メタビルドシステムCMakeによるFrontISTRの構築

-FrontISTR v5.0に向けて-

アドバンスソフト株式会社
技術第2部 徳永 健一

独立研究開発法人 海洋研究開発機構
地球情報基盤センター 小川道夫

本日の流れ(1)

- 次期バージョンFrontISTR v5.0に向けて
- CMakeとは
- CMakeの簡単な実行例
- CMakeの基礎
- FrontISTR, REVOCAP_Refiner/REVOCAP_Meshへの適用
- CMakeによるFrontISTR構築
- CMakeによるREVOCAP_Refiner/REVOCAP_Meshの構築
- Tips

本日の流れ(2)

- REVOCAP_Refiner/REVOCAP_Meshの構築および、テストモジュールCTestについてCMakeのとは
- Ctestとは
- FrontISTRのテスト
- テストの実行方法
- テストの実行結果
- CMakeLists.txtでの記述
- テスト判定ルーチン
- より良いテストのために
- REVOCAPのテスト

次期バージョン FrontISTR v5.0 に向けて

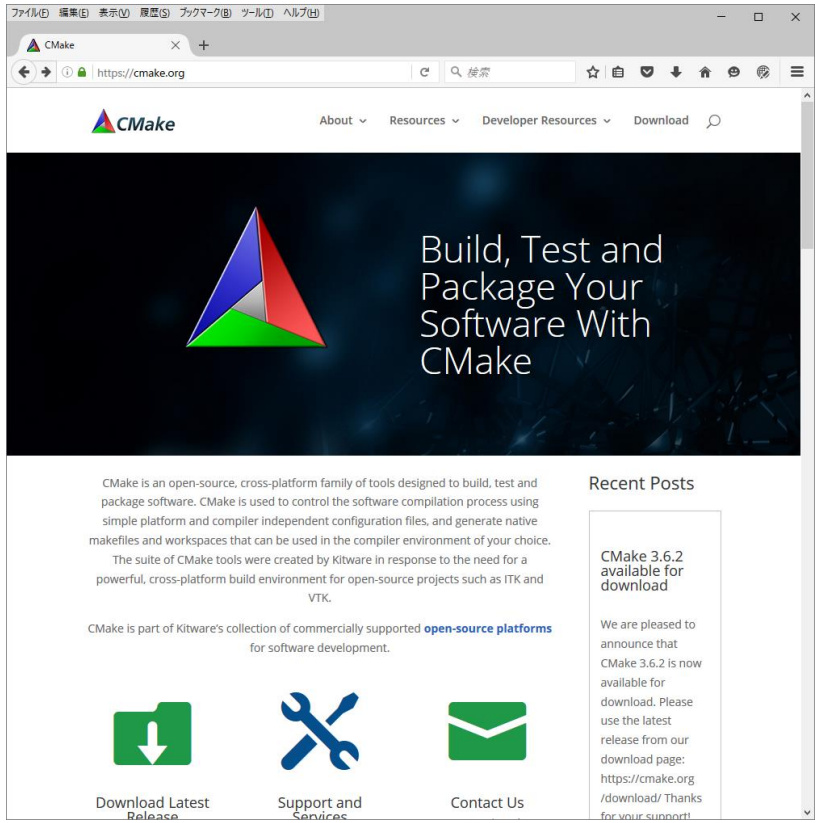
- FrontISTRの構築を簡素化するため cmake を使ったビルドプロセスをテスト的に導入しました。

- Makefile.confの編集が不要になります。
- 外部ライブラリの探索がほぼ自動になります。
- UNIX系以外のプラットフォームでのビルドも簡単になります。
  - Windows 10(MinGW)でのビルドは確認しています。
  - Mac系も確認環境があれば出来るかもしれません。

(※ 予定も含む)

CMakeとは (1)

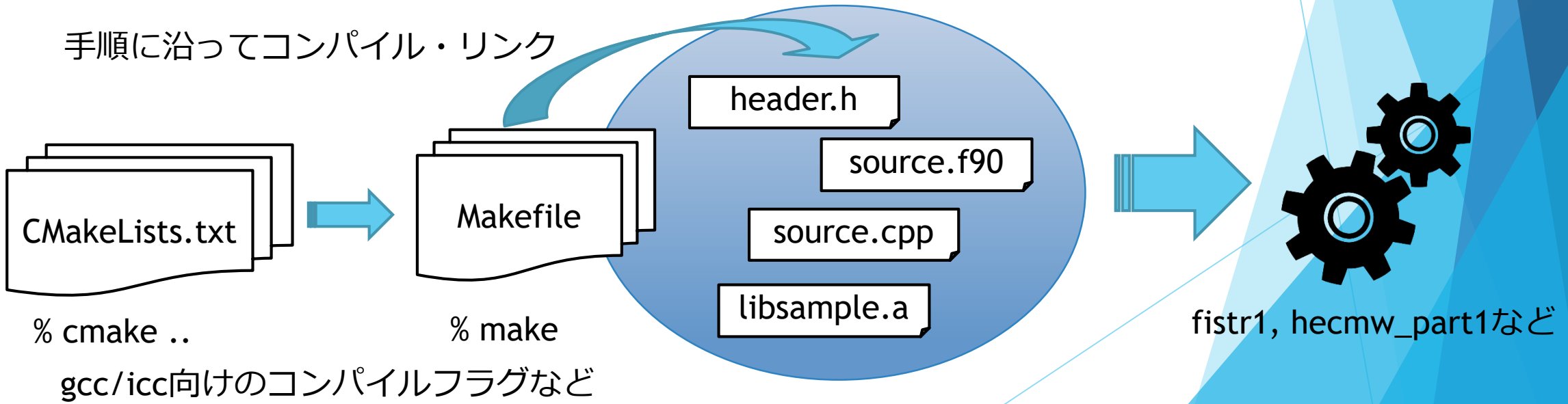
CMake https://cmake.org/



- CMakeとは、修正BSDライセンスで配布されているメタビルドツールです。
  - 手順書のための手順書を書くという意味で「メタ」です。
- CMake単体では、ビルドは出来ません。
- MakefileやVisual Studioのプロジェクトファイルを生成します。
- CMakeは、https://cmake.org/から入手出来るほか、多くのLinux系ディストリビューションで最初から提供されています。
- ビルド環境のみならず、テスト環境(CTest)、配付パッケージ構築ツール(CPack)なども内包しています。

CMakeとは (2)

- ビルド手順をマネージメントするためのツール
- コンパイラ・OSへの依存部分をCMakeが肩代わりしてくれます
- CMakeLists.txtにビルドに必要な情報を記述することで、Makefileを生成
- マルチ・クロスプラットフォームの細々とした部分をcmakeに任せられます
- テストもCMakeで提供されるツール(CTest)で実行できます



CMakeとは (3)

メリット

- CMakeによってビルド手順の簡素化できる
- マルチプラットフォームへの対応が容易になる
- 簡易テストも可能
- 配付パッケージの作成も可能
- CMakeの文法のみで完結できる
- 既に多数のプロジェクトで採用されている実績がある
  - elmer, HDF, metis, trilinosその他多数

デメリット

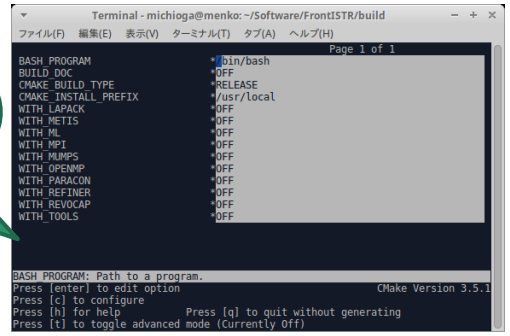
- CMakeLists.txtの書き方を覚えなければならない
- CMakeのマニュアルには、書き方の例が不足している
- cmakeがインストールされていない場合、cmakeをインストールする必要がある
- コンパイラの挙動を細かく制御したい場合、若干手間がかかる

CMakeの簡単な実行例

- CUIで

```
$ cd hello_cmake
$ mkdir build
$ cd build
$ cmake -G "MSYS Makefiles" ..
$ make
```

