August 2014

Volume 5 Issue 1

JoCsE

Journal Of Computational Science Education

Promoting the Use of Computational Science Through Education

ISSN 2153-4136 (online)

Volume 5, Issue 1 **August 2014**

JOCSE

Journal Of Computational Science Education

The Journal Of Computational Science Education (JOCSE), ISSN 2153-4136, is published quarterly in online form, with more frequent releases if submission quantity warrants, and is a supported publication of the Shodor Education Foundation Incorporated. Materials accepted by JOCSE will be hosted on the JOCSE website, and will be catalogued by the Computational Science Education Reference Desk (CSERD) for inclusion in the National Science Digital Library (NSDL).

Subscription: JOCSE is a freely available online peer-reviewed publication which can be accessed at `http://jocse.org`.

Contents

Introduction to Volume 5 Issue 1

Steven I. Gordon
Editor
Ohio Supercomputer Center
Columbus, OH
sgordon@osc.edu

Forward

The articles in this issue of the Journal of Computational Science Education touch on an array of approaches to learning both computational techniques and science concepts. They present examples that teach concepts using workstation environments and using parallel computing architectures.

The article by Winfield et. al. discusses the use of several tools associated with computer aided drug design and how they were integrated into the undergraduate curriculum. They describe the software they used and the case study assignment. They then provide data on the learning outcomes associated with the course.

Joiner and Walters provide a description of a new version of GalaxSeeHPC that can be used for large scale galactic dynamics simulations. After reviewing the technical changes in the software, they outline results of several example simulations and scenarios for students to investigate the structure of galaxies.

Akman provides a framework for introducing the use of genetic algorithms for teaching concepts in regression analysis and optimization of regression models. He presents an Excel macro that can be used as a teaching tool to demonstrate the how the genetic algorithm can be used to arrive at an approximation of the best solution to a multiple linear regression problem.

Finally, Dillon, Anderson-Herzog, and Brown discuss the pros and cons of using visual programming environments for teaching introductory programming skills. They compare the trade-off between the easier learning curve in such an environment with the possibility of misconceptions about the steps required to program, compile, and execute a program in a command line environment.

Characterizing Ligand Interactions in Wild-type and Mutated HIV-1 Proteases

Leyte L. Winfield
Spelman College
Department of Chemistry & Biochemistry
350 Spelman LN, SW, Box 231
Atlanta, GA 30314
Voice: (404) 270-5748
lwinfield@spelman.edu

Rosalind Gregory-Bass
Spelman College
Department of Biology
350 Spelman LN, SW, Box 1183
Atlanta, GA 30314
RBass@spelman.edu

Jordan Campbell
Spelman College
Department of Chemistry & Biochemistry
350 Spelman LN, SW, Box 231
Atlanta, GA 30314
j.campbell529@gmail.com

Andy Watkins
New York University
Department of Chemistry
100 Washington Square East
New York, New York 10003
andy.watkins2@gmail.com

ABSTRACT
A computational module has been developed in which students examine the binding interactions between indinavir and HIV-1 protease. The project is a component of the Medicinal Chemistry course offered to upper level chemistry, biochemistry, and biology majors. Students work with modeling and informatics tools utilized in drug development research while evaluating wild-type and mutated forms of the HIV-1 protease in complex with the inhibitor indinavir. By quantifying the molecular interactions within protease-inhibitor complexes, students can characterize the structural basis for reduced efficacy of indinavir.

Categories and Subject Descriptors
K.3.1 **[Computer Uses in Education -Computer-assisted instruction (CAI)]**

General Terms Measurement
Human Factors, Measurement

Keywords
HIV, molecular modeling, indinavir, computational lab, drug resistance, mutations, protease, ligand interactions, cheminformatics

1. INTRODUCTION

1.1 Computer Aided Drug Design in the Undergraduate Curriculum

Computers have been utilized throughout chemistry and biology curricula to visualize molecular information and simulate fundamental concepts. To this end, visualization and simulation exercises have been useful for introducing undergraduates to biomolecular interactions associated with drug activity. Early examples of such activities involved the use of Microsoft Excel to conduct quantitative structure activity relationship (QSAR) analysis [1]. Recent efforts have focused on the use of supercomputing, molecular modeling software, and informatics tools to allow students to see the benefit of such technologies in explaining biological processes such as protein function as well as the molecular rationale for drug action [2-6]. Two specific examples allow undergraduate students to visualize the conserved regions of selected kinases and observe the impact of site-directed mutagenesis [7, 8]. In addition, computational modules have been developed for the graduate and K12 curriculums as well. In the graduate curriculum, computational tools have been used to assist pharmacy students understand drug-receptor interactions using Pymol and the Keele Active Virtual Environment (KAVE) [9, 10]. However, students primarily observed the three-dimensional molecules assigned by the instructor and were provided with little opportunity to explore the function of the software. In the K-12 classroom, learning strategies involving molecular visualizations have been used to address the relevance of biomolecules to everyday life [11].

The preceding examples focus primarily on visualizing biological target. An activity more directly related to what is considered rational drug design and development involves assessing the efficacy of inhibitors for various diseases using Autodock and Pymol [12]. The pharmacological aspects of drug action (i.e. drug absorption, metabolism, and excretion-ADME) where exploited by Kim et. al. who implemented a computational experiment in which the physiochemical parameters of known drugs were utilized to predict the ability of the drugs to cross the blood-brain

barrier [13]. Similar to this, undergraduates completing a medicinal chemistry course in Australia created mathematical models that demonstrate the relationship of the known activities of adrenoceptor ligands to their calculated physicochemical properties [14].

The HIV-1 module presented here represents an option for introducing multiple aspects of computer-aided drug design and related software tools. The module goes beyond the visualization of biomolecular structures and simulation of their function, the primary focus of many of the published computational activities. The module allows students to calculate physiochemical parameters of indinavir, Figure 1, to gain insight into its binding interactions with the wild-type and mutated forms of HIV-1 protease. Basic drug design terminology was assessed via a quiz and rubric graded reports. A survey was administered to assess the success of the module.

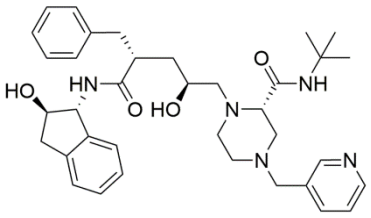


Figure 1: Indinavir

1.2 HIV-1 Protease

The HIV-1 protease is a key modulator of the HIV lifecycle [15, 16]. The proteolytic enzyme, comprised of 99 amino acid residues, functions to cleave the mature virus leading to its activation. Its activity is characterized by a base mediated amide hydrolysis that is catalyzed by two aspartate residues, ASP 25 and ASP 25'. The anionic form of the aspartate residue is required for the reaction to occur. Based on the mechanism shown in Figure 2, the active conformation of the enzyme is stabilized by a tetrahedral-like transition state [17]. Note that Brik and Wong provide a more acceptable mechanism that is concerted. However, the electron pushing scheme is not clearly delineated and doesn't coincide with what is understood at the undergraduate level. Protease inhibitors contain polar groups that facilitate van der Waals interactions within the active site of the enzyme, allowing the inhibitor to mimic the binding of the natural substrate and stabilize the transition state [18]. Such interactions prevent the enzyme from hydrolyzing the precursor polypeptides. Therefore, inhibiting the protease prevents the activation of the retrovirus and subsequent spread of HIV. Like other protease inhibitors, indinavir acts as a peptidomimetic. The molecule binds to the protease by forming a water mediated interaction to Ile50 and Ile50', inducing the active conformation of the enzyme and preventing the protease from interacting with its natural substrate.

1.3 Indinavir

Indinavir is one of nine drugs, not counting fosamprenavir which is hydrolyzed in the stomach to form amprenavir, licensed for the treatment of HIV-1 [19]. Despite the success of treatments with drugs like indinavir, HIV is a disease that quickly adapts to ensure its survival resulting in therapeutic resistance to many drugs. In the case of indinavir, its effectiveness against HIV-1 can be reduced due to mutations of only one amino acid residue. The mutation alters the conformation of the protease and its interactions with indinavir [20]. This module allows students to explore the three-dimensional structure of indinavir in complex with various mutated forms of the HIV-1 protease. Students utilize available X-ray data to observe the conformational differences between three indinavir-HIV-1 protease complexes (wild-type and two mutant complexes), gaining insight into the molecular characteristics of drug resistance. In addition, students are introduced to molecular modeling tools commonly used in the field of medicinal chemistry.

2. SYSTEM REQUIRMENTS

2.1 Hardware Requirements and Software Installation

Tools found in the Molecular Operating Environment (MOE, Chemical Computing Group, http://www.chemcomp.com/) software, were utilized to accomplish modeling and computational tasks. The software is available free of charge for institutions that have an academic research license which costs approximately $2,500 annually. However, the exercise can be adapted for use with other software including Sybyl (http://tripos.com/index.php) and Maestro/MacroModel (http://www.schrodinger.com/) which range in price. Some aspects can also be completed using Pymol (http://pymol.org/educational/) and MarvinSpace with appropriate calculator plugins (http://www.chemaxon.com/products/). The latter two are limited in their ability to manipulate the three-dimensional structure of the protease and measure binding energies as describe herein, but are free for academic use. Instructions for completing the module using MOE, Maestro/MacroModel, and Pymol are included in the appendix. However, the results reported here were generated using MOE. The software, distributed to the students on USB flash drives along with license keys, runs on both Windows (XP, Vista, or Windows 7) and Mac OS X (10.5 or higher) operating systems. A three-button mouse is strongly suggested for manipulating the controls in the software; laptop users should use an external mouse as the mouse pad will not provide the needed functionality. MOE operates using a server-based license and is only functional on systems with access to the server. Despite the limit of mobile use, the automated functions (structure rendering, sequence alignments, energy minimization, protonation, etc.) make the software ideal for introducing informatics and computational methodologies to students.

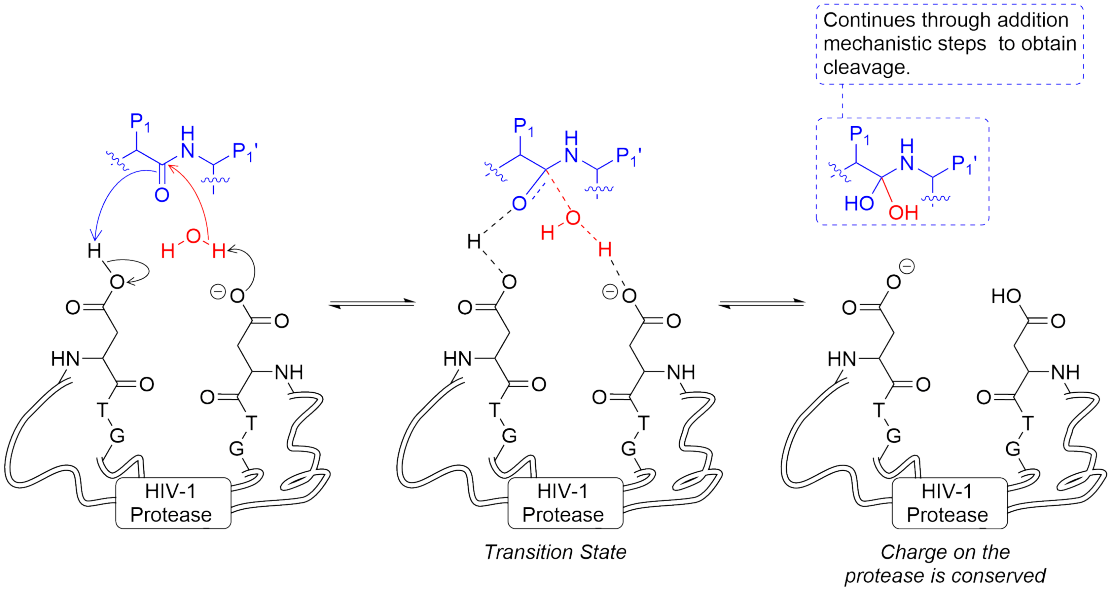
2.2 Molecular Structure Files

The known three-dimensional structures of wild-type and mutant proteins and their ligands can be obtained from the Research Collaboratory for Structural Bioinformatics (RCSB) Protein Data Bank (PDB, http://www.rcsb.org/pdb/home/home.do). The X-ray data of the co-crystallized protease-ligand complexes used in this module were PDB codes 1SDT (wild-type), 1SDU (L90M mutant), and 1SDV (V82A mutant). The binding energies and inhibitory activity of indinavir in each complex were obtained from the RCSB Protein Data Bank as well. With the exception of adding protons, all computational analyses are based on the conformations of the complex imported from the RCSB Protein Data Bank.

3. DESCRIPTION OF COMPUTATIONAL MODULE

3.1 Course Description

Figure 2: Proposed mechanism and tetrahedral-like transition state for the proteolytic activity of the HIV-1 protease.



To date, the HIV module has been completed by three cohorts of students (average cohort size = 7). The module is a component of the Medicinal Chemistry course, a one semester advanced elective offered to junior and senior chemistry, biochemistry, and biology majors. The course introduces students to computer-aided drug design (CADD) and cheminformatics tools utilized to characterize the molecular aspects of the disease. The course follows a blended format in which student led discussions and computational modules are the primary modes of instruction. The discussions are moderated by the instructor to ensure the correct information is being shared and to interject additional information as needed. The discussions are used to share information on the history of compounds with medicinal properties; the political, economic, environmental, and scientific frameworks for defining diseases; and the key concepts associated with drug design. Students are required to review information from the New York Times, scientific journals, and the Howard Hughes Medical Institute (HHMI) website in preparation for the discussions. For the computational components of the course, the instructor provides a brief overview on the purpose of the project and mini tutorials on the use of relevant software and databases.

3.2 Module Structure

The HIV module is completed over three weeks, each week containing two 2-hour class periods. The time was used for data collection and discussion of background information and results. Before beginning the computational component, students completed a literature search to identify background information and define key terms related to HIV/AIDS. Students were also required to watch the "HIV Lifecycle" and "Protease Inhibitors" found on the HHMI website [15, 16]. In addition, students were provided with information on the mechanism which leads to the photolytic activity of the molecule, described in the introduction. Students were asked to analyze indinavir in complex with the wild-type HIV-1 protease and the two mutated proteases. The L90M (non-binding site mutation) and the V82A (binding site mutation) complexes were utilized. For the purpose of this assignment, it is sufficient for students to explore one mutated complex, but both should be examined if time permits. The mutated complexes are compared to the wild-type complex to understand how conformational changes reduce drug affinity and potentially reduce drug efficacy.

3.2.1 Examining Binding Site Interactions

Using the LigX function, protons were added to the three-dimensional structure of each inhibitor-protease complex and the overall conformation was optimized (root mean square deviation, RMSD, of the final conformations were < 0.2 Å from the original ligand in each case). The software assigns ionization states and adds hydrogens to structures based on the steric environment and protonation state of the chemical groups [21]. Standard force field parameters defined by Merck Molecular Force Field (MMFF94) were used to calculate the potential energy of the inhibitor and its affinity for the protease. The method is typically applied to calculations involving small molecules and provides suitable values for the type of comparative analysis conducted in this assignment. Key interactions between the ligand and the wild-type protease were identified and characterized as hydrogen bonding interactions (based on interaction distances and angles) or non-hydrogen bonding interactions. The cut-off for potential ligand-receptor interactions was set at 4.5 Å for non-hydrogen bonding interactions with hydrogen bonding being defined at 1.4 – 2.2 Å. The *in vitro* binding energy ($\Delta G°$) for the indinavir-protease complex was provided to the students to define the relative ability of the ligand to bind to each complex in an energetically favorable conformation. Students also received data on the *in vitro* inhibitory activity of indinavir (K_i).

3.2.2 Comparing the Conformations of Indinavir

The two protease files were opened in the MOE window. Each ligand structure was rendered in tube formula and given a distinct color. The conformations of the ligand-receptor complexes were then superposed. To accomplish this, the software treats the structures as rigid bodies and aligns the structures based on the three-dimensional trace of the alpha helices [22]. The receptor was hidden to show the structural overlay of the two inhibitor conformations. Students measured the distance between the indane and benzene rings and the dihedral angle as defined by the pyridine

and piperazine rings to quantify the differences between the two conformations. Students also examined the conformational differences between the wild-type and the mutated receptors by measuring intramolecular receptor interactions and the change in the water-mediated interaction between Ile50 and Ile50'. Finally, students identified the mutation present in their assigned complex and determined if the mutation occurred in the binding site of the protease.

3.3 Evaluation

Quiz questions, student reports, and survey responses were utilized as a preliminary gauge of the success of the module. All assessment materials were developed by the authors. The quiz contained a mixture of multiple choice and short answer questions created to determine students' basic concepts of drug design and HIV-1. The maximum score on each question was scaled to 1 point. Reports submitted by students were assessed using a rubric to determine their level of comprehension and ability to analyze the data collected in the completion of the module. The survey was administered through Survey Monkey at the completion of the course in years two and three of the module being offered. Part 1 of the survey contained fifteen questions that were posed to assess students' previous knowledge of drug design, attitudes towards course resources and technology, and future interest in research and drug design. Part 2 of the survey contained nine questions developed to measure students' confidence with the concepts and technology associated with the module. Anecdotal observations, gathered informally through class discussions and student reports, were documented to complement the survey results. The assessment of this module will be ongoing for future cohorts.

4. RESULTS AND DISCUSSION

4.1 Computational Module

Students submitted a report containing a summary of their results and selected images. The measurements obtained for the HIV-1 protease-indinavir complex along with the *in vitro* binding energies (ΔG°) of indinavir in complex with the protease and the concentration of indinavir needed to inhibit protease activity (inhibition constant, K_i) are given in Table 1 [20, 23]. The energy of the interactions of indinavir with the protease, calculated binding energy (Table 1), was measured using standard force field parameters defined by Merck Molecular Force Field (MMFF94). The values were used by the students to characterize interactions that reduce the affinity of the drug for the protease. The calculated binding energies of indinavir in each complex correlate well with the experimental data (ΔG° and K_i). There were cases in which students did not calculate the energies appropriately and did not see the expected trend. Students reporting this discrepancy were able to communicate it as an unexpected outcome. The number of students that did not correctly calculate the energies is accounted for in section 4.2.2.

Table 1. Sample of Student Data

HIV-1 PR	K_i (nM)	ΔG° (kJ/mole)	Calculated Binding Energy (kcal/mole)
Wild type	0.60	-54.75	-12.14
mutant L90M	0.80	-54.01	-11.78
mutant V82A	1.34	-52.68	-11.55

Students characterized the conformational changes of both the protease and the inhibitor that account for varied affinity towards binding to each protease. It should be noted for students that MOE classifies the B chain of the protease different from the notation commonly used (i.e. Ile50' is annotated in the software as Ile150). The water mediated interaction between Ile50, Ile50', and indinavir (Figure 3) is believed to stabilize the active transition state of the protease. To calculate the angle of the interaction, hydrogen atoms were ignored. The angle, designated by the green dashed-lines in Figure 4, was defined using a nitrogen atom (of Ile50 or Ile50'), the oxygen of the water molecule, and a carbonyl oxygen of indinavir. This produced two water mediated interaction between indinavir and the protease. In the transition state described in Figure 2, the protease engages in a water catalyzed reaction with its natural substrate resulting in the cleavage of acyl bond. Inhibitors have been designed to bind to the protease in a similar fashion without a subsequent cleavage of the acyl bond. The interactions hold the protease in its active conformation and prevents the binding of the natural substrate. Therefore, the water-mediated interactions define an ideal orientation of Ile50 and Ile50' in the conformation of the protease when an inhibitor is present. The angles formed with Ile50' in the mutated proteases are wider than those formed in the wild-type protease, indicating the mutation causes a conformational change that prevents the inhibitor for occupying an optimal binding orientation when interacting with the protease. The change in the degree of the angle correlates well with the calculated affinities and the *in vitro* binding energies. Students identified the location of the mutation and noted that when the mutation occurred in the binding site (mutant V82A), the loss of affinity was more significant than when the mutation was at another point on the protease (mutant L90M). Based on the additional data collected (see supplemental documentation), students were able to elaborate on these observations in the context of drug resistance; in particular, how variations in the angles of these interactions could explain the differing activities of indinavir in the wild-type and mutated forms of the HIV-1 protease.

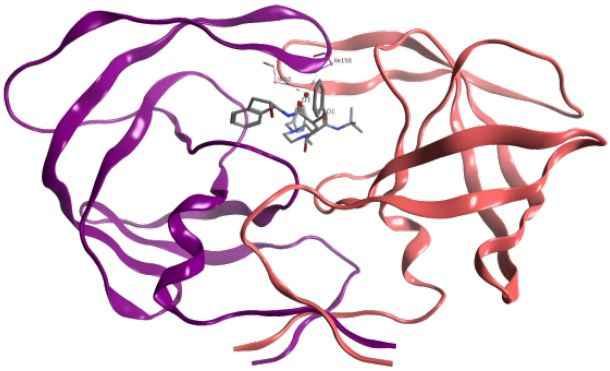
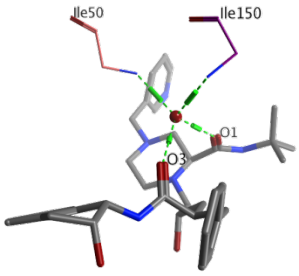


Figure 3: Relative Position of the Indinavir Binding Site in the HIV-1 Protease. Chain A in pink ribbon, Chain B in purple ribbon, indinavir grey tube formula, and protease residues in stick formula.



HIV-1 PR	*Interactions with Ile50 (°)* O1	O3	*Interactions with Ile50' (°)* O1	O3	Change (°)	Binding site mutation?
Wild-type	106.6	130.4	103.8	92.8	–	–
Mutant L90M	101.5	135.0	110.8	102.6	6.6	no
Mutant V82A	100.1	116.3	123.3	108.8	9.7	yes

Figure 4. Ile50' (purple stick formula) and Ile50 (pink stick formula) interact with indinavir via a single water molecule, green dashed lines. The interaction is defined by an atom on the protease residues (the nitrogen of Ile50 or Ile50'), an atom from indinavir (one of the carbonyl oxygen atoms), and the oxygen of the water molecule (represented as a red ball). O1 and O2 represent oxygen atoms 1 and 2, respectively, of indinavir in each complex. The average change is calculated as the difference is the angle between the atom of the wild-type protease for a given residue and that of the mutated proteases.

