CS840 Project 5:

Constructive Cost Model

Samuel Gluss

Jozo Dujmovic

4-2-2016

Abstract – Software engineering is the process of applying software development tools and techniques to develop a useful program. This program might solve a novel problem, outperform a competitor in some category, or in some other way deliver a product that is useful to someone. At some point in this process, software development is likely to take place, and the cost of a software engineer’s hourly rate can make this quite dear indeed.

Where money is spent, science is soon to follow, and the development of software is no exception. Constructive cost modeling reveals the factors which determine the relative challenge and cost of undertaking a given software project, based on perceived production challenge factors such as how much specifications are likely to change, how effective the software engineers are in various capacities, and how well the undertaking is understood.

# I. Introduction

Constructive Cost Modelling (COCOMO) is based on equations that have been developed by observing many software development projects of diverse types over the years. Through this analysis, effective correlations were formed between the configuration of the project, characteristics such as the analytical capabilities of the developers and the number of logical lines of code to be produced (LLOC) and cost factors for the project, such as number of required developers, total cost of the project, and number of months to produce.

Each program that will be analyzed must have an associated config file. This config file will contain all necessary parameters such as name, executable size, mode, level of detail, and whatever COCOMO parameters are necessary to determine the model multiplier. Depending on the level of detail used, either no parameters will be needed, or 15-16. The need for higher level of detail for the model depends on how much the project is expected to deviate from the characteristics of ‘nominal’ projects. Each parameter may be assigned some value from very low to extra high, as long as these selections are represented in the COCO model.

The purpose of the COCOMO program is to parse these ‘config’ files which specify the parameters of a project, match the parameters to multiplier values, then evaluate the project for anticipated cost, staff, and time requirements. The values from the config file are read into a python object, which is then passed to a python script which checks the parameter values, and produces the multiplier for project effort. Once this is done, the KLLOC (1000s of Logical Lines of Code) are produced using the formula determined in Project 3: KLLOC = (exeSize - 188000) / (28.8314 \* 1000)

Once effort is computed, it is a simple task to compute the remaining COCO outputs using the COCOMO formulas. These values are output into a row in the output file, with an associated index, and the appropriate names for the project. Each row, then, represents an evaluated project.

Once the output file is generated, it can be parsed to discover useful relationships in the data. To this end, Gnuplot was utilized to produce high-quality charts. In order to abstract out this visual generation, Python was utilized to form a type of Model-View-Controller arrangement. The model is represented by the output file, where the data is stored. Views are different Gnuplot scripts, which parse the output file to produce plot and chart graphics. The Python program takes the role of the Controller, creating different versions of generic Gnuplot scripts which are then modified to access the correct columns in the output file, as well as have the correct labels and outputs for graphing. By organizing data presentation this way, it is very easy to simply specify the type of plots to generate, instead of writing the scripts by hand each time.

The python program reads lines from a generic gnuplot script, then searches the script for anchors which indicate where to make adjustments. Each search increments a cursor through the lines of the script, which prevents the need to start from the beginning each time. As a result, anchors are searched in descending order, as they appear in the script. When an anchor is found, its line number is used to determine where to add the necessary modifications. Doing the modifications with anchor-searching means that the script can be reconfigured, and as long as the anchors and their associated directives are not changed, the system will work normally. Once the script modification is complete, the script is saved to disk, and executed to produce the correct plot.

Contents

[I. Introduction 1](#_Toc451205628)

[II. KLLOC of projects 4](#_Toc451205629)

[III. Development Time Required 5](#_Toc451205630)

[IV. Staff Required 7](#_Toc451205631)

[V. Project Cost 9](#_Toc451205632)

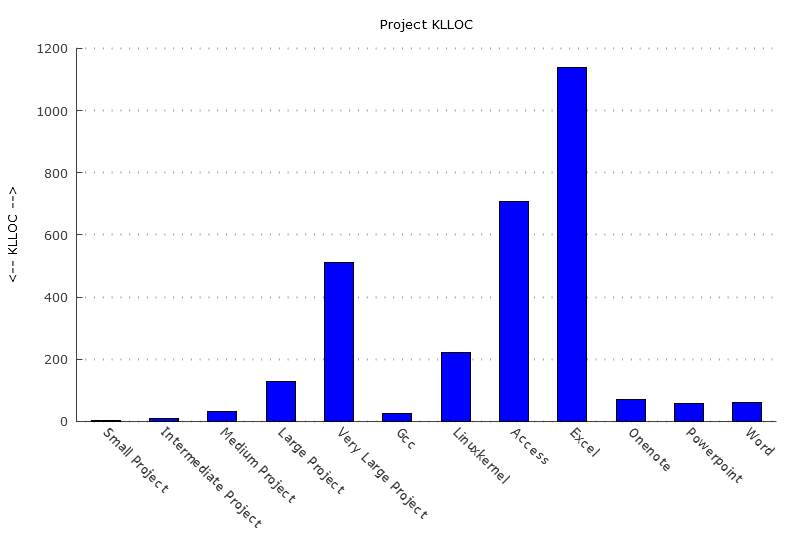
[VI. Conclusion 11](#_Toc451205633)