CUIベースのツールもあります

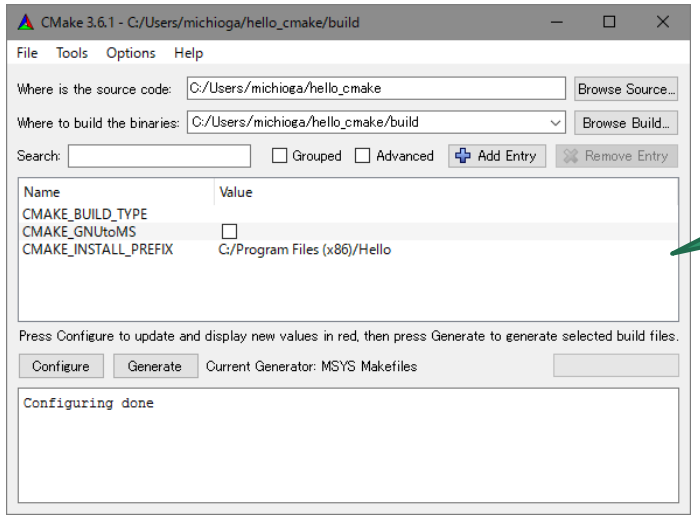


ccmake

Makefileを直接記述する方法と比べると、コンパイルフラグやライブラリ・ヘッダファイルの場所などを自動探索する機能が優れている。

- GUIで

```
$ cd hello_cmake
$ mkdir build
$ cd build
$ cmake-gui ..
```

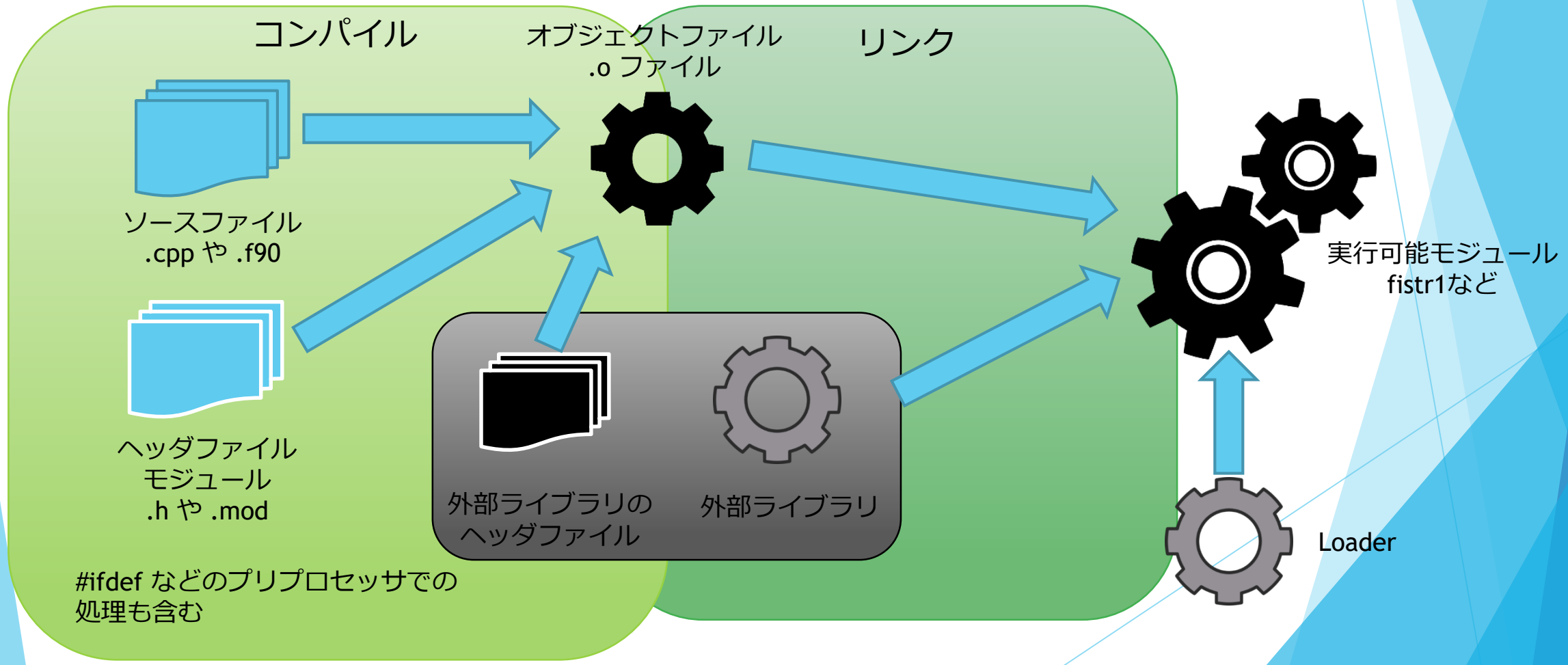


GUIツールが標準添付されています。

```
$ make
```

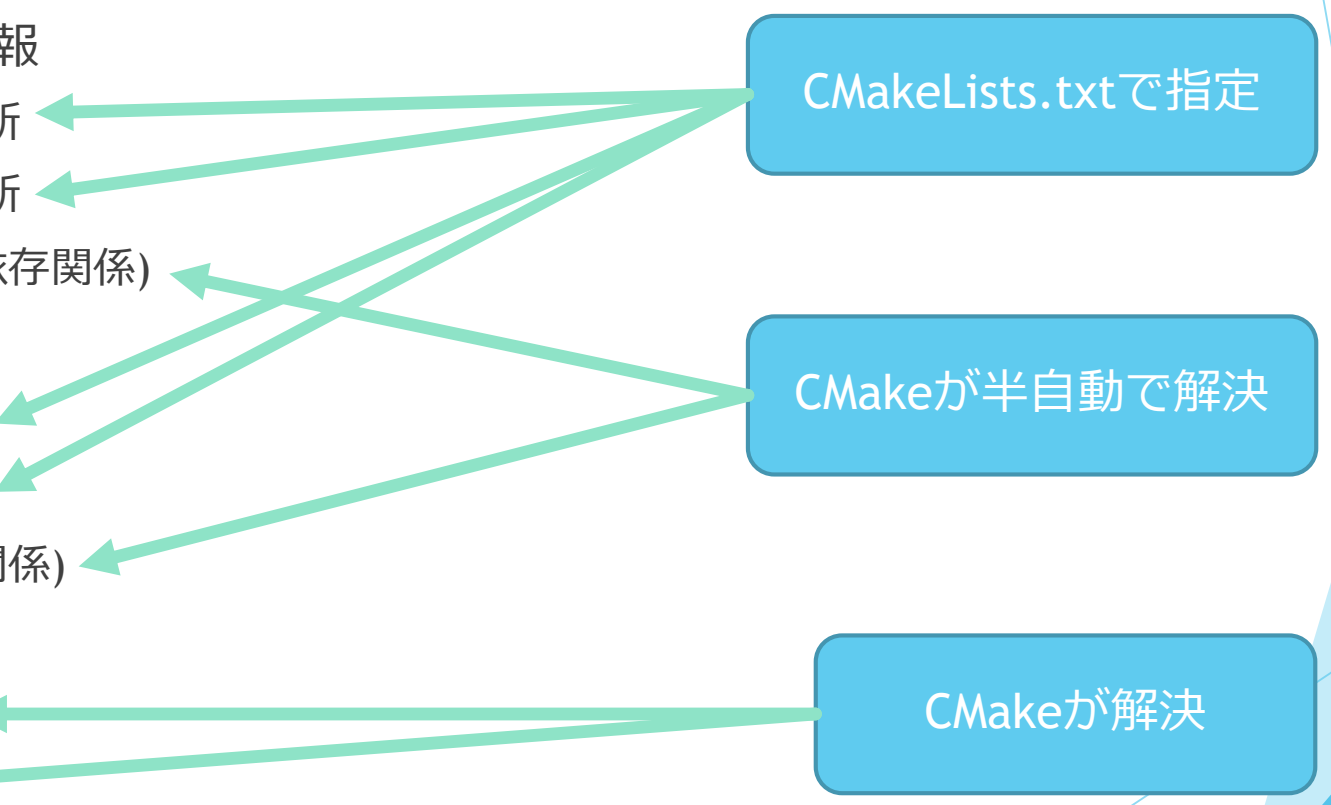
CMakeの基礎 (1)

実行可能モジュールを生成するには



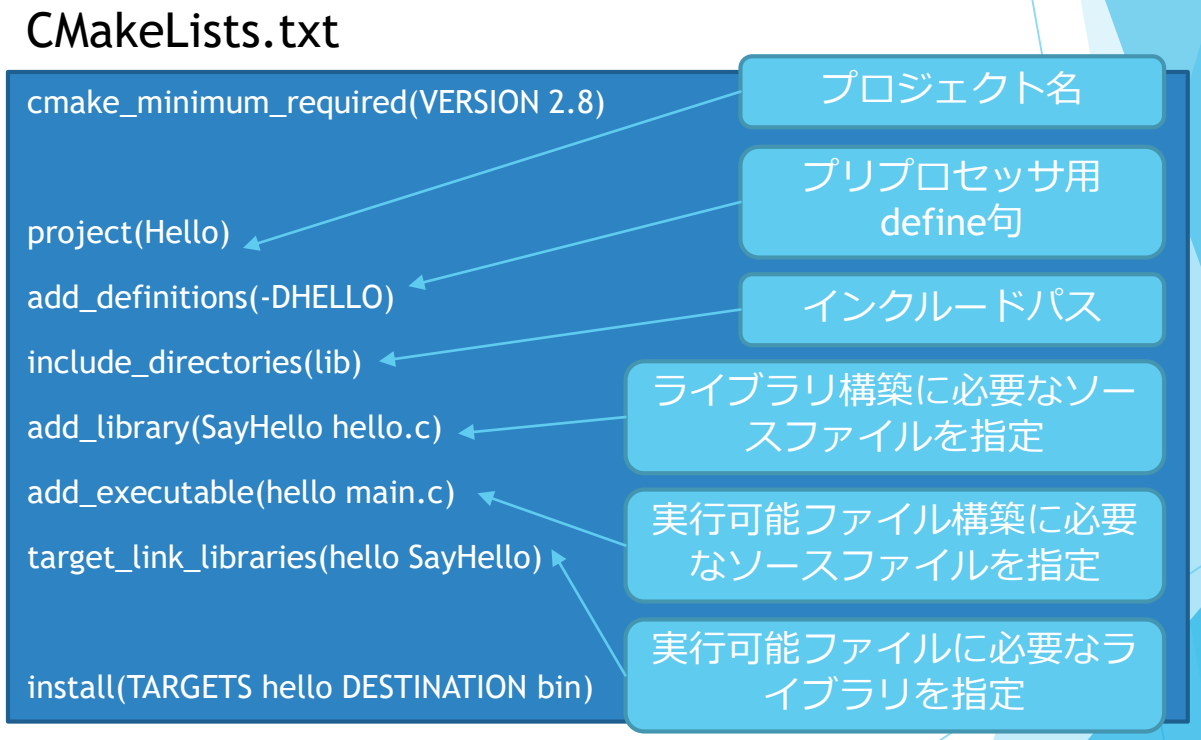
CMakeの基礎 (2)

- コンパイルに必要な情報
  - ソースファイルの場所
  - ヘッダファイルの場所
  - コンパイルの順番 (依存関係)
- リンクに必要な情報
  - オブジェクトの場所
  - ライブラリの場所
  - リンクの順番 (依存関係)
- その他
  - コンパイルフラグ
  - リンクフラグ



CMakeの基礎 (3)

- CMakeではMakefileは書きません。
- 代わりにCMakeLists.txtを書いていきます。
- MakefileはcmakeコマンドをCMakeを実行する事で自動生成させます。



CMakeの基礎 (4)