4.2 Assessment and Evaluation

4.2.1 Evaluation of Student Learning

To formally assess students' knowledge and comprehension of basic terminology, a quiz was administered. The assessment contained a mix of multiple choice and short answer questions (Table 1). In general, students' performed well on the quiz. The lowest performance was on the question of hydrogen bonding in drug action. Nevertheless, most students were able to elaborate on the need for electrostatic interactions to facilitate binding to the biological target. Assessment of student lab reports provided additional insight into students' conceptual understanding of the terminology. The reports were evaluated using a rubric which addressed the four questions given in Table 2. Students utilized information from assignments completed outside of class (literature search, assigned readings, and videos) and the group discussions to develop the background section of the reports. Most students were able to identify the appropriate literature and write an introduction that was relevant to the purpose of the assignment (Question 1). Although the transition state formed by the protease and its natural ligand was discussed and was the focus of the assignment, some students were unable to accurately describe its formation in their report (Question 2). Nevertheless, most students were able to collect accurate data and describe the correlation between the calculated binding energies and the actual binding energies or inhibitory activity (Question 3). In addition, students were able to correctly identify at least 80% of the ligand atoms and their corresponding binding interactions with the protease residues. Further, they were able to classify the interactions as water mediated or non-water mediated and identify the ligand atom as an H-bond donor or acceptor. This is especially important as hydrogen bonding is key to indinavir's ability to bind to the protease, as is the case for many protein-drug interactions.

4.2.2 Evaluation of Course Structure

A survey was used as a preliminary gauge of the benefits and success of implementing the module. Seven of the seventeen students from cohorts two and three responded to the survey. Part 1 of the survey (Table 3) addressed students' prior knowledge of course content, attitude towards the course, and future career goals and professional development activities. Most students indicated having little knowledge of drug design concepts or experience with computational modeling prior to the course. The module was designed to be a self-guided assignment, but the students received a tutorial from the instructor on how to work within the software. In addition, students could receive assistance from the instructor as needed, and were encouraged to work

Table 1. Post-Assessment of Student's Knowledge of Basic Terminology. Score (mean) represents the average score earned by students on a given question where 1 = 100%.

Question	Score (mean)	Standard Deviation
1. Excluding solubility as a reason, name two ways that H-bonding impacts drug activity	0.50	0.47
2. List the characteristics that describe the structure of a molecule.	0.60	0.44
3. Explain why we need new drugs	0.80	0.42
4. Define syndrome and explain why HIV/AIDS was initially identified as a syndrome and not a disease.	1.00	0.00
5. What type of intermolecular force is illustrated in the bonding interaction shown?	1.00	0.00
6. Which molecule has the best therapeutic index?	1.00	0.00
7. Define CADD.	0.80	0.42
8. Why is it important to evaluate mutation in drug targets?	0.75	0.35
9. Which molecule is most potent?	0.90	0.32
10. Which molecule is most efficacious?	0.90	0.32

Table 2. Assessment of Student Reports. Score (mean) represents the average score earned by students on a given question. Scores were assigned based on excellent = 4, good = 3, fair = 2, and poor = 1.

Question	Score (mean)	Score (median)	Standard Deviation
1. Summary of background information related to the assignment including the significance of the drug indinavir and the purpose/objective of the module.	3.42	4.00	0.86
2. Summary of the mechanism of action for the HIV-1 protease and description of related transition state.	2.50	2.50	1.26
3. Explanation of the correlation of the calculated energies to the known values.	2.58	3.00	1.38
4. Identification of ligand- protease interactions (identify the interacting ligand atoms as H-bond donors or acceptors and protease residues) and the classification of the interactions as water mediate or non-water mediate.	2.67	3.00	1.31

Table 3. Survey Part 1. Score (mean) represents students' self-reported level of agreement with each question, weighed on a scale of 0 to 4 (disagree = 0, somewhat disagree = 1, neither agree nor disagree = 2, somewhat agree = 3, and agree = 4).

Questions	Score (mean)	Standard Deviation
1. I knew about drug design before taking this course.	2.14	1.77
2. I had taken a computational chemistry course or had conduct research in the area prior to taking the course.	1.86	2.04
3. This course allowed me to analyze and interpret data; think critically.	3.71	0.49
4. This course challenged me to think and work "outside of the box."	3.29	0.76
5. It was useful to work with other students in a laboratory group.	3.14	1.07
6. The group discussions with the professor were useful.	3.14	1.46
7. The topics of this course were engaging.	3.71	0.49
8. I enjoyed the computational aspects of the course.	2.57	1.81
9. The course material provided sufficient detail to support self-guided and project-based learning.	2.57	1.62
10. The handouts were easy to follow.	2.71	1.38
11. The software was difficult to use.	2.57	1.51
12. I am now interested in research in the area of drug design and medicinal chemistry.	3.14	1.46
13. I may consider graduate school now.	2.57	1.90
14. This course is useful to students going to health professional or graduate school in STEM.	3.86	0.38
15. This course made me consider pursuing a summer research experience.	3.43	1.51

Table 4. Survey Part 2. Score (mean) represents students' self-reported level of confidence with concepts and technology related to the module, weighed on a scale of 4 to 1 (I am familiar with and understand the concept = 4, I am familiar with and somewhat understand the concept = 3, I am familiar with, but do not understand the concept = 2, I am not familiar with the concept = 1).

Question	Score (mean)	Standard Deviation
1. List the source of drugs	3.00	1.15
2. Explain the need for new drugs	3.29	0.49
3. Define CADD	2.43	1.13
4. Utilize the MOE software to draw 3D molecules	2.86	1.21
5. Utilize the MOE software to create a structural overlay	2.43	1.27
6. Utilize the MOE software to change the rendering of a protein and label residues	2.57	1.13
7. Utilize the MOE software to evaluate binding sites and ligand binding interactions	2.57	1.40
8. Describe the mechanism of action of protease inhibitors	2.71	1.60
9. Understand structurally what occurs when indinavir binds to the HIV-1 protease	2.71	1.11

collaboratively in completion of the work. Based on responses and informal conversations with students, key dynamics associated with access to and ease of use of the software must be considered. In informal conversations, students expressed resistance to only having access to the software while on campus. There were also difficulties with executing certain commands when students attempted to use the software on a Mac. There was also difficulty using the software without a three-button mouse. Therefore, convenience played a role in the students' level of intimidation and comfort with this project-based assignment. Nevertheless, most students found the module to be straight-forward and engaging once overcoming the initial difficulties of learning to use the software. One student stated, "This was an overall successful lab. The MOE software was difficult to understand at first. After the initial confusion, the data gathered from MOE was very interesting." Students had some difficulty with depth perception when looking at the three-dimensional structure, a difficulty that prevented them from being able to select the appropriate atoms when measuring distances and angles. Students also expressed difficulties manipulating the three-dimensional structure utilizing the various mouse functions to move and rotate the complex, which is not unusual for most individuals when first learning to work in a three-dimensional environment. Nevertheless, it initially caused students to have trepidation about completing the assignment.

In Part 2 of the survey, Table 4, students reported their level of understanding of the concepts. Across all questions, the majority of students reported a basic understanding of the concepts. Due to the small number of respondents, a formal statistical analysis was not performed. Since this is a preliminary assessment, interpretations of the outcomes should be cautious and cannot be generalized as a final measure of the success of this module. It is believed that exploring teaching strategies aimed at enhancing students' level of comfort with the software and decreasing intimidation with project-based learning will lead to the sustainability of this and other computational modules.

5. Final Thoughts

A major objective of the exercise was to introduce tools used in the design and virtual screening of medicinal molecules. The module illustrates for students how computational studies enhance the understanding of a drug's mechanism of biological systems. It is important to point out to students that computational models cannot stand alone in describing drug interactions, but should be used in context with experimental data. In a follow-up project, students might substitute the structural units of indinavir with an appropriate bioisostere/isostere and compare the interaction and energies of the modified structure to that of indinavir. Based on the theoretical results of such structural modifications, students can predict if the change would improve or reduces the affinity of the inhibitor for either of the HIV-1 proteases with respect to indinavir. Alternatively, students could mutate the protease further and determine the stability of the resulting indinavir-receptor complex in comparison to that of the wild-type protease. The procedure could be utilized to explore mutations and decreased drug effectiveness in *Mycobacterium tuberculosis* Protein Kinase B (MTB pknB), a Ser/Thr kinase which impacts cell growth and division in the disease. Doing so would expand the discussion of infectious diseases. Available PDB codes for this purpose are 1MRU, 3ORK, and 3ORK.

6. ACKNOWLEDGEMENTS

Contributions to and support for the development of this module was provided by the Faculty Resource Network and Dr. P. Arora, New York University (National Science Foundation Award No**.** CHE1151554 – Education Supplement). A.M.W. thanks New York University for the Kramer Fellowship. In addition, the HIV module is connected to a college-wide initiative at Spelman College to promote cross-divisional collaborations and interdisciplinarity. Drs. Marrionette C. Holmes and Shanie Harris are members of the team developing academic content related to HIV and AIDS.

This investigation has been funded in part by a grant from National Science Foundation Historically Black Colleges and Universities–Undergraduate Program (HBCU-UP) Grant Award No. 0714553, Advancing Spelman's Participation in Informatics Research and Education (ASPIRE), and National Science Foundation Award No. CHE1151554, Education Supplement. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies listed.

There are no conflicts of interest to declare.

7. REFERENCES

[1] Muranaka, K. Anticancer activity of estradiol derivatives: A quantitative structure-activity relationship approach. *J Chem Educ*, 78, 10 (Oct 2001), 1390-1393.

[2] Badotti, F., Barbosa, A. S., Reis, A. L. M., do Valle, I. F., Ambrosio, L. and Bitar, M. Comparative modeling of proteins: A method for engaging students' interest in bioinformatics tools. *Biochem Mol Biol Edu*, 42, 1 (Jan 2014), 68-78.

[3] Carvalho, I., Borges, A. D. L. and Bernardes, L. S. C. Medicinal chemistry and molecular modeling: An integration to teach drug structure-activity relationship and the molecular basis of drug action. *J Chem Educ*, 82, 4 (Apr 2005), 588-596.

[4] Hayes, J. M. An Integrated Visualization and Basic Molecular Modeling Laboratory for First-Year Undergraduate Medicinal Chemistry. *J Chem Educ*, 91, 6 (Apr 2014), 919-923.

[5] Roy, U. and Luck, L. A. Molecular modeling of estrogen receptor using molecular operating environment. *Biochem Mol Biol Edu*, 35, 4 (Jul-Aug 2007), 238-243.

[6] Rudnitskaya, A., Torok, B. and Torok, M. Molecular docking of enzyme inhibitors: A computational tool for structure-based drug design. *Biochem Mol Biol Edu*, 38, 4 (Jul-Aug 2010), 261-265.

[7] Chiang, H., Robinson, L. C., Brame, C. J. and Messina, T. C. Molecular mechanics and dynamics characterization of an in silico mutated protein: A stand-alone lab module or support activity for in vivo and in vitro analyses of targeted proteins. *Biochem Mol Biol Edu*, 41, 6 (Nov 2013), 402-408.

[8] Jones, M., Shoffner, R. and Friesen, J. Use of computer modeling of site-directed mutagenesis of a selected enzyme: A class activity for an introductory biochemistry course. *J Sci Edu Technol*, 12, 4 (Dec 2003), 413-419.

[9] Richardson, A., Bracegirdle, L., McLachlan, S. I. H. and Chapman, S. R. Use of a three-dimensional virtual environment to teach drug-receptor interactions. *Am J Pharm Educ*, 77, 1 (Feb 2013).

[10] Satyanarayanajois, S. D. Active-Learning Exercises to Teach Drug-Receptor Interactions in a Medicinal Chemistry Course. *Am J Pharm Educ*, 74, 8 (Oct 2010).

[11] Bethel, C. M. and Lieberman, R. L. Protein structure and function: An interdisciplinary multimedia-based guided-inquiry education module for the high school science classroom. *J Chem Educ*, 91, 1 (Jan. 2014), 52-55.

[12] Franco, D. T. a. J. Using Supercomputing to Conduct Virtual Screen as Part of the Drug Discovery Process in a Medicinal Chemistry Course. *J Comp Sci Educ*, 3, 2 (Feb 2012), 18-25.

[13] Kim, H., Sulaimon, S., Menezes, S., Son, A. and Menezes, W. J. C. A Comparative Study of Successful Central Nervous System Drugs Using Molecular Modeling. *J Chem Educ*, 88, 10 (Oct 2011), 1389-1393.

[14] Manallack, D. T., Chalmers, D. K. and Yuriev, E. Using the beta(2)-Adrenoceptor for Structure-Based Drug Design. *J Chem Educ*, 87, 6 (Jun 2010), 625-627.

[15] *HIV Lifecycle*. HHMI BioInteractive Animation, City, 2007. http://www.hhmi.org/biointeractive/disease/hiv_life_cycle.html

[16] *Protease Inhibitors*. HHMI BioInteractive Animation, City, 2007. p://www.hhmi.org/biointeractive/disease/protease_inhibitor.html

[17] Brik, A. and Wong, C. H. HIV-1 protease: mechanism and drug discovery. *Organic & Biomolecular Chemistry*, 1, 1 (Jan 2003), 5-14.

[18] Drag, M. and Salvesen, G. S. Emerging principles in protease-based drug discovery. *Nat Rev Drug Discov*, 9, 9 (Sep 2010), 690-701.

[19] Ali, A., Bandaranayake, R. M., Cai, Y. F., King, N. M., Kolli, M., Mittal, S., Murzycki, J. F., Nalam, M. N. L., Nalivaika, E. A., Ozen, A., Prabu-Jeyabalan, M. M., Thayer, K. and Schiffer, C. A. Molecular basis for drug resistance in HIV-1 protease. *Viruses-Basel*, 2, 11 (Nov 2010), 2509-2535.

[20] Mahalingam, B., Wang, Y. F., Boross, P. I., Tozser, J., Louis, J. M., Harrison, R. W. and Weber, I. T. Crystal structures of HIV protease V82A and L90M mutants reveal changes in the indinavir-binding site. *Eur J Biochem*, 271, 8 (Apr 2004), 1516-1524.

[21] Labute, P. *Protonate 3D: Assignment of Macromolecular Protonation State and Geometry*. Chemical Computing Group Inc., City, 2007. http://www.chemcomp.com/journal/proton.htm

[22] Kelly, K. *Protein Analysis in MOE: The Serine Proteases*. Chemical Computing Group Inc, City, 2012. http://www.chemcomp.com/journal/triad.htm

[23] Vacca, J. P., Dorsey, B. D., Schleif, W. A., Levin, R. B., McDaniel, S. L., Darke, P. L., Zugay, J., Quintero, J. C., Blahy, O. M., Roth, E. and et al. L-735,524: an orally bioavailable human immunodeficiency virus type 1 protease inhibitor. *Proc Natl Acad Sci U S A*, 91, 9 (Apr 1994), 4096-4100.

Scaling and Visualization of N-Body Gravitational Dynamics with GalaxSeeHPC

David A. Joiner
Kean University
1000 Morris Ave
Union, NJ
djoiner@kean.edu

James Walters
Kean University
1000 Morris Ave
Union, NJ
walterj1@kean.edu

ABSTRACT
In this paper, we present GalaxSeeHPC, a new cluster-enabled gravitational N-Body program designed for educational use, along with two potential student experiences that illustrate what students might be able to investigate at larger N than available with earlier versions of GalaxSee. GalaxSeeHPC adds additional force calculation algorithms and input options to the previous cluster-enabled version. GalaxSeeHPC lessons have been developed focusing on two key studies, the structure of rotating galaxies and the large scale structure of the universe. At large N, visualizing the results becomes a significant challenge, and tools for visualization are presented. The canonical lesson in the original version of GalaxSee is the rotation and flattening of a cluster with angular momentum. Model discrepancies that are not obvious at the range of N available in previous versions become quite obvious at large N, and changes to the initial mass and velocity distribution can be seen more readily. For the large scale structure models, while basic clearing and clustering can be seen at around N=5,000, N=50,000 allows for a much clearer visualization of the filamentary structure at large scale, and N=500,000 allows for a more detailed geometry of the knots formed as the filaments combine to form superclusters. For the galactic dynamics simulations, we found that while a flattening due to overall angular momentum can be explored with N=1,000 or smaller, formation of spiral structure requires not only a larger number of objects, typically on the order of 10,000, but also modifications to the default initial mass and velocity distributions used in older versions of GalaxSee.

Keywords
N-Body simulations. Gravitational dynamics. Scaling, Visualization.

1. INTRODUCTION

1.1 Motivation
GalaxSee is a gravitational dynamics program initially developed by Mike South and the Shodor Education Foundation, Inc.[5]. The original version was designed for the Macintosh, and focused on allowing users to create small N-Body simulations using a point and click interface, to solve the problem of gravitational dynamics, where the force on any object i due to any other object j is given by:

$$F_{ij} = G\frac{M_i M_j}{\left|\vec{x}_j - \vec{x}_i\right|^3}\left(\vec{x}_j - \vec{x}_i\right)$$

A wide variety of approaches have been developed to solve the gravitational N-body problem[1], including many state of the art computational tools designed for research (see for example [15]), as well as many educational tools. Most research grade tools for N-Body simulation have obstacles to their adoption as a classroom tool—notably a reliance on non-standard compilers, multiple software dependencies, and non-human-readable file formats. Most educational N-Body tools, however, focus on the use of graphical user interface to remove obstacles for students, but replace those obstacles with limitations on the size of N, either hard-coded in the tool itself or self-imposed by the CPU requirements of real-time visualization of results.

Typical classroom use simulations for N-Body problems using tools with limits on the size of N range from 2-Body problems such as the orbit of the Earth around the Sun up to simulations of simple gravitational dynamics, exotic solutions of the few body problem [3], where users could create initial mass distributions with or without angular momentum and explore the disk formation that resulted from a spinning cluster of gravitationally bound masses, or collisions of disk galaxies under the assumption of small objects orbiting two massive cores [9].

1.2 GalaxSeeHPC Learning Goals
The two scenarios presented in this paper focus on studies of structure, the first of the formation and stability of spiral structure and the second of elements in large scale structure. Both of these are meant to be viewed qualitatively, as there are many physical elements left out of the model. In the case of the spiral structure scenario, the galaxy model presented does not account for drag due to the interstellar medium. The large scale structure scenario assumes Newtonian gravity in a constantly expanding universe. Even with these phenomena left out, however, key concepts in gravitational dynamics can be quickly and easily seen by students.

As the tool has been created for general purpose, specific learning goals would be largely implementation specific and would depend on the goals the instructor wanted to emphasize. An instructor

focusing on performance or algorithms might have different goals than an instructor focusing on a science lesson. Like many of the tools developed at Shodor, GalaxSee has always followed the paradigm that it should be able to address both (computational science) education and computational (science education).

In the case of spiral structure, students might learn that the formation of spiral arms is a natural occurrence given a velocity profile that is gravitationally stable, and that not all velocity profiles will be gravitationally stable. Students can, in the process of exploring spiral structure get practice creating velocity curves for model galaxies which could then be compared to those of real galaxies, which might then prompt a discussion of dark matter or other issues of interest.

In the case of large scale structure, students can explore the interplay between expansion velocity and initial mass density for an expanding cube with periodic boundary conditions and "wrapped" gravity. While this leaves out some key features of the Lambda-CDM model, it will allow students to see a trend towards initial clumping along filaments, provided sufficiently high mass density and sufficiently low expansion velocity. The stability of those filaments over time can be seen to be strongly affected, with a tendency towards a "big crunch" for more dense and more slowly expanding systems.

Both of these cases lead naturally to goal-seeking exercises ("How can I change the velocity profile of this galaxy? What if this universe has more mass in a given expanding cube?") that focus on simple conceptual questions related to the balance of gravity, angular momentum, and expansion.

In terms of the computational science learning enabled by these lessons, students get practice using tools running at a command line, input file creation, management, and analysis, parallel job submission and monitoring. The data sets created are rich, with significant challenges in the visualization of results. The simulation includes a variety of force calculation methods, which, while not necessarily state-of-the-art, provide an entry level into two of the key methods used in modern N-Body, tree-based and particle-mesh methods.

1.3 GalaxSee revision history

The original GalaxSee, like many educational N-Body tools, took the approach of a graphical user interface with the ability to pre-create systems at random with a small number of parameters. Later versions of the code included GalaxSee 2.0 for Windows, which kept the look and feel of the original, but added the ability to use a Barnes-Hut force calculation, and a Java based web-start version. GalaxSee-MPI was written to explore parallel computing, removing the GUI interface, as well as the Barnes-Hut force calculation, and allowing for MPI based parallelization of a direct force calculation[8]. GalaxSee-MPI was originally intended just as an exploration of parallelism, and lacked any features to control the input to the simulation, nor did it have any advanced features for visualization, limiting itself to a non-interactive top-down-side-view image of the simulation.

1.4 GalaxSeeHPC Software Goals

The purpose in writing GalaxSeeHPC was to provide students with an N-Body code that (a) allowed students to explore the types of problems that cannot be solved at smaller values of N, (b) allowed students to see examples of some of the force calculation algorithms that have allowed for the increased use of N-Body algorithms, (c) was written in code that is designed for readability and modification, (d) had a simplified dependency stack so that some functionality would be available even without any additional code and that other features could be enabled easily as software dependencies were met, and (e) allowed for human-readable input and output files—so that students would not have to simultaneously learn modern hierarchical file structures at the same time as learning either the physics or algorithms of the N-Body problem.

GalaxSeeHPC is a re-write of the GalaxSee-MPI C++ code Lessons are available from the Blue Waters Petascale Education website[11] and source code is available from Sourceforge[6]. GalaxSeeHPC was written in C to allow for greater portability, and includes both the ability to perform a Barnes-Hut style force calculation algorithm as well as a Particle-Particle Particle-Mesh (PPPM) algorithm. While still command line based, GalaxSeeHPC allows for the user to use a text input file to specify model parameters, including changing the scaling and units used for the problem, allowing a linear expansion of the spatial units (e.g. for a simulation in an expanding universe), force calculation method and parameters, softening factors, numerical integration options, and output features. X-Window based output is still available, but a more interactive SDL-based visualization is also an option, as are multiple different graphics and text output options. CMake is used for configuration and build management, and the code can be configured at compile-time to ignore any options that require numerical or graphical libraries not present on the system.

