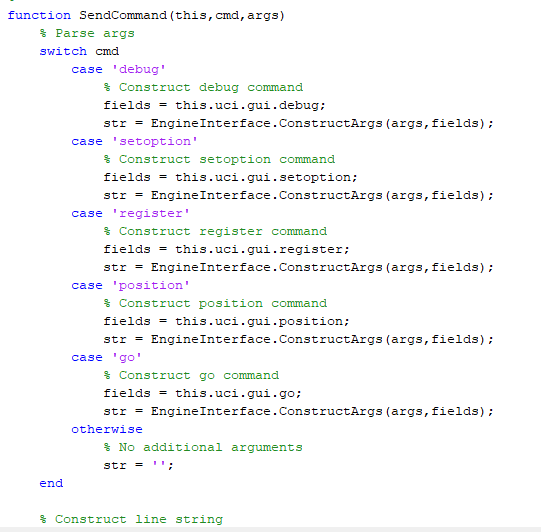
Still looking on how to manipulate an exe file from matlab script. The chessmaster GUI has some functions, and from examining some of the codes, it seems that our required task of passing commands to the engine is done on a file, EngineInterface.m . However I haven’t got a clue how the interfacing is done despite the functions are in front of me.

The required function is SendCommand(), which again uses another structure called uci.gui whatever.



Machine design

I have some experience on designing robotic parts, in paper of course, and I made a robotic arm for an industrial automation challenge before with PVC sheets. This time I might use pvc too as I don’t have access to machine shop to make parts out of metal etc. And maybe use solidworks for the designing bit so that anyone might reconstruct the whole thing.

Looking into basic mechanisms in CNC machines

<http://carbide3d.com/shapeoko/>

Didn’t find much useful stuff, I need designs to work on.

Google XY plotter instead

Linear bearing

EngineInterface.m script contains the class definition for the class EngineInterface, this class necessarily handles all the work with the exe file aka the engine.

EngineInterface class has, private and public variables.

Class EngineInterface()

{

Private variables

Const PAUSE\_TIME

Const DEF\_WAIT\_TIME

% Engine search info

info; % Info structure

**% Communication variables**

**p; % Engine process**

**in; % GUI input (engine output) stream**

**out; % GUI output (engine input) stream**

timerobj; % Asynchronous communication timer

% Internal variables

uci; % UCI commands structure

maxCPU = 1; % Max CPU usage in [0 1]

registerNow = true; % Register now flag

Public variables

name = 'Engine'; % Engine name

author = '??????'; % Engine author

options = {}; % Engine options

% Locks

qlock = false; % Quiet timer-stop lock

rlock = false; % Reading lock

% Engine-associated objects

obj; % ChessEngine/GameAnalyzer parent

EL; % EngineLog object

EO; % EngineOptions object

% Paths

dir;

The underlined three variables are my point of interest as they are doing the main task of communication with the engine.

function [isReady cmd args] = GetCommand(this)

% Get line from engine

[isReady line] = this.GetLine();

GetCommand function has been used in places where we used to expect to see text generated by the engine after hitting enter after commands. We are interested in **Getline()**

function SendCommand(this,cmd,args)

% Parse args

switch cmd

case 'debug'

% Construct debug command

fields = this.uci.gui.debug;

str = EngineInterface.ConstructArgs(args,fields);

case 'setoption'

% Construct setoption command

fields = this.uci.gui.setoption;

str = EngineInterface.ConstructArgs(args,fields);

case 'register'

% Construct register command

fields = this.uci.gui.register;

str = EngineInterface.ConstructArgs(args,fields);

case 'position'

% Construct position command

fields = this.uci.gui.position;

str = EngineInterface.ConstructArgs(args,fields);

case 'go'

% Construct go command

fields = this.uci.gui.go;

str = EngineInterface.ConstructArgs(args,fields);

otherwise

% No additional arguments

str = '';

end

% Construct line string

line = strtrim(sprintf('%s %s',cmd,str));

% Send line, if non-empty

if ~isempty(line)

