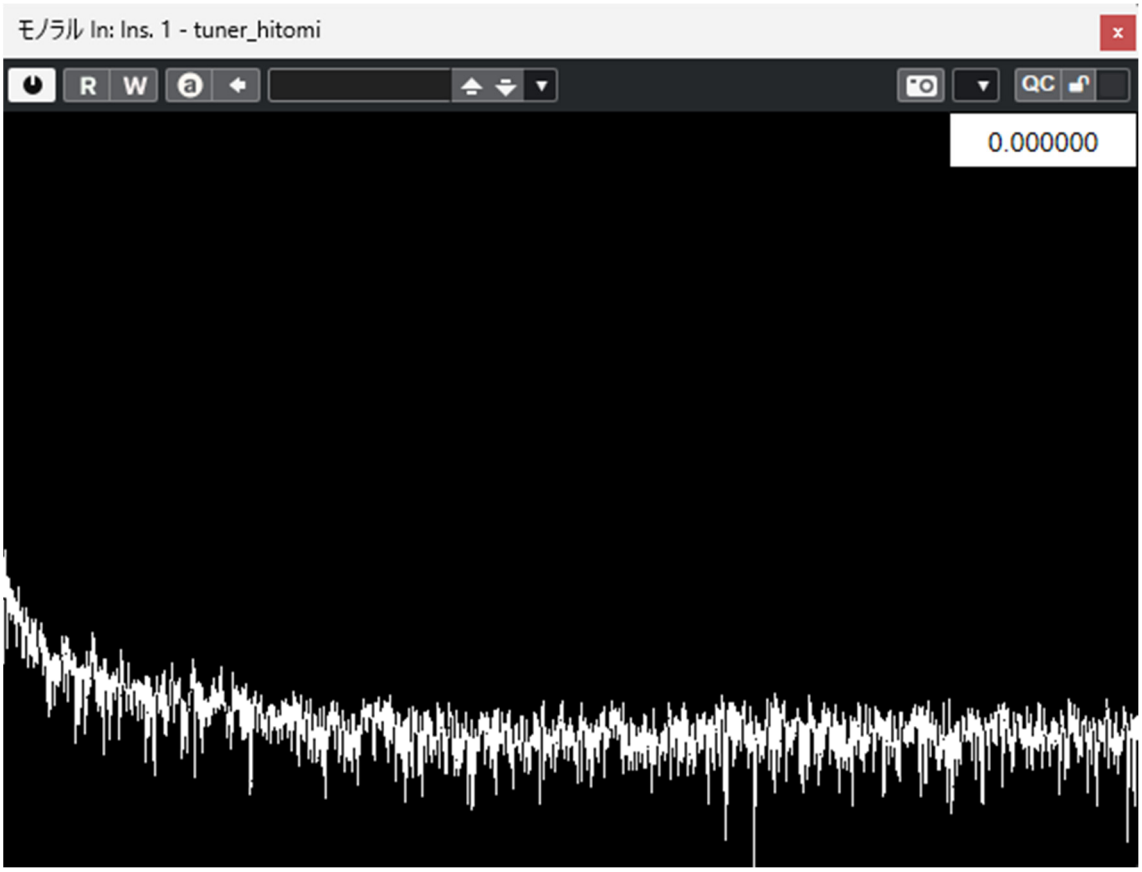

図１　処理された波形の描画

2.2　研究方法

2.1 で記述したプログラムを用いて、既存機器との比較を行う。実験では、音叉と吹奏楽器の一つであるファゴットを用いて比較を行った。入力した音を 1 秒あたり 44,100 個のデータにサンプリングし、そこから 2 の乗数分の音のデータを切り取ってフーリエ変換し、420 Hz〜460 Hzの範囲内でピークとなる周波数を検出する。作成したプログラムの精度を確かめるため、切り取るデータ数を変更して実験を行い、ファゴットでは 1 オクターブ毎の 3 種類の音で比較した。また、音叉、楽器ともに、5 回ずつ実験を行い、音を鳴らし始めてからその音の周波数が確定するまでの時間の平均、作成したチューナーを使用した時に検出された周波数を調べる。

チューナーの基準周波数は一般的に 440 Hzとされており、音叉も 440 Hzを発するものを用いて比較を行った。ただし、吹奏楽では 442 Hzを基準周波数とすることが一般的であることから、ファゴットではその基準に従い比較を行った。