GalaxSeeHPC has been used and tested in multiple sessions for Physics faculty at the SC09 and SC10 education programs. The visualization of results from GalaxSeeHPC has been a feature of multiple SC and NCSI workshops on scientific visualization. GalaxSeeHPC has been used in two successive summer camp environments with high-school age students.

2. GALAXSEEHPC ALGORITHMS

As every object can interact with every other object, this potentially leads to $N(N-1)$ forces that need to be calculated, though in practice half of these forces will be redundant as each force pair is equal and opposite. As an $O(N^2)$ problem, as N grows large the computational time requirements of the problem can quickly grow beyond the limitations of a typical classroom PC or laptop.

The three approaches that are used to alleviate this problem are parallelism to spread the work over multiple processes, binary tree based sorting of masses to determine which forces can be approximated by substituting a point mass in place of a large number of distance masses, and spectral techniques that interpolate onto a density grid which can be solved using Fourier techniques.

2.1 Barnes-Hut

The Barnes-Hut algorithm is a tree based approach to approximating the force field due to distant particles[2]. An oct-tree is constructed for the space modeled, with the tree recursively refined until each sub-element contains only one object. As the force is calculated, nearby objects, which typically will be close by on the oct-tree and can be located quickly, are used in a direct force calculation, and as objects are further away, branches of the tree can be approximated as a point mass, averaging the masses and positions of many masses into a single force calculation.

Tree methods work on the principle that one can organize an n-body model in a data structure that ensures that nearest neighbors can be easily defined for any one body, and that distant neighbors can be easily approximated using a center of mass treatment. In one dimension, this can be thought of as a binary tree, which can be extended to three dimensions using an oct-tree structure.

A simple implementation of a tree-based structure might assume that physical proximity is equivalent to being leaves on the same branch, but problems can occur for particles at the edge of a high level branch boundary, that are physically close to each other, but separated by many branching on the oct-tree. A modification of the tree algorithm to take into account issues like this might check to see if a node being tested is close enough to the object of interest to be suspect. As one descends the tree, this "closeness radius" can get smaller and smaller. If we consider s_l to be the scale of a tree segment at depth l, one might attempt the following force calculation method

1. For a given object, start at the top of the tree
2. Descend tree
    a. If child node is not a predecessor (along the same branch) of the object being calculated AND the object in question is at a distance greater than ks_l from the center of mass of the node, stop and use the total mass and center of mass of that node
    b. If child node is a predecessor (along the same branch) of the object being calculated OR the object in question is at a distance less than ks_l from the center of mass of the node, but does not SOLELY contain the object being calculated, descend all children of node

The accuracy of the method can be controlled by the closeness criterion k. Figure 1 gives a visualization of this in 1-dimension using a binary tree structure. Note that in the case $k = 0$ this reduces to the previous algorithm, and in the case $k \to \infty$ this approaches a direct force calculation. The total number of forces to be calculated will scale as $N \log(N)$ in this situation instead of N^2 for large models, and the accuracy of the tree calculation (and associated trade-off in speed) can be adjusted by use of the closeness criteria.

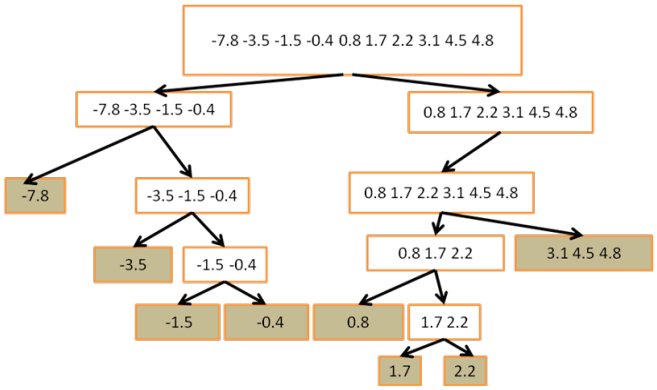


Figure 1: Use of a closeness checking factor can eliminate errors due to aggressive tree pruning

2.2 Particle-Particle Particle-Mesh

Spectral methods, typically solved using the FFT algorithm, reduce the discrete n-body problem to a continuous gravitational problem solved on periodic boundary conditions[4]. Computationally, the advantage of spectral techniques is that it allows you to separate the long-range forces from the short-range forces, and use a direct calculation of short range forces while replacing long range forces with the solution of a potential function that satisfies Poisson's equation.

$$\nabla^2 \Phi = 4\pi G\rho$$

If a function for the density of space can be approximated, this can be solved easily as the Laplacian of the Fourier transform of a function is given by

$$\nabla^2 \hat{\Phi} = -4\pi^2 |\mathbf{k}|^2 \hat{\Phi}$$

where $\hat{\Phi}$ is the Fourier transform of Φ. This gives for the solution of Poisson's equation

$$\hat{\Phi} = \frac{-G}{\pi |\mathbf{k}|^2} \hat{\rho}$$

which can be solved using a discrete Fourier transform, typically the Fast Fourier Transform (FFT) algorithm.

For the implementation in GalaxSeeHPC, the point mass distribution is first interpolated onto a density grid at evenly spaced intervals in x, y, and z. Each mass is treated as if it's mass is spread out over a Gaussian with standard deviation $\sigma = k_\sigma n / S$ where n is the number of grid points in each dimension (assumed to be equal in all dimensions in GalaxSeeHPC), S is the scale of a periodic box in the model, and k_σ is a user supplied parameter.

The Particle-Particle correction is applied to all points within some distance $\sigma_{near} = k_{near} n / S$, where k_{near} is a user supplied constant. Default values of $k_\sigma = 2.0$ and $k_{near} = 1.0$ are used in the code. For the purposes of the periodic boundary conditions in the PPPM algorithm, particles are "ghosted" across a periodic boundary if it results in a particle being closer to a second for the purposes of force calculation.

2.3 Parallelism

2.3.1 Direct Force Calculation

The wall-time when using a direct force calculation is dominated by the nested loop over all particles. This is parallelized in GalaxSeeHPC using MPI, and a round robin scheduling scheme to determine which particle's forces are calculated by which process.

2.3.2 Tree-Based Force Calculation

The tree creation takes sufficiently little time compared to the force calculation that we parallelize only the calculation of the forces from the built tree. The tree is typically built every timestep, but this can be reduced by the user. The loop over all particles to calculate forces from the tree is scheduled using MPI in a round-robin fashion.

2.3.3 PPPM Method

The creation of the density grid and the interpolation of forces from the density grid both consume a significant portion of the force calculation in the PPPM method. Each of these processes are parallelized in MPI using a round robin scheduled loop.

2.4 Softened Potentials

An issue occurs due to the $1/r$ potential in the gravitational N-Body problem in that there is a singularity in the force as particles get very close to each other. Typically, one uses some method of altering the potential to remove any singularities. This can be done by one of two methods in GalaxSeeHPC. The first is through use of a shield radius, as is done in previous versions of GalaxSee, in which the user specifies a parameter which defines a cutoff radius, within which forces are ignored. In practice GalaxSeeHPC uses an adaptive algorithm that depends on the central mass causing the force and the timestep being used, and the actual shield radius is given by

$$r_s = k_{sr}\sqrt[3]{GM\Delta t^2}$$

where the shield radius scaling factor k_{sr} is taken to be 5 by default.

Traditionally, most codes in the literature use what is referred to as a softened potential, in which the potential (and hence force) functions can be modified to include a softened distance, effectively treating all distances as if they were some small distance ε greater than they actually are.

$$P = -\frac{GM}{\left(R^2+\varepsilon^2\right)^{\frac{1}{2}}}$$

with accelerations

$$\frac{\vec{F}_i}{M_i} = \sum_{j\neq i} GM_j \frac{\vec{x}_j - \vec{x}_i}{\left|\vec{x}_j - \vec{x}_i\right|^2 + \varepsilon^2}$$

and potential energy

$$PE = -\sum_{i=1}^{N}\sum_{j=1\neq i}^{i-1} GM_iM_j \frac{\left|\vec{x}_j - \vec{x}_i\right|^2}{\left(\left|\vec{x}_j - \vec{x}_i\right|^2 + \varepsilon^2\right)^{3/2}}$$

3. SCENARIOS

One question that has arisen in many presentations of GalaxSee-MPI to faculty, particularly Physics faculty interested in the science that could be learned by such a simulation rather than computer science faculty interested in scaling properties, has been whether or not students working on projects involving N-body simulations need to run models with enough points to warrant high performance computing resources. A large class of astrophysical problems traditionally fit into what are often described as "million-body" problems—problems that require enough points for study that statistical or hydrodynamical approaches are not appropriate, but for which using too few points in an N-body solution will result in approximation error such that results are qualitatively incorrect[7]. Two problems are presented here that fit into this category, the modeling of galactic structure and the modeling of large scale structure in the universe.

3.1 Galactic Structure

3.1.1 Potential Learning Goals, Science

Students performing this exploration might, depending on implementation, focus on the velocity profiles required to maintain a gravitationally stable structure and the patterns that develop, as well as how the patterns that develop depend on the initial anisotropy of the mass distribution.

3.1.2 Potential Learning Goals, Skills

As N is increased, the computational overhead of a direct force calculation rapidly will increase the computational requirements of each run. The use of a tree-based method would be appropriate in this case as a periodic solution is not needed and the problem domain will have large regions of physical space in which there are few stars. Students can explore performance of tree-based methods as compared to direct force calculations. The parallelization method currently implemented does not truly split bodies across processors but merely shares the results of force calculation at each step. Students can explore the effect of communication on scaling as the code moves from a computation bound problem to a communication bound problem when increasing the number of processes.

3.1.3 Overview

Galaxies are large collections of stars, gas, and dust surrounded by relatively empty space, typically on the order of many kiloparsecs in size and containing hundreds of billions of stars. A key feature of galactic structure is the shape as classified on a tuning-fork diagram, categorizing galaxies as elliptical, spiral, or barred spiral[12]. (Teachers and students can find public domain images of many of these objects online, organized by galaxy type[13].) A feature of the original GalaxSee code was the exploration of how the interplay between gravity and angular momentum tended to flatten a large rotating mass of gravitationally bound objects. However, running models larger than a few thousand points was impractical, both due to hard coded features in early version of the code and the lack of an ability to operate in a command line mode with saved snapshots for models that required longer to run. Additionally, while it was possible to create models with different mass distributions and rotation curves, the default initial mass distributions and rotation curves in GalaxSee did not produce results that could be easily compared to images of spiral galaxies.

As GalaxSeeHPC makes for a more practical approach to running models with larger N, simulations were run to test the results at N=5,000, 50,000, and 500,000. Additionally, models were run with the default initial distribution and velocity profile in GalaxSee, with a mass distribution that is more heavily weighted to the center of the initial distribution, and with a velocity profile that is lowered for object near the center of the mass distribution.

3.1.4 Initial Conditions

The original windows GalaxSee used as its initial conditions a random uniform distribution within a sphere, and a velocity distribution associated with a circular orbit with centripetal acceleration equal to the central force being provided by gravity.

As the number of particles is increased, certain issues related to the default GalaxSee initial conditions are seen. In particular, a uniform distribution does not have enough mass in the core to keep the entire structure cohesively bound, and the distribution breaks up into many small clusters in orbit around each other. Additionally, the assumption of velocity set to centripetal acceleration works well at the edges of the galaxy, but towards the center this overestimates the actual orbital speeds, and simulations see a clearing effect wherein a ring structure is formed as opposed to something that looks like an elliptical, spiral, or lenticular galaxy.

As a result, our initial conditions are taken to be normal distributions in x, y, and z for position, parameterized by the standard deviations of the normal distributions σ_x, σ_y, and σ_z. Velocities are calculated by modifying the assumption of

centripetal acceleration caused by gravitational force to allow for a slower velocity towards the center.

$$\vec{a}_i = \sum_j -GM_j \frac{\vec{r}_{ij}}{\left|r_{ij}\right|^3}$$

$$\vec{a}_{Ti} = \left\lfloor a_{xi}, a_{yi}, 0 \right\rfloor$$

$$\vec{v}_{Ti} = \frac{1}{2}\left(1 + erf\left(\frac{\rho_i}{\mathrm{P}} - 1\right)\right)\sqrt{a_{Ti}\rho_i}$$

where $\rho_i = \left\lfloor x_i, y_i, 0 \right\rfloor$ assuming the entire mass distribution is centered at the origin, and P is the point at which the slower velocities towards the core switch over to a more typical centripetal acceleration-based velocity towards the edges. For each of the models here, we have assumed $\mathrm{P} = \sigma_x / 5$.

3.1.5 *Results of Galactic Structure Simulations*

A simulation was run with an initial distribution with $\sigma_x = 383pc$, $\sigma_y = 0.8\sigma_x$, and $\sigma_z = 0.1\sigma_x$, at sizes of 1,000, 5,000, 50,000, and 500,000 points (see Figure 2**Error! Reference source not found.**). At 1,000 points, typical of the problem sizes one would use with the Windows version of GalaxSee, the possibility of a spiral structure is hinted at by the results, but cannot be clearly seen with so few points. Increasing the size to 5,000 points makes the spiral structure more visible, and 50,000 points allows for a clear structure of spiral arms with clusters along the arms. Models were run for 1 billion years at a timestep of 500,000 years, using an Adams-Bashforth-Moulton integration scheme and a Barnes-Hut force calculation scheme.

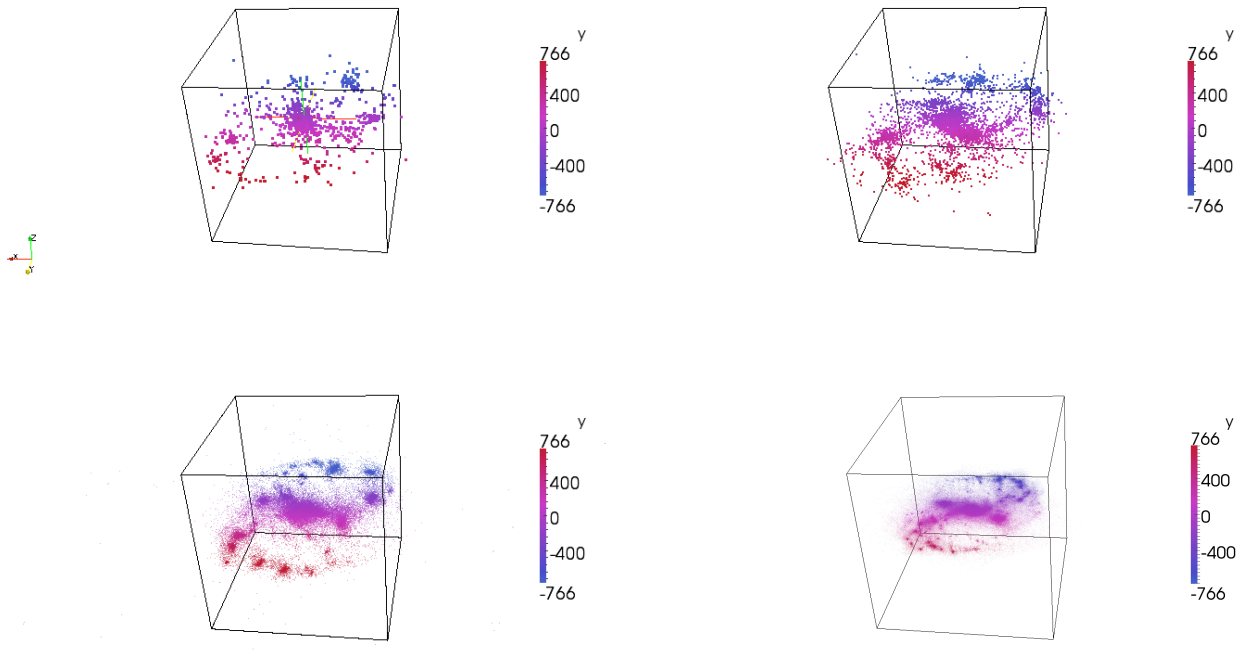


Figure 2 Spiral Galaxy Model with varying values of N. From top left to bottom right N= 1000, 5000, 50000, and 500000.

As can be seen in the comparison of the N=1,000 point and N=5,000 point simulation, 5,000 points was the bare minimum to begin seeing clearly any spiral structure that formed in these models, and on the order of 10,000 points is preferred. A 5,000 point model for GalaxSeeHPC with 8 processes ran in 3 minutes 54 seconds, and a 50,000 point model with 16 processes ran in 43 minutes. For practical use in a classroom lab, 5,000 point models are best run on a multi-core workstation or small cluster, and 50,000 point models are best done as either a single model run at the beginning of a class and analyzed afterwards or overnight or as part of longer term student projects. Models with 500,000 points showed more detail, but did not have qualitatively different features for this problem than those with N=50,000, while requiring significantly longer to run.

3.1.6 *Scenarios for students to investigate*

One key issue in the formation of classic spiral and barred spiral structures is the need for some difference in the scale in the x and y directions for the initial conditions. The lower the eccentricity of the initial material, the less likely it is that the resulting galaxy will have a classic two-armed spiral structure.

A second issue for students to study is the distribution of mass in the forming galaxy—looking at the difference between normally distributed matter and uniformly distributed matter, without an elevated density towards the center of the galaxy there will not be enough gravity to hold the center together, and students will see systems that fragment into many smaller rotating clusters. Additionally, it is possible to overestimate the acceleration of

objects towards the center if one simply sets centripetal force equal to the gravitational force exerted on each object. Both of these are shown in Figure 3.

Testing

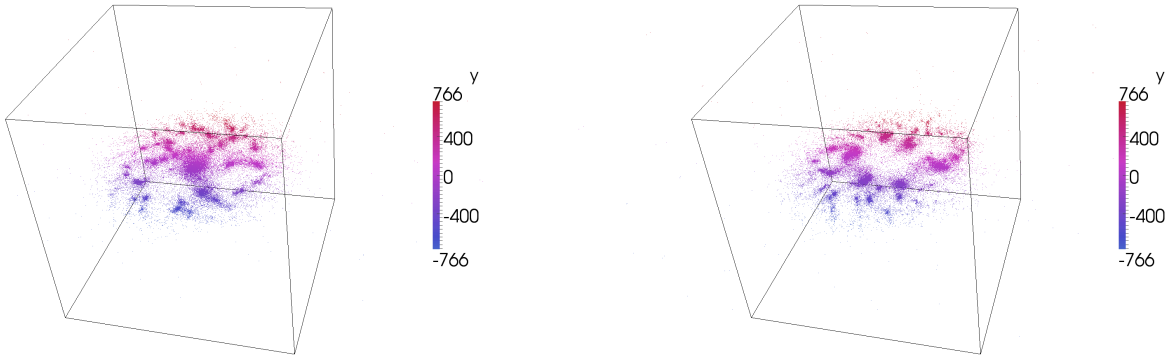


Figure 3 Simulation of galaxy formation without any eccentricity to induce spiral arm formation (left) and without a higher density in the central region to form a core (left)

This can be seen by example with a though experiment in which two equal stars orbit each other. Since it is not a case of a single object orbiting a more massive one, the actual velocities required to maintain a stable orbit are half what it would be otherwise.

This is addressed in the initial velocity function used in this paper by using an error function to create an interior zone where the objects are treated as if they are orbiting each other, and an exterior zone in which objects are orbiting a central mass. Having too little mass in the center can lead to fragmentation of the galaxy being modeled, and having too high of a speed for the interior objects can lead to clearing of the inner regions—and thus fragmentation of the galaxy being modeled.

3.2 Large Scale Structure of the Universe

Issues of cosmology on a large scale are both of interest to many students and are well reported in current media and research literature. Recent advances in computational simulations have led to understandings of the structure of the universe and the connection to the $\Lambda - CDM$ model of big-bang cosmology[4]. One of the largest N-body simulations ever run—the Millennium Simulation—focuses on this problem[14].

Modeling of large scale structure is complex—the distance scales change as the universe expands, and results depend sensitively on both the initial anisotropy of the mass distribution as well as the density. Computationally the problem requires treating the space modeled as a unit cell with periodic boundary conditions. However, students can explore at some level conceptual ideas with a simple Newtonian model. Our approach in GalaxSee is to let the students explore self-gravitation of a random anisotropic initial mass distribution in an expanding periodic box.

Initial student exploration into large scale structure can include an overview of the existing data on large scale structure, and attempts to fit models of big-bang expansion, gravitational condensation of galaxies, and freezing out of structures as the universe expands to that data, particularly with regards to the eventual end fate of our universe. While recent studies suggest that there is sufficient inflation to sustain the universe and have it continue its expansion, until recently it was unknown by scientists whether the universe's gravitational pull would ever result in an eventual "big crunch" collapse. This provides a compelling question for students to investigate, and allows them to understand the process by which computational science has informed us about this phenomenon. Even without allowing for either an expanding universe or any inflation to that expansion, students can, with only Newtonian gravity, explore the creation of filamentary structure and the eventual progression to a collapse event without expansion to prevent it. (As of version 1.1, GalaxSeeHPC supports the ability to model an expanding universe with a constant expansion rate, but does not allow for inflation—though this is a modification that a student could make.)

The models used in studying cosmological structure are often referred to as "universe-in-a-box" models, in that they take what might be considered a unit-cell of the universe, and approximate the gravitational effect of the surrounding universe by assuming that things are isotropic enough that whatever is happening on the left side of the cell is just as likely as anything else as to be a representation of what might be happening beyond the right edge of the unit cell. As such, periodic boundary conditions are applied, in effect giving us a toroidal geometry in order to approximate a piece of a larger universe. Students can change the initial mass density and size of this universe in a box, start with a random initial distribution, and simulate the initial clustering and eventual collapse that occurs. Students can see an interim stage before collapse where the types of structures formed closely resemble both the more accurate cosmological models being run on research codes.

3.2.1 Initial Conditions

The initial mass density and unit cell size were chosen so to ensure that the simulation would results in visible creation of filamentary structure, and the image shown are taken at the peak of the filamentary nature of the structure before further collapse

occurred. The models shown here were run with no expansion and 1.0e14 solar masses randomly distributed in a 1 megaparsec cubed box. (Note that these numbers are chosen simply to produce qualitative results and are not meant to be physical. While these initial conditions can qualitatively show filamentary structure it results in a mass density of the universe that is orders of magnitude greater than observed and not stable for the lifetime of the universe.)