% Send line to engine

this.SendLine(line);

end

end

as you can see, the input arguments used for SendCommand are the same ones we used as commands for the stockfish.exe file, so our point of interest is the function **SendLine()**

function InitializeEngine(this,book)

% Clear info list

this.ClearInfo();

% Tell the engine to use UCI

this.SendCommand('uci');

% Read until 'uciok' is received

this.ReadUntilCMD('uciok');

% Ready handshake

this.ReadyHandshake();

% Make sure debug mode is off

%this.SendCommand('debug',struct('off',true));

% Make sure pondering is off (not supported by ChessEngine)

this.SetOption('Ponder','false');

% Set minimum thinking time to zero

this.SetOption('Minimum Thinking Time','0');

try

% Get # CPU cores

% NOTE: feature() is undocumented

numCores = feature('numCores');

% Set # threads to give desired max CPU usage

% NOTE: Only succeeeds if engine supports "Threads" option

numThreads = sprintf('%.0f',floor(this.maxCPU \* numCores));

this.SetOption('Threads',numThreads);

catch %#ok

warnStr = 'Undocumented feature(''numCores'') has failed';

this.EL.AppendWarningLine(warnStr);

end

% Set up opening book access

if ~isempty(book)

% Tell the engine to use its own book

% NOTE: Only succeeds if engine supports "OwnBook" option

this.SetOption('OwnBook','true');

% Give the engine the specified opening book path

% NOTE: Only succeeds if engine supports "Book File" option

this.SetOption('Book File',this.AbsPath(book));

else

% Tell the engine \*not\* to use its own book

% NOTE: Only succeeds if engine supports "OwnBook" option

this.SetOption('OwnBook','false');

end

% Tell the engine to start a new game internally

this.SendCommand('ucinewgame');

end

This function InitializeEngine() is used to optimize the engine for gameplay, there are some functions here that we have not discussed, so we will be replacing them with our own functions, namely by sending the specified commands and processing the responses ourselves, with less funcitions.

function [isReady line] = GetLine(this)

% Read line from input stream

isReady = this.in.ready();

if (isReady == true)

% Get line from input buffer

line = char(this.in.readLine());

% Write nonempty lines to engine log

if ~isempty(line)

this.EL.AppendEngineLine(line);

end

else

% Engine had nothing to send

line = '';

end

end

function SendLine(this,line)

% Send line to output stream

this.out.println(line);

%this.out.flush();

% Write line to engine log

this.EL.AppendGUILine(line);

end

we have found the two functions we have been looking for, now the problem is that we still don’t know what in.ready(),in.readLine(),out.println(line); functions are doing here. So we have to find the **in** & **out** variables.

Luckily, the function immediately after these explains a lot for us

function OpenLink(this,path)

% Make sure we have the absolute path

path = this.AbsPath(path);

% (Try to?) set execute permissions

if (ispc == true)

% Windows

usr = 'Everyone';

cmd = sprintf('icacls "%s" /grant %s:RX',path,usr);

[~,~] = system(cmd);

else

% Mac

usr = 'ugo';

cmd = sprintf('chmod %s+x "%s"',usr,path);

[~,~] = system(cmd);

end

% Spawn engine process

%this.p = java.lang.Runtime.getRuntime().exec(path);

this.p = java.lang.ProcessBuilder(path).start();

% Connect to engine's stdout

iStream = this.p.getInputStream();

iStreamReader = java.io.InputStreamReader(iStream);

this.in = java.io.BufferedReader(iStreamReader);

% Connect to engine's stdin

oStream = this.p.getOutputStream();

this.out = java.io.PrintWriter(oStream,true);

%this.out = java.io.PrintWriter(oStream);

end

The underlined three lines are the definitions of the variables that we had discussed earlier, our point of interest. Now lets take a look into them.

p, in and out are basically three objects, namely java objects. P is a process object, java.lang.ProcessBuilder(path) is a processbuilder object, that works on a file specified by ‘path’, and path will be the path to our stockfish.exe. Now java.lang.ProcessBuilder(path).start() will start a process, and we get our p process object.

