ARL-TR-8410 • JULY 2018

US Army Research Laboratory

Controlling Experiments Using Mathematical Sequences

by Sidney C Smith and Robert J Hammell II

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8410 ● JULY 2018

US Army Research Laboratory

Controlling Experiments Using Mathematical Sequences

by Sidney C Smith
Computational and Information Sciences Directorate, ARL

Robert J Hammell II
Department of Computer and Information Sciences, Towson University, Towson, MD

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
July 2018	Technical Report	October 2014–December 2017

4. TITLE AND SUBTITLE
Controlling Experiments Using Mathematical Sequences

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Sidney C Smith and Robert J Hammell II

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIN-D
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8410

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
primary author's email: <sidney.c.smith24.civ@mail.mil>.

14. ABSTRACT
This report explores the use of mathematical sequences to control the experiments used in the US Army Research Laboratory's exploration of the impact of packet loss on network intrusion detection and Kelly criterion–based lossy network compression. Herein, we describe initial attempts to control these experiments and review the problems associated with these attempts. The solution we propose is a tool suitable for use in Bourne-Again Shell scripts that will generate mathematical sequences. We then outline the requirements of this tool, discuss the approach for building this tool, and examine the results. We conclude by demonstrating the use and utility of the tool, and explore its superiority over previous methods.

15. SUBJECT TERMS
mathematical sequence, arithmetic, geometric, triangular, square, cube, Fibonacci

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Sidney C Smith
Unclassified	Unclassified	Unclassified	UU	91	19b. TELEPHONE NUMBER (Include area code) 410-278-6235