- リンクすべき自前ライブラリが一つあるHelloWorldを例にCMakeLists.txtの書き方を説明します。

src/main.c

lib/hello.h
lib/hello.c

CMakeの基礎 (5)

```
$ tree -a
.
├ CMakeLists.txt
├ lib
│   ├── CMakeLists.txt
│   ├── hello.c
│   └── hello.h
└ src
    ├── CMakeLists.txt
    └── main.c

2 directories, 6 files
```

```
cmake_minimum_required(VERSION 2.8)

project(Hello)

include_directories(lib)

add_subdirectory(lib)
add_subdirectory(src)
```

```
cmake_minimum_required(VERSION 2.8)

add_library(SayHello hello.c)
```

libSayHello.a

main.c

```
#include "hello.h"
#include <stdlib.h>

int main(int argc, char** argv)
{
    const char* hello = "hello cmake!";
#ifdef HELLO
    say_hello(hello);
#else
    #include <stdio.h>
    puts("hello");
#endif

    return EXIT_SUCCESS;
}
```

```
cmake_minimum_required(VERSION 2.8)

add_executable(hello main.c)
target_link_libraries(hello SayHello)

install(TARGETS hello DESTINATION bin)
```

hello

hello.c

```
#include "hello.h"
#include <stdio.h>

void say_hello(const char* msg)
{
    puts(msg);
}
```

hello.h

```
#pragma once
extern void say_hello(const char* msg);
```

hello worldをサンプルに

CMakeの基礎 (6)

- インクルードディレクトリを指定するには
  - include_directories(<directory name> <directory name>)
- ライブラリを生成するには
  - add_library(<TARGET_NAME> <library> <library>)
  - <TARGET_NAME>は、後にライブラリの場所を指定するのに利用できます。
  - ライブラリは絶対パスで指定されることに注意。
- 実行ファイルを作成するには
  - add_executable(<EXE_TARGET_NAME> <source file> <source file>)
  - <EXE_TARGET_NAME>は、後にインストール先を指定するのに利用できます。
- 実行ファイルにライブラリをリンクするには
  - target_link_libraries(<EXE_TARGET_NAME> <library> <library>)
- 定義分を追加するには
  - add_definitions(-D<DEFINE_NAME>)
  - プリプロセッサで利用する事ができます

-I に対応

-Lや-lに対応

-D に対応

CMakeの基礎 (7)

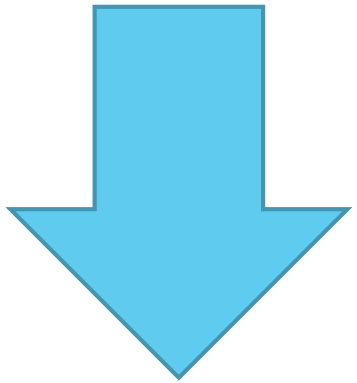
```
~/hello_cmake$ ls
CMakeLists.txt  lib  src
~/hello_cmake$ mkdir build
~/hello_cmake$ cd build
~/hello_cmake/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/michioga/hello_cmake/build
```

```
~/hello_cmake/build$ make
Scanning dependencies of target SayHello
[ 25%] Building C object lib/CMakeFiles/SayHello.dir/hello.c.o
[ 50%] Linking C static library libSayHello.a
[ 50%] Built target SayHello
Scanning dependencies of target hello
[ 75%] Building C object src/CMakeFiles/hello.dir/main.c.o
[100%] Linking C executable hello
[100%] Built target hello
~/hello_cmake/build$ src/hello
hello cmake!
```

FrontISTR・REVOCAP_Refiner/REVOCAP_Mesh への適用

```
% tar xvf FrontISTR.tar.gz
% cd FrontISTR
% mkdir build
% cd build
% cmake ..
% make
% make install
```

という手順でFrontISTR・REVOCAP_Refiner/REVOCAP_Mesh をインストール出来るようcmake を導入してみました。

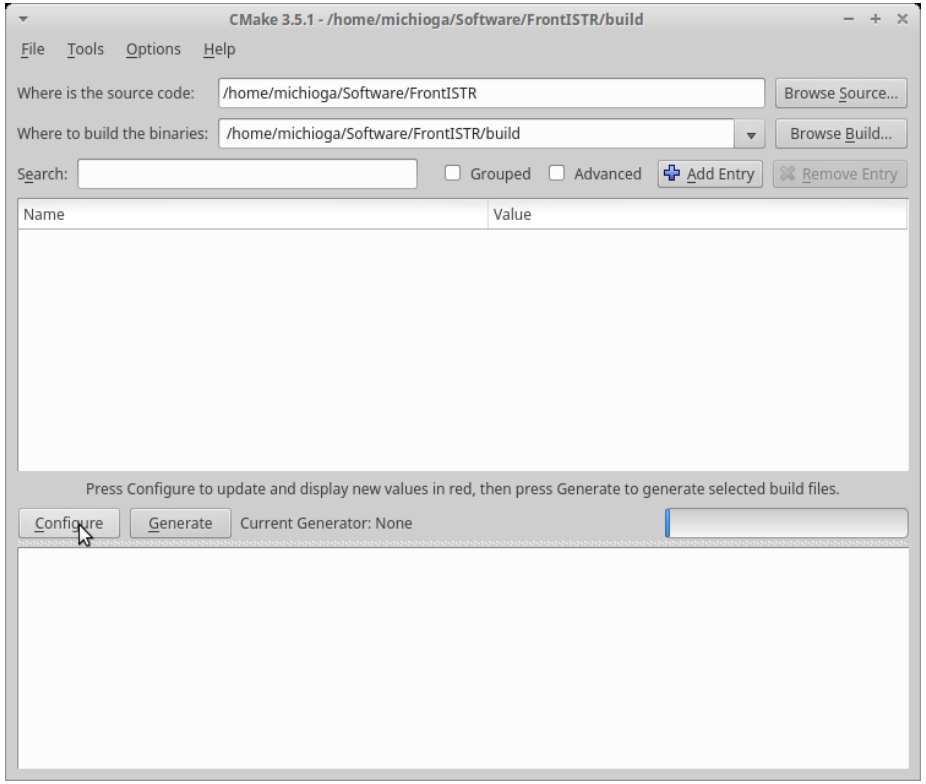


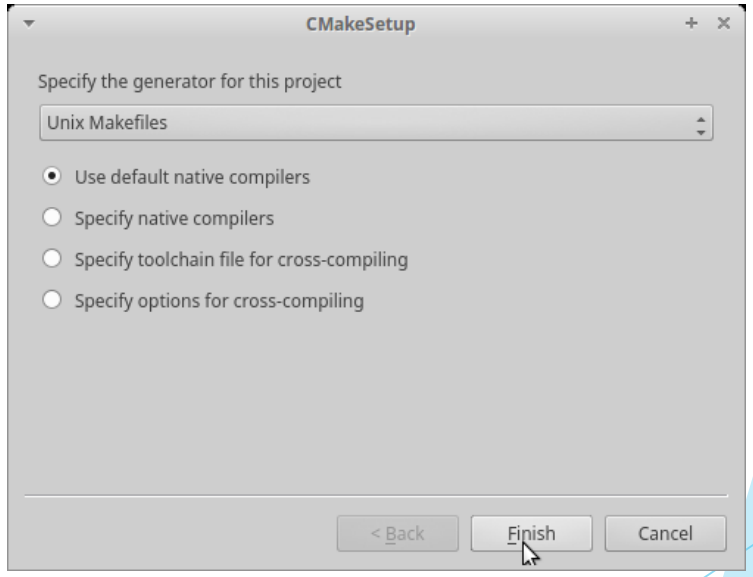
- 構築用のCMakeLists.txtを記述

- Makefile.confの編集が不要
- コンパイルの設定にGUIを利用

CMakeによるFrontISTRの構築 (1)

- OpenBLAS, metis-5, scalapack, trilinosは $HOME/local にインストール
- MUMPSは、FrontISTRを展開しているディレクトリと同じ場所でコンパイルとします。

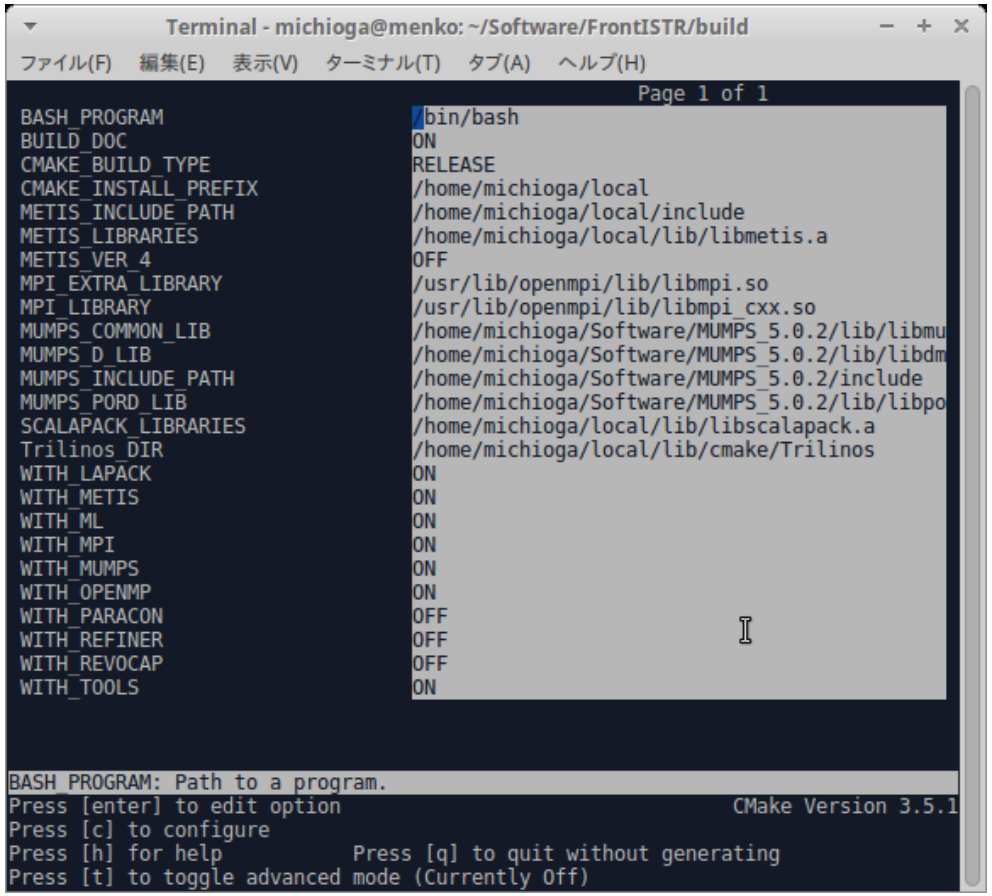




%cmake-gui

CMakeによるFrontISTRの構築 (2)

- CUI用のツールもあります。

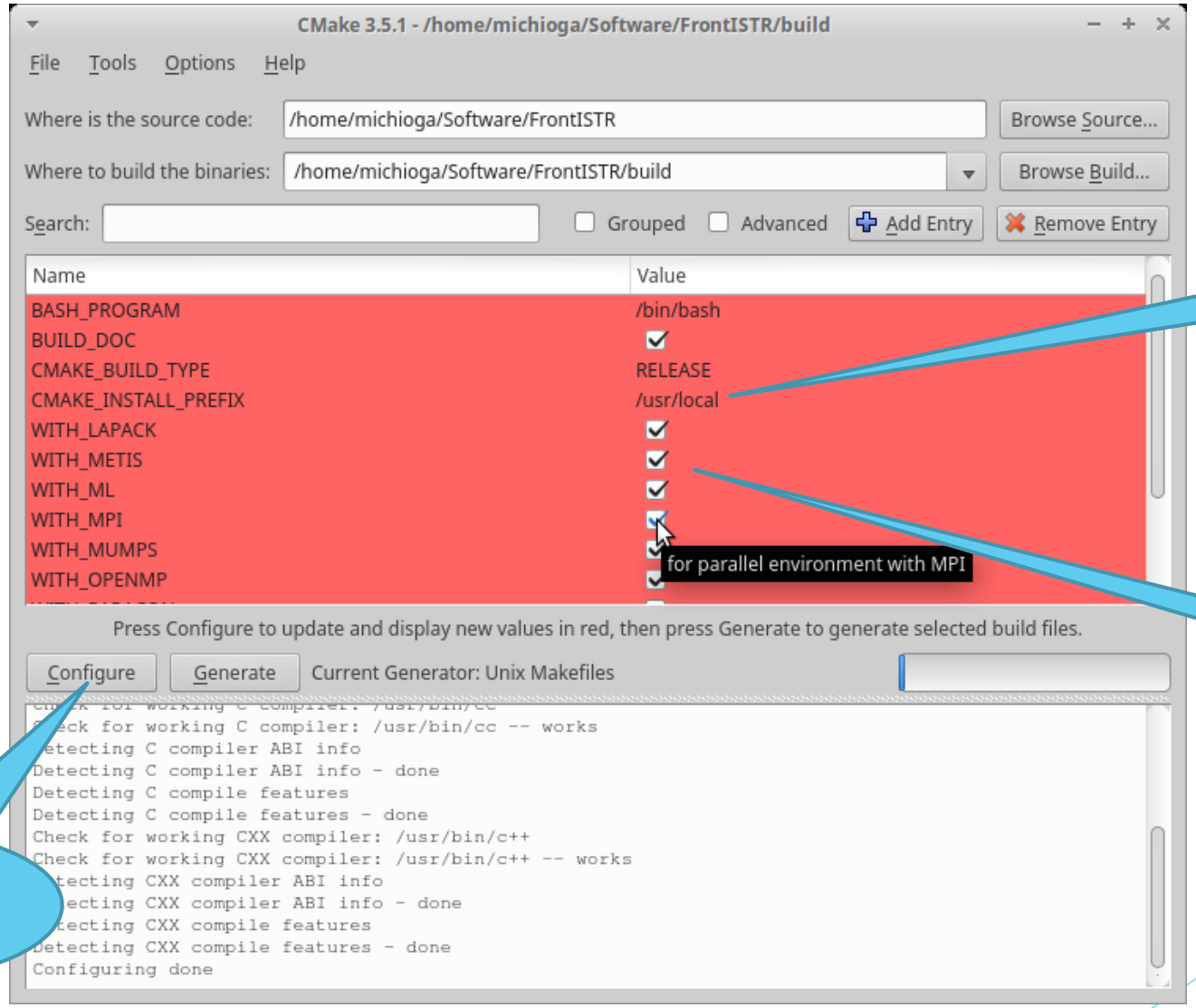



```
~/work/FrontISTR/build % cmake -DCMAKE_INSTALL_PREFIX=$HOME/local ..
```

直接指定も可能

%ccmake

CMakeによるFrontISTRの構築 (3)

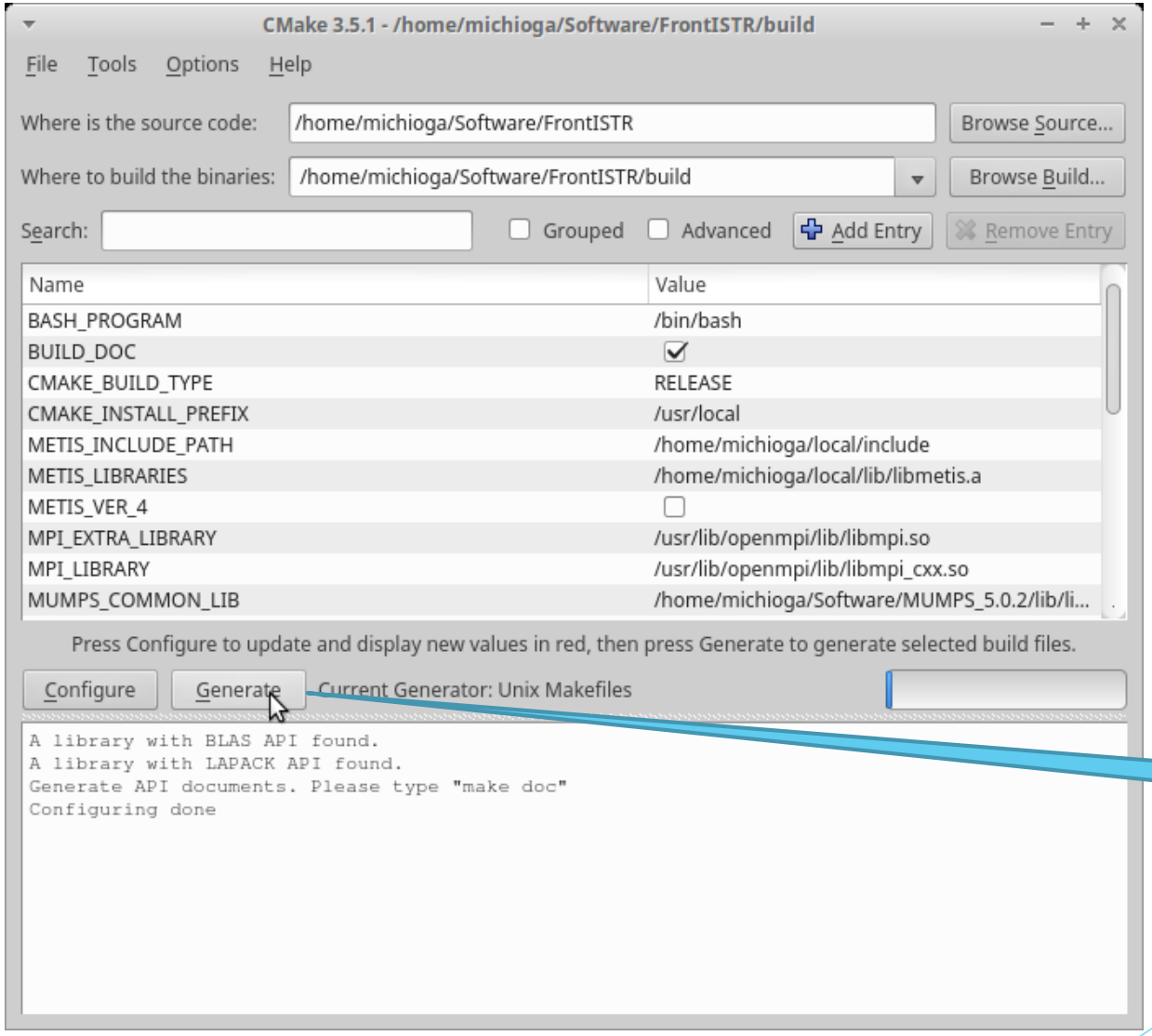


自動検出されなかったライブラリなどを指定

必要な機能をチェック

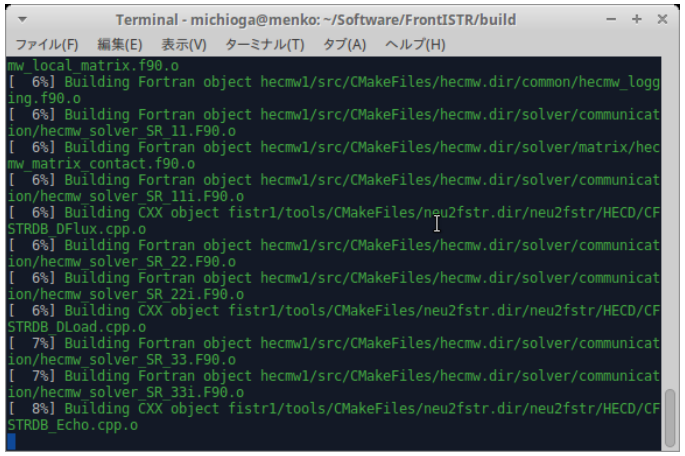
赤い表示が無くなるまで押す

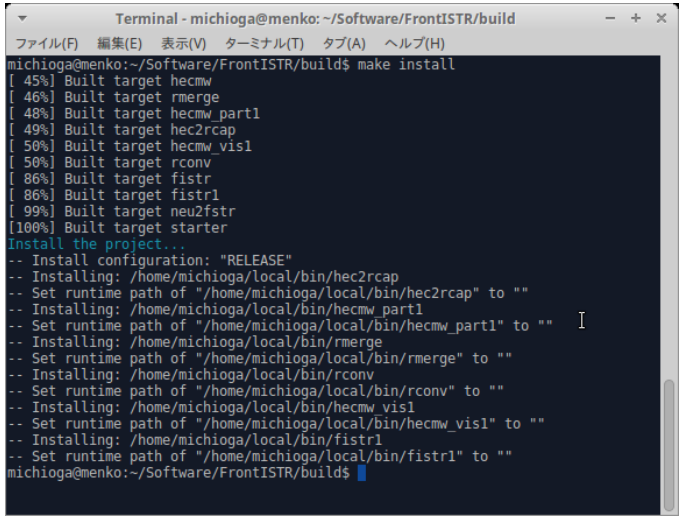
CMakeによるFrontISTRの構築 (4)



Makefileを生成

CMakeによるFrontISTRの構築 (5)





%make

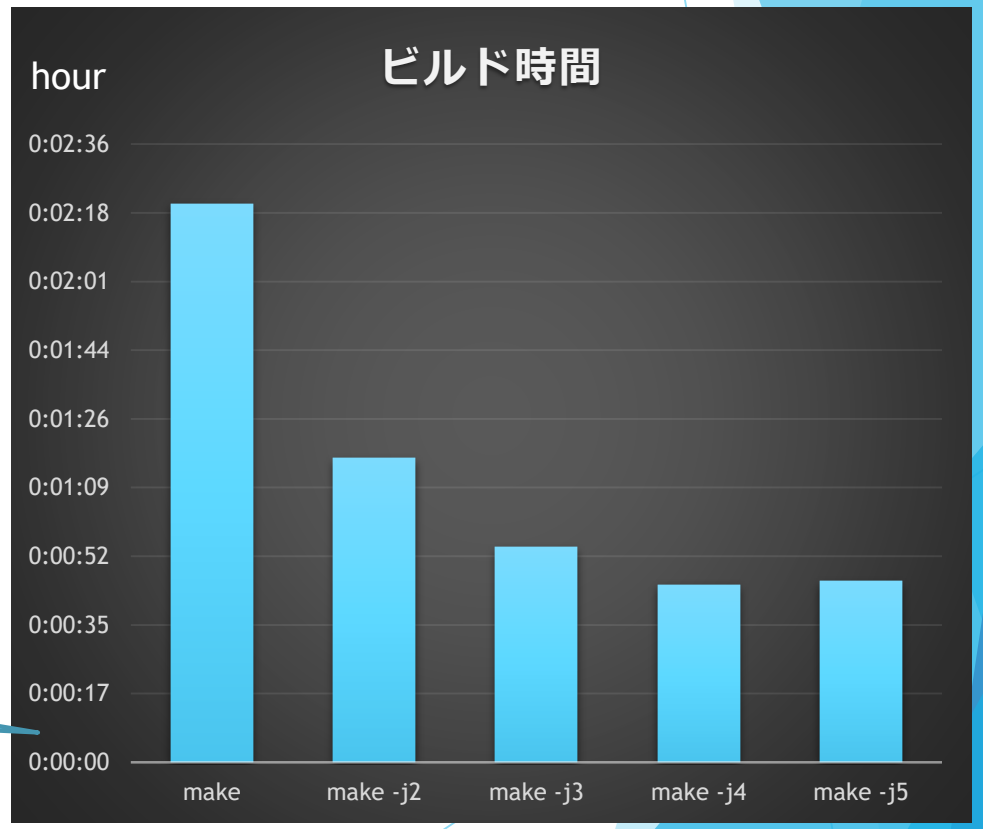
%make install

CMakeによるFrontISTRの構築 (6)

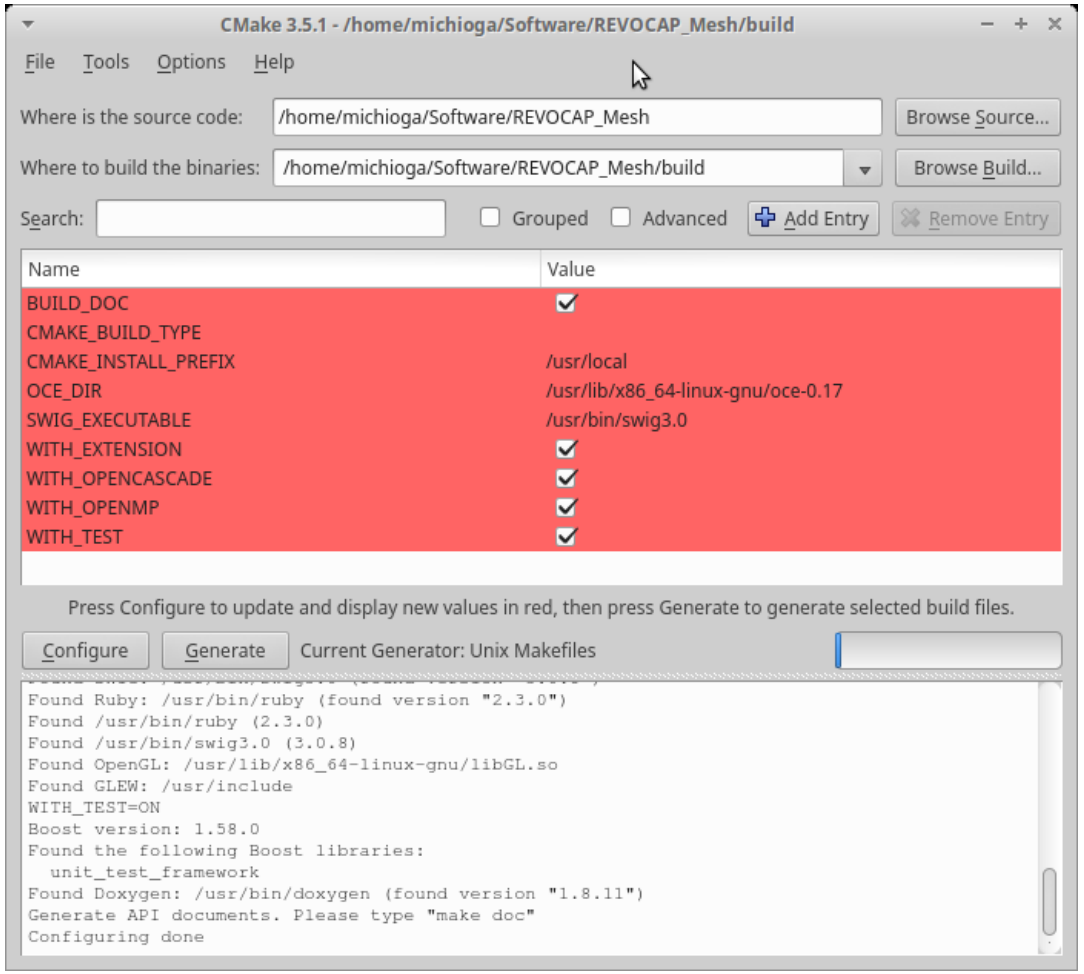
- 並列makeが可能
  - 依存関係のないファイルは、別コアでmake
  - 依存関係がある場合は、順序を守ってmake
  - それらを半自動的に行ってくれる

% make -j<n>　※ nは並列度

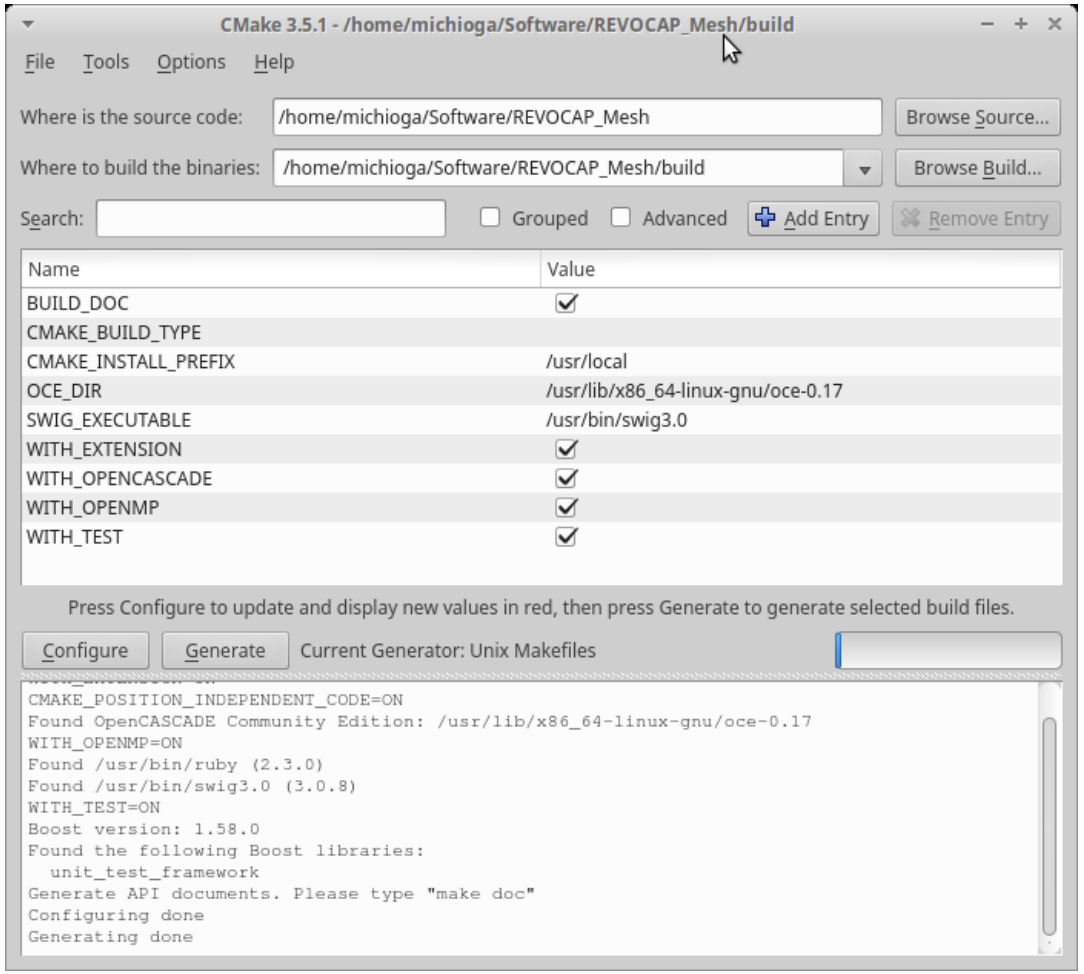
並列度を上げるとコンパイル時間が短縮される



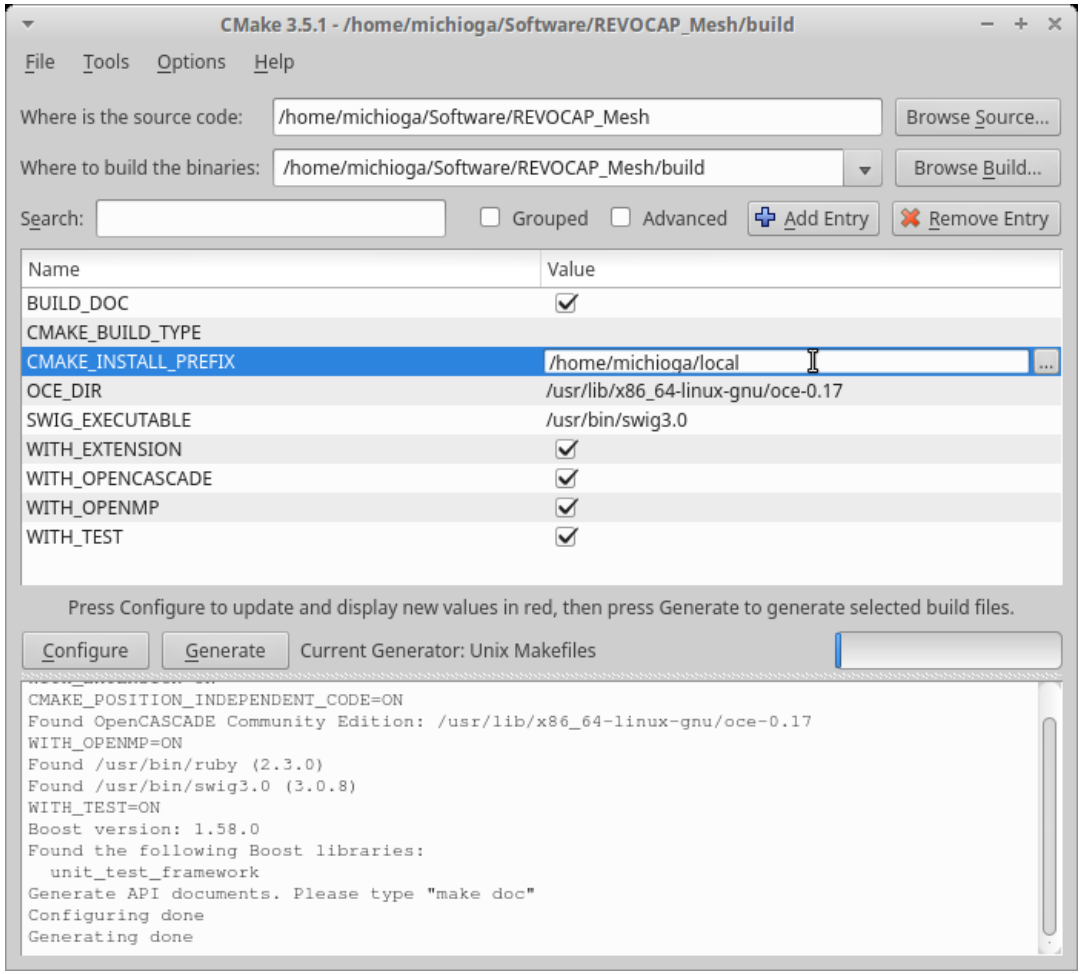
CMakeによる REVOCAP_Refiner/REVOCAP_Mesh(1)



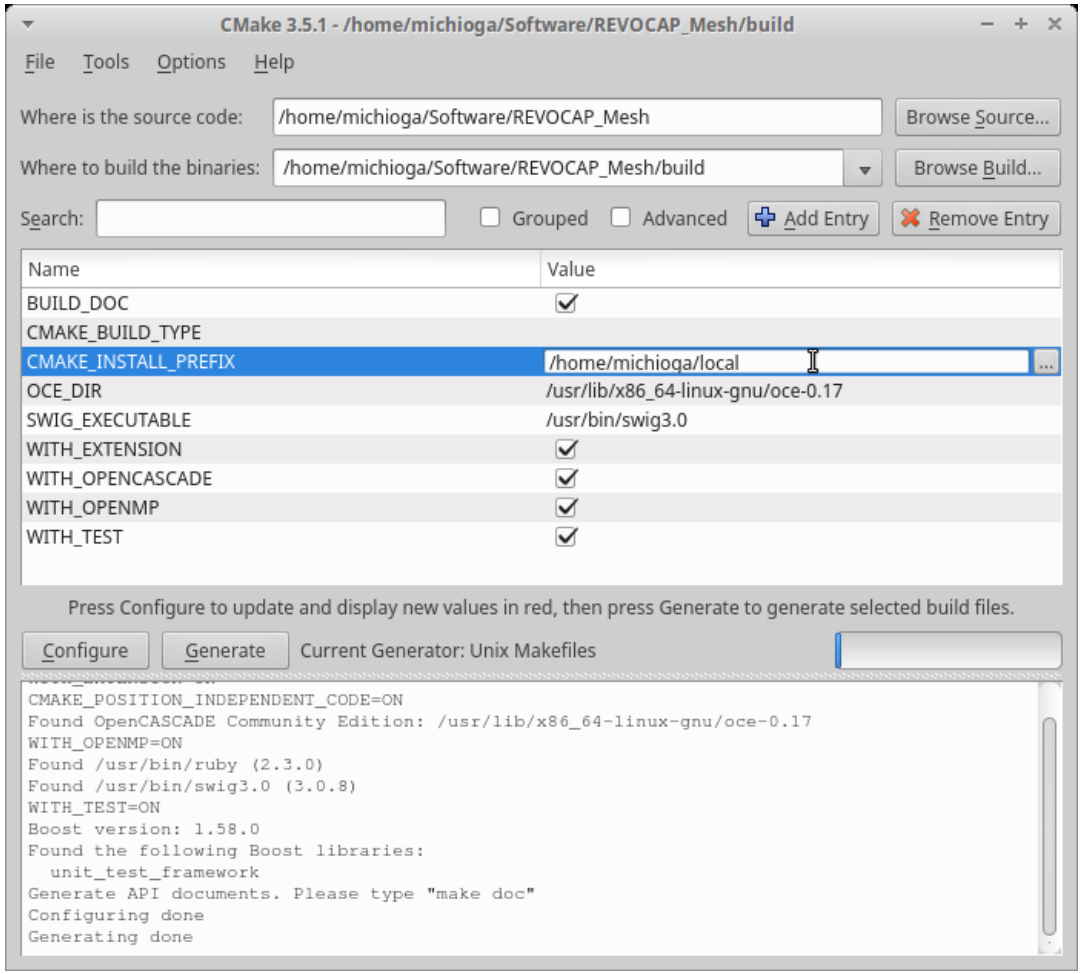
CMakeによる REVOCAP_Refiner/REVOCAP_Mesh(2)



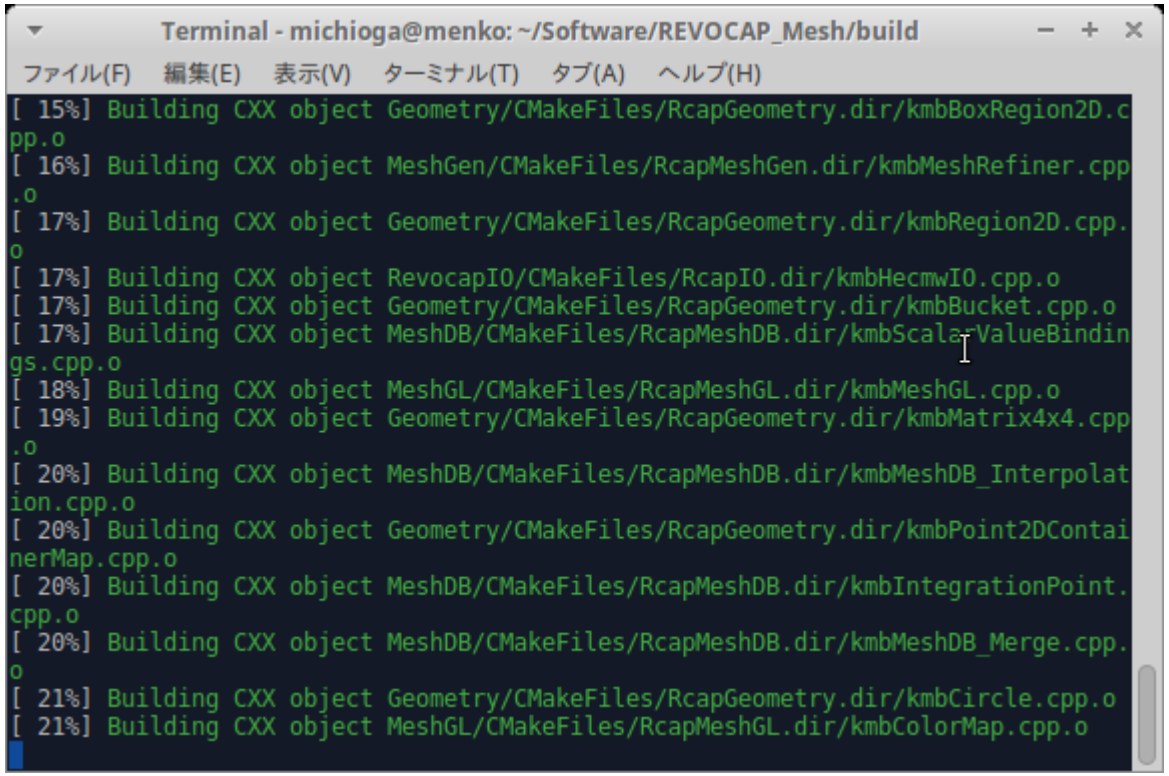
CMakeによる REVOCAP_Refiner/REVOCAP_Mesh(3)



CMakeによる REVOCAP_Refiner/REVOCAP_Mesh(3)



CMakeによる REVOCAP_Refiner/REVOCAP_Mesh(4)

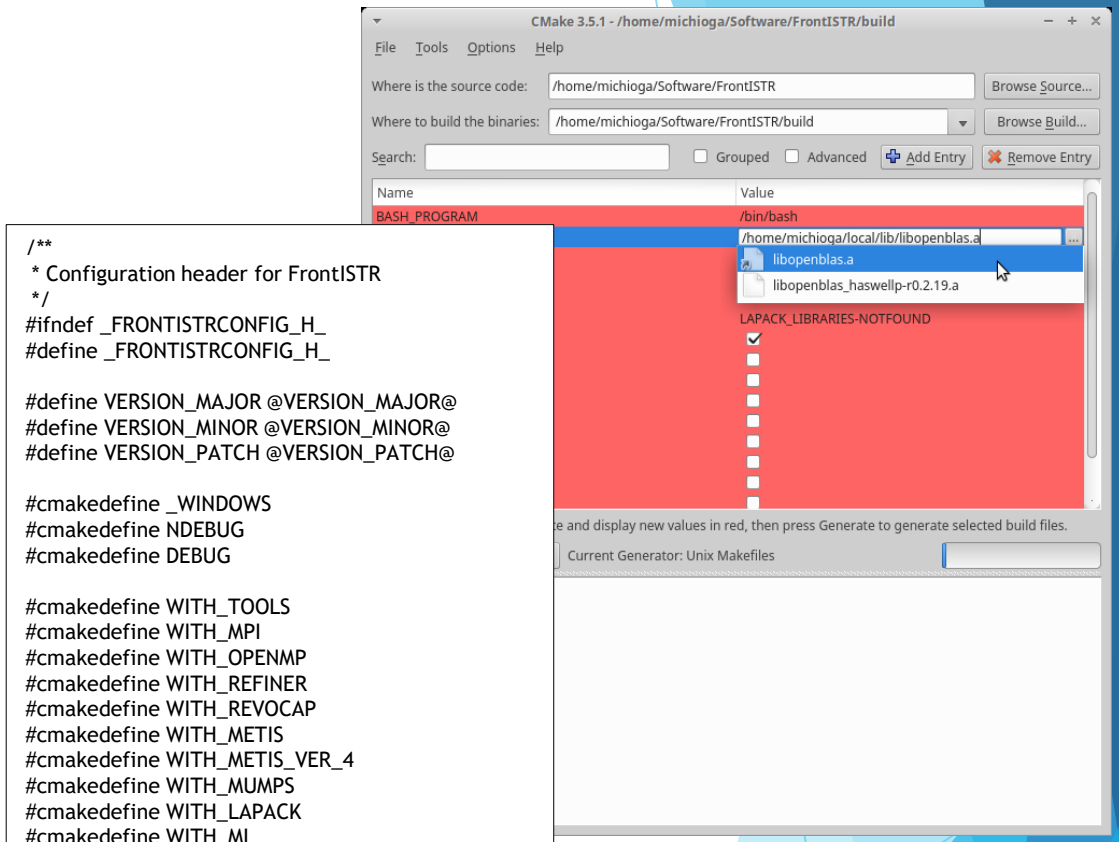


ここまでのまとめ

- FrontISTR・REVOCAP_Mesh/REVOCAP_Refinerをcmakeでビルド出来るようにしました。
- これまでよりもビルドが楽になります(かもしれません?)
- コンパイル時間が短縮されます。

幾つかのTips

- インストールするパスを指定する場合は
  - % cmake -DCMAKE_INSTALL_PREFIX=$HOME/local ..
- 外部ライブラリの探索を自動的にさせるためには
  - インストール先を/usr/local, /home/local(Ubuntu), /home/.local(CentOS)に設定するとよい。
  - 自動的に見つからない場合、ライブラリは絶対パスで指定。
- コンパイラを変更した時は「Configure」をし直す。
- config.hのようなものを生成する事ができます。
  - プログラム中でIncludeすることで、コンパイル時にDEFINEするのと同じ効果が得られます。




```
/**
 * Configuration header for FrontISTR
 */
#ifndef _FRONTISTRCONFIG_H_
#define _FRONTISTRCONFIG_H_

#define VERSION_MAJOR @VERSION_MAJOR@
#define VERSION_MINOR @VERSION_MINOR@
#define VERSION_PATCH @VERSION_PATCH@

#cmakedefine _WINDOWS
#cmakedefine NDEBUG
#cmakedefine DEBUG

#cmakedefine WITH_TOOLS
#cmakedefine WITH_MPI
#cmakedefine WITH_OPENMP
#cmakedefine WITH_REFINER
#cmakedefine WITH_REVOCAP
#cmakedefine WITH_METIS
#cmakedefine WITH_METIS_VER_4
#cmakedefine WITH_MUMPS
#cmakedefine WITH_LAPACK
#cmakedefine WITH_ML
#cmakedefine WITH_PARMETIS
#cmakedefine WITH_MKL
#cmakedefine WITH_PARACON
#cmakedefine WITH_PARADISO

#cmakedefine PARA_CONTACT
#cmakedefine HECMW_SERIAL
#cmakedefine HECMW_WITH_REFINER
#cmakedefine HECMW_WITH_METIS
#cmakedefine HECMW_PART_WITH_METIS
#cmakedefine HECMW_METIS_VER
@HECMW_METIS_VER@
#cmakedefine HECMW_WITH_ML

#endif /* _FRONTISTRCONFIG_H_ */
```

REVOCAP_Refiner/REVOCAP_Meshの構築および、テストモジュールCTestについて

- REVOCAP_Refiner⇒CMakeでビルドできるようになったので、ソースからビルドしてください。

- REVOCAP_Mesh（REVOCAP_PrePostのメッシュ処理カーネル部）⇒CMakeでビルドできるようになりましたが、依存するライブラリが多いので、特にWindows環境の場合に、ご自分でやることはあまりお勧めしません。
REVOCAP_PrePostで利用する実行体はインストーラーで提供します。

REVOCAP_Refinerの構築

- FrontISTR 等に組み込まれるメッシュ細分化ツールのライブラリをビルドします。

```
% tar xvf REVOCAP_Refiner-X.Y.Z.tar.gz
% cd REVOCAP_Refiner-X.Y.Z
% mkdir build
% cd build
% cmake -DWITH_OPENCASCADE=OFF -DWITH_TEST=OFF ..
% make
% make install
```

- 作成されたlibRcapRefiner.aは今までと同様に使えます。
- このバージョンからFortran用のモジュールrcaprefiner.modも作成しています。

REVOCAP_Mesh の構築

- REVOCAP_PrePostのメッシュ処理カーネル部のライブラリを作成します。
  - 必須のもの：swig、glew
  - あるとよいもの：OpenCASCADE、boost

```
% tar xvf REVOCAP_Mesh-X.Y.Z.tar.gz
% cd REVOCAP_Mesh-X.Y.Z
% mkdir build
% cd build
% cmake -DWITH_OPENCASCADE=OFF -DWITH_TEST=OFF ..
% make
```

- 作成されたライブラリ
  - RevocapMesh.so、RevocapMeshGen.so、RevocapIO.so、RevocapGL.so、RevocapShape.so
  - REVOCAP_PrePostのディレクトリにコピーして使います。（これはまたの機会に）

CTestとは

- CMake に付属するテスト実行の支援ツールです。
- CMakeLists.txtにテストのためのコマンドを記述しておけば、以下のコマンドでテストが実行できます。
  - $ ctest
  - $ make test
- テストそのものを記述するものではありません。
  - その目的では Boost Test や Google Test、Junit などが良く使われています。

FrontISTRのテスト

- FrontISTRにはもともと examples ディレクトリにいくつかテスト用の例題が含まれています。
  - シェルスクリプトで実行するように作られています。
- 今までのテストでは、結果が妥当かどうかは人間が判断していました。
- これらを CTestから呼び出せるようにしました。

テストの実行方法

- CMake を適用した Makefile で FrontISTR の実行体を作成した後で実行する。

```
% tar xvf FrontISTR.tar.gz
% cd FrontISTR
% mkdir build
% cd build
% cmake ..
% make
% make install
% make test
```

テストの実行結果

- Start 1: Static_exA_Test
- 1/23 Test #1: Static_exA_Test ..... Passed 0.65 sec
- 以下、examplesにある23個（すべてではない）のテストを実行します。

- テスト結果の詳細はTesting/Temporaryディレクトリの中にあります。

CMakeLists.txtでの記述

- もとのCMakeLists.txt

```
enable_testing()
add_subdirectory(examples)
```

- ExamplesディレクトリのCMakeLists.txt
  - Add_test コマンドでテストを追加します。
  - NAME：識別子
  - COMMAND：テストプログラム起動命令
  - WORKING_DIRECTORY：作業時のディレクトリ

```
add_test(
 NAME Static_exA_Test
 COMMAND ruby ./test_FrontISTR.rb ${Fistr_BINARY_DIR}/src static/exA
 WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
)
```

テスト判定ルーチン

- あらかじめ、正しく計算できた時の 0.log を正解データとして取っておきます（一般ユーザには正解データは配布されます）。
- 実行時に出力された0.logをテスト判定ルーチンの入力データとして、正解データと比較します。
- テスト判定のロジックはrubyで書きました。
- 比較しているのは、0.logに出力される Global Summary および @Element の最大最小値です。

- 動解析（Dynamic）のexamplesは0.logが非常に大きいので、最後のステップの結果だけを比較することにしています。

より良いテストのために

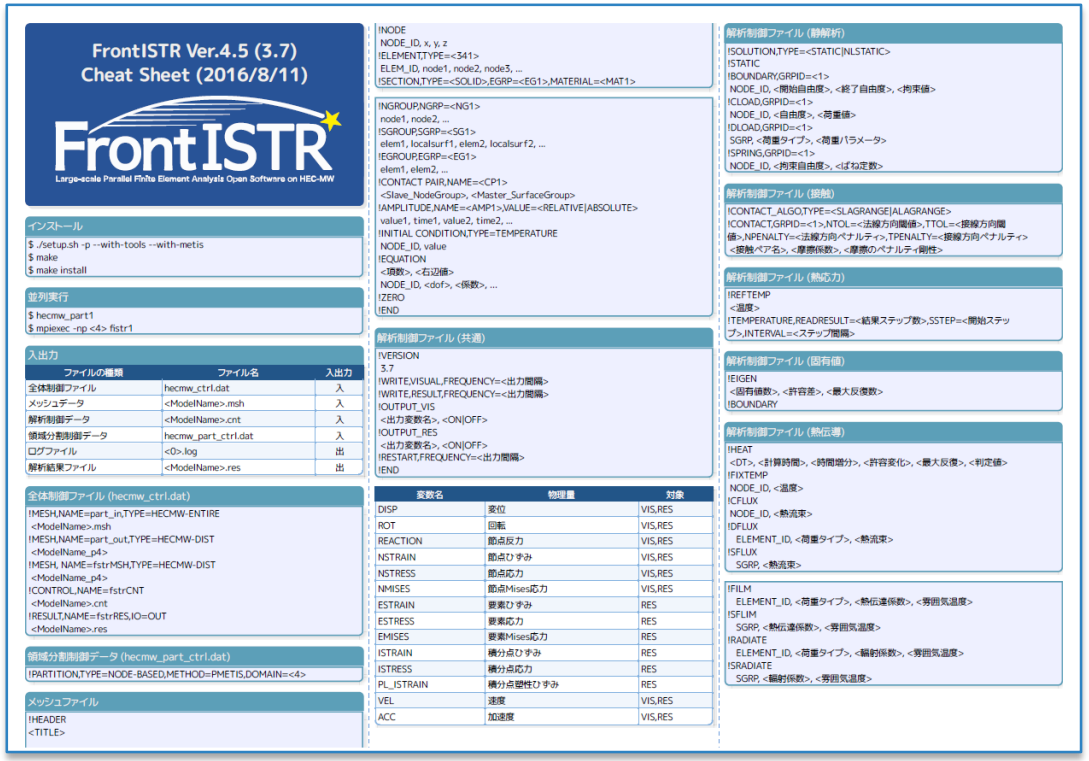
- テスト合格の判定基準をどうすればよいか
  - 今は適当な閾値以下であれば合格としています。
  - 線形と非線形、静解析と動解析で閾値は変えるほうが望ましいと思いますが、まだそこまではできていません。
- 理論解があるようなテスト問題と比較するほうが良いと思います。
- 大規模問題や、並列数による挙動の違いの検査はこのテストには含まれていません。

REVOCAPのテスト

- ライブラリの関数単位のテストをいくつか準備しています。
  - 幾何計算に関するもの。（面積の計算、距離の計算など）
  - 入出力ルーチンに関するもの。（格子ファイルが正しく読み込まれているか）
- テストの実行方法はFrontISTRと同じ仕組みを導入しました。
  - Cmake + CTest でテストができます。
  - C++でテスト判定を行うのでBoost Testを使っています。

おまけ

- FrontISTRの機能がすぐに分かるチートシート配布中
  - 1枚にFrontISTRの機能をまとめました。



FrontISTR Ver.4.5 (3.7)
Cheat Sheet (2016/8/11)

FrontISTR
Large-scale Parallel Finite Element Analysis Open Software on HEC-MW

インストール

```
$ ./setup.sh -p --with-tools --with-metis
$ make
$ make install
```

並列実行

```
$ hecmw_part1
$ mpiexec -np <4> fistr1
```

入出力

ファイルの種類	ファイル名	入出力
全体制御ファイル	hecmw_ctrl.dat	入
メッシュデータ	<ModelName>.msh	入
解析制御データ	<ModelName>.cnt	入
領域分割制御データ	hecmw_part_ctrl.dat	入
ログファイル	<0>.log	出
解析結果ファイル	<ModelName>.res	出

全体制御ファイル (hecmw_ctrl.dat)

```
!MESH,NAME=part_in,TYPE=HECMW-ENTIRE
 <ModelName>.msh
!MESH,NAME=part_out,TYPE=HECMW-DIST
 <ModelName_p4>
!MESH, NAME=fstrMSH,TYPE=HECMW-DIST
 <ModelName_p4>
!CONTROL,NAME=fstrCNT
 <ModelName>.cnt
!RESULT,NAME=fstrRES,IO=OUT
 <ModelName>.res
```

領域分割制御データ (hecmw_part_ctrl.dat)

```
!PARTITION,TYPE=NODE-BASED,METHOD=PMETIS,DOMAIN=<4>
```

メッシュファイル

```
!HEADER
<TITLE>
!NODE
 NODE_ID, x, y, z
!ELEMENT,TYPE=<341>
 ELEM_ID, node1, node2, node3, ...
!SECTION,TYPE=<SOLID>,EGRP=<EG1>,MATERIAL=<MAT1>
```

```
!NGROUP,NGRP=<NG1>
 node1, node2, ...
!SGROUP,SGRP=<SG1>
 elem1, localsurf1, elem2, localsurf2, ...
!EGROUP,EGRP=<EG1>
 elem1, elem2, ...
!CONTACT PAIR,NAME=<CP1>
 <Slave_NodeGroup>, <Master_SurfaceGroup>
!AMPLITUDE,NAME=<AMP1>,VALUE=<RELATIVE|ABSOLUTE>
 value1, time1, value2, time2, ...
!INITIAL CONDITION,TYPE=TEMPERATURE
 NODE_ID, value
!EQUATION
 <項数>, <右辺値>
 NODE_ID, <dof>, <係数>, ...
!ZERO
!END
```

解析制御ファイル (共通)

```
!VERSION
 3.7
!WRITE,VISUAL,FREQUENCY=<出力間隔>
!WRITE,RESULT,FREQUENCY=<出力間隔>
!OUTPUT_VIS
 <出力変数名>, <ON|OFF>
!OUTPUT_RES
 <出力変数名>, <ON|OFF>
!RESTART,FREQUENCY=<出力間隔>
!END
```

変数名	物理量	対象
DISP	変位	VIS,RES
ROT	回転	VIS,RES
REACTION	節点反力	VIS,RES
NSTRAIN	節点ひずみ	VIS,RES
NSTRESS	節点応力	VIS,RES
NMISES	節点Mises応力	VIS,RES
ESTRAIN	要素ひずみ	RES
ESTRESS	要素応力	RES
EMISES	要素Mises応力	RES
ISTRAIN	積分点ひずみ	RES
ISTRESS	積分点応力	RES
PL_ISTRAIN	積分点塑性ひずみ	RES
VEL	速度	VIS,RES
ACC	加速度	VIS,RES

解析制御ファイル (静解析)

```
!SOLUTION,TYPE=<STATIC|NLSTATIC>
!STATIC
!BOUNDARY,GRPID=<1>
 NODE_ID, <開始自由度>, <終了自由度>, <拘束値>
!CLOAD,GRPID=<1>
 NODE_ID, <自由度>, <荷重値>
!DLOAD,GRPID=<1>
 SGRP, <荷重タイプ>, <荷重パラメータ>
!SPRING,GRPID=<1>
 NODE_ID, <拘束自由度>, <ばね定数>
```

解析制御ファイル (接触)

```
!CONTACT_ALGO,TYPE=<SLAGRANGE|ALAGRANGE>
!CONTACT,GRPID=<1>,NTOL=<法線方向閾値>,TTOL=<接線方向閾
値>,NPENALTY=<法線方向ペナルティ>,TPENALTY=<接線方向ペナルティ>
 <接触ペア名>, <摩擦係数>, <摩擦のペナルティ剛性>
```

解析制御ファイル (熱応力)

```
!REFTEMP
 <温度>
!TEMPERATURE,READRESULT=<結果ステップ数>,SSTEP=<開始ステッ
プ>,INTERVAL=<ステップ間隔>
```

解析制御ファイル (固有値)

```
!EIGEN
 <固有値数>, <許容差>, <最大反復数>
!BOUNDARY
```

解析制御ファイル (熱伝導)

```
!HEAT
 <DT>, <計算時間>, <時間増分>, <許容変化>, <最大反復>, <判定値>
!FIXTEMP
 NODE_ID, <温度>
!CFLUX
 NODE_ID, <熱流束>
!DFLUX
  ELEMENT_ID, <荷重タイプ>, <熱流束>
!SFLUX
  SGRP, <熱流束>
```

```
!FILM
  ELEMENT_ID, <荷重タイプ>, <熱伝達係数>, <雰囲気温度>
!SFLIM
  SGRP, <熱伝達係数>, <雰囲気温度>
!RADIATE
  ELEMENT_ID, <荷重タイプ>, <輻射係数>, <雰囲気温度>
!SRADIATE
  SGRP, <輻射係数>, <雰囲気温度>
```

http://www.multi.k.u-tokyo.ac.jp/FrontISTR/

の 「FrontISTRリザーバ」からダウンロードできます。