The process has to get an input stream, input stream of **java(matlab)** receives the stream of data that the **engine(process)** is generating(outputting) for us, which is done in the % Connect to engine's stdout section. Note that by this, we are getting output data generated by the engine.

Output stream of **java(matlab)** sends the stream of data that the code is generating(outputting) by us, and **engine(process)** takes as input, which is done in the % Connect to engine's stdin section. Note that by this, we are sending output data generated by our code.

Okay, so, that brings us to the final code that I have created by changing some stuff

clear all; %just my habit

clc;

%import all the java class files

import java.lang.ProcessBuilder;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.io.BufferedReader;

command = 'position startpos'; %dummy command

path = "D:\stockfish.exe"; %setting the path to our engine

%process started

processBuilderObject = ProcessBuilder(path);

process = processBuilderObject.start();

%output stream connected(the commands that we will send)

oStream = process.getOutputStream();

out = PrintWriter(oStream,true);

%input stream connected(the results that engine generates)

iStream = process.getInputStream();

iStreamReader = InputStreamReader(iStream);

in = BufferedReader(iStreamReader);

%functions we will use to send and receive data

%in.ready();

%in.readLine();

%out.println(command);

I have written these in a script, and then executing the script brings us to the step from where we can use the command window to either communicate with the engine, as well as do whatever other processes we wish to execute.

For example, ive run the script once, and then run the following from the command window:

>> in.ready()

ans =

logical

1

>> in.readLine()

ans =

Stockfish 9 64 by T. Romstad, M. Costalba, J. Kiiski, G. Linscott

>> out.println('position fen rnbqkb1r/p2ppppp/5n2/1ppP4/2P5/8/PP2PPPP/RNBQKBNR w KQkq -');

>> out.println('go');

>> in.ready()

ans =

logical

1

>> in.readLine()

ans =

info depth 1 seldepth 1 multipv 1 score cp 150 nodes 57 nps 19000 tbhits 0 time 3 pv c4b5 d8a5 b1c3

>> in.readLine()

ans =

info depth 2 seldepth 3 multipv 1 score cp 150 nodes 111 nps 37000 tbhits 0 time 3 pv c4b5 d8a5 b1c3

>> in.readLine()

ans =

info depth 3 seldepth 4 multipv 1 score cp 80 nodes 206 nps 68666 tbhits 0 time 3 pv c4b5 d8a5 b1c3 d7d6

>> in.readLine()

ans =

info depth 4 seldepth 5 multipv 1 score cp 120 nodes 281 nps 93666 tbhits 0 time 3 pv c4b5 d8a5 b1c3 d7d6 e2e3

>> in.readLine()

ans =

info depth 5 seldepth 6 multipv 1 score cp 73 nodes 805 nps 268333 tbhits 0 time 3 pv c4b5 d7d6 b1c3 b8d7 c1d2

>> in.readLine()

ans =

info depth 6 seldepth 6 multipv 1 score cp 112 nodes 1379 nps 344750 tbhits 0 time 4 pv c4b5 d7d6 b1c3 c8f5 e2e4 f6e4

>> in.readLine()

ans =

info depth 7 seldepth 8 multipv 1 score cp 102 nodes 2270 nps 454000 tbhits 0 time 5 pv c4b5 e7e6 b1c3 e6d5 c3d5 d7d6 d5c3

>> in.readLine()

ans =

info depth 8 seldepth 9 multipv 1 score cp 118 nodes 4797 nps 599625 tbhits 0 time 8 pv c4b5 a7a6 e2e4 f6e4 d1e2 e4d6 b1c3 a6b5 c3b5 d6b5

>> in.readLine()

ans =

info depth 9 seldepth 12 multipv 1 score cp 82 nodes 12900 nps 921428 tbhits 0 time 14 pv c4b5 a7a6 e2e3 a6b5 f1b5 d8a5 b1c3 e7e6 d5e6 f7e6

>> in.readLine()

ans =

bestmove c4b5 ponder a7a6

So, in.ready() is to check if there is any output by the engine that we haven’t received yet, don’t use in.readLine() without checking first, otherwise if there is no engine output, the process keeps standing there for eternity. Out.println(command) sends the command as necessary.