[Appendix Data: 12](#_Toc451205634)

[Appendix: Python Code 14](#_Toc451205635)

[Appendix: Gnuplot Scripts 22](#_Toc451205636)

# II. KLLOC of projects



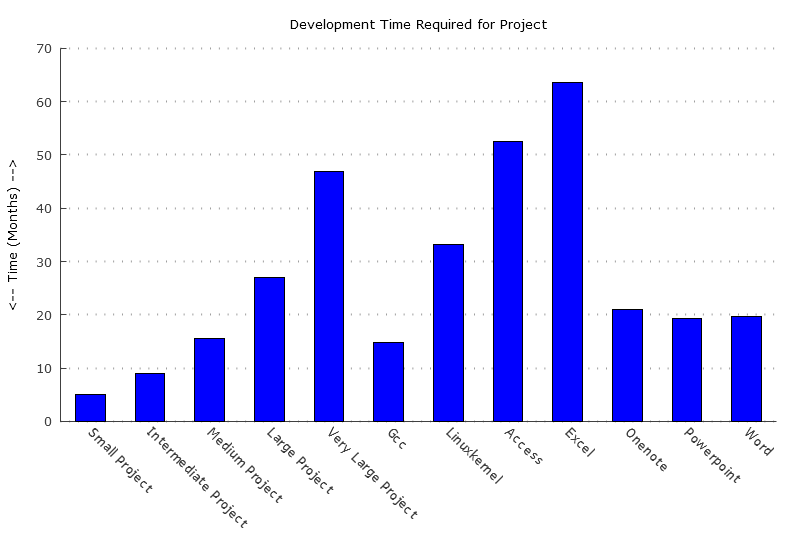
KLLOC is determined either by counting the number of lines of logical code a program contains, or making an estimate based on the size of the executable. In the case of the generic projects, a KLLOC value was assigned, whereas for the actual software programs that were analyzed, KLLOC was inferred from executable size according to this equation:

KLLOC = (sizeInBytes - 188000) / (28.8314 \* 1000)

KLLOC can vary dramatically between projects. Some projects can easily be orders of magnitude larger than others, depending on their scope. For example, a complete operating system kernel for Linux would be expected to be quite a bit larger than a relatively simple compiler such as GCC.

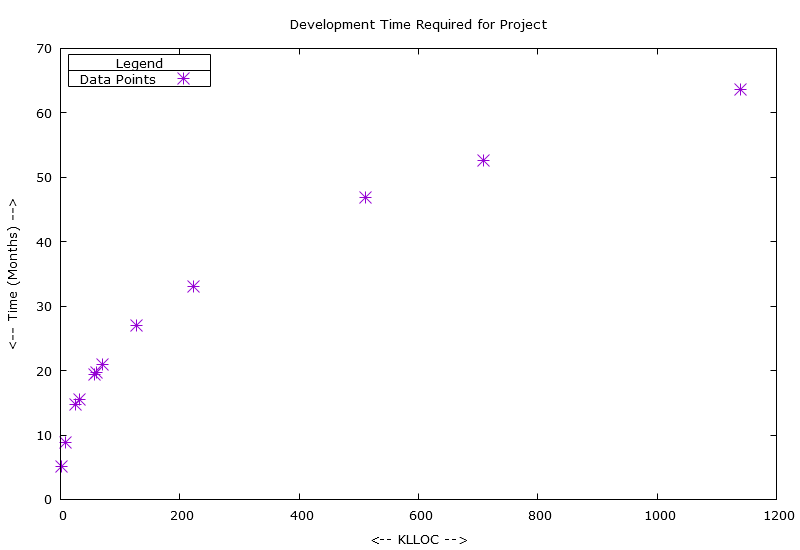
The outliers in this plot are certain Microsoft Excel and Access. The massive size of these two program’s executables suggests that additional libraries and modules were included in compilation, resulting in their overly-inflated size. Since this space is most likely not accounted for by human-generated code, this means that the calculations for these programs will not necessarily reflect expected development parameters.

# III. Development Time Required



Development time is measured as the amount of time a project is expected to take. This is a very important metric because it determines the time from starting a project to completion. This helps determine whether a competitor will be beat to market, and how long engineering teams will be tied up with this particular project. Of course, the opportunity cost of working on a project is that it draws resources away from other projects while it’s being worked on. The development time aspect of COCOMO helps to determine what this opportunity cost is.

The first thing that is noticeable in this barplot, versus the KLLOC values above, is that the variance is much lower. This suggests that actually coding more lines of code is not necessarily very time intensive, once a project is already underway.