Contents

List of Figures vi

List of Tables vi

1. Introduction 1

2. Background 1

3. Requirements 4

3.1 Functional Requirements 4

3.1.1 Overflow 4

3.1.2 Sequences 4

3.1.3 Command Line Interface (CLI) 4

3.1.4 Output 4

3.2 Nonfunctional Requirements 5

3.2.1 Online Manual 5

3.2.2 Developer's Guide 5

3.2.3 Report 5

3.2.4 Extensible 5

3.2.5 Portable 5

3.2.6 Tested 5

4. Approach 6

4.1 Software Development Environment 6

4.1.1 Language 6

4.1.2 Build Environment 6

4.1.3 Documentation Tools 7

4.1.4 Unit Testing Framework 7

4.1.5 Integrated Development Environment 7

4.2 Software Design 8

4.2.1 Configuration Module 8

4.2.2 Sequence Module 8

4.2.3 Main Function 16

5. Results 17

5.1 Requirements Revisited 17

5.1.1 Functional Requirements 17

5.1.2 Nonfunctional Requirements 18

5.2 Evaluation 18

5.3 Case Study 19

5.3.1 Geometric 20

5.3.2 Fibonacci 21

5.3.3 Cube 22

5.3.4 Square 23

5.3.5 Triangular 24

5.3.6 Arithmetic 24

5.3.7 Summary 24

5.4 Examples 25

5.4.1 Dropit 25

5.4.2 Entropit 26

5.4.3 Snapit 26

6. Conclusion 27

7. References 29

Appendix A. Sequence User's Manual Page 32

Appendix B. ConfigItem Programmer's Manual Page 36

Appendix C. Configuration Programmer's Manual Page 40

Appendix D. Sequence Programmer's Manual Page 45

Appendix E. Arithmetic Programmer's Manual Page 50

Appendix F. Cube Programmer's Manual Page 53

Appendix G. Fibonacci Programmer's Manual Page **56**

Appendix H. Geometric Programmer's Manual Page **59**

Appendix I. Square Programmer's Manual Page **62**

Appendix J. Triangular Programmer's Manual Page **65**

Appendix K. CUTE Test Results **68**

Appendix L. Raw Data from EE4D984HC2000R20 **79**

List of Symbols, Abbreviations, and Acronyms **82**

Distribution List **83**

List of Figures

Fig. 1 Sequence class diagram....9
Fig. 2 Simple sequence plot....10
Fig. 3 Arithmetic sequence where a = 5 and d = 5 11
Fig. 4 Triangular and arithmetic sequences 12
Fig. 5 Square and triangular sequences 13
Fig. 6 Cube and square sequences 14
Fig. 7 Fibonacci and cube sequences....15
Fig. 8 Geometric and Fibonacci sequences 16
Fig. 9 Experimental environment 4....20
Fig. 10 PLR vs. the ALR using the geometric sequence....21
Fig. 11 PLR vs. the ALR using the Fibonacci sequence 22
Fig. 12 PLR vs. the ALR using the cube sequence....23
Fig. 13 PLR vs. the ALR using the square sequence 23
Fig. 14 PLR vs. the ALR using the triangular sequence 24
Fig. 15 PLR vs. the ALR using the arithmetic sequence....25

List of Tables

Table 1 Configuration item attributes....8
Table 2 Sequence values 9
Table 3 Summary of experiments 25
Table 4 Traceability matrix 28
Table L-1Raw Data from EE4D984HC2000R20....80

1. Introduction

During our exploration into the impact of packet loss on network intrusion detection, we needed to conduct several iterations of an experiment altering a single control variable (e.g., the drop rate for the packet dropper[1] or the replay speed multiplier for `pcapreplay`[2] and `tcpreplay`[3]). We needed to automate each iteration to allow an experiment to be completed quickly. We found enumerating the values for each iteration in the script that automated the trial to be tedious and error prone. The `expr`[4] command, which is typically used to make calculations in `bash`[5] scripts, will only perform simple arithmetic on integers. The `bc`[6] arbitrary precision calculator language supports more complicated calculations, and we did implement shell functions for the arithmetic and geometric sequence this way. However, we wanted a more general tool with a simpler interface. We designed the `sequence` program to allow the user to select a sequence and specify the values of key variables either in a configuration file, the environment, or the command line. It will print the numbers of the sequence all on the same line suitable for a shell `for` statement or print each number on its own line suitable for a shell `while` statement.

In Section 2, we review the evolution of the `sequence` program and explain the problems it was written to solve. In Section 3, we enumerate the requirements for the `sequence` program. In Section 4, we present the design of the program and discuss the design decisions. In Section 5, we report the results of incorporating the `sequence` program into our experiment control scripts. In Section 6, we provide our conclusions and review possibilities for future work.

2. Background

While conducting the theoretical exploration of the impact of packet loss on network intrusion detection,[1] we needed to conduct repeated iterations of the packet dropper program altering the drop rate. In the beginning, we automated this through the configuration files of the XWray-SPEX platform.[7] Later execution of the packet dropper program was controlled through a shell script. First, we conducted 20 iterations at 5% intervals. Later, we conducted 97 iterations at 1% intervals. We placed these values into a shell variable and iterated over them using the shell's built-in `for` command. This approach is illustrated in Listing 1.

Listing 1 Shell variable example

```
#!/bin/sh
ALGORITHM=chance
DATASET=~/data/darp98fourhour.pcap
RULESET=~/rules/Circa2000/etc/snort.conf
DROPRATES="5␣10␣15␣20␣25␣30␣35␣40␣45␣50\
␣␣␣␣␣␣␣␣␣55␣60␣65␣70␣75␣80␣85␣90␣95"
for droprate in ${DROPRATES}; do
        mkdir ${droprate}
        pcktdrpr --algorithm ${ALGORITHM}\
                --droprate $droprate\
                ${DATASET} |
        snort -N -c ${RULESET} -r -\
                -l $droprate >
                $droprate/snort.out 2>&1
done
```