3.2.2 Results of Universe in a Box simulations

Simple effects can be seen with a fairly modest value of N. Consider the following simulation result, using GalaxSeeHPC with the PPPM algorithm and N=5,000. Figure 4 shows the results of a model with N=5,000 using the PPPM algorithm required roughly 1 second per timestep running in serial on a Xeon-based machine, with parallel performance peaking at only a few processes, though larger values of N were able to scale to more processes. With a typical model requiring on the order of a thousand timesteps, this is well within the range of what a student might do in a lab setting, running a simulation every 10-20 minutes on typical hardware

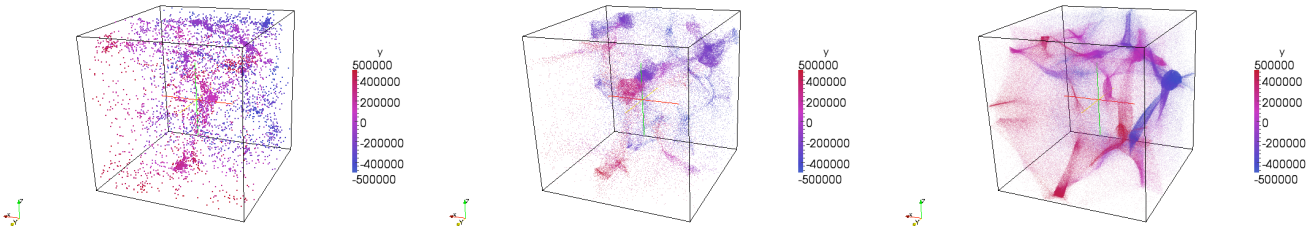


Figure 4 Large scale structure simulation, N=5000, 50000, 500000

Looking at the same model for greater values of N, students will be able to see more detail. At N=50,000, the knots in the middle of the filaments become more readily apparent and additional structure in the filaments can be seen The connectedness of the filaments is much clearer. The typical CPU time for models of this size in our tests was on the order of 2 days. Scaling up to 8 processes for this problem size on our test cluster was reasonably efficient; making this a simulation that students could run multiple times in one day on a quad-core or 8-core system.

When looking at the simulation results with N=500,000 the structure of the filaments themselves becomes much more clear, as does the morphology of the knots where filaments intersect. Scaling of this problem to 16 processes was reasonably efficient, and while models with millions of objects might run in days to weeks, depending on the number of timesteps required, students with access to a 8-core system or small cluster could run models in less than a day to a few days.

3.2.3 Scenarios for Students to Investigate

Two key questions students can try to address with these models are the sensitivity to the initial mass density of the universe of large scale structure and the effect of the expansion of the universe on large scale structure.

An initial study students may make is to look at the timescales needed for gravitational collapse of a large area of the universe with the current mass density and without any expansion. Starting with a random initial configuration, students should see that there is an initial clustering into a filamentary structure and that these filaments feed into superclusters which then themselves combine, but that the timescale for this happening is so short compared to the age of the universe that some degree of expansion is required to understand the structure of our current universe. The mass density in **Error! Reference source not found.-Error! Reference source not found.** shown in the previous section, for example, require a mass 4 orders of magnitude greater than observed, and would result in gravitational collapse within a few billion years.

GalaxSeeHPC has an EXPANSION variable in the input file which allows for a constant expansion rate. The timescales in which major change occurs will vary greatly as the universe expands, so for practical purposes it is useful to also have a scaled timestep that gets larger as the model progresses, and for that reason the student has an option of setting the timestep as a ratio of the current time using the TIMESTEP_RATIO variable rather than as a fixed number. Tracking the initial formation of anisotropy back to the point when gravitation began will likely require timesteps and numbers of objects that go beyond the architecture students have available, however the students can still start with a largely anisotropic random distribution of points at some later time, such as $1/100^{th}$ the age of the universe, and evolve forward with mass densities near the current mass density of the universe. By changing the initial mass, they can see that the difference between structure never forming, filamentary structure of the type seen today, or a "big crunch" is only a few orders of magnitude, and that the qualitative types of structures found in more detailed models can be seen as naturally resulting from a combination of self-gravitation, mass density, and expansion.

Care must be taken in interpretation of results. While it is possible to independently set expansion rate and mass density in the GalaxSeeHPC input file, in practice it would be expected that these two parameters are related.

3.3 Scaling of the N-Body problem

An important concern with the N-Body problem is the scaling of the problem, both in terms of how the computing requirements scale with algorithm and problem size as well as how well the parallel implementation of the problem scales across a parallel architecture.

The first type of scaling is often referred to in terms of the "Big O" of the problem—if one were to write a function of the number of total computations needed as a function of the problem size, what term in that will dominate as the problem size gets large. In this sense, a direct force calculation is order N^2, and tree and PPPM methods are both order $N\log(N)$.

Parallel scaling, on the other hand, is typically referred to as either weak or strong. Parallel implementations with weak scaling allow for larger problems to be solved in roughly equal time on larger (i.e. more CPU cores) systems. Parallel implementations with strong scaling allow for same sized problems to be solved in less time on larger systems.

GalaxSeeHPC allows students to explore the big O scaling of direct, tree-based, and PPPM methods, and to begin exploring questions related to parallel scaling. It should be noted that the parallel implementation used in GalaxSeeHPC is limited in its parallel scaling, particularly for moderate and large clusters.

Students and teachers interested in pursuing questions related to state-of-the-art tools that exhibit strong scaling on larger systems are encouraged to look at the many professional-grade N-Body solvers. Of particular note is Gadget-2, which compiles with standard C compilers on many systems and has a fairly small number of dependencies required to run. GalaxSeeHPC includes an option to translate its own input files into Gadget-2 format.

3.3.1 Timing and Scaling of Galactic Structure Simulations

Running a simulation with 1,000 points and a Barnes-Hut calculation as described in the previous section, GalaxSee for Windows required roughly 17 minutes on an EEE PC with 1 1.7GHz Atom chip running Windows XP. Similar speeds with GalaxSee for Windows were seen on a HP EliteBook with a 2.5 GHz Centrino running Windows Vista. GalaxSeeHPC running on a single process on a Dell PowerEdge 1850 with a 2.7GHz Xeon running RedHat Linux finished in 1.7 minutes, on 4 processes finished in 31 seconds. For comparison on similar hardware, GalaxSee-MPI (which was largely based on the Windows GalaxSee codebase) using Barnes-Hut and a 4th order Runge Kutta took 4 minutes 18 seconds (GalaxSee-MPI does not currently support ABM integration methods). GalaxSeeHPC using Runge Kutta 4 took 3 minutes 24 seconds.

Many of these models could be run with a larger timestep, bringing running times on all platforms down (typical class presentations for the rotation and flattening of a spherical cluster are done with timesteps of 8 million years as opposed to 0.5 million years), however even for larger timesteps running models with on the order of 1,000 points is the practical classroom application limit of GalaxSee Windows.

Wall times per timestep for serial jobs are shown for N=5,000, 50,000, and 500,000 in **Error! Reference source not found.**. The tree-based implementation in GalaxSeeHPC scales closer to N log N than the N squared scaling expected of a direct force calculation. The parallel scaling of GalaxSeeHPC with the tree-based force calculation method was consistent across problem sizes, scaling to speedups of on the order of 10-15 on our cluster. Efficiency typically peaked once a few 8 core nodes were involved in the solution of the problem. For each of the problem sized tested, parallel efficiency dropped to 50% at about 16 processes. All are shown in Figure 5.

3.3.2 Timing and Scaling of Large Scale Structure Calculations

Like tree-based methods, the PPPM method in GalaxSeeHPC scales as roughly N log(N). The compute time required for the force calculation is dominated by the mapping of points to a grid and the interpolation of forces on that grid back onto the points, combined with nearest neighbor direct force calculations.

Speedup peaked at around 8 for models with N ranging from 5,000 to 500,000 on the cluster used in this study, with parallel efficiency dropping off somewhat faster for the PPPM methods compared to tree-based methods. Results are shown in Figure 5.

4. VISUALIZATION

4.1 Need for higher end hardware and software at large N

In addition to the computational challenges of increasing N in GalaxSee by many orders of magnitude, the resulting data also poses challenges in how it can be visualized, as traditional method of filling in a pixel if there is a mass in the line of sight for that pixel quick saturates at large N, even for very high resolution images. Even in relatively low-density regions of the simulation, foreground objects can obscure more important details. Masking the image by only showing a subset of points can result in loss of detail for structures of interest. This can impact both the type of hardware and software that is needed for students to work with large datasets. While modest computers with embedded video may be able to load and render larger datasets, such hardware can experience much longer frame rates when loading data for a new time or when attempting to re-render data for a different perspective (such as by rotating a rendered dataset in ParaView.) Figure 6 shows the effect of not allowing for any opacity when drawing a large number of point masses, as well as the loss of resolution and structure that can occur from masking points.

Many visualization packages exist that are available to students that allow for advanced features such as changing the opacity of points, volume rendering, and creating contours and slices of regular gridded data. ParaView[10] and VisIt[16] are two such examples that are available as open source, and will work with a variety of input data types included methods of opening simple comma separated files.

The images created for this paper were made using ParaView. ParaView is multi-platform, and has been designed to work in a distributed fashion for massive data sets. Developed by Kitware Inc. and Los Alamos National Laboratory, ParaView is also supported by Sandia National Laboratory and the Army Research Laboratory.

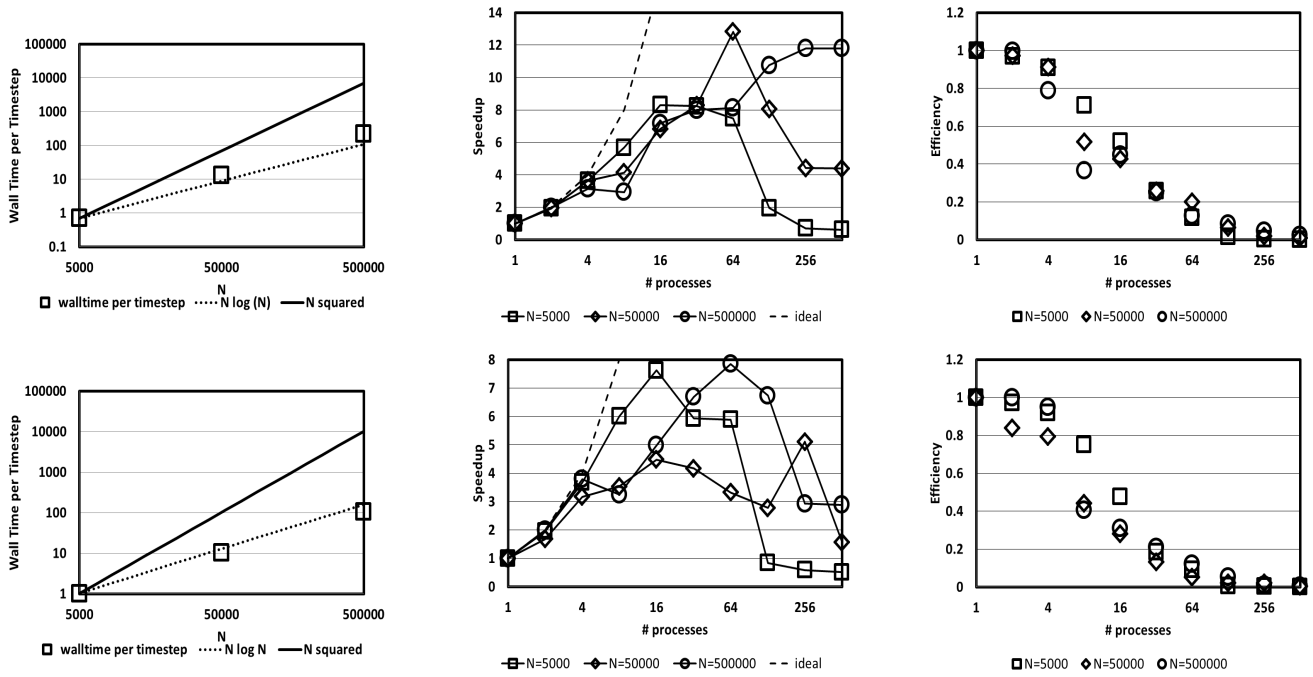


Figure 5 Scaling properties of example problems. Top row shows serial performance of tree-based algorithm run in serial relative to direct force calculation, followed left to right by speedup and efficiency (ideal would be 1.0) of tree algorithm in parallel. Bottom row shows serial performance of PPPM relative to problem size followed by speedup and performance.

4.2 Use of CAVE for visualization

Additionally, a CAVE system was used with students to visualize the results of GalaxSeeHPC, using a simple package written in OpenGL with CAVELib. Rendering was limited to no lighting effects and pixels for each mass, and up to N=500,000 could be viewed with zero masking and a frame rate high enough for the user to walk through the image without noticeable lag. The CAVE system used was a three wall system with ART head tracking and a dedicated render node using separate NVidia Quadro cards for each wall.

Our initial use of the CAVE has focused on the feasibility of using it for education. Technically, we wanted to know whether there were easy methods of getting student data into the CAVE and whether it would provide an obstacle that interrupted class flow.

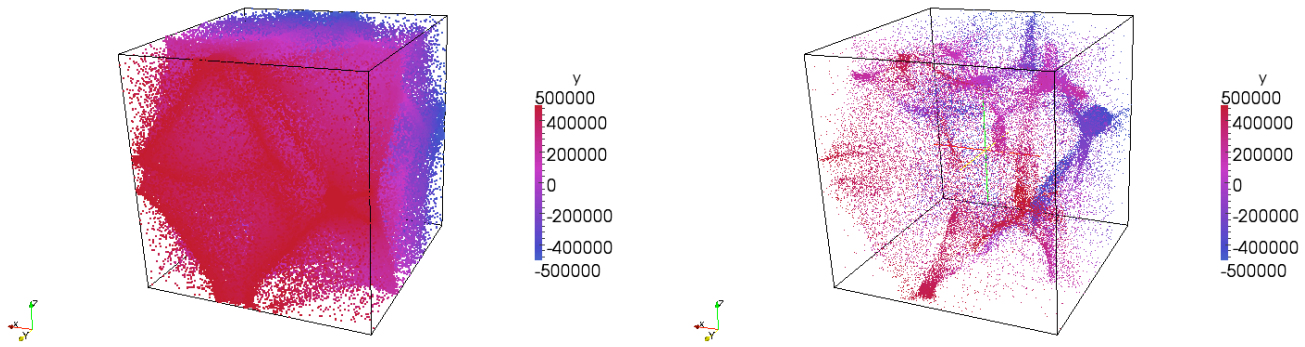


Figure 6 N=500,000. Shown at left is without any masking or opacity. At right masking is used, but no opacity is enabled to enhance visualization.

5. PEDAGOGICAL CONCERNS

5.1 Sample Lesson Plan

GalaxSeeHPC is meant to be a general purpose pedagogical tool around which a variety of lessons might be built, focusing on both topics in computational science education as well as topics in physics and astronomy.

The following lesson plan is designed based on past use of GalaxSeeHPC with high school students. It assumes the use of a helper code "GalaxSeeUI," available on the Sourceforge site for

GalaxSeeHPC, to generate input files for the investigation of spiral galaxy shapes.

Subject: Physics

Grade: 11-12

Lesson Length: 90 minutes (2 classes of 45 minutes)

Title: Galaxy Structure Simulations Using Computer Applications

Overview:

Galaxies are large collections of stars rotating around a central point in space, while moving about in the universe. These bodies of stars tend to crash and collide with each other, and take on new and varying forms. Through Hubble, galaxies have gained classifications based on their structures as they form over time. This lesson will have the students learn about how to classify galaxies by their structure, using Hubble's model and computer simulations of their own design.

Preparations and Materials:

- The teacher should become familiar with the GalaxSeeUI application, GalaxSeeHPC application, and the ParaView application. (GalaxSeeUI application is available on Sourceforge site along with GalaxSeeHPC and can be used to generate input files for this lesson.)
- The teacher will need an internet browser in order to access this site

  http://cosmictimes.gsfc.nasa.gov/teachers/guide/1929/guide/classifying_nebulae.html
- The teacher should have a text editor, such as notepad, loaded along with the applications, and some way of displaying all of the applications on the monitor to the class.
- The students will need access to computers to utilize text editing software, in order to generate their initial conditions.

Objective:

- Students will be able to distinguish the different galactic structures, using the tuning fork model and computer simulations.
- Students will be able to apply their previous computer knowledge to generate input files for GalaxSeeUI and utilize ParaView.
- Students will be able to compose an argument about their own observations and defend their point of view.
- Students will be able to infer things about natural phenomenon based off of the activities conducted during this lesson.

Standards:

- NSES.9-12.A: Science as inquiry.
  - Use technology and mathematics to improve investigations and communications.
  - Communicate and defend a scientific argument.
- NSES.9-12.D:
  - Origin and evolution of the universe.
- NSES.9-12.E:
  - Understanding about science and technology.
- DoDD.Science.9:
  - Use of computational models.
  - Use careful systematic observation and data collection to obtain valid information.
  - Relate force, motion, energy, and power.

Procedure and Activities:

Day 1 – 45 minutes

1. The teacher will define the term galaxy.
   a. Galaxy - a system of stars, numbering in the millions to billions that, along with gas and dust, are held together by gravitational attraction.
   b. An example that can be given is the Andromeda galaxy, the closest spiral galaxy to the Milky Way galaxy.
2. The teacher will define the types of galaxies:
   a. Elliptical - a galaxy, generally having an elliptical shape and no obvious inner structure or spiral arms
   b. Spiral - a galaxy, that exhibits a central nucleus from which many curved arms extend
   c. Bar Spiral – a galaxy, that contains a central bar structure from which two large arms extend
   d. Irregular – a galaxy, that cannot be labeled by the previous definitions
3. The teacher will utilize the Hubble Classification, or the Tuning Fork Diagram, to discuss the development of galactic structure over time. A version of this diagram can be found on:

   http://skyserver.sdss.org/dr1/en/proj/advanced/galaxies/tuningfork.asp
4. The teacher will show students a general input file of GalaxSeeUI in Notepad, and will explain to the students the proper way to input and save the data. All files that are being submitted to GalaxSeeUI are .in files and can be saved with this extension when saving and asked to name file (Example: test.in).
5. The students will be paired into small, 3-4 person, groups to work on their own input files.
6. The students will utilize the computers to create an input file, using Notepad, following the teacher's example on how to setup the text file and save it with the proper extension.
7. The teacher will tell the students to finish what they are doing, and to return to their groups. The teacher will, then, have the students choose one member of their group to submit their file to GalaxSeeUI, and save the file to a folder, the teacher should have access to this folder. Advise that this folder should be a shared folder that the entire class can access, but the teacher can control.
8. The teacher will show a variety of galaxy pictures to the students, and ask the students to make a classification of the galaxy's structure, as well as provide their reasoning for coming to that conclusion.

Example images can be found on: http://hubblesite.org/gallery/album/galaxy/

9. The teacher will ask if there are any final questions or comments, and conclude the lesson. This time can also be used to aid the students with any errors that may arise.

Day 2 – 45 minutes

10. The teacher will show the students how to start ParaView, and how to configure ParaView to read in their data files. The teacher will, then, show the students how to play their animation, and how to download the images needed to examine the structure of the galaxies.
11. The teacher will have the students retrieve their data from, a flash drive the teacher controls or, the folder used previously. The folder should have individual folders with the group of students' names, and inside the folders should be the input file the students created, and the output from GalaxSeeHPC.
12. The students will observe their galaxies, analysis the results they note, and make an educated conclusion on the structure of their galaxy.
13. The teacher will instruct the students to use ParaView to take a picture of their "initial" step and their "final" step.
14. The student groups will share their results with the class, having the students present a small summary of their results and making their final classification of their galaxy.
15. During the ending to the period, make references to stable and unstable initial conditions. Care must be taken to differentiate between a student's set of initial conditions and actual data. Possible wording would be to always refer to the students simulations as models and never as "a galaxy."
    a. Stable – initial conditions such that the model does not exhibit overall change in structure or makeup as the simulation evolves. A stable simulation that additionally exhibits behavior similar to data is one in which the initial conditions are likely to correspond with real galaxies.
    b. Unstable – simulation exhibits behavior that changes greatly during evolution, particularly changes in the size, rotational speed, and overall geometric makeup. This may be due to numerical instability (have students try reducing timestep), or it may be due to initial conditions that are not physically likely.
16. The teacher will ask if there are any final questions or comments, and conclude the lesson. This time can also be used to aid the students with any errors that may arise anywhere during the lesson.

Extensions:

17. Show the students how to plot the velocity of their galaxies in ParaView as star color. Show them how the velocity curve of their galaxies plays a role in how stable their structure is.

5.2 Choice of Time-step

Currently none of the versions of GalaxSee (GUI-based, the original command-line MPI, or the latest GalaxSeeHPC release) allow for an adaptive timestep in solution. While this change is planned in the future, this makes it especially important that an appropriate time-step is used. Even with professional grade codes using higher order and/or adaptive integration schemes, great care must be taken with choice of time-step.

If students are not familiar with time-stepping methods, they should get some information on the drawbacks of a time-step that is either too large or too short. Any of the versions used with some form of visualization (GUI-based have built in visualization, GalaxSee-MPI and GalaxSeeHPC have the option of compiling X-based visualization into the program if supported by your platform) will show this clearly, with demonstrably wrong results and instabilities occurring with too large a time-step, and with visibly slower computation occurring with too small a time-step.

5.3 Choice of Integration Method

Integration methods available in GalaxSeeHPC mirror some of the more standard options used, as well as some options that are pedagogically easy to introduce yet not stable enough for professional work. The Euler method is included for pedagogical purpose as it is often the first numerical integration method students learn, and the easiest to code. The so-called "improved Euler" or second order Runge-Kutta scheme as well as the mid-point Euler method and leapfrog methods are also allowed in the code as these are often introduced in numerical analysis classes as incremental improvements to Euler's method. In practice, however, one would not want to run professional integration with these schemes. The fourth order Runge-Kutta algorithm is generally considered the simplest numerical integration scheme one would want to use for professional work, and is a standard method used across computational science disciplines. Additionally, predictor-corrector schemes, such as the Adams-Bashforth-Moulton available in GalaxSeeHPC attempt to use previous timesteps to better predict future behavior. For anything other than investigating the numerical impact of using lower order integration schemes, students should use either the fourth order Runge-Kutta or Adams-Bashforth-Moulton integrators.