Development Time is defined by COCOMO as:

T = r \* Es

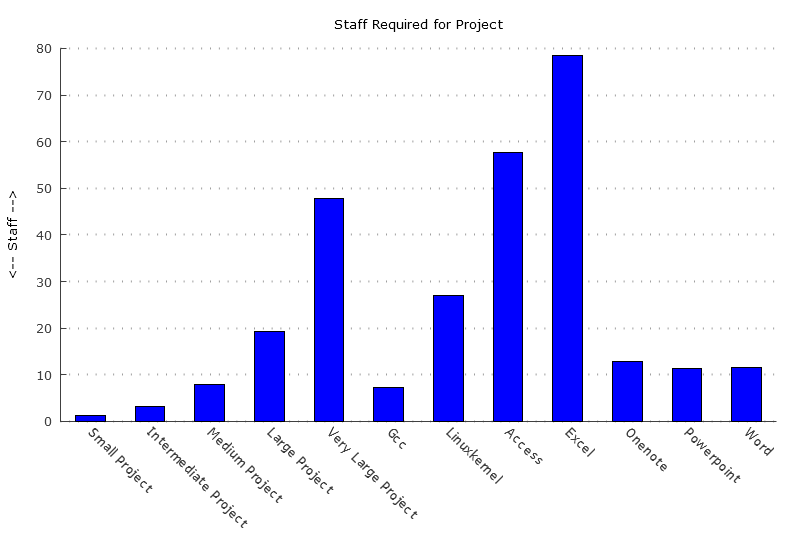
Where r and s are schedule parameters determined by the mode of the project

E is the effort required by the project in man-hours

The scatter plot for development time reveals the relationship between the root of KKLOC and development time. What this means is that very large projects can be tackled in reasonable time, simply by throwing more people at the problem. This may not hold valid for extremely large projects, well over 512 KLLOC, where management of the project will become extremely troublesome.

Furthermore, there is setup time for even the smallest project, a minimum amount of time of a few months to get the team organized, design the architecture, and implement. This minimum time is then further extended to encompass larger and larger projects.

# IV. Staff Required



As Software engineering projects become larger in scope, more engineers will be needed to complete it in a reasonable timeframe. Of course, the mythical man month indicates that simply throwing more engineers at a problem does not mean it will be done any faster. As a result of this, there is an optimal staff count for any given project. This staff count is determined by the equation:

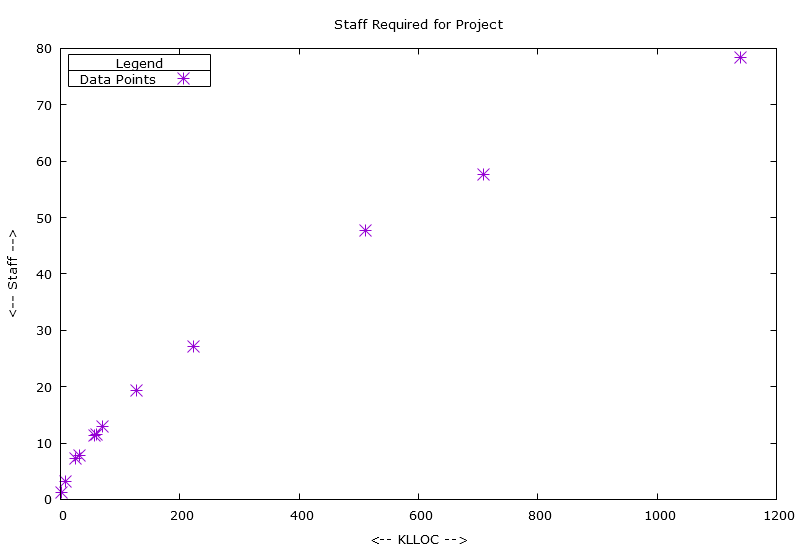
N = E/T

Where N is the number of Software Engineers

E is the Effort required by the project, in man-months

T is the software development time in months

This equation means that the number of people required is proportional to the Effort, or man-months, that a project requires.



So, if the staff size does not directly depend on KLLOC, then a calculation for effort is required to determine the optimal staff to complete a project. Effort is defined by:

E = a \* Lb \* m

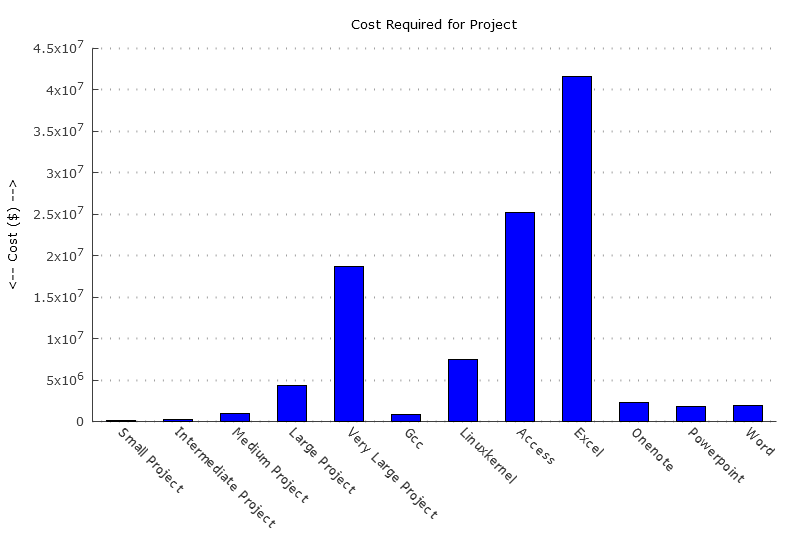
Where a and b are COCOMO level of detail parameters

m is generated by the cost drivers

L represents thousands of logical lines of code (KLLOC)