While conducting the experimental exploration of the impact of packet loss on network intrusion detection,[2,8–10] we found control of the experiments much more complicated. The experiments replayed the network traffic using `pcapreplay`[2] and `tcpreplay`[3] at multiples of the original speed to create packet loss. Repeatedly replaying traffic can be very time consuming because it is bound by the original time required to collect the traffic. We employed a simple geometric sequence, 2^n, to quickly discover the most interesting range of speed multipliers to study because theoretically the entire range can be explored in twice the original elapse time of the network traffic. In practice, this takes a little longer. Originally, we implemented this using the shell variable and the `for` command, but we quickly abandoned this technique. Next, we used the `expr` command to multiply the speed by 2 for each iteration. Based upon these results, we conducted a second experiment using an arithmetic sequence to explore the range of interest. The problem with this approach is that we needed to modify the controlling script each time we changed the sequence—a process that was tedious and prone to errors. This was especially true for experiments for host-level[9] and multilevel[10] packet loss because these experiments required 2 scripts, one on the sending side and one on the receiving side, that we needed to correctly modify for each experiment. Our first step in resolving this problem was to write the shell scripts `sendpcap` and `receivepcap`. These scripts each contained the shell functions `arithmetic` and `geometric`.

Listing 2 Arithmetic shell function

```
function arithmetic() {
        AsubN=`expr ${asub1} + \( $1 - 1 \) \* $d`
        echo -n ${AsubN}
        return 0
}
```

Listing 3 Geometric shell function

```
function geometric() {
        AsubN=`echo "${asub1} * $r ^ ($n - 1)" | bc`
        echo -n ${AsubN}
        return 0
}
```

The `arithmetic` function implements the arithmetic sequence described in Eq. 1 where n is the number of the term, a_n is the n^{th} term in the sequence, a_1 is the first term in the sequence, and d is the common difference. We present the code to implement the arithmetic shell function in Listing 2. The `geometric` function implements the geometric sequence described in Eq. 2, where a_n, a_1, and n are the same as above and r is the common ratio. We present the code to implement the geometric shell function in Listing 3. These scripts allowed the first term, common difference, common ratio, and the range of n to be specified on the command line. This improvement allowed us to conduct multiple experiments without modifying the controlling scripts.

$$a_n = a_1 + (n-1)d. \tag{1}$$

$$a_n = a_1 r^n. \tag{2}$$

Although this was an improvement, we still had issues to resolve. In addition to the script to send the traffic and the script to receive the traffic, a script was written to estimate how long an experiment will take. This means that we needed to maintain the functions to compute the sequences in 3 different scripts. Further, the geometric sequence grows very quickly and is excellent for getting an overview of

the relationship between packet loss rate (PLR) and alert loss rate (ALR). The arithmetic sequence trudges along at a steady pace that can be too time consuming for a given area of interest. For these reasons we wanted to implement other sequences that might be a better compromise between time and completeness. Adding a new algorithm would require modifying at least 3 separate scripts. The solution was to pull the sequence generation into a single separate program that all 3 scripts may leverage.

3. Requirements

Any software development effort must begin with a clear understanding of the requirements. In the following subsections we will list the requirements for the sequence tool.

3.1 Functional Requirements

3.1.1 Overflow

The Fibonacci and geometric sequences have rapid growth rates, and arithmetic overflow can happen quickly. The sequence program will detect and terminate cleanly in the event of an arithmetic overflow.

3.1.2 Sequences

The sequence program will implement the following mathematical sequences to cover the growth rates from the arithmetic to the geometric: arithmetic, triangular, square, cube, Fibonacci, geometric.

3.1.3 Command Line Interface (CLI)

The sequence program is required to be usable inside a script. This means that it must be configured through the command line, the environment, or a configuration file. Menus and graphical user interfaces will not be useful in this environment.

3.1.4 Output

Each number in the sequence will be written to standard output separated by spaces or new lines. The separator, either a space or a new line, will be a configurable item. All diagnostic information will be written to standard error.

3.2 Nonfunctional Requirements

3.2.1 Online Manual

The sequence program itself and the major modules that compose it will be documented with online manual pages.

3.2.2 Developer's Guide

The classes, methods, functions, and files will be documented with comments that can be interpreted by the Doxygen[11] tool to produce a guide for developers to aid in the expansion of the program.