5.4 Limitations of GalaxSee-MPI

The primary limitations of GalaxSee-MPI from a classroom perspective was the inability to use it to teach any concept beyond which it was originally intended. GalaxSee-MPI as first written was designed to show scaling of the parallelization of direct force calculation using MPI, however all of the features of previous versions of GalaxSee that made it a useful tool for classroom exploration had been removed—the ability to easily modify input for new scenarios, the ability to design input files to meet your own problem, the ease of visualization had been removed in making a command line version of the program. Moving the program to a command line version in a HPC environment, however, did allow for much larger values of N—which the visualization abilities of early versions of GalaxSee would not handle well anyway.

Additionally, over many years of using GalaxSee-MPI in faculty workshops with Physics faculty, many faculty expressed skepticism as to whether there would be benefit for their students to running N-Body simulations with a larger value of N—whether

there was anything the students would learn at large N that they would not learn at small N. Also, the lack of a feature to allow for periodic boundary conditions limited the types of situations that could be modeled.

From a technical perspective, the use of GalaxSee-MPI in new environments was often hampered by the choice of C++ as a language. While C++ is largely standard and widely adopted as a language, the C++ version of GalaxSee-MPI suffered from portability issues as it was deployed on different clustering platforms. The dependency on specific standard libraries often caused software to fail to run as expected, and different mpicxx executables, from one MPI implementation to another, often required minor code changes to in order to deploy the software on a new platform.

5.5 Changes Made

The following feature comparison shows changes made in GalaxSeeHPC compared to previous GUI based and command line based versions.

Feature	GUI versions	GalaxSee MPI	GalaxSee HPC
Runs from input file	✓		✓
Users can specify individual particle properties	✓		✓
Problem scale	Choose from menu list		User specified
Change integration method (Euler, Improved Euler, RK4, ABM)	✓		✓
Barnes-Hut	✓		✓
PPPM			✓
Passive visualization	✓	(with X11)	(with X11)
Interactive visualization	✓		(with SDL)
Command Line option		✓	✓
MPI		✓	✓
Write to snapshot files			✓
Additional output options			✓
Softened potential	Adaptive shield radius	Adaptive shield radius	Adaptive shield radius or fixed softened potential

5.6 Effect of Modifications

One concern in moving to GalaxSeeHPC was whether the removal of the GUI component would make exploration of science questions significantly more difficult for students using GalaxSee. In previous workshops with students, typical use was to use the Windows, Mac, or Java version of GalaxSee when exploring science questions and to use the GalaxSee-MPI version of the code when exploring problems with parallel efficiency and scaling. Our first use of GalaxSeeHPC in a informal education setting in summer 2010 did show that the constant flow back and forth between windowed versus command line environments slowed the pace of activities down, and when given the choice students tended to stick with the GUI driven tools. In summer 2011, we focused more specifically on using the command line tools, with more instruction on the use of the command line interface and activities that included visualization of results solved with larger N in a CAVE environment, which seemed to make for a more natural use of the command line driven HPC tools.

Since the move to C, we have seen significantly reduced issues with portability. The new version of the code has been tested on multiple platforms with both GNU and Intel compilers.

5.7 Visualization tools

Any effort to bring scalable supercomputing applications into the classroom will need problems of significant size to (a) require supercomputing resources, and (b) scale on those resources. This provides an additional concern for the educator in that large problems produce large sets of results, and visualization of those results will need to be part of the plan for implementing the use of such tools in the classroom. The use of common data formats is encouraged in order to be able to make the best possible use of open source visualization tools. Comma Separated Value text files provide a low barrier for creation of files, and are readable by many visualization tools, but will typically require the configuration of many options within the tool to define how the CSV file should be interpreted. Other Common Data Formats, such as NetCDF or HDF, are well supported by the open source community, and are standard input file formats for most visualization tools, however this will provide an additional challenge for implementation as code libraries for those formats may have to be installed on the systems on which students are computing their results.

5.8 Storage limitations

Another concern for problems involving large N, particularly in a classroom situation in which many students will be running multiple sets of such models, is disk storage. For our N=500,000 models, 30Mbytes per snapshot was typical, stored in NetCDF format. Keeping enough snapshots to create a smooth animation for N=500,000 typically required 3Gbytes per simulation. Storage requirements were linear with N.

6. FUTURE WORK

6.1 CAVE Visualization

Our initial work in incorporating the CAVE into the visualization of GalaxSeeHPC has focused primarily on technical issues of how to get the data into the CAVE as well as the feasibility of incorporating a CAVE system into the flow of a class. While our general finding is that stereo immersive visualization, as it is inherently focused on one individuals point of view, is difficult to use in a large class setting it can be inspirational for students. We noticed a clear “wow factor” when bringing participants into the CAVE. It is easier to incorporate immersive visualization into individual student projects, as there is less of an issue with contention for the resource.

Our initial work with students has used custom written software, and we are investigating whether we can replace this by using VisIt, for which a Conduit interface exists, or ParaView, which has been ported to other CAVE systems using FreeVR.

We have not yet investigated whether participants learn differently in an immersive environment from a non-immersive

environment, or from viewing 3-D data in other, non-immersive, stereo visualization systems.

While CAVE systems are unlikely for typical classroom use, students may consider using non-immersive stereo rendering in ParaView through more readily available 3D monitors, TVs, or projectors.

7. ACKNOWLEDGEMENTS

The GalaxSeeHPC revisions were funded as a module project under the Blue Waters Petascale Education program. The cluster used for this project was funded by NSF award OCI-722790. The CAVE used in this project was funded by NSF award OCI-0959504.

8. REFERENCES

[1] Aarseth, S. 2003. *Gravitational N-Body Simulations*. Cambridge Monographs on Computational Physics.

[2] Barnes, J.E. and Hut, P. 1989. Error analysis of a tree code. *Astrophysical Journal Supplement*. 70, (Jun. 1989), 389–417.

[3] Christian, W. 2010. EJS CSM Textbook Chapter 5: Few-Body Problems. *An Introduction to Computer Simulation Methods - Draft EJS edition*.

[4] Efstathiou, G. and Eastwood, J.W. 1981. On the clustering of particles in an expanding universe. *Monthly Notices of the Royal Astronomical Society*. 194, (Feb. 1981), 503–525.

[5] GalaxSee Curriculum Resources: *http://www.shodor.org/master/galaxsee/*.

[6] GalaxSeeHPC: *http://sourceforge.net/projects/galaxseehpc/*.

[7] Heggie, D. and Hut, P. 2003. *The Gravitational Million-Body Problem*. Cambridge University Press.

[8] Joiner, D.A. et al. 2008. Supercomputer based laboratories and the evolution of the personal computer based laboratory. *American Journal of Physics*. 76, 4 (2008), 379.

[9] Mihos, C. et al. 1999. GalCrash: N-body Simulations on the Student Desktop. *American Astronomical Society Meeting Abstracts* (Dec. 1999), #101.04.

[10] ParaView - Open Source Scientific Visualization: *http://www.paraview.org/*. Accessed: 2012-04-25.

[11] Petascale: GalaxSeeHPC: 2011. *http://www.shodor.org/petascale/materials/UPModules/NBody/*.

[12] Rood, H.J. and Sastry, G.N. 1971. ”Tuning Fork” Classification of Rich Clusters of Galaxies. *Publications of the Astronomical Society of the Pacific*. 83, (Jun. 1971), 313.

[13] SEDS Messier Database: *http://messier.seds.org/*.

[14] Springel, V. et al. 2005. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature*. 435, 7042 (Jun. 2005), 629–636.

[15] Springel, V. 2005. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*. 364, (2005), 1105–1134.

[16] VisIt Visualization Tool: *https://wci.llnl.gov/codes/visit/*. Accessed: 2012-04-25.

Introducing Evolutionary Computing in Regression Analysis

Olcay Akman
Department of Mathematics
Illinois State University
Normal, IL 61790 USA
oakman@ilstu.edu

ABSTRACT

A typical upper-level undergraduate or first-year graduate level regression course syllabus treats "model selection" with various stepwise regression methods. In this paper, we implement the method of *evolutionary computing* for "subset model selection" in order to accomplish two goals: i) to introduce students to the powerful optimization method of genetic algorithms, and ii) to transform a regression analysis course into a regression and modeling course without requiring any additional time or software commitment. Furthermore, we employ the Akaike Information Criterion (AIC) as a measure of model fitness instead of the commonly used measure of R-square. The model selection tool uses Microsoft Excel, which makes the procedure accessible to a very wide spectrum of interdisciplinary students, with no specialized software requirement. An Excel macro, to be used as an instructional tool, is freely available through the author's website.

Keywords

Genetic Algorithm, Model Selection, AIC

1. INTRODUCTION

Predictive model selection can be a difficult procedure for data sets with a large number of explanatory variables. Determining what variables best explain the system by exhaustive search becomes unreasonable as the number of variables increases. For example, over one billion possible models exist for data with 30 explanatory variables.

One area in which model selection is important is multiple linear regression. In ecological studies one of the commonly used methods for selection is step-wise regression, with forward or backward variable selection algorithms. These methods have been criticized for lacking the ability to truly pick the best model for several reasons [2, 12]. One problem is that the choice in which the variables enter the selection algorithm is not justified theoretically. In addition, the probabilities for the selection procedure are chosen arbitrarily, which may lead to a poorly selected model. One of the leading advocates of implementing cutting-edge methods in predictive model selection [11] provides a solid foundation for evolutionary computing. Finally, since these methods employ local search, it is unlikely that the global maximum set of variables will be found [9, 6, 7, 10].

Because of the drawbacks of the current model selection procedures, we propose to use a genetic algorithm to optimally determine the subset of variables for a multiple regression model. Genetic algorithms are a wise choice for this procedure. They are a global search tool and are not prone to the problems associated with stepwise selection being a local search. Genetic algorithms operate by considering many models at the same time; through selection, components of the best models come together to form the maximal model. We will now go through the basics of genetic algorithms. This is a brief explanation; a thorough one can be found in [5].

2. GENETIC ALGORITHMS

Genetic algorithms (GAs) are a set of optimization techniques inspired by biological evolution operating under natural selection. First developed by Holland [8], they have grown in popularity because of the ability of the algorithm to perform well on many different types of problems. In a genetic algorithm, possible solutions are coded using binary strings, which are called chromosomes. Each chromosome has a fitness value associated with it based on how well the string is optimizing the problem. During each generation, the time step of the algorithm, a population of chromosomes compete to have their "genes" passed on to the next generation. The selection step is used to pick the chromosomes for the next generation based on their fitness. Those selected enter the mating pool, where two chromosomes mate using crossover. During this phase, parts of each parent string are swapped to form two new chromosomes that have certain aspects of their parents. After crossover, mutation is applied. Mutation occurs with a small probability, and is defined as a change from 0 to 1 or 1 to 0 at a certain location in the binary string. Mutation allows the introduction of new "genes" that were either lost from the population or were not there to start with. Through successive generations, increasingly better chromosomes come to dominate the population and the optimal (or close enough) solution is realized.

3. AKAIKE INFORMATION CRITERION

A key parameter of a GA is a method to evaluate the fitness of a chromosome. In order to use a GA for model selection in multiple regression, a way to evaluate the chromosomes is needed. In other words, a method is needed to determine how well the subset explains the system.

Akaike introduced AIC in 1973 [1] as a measure of the complexity of a model. It measures the bias due to the estimation of the model from the true distribution based on the data. Additionally, AIC takes into account the number of parameters used to create the model. The formula for AIC is given as

$$AIC(k) = -2\log L(\hat{\theta_k}) + 2m(k), \quad (1)$$

where $L(\hat{\theta_k})$ denotes the maximum likelihood function, $\hat{\theta_k}$ is the maximum likelihood estimate of parameter vector θ_k under the model m_k, and $m(k)$ is the number of parameters in the model. The first term of AIC gives the lack of fit of the model, and the second term is a penalty for the number of parameters in the model. The model with the lowest AIC value is considered the best, because the model best determines the underlying stochastic process with the least number of parameters.

4. A GENETIC ALGORITHM FOR MULTIPLE LINEAR REGRESSION MODEL SELECTION

4.1 Background

The first step to implementing a genetic algorithm for any optimization problem is to determine a way to encode the problem into a binary string. In the case of multiple linear regression, we have q data points with n explanatory variables and one response variable. We wish to fit the data to

$$y = X\beta + \epsilon, \quad (2)$$

where y is an $n \times 1$ response vector, X is an $n \times q$ matrix of the data points, β is a $q \times 1$ coefficient matrix, and ϵ is an $n \times 1$ error vector with entries from independent normal distributions ($N(0, \sigma^2)$ for all components). The dataset contains n explanatory variables. The encoding is done by creating a binary string with $n + 1$ bits, where each bit represents a different parameter of the model. The additional parameter in the binary string is the intercept for the linear model. A parameter is included in the model if the value of the bit for that parameter is a 1 and is excluded if it is a 0. A quick example will help explain this procedure.

Suppose that we have a dataset where we are interested in what variables explain the reproductive fitness of a species of trees. The possible explanatory variables will include

1. Age of tree
2. Height of tree
3. Soil pH
4. Density of trees in the surrounding area
5. Average temperature of environment
6. Average rainfall of environment
7. Circumference of trunk
8. Longitude of environment
9. Latitude of environment
10. Prevalence of disease in environment.

In order to use a genetic algorithm in choosing the best model, each binary string will have 11 bits.The first bit is for the intercept and the following 10 correspond to the possible explanatory variables. For example, the string 10010111101 would code for a model which includes the intercept, soil pH, average temperature of environment, average rainfall of environment, circumference of trunk, longitude of environment, and prevalence of disease in environment. To further demonstrate this point, the string 00001000110 is a model that has no intercept, and includes density of trees in the surrounding area, longitude of environment, and latitude of environment (see Table 1).

Chromosome	Variables Included
10010111101	Intercept,3,5,6,7,8,10
00001000110	4,8,9

Table 1: Chromosomes and variables included by the model it represents

Once we have a method of encoding, we need a way to evaluate the binary strings in order to choose the best model. Although several choices for evaluation are available, this paper focuses on AIC. Since the model with the lowest AIC value is considered the best, the genetic algorithm chooses strings biased towards those with the lowest value.

The probability that a string will be chosen for the mating pool is proportional to its transformed fitness. For example, if one string has a value of k for its fitness and a second has $5k$ for its fitness, the second string has 5 times a better chance of being in the mating pool. Additionally, note that the string with the worst AIC value will never be picked for the mating pool, as its fitness will be 0.

Now that we have a method of encoding and a way to evaluate the strings, we will determine some parameters of the genetic algorithm. The first one we consider is the method of creating the initial population and determining its size. Unless previous knowledge about the problem is available, it is commonplace in genetic algorithms to randomly generate binary strings [5]. However, in the case of model selection, a user may want to force a parameter(s) to be included, even if it is not part of the model with the lowest complexity. In this case, the initial population can be generated in such a way that certain parameters are always in the model. In addition to how the population is initially generated, the user must choose the size of the initial population. This process can be difficult. Generally the size should not be too large because it will slow down the algorithm, and should not be so small that genetic drift takes over the course of evolution of the population. In typical genetic algorithms, the size of the population stays the same; however, this may not be an effective use of computation. We will see in the next section

that starting with a larger size then reducing it may be more effective.

Finally, we discuss the genetic operators which allow the algorithm to find the optimal model. There are two operators that are generally implemented in genetic algorithms: crossover and mutation. Crossover mimics biological crossover in a simplified manner. First, the probability of crossover (p_c) is chosen. When in the mating pool, a pair of strings are chosen along with a random number from $[0, 1]$. If that number is less than the probability of crossover, crossover occurs. Thus, if $p_c = 1$, then every pair will cross, and if $p_c = 0$, then the strings will not be altered by crossover. After the choice of p_c, the number of crossover points must be chosen. The location of the crossover points is chosen at random. Then the bits from the parent strings are swapped to create two new offspring strings (see Figure 1). The purpose of crossover is to bring together models which have components that reduce complexity. In the previous example about trees, we had two parent strings where Parent 1 coded for a linear model with the intercept, soil pH, average temperature of environment, average rainfall of environment, circumference of trunk, longitude of environment, and prevalence of disease in environment. Parent 2 coded for density of trees in the surrounding area, longitude of environment and latitude of environment. Applying crossover of the two parents created two offspring (see Figure 1), where Offspring 1 coded for a model with an intercept, soil pH, average temperature of environment, longitude of environment, latitude of environment, and prevalence of disease in environment. Offspring 2 is a model that includes density of trees in the surrounding area, average rainfall of environment, circumference of trunk, and longitude of environment. Through successive generations and application of crossover of low complexity models, the algorithm is able to find the least (or close enough) complex model to explain the data.

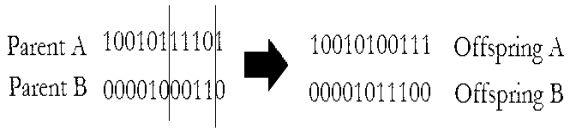


Fig. 1: Diagram of crossover with 2 points

Crossover can only generate models that include parameters that already exists in the population. What if the least complex model includes a parameter that is not present in the population? That is, the position in the string that codes for that parameter is fixed at 0. Mutation alleviates this problem. Mutation in genetic algorithms is similar to mutation that occurs in DNA. First, the probability of mutation (p_m) has to be determined. This value gives the probability that at each location in the string the bit will be flipped. Flipping is defined as the change of a 0 to 1 or a 1 to a 0. Typically, mutation rates are low, on the order of 10^{-3} to 10^{-5}, however, strings are usually longer for other applications of genetic algorithms than they are for determining least complex models.

We conclude this section with pseudo code for a genetic algorithm used to find the least complex model that sufficiently describes the data.

1. Generate initial population
2. While (t<max generations OR the maximum number of computations have not been executed)
   (a) Calculate AIC for the model each string encodes
   (b) Select strings for the mating pool
   (c) Create a new population using crossover
   (d) Mutate new population
   (e) t=t+1
3. End

5. CLASSROOM IMPLEMENTATION

Since the goal of this approach is multifaceted, the classroom implementation has three basic components as a part of a typical regression analysis syllabus.

The first component is a one lecture-hour introduction to genetic algorithm concepts that is best initiated just before the time where a typical syllabus introduces model building via stepwise regression. At this point students are familiar with multiple regression, goodness of fit measures, and their use in model comparison. This component does not need to cover detailed descriptions of crossover options, optimal mutation rates, population sizes, required number of generations, or stopping rule conditions, but rather should be designed to expose the students to the basic notions of evolutionary computing. In fact this is where, for instance, an idea of "numbers mating to create new numbers" fascinates most students, making them eager to learn what comes next.

The second component is also an hour-long demonstration of evolutionary computing. It is made of an exercise where students can witness how an initial guess evolves to be the solution of a problem by following the performances of populations within each generation. One of the best examples of this classroom exercise is obtaining the solution for a Diophantine equation using MS Excel. For this step, using MS Excel, as opposed to an R-code written exclusively to perform GA optimization, a canned GA program, or an online applet, would be highly recommended since this approach allows the students to store each population on a sheet and enables them to compare the improvement in fitness from generation to generation. This step doesn't have to be very sophisticated. In fact performing the GA only for a few generations (a few Excel worksheets) within a lecture will be enough to convey the underlying message of evolutionary computing. In particular, the recommended exercise is as follows:

1) We are looking for positive integers a, b, c, and d such that $a + 2b + 3c + 4d = 30$. This is the famous Diophantine equation, which has now become a benchmark tool in teaching evolutionary computing or GA based optimization in undergraduate level courses due to its simple but yet non-analytic solution structure. It is a commonly used example

for teaching GA. Another use of it using C++ code can be seen in [3].

2) Generate 4 uniform integers from $(0, 30)$ and save them to columns A2 through D2. Enter the formula $= ABS(A2+2*B2+3*C2+4*D2-30)$ to cell E2 (Use A1 to E1 for labels). Notice that since we are seeking a solution in terms of these four positive integers, none of them can be greater than 30. At this step the user needs to know how to hold the random numbers static, else each time Enter is hit the random numbers will change. This can easily be achieved by setting the Calculation option to Manual in Excel 2010 (Click Formulas ribbon, click Calculation Options,select Manual to disable auto-calculation). In older versions use Click Tools > Options > Calculation tab.

3) Repeat step 2, say 50 times (using the cells A51 to E51), which creates the "1st generation" of 50 "chromosomes" each of which has 4 "genes" (a through d) with "fitness" given in column E. In this case, the smaller the value in column E, the fitter the corresponding chromosome is. Hence, entering the formula $1/fitness$ into column F will be useful during the rest of the process.

4) Normalize fitness of each chromosome saved in column F (by dividing each fitness by the total fitness) to create the probability of selecting a chromosome proportional to its fitness. As a suggestion, the instructor might use conditional formatting in the fitness probability column to help students to visualize the relative magnitudes of the selection probabilities,. In Excel 2010, this can be achieved by using Data Bars-Gradient Fill option.

5) Copy and boldface the "best" solution for these 50 individuals in row 52.

6) Randomly select one pair with probability given in column F and copy them into into the first two rows of column G. These are the two parents of the first offspring. Specifically select two rows using the probabilities in column F.

7) Generate a uniform integer from (1,3) to determine the crossover point. Swap the tails of the two parents to create two children. Copy them to column H.

8) Repeat this process for 25 pairs, forming 50 new children.

9) Randomly choose 5 children (rows). For each selected chromosome generate a uniform integer from (1,4). This will indicate which gene to "mutate". At this point, we explain that 10% of the individuals are subject to mutation for some "divine" reason.

10) Replace the selected genes with newly generated integers from (0,30).

11) Copy these 50 rows onto a new worksheet with their respective fitnesses.

12) The fitness of the best individual in row 52 of the 2nd worksheet can now be compared with that of the first worksheet.

13) Repeat this process for 3 more generations so that a total 5 generations of 50 individuals can be used to to discuss the principles evolutionary computing.