第 3 章　研究結果

本研究の実験において、65536 個を超えるデータ数を切り取った際に処理速度の関係からプログラムが実行不可になった。したがって、切り取るデータ数の最大値は 65536 個として比較を行った。

3.1　反応速度についての比較検討

16384 個未満でデータを切り取った場合には、音を鳴らすと俊二に周波数が表示されたため、結果のグラフは省略する。

3.1.1　音叉における反応速度

音叉で比較検討した際の反応速度についての結果は、図 2，3，4 に示す通りである。

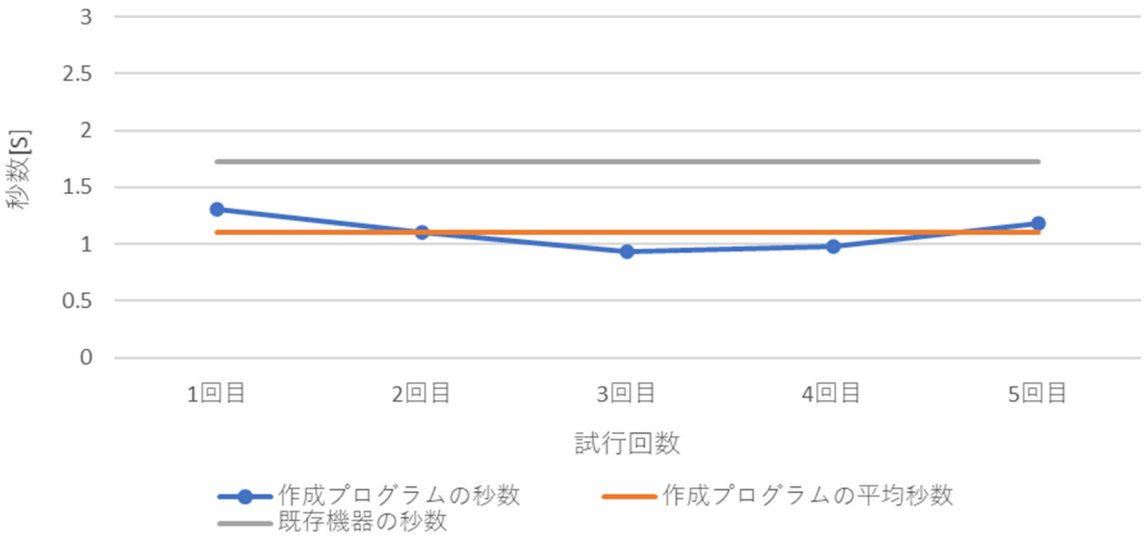

図2　音叉/65536個のデータ切り取り時

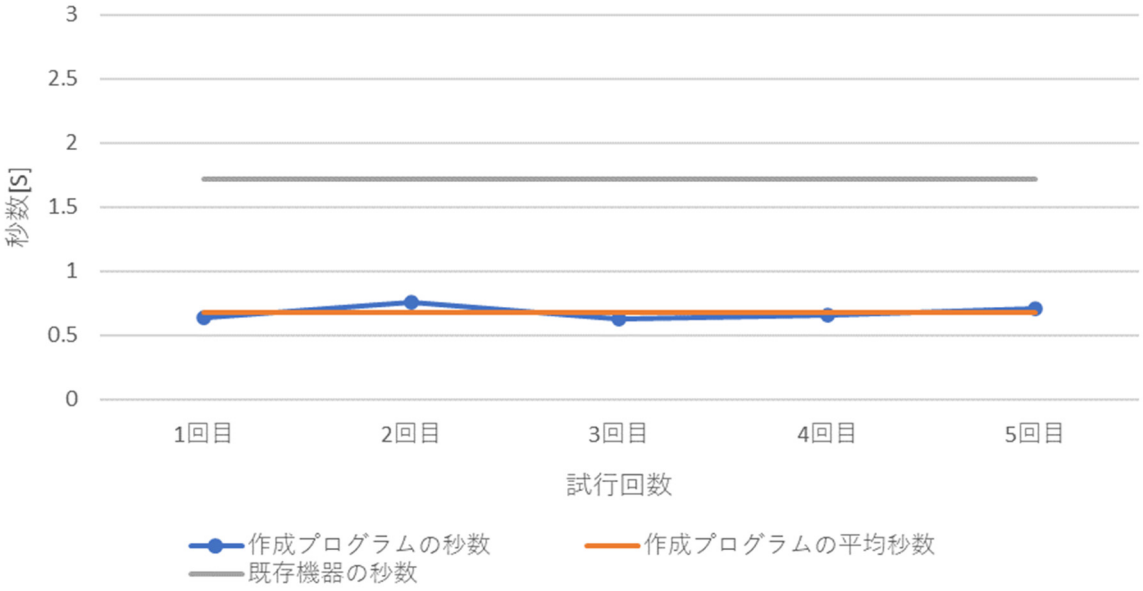