3.2.3 Report

A technical report will be written documenting the motivation, requirements, approach, and the results of integrating the sequence tool into scripts designed to conduct experiments.

3.2.4 Extensible

The sequence tool will be designed to allow additional mathematical sequences to easily be added.

3.2.5 Portable

The tool will be portable to various POSIX[12] compliant environments. It will employ a build environment that will discover and account for any operating system idiosyncrasies.

3.2.6 Tested

Unit tests will be written and executed to ensure that each method performs as intended. Attention will be given to edge cases especially arithmetic overflow conditions.

4. Approach

We will divide our approach into 2 sections. The first section will deal with the software development environment (SDE). The second section will deal with the software design.

4.1 Software Development Environment

The SDE includes the computer language, build environment, documentation tools, unit testing framework, and integrated development environment (IDE). We also employed a source code repository and issue tracking system; however, these are outside of the scope of this report.

4.1.1 Language

We chose to implement the sequence tool using C++. C++ has good support for querying the environment, allowing us to easily meet Requirement 3.1.3. The object-oriented features of C++ allow us to meet Requirement 3.2.4.

The C++ standard library contains the `getenv` function to query the environment and the `getopt_long` function, which supports both long and short command line options. We were able to leverage the configuration module that had been written for the packet dropper.[1]

The object-oriented features of C++ allowed the bulk of the housekeeping associated with the implementation of each sequence to be inherited from the base class. This greatly reduces the work involved when adding a new mathematical sequence.

The sequence program could have been written in Java, Perl, or Python. A prototype was written in Java. Producing command line tools in Java is not as simple as producing command line tools in either C++, Perl, or Python. The final reason for selecting C++ is that the development of this program was used to explore and demonstrate the SDE, including the IDE, build system, documentation, and unit testing frameworks.

4.1.2 Build Environment

We used GNU Autotools[13] as the build environment to support Requirement 3.2.5. GNU autoconf[14] will query the environment and configure the program to properly compile and run. GNU automake[15] will create the makefile that properly builds the

system. This includes adding standard targets (e.g., "distclean" to remove all files created in the build process, "install" to properly install the executables, libraries, and manual pages, and "check" to run the unit test). GNU libtool[16] handles all of the different ways that share libraries are implemented. In light of Miller's paper "Recursive make considered harmful",[17] we built the system using nonrecursive make.

4.1.3 Documentation Tools

We provided documentation written using the `troff`[18,19] text processor to be compatible with on-line manual provided by many UNIX[20] and UNIX-like systems. The user's manual page for the `sequence` program may be found in Appendix A. The programmer's manual pages for modules that compose the `sequence` program may be found in Appendixes B–J. We used the Doxygen[11] tool to produce the developer's guide mentioned in Requirement 3.2.2. It also produces programmer's manual pages based upon the comments contained in the code. We leveraged this capability to satisfy Requirement 3.2.1. Doxygen allows each file, class, function, and variable to be documented just before or near to the implementation. This has the potential to greatly aid in keeping the documentation timely and complete.

4.1.4 Unit Testing Framework

We used the CUTE[21] C++ unit testing framework and fulfilled Requirement 3.2.6. This choice was based upon the ready availability of the plug-in for Eclipse[22,23] in the Eclipse Marketplace. Other unit testing frameworks could have been used; in fact, the Configuration module has been tested using Boost,[24] CPPUNIT,[25] Google Test,[26] and TAP[27,28] unit testing frameworks. A comparison of these unit testing frameworks is beyond the scope of this report; however, the code for the sequence tool may serve as a good platform to conduct that testing.

4.1.5 Integrated Development Environment

We used the Eclipse[22,23] IDE because it happens to be the IDE of choice for our team. A comparison of the strengths and weaknesses of IDEs is outside the scope of this report. Eclipse does support development in C++. It has native support for the autotools[13] suite. The Eclox[29] plug-in is available in the marketplace to provide integrated support for Doxygen.[11] There exists a CUTE[21] plug-in in the marketplace to provide integrated support for unit testing.