After these steps, the students are now ready to work with GAs for subset model selection, which is the third component of this approach. So far, with the use of the Diophantine equation we demonstrated how chromosomes (variables of the equation) evolve improving the fitness generation after generation. We now make the connection to regression modeling, where the fitness is the quality of the model, say R-square or AIC (another commonly used more robust goodness of fit measure for students of this level). We will see how R-square improves as models evolve. Since the idea of subset model selection requires comparing regression models containing different predictors we will treat models with different set of predictors as different chromosomes. This is exactly what we did for the Diophantine equation where different values of the variables were defined different chromosomes. As an alternative to stepwise regression, we generate a binary string of length equal to the number of possible predictors, and include only. the variables that correspond to 1's. Repeating this process generates a population of regression models each one has its own fitness. We then proceed with the natural selection process as described in Section 4. For this step, it is recommended that the first few steps are demonstrated, again using MS Excel. However, for model selection discussions using real-life data sets, it is recommended that the freely available Excel macro on the author's website[1] should be used to focus on the modeling discussions without allocating any more class time to the process itself. The first few steps can be implemented as follows:

1) Save the response variable to column A, and explanatory variables into columns starting with B.

2) Generate a binary string of length equal to the number of explanatory variables.

3) Form a regression model by including only the explanatory variables that correspond to 1 in the binary string.

4) Run MS Excel regression utility for the model.

5) Repeat from step 2 for another model, pointing out the different models due to the different set of explanatory variables being used.

6) Use the MS Excel macro for a desired number of generations, discussing the model sensitivity aspects of different runs.

The steps given above were implemented as a part of a section where subset model selection was covered in an upper level undergraduate/lower level graduate Regression Analysis class. Since no regression analysis book has a chapter allocated to this approach, the supplementary notes, Excel files were distributed. Although the lectures were also supplemented by Power Point presentations, this is optional and not an essential part of the approach described here.

[1] www.ilstu.edu/∼oakman

6. CONCLUSIONS

Treating predictive model selection via GA in a regression analysis course serves two very useful purposes. First, it introduces students to the notion of evolutionary computing by blending its basic concepts within the very familiar framework of of regression methods. This requires no background beyond some introductory statistics knowledge. Second, it arms students, especially those with diverse interests such as biology, sociology, economics and so on, with a very powerful and cutting-edge method of model building. Additionally, the genetic algorithm approach combined with the use of AIC is better at handling data in which collinearity exist than the traditional selection methods such as forward, backward, and stepwise selection. Although no formal study of student performance was conducted, every student, even the ones who perform less than perfect seem to relate to the material much better than they do to the traditional approaches. Course evaluations consistently indicate this chapter as of the their favorite chapters. In fact several independent Study projects, two M.S. theses were produced on the topic by the students who approach the instructor after this chapter was covered.

7. REFERENCES

[1] Akaike, H. (1973). Information theory and an extension of the maximum likelihood principle. In B.N. Petrov and F. Csaki (Eds.), *Second international symposium on information theory*, Academiai Kiado, Budapest, 267-281.

[2] Boyce, D. E., Farhi, A., and Weischedel, R. (1974). *Optimal Subset Selection: Multiple Regression, Interdependence, and Optimal Network Algorithms.* Springer- Verlag, New York.

[3] http://www.generation5.org/content/1999/gaexample.asp?Print=1

[4] Fisher, R.A. (1930) *The Genetical Theory of Natural Selection* Clarendon Press, Oxford.

[5] Goldberg, D. E.(1989) *Genetic algorithms in search, optimization, and machine learning.* Reading, MA: Addison-Wesley.

[6] Hocking, R. R. (1976). The analysis and selection variables in linear regression, *Biometrics*, 32, 1044.

[7] Hocking, R. R. (1983). Developments in linear regression methodology: 1959-1982, *Technometrics*, 25, 219-230.

[8] J. Holland. (1975) *Adaptation in Natural and Artificial Systems.* The MIT Press.

[9] Mantel, N. (1970). Why stepdown procedures in variables selection, *Technometrics*, 12, 591-612.

[10] Moses, L. E. (1986). *Think and Explain with Statistics*, Addison-Wesley, Reading, MA.

[11] J. Whittingham, Philip A. Stephens, Richard B. Bradbury and Robert P.Freckleton Why Do We Still Use Stepwise Modelling in Ecology and Behaviour? Journal of Animal Ecology, Vol. 75, No. 5 (Sep., 2006), pp. 1182-1189

[12] Wilkinson, L. (1989). *SYSTAT: The System for Statistics*, SYSTAT, Evanston, IL.

Teaching Students to Program Using Visual Environments: Impetus for a Faulty Mental Model?

Edward Dillon
Clemson University
School of Computing (HCC Division)
100 McAdams Hall
Clemson, SC 29634
(864) 656-1266
edillon@g.clemson.edu

Monica Anderson-Herzog
The University of Alabama
Department of Computer Science
Box 870290
Tuscaloosa, AL 35487-0290
205-348-1667
anderson@cs.ua.edu

Marcus Brown
The University of Alabama
Department of Computer Science
Box 870290
Tuscaloosa, AL 35487-0290
205-348-5243
mbrown@cs.ua.edu

ABSTRACT
When learning to program, students are typically exposed to either a visual or command line environment. Visual environments are usually adopted to help engage students with programming due to their user-friendly feature capabilities. This article explores the effect of using visual environments such as *Integrated Development Environments* and *syntax-free tools* to teach students how to program.

Prior studies have shown that some visual environments can have a productive impact on a student's ability to learn and become engaged with programming. However, the functional behavior of visual environments may cause a student to develop a faulty mental model for programming. One possible reason is due to the fixed set of skills that a student acquires upon initial exposure to programming while using a visual environment.

Two systematic studies were conducted for exposing students to programming in introductory courses using both visual and command line environments. From the first study, it was found that visual environments can initially impose a lower learning curve for students. However, the second study revealed that visual environments may present a challenge for students to directly transfer their acquired skills to other programming environments after initial exposure.

Categories and Subject Descriptors
H.5.2 [**Information Interfaces and Presentation**]: *Interaction styles*; K.3.2 [**Computers and Education**]: Computer science education

General Terms
Design, Human Factors

Keywords
Visual Environments, Human-Computer Interaction, Education, Learning

1. INTRODUCTION
Programming can be considered a skill for solving problems computationally. However, teaching students how to program has been a challenge. It has been argued that students sometimes fail to develop an accurate mental model for programming [3, 18]. Because of this deficiency, students can encounter programming as a barrier and in many cases leave fields that typically view this skill as a necessity. For example, Computer Science departments generally face the challenge of retaining incoming majors. Beaubouef and Mason detailed many factors that could cause students to leave Computer Science with one being the lack of skills for problem solving [1].

Attention has been placed on ways to improve a student's ability to learn and apply programming skills. One area of focus has been programming environments. Guzdial advocates "the greatest contributions to be made in this field are not in building yet more novice programming environments but figuring out how to study the ones we have" [8]. Kelleher and Pausch noted that programming environments have been built since the 1960s with the purpose of making programming accessible to people of various ages and backgrounds [10]. Visual environments like integrated development environments (IDEs) and syntax-free tools have become more common for teaching programming. There have also been efforts to expose and engage students at earlier learning stages to programming using visual environments [11, 12, 13].

Because of their functional behavior, there is the potential concern whether visual environments cause students to develop a faulty mental model for programming. Visual environments are typically constructed in a way that hides basic programming behaviors (*ex. compilation, debugging,* and *execution)* under a GUI interface. This style of construction can restrict students from direct exposure to essential programming concepts and functionalities. For instance, syntax-free tools like Alice and Scratch can cause a student to learn a limited set of programming skills by restricting exposure to code syntax, program compilation, and file systems. IDEs can provide program compilation and file system scaffolding, but disguises these and related behaviors as GUI options that are embedded into a menu item, widget, or icon. It has been found that students can depend too much on the GUI options that an IDE offers with insufficient understanding of what they are doing [2].

This article explores visual environments and their potential effect on a student's productivity for programming. Section 2 discusses prior studies regarding visual environments and their effect on students. Section 3 expounds upon the construction, feature sets, and operation behavior of visual environments. Section 4 shows

two studies that were conducted to evaluate student behavior when a visual environment is used to teach programming. Section 5 gives the conclusion, threats to validity, and future work for this research.

2. RELATED WORK

Previous studies have shown the impact of using visual environments to teach students how to program. Below is a summary of studies that evaluated visual environments and their effect on students at introductory stages of programming (Figure 1). Measurements that were used to evaluate these environments were either subjective (ex. attitudes, motivation) or objective (ex. retention rates, time on task).

Visual Environment	Course Level	Effect on Novices		Specific Effect
		Positive	Negative/None	
Alice	CS1	X		Performance, Retention Rate, and Attitudes
BlueJ	CS1	X		Attitudes
CS1 Sandbox (with subsets)	CS1	X		Time on Task
Eclipse	CS2		X	Complexity of Usage
LEGO®Mindstorms	CS1		X	Extrinsic Motivation

Figure 1. Prior Evaluations of Visual Environments and their Effect on Novices

Moskal, Lurie, and Cooper [15] measured the effect of Alice, a syntax-free environment, on CS1 students over a period of two years. Their results showed that Alice had a positive impact on performance, retention, and attitudes of the students, especially those who were considered at-risk (students with little to no programming experience prior to CS1 enrollment or a weak mathematical background) [15].

Hagan and Markham [9] studied the effect of BlueJ, a Java IDE, for teaching CS1 students object-oriented programming. They found that initially students were indifferent towards BlueJ, but gradually their attitudes became more positive for using this environment as the semester progressed. The authors believed that the difficulty of installing and learning to use BlueJ might have influenced the students' initial attitude toward this environment [9].

DePasquale [4] evaluated the ease of use of the CS1 Sandbox IDE (with and without language subsets) against Microsoft Visual C++ .Net on CS1 students. He found that students were more efficient with their tasks when using CS1 Sandbox than Microsoft Visual C++ .Net when language subsets were applied. In addition, DePasquale discovered that students who used CS1 Sandbox at the beginning of the study later migrated more readily to using Microsoft Visual C++ .Net [4].

Chen and Marx [2] measured an Eclipse IDE against an IDE called Ready to Program in a CS2 course for a period of two years. During the first semester of this study, the students preferred Eclipse over Ready to Program due to their initial excitement for this environment during an in-class demonstration. However, many of these students chose to use Ready to Program to complete take-home projects. Some of the reasons for not using Eclipse were based on the lack of experience, installment issues, and the difficulty of using this environment in the absence of the instructor [2]. During the following two semesters, the students enrolled were given a CD that provided hands-on experience with using Eclipse. Chen and Marx found that these particular students showed slightly better attitudes toward Eclipse. During the final semester of this study, Chen and Marx expanded the study into CS1 by exposing students in this course to Eclipse. They found that students depended too much on the wizards that Eclipse offered with insufficient understanding of what they were doing. Therefore, no IDE was used for programming during the following semester but rather Notepad and the Command Prompt terminal. The reason for this change was to help the students get a broader understanding of compilation, execution, and editing of programs. The authors also believed that this change would help the students better understand the usefulness of an IDE [2].

McWhorter and O'Connor [14] performed a study on LEGO® Mindstorms to determine if this application could influence motivation (intrinsic or extrinsic) for students learning to program in a CS1 course. They found that students using LEGO® Mindstorms showed a barely significant decrease in their extrinsic motivation from the control group. McWhorter and O'Connor concluded that LEGO® Mindstorms scarcely had any substantial effect on their students' overall motivation for programming [14].

From these studies, there were different conclusions about the effect of visual environments on students while learning to program. Environments like Alice, BlueJ, and CS1 Sandbox were able to influence positive productivity in the students. On the other hand, Eclipse and LEGO® Mindstorms revealed a different outcome. In particular, Chen and Marx found that the appearance of Eclipse excited their students. However, its complexity and implied behavior for programming procedures caused the authors to move later students to a command line environment.

3. THE CONSTRUCT OF VISUAL TOOLS

Visual environments are typically built using a WIMP format (window, icon, menu, and pointing device) for operation. IDEs are composed of a menu bar with a list of menu options and icons, a text editor for writing code, a built-in compiler/interpreter, and a debugger for conducting programming tasks via a mouse. In many cases, these features are integrated into one window for operation (Figure 2). Syntax-free environments like Alice and Scratch are also constructed using the WIMP format with additional features for drag-and-drop coding.

Visual environments are usually constructed differently from command line environments. Command line environments use a text editor to write and edit code but depend upon an external command terminal for code compilation/interpretation, debugging, and execution (Figure 3). In addition, students may be required to learn a variety of command arguments to effectively operate a command terminal. There are cases where certain text editors may provide a WIMP-oriented background to create and edit a program (Figure 4), but still require a command terminal to generate the program's output.

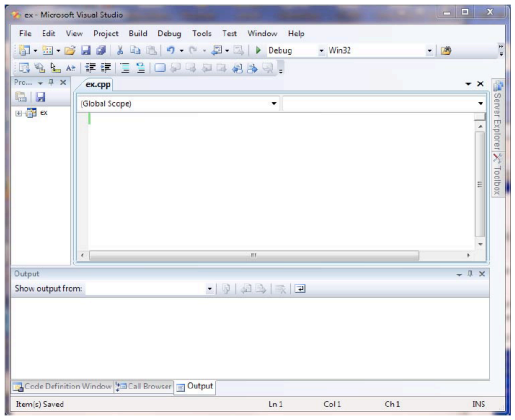


Figure 2: Microsoft Visual Studio IDE 2008

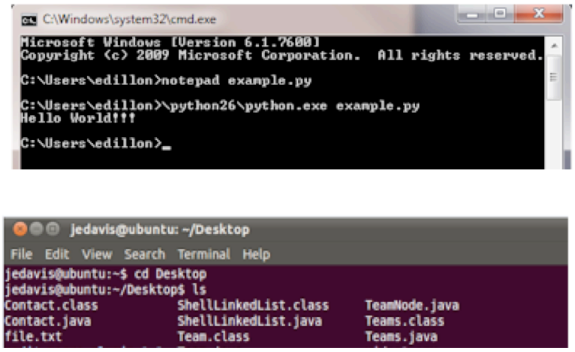


Figure 3: Command Terminals for Windows and Linux Platforms (respectively)

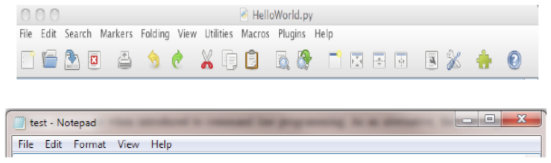


Figure 4: Text Editors for JEDIT and Notepad (respectively)

3.1 Feature Sets

The feature sets within visual environments typically provide a higher level of assistance to students when learning to program [5]. For example, IDEs can provide a large quantity of features that are designed specifically to assist users with programming; these include *syntax highlighting, error highlighting, auto completion, mouse usage,* and *integrated compilation/execution*. Usually, command line environments are not built with these capabilities, which restrict students to use a fixed set of features for operation. The next subsection provides more detail about the different levels of assistance that occur between visual and command line environments.

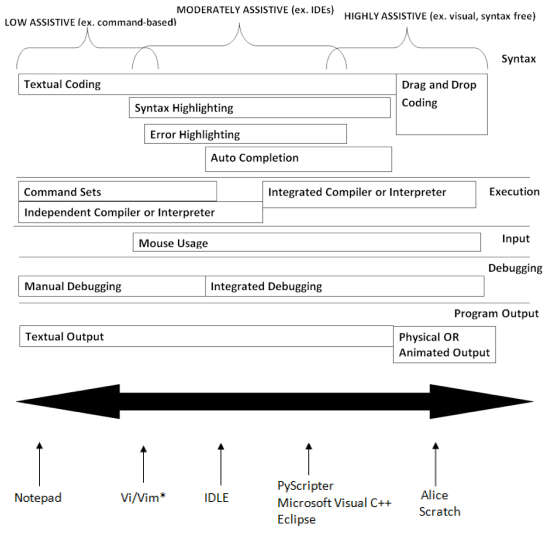


Figure 5: Programming Environments: Feature Sets Continuum [5]

Feature set can readily be altered

3.1.1 Continuum

Figure 5 illustrates a continuum of basic feature sets that can be seen amongst visual and command line environments [5]. Feature sets enable these environments to provide low, moderate, or high assistance to a programmer. The continuum provides clarity for how specific environments are categorized based on their default feature sets. There are cases where individual features can be enabled or disabled within environments (notice the asterisk beside the Vi/Vim editor in Figure 5). This can alter an environment's behavior, which can also cause an environment to shift either left or right on the continuum.

Low assistive environments (left region of the continuum) typically possess basic essential features for programming. Some of these environments may only provide the user with an editing window and a window for compilation/execution or interpretation. These environments typically allow the user to perform textual coding, command usage, and manual debugging. Users depend on some independent compiler or interpreter to run a written program that usually generates a textual output. Example environments that provide low assistance are plain text editors and text editors with very limited features. As listed on the continuum, Vi/Vim is an example text editor that provides limited features, which include syntax highlighting and mouse usage for programming. In addition, environments that represent this region of the continuum tend to be command-line oriented [5].

Moderately assistive environments (middle region of the continuum) can provide a larger quantity of assistive features for programming. Some of these features consist of syntax highlighting, error highlighting, auto completion, mouse usage, integrated compilation/execution (or interpretation), and integrated debugging. Usually, these environments can also provide textual feedback. There are some full-featured environments that possess similar traits seen in low assistive environments. These traits include: command sets, independent window for compiling/executing (or interpreting), and manual debugging. Example environments that represent this region of the continuum are rich-featured editors, intermediate and advanced/commercial IDEs [5].

Highly assistive environments (right region of the continuum) can also possess a larger quantity of assistive features for programming. Usually, these environments are built specifically to teach novices how to program. Therefore, many of these environments can also provide features that restrict the user to foundational programming concepts. Some highly assistive environments also require the user to perform drag and drop programming rather than syntax programming. In addition, physical or animated output can be used as an alternative to textual output. Example environments that represent this region of the continuum are graphical environments like Alice and Scratch, and pedagogical IDEs [5].

For additional details about the feature set continuum, see our paper published in the ***Journal of Computing Sciences in Colleges*** [5].

3.1.2 Familiarity

As part of feature assistance, there are features within programming environments, particularly those that are visual (Figure 6), that can provide a student with a familiar clue or *affordance* of how a particular action can be performed while programming [17]. Some of these features can also be seen in common software applications that provide service to users with

different levels of computational experience, which include *Microsoft Office suites, Internet Safari,* and *ITunes* (Figure 7).

It is likely that students have been exposed to these software applications to *surf the web*, *chat online*, *write an essay or term paper*, or *listen to music* prior to their first programming class. Because of these similar features, there is the potential for a visual environment's behavior to be familiar to students while learning to program. For example, a student could perceive the procedures for using a visual environment to be relative to a word processor. This sense of familiarity could also lessen the learning curve for understanding the operations of a visual environment upon initial exposure.

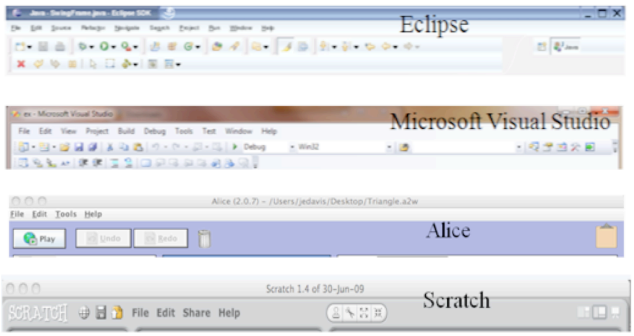


Figure 6: Examples of Visual Environments and their Relative Features [6]

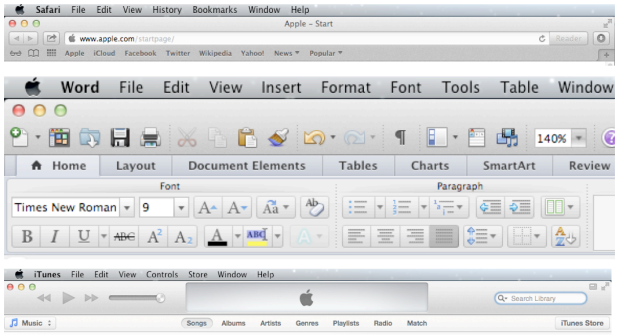


Figure 7: Software Applications and their Relative Features to Visual Environments (*Internet Safari, Microsoft Word, and iTunes*)

3.2 Operation Behavior

While students are learning to program, understanding programming concepts or language syntax is one aspect. Another is becoming accustomed to the procedures for operating a programming environment. When operating a command line environment, students typically cannot bypass one procedure and complete another. This is not the case for many visual environments. Instead, programmers can perform certain procedures automatically with a click of the mouse. The next subsections discuss the operation of command line environments, IDEs, and syntax-free environments respectively along with a brief discussion about their potential effect on students.

3.2.1 Command Line Programming

When conducting command line programming, students are usually directed to an editing window to begin composing (or *writing*) their program. Students must also save their program as a file for the remaining procedures. Next, students should test the correctness of their written program by compiling their saved file. Since a command terminal is typically used for compilation, students are required to use command sets for operation. Based on the command terminal and language being used for programming, there are certain commands that will enable the students to compile their program file. Upon compilation, students are faced with one of two scenarios: 1) If a syntax error(s) is detected during compilation, this error must be corrected before proceeding to the next step. 2) If no errors are detected during compilation, the program file undergoes the process of linking. When linking occurs, the program file is converted into an executable file in preparation for execution. After program linking is completed, the students must type a certain command in the terminal to invoke the execution of their program. Upon execution, the students are faced with one of two more scenarios: 1) If a semantic (or logical) error(s) occurs, this error must be corrected and would require the students to repeat the compilation and linking process again. 2) If no errors are detected during execution, the output of the program would be generated and viewed in the terminal window. Figure 8 provides an outline of the typical operations for command line programing. Table 1 provides a summarized list of these operations in their respective order.

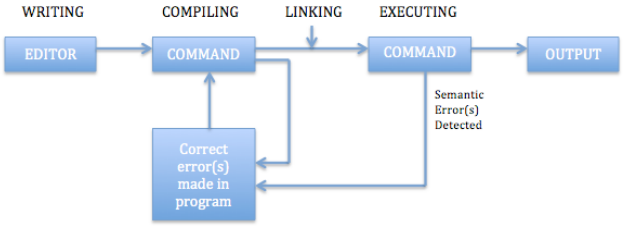


Figure 8: Outline of Command Line Programming

Table 1. Command Line Programming Operations

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is compiled and checked for syntax errors. (Students must use the appropriate command to invoke this behavior)
Step 3*	The file of the program is converted into an executable file for execution.
Step 4*	The executable file of the written program is executed to acquire the intended output. (Students must use the appropriate command to invoke this behavior)
Step 5	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

It is also important to note that certain languages are not compiled, but rather interpreted. The operations for interpreted languages are almost identical to a compiled language with exception to the procedures for compiling and linking the program file. Instead, the program file containing the written code has to be interpreted. There are certain commands that will enable students to interpret the code in their program file. Upon interpretation, the students are faced with one of three scenarios: 1) If a syntax error(s) is detected

during interpretation, this error must be corrected before proceeding to the next step. 2) If a semantic (or logical) error(s) occurs, this error must be corrected before proceeding to the next step. 3) If no errors are detected during interpretation, the output of the program would be generated and viewed in the terminal window. Figure 9 provides an outline of the typical operations for command line programming with interpreted languages. Table 2 provides a summarized list of these operations in their respective order.