図3　音叉/32768個のデータ切り取り時

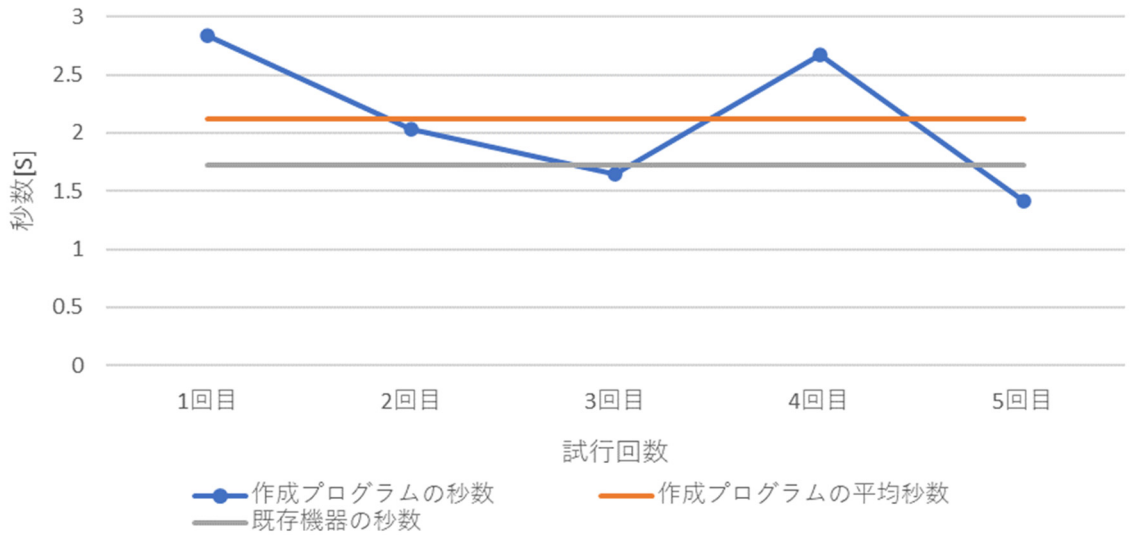

図4　音叉/16384個のデータ切り取り時

以上のグラフから、音の反応速度に関して、既存機器のチューナーよりも作成したチューナーの方が比較的反応速度が速いことが分かる。しかし、16384 個のデータを切り取った時のみ、周波数が安定して表示されるまで、平均 2.12 秒と既存機器よりも時間がかかっていることが見て取れる。更に実験回数を追加して計 10 回試行したが、同様の結果となったことから、16384 個のデータを切り取った時には処理に時間を要することが分かる。その理由として、切り取ったデータのうち、ピークとなる周波数のばらつきが大きかったことが考えられる。