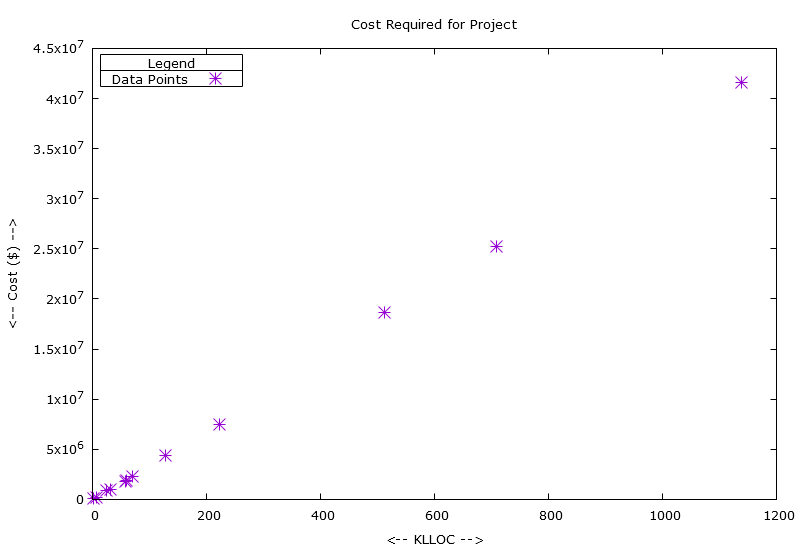
This means that the size of the project in KLLOC and the size of the staff required to optimally prosecute it are not linearly related. Rather, staff is proportional to the bth power of KLLOC. This is visible in the scatterplot above, where the curve formed by the points is in the form of a parabolic root function.

# V. Project Cost



Development cost is another crucial factor for a project in the planning phase. Being able to analyze and determine the cheaper of two approaches to a problem can save a company a massive amount of money. Being able to determine the expected cost of a project has broad benefits in industry and academia, as being able to afford a project, or even to secure funding, is often the first step.

This plot demonstrates the significant variance in cost between different software projects. Some projects cost far more than others, and rightly so. To produce more lines of code, more engineers must be paid, and additional funding is directed to bureaucratic overhead. This is crucial to keep in mind for projects of aggressive scope, as a failure could be catastrophic for the organization.



The interesting feature of this plot is that cost is linearly related to KLLOC. So, this means that there is not necessarily any economy of scale with software projects, at least with regards to cost. At least with project duration, there is at least the possibility to hire more programmers to develop more code per month, but that code will all end up costing the same per line.

Project cost is defined as:

C = q \* E

Where q is the average monthly salary of a software engineer

E is the Effort required by the project, in person-months

E = a \* Lb \* m

So, while the cost seems linear from this plot, Effort is proportional to Lb, and while b is close to 1 for organic projects, which have well defined requirements up front, more complicated projects of the semi-detached or embedded type will exhibit a parabolic curve for cost. This means large embedded projects are likely to contain extremely expensive surprises.

# VI. Conclusion

COCOMO offers a powerful to determine the costs associated with future projects. These costs include manpower, time, and money. One of the biggest issues with COCOMO however, is the complexity of the model parameter selection process. It is hard for the initiate to analyze a project, and determine exactly where their team lies on the spectrum of programmer capability, and how to classify the project itself. The proper method to determine this is to track projects over time, and adapt the COCOMO method to a team or division’s historical capability. Nominally, this will produce a convergence of the COCO estimates, and the actual performance of the team.

The unfortunate side effect of this is that COCOMO will need different parameters for each team, since the qualities of groups of programmers can vary quite wildly. This means any implementation of COCOMO in a software engineering organization will have to be extremely steamlined, and incorporated at the team level, to attain maximum effect. This may not be terribly difficult, as a code counter could monitor a team’s source control commits.

One interesting aberration in the dataset used for this project was the data used for Excel and Access, which had extremely large executable sizes, considering the relative complexity of these programs to the others in the comparison. Whether this is due to massive libraries or other modules is unclear, but what is clear is that these executables are 10-15 times larger than the other Microsoft programs, so clearly there is more than human created code in these projects.