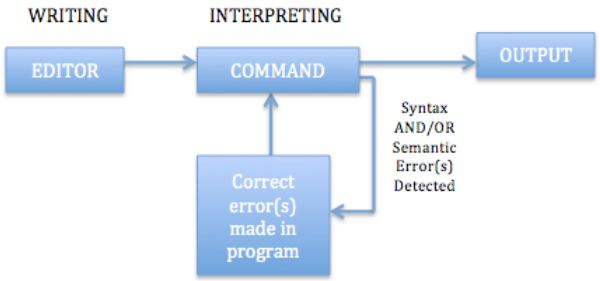


Figure 9: Outline of Command Line Programming (using an Interpreted Language)

Table 2. Command Line Programming Operations (Interpreted Language)

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is interpreted and checked for syntax and semantic errors. (Students must use the appropriate command to invoke this behavior)
Step 3	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

3.2.2 IDE Programming

Similar to command line programming, students are directed to an editing window to begin composing their program in an IDE. Students must also save their program as a file for the remaining procedures. Next, students must test the correctness of their written program. Depending upon the IDE, this can occur in different ways. For example, many IDEs provide a menu option that enables students to automatically compile, link, and execute their program file with a single mouse click. During this process, students are faced with one of three scenarios: 1) If a syntax error(s) is detected during compilation, this error must be corrected before the file automatically proceeds to the linking phase. 2) If a semantic (or logical) error(s) occurs, this error must be corrected before the file is successfully executed. 3) If no errors are detected during this process, the output of the program would be generated and viewed either within the same window of the editor or in an independent window.

Other IDEs follow a similar procedure seen in command line environments, which allow students to compile (and link) their program independently of execution. Instead of using a command terminal to do so, a menu option is provided to conduct this procedure. The output generated during execution from these particular IDEs can also be viewed either within the same window of the editor or in an independent window.

For languages that are interpreted, certain IDEs are built to interpret a written language using a menu option that invokes this behavior using a single mouse click. Upon interpretation, the output is also generated and viewed either within the same window of the editor or in an independent window. Figure 10 provides an outline of IDE programming that includes all three styles of operation. Table 3-5 provides a summarized list of each style of IDE operation respectively.

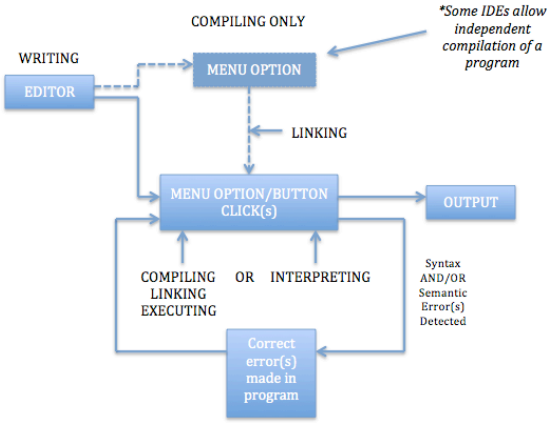


Figure 10: Outline of IDE Programming

Table 3. IDE Programming Operations (Compiling, Linking, and Executing automatically)

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is compiled, linked, and executed based upon the correctness of the written code. (Students must use the appropriate menu option to invoke this behavior). During this process, the file is checked for syntax and semantic errors.
Step 3	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

Table 4. IDE Programming Operations (Compiling/Linking and Executing independently)

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is compiled and checked for syntax errors. (Students must use the appropriate menu option to invoke this behavior)
Step 3*	The executable file of the written program is executed to acquire the intended output. (Students must use the appropriate menu option to invoke this behavior)
Step 4	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

Table 5. IDE Programming Operations (using an Interpreted Language)

Step 1	Editing window is used to compose (or write) program. (Program should be saved as a file)
Step 2*	The file of the written program is interpreted. (Students must use the appropriate menu option to invoke this behavior). During this process, the file is checked for syntax and semantic errors.
Step 3	The program's output is generated and viewed.
**May require multiple attempts due to syntax or semantic errors.*	

3.2.3 Syntax-Free Programming

Syntax-free programming also provides students with an editing window to create their program. Instead of using syntax as a method for composing a program, students in many cases must *drag* snippets of code from other windows of the environment and *drop* them into the editing window. Once a program has been created, students must also save their program as a file for the upcoming procedures. Similar to IDEs, syntax-free environments provide a menu option for students to test the correctness of their written program. During this process, students are faced with one of two scenarios: 1) If a semantic (or logical) error(s) occurs, this error must be corrected before the program can be executed. In many cases, errors can be corrected by either *discarding* inappropriate code from the composed program or dragging/dropping additional snippets of code into the same program. 2) If no errors are detected during this process, the output of the program would be generated and viewed either within the same window of the editor or in an independent window. Figure 11 provides an outline of syntax-free programming. Table 6 provides a summarized list of these operations in their respective order.

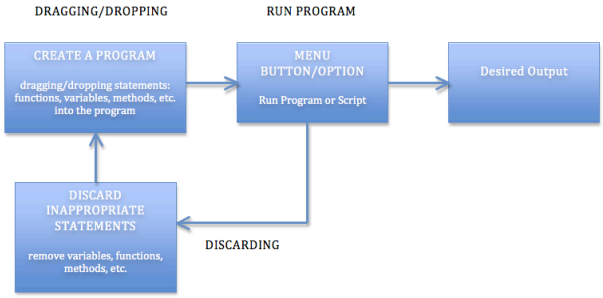


Figure 11: Outline of Syntax-Free Programming

Table 6. Syntax-Free Programming Operations

Step 1	Editing window is used to compose program. (Program should be saved as a file)
Step 2*	The file of the composed program is tested. (Students must use the appropriate menu option to invoke this behavior). During this process, the file is checked for semantic errors.
Step 3	The program's output is generated and viewed.
**May require multiple attempts due to semantic errors.*	

3.2.4 Discussion

Command line programming directly exposes students more to basic procedures for programming, such as compiling a written program, generating an executable file of a program through linking, and executing the executable file to generate the program's output. Students have to manually perform each procedure using certain commands to obtain the output of their written program. In contrast, visual environments can potentially provide a shorter process for students to conduct the same behavior. Because visual environments are usually operated using menu bars, icons, and mouse clicks, students are exposed to a higher level of abstraction for operation and navigation while programming. However, this style of construct may misrepresent some of the basic procedures for programming. For example, a student who is initially exposed to programming through an IDE may get the impression that clicking the appropriate menu option magically makes their program work while disregarding the actions of compiling, linking, executing, or interpreting.

4. STUDIES

To further examine the effects of visual environments on students while learning to program, a study was conducted on a CS1 lab and lecture course respectively at *The University of Alabama*. Section 4.1 discusses the first study that was conducted on the CS1 lab. Section 4.2 talks about the second study that was conducted as a semester-long assessment on the CS1 lecture course.

4.1 Study #1

The first study was conducted as a one-day pilot study for measuring the initial effects of visual and command line programming on students. The CS1 lab course generally introduces students to robotic programming through a syntax-free environment called PREOP that allows them to program real robots using drag-and-drop procedures in Alice. This particular course has no prerequisites and two or three sections are usually offered per semester. Three sections were offered during the time of this study (Spring 2011).

4.1.1 Methods & Procedures

For this study, each section received an environment to conduct Python programming: Section 1 received an *IDLE IDE* (Figure 12), Section 2 was given a *PyScripter IDE* (Figure 13), and Section 3 used *Notepad/Command Prompt* (Figure 14). Three measures were conducted for student assessment: *Computer Programming Self-Efficacy Scale* [16], *a time on task assessment,* and a *usability survey*.

The number of students enrolled in the CS1 lab course was 133. There were 45, 45, and 43 students enrolled in the IDLE, PyScripter, and Notepad sections respectively. The student population for this study varied for each procedure. This was due to students either arriving late to class or not correctly following the instructions. Therefore, the student population represented in this study ranged from 91-102 students. Tables 7-20 (with exception to Table 13) list the numbers of students who participated during each assessment.

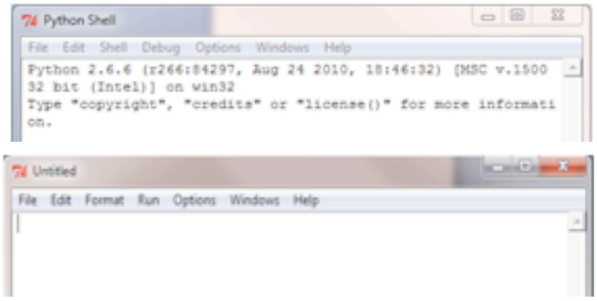


Figure 12: IDLE IDE version 2.6.6

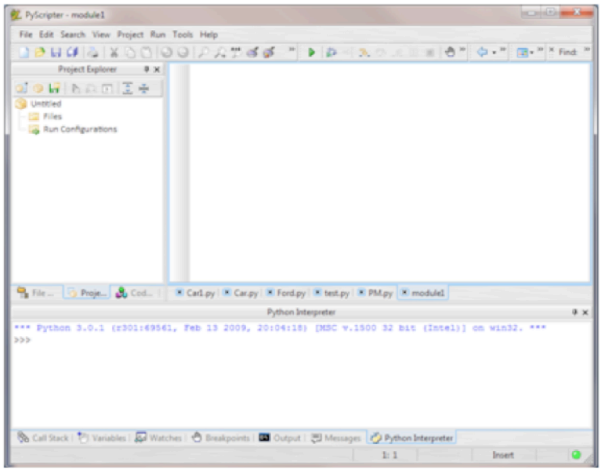


Figure 13: PyScripter version 1.9.9.6

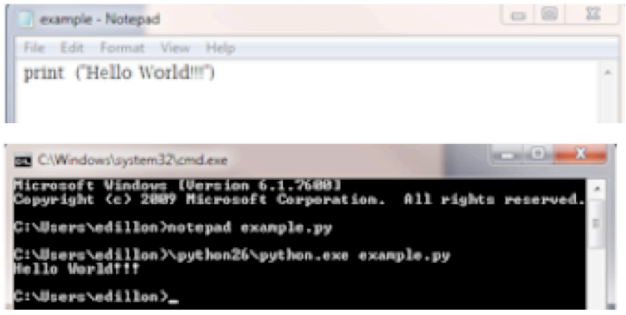


Figure 14: Notepad/Command Prompt – Windows Platform

To begin the study, each student received a self-efficacy survey. This survey consisted of 31 questions from the Computer Programming Self-Efficacy Scale. The responses were given on a 7-point Likert scale that ranged from *not confident at all* to *absolutely confident*. As part of this survey, a demographics section was provided in order to acquire feedback about the students, which included *academic major*, *classification,* and *prior programming experience*.

Next, the students received an introductory lecture on the Python language. This lecture introduced basic Python concepts that the students would need to complete the exercise. Students were exposed to concepts of *code syntax and semantics*, *selection*, and *information hiding*. Topics that were covered included *print statements, variable usage and assignment*, *reserved keywords* and *mathematical operations (with inferences on division and modulus usage)*. The lecture concluded by showing an example program using every topic. *This program converted* ***x*** *number of minutes into* ***h*** *hours and* ***m*** *minutes remaining.* The behavior of this program resembled the assignment that the students would be asked to write.

After the lecture, the students received a demonstration on how to use their respective environment and were required to write a small program that converted 700 days into *y* years, *m* months, and *d* days remaining. During this process, their time to complete this task was measured. The objective was to measure the students' time on task for writing the required program using their respective environment. For the IDLE group, a process monitoring application was used to measure time on task. In order to access their time logs, the students first accessed the process monitoring application before using IDLE, and then remain logged onto their computers after completing the assignment. However, some students did not follow these directions correctly which resulted in their time logs being lost. Therefore, the remaining two sections did not use the software. Instead these students were required to start at the same time and were required to raise their hands upon completing the assignment. The time on task for these sections was calculated by subtracting time of completion from the starting time.

After the time on task assessment, a usability survey was issued. This survey was composed of questions that directly focused on the students' experience with their respective tool. These questions measured subjective attributes regarding attitudes and feelings about using these environments respectively.

4.1.2 Results

The student demographics consisted of different majors at varying classification levels with contrasting levels of prior programming experience (Tables 7 - 10). For instance, the PyScripter group had more *Electrical Engineering* majors than *Computer Science.* The PyScripter and Notepad groups had significantly more *juniors* than the IDLE group *($p<0.05$)*. The IDLE group had less prior programming experience than the PyScripter group, which was also significantly less *($p<0.05$)* than the Notepad group. In addition, the Notepad group had a higher percentage *(50%)* of students who were taking the CS1 lecture course in conjunction with this lab. Traditionally, CS1 teaches Python programming using the VIM command editor on the Linux platform.

4.1.2.1 Self-Efficacy

The self-efficacy survey was used as an indicator for initially determining the students' self-efficacy for programming prior to their participation in this study. This survey measured the students' confidence for performing certain programming procedures ranging from *writing syntactically correct programs* to *writing a program that someone else could successfully comprehend.* The students' scores on this survey reflected their self-efficacy, meaning that a high score indicated an individual to have a high self-efficacy toward programming (and vice versa). The highest score that could have been made on this survey was 217. From this survey, the students showed an overall mean self-efficacy score of *114.85* out of 217 (with a normalized mean of *0.51* on a scale of 0 to 1).

The mean self-efficacy scores (see Table 11) amongst the three sections were tested using a one-way ANOVA. The ANOVA showed a significant variation amongst the three sections *($p<0.01$)*.

Table 7. CS1 Lab Demographics
**Number of responses before Time on Task was conducted.*

Participants (N=94*)	
Major	Computer Science - 33% Electrical Engineering - 29% Computer Engineering - 15% MIS - 3% Math - 5% Other - 18%
Classification	Freshmen - 41% Sophomore - 32% Junior - 22% Senior - 3% Other - 1%
Programming Experience	CS1 programming - 31% High School programming - 26% Another College Course - 18% No Experience - 26%

Table 8. CS1 Lab Demographics – IDLE
**Number of responses before Time on Task was conducted.*

IDLE Group (N=30*)	
Major	Computer Science - 37% Electrical Engineering - 27% Computer Engineering - 23% MIS - 7% Math - 7% Other - 7%
Classification	Freshmen - 57% Sophomore - 37% Junior - 7% Senior - 0% Other - 0%
Programming Experience	CS1 programming - 17% High School Course - 17% Another College Course - 17% No Experience - 40%

Table 9. CS1 Lab Demographics – PyScripter
**Number of responses before Time on Task was conducted.*

PyScripter Group (N=38*)	
Major	Computer Science - 24% Electrical Engineering - 42% Computer Engineering - 13% MIS - 3% Math - 3% Other - 18%
Classification	Freshmen - 32% Sophomore - 37% Junior - 39% Senior - 0% Other - 3%
Programming Experience	CS1 programming - 34% High School Course- 16% Another College Course - 24% No Experience - 26%

Table 10. CS1 Lab Demographics – Notepad
**Number of responses before Time on Task was conducted.*

Notepad Group (N=26*)	
Major	Computer Science - 42% Electrical Engineering - 12% Computer Engineering - 8% MIS - 4% Math - 8% Other - 31%
Classification	Freshmen - 38% Sophomore - 19% Junior - 31% Senior - 12% Other - 0%
Programming Experience	CS1 programming - 50% High School Course - 27% Another College Course - 8% No Experience - 15%

The ANOVA test was followed by T-tests to determine whether specific differences existed amongst the sections. The T-tests showed a significant difference between the IDLE and PyScripter groups *(p<0.01)* as well as the IDLE and Notepad groups *(p<0.01)* respectively. There was no significant difference between the PyScripter and Notepad groups. This indicated that students in the IDLE group were less confident in their programming abilities than their counterparts in the PyScripter and Notepad groups respectively.

Table 11. Self-Efficacy Descriptive Data for CS1 Lab

Group	N	Mean Score (Possible Score)	StdDev	Normalized Mean (scaling from 0 to 1)
IDLE	30	88.30 (out of 217)	38.91	0.42
PyScripter	38	125.63 (out of 217)	49.57	0.53
Notepad	26	129.73 (out of 217)	38.90	0.59
All Groups	94	114.85 (out of 217)	46.83	0.51

4.1.2.2 Time on Task

Overall, the average performance time for students to complete the assignment was *24.63 minutes* (Table 12). A one-way ANOVA showed a significant difference *(p<0.01)* between the average performance times amongst the three sections. The ANOVA test was followed by T-tests which showed a significant difference between the IDLE and PyScripter groups *(p<0.05)*, the IDLE and Notepad groups *(p<0.01)*, and the PyScripter and Notepad groups *(p<0.01)*. This indicated that students who used PyScripter finished their required task quicker than the students using IDLE and Notepad respectively. At the same time, students who used IDLE completed their task quicker than the students using Notepad.

Table 12. Time on Task Descriptive Data for CS1 Lab

Group	N	Average Time	StdDev
IDLE	21	23.05 minutes	12.62
PyScripter	40	15.88 minutes	10.89
Notepad	30	34.97 minutes	16.83
All Groups	91	**24.63 minutes**	**13.45**

4.1.2.3 Environment Usability

This survey was composed of several attributes to measure the environments' usability. Questions in the survey are listed in (Table 13). The results that were generated from the students' response to each question are also discussed in further detail. Tables 14-20 provide statistical analysis for each attribute measured.

Table 13. Usability Attributes
(OE = Open Ended; MC = Multiple Choice)

Attribute	Question
Initial Impression of Environment	OE
Comfort with Environment	MC
Confidence with Doing Another Assignment with Environment	MC
Fondness of Environment	MC
Easiest Attributes about the Environment	OE
Hardest Attributes about the Environment	OE
Experiences with Other Environments (besides PREOP)	OE

Initial Impression about the Environment. The responses were quantified into three categories: positive, non-positive, and no response. Non-positive responses consist of either neutral/confused or negative feelings about the environment. For quantification, the positive responses received a value of 1, and the non-positive and no responses received a value of 0.

A one-way ANOVA indicated a significant difference *(p<0.01)* amongst the three groups. Afterwards, T-tests indicated a significant difference for each T-test: IDLE vs. PyScripter *(p=0.05)*, IDLE vs. Notepad *(p=0.05)*, PyScripter vs. Notepad *(p<0.01)*. These results showed that the Notepad group had a less positive initial impression than the IDLE and PyScripter groups respectively. In addition, students in the IDLE group had a less positive initial impression than the PyScripter group. Table 14 provides further analysis about this measure.

Table 14. Initial Impression of Environment

Group	N	Mean	StdDev
IDLE	34	0.35	0.49
PyScripter	38	0.55	0.50
Notepad	30	0.17	0.38
All Groups	102	0.37	0.48

The mean was calculated by labeling Positive Responses = 1, and Non-Positive and No Responses = 0.

Comfort with Environment. Based on the response choices ranging from *not comfortable at all* to *absolutely comfortable*, a one-way ANOVA indicated a significant difference *(p<0.01)*. Afterwards, T-tests indicated a significant difference for two of the pairings: IDLE vs. PyScripter *(p<0.01)* and IDLE vs. Notepad *(p<0.05)*. These results showed that the IDLE group was less comfortable with using IDLE than the PyScripter group with PyScripter and the Notepad group with Notepad respectively. The PyScripter and Notepad groups showed no significant difference between each other. Table 15 provides further analysis about this measure.

Table 15. Comfort with Environment

Group	N	Mean	StdDev
IDLE	34	3.44	1.52
PyScripter	38	4.63	1.53
Notepad	30	4.30	1.74
All Groups	102	4.14	1.65

*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Not Comfortable at All** to **7 = Absolutely Comfortable***

Confidence with Doing Another Assignment with the Environment. Based on the response choices ranging from *not confident at all* to *absolutely confident*. A one-way ANOVA indicated a significant difference *($p<0.01$)*. Afterwards, T-tests indicated a significant difference for two of the pairings: IDLE vs. PyScripter *($p<0.01$)* and IDLE vs. Notepad *($p<0.05$)*. These results showed that the IDLE group was less confident with using IDLE to do another assignment than the PyScripter group with PyScripter and the Notepad group with Notepad respectively. The PyScripter and Notepad groups showed no significant difference between each other. Table 16 provides further analysis about this measure.

Table 16. Confidence with Doing Another Assignment with Environment

Group	N	Mean	StdDev
IDLE	34	3.38	1.67
PyScripter	38	4.74	1.64
Notepad	30	4.37	1.96
All Groups	102	4.18	1.82

*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Not Confident at All** to **7 = Absolutely Confident***

Like the Environment. Based on the response choices ranging from *not at all* to *absolutely like*. A one-way ANOVA indicated a significant difference *($p<0.01$)*. Afterwards, T-tests indicated a significant difference for two of the pairings: IDLE vs. PyScripter *($p<0.01$)* and PyScripter vs. Notepad *($p<0.01$)*. The students in the IDLE and Notepad groups liked IDLE and Notepad respectively less than the PyScripter group with PyScripter. No significant variations were noted between the IDLE and Notepad groups. Table 17 provides further analysis about this measure.

Table 17. Fondness of Environment

Group	N	Mean	StdDev
IDLE	34	3.41	1.73
PyScripter	38	4.87	1.66
Notepad	30	3.77	1.79
All Groups	102	4.06	1.81

*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Not at All** to **7 = Absolutely Like***

Easiest Attributes about the Environment. The responses were quantified into five categories: *Python Attributes*, *Environment Attributes*, *Familiarity*, *Nothing/No Response* and *I Don't Know*. Python Attributes represented students who gave a response about the Python language. Environment Attributes represented students who gave a response about their respective environment based on its features. Familiarity represented students who responded based on a previous experience with programming. The categories of Nothing/No Response and I Don't Know represented students who actually provided such responses. For quantification, responses that were categorized as Environment Attributes received a value of 1. All other responses received a value of 0.