3.1.2　ファゴットにおける反応速度

ファゴットで比較検討した際の反応速度についての結果は、図 5，6，7 に示す通りである。

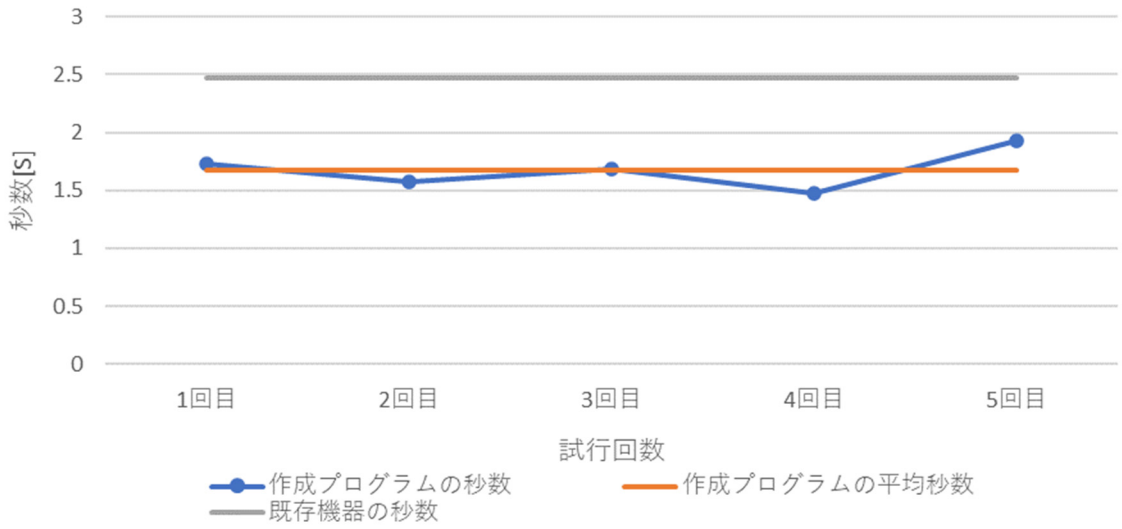

図5　ファゴット/65536個のデータ切り取り時

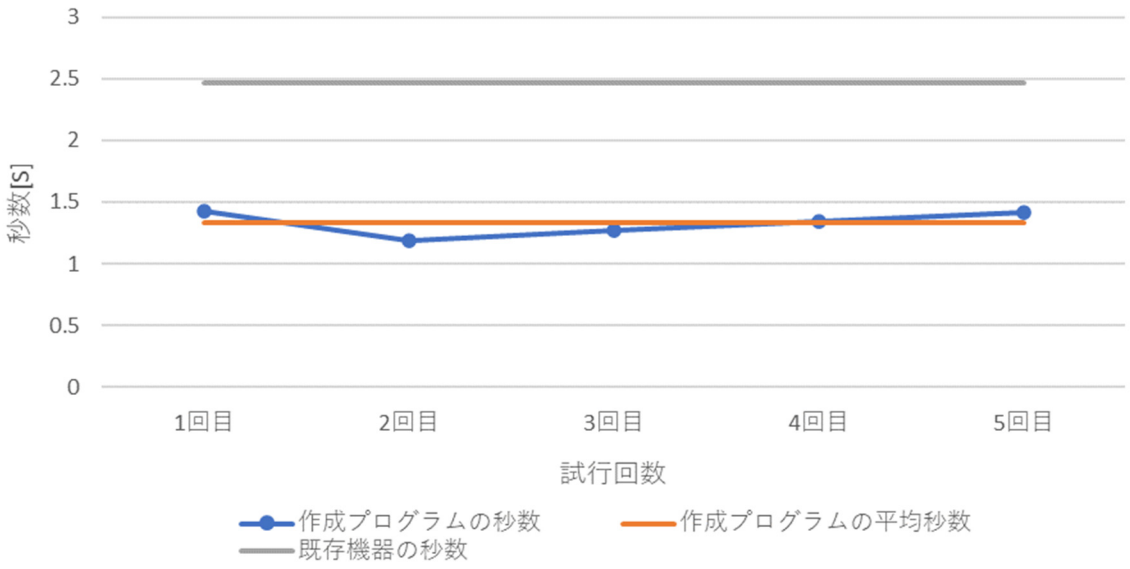

図6　ファゴット/32768個のデータ切り取り時

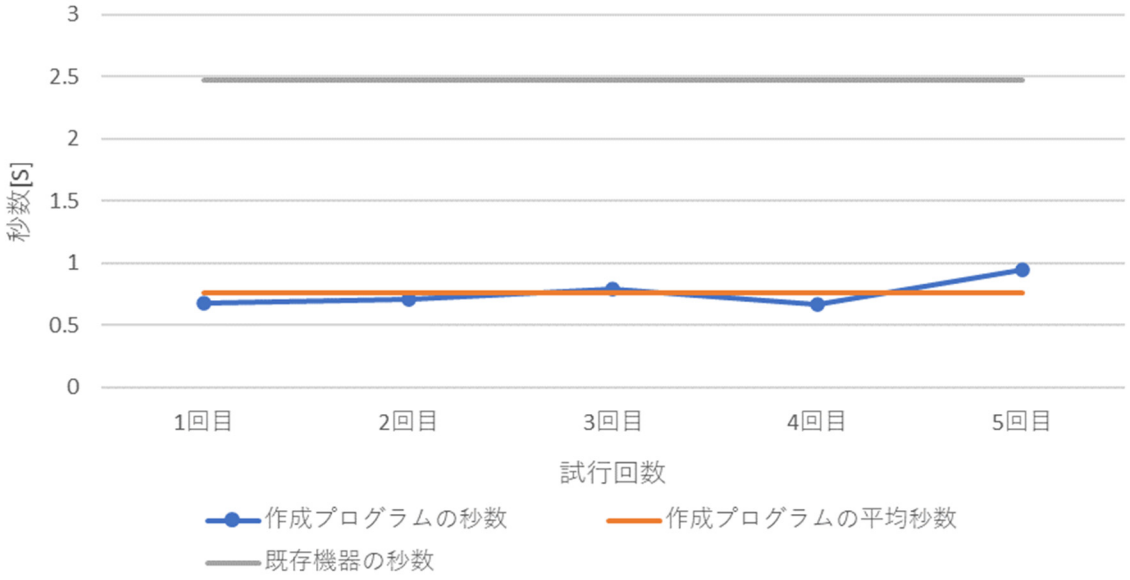

図7　ファゴット/16384個のデータ切り取り時

以上のグラフより、音叉での実験結果と同様に、切り取るデータ数を少なくすればするほど、反応速度は速くなっていくことがわかる。また、本研究での最大値である 65536 個で切り取った際でも、既存機器よりも速い反応速度で周波数が表示されている。

3.1.3　反応速度についての考察