4.2 Software Design

We decomposed the sequence program into 2 modules. The first module collects configuration information from the environment (i.e., configuration files, the environment, and the command line). The second module implements the various mathematical sequences.

4.2.1 Configuration Module

To satisfy Requirement 3.1.3, this program leverages the configuration module that was developed for the packet dropper. It is composed of the configuration item class and the configuration class. A configuration item allows the user to specify various attributes of the item defined in Table 1. The user creates and populates the attributes, and adds each configuration item to the configuration. The configuration class implements the singleton design pattern, which allows one and only one instance of the configuration class to exist. This allows each function to obtain their own pointer to the configuration without having to pass that pointer from function to function. The programmer's manual pages for the configuration item and configuration classes may be found in Appendixes B and C, respectively.

Table 1 Configuration item attributes

Name	Description
name	The name of the item
shortOption	The one letter command line option
longOption	The multi letter command line option
has_arg	Specify the kind of arguments
environment	The name of the environment variable
description	A description of the item
value	The default value of the item

4.2.2 Sequence Module

This program leverages the sequence module that contains the sequence class, which is a super class that all of the other classes inherit. This super class also contains the static functions necessary to implement the factory design pattern. This allows the static method makeSequence to be passed a string containing the name of the sequence. The static method will then match that name with the correct child class that implements that sequence. This also allows class names and descriptions to be registered and queried. This decouples the implementation of class from the main program. The main program can provide a list of available classes provided by

a static method of the sequence class, allowing the user to request that sequence without the main program ever having to know which sequences have been implemented. Figure 1 is a class diagram for the sequence module.

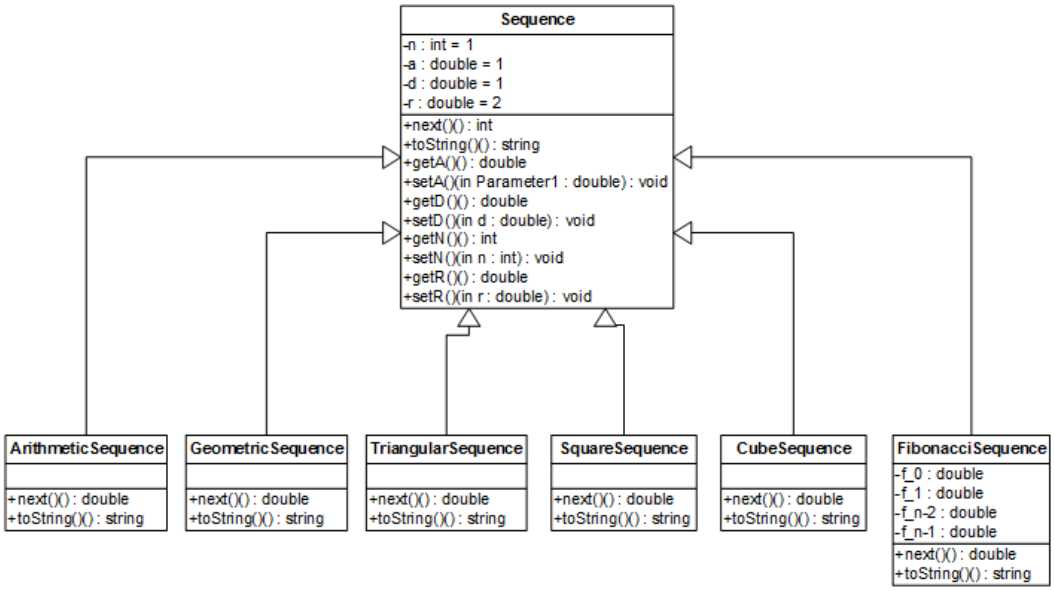

Fig. 1 Sequence class diagram

A summary of the currently implemented sequences follows. Each sequence is represented by an equation that uses the same set of variables that are defined in Table 2. (Not every variable is used by every sequence.)