A one-way ANOVA indicated no significant difference amongst the three groups. Since many of the students were not exposed to Python prior to this study, several of them responded more frequently about the easiest attributes of the Python language itself rather than their respective environment. A T-test indicated a significant difference *($p<0.05$)* between responses towards the Python language and the respective environments. Additional T-tests were used to determine any significant differences within each group. The results indicated a significant difference *($p<0.01$)* for only the IDLE group. These results showed that the IDLE group responded more frequently about the easy attributes of the Python language rather than the IDLE environment. The frequency of responses to Familiarity, Nothing/No Response, and I Don't Know were insignificant. Table 18 provides further analysis about this measure.

Table 18. Easiest Attributes of the Environment

Group	N	Mean	StdDev
IDLE	34	0.18	0.37
PyScripter	38	0.37	0.49
Notepad	30	0.37	0.49
All Groups	102	0.30	0.46

The mean was calculated by labeling Environment Attributes = 1 and all other categories = 0.

Hardest Attributes about the Environment. The responses were also quantified using the same categories as shown for the easiest attributes. For quantification, responses that were categorized as Environment Attributes received a value of 1. All other responses received a value of 0. A one-way ANOVA indicated a significant difference *($p<0.01$)*. Afterwards, T-tests indicated a significant difference for two of the pairings: IDLE vs. Notepad *($p<0.01$)* and PyScripter vs. Notepad *($p<0.01$)*. These results showed that Notepad received more responses concerning its hard attributes than IDLE and PyScripter respectively.

In regards to the responses about the Python language itself, a one-way ANOVA indicated a slight significant difference *($p=0.054$)*. Afterwards, T-tests indicated a significant difference for two of the pairings: the IDLE group vs. the Notepad group *($p=0.01$)* and the PyScripter group vs. the Notepad group *($p<0.05$)*. These results showed that students in the Notepad group gave fewer responses about the hardest attributes of the Python language than the IDLE and PyScripter groups respectively. The frequency of responses to Familiarity**,** Nothing/No Response, and I Don't Know were insignificant. Table 19 provides further analysis about this measure.

Table 19. Hardest Attributes of the Environment

Group	N	Mean	StdDev
IDLE	34	0.06	0.24
PyScripter	38	0.11	0.31
Notepad	30	0.40	0.50
All Groups	102	0.18	0.38
The mean was calculated by labeling Environment Attributes = 1 and all other categories = 0.			

Experiences with Other Environments (Besides PREOP). This particular question was asked in conjunction with another question: *Was the environment mandatory for a course?* Statistical analyses were conducted for both questions.

A one-way ANOVA was used to determine if certain sections had more prior experience with other environments besides PREOP. The results indicated a significant difference *($p<0.01$)*. Afterwards, T-tests were used to compare each group against another. A significant difference was found for two of the T-tests: the IDLE group vs. the PyScripter group *($p<0.01$)* and the IDLE group vs. the Notepad group *($p<0.01$)*. These results showed that the IDLE group has less experience with using other environments (besides PREOP) than the PyScripter and Notepad groups respectively.

A one-way ANOVA was also used to determine if these other environments were mandatory for another course. The results indicated a significant difference *($p<0.05$)*. Afterwards, T-tests indicated a significant difference for only one of the pairings: the IDLE group vs. the PyScripter group *($p<0.01$)*. These results not only showed that the PyScripter group had more experience with other environments than the IDLE group, but also that they were mandatory for another course. The PyScripter and Notepad groups showed no significant difference amongst each other. Table 20 provides further analysis about this measure.

Table 20. Experiences with Other Environments (besides PREOP)

Group	N	Mean	StdDev
IDLE	34	0.26	0.45
PyScripter	38	0.68	0.47
Notepad	30	0.50	0.51
All Groups	102	0.49	0.50
The mean was calculated by labeling Yes = 1 and No = 0.			

An additional T-test was used for the PyScripter group to determine whether their experience with other environments were actually IDEs. For the PyScripter group, the results were significant *($p<0.01$)*. These results showed that most of these students (68%) had prior experience with IDEs. As previously mentioned, many of the students in the PyScripter group were ECE majors. Traditionally at this university, all ECE majors must take CS285, which teaches the C language using the CodeBlocks IDE. Similar to PyScripter, CodeBlocks is an IDE rich with features. Out of the 68% of these students who had prior exposure to IDEs, 90% of them had experience with CodeBlocks prior to this study.

4.1.3 Discussion

The IDLE group had less prior programming experience than their counterparts in the PyScripter and Notepad groups. This factor may have impacted a majority of the results seen from this group. They were found to be less confident in their programming abilities, less comfortable with IDLE after using it, and less confident about doing another assignment. They also did not like IDLE as much as students who liked PyScripter. Their lack of programming experience was obvious when asked about the ease or difficulty of using IDLE. Instead of providing positive responses about IDLE, they expressed comfort about the Python language. Despite lacking programming experience, the IDLE group completed their task significantly faster than the Notepad group.

Students in the PyScripter and Notepad groups showed no differences in their programming experience. They also showed no differences in their comfort with their respective environments as well as their confidence of doing another assignment. However, the PyScripter group had a more positive initial impression, more of a fondness with PyScripter, and a faster completion time than the students using Notepad. Students in the Notepad group (not significantly) had more prior exposure to command line programming through CS1. However, they frequently showed difficulties with using Notepad, which influenced their time to complete the required exercise. In contrast, students using PyScripter rarely demonstrated difficulties about using PyScripter, and a majority of them had prior exposure to IDEs. In addition, 45% of the PyScripter group had a non-positive initial impression. On the other hand, 70% of the Notepad group had a non-positive initial impression. Fifty-three percent of the IDLE group showed a non-positive impression. However, many of the IDLE students did not have prior programming experience unlike the other groups.

4.2 Study #2

This study was conducted as part of a larger empirical evaluation of visual and command line programming in CS1 over the course of a semester. As previously mentioned, the CS1 course at the University of Alabama traditionally teaches Python using the VIM command line environment on the Linux platform. During the Fall 2011 semester, this course was altered to allow certain sections to use IDLE (in Linux) as an alternative to VIM. Four sections were offered during this particular semester; two sections were taught programming using VIM (Figure 15) and one section used IDLE (Figure 16). The remaining section, an honors section, was given the option of either tool. During the latter part of the semester, the non-honor sections were required to switch environments.

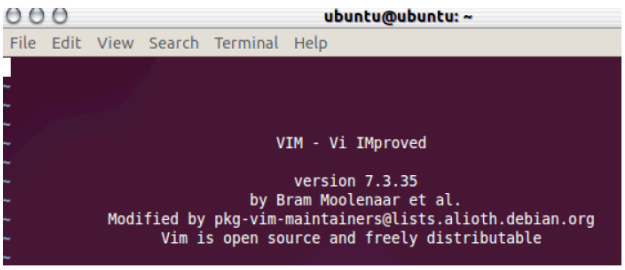


Figure 15. VIM version 7.3.35

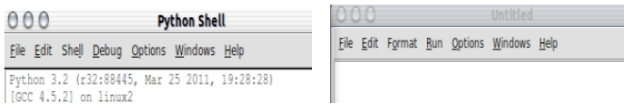


Figure 16. IDLE-Python 3.2

4.2.1 Methods/Procedures

As part of this empirical assessment, a demographic survey, three usability surveys and a protocol analysis were given during the semester. The usability surveys were administered twice before switching environments and once afterwards. After the environment switch, a protocol analysis was conducted on a small group of students to study their mental model for operating a visual or command line environment.

The number of students enrolled in the CS1 course was 179. There were 46, 88, and 45 students enrolled in the IDLE, two VIM, and honor sections respectively. Tables 21-27 list the numbers of students who participated during each assessment.

4.2.2 Results

The demographics shown in Tables 21-24 respectively are a representation of the CS1 student population *(N=119)* at the beginning of the semester. However, there were students who stopped attending class, dropped the CS1 course, or became agitated with participating in this study. These factors influenced a decrease in sample representations and student participation as the semester progressed, especially during the final assessments of this study.

Table 21. CS1 Demographics

Participants (N=119)	
Major	Computer Science - 61% Electrical Engineering - 3% Computer Engineering - 3% MIS - 1% Math - 6% Other - 22% Double Major (including CS) - 1% Double Major (excluding CS) - 3%
Classification	Freshmen - 40% Sophomore - 32% Junior - 19% Senior - 8% Other - 3% **one student did not provide an answer*
Programming Experience	High School programming - 16% Another College Course - 16% No Prior Experience - 68% **three students did not provide an answer*

Table 22. CS1 Demographics - IDLE Section

IDLE Section (N=33)	
Major	Computer Science - 85% Electrical Engineering - 0% Computer Engineering - 0% MIS - 0% Math - 6% Other - 9% Double Major (including CS) - 0% Double Major (excluding CS) - 0%
Classification	Freshmen - 34% Sophomore - 42% Junior - 15% Senior - 9% Other - 0%
Programming Experience	High School programming - 9% Another College Course - 25% No Prior Experience - 66%

Table 23. CS1 Demographics – VIM Sections

VIM Sections (N=46)	
Major	Computer Science - 49% Electrical Engineering - 2% Computer Engineering - 0% MIS - 2% Math - 9% Other - 29% Double Major (including CS) - 2% Double Major (excluding CS) - 7% **one student did not provide an answer*
Classification	Freshmen - 31% Sophomore - 27% Junior - 29% Senior - 11% Other - 2% **one student did not provide an answer*
Programming Experience	High School programming - 11% Another College Course - 9% No Prior Experience - 80%

Table 24. CS1 Demographics – Honor Section

Honor Section (N=40)	
Major	Computer Science - 56% Electrical Engineering - 5% Computer Engineering - 7% MIS - 0% Math - 5% Other - 27% Double Major (including CS) - 0% Double Major (excluding CS) - 3%
Classification	Freshmen - 55% Sophomore - 28% Junior - 10% Senior - 3% Other - 5%
Programming Experience	High School programming - 25% Another College Course - 17% No Prior Experience - 58%

4.2.2.1 Usability

One of the attributes measured in this survey was *Tool Mishandling* (Tables 25 and 26). Tool Mishandling was defined on the basis of how often students found themselves making errors, due to using IDLE or VIM incorrectly. This attribute was based on a 7-point Likert scale (*where 1 = absolutely often & 7 = absolutely NOT often*). *The results discussed are strictly based on the behavior of the non-honor sections.*

In the IDLE section, the results from a one-way ANOVA indicated a significant difference ($p<0.05$). Afterwards, T-tests indicated a significant difference for two of the pairings: 1st vs. 3rd surveys *($p<0.05$)* and 2nd vs. 3rd surveys *($p<0.01$)*. The results indicated two things: students in the IDLE section mishandled VIM more often than IDLE and the mishandling of a tool increased significantly after the switch. In the VIM sections, the results from a one-way ANOVA and T-Tests showed no significant difference. These results indicated students in the VIM sections did not mishandle one tool more often than the other.

Table 25. Tool Mishandling Results – IDLE Section
Avg = Average; SD = standard deviation

IDLE Section (IDLE to VIM)				
Tool	**Survey**	**N**	**Avg**	**StdDev**
IDLE	1st	31	4.10	1.42
IDLE	2nd	26	4.42	1.36
VIM	3rd	13	3.08	1.75
*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Absolutely Often** to **7 = Absolutely Not Often***				

Table 26. Tool Mishandling Results – VIM Sections
Avg = Average; SD = standard deviation

VIM Sections (VIM to IDLE)				
Tool	**Survey**	**N**	**Avg**	**StdDev**
VIM	1st	29	3.90	1.40
VIM	2nd	49	4.22	1.21
IDLE	3rd	39	4.41	1.58
*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Absolutely Often** to **7 = Absolutely Not Often***				

When comparing the average mishandling score between both groups after the environment switch (Table 27), the VIM sections showed a significantly higher average than the IDLE section *($p < 0.05$)*. This indicated that the VIM sections mishandled IDLE less often than the IDLE section did with VIM.

Table 27. Tool Mishandling Results *(after environment switch)*
Avg = Average; SD = standard deviation

Section	Tool	N	Avg	StdDev
IDLE	VIM	13	3.08	1.75
VIM	IDLE	39	4.41	1.58
*The mean was calculated using weights from a 7-point Likert scale, ranging from **1 = Absolutely Often** to **7 = Absolutely Not Often***				

*For further details about these results and other attributes measured during the usability assessment, see our paper published in the **Proceedings of the Human Factors and Ergonomics Society 56th Annual Meeting*** [7].

4.2.2.2 Protocol Analysis

This assessment was conducted during the week of the environment switch. The structure of this assessment allowed for the collection of both qualitative data and first-hand information about the CS1 students' mental model for programming. The objective was to determine whether certain features within these respective environments could shape the students' mental model for programming. The selection process for this assessment was based on random volunteers.

There were seven students who volunteered to participate in this study (all from non-honor sections); four were enrolled in the VIM sections and three were registered in the IDLE section. The same programming assignment was given to each student. Table 28 provides background information about each student. Similar to the assignment given during the CS1 lab study, the students had to write a program that converted 700 days into y years, m months, and *d* days remaining. A video camera was used to record the behavior of each student while completing this assignment. During the recording, each student had to "think aloud" about their approach for writing this program using their new environment. Each student was given 30 minutes to complete the assignment.

Table 28. Subject Background Information

**Student #4 was in the IDLE section but chose to use VIM in the course; **Student #6 was repeating the CS1 course;*

Student	Gender	Ethnicity	Prior Programming Experience	Environment (after switch)
S1	M	Caucasian	None	IDLE
S2	M	Caucasian	HTML	VIM
S3	M	Caucasian	HTML	VIM
S4	F	African American	None	IDLE**
S5	F	Caucasian	None	IDLE
S6	F	African American	VIM*	VIM
S7	M	African American	VI, C++, Java, Fortran	VIM

Each student who used VIM in this study (original IDLE users) indicated prior exposure to some form of programming before taking CS1. Each student from the VIM sections indicated otherwise. After conducting this assessment, the results showed that the students from the VIM sections had less challenges with using IDLE. Two of these particular students completed their assignment within the allotted time. The other two students' inability to complete the assignment was due to the difficulty of the assignment rather than IDLE. The three students from the IDLE section were not able to complete the assignment due to the challenges of using and understanding the VIM command editor. Table 29 provides a summarized description of the subjects' behavior during assessment.

Table 29. Subject Behavior

Student	Completed Assignment *YES*	*NO*	Reason for NOT Completing Assignment
S1	X		
S2		X	S2 spent the entire time trying to understand the functionality of the VIM editor.
S3		X	S3 spent most of her time trying to understand the functionality of the VIM editor.
S4		X	S4 struggled with understanding how to approach the assignment; She encountered several syntactical errors and struggled with correcting them.
S5	X		
S6		X	S6 struggled with understanding how to approach the assignment; She encountered semantic errors, which was due to her inability to determine the appropriate conversions for her program.
S7		X	S7 spent most of his time trying to understand the functionality of the VIM editor.

Another notable observation from this assessment relates to the subjects' tendency of reverting back to familiar procedures from their original tool if they felt lost or confused while using the new one. For example, the recording showed two of the original IDLE users attempting to use the menu bar of the command terminal assuming that VIM possessed relative features to IDLE. One of the original VIM users began using the command terminal to interpret her program when she felt unsure about performing this procedure in IDLE, but managed to complete this assignment.

We concluded from this assessment that feature sets in programming environments could play a role in shaping a novice's perception of programming. This study also showed that visual environments could potentially enable students to develop an inaccurate depiction of programming. For further detail about the results from this assessment, see our paper published in the ***Proceedings of the 50th Annual ACM Southeast Conference*** [6].

4.2.3 Discussion

Students from the IDLE section showed a significant decrease in their ability to use a different tool after being exposed to IDLE. However, students from the VIM sections showed a slight increase in their ability to use a different tool after their exposure to VIM. After switching environments, the mean score for mishandling tools in the VIM sections remained significantly higher than the IDLE section. These results also support the findings from the protocol analysis. Participants from the IDLE section found it more challenging to transition to a command line tool after using IDLE, while students in the VIM sections had a better transitioning to a visual tool after exposure to VIM.

5. CONCLUSION

The objective of this article was to study visual environments and their potential effect on students who are learning to program. Prior studies have shown that visual environments can have both productive and unprofitable effects on a student's ability to become accustomed to programming. From our studies, it was shown that visual environments could provide students with a lower learning curve for operation, while having the potential of placing limitations on their mental depiction of programming.

In the first study, the familiarity of features in IDLE and PyScripter possibly played a role in lowering the learning curve for the students in the CS1-lab course. By the same token, some of these features may have placed a limitation on the skills that the IDLE students in the CS1 course acquired during the second study. Table 30 summarizes the outcomes from both studies.

The question remains of whether visual environments are "ideal" for teaching students how to program. Even though prior studies have shown visual environments to promote student retention [15], positive attitudes [9], and motivation [11] during exposure, our findings show that these environments may also cause students to develop a faulty mental model for programming. These results also support Chen and Marx's reasoning for moving their students from an IDE to command line programming [2]. Certain visual environments may be too restrictive for learning specific programming concepts and procedures. In this case, it may be necessary for students to be exposed to other programming environments that are more inclined to round out their skill sets.

As an alternative solution, it may be appropriate to train students to understand the implied behavior of visual environments. For instance, students may need to receive appropriate training for understanding programming procedures before being exposed to a

Table 30. Study Outcomes

	Outcome	Reason
Study 1	Visual environments can initially impose a lower learning curve	The IDLE group completed their programming tasks significantly faster than their counterparts who used Notepad despite having less prior experience and a lower self-efficacy for programming. Students in the PyScripter and Notepad groups had more prior programming with using IDEs and command line environments respectively, however the PyScripter group completed their programming tasks significantly faster.
	Outcome	**Reason**
Study 2	Visual environments may impose a greater challenge for a student to directly transition to a command line environment	From the usability assessment, it was found that the students from the IDLE section showed a significant decrease in their ability to use VIM after being exposed to IDLE. From the protocol analysis, it was found that all of the IDLE participants were unable to complete their tasks due to struggling with using and understanding the VIM editor.

visual environment. By understanding these underlying factors, it may be possible for a student to avoid the acquisition of a faulty mental model for programming while also being able to make a smoother transition to other types of environments.

5.1 Threats to Validity

There are potential threats that could affect the validity of our findings from these studies. One threat is the finite set of environments that were evaluated during these assessments. Every visual environment that is used to teach programming was not evaluated during these studies. Instead, our studies were conducted while using theories, prior conclusions, and anecdotal evidence as point of references. Another threat relates to the short-term duration of the CS1 Lab study. This particular study was only composed of a one-day assessment. A third issue relates to the low students samples during the latter assessments in the CS1 lecture course. As previously mentioned, there were students who stopped attending class, dropped the course, or showed agitation toward participation in this study due to the repeated assessments.

5.2 Future Work

One future work is to improve student participation during these empirical assessments. This could be done by adjusting the amount of instruments employed during a study to obtain a high number of responses at a consistent level. A related future work is to assess students at particular times of the semester when the attendance rate tends to be high on a consistent basis.

Another area of future work relates to the actual programming environments. Some of the environments used during the CS1 lab and lecture studies consisted of tools primarily for Python programming. A primary future work is to apply evaluations to environments outside of the Python language.

6. FUNDING SOURCE

This work was conducted independent of any financial support.

7. REFERENCES

[1] Beaubouef, T. & Mason, J. (2005). Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *SIGCSE Bulletin, 37(2)*, 103-106.

[2] Chen, Z. & Marx, D. (2005). Experiences with Eclipse IDE in programming courses. *Journal of Computing Sciences in Colleges, 21(2)*, 104-112.

[3] Crosby, M. E. & Stelovsky, J. (1990). How Do We Read Algorithms? A Case Study. *Computer 23(1)* 24-35.

[4] Depasquale, P. J. (2003) Implications on the Learning of Programming Through the Implementation of Subsets in Program Development Environments. *Doctoral Thesis. UMI Order Number: AAI3095195., Virginia Polytechnic Institute and State University.*

[5] Dillon E., Anderson M., & Brown M. (2012). Comparing Feature Assistance Between Programming Environments and Their Effect on Novice Programmers. *Journal for Computing Sciences in Colleges, 27(5),* 69-77.

[6] Dillon E., Anderson M., & Brown M. (2012). Comparing Mental Models of Novice Programmers when using Visual and Command Line Environments. *In Proceedings of the 50th Annual ACM Southeast Conference,* 142-147.

[7] Dillon E., Anderson M., & Brown M. (2012). Studying the Novice's Perception of Visual Vs. Command Line Programming Tools in CS1. *In Proceedings of the Human Factors and Ergonomics Society 56th Annual Meeting, vol. 56(1),* 605-609.

[8] Guzdial, M. (2004). Programming environments for novices. *In Computer Science Education Research. S. Fincher and M. Petre (Eds.). Swets and Zeitlinger. Chapter 3.*

[9] Hagan, D., & Markham, S. (2000). Teaching Java with the BlueJ environment. *In 17th Annual Proceedings of Austrailian Society for Computers in Learning in Tertiary Education.*

[10] Kelleher, C. & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys. 37(2),* 83-137.

[11] Kelleher, C. Pausch, R., & Kiesler, S. (2007). Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems,* 1455-1464.

[12] Lewis, M. (2010). How programming environment shapes perception, learning and goals: logo vs. scratch. In *Proceedings of the 41st ACM technical symposium on Computer Science Education,* 346-350.

[13] Maloney,J., Peppler K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: urban youth learning programming with scratch. *SIGCSE Bulletin, 40(1),* 367-371.

[14] McWhorter, W. I.& O'Connor, B. C. (2009). Do LEGO® Mindstorms® motivate students in CS1*?. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education.* 438-442.

[15] Moskal, B., Lurie, D. & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. *In Proceedings of the 35th ACM Technical Symposium on Computer Science Education,* 75-79.

[16] Ramalingam, V. & Wiedenbeck, S. (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. *In Papers Presented At the Seventh Workshop on Empirical Studies of Programmers*,124-139.

[17] Sharp, H., Rogers, Y., & Preece, J. (2007). *Interaction Design: Beyond Human-Computer Interaction* . Hoboken, NJ: John Wiley & Sons Inc.

[18] Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers, 11(3),* 255-282.

August 2014

Volume 5 Issue 1

TABLE OF CONTENTS

ISSN 2153-4136 (online)