音叉とファゴットのどちらの実験結果においても、切り取るデータ数が少ないほど処理するデータが少なく、時間を要さないことが分かった。また、最大値である 65536 個のデータを切り取った際においても、どちらも既存機器よりも速い反応速度で周波数を表示した。このことから反応速度において、作成したプログラムは既存機器よりもリアルタイムに近く処理することができるプログラムであると言える。

3.2　周波数についての比較検討

3.2.1.　音叉における周波数

作成したチューナーで、音叉を鳴らしたときの結果は図 8 に示す通りである。

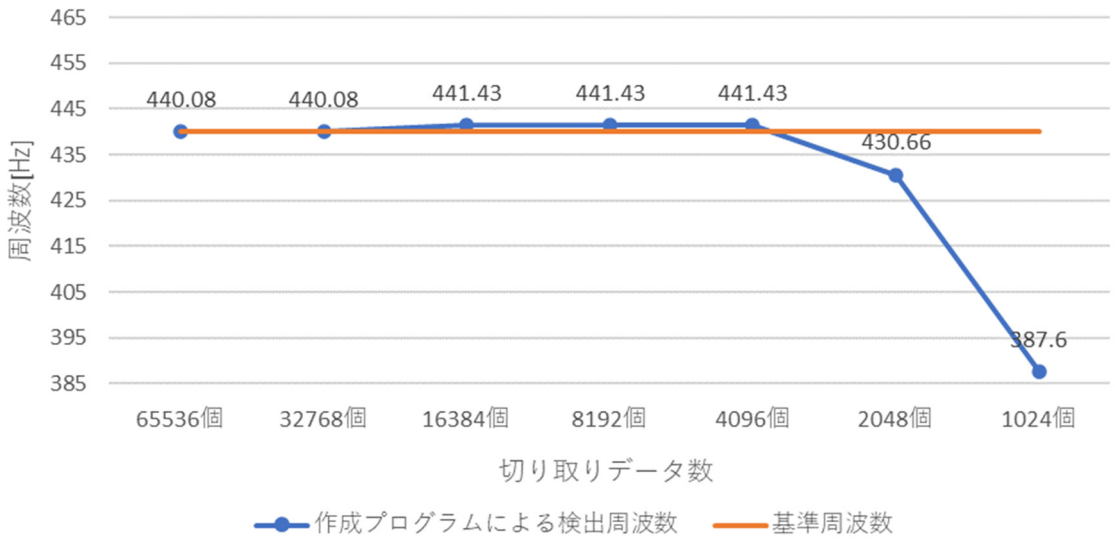

図8　音叉における周波数

4096～65536 個のデータを切り取った場合のピークの周波数は基準値である 440 Hzに近くなっており、チューナーとして機能することが分かる。しかし、2048 個以下でデータを切り取った場合、ピークとなる周波数は基準値から大きく外れてしまうことから、最低でも 4096 個のデータを切り取る必要があるとわかる。