While the executable size to KLLOC conversion is somewhat unreliable here, it is quite easy to count lines of code in software projects, provided the source code is available. Since it is generally possible to count code developed in house, the issues of converting executable size to KLLOC shouldn’t create any issues for implementing COCOMO in the field.

# Appendix Data:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Long Name | Short Name | Cost | Staff | Time (months) | KLLOC |
| Small Project | Small Project | 55211.92 | 1.29 | 5.13 | 2 |
| Intermediate Project | Intermediate Project | 236698.69 | 3.19 | 8.92 | 8 |
| Medium Project | Medium Project | 1014749.48 | 7.85 | 15.5 | 32 |
| Large Project | Large Project | 4350326.25 | 19.37 | 26.96 | 128 |
| Very Large Project | Very Large Project | 18650256.93 | 47.75 | 46.87 | 512 |
| GCC compiler | Gcc | 884756.67 | 7.21 | 14.72 | 24.73414 |
| vmlinuz-3.19.0-58-generic | Linuxkernel | 7482224.34 | 27.11 | 33.13 | 222.6055 |
| Microsoft Office Access | Access | 25229106.99 | 57.59 | 52.57 | 708.3866 |
| Microsoft Office Excel | Excel | 41555499.79 | 78.47 | 63.55 | 1139.401 |
| Microsoft Office Onenote | Onenote | 2256313.59 | 12.89 | 21.01 | 71.07168 |
| Microsoft Office Powerpoint | Powerpoint | 1819288.5 | 11.28 | 19.36 | 57.89632 |
| Microsoft Office Word | Word | 1907249.8 | 11.61 | 19.71 | 60.55925 |

The data is generated row by row by the python script cocomo.py. Once the calculations are complete, the results are joined with commas and appended as a row to the outfile.

Gnuplot is able to read comma delimited files, and use columns as data sources. By manipulating the column select values with a python script, Gnuplot can be made to generate plots that use different values from this table as their source.

**Sample Project Config File:**

# Note - parser anchors on comment lines, do NOT edit them

# Software Name

GCC compiler

# Executable Size (bytes)

901120

# Salary ($ per software enginer per month)

8333

## Level Of Detail (0: m = 1, 1: intermediate1, 2: intermediate2)

1

## Mode (organic, semidetached, embedded)

organic

## Product Attributes (must be very low, low, nominal, high, very high, or extra high)

# m1. requiredSoftwareReliability

very high

# m2. dbSize

nominal

# m3. productComplexity

high

# Computer Attributes

# m4. executionTimeConstraint

high

# m5. mainStorageConstraint

nominal

# m6. virtualMachineVolatility

nominal

# m7. computerTurnaroundTime

nominal

## Personnel Attributes

# m8. AnalystCapabilities

high

# m9. applicationsExperience

high

# m10. programmerCapability

high

# m11. virtualMachineExperience

nominal

# m12. programmingLanguageExperience

high

## Project Attributes

# m13. useOfModernProgrammingPractices

nominal

# m14. useOfSoftwareTools

nominal

# m15. requiredDevelopmentSchedule

nominal

## Defines how much the specifications of the product are expected to change

# m16. requirementsVolatility

nominal

# Appendix: Python Code

**Cocomo.py - Main COCOMO program**

# Samuel Gluss

# 4-10-2016

# CSC 840

# Jozo Dujmovic

# Project 5: COCOMO calculator

#

####################################

from costDrivers import Drivers

from getSWParams import SWParams

import re

gccPath = "../projectSpecs/gcc.txt"

linuxKernelPath = "../projectSpecs/linuxkernel.txt"

accessPath = "../projectSpecs/access.txt"

excelPath = "../projectSpecs/excel.txt"

onenotePath = "../projectSpecs/onenote.txt"

powerpointPath = "../projectSpecs/powerpoint.txt"

wordPath = "../projectSpecs/word.txt"

genericPath = "../projectSpecs/generic.txt"

# costDrivers.py contains all the cost drivers, modes, and levels of detail for COCOMO

drs = Drivers()

# development schedule parameters

r = 2.5

s = [0.38,0.35,0.32]

def exeSizeToKLLOC(sizeInBytes):

retVal = (sizeInBytes - 188000) / (28.8314 \* 1000)

return retVal

def runCOCOMO(pathToConfig, filePos, KLLOC=-1, name=""):

settings = SWParams(pathToConfig)

if name == "":

match = re.findall("(/)([^/.]+)(\.txt)",pathToConfig)

name = match[0][1]

# make sure name is capitalized

if name[0] >= 'a' and name[0] <= 'z':

name = chr(ord(name[0]) - 32) + name[1:]

# Compute m as composite of driver files

m = 1

m \*= drs.computeM(settings)

# Effort Formula

# Values come from COCOMO Parameters table, indexing on Mode and Level of Detail

a = drs.COCOMOParam[settings.mode][settings.levelOfDetail][0]

b = drs.COCOMOParam[settings.mode][settings.levelOfDetail][1]