Table 2 Sequence values

Variable	Full name	Description
a	First term	The first number in the sequence
d	Common difference	The common difference
n	Term number	The current position in the sequence
r	Common ratio	The common ratio
s_n	n^{th} term	The value of the given sequence term

C++ does not handle arithmetic overflow well or at all; therefore, it is important that each method tests for possible arithmetic overflow before computing the next value in the sequence to satisfy Requirement 3.1.1. The following sequence summaries briefly discuss how we test for overflow.

4.2.2.1 Simple

The simple sequence type implements the identity sequence as shown in Eq. 3. Figure 2 shows a plot of the simple sequence. This is an extremely slow and plodding progression. The sequence class is fully documented in Appendix D.

$$s_n = n. \tag{3}$$

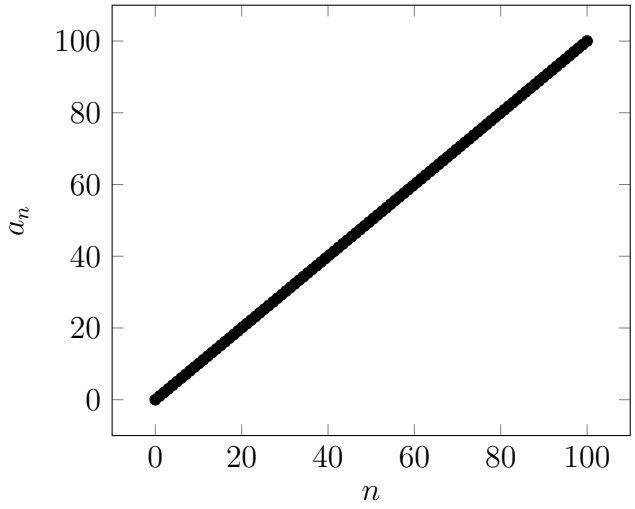

Fig. 2 Simple sequence plot

Testing for overflow in the simple sequence may be done by testing the value of n against $LONG\_MAX$.

4.2.2.2 Arithmetic

The Arithmetic sequence is any sequence of numbers where there is a common difference between the terms as seen in Eq. 4. Figure 3 shows a plot of the arithmetic sequence. Simliar to the simple sequence, this is an extremely slow and plodding progression. This makes sense since the simple sequence is an arithmetic sequence where $a = 0$ and $d = 1$. It is useful for closely examining a small area of interest. We used it because we wanted to understand what was happening in some of the large jumps in ALR. The arithmetic class is fully documented in Appendix E.

$$s_n = a + d(n - 1). \tag{4}$$

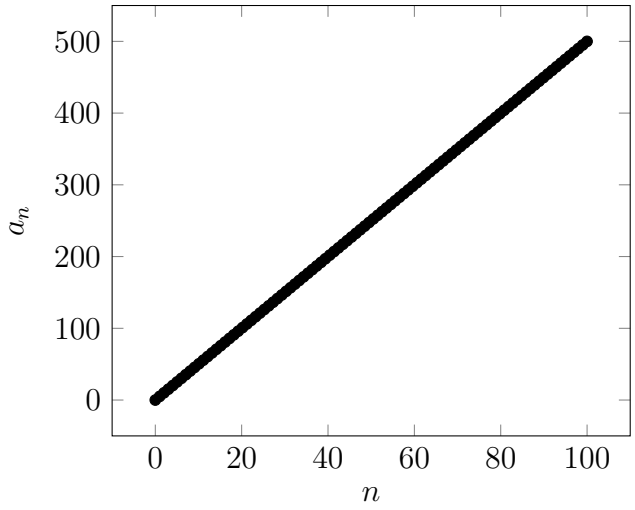

Fig. 3 Arithmetic sequence where a = 5 and d = 5

Testing for overflow in the arithmetic sequence may be done by solving for n and replacing s_n with DUB_MAX to discover the largest acceptable value for n as in Eq. 5.

$$n = \frac{DUB\_MAX - a}{d} + 1. \tag{5}$$