3.2.2　ファゴットにおける周波数

作成したプログラムにおいて、ファゴットを吹いた時の周波数の結果を以下に示す。

チューニング B♭を吹いた時の切り取るデータ数によるピークの周波数の違いは図 9 に示す通りである。

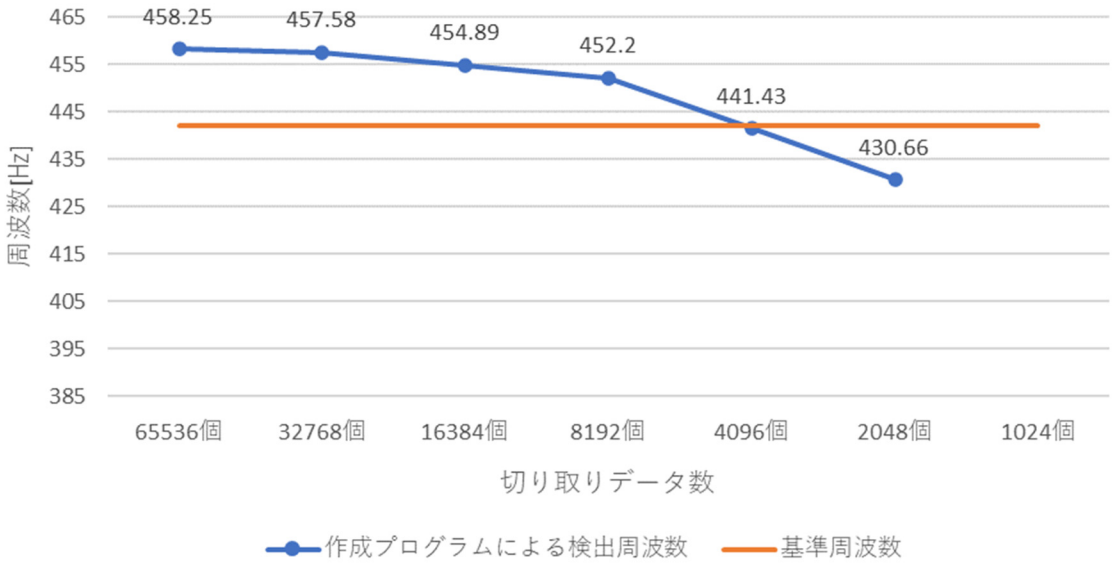

図9　チューニングB♭における周波数

チューニング B♭において、基準値である 442 Hzに最も近いピークを検出したのは、データを 4096 個切り取った場合であり、最大値である 65536 個で切り取った時の周波数と約 17 Hzの差があった。8192～65536 個のデータを切り取った時の周波数のピークに大きな差はないが、4096 個以下の個数のデータを切り取った際にはピークに大きく差が出ており、また、1024 個のデータを切り取った際、周波数は表示されず、測定不可であった。