# KLLOC may be passed in for test programs, otherwise it will be computed from the exe size

# Get LLOC from exe size using formula determined in project 3

# Adjust this by a factor of 1000 to get KLLOC

if KLLOC == -1:

KLLOC = exeSizeToKLLOC(settings.exeSize)

else:

# set custom name as well, for generic projects

settings.name = name

Effort = a \* (KLLOC \*\* b) \* m

# Productivity formula

Productivity = KLLOC / Effort

# Development Schedule Formula

devTime = r \* (Effort \*\* s[settings.mode])

# Average Staffing Formula

staffCount = Effort / devTime

cost = settings.salary \* Effort

# round cost to nearest cent

# execute source code, writing results to outfile

with open(outputFilename, "a") as outfile:

# The below line of code seeks to the end of the outfile

outfile.seek(0, 2)

outfile.write(",".join([str(filePos[0]), settings.name, name, str('%.2f'%cost), str('%.2f'%staffCount), str('%.2f'%devTime),str(KLLOC)]) + "\n")

filePos[0]+=1

outputFilename = "../gnuplot/outfile.txt"

# clear outfile

open(outputFilename, 'w').close()

# Output file cursor

filePos = [0]

# execute calculations

# get results for typical projects

runCOCOMO(genericPath,filePos,2,"Small Project")

runCOCOMO(genericPath,filePos,8,"Intermediate Project")

runCOCOMO(genericPath,filePos,32,"Medium Project")

runCOCOMO(genericPath,filePos,128,"Large Project")

runCOCOMO(genericPath,filePos,512,"Very Large Project")

# get results for specific projects

runCOCOMO(gccPath,filePos)

runCOCOMO(linuxKernelPath,filePos)

# Results for Microsoft products

runCOCOMO(accessPath,filePos)

runCOCOMO(excelPath,filePos)

runCOCOMO(onenotePath,filePos)

runCOCOMO(powerpointPath,filePos)

runCOCOMO(wordPath,filePos)

**costDrivers.py -** **Cost Drivers File**

**Stores cost driver values**class Drivers:

# Product Attributes

requiredSoftwareReliability = [0.75,0.88,1,1.15,1.4,0]

dbSize = [0,0.94,1,1.08,1.16,0]

productComplexity = [0.7,0.85,1,1.15,1.3,1.65]

# Computer Attributes

executionTimeConstraint = [0,0,1,1.11,1.3,1.66]

mainStorageConstraint = [0,0,1,1.06,1.21,1.56]

virtualMachineVolatility = [0,0.87,1,1.15,1.3,0]

computerTurnaroundTime = [0,0.87,1,1.07,1.15,0]

# Personnel Attributes

analystCapabilities = [1.46,1.19,1,0.86,0.71,0]

applicationsExperience = [1.29,1.13,1,0.91,0.82,0]

programmerCapability = [1.42,1.17,1,0.86,0.7,0]

virtualMachineExperience = [1.21,1.10,1,0.90,0,0]

programmingLanguageExperience = [1.14,1.07,1,0.95,0,0]

# Project Attributes

useOfModernProgrammingPractices = [1.24,1.10,1,0.91,0.82,0]

useOfSoftwareTools = [1.24,1.10,1,0.91,0.83,0]

requiredDevelopmentSchedule = [1.23,1.08,1,1.04,1.10,0]

# Defines how much the specifications of the product are expected to change

requirementsVolatility = [0,0.91,1,0,0,1.62]

# nesting level: [mode][Level of Detail][ai,bi]

COCOMOParam = [[[2.4,1.05],[3.2,1.05],[2.6,1.08]],

[[3.0,1.12],[3.0,1.12],[2.9,1.12]],

[[3.6,1.2],[2.8,1.2],[2.9,1.2]]]

def computeM(self, settings):

mVal = 1

if settings.levelOfDetail != 0:

mVal \*= self.multiplyIfValid(self.requiredSoftwareReliability,settings.requiredSoftwareReliabilitySetting,0)

mVal \*= self.multiplyIfValid(self.dbSize,settings.dbSizeSetting,1)

mVal \*= self.multiplyIfValid(self.productComplexity,settings.productComplexitySetting,2)

mVal \*= self.multiplyIfValid(self.executionTimeConstraint,settings.executionTimeConstraintSetting,3)

mVal \*= self.multiplyIfValid(self.mainStorageConstraint,settings.mainStorageConstraintSetting,4)

mVal \*= self.multiplyIfValid(self.virtualMachineVolatility,settings.virtualMachineVolatilitySetting,5)

mVal \*= self.multiplyIfValid(self.computerTurnaroundTime,settings.computerTurnaroundTimeSetting,6)

mVal \*= self.multiplyIfValid(self.analystCapabilities,settings.analystCapabilitiesSetting,7)

mVal \*= self.multiplyIfValid(self.applicationsExperience,settings.applicationsExperienceSetting,8)

mVal \*= self.multiplyIfValid(self.programmerCapability,settings.programmerCapabilitySetting,9)

