Cancer Imaging Phenomics Toolkit (CaPTk): Technical Overview

Sarthak Pati
Lead Software Developer

Center for Biomedical Image Computing & Analytics (CBICA)
Introduction

• A brief history of CaPTk

• Exploring the dependencies + guts of CaPTk

• Integrating “standalone” applications
  o MATLAB
  o Python

• Integrating native C++ applications
What does one get out of it?

• Knowledge of integrating their application into CaPTk

• Leverage existing bodies of validated APIs/algorithms

• Improved visibility and increased citations of your application

• Easier integration with clinical workflow

• Become part of a larger scientific community
A Brief History

The Center for Biomedical Image Computing and Analytics (www.med.upenn.edu/cbica) consists of multiple Principal Investigators, Post Docs, Data Analysts and developers; all producing state-of-the-art algorithms published in high impact publications.

With such a diverse group, development skills vary considerably and finding a common mechanism for distribution is difficult.

Imperative to find a unified distribution pipeline that would enable non-computational researchers to leverage our research.
• Needed a way to decrease the learning curve for everyone to use our computational algorithms

• Existing packages (FSLView, MIPAV, 3DSlicer) had some deficiencies which we wanted to address with our package:
  - FSLView and MIPAV were not extensible with custom algorithms
  - 3DSlicer had a steep learning curve for our collaborators (clinicians and researchers alike)
  - 3DSlicer also did not provide an easy mechanism to extend the basic UI controls which were needed for our algorithms; for instance initializing a tumor point (coordinate + radius information).
• With the U24 Grant (funded by NIH ITCR), it became possible to design a common framework for distribution.

• With these points in mind, I would like to present the technical details of the Cancer Imaging Phenomics Toolkit (CaPTk) with the idea to contribute the work we have done to the greater scientific community.
C++

• Efficient
• Truly cross-platform
• Natively supported by **ALL** major operating systems
• Code can be distributed as a stand-alone package
• Community and industry driven libraries/toolkits/packages for medical imaging
CMake

• Configuration tool for C++ projects
• Cross-platform
• Integrated Testing and Packaging environment
• Supported by virtually all IDEs, making development easier

https://cmake.org
Insight Toolkit (ITK)

- Community and industry driven package designed specifically for medical imaging
- Open source
- Well documented and validated
- Provides solid building blocks for developers via native C++ and Python wrapped calls
- Gives low and high-level access for developers, thereby providing a good starting point for developers with all skill levels

https://itk.org
Qt

• Industry-leading open source user interface design toolkit
• Cross-platform
• Support ranging from IOT to server class machines
• Flexible and can be used from native C++ and Python wrappings
Visualization Toolkit (VTK)

- Industry-leading visualization framework
- Cross-platform
- Supports a wide variety of visualization algorithms as well as advanced modeling techniques via easy-to-use abstract classes
- Provides methods to perform 2D and 3D visualization

https://www.vtk.org/
Open Source Computer Vision Library (OpenCV)

• Industry-leading library for computer vision and machine learning
• Cross-platform
• Designed for computational efficiency and with a strong focus on real-time applications
• Wide ranging applications from geospatial image analysis, robotics, interactive art, medical imaging

https://opencv.org/
CBICA Toolkit

• Higher level functions/classes to take care of basic software design needs
  o Command Line Interface design + parsing
  o Logging with auto-generated time-stamp
  o CSV Parsing
  o System-level file/folder checks (file extensions, deleting folder, etc.)
  o Basic Statistics (ROC, etc.)

• Cross-platform and continuous validation
CBICA Toolkit - Continued

• Functions wrapped around ITK and OpenCV data structures to provide common image processing tasks
  o Single line image reading/writing for ITK with sanity checks
  o DTI scalar computation
  o Image structure manipulation (vectorize, etc.)

• Available via GitHub as a separate module!
Internals of the Code

- The root CMake configuration file
- Handles the project setup, looks for dependencies, structures the package overall

${\text{CaPTk\_Source}}$
CMakeLists.txt
/src
/data
/docs
...

 Penn
Division of Biomedical Imaging Analysis
Sets up the main executable and the path(s)

Creates the main executable

Location of all the source code, which is broken down to the following: applications, cbica_toolkit and gui
${\text{CaPTk\_Source}}$

CMakeLists.txt

/src

/applications

/cbica_toolkit

/gui

• /applications: all the code for the native and stand-alone applications that are part of CaPTk

• /cbica_toolkit: CBICA Toolkit

• /gui: user interface files that handle the visualizer, dialogs, etc.
Internals of the Code – Continued

- **fMainWindow** constructs the main UI of CaPTk
- Construction of all the menus, image viewer, image and mask loading/saving
- Handles communication between user and applications
- Communication from the main command line (CAPTk.cpp) uses file names

```
$ {{CaPTk_Source}}
CMakeLists.txt
/src
   /gui
fMainWindow.h
fMainWindow.cpp
ui_fMainWindow.h
```
• **fMainWindow** components (h, cpp and ui) are to be edited for adding any application

• **UI file**: declaring the menu items
  
  ```cpp
  ui_fMainWindow::setupUi( )
  ```

• **Header file**: declaring the functions and slots

• **Implementation file**: defining the functions

```bash
$ {CaPTk_Source}
CMakeLists.txt
/src
  /gui

fMainWindow.h
fMainWindow.cpp
ui_fMainWindow.h
```
SlicerManager stores the image data as \texttt{itk::Image} \texttt{vtk::Image} data structures and keep both in sync.

In addition, it stores all the other metadata of the image (file path, seed points, etc.) that can be called from \texttt{fMainWindow} using the member variable \texttt{mSlicerManagers}. 

\begin{quote}
\texttt{$\{\text{CaPTk\_Source}\}$}
\texttt{CMakeLists.txt}
\texttt{/src}
\texttt{/gui}
\texttt{SlicerManager.h}
\texttt{SlicerManager.cpp}
\end{quote}
What are Standalone Applications?

These are independent binaries that can “exec-ed” from the command line or terminal.

• Their dependencies can either be part of the distribution or can be downloaded from an external site.

• The simplest integration of an application into CaPTk.
Converting a MATLAB script (or a collection of scripts) into a single binary using MCC (using LIBRA as example):

```matlab
[sourceDir,~,~] = fileparts( mfilename('fullpath') );
addpath(sourceDir);
addpath(fullfile(sourceDir,'Code'));
addpath(fullfile(sourceDir,'Model'));
modelDir=fullfile(sourceDir,'Model');
```
Converting a MATLAB script (or a collection of scripts) into a single binary using MCC (using LIBRA as example):

```matlab
[sourceDir,~,~] = fileparts( mfilename('fullpath') );
addpath(sourceDir);
addpath(fullfile(sourceDir,'Code'));
addpath(fullfile(sourceDir,'Model'));
modelDir=fullfile(sourceDir,'Model');

eval(['mcc -m -R -singleCompThread -R -nosplash -a '''
modelDir, ''' -d '''', install_prefix, ''' -o libra libra.m']);

outputBinary=fullfile(install_prefix,'libra');
if isunix
  % Change file permission (chmod 755)
  fileattrib(outputBinary,'+x','a');
  fileattrib(outputBinary,'+w','u');
end
```
Since LIBRA uses MCR (CaPTk ships with MCRv2014), a setup file is needed to ensure CaPTk picks up MCR at runtime.

```bash
${CaPTk_Source}
   /applications
   /individualApps
   /libra

libra.bat

set inputimage = %1
set outputdirectory = %2
set curdrive = %CD:~0,2%
set curdir = %~dp0
%curdrive%
cd %curdir%

PATH=%PATH%;%~dp0\Mathworks\MCR\v81\runtime\win64
.libra.exe %1 %2 %3 %4
```
Integrating Standalone Applications – Continued

Converting a Python script (or a collection of scripts) into a single binary using PyInstaller (using Confetti as example):

```python
import os
import PyInstaller
print "Creating exe"

runPyInstaller.py
```

```bash
$ {CaPTk_Source} /applications /individualApps /confetti_source /src /pyGUI
```
Integrating Standalone Applications – Continued

Converting a Python script (or a collection of scripts) into a single binary using PyInstaller (using Confetti as example):

```python
import os
import PyInstaller
print "Creating exe"

os.system("pyinstaller.exe --onefile --windowed --icon=icons/confetti.ico --exclude-module=matplotlib --exclude-module=tkinter --exclude-module=zmq --exclude-module=twisted ConfettiGUI.py")

print "All Done"
```
Integrating Standalone Applications – Continued

Converting a Python script (or a collection of scripts) into a single binary using PyInstaller (using Confetti as example):

```python
import os
import PyInstaller
print "Creating exe"

os.system("pyinstaller.exe --onefile --windowed --icon=icons/confetti.ico --exclude-module=matplotlib --exclude-module=tkinter --exclude-module=zmq --exclude-module=twisted ConfettiGUI.py")

print "All Done"
```
• Docker
  o This is another mechanism to provide standalone executables that do not have any UI
  o While this is a perfectly valid way to package an application, we have not yet found an application which requires a Docker container, and cannot provide an example
  o If you have a package that is available as a Docker-ized container and are interested in integrating it with CaPTk, please let us know how we can help!
To set up the package, place your compiled application (let’s use the example of LIBRA again) in the following location of ${CaPTk_source}

```
${CaPTk_Source}
/src
/applications
/individualApps
/libra
```
Edit the applications/CMakeLists.txt file which sets up the installation of individual applications

```
FOREACH(subdir ${SUBDIRECTORIES})
    IF (${subdir} STREQUAL "libra")
        INSTALL(FILES ${SUBDIRPATH}/libra.exe
                DESTINATION bin # this location is important
        )
        INSTALL(FILES ${SUBDIRPATH}/libra.bat
                DESTINATION bin # this location is important
        )
    ENDIF()
ENDFOREACH()
```

${CaPTk_Source}
/src
/applications
CMakeLists.txt
Edit the `applications/CMakeLists.txt` file which sets up the installation of individual applications

```cmake
FOREACH(subdir ${SUBDIRECTORIES})
  IF (${subdir} STREQUAL "libra")
    INSTALL(FILES ${SUBDIRPATH}/libra.exe
                DESTINATION bin # this location is important
            )
    INSTALL(FILES ${SUBDIRPATH}/libra.bat
                DESTINATION bin # this location is important
            )
  ENDIF()
ENDFOREACH()
${CaPTk_Source}
/srcc
/applications
CMakeLists.txt
```
The source-level functions to add/edit/copy are in `fMainWindow.h`, `fMainWindow.cpp` and `ui_fMainWindow.h`

```
${CaPTk_Source}
/src
/gui
  ui_fMainWindow.h
  fMainWindow.h
  fMainWindow.cpp
```
Edit the relevant Application List (the applications are delineated by a single space).

```cpp
auto brainAppList = " WhiteStripe PopulationAtlases confetti EGFRvIISurrogateIndex RecurrenceEstimator SurvivalPredictor MolecularSubtypePredictor";
auto breastAppList = " librasingle librabatch";
auto lungAppList = " SBRT_Segment";
auto miscAppList = " itksnap GeodesicSegmentation DirectionalityEstimate PerfusionDerivatives PerfusionPCA DiffusionDerivatives"
```

```cpp
${CaPTk_Source}/
/src
/gui
 ui_fMainWindow.h
 fMainWindow.h
 fMainWindow.cpp
```
Add the Qt “slots” that connects with the application

```cpp
public slots:

void ApplicationLIBRASingle(); // single image mode of LIBRA
void ApplicationLIBRABatch(); // batch mode of LIBRA
```

${CaPTk_Source}
/src
/gui
    ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Customize the menu text

```cpp
void fMainWindow::fMainWindow()
{
  ...
  for (size_t i = 0; i < vectorOfBreastApps.size(); i++) {
    if (vectorOfBreastApps[i].name.find("librasingle") != std::string::npos) {
      vectorOfBreastApps[i].action->setText(" Breast Density Estimator (LIBRA) SingleImage");
      connect(vectorOfBreastApps[i].action, SIGNAL(triggered()),
              this, SLOT(ApplicationLIBRASingle()));
    }
  }
  ...
}
```

${CaPTk_Source}$

/src
/gui
  ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Connect the menu item with the function slot

```cpp
void fMainWindow::fMainWindow()
{
    ...  
    for (size_t i = 0; i < vectorOfBreastApps.size(); i++) {
        if (vectorOfBreastApps[i].name.find("librasingle") != std::string::npos) {
            vectorOfBreastApps[i].action->setText("Breast Density Estimator (LIBRA) SingleImage");
            connect(vectorOfBreastApps[i].action, SIGNAL(triggered()), this, SLOT(ApplicationLIBRASingle()));
        }
    }
    ... 
}
```

${CaPTk_Source}$

/src
/gui
ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Populate the function calls

```cpp
void fMainWindow::ApplicationLIBRASingle()
{
    auto items = m_imagesTable->selectedItems();
    if (items.empty())
    {
        ShowErrorMessage("At least 1 supported image needs to be loaded and selected");
        return;
    }
    ...
}
```

${CaPTk_Source}$
/src
/gui
    ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Populate the function calls

```cpp
void fMainWindow::ApplicationLIBRASingle()
{
    auto items = m_imagesTable->selectedItems();
    if (items.isEmpty())
    {
        ShowErrorMessage("At least 1 supported image needs to be loaded and selected");
        return;
    }
    ...
}
```
Integrating Standalone Applications – Continued

Populate the function calls

```cpp
void fMainWindow::ApplicationLIBRASingle()
{
    ...
    auto scriptToCall = m_allNonNativeApps["libra"];
    cbica::replaceString( scriptToCall, ".bat", ".exe" );
    ...
}
```

`${CaPTk_Source}`

/src
/gui
ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Populate the function calls

```cpp
void fMainWindow::ApplicationLIBRASingle()
{

 ...

    auto scriptToCall = m_allNonNativeApps["libra"];

    cbica::replaceString( scriptToCall, ".bat", ".exe" );

 ...

}
```

```bash
${CaPTk_Source}
/src
/gui
    ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
```
Populate the function calls

```cpp
void fMainWindow::ApplicationLIBRASingle()
{
    ...
    QStringList args;
    args << dicomfilename.c_str() << tempFolderLocation.c_str();
    if (startExternalProcess(scriptToCall.c_str(), args) != 0)
    {
        ShowErrorMessage("LIBRA failed to execute. Please check installation requirements and retry.");
    }
}
```
Populate the function calls

```cpp
void fMainWindow::ApplicationLIBRASingle()
{
    ...
    QStringList args;
    args << dicomfilename.c_str() << tempFolderLocation.c_str();
    if (startExternalProcess(scriptToCall.c_str(), args) != 0)
    {
        ShowErrorMessage("LIBRA failed to execute. Please check installation requirements and retry.");
        return;
    }
    readMaskFile(tempFolderLocation + "/Result_Images/totalmask/totalmask.dcm");
}
```
Populate the function calls

```cpp
void fMainWindow::ApplicationLIBRASingle()
{
    ...
    QStringList args;
    args << dicomfilename.c_str() << tempFolderLocation.c_str();
    if (startExternalProcess(scriptToCall.c_str(), args) != 0)
    {
        ShowErrorMessage("LIBRA failed to execute. Please check installation requirements and retry.");
        return;
    }
    LoadDrawing( tempFolderLocation + "/Result_Images/totalmask/totalmask.dcm");
}
```

`$\{\text{CaPTk\_Source}\}$

/src
/gui
   ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Flowchart of Standalone App running from CaPTk

- Image Loaded from CaPTk
  - DICOM/NIfTI
  - Visualization of ROI

- Loaded Image(s) called for processing
  - System call to application
  - MATLAB/Python

- Result to be inferred by developer/user
  - Image or a number
  - Need special processing for integration
Integrating Native Applications

• Written in C++ and follows a **Object Oriented** Structure

• Well-defined API which uses ITK Image(s) as input and gives either another image or a value as output
  - `SetInput()` or `SetInputs()`
  - `Update()`
  - `GetOutput()`
  - No file parsing or I/O should be done within the class (sanity checks for data are all done within CaPTk)
To set up the package, place the application files (let’s use the example of EGFRvIII PHI Estimator) in the following location of `{$CaPTk_Source}`

```bash
${CaPTk_Source}/src/applications
```

- `EGFRvIIIISurrogateIndex.h`
- `EGFRvIIIISurrogateIndex.cpp`
- `EGFRvIIIISurrogateIndex.cxx`

Application header and implementation files that contain all the processing information.
Integrating Native Applications – Continued

To set up the package, place the application files (let’s use the example of EGFRvIII PHI Estimator) in the following location of `${CaPTk_source}`

```
${CaPTk_Source}
/src
   /applications
EGFRvIIIISurrogateIndex.h
EGFRvIIIISurrogateIndex.cpp
EGFRvIIIISurrogateIndex.cxx
```

Command Line Executable generator (optional)
Edit the `applications/CMakeLists.txt` file which sets up the installation of individual applications

```cmake
SET( APPLICATIONS 
    ... 
    EGFRvIIIISurrogateIndex 
    ... 
)

FOREACH(application ${APPLICATIONS})
    IF (${application} STREQUAL EGFRvIIIISurrogateIndex)
        ADD_APPLICATION(${application})
    ENDIF()
ENDFOREACH()
```
Edit the relevant Application List (the applications are delineated by a single space).

```cpp
void setupUi(QMainWindow *fMainWindow)
{
    ...
    auto brainAppList = "WhiteStripe PopulationAtlases confetti EGFRvIII SurrogateIndex RecurrenceEstimator SurvivalPredictor MolecularSubtypePredictor";
    ...
}
```
Edit the relevant Application List (the applications are delineated by a single space).

```c++
void setupUi(QMainWindow *fMainWindow)
{
    ...
    auto brainAppList = "WhiteStripe PopulationAtlases confetti EGFRvIII SurrogateIndex RecurrenceEstimator SurvivalPredictor MolecularSubtypePredictor";
    ...
}
```
Add the “slot” that calls the function that does the actual calculation

```cpp
public slots:

void ApplicationEGFR();
```

```
${CaPTk_Source}
/src
/gui
    ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
```
Customize the menu text

```cpp
void fMainWindow::fMainWindow()
{
  ...
  for (size_t i = 0; i < vectorOfGBMApps.size(); i++) {
    if (vectorOfGBMApps[i].name.find("EGFR") != std::string::npos) {
      vectorOfGBMApps[i].action->setText("Glioblastoma EGFRvIII Surrogate Index (PHI Estimator)");
      connect(vectorOfGBMApps[i].action, SIGNAL(triggered()),
               this, SLOT(ApplicationEGFR()));
    }
  }
  ...
}
```
Connect the menu item with the function slot

```cpp
void fMainWindow::fMainWindow()
{
    ...
    for (size_t i = 0; i < vectorOfGBMApps.size(); i++) {
        if (vectorOfGBMApps[i].name.find("EGFR") != std::string::npos) {
            vectorOfGBMApps[i].action->setText("Glioblastoma EGFRvIII Surrogate Index (PHI Estimator)");
            connect(vectorOfGBMApps[i].action, SIGNAL(triggered()), this, SLOT(ApplicationEGFR()));
        }
    }
    ...
}
```

${CaPTk_Source}
/src
/gui
ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Check if a mask is defined or not

```cpp
void fMainWindow::ApplicationEGFR()
{
    if (!isMaskDefined())
    {
        ShowErrorMessage("EGFRvIII Estimation requires Near and Far regions to be initialized");
        help_contextual("Glioblastoma_PHI.html");
        return;
    }
    updateProgress(5);
    ...
}
```
Populate the function calls

```cpp
void fMainWindow::ApplicationEGFR()
{
    if (!isMaskDefined())
    {
        ShowErrorMessage("EGFRvIII Estimation requires Near and Far regions to be initialized");
        help_contextual("Glioblastoma_PHI.html");
        return;
    }
    updateProgress(5);
    std::vector< ImageTypeFloat3D::IndexType > nearIndices, farIndices;
    ...
}
```
If mask is defined, then get the near and far indices that were initialized as ROIs

```cpp
void fMainWindow::ApplicationEGFR()
{
    auto mask = convertVtkToItk<float, 3>(mSlicerManagers[0]->mMask);
    ImageTypeFloat3DIterator maskIt(mask, mask->GetLargestPossibleRegion());
    for( maskIt.GoToBegin(); !maskIt.IsAtEnd(); ++maskIt ) {
        if (maskIt.Get() == 1)
            nearIndices.push_back(maskIt.GetIndex());
        else if (maskIt.Get() == 2)
            farIndices.push_back(maskIt.GetIndex());
    }
}
```
If mask is defined, then get the near and far indices that were initialized as ROIs

```cpp
void fMainWindow::ApplicationEGFR()
{
    auto mask = convertVtkToItk< float, 3>(mSlicerManagers[0]->mMask);
    ImageTypeFloat3DIterator maskIt(mask, mask->GetLargestPossibleRegion());
    for( maskIt.GoToBegin(); !maskIt.IsAtEnd(); ++maskIt; ) {
        if (maskIt.Get() == 1)
            nearIndices.push_back(maskIt.GetIndex());
        else if (maskIt.Get() == 2)
            farIndices.push_back(maskIt.GetIndex());
    }
    ...
}
```
If mask is defined, then get the near and far indices that were initialized as ROIs

```cpp
void fMainWindow::ApplicationEGFR()
{
    auto mask = convertVtkToItk<float, 3>(mSlicerManagers[0]->mMask);
    ImageTypeFloat3DIterator maskIt(mask, mask->GetLargestPossibleRegion());
    for( maskIt.GoToBegin(); !maskIt.IsAtEnd(); ++maskIt; ) {
        if (maskIt.Get() == 1)
            nearIndices.push_back(maskIt.GetIndex());
        else if (maskIt.Get() == 2)
            farIndices.push_back(maskIt.GetIndex());
    }
}
```
Populate the function calls

```cpp
void fMainWindow::ApplicationEGFR()
{
    ...
    for (unsigned int index = 0; index < mSlicerManagers.size(); index++) {
        if (mSlicerManagers[index]->mImageSubType == IMAGE_TYPE_T1CE)
            T1CEImagePointer = mSlicerManagers[index]->mITKImage;
        else if (mSlicerManagers[index]->mImageSubType == IMAGE_TYPE_T2FLAIR)
            T2FlairImagePointer = mSlicerManagers[index]->mITKImage;
        else if (mSlicerManagers[index]->mImageSubType == IMAGE_TYPE_PERFUSION)
            perfusionImage = mSlicerManagers[index]->mPerfusionImagePointer;
        ...
    }
}
```

${CaPTk_Source}/
/src
/gui
  ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Populate the function calls

```cpp
void fMainWindow::ApplicationEGFR()
{
    for (unsigned int index = 0; index < mSlicerManagers.size(); index++) {
        if (mSlicerManagers[index]->mImageSubType == IMAGE_TYPE_T1CE)
            T1CEImagePointer = mSlicerManagers[index]->mITKImage;
        else if (mSlicerManagers[index]->mImageSubType == IMAGE_TYPE_T2FLAIR)
            T2FlairImagePointer = mSlicerManagers[index]->mITKImage;
        else if (mSlicerManagers[index]->mImageSubType == IMAGE_TYPE_PERFUSION)
            perfusionImage = mSlicerManagers[index]->mPerfusionImagePointer;
    }
}
```
Populate the function calls

```cpp
${CaPTk_Source}
/src
/gui
    ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp

void fMainWindow::ApplicationEGFR()
{
    EGFRStatusPredictor EGFRPredictor;
    auto EGFRStatusParams = EGFRPredictor.PredictEGFRStatus<
ImageTypeFloat3D, PerfusionImageType >(perfusionImage,
Perfusion_Registered, nearIndices, farIndices, NIfTI);
    ...
}
Integrating Native Applications – Continued

Display the results

```cpp
void fMainWindow::ApplicationEGFR()
{
    auto msg = "PHI = " + QString::number(EGFRStatusParams[0]) + "\n\n\n(Near:Far) Peak Height ratio = " +
    QString::number(EGFRStatusParams[1] / EGFRStatusParams[2]) + "\n\nNear ROI voxels used = " +
    QString::number(EGFRStatusParams[3]) + "\n\nFar ROI voxels used = " +
    QString::number(EGFRStatusParams[4]) + "\n\nPHI Threshold = 0.1377\n[based on 142 UPenn brain tumor scans]";
    ...
}
```

${CaPTk_Source}$

```
$CaPTk_Source$
```

```
/srcc/Gui/ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
```

Section of Biomedical Imaging Analysis

University of Pennsylvania

SBI
Display the results

```cpp
void fMainWindow::ApplicationEGFR()
{
    ShowMessage(msg.toStdString());
    updateProgress(100);
}
```
Flowchart of Native App running from CaPTk

1. **Image Loaded from CaPTk**
   - DICOM/NIfTI
   - Visualization of ROI

2. **Loaded Image(s) called for processing**
   - Application object created
   - Data passed from CaPTk to object

3. **Results can be shown from CaPTk**
   - Image or a number
   - Either Image/ROI load or a pop-up showing result(s)
Considering WhiteStripe as an example, we follow the same procedures as EGFRvIII

${\text{CaPTk\_Source}}$
/src
/aplications

WhiteStripe.h
WhiteStripe.cpp
WhiteStripe.hxx

Application header, implementation files and command line executable generator
Application requiring User Parameterization

Since WS needs user-defined parameters, we need to create a dialog box which emits the final “Run” signal with the provided parameters (radius, kernel size, etc.)

```
${CaPTk_Source}
/src
/gui

ui_fWhiteStripeDialog.h
fWhiteStripeDialog.h
fWhiteStripeDialog.cpp
```

Creation of the dialog box interface and its relevant controls
Integrating Native Applications – Continued

Edit the `applications/CMakeLists.txt` file which sets up the installation of individual applications:

```bash
${CaPTk_Source}
/src
/applications/CMakeLists.txt

SET( APPLICATIONS
    ...
    WhiteStripe
    ...
  )

FOREACH(application ${APPLICATIONS})
    IF (${application} STREQUAL WhiteStripe)
        ADD_APPLICATION( ${application} )
    ENDIF()
ENDFOREACH()
```
Edit the relevant Application List (the applications are delineated by a single space).

```cpp
void setupUi(QMainWindow *fMainWindow)
{
    ...
    auto brainAppList = " WhiteStripe PopulationAtlases confetti EGFRvIIISurrogateIndex RecurrenceEstimator SurvivalPredictor MolecularSubtypePredictor";
    ...
}
```

`{CaPTk_Source}
/src
/gui
    ui_fMainWindow.h
    fMainWindow.h
    fMainWindow.cpp`
Add the “slot” that calls the function that does the actual calculation

```csharp
public slots:

    void ApplicationWhiteStripe();
```

${CaPTk_Source}

/src
/gui

ui_fMainWindow.h
fMainWindow.h
fMainWindow.cpp
Customize the menu text

```cpp
void fMainWindow::fMainWindow()
{
    ...
    for (size_t i = 0; i < vectorOfGBMApps.size(); i++) {
        if (vectorOfGBMApps[i].name.find("WhiteStripe") != std::string::npos) {
            vectorOfGBMApps[i].action->setText("WhiteStripe Normalization");
            connect(vectorOfGBMApps[i].action, SIGNAL(triggered()), this, SLOT(ApplicationWhiteStripe()));
        }
        ...
    }
    ...
}
```
Connect the menu item with the function slot

```cpp
void fMainWindow::fMainWindow()
{
  ...
  for (size_t i = 0; i < vectorOfGBMApps.size(); i++) {
    if (vectorOfGBMApps[i].name.find("WhiteStripe") != std::string::npos) {
      vectorOfGBMApps[i].action->setText("WhiteStripe Normalization");
      connect(vectorOfGBMApps[i].action, SIGNAL(triggered()), this, SLOT(ApplicationEGFR()));
    }
  }
  ...
}
```

${CaPTk_Source} /src /gui ui_fMainWindow.h fMainWindow.h fMainWindow.cpp
Application requiring User Parameterization

Connect the “Run” signal from the dialog with the MainWindow Interface.

```cpp
void fMainWindow::fMainWindow()
{
    connect(&whiteStripeNormalizer,
            SIGNAL(RunWhiteStripe(double, int, int, int, double,
                                    double, int, bool, const std::string)),
            this,
            SLOT(CallWhiteStripe(double, int, int, int, double,
                                  double, int, bool, const std::string)));
}
```
Connect the “Run” signal from the dialog with the MainWindow Interface.

```cpp
void fMainWindow::fMainWindow()
{
    connect(&whiteStripeNormalizer,
            SIGNAL(RunWhiteStripe(double, int, int, int, double, double, int, bool, const std::string)), this,
            SLOT(CallWhiteStripe(double, int, int, int, double, double, int, bool, const std::string)));
}
```

`${CaPTk_Source}`

```bash
$ src
  /gui

ui_fWhiteStripeDialog.h
fWhiteStripeDialog.h
fWhiteStripeDialog.cpp
```
Application requiring User Parameterization

Connect the “Run” signal from the dialog with the MainWindow Interface.

```cpp
void fMainWindow::fMainWindow()
{
    connect(&whiteStripeNormalizer,
             SIGNAL(RunWhiteStripe(double, int, int, int, double, double, int, bool, const std::string)), this,
             SLOT(CallWhiteStripe(double, int, int, int, double, double, int, bool, const std::string)));
}
```

${CaPTk_Source}

/src
/gui

ui_fWhiteStripeDialog.h
fWhiteStripeDialog.h
fWhiteStripeDialog.cpp
This checks for the inputs within CaPTk and then loads up the WhiteStripe dialog box

```cpp
void fMainWindow::ApplicationWhiteStripe()
{
    ...
    if ((mSlicerManagers[index]->mImageSubType == IMAGE_TYPE_T1) || (mSlicerManagers[index]->mImageSubType == IMAGE_TYPE_T2)) {
        auto tmp = mInputPathName.toStdString();
        whiteStripeNormalizer.SetImageModality(mSlicerManagers[index]->mImageSubType);
        whiteStripeNormalizer.exec();
    }
}
```
This checks for the inputs within CaPTk and then loads up the WhiteStripe dialog box

```cpp
#include "WhiteStripeDialog.h"

void fMainWindow::CallWhiteStripe(...) {
    WhiteStripe normalizer;
    normalizer.setParams(twsWidth, sliceStartZ, sliceStopZ, tissuesMax, smoothMax, smoothDelta, histSize, T1Image);

    ImageTypeFloat3D::Pointer mask = normalizer.process(mSlicerManagers[index]->mITKImage, mask);
}
```

${CaPTk_Source}/src/gui/ui_fWhiteStripeDialog.h
fWhiteStripeDialog.h
fWhiteStripeDialog.cpp
This checks for the inputs within CaPTk and then loads up the WhiteStripe dialog box

```cpp
void fMainWindow::CallWhiteStripe(...) {
    WhiteStripe normalizer;
    normalizer.setParams(twsWidth, sliceStartZ, sliceStopZ, tissuesMax, smoothMax, smoothDelta, histSize, T1Image);
    ImageTypeFloat3D::Pointer mask;
    auto normImage = normalizer.process(mSlicerManagers[index]->mITKImage, mask);
}
```