チューニング B♭の 1 オクターブ上の B♭を吹いた時の、切り取るデータ数によるピークの周波数の違いは図 10 に示す通りである。

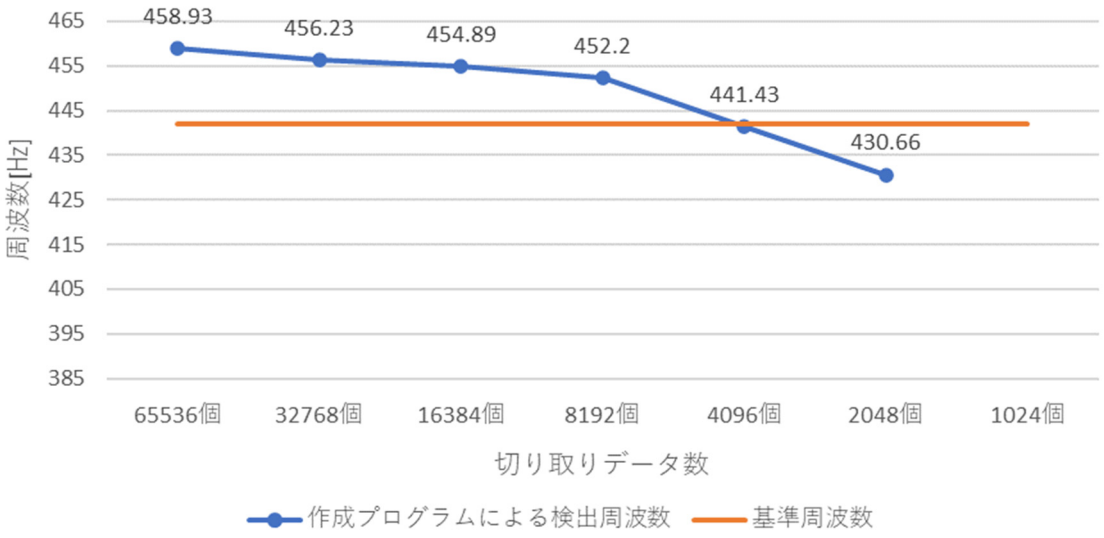

図10　高いB♭における周波数

図 9 と比べてみると、ピークの周波数の値はチューニング B♭を吹いた時とほとんど変わらない結果となり、4096 個のデータを切り取った時に最も基準値である 442 Hzに近くなった。また、1024 個のデータを切り取った時の周波数は測定不可であった。

チューニング B♭の 1 オクターブ下の B♭を吹いた時の、切り取るデータ数による周波数の違いは図 11 に示す通りである。

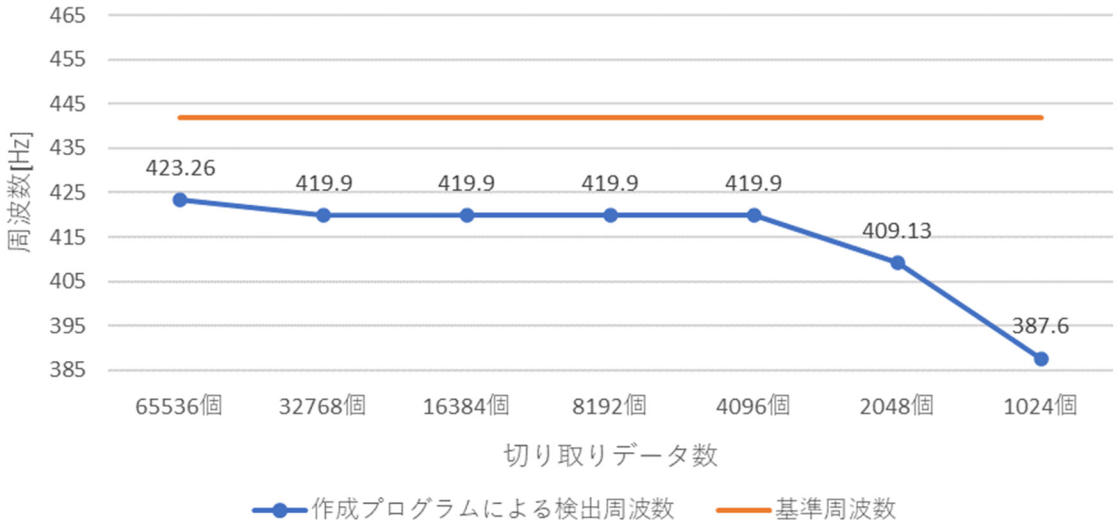

図11　低いB♭における周波数

チューニング B♭の 1 オクターブ下の B♭において、いずれのデータ数で切り取っても基準値を下回った。基準値に最も近いピークを検出したのは、65536 個のデータを切り取った時であるが、18 Hz以上下回る結果となった。4096～65536 個のデータを切り取った時のピークの周波数には大きな差はなく、2048 個以下のデータを切り取った際には大きく差が出ており、前述の 2 つの結果とは違って、1024 個のデータを切り取った時のピークとなる周波数も安定して検出することができた。また、それぞれのピークの周波数の差は最大で約 46 Hzであった。

3.2.3　追加実験　安定した音における周波数

3.2.2 で示した結果は人が実際に吹いた音を使用しているため、安定した音とは言えず、結果に多少の影響が出ることが危惧される。そこで、チューニング B♭と同じ高さである 442 Hzの B♭の音を機械で流し、同様に実験を行った。