mVal \*= self.multiplyIfValid(self.virtualMachineExperience,settings.virtualMachineExperienceSetting,10)

mVal \*= self.multiplyIfValid(self.programmingLanguageExperience,settings.programmingLanguageExperienceSetting,11)

mVal \*= self.multiplyIfValid(self.useOfModernProgrammingPractices,settings.useOfModernProgrammingPracticesSetting,12)

mVal \*= self.multiplyIfValid(self.useOfSoftwareTools,settings.useOfSoftwareToolsSetting,13)

mVal \*= self.multiplyIfValid(self.requiredDevelopmentSchedule,settings.requiredDevelopmentScheduleSetting,14)

if settings.levelOfDetail == 2:

mVal \*= self.multiplyIfValid(self.requirementsVolatility,settings.requirementsVolatilitySetting,15)

return mVal

# does a check to ensure that a valid driver setting was selected

def multiplyIfValid(self,driverArr, setting, index):

if driverArr[setting] == 0:

print "Invalid setting for m" + str(index)

exit(-1)

else:

return driverArr[setting]

**getSWParams.py**

**Parses project config files to determine project settings**

class SWParams:

name = ""

exeSize = -1

salary = -1

requiredSoftwareReliabilitySetting = -1

dbSizeSetting = -1

productComplexitySetting = -1

executionTimeConstraintSetting = -1

mainStorageConstraintSetting = -1

virtualMachineVolatilitySetting = -1

computerTurnaroundTimeSetting = -1

analystCapabilitiesSetting = -1

applicationsExperienceSetting = -1

programmerCapabilitySetting = -1

virtualMachineExperienceSetting = -1

programmingLanguageExperienceSetting = -1

useOfModernProgrammingPracticesSetting = -1

useOfSoftwareToolsSetting = -1

requiredDevelopmentScheduleSetting = -1

requirementsVolatilitySetting = -1

levelOfDetail = -1

mode = -1

# Constructor

def \_\_init\_\_(self, filepath):

with open(filepath, 'r') as file:

# read a list of lines into source code

txtData = file.readlines()

# get name of software (slice eliminates newline at end)

self.name = txtData[2][:-1]

# get size of program

self.exeSize = int(txtData[5][:-1])

# Get salary per month of software engineers

self.salary = int(txtData[8][:-1])

# get m vars

levelOfDetail = txtData[11][:-1]

# check for valid Level of Detail

if levelOfDetail != '0' and levelOfDetail != '1' and levelOfDetail != '2':

print "invalid level of detail entered in " + filepath

exit(-1)

self.levelOfDetail = int(levelOfDetail)

# get mode

mode = txtData[14][:-1]

# By skipping the first letter, this will work with whatever case is used in the input file

if mode.count("rganic"):

self.mode = 0

elif mode.count("emidetached"):

self.mode = 1

elif mode.count("mbedded"):

self.mode = 2

else:

print "Invalid entry for mode in file " + filepath

# file search cursor

fileCursor = [0]

# get m values as indicated by level of detail

if levelOfDetail == '1' or levelOfDetail == '2':

self.requiredSoftwareReliabilitySetting = parseMVal(txtData, fileCursor, "m1. requiredSoftwareReliability")

self.dbSizeSetting = parseMVal(txtData, fileCursor, "m2. dbSize")

self.productComplexitySetting = parseMVal(txtData, fileCursor, "m3. productComplexity")

self.executionTimeConstraintSetting = parseMVal(txtData, fileCursor, "m4. executionTimeConstraint")

self.mainStorageConstraintSetting = parseMVal(txtData, fileCursor, "m5. mainStorageConstraint")

self.virtualMachineVolatilitySetting = parseMVal(txtData, fileCursor, "m6. virtualMachineVolatility")

self.computerTurnaroundTimeSetting = parseMVal(txtData, fileCursor, "m7. computerTurnaroundTime")

self.analystCapabilitiesSetting = parseMVal(txtData, fileCursor, "m8. AnalystCapabilities")

self.applicationsExperienceSetting = parseMVal(txtData, fileCursor, "m9. applicationsExperience")

self.programmerCapabilitySetting = parseMVal(txtData, fileCursor, "m10. programmerCapability")

self.virtualMachineExperienceSetting = parseMVal(txtData, fileCursor, "m11. virtualMachineExperience")

self.programmingLanguageExperienceSetting = parseMVal(txtData, fileCursor, "m12. programmingLanguageExperience")

self.useOfModernProgrammingPracticesSetting = parseMVal(txtData, fileCursor, "m13. useOfModernProgrammingPractices")

self.useOfSoftwareToolsSetting = parseMVal(txtData, fileCursor, "m14. useOfSoftwareTools")

self.requiredDevelopmentScheduleSetting = parseMVal(txtData, fileCursor, "m15. requiredDevelopmentSchedule")

if levelOfDetail == '2':

self.requirementsVolatilitySetting = parseMVal(txtData, fileCursor, "m16. requirementsVolatility")

def parseMVal(file, cursor, anchor):

lineNo = getLineOfData(file, cursor, anchor)

# in order of likelihood, slice newline off

lineData = file[lineNo][:-1]

if lineData == "nominal":

return 2

elif lineData == "low":

return 1

elif lineData == "high":

return 3

elif lineData == "very low":

return 0

elif lineData == "very high":

return 4

elif lineData == "extra high":

return 5

else:

print "Failed at line " + str(lineNo)

exit(-1)

def getLineOfData(file, cursor, anchor):

# scan for anchor, if found return next line which should have data

for i in range(cursor[0], len(file)):

if file[i].count(anchor):

cursor[0] = i + 1

return cursor[0]

# couldn't find anchor, something bad happened

print "could not find string to match: " + anchor

exit(-1)

**genBarPlots.py – Gnuplot script editor**

**Modifies contents of gnuplot script to generate desired barplot figures**

**A similal file is used to generate scatterplots**

# external library which allows cmd calls

from subprocess import call

import os

pathToPlotter = ["../gnuplot/","barplot.plt"]

origDir = "original/"

###

# Function Section

###

def getLineOfData(file, cursor, anchor):

# scan for anchor, if found return next line which should have data

for i in range(cursor[0], len(file)):

if file[i].count(anchor):

cursor[0] = i + 1

return cursor[0]

# couldn't find anchor, something bad happened

print "could not find string to match: " + anchor

exit(-1)

def editPlotFile(plotFileLines, pngName, title, ylabel, dataCol):

# use cursor to scan file lines for correct anchors

cursor = [0]

# Set name of output png

lineNo = getLineOfData(plotFileLines, cursor, "define output")

plotFileLines[lineNo + 1] = "".join(['set output "', pngName, '.png"\n'])

# Set title and ylabel of plot

lineNo = getLineOfData(plotFileLines, cursor, "set title, labels, key position")

plotFileLines[lineNo] = "".join(['set title "', title, '"\n'])

plotFileLines[lineNo + 1] = "".join(['set ylabel "<-- ', ylabel, ' -->"\n'])

# set data to use

lineNo = getLineOfData(plotFileLines, cursor, "plot barplot")

plotFileLines[lineNo + 1] = "".join(['1:', str(dataCol), ':xtic(3) \\\n'])

def genPlot(pngName, title, ylabel, dataCol):

# open backup read file to avoid destruction by writing

with open("".join([pathToPlotter[0], origDir, pathToPlotter[1]]), 'r') as file:

# read a list of lines into source code

plotData = file.readlines()

# make necessary changes to Gnuplot script

editPlotFile(plotData, pngName, title, ylabel, dataCol)

# write lines back to appropriate plot file in gnuplot directory

with open("".join([pathToPlotter[0],pngName,".plt"]), 'w') as file:

# write back to gnuplot file

file.writelines(plotData)

# Switch to working dir to /Gnuplot, and run gnuplot on appropriate plot file

CWD = os.getcwd()

os.chdir(os.path.join(CWD, pathToPlotter[0]))

call(["gnuplot", "".join([pngName,".plt"])])

# return to original directory

os.chdir(CWD)

###

# Execution Section

###

genPlot("cost", "Cost Required for Project", "Cost ($)", "4")

genPlot("staff", "Staff Required for Project", "Staff", "5")

genPlot("time", "Development Time Required for Project", "Time (Months)", "6")

genPlot("KLLOC", "Project KLLOC", "KLLOC", "7")

# Appendix: Gnuplot Scripts

**Barplot Generator**

# Samuel Gluss

# 5-10-2016

# Barplot Generator

reset

# define output

set terminal pngcairo size 800,550 enhanced font 'Verdana,10'

set output "staff.png"

# set delimiter

set datafile separator ","

# set title, labels, key position

set title "Staff Required for Project"

set ylabel "<-- Staff -->"

set key off

set grid ytics lw 2 lc rgb "#868686"

set xtics border in scale 0,0 nomirror rotate by -45

# adjust the border

set style line 11 lc rgb '#404040' lt 1

set border 3 back ls 11

set tics nomirror

set boxwidth 0.5

set style fill solid border 0

# plot barplot

plot "outfile.txt" using \

1:5:xtic(3) \

with boxes lc rgb "blue"

**Scatterplot Generator**

# Samuel Gluss

# 5-10-2016

# define output console

set terminal pngcairo size 800,550 enhanced font 'Verdana,10'

set output "costvsKLLOC.png"

# set delimiter

set datafile separator ","

# set title, axis labels

set title "A graph"

set xlabel "<-- KLLOC -->"

set ylabel "<-- (y) -->"

# configure legend

set key top left title 'Legend' box 3

# show axes aligned on origin

set zeroaxis

# define function

a = 10.0

f(x) = a \* x\*\*2

# plot scatterplot

plot "outfile.txt" using \

7:4 \

with points pt 3 ps 2 title 'Data Points'