その結果は図 12 に示す通りである。

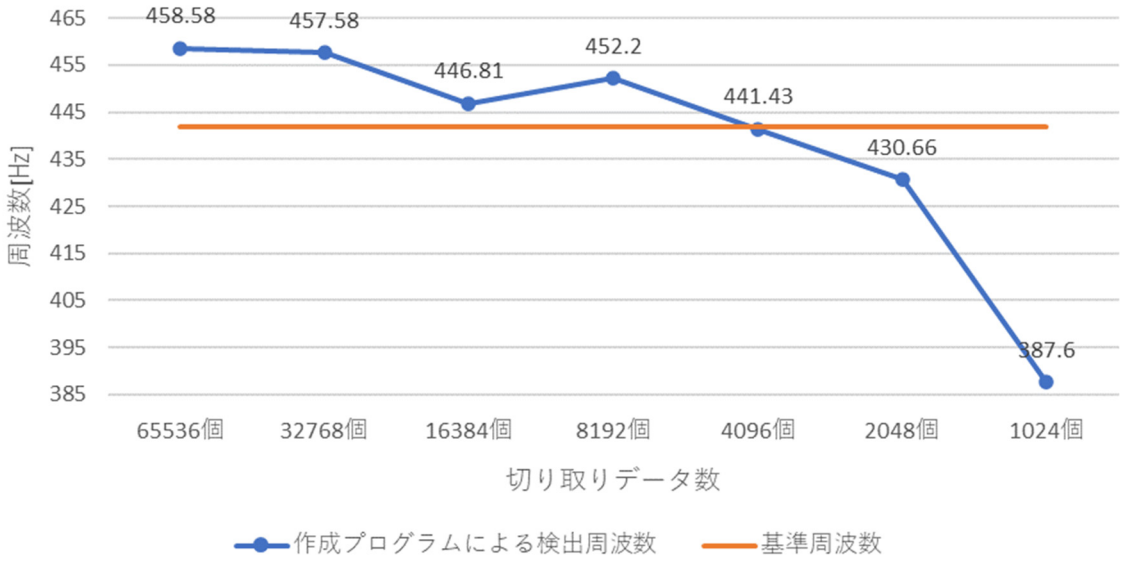

図12　安定した音における周波数

安定した音において、1024、16384 個のデータを切り取った時以外はチューニング B♭で実験した場合と大きくは変わらない結果となった。最も基準値に近いピークを検出したのは、チューニング B♭の時と同様に 4096 個のデータを切り取った時であった。

3.2.4　ピークの周波数における考察

3.2.2 の結果より、本研究で作成したプログラムは低音に関して、切り取るデータ数が少なくても機能したことから、処理の可否の観点のみで評価すれば、中高音域の音よりも低音域の音を処理する方が優れていると言える。ただし、低音に関しては、いずれのデータ数で切り取っても正確なチューナーとしては成り立たない。また、中高音に関しては 4096 個のデータを切り取った時のピークが最もチューナーとして正確に機能すると言える。音の高さによって結果に差が出ていることから、どの領域の音にも対応できるプログラムではないことが分かる。

また、3.2.3 の結果より、機械が出す安定した音でも人が吹いた安定していない音でも、作成したプログラムにおいて、4096 個のデータを切り取る場合に最もチューナーとして正確に機能する、という同じ結果を得ることが出来た。このことから、本研究において作成したプログラムは 4096 個のデータで切り取る場合に正しいチューニングができるものであると言える。

3.3 考察

実験結果より、作成したプログラムは既存機器と比べて、反応速度においてはいずれのデータ数で切り取ったとしてもリアルタイムに近い処理ができるプログラムであり、中高音に関しては4096個のデータ数で切り取った時に正確にチューナーとして機能するプログラムであると言える。しかし、低音ではピークとなる周波数が基準値より大幅に下がっており、音の周波数範囲によってプログラムの正確性に差が出る結果となった。実際にチューニングをする際にこのプログラムを使用する場合には、チューニング B♭でチューニングを行うため問題はないと考えられるが、個々の音の音程を確かめる際には問題があると言える。

第 4 章　結論

実験結果および考察より、本研究の目的であった「リアルタイムに正確に音を処理することが出来るプログラムを作る」ことは、中低音域の音については、作成したプログラムで達成することができたが、低音域ではピークとなる周波数が中高音と比べて大幅に低いデータが検出されたことから、切り取るデータ数の増加やプログラムの見直しを検討することが必要であると考えられる。

また、今回は中低音の吹奏楽器であるファゴットでの実験であったことから、高音域の楽器や弦楽器での実験を行った場合には、正確にチューニングできる時の切り取るデータ数や反応速度に違いが出るだろう。あらゆる楽器で比較検討を行うことが今後の課題である。

参考文献

[1] Microsoft Visual Studio, https://visualstudio.microsoft.com/ja/.

[2] Intel Integrated Performance Primitives,
https://www.intel.com/content/www/us/en/developer/tools/oneapi/ipp.html.

[3] うつぼかずら、VST プラグイン、工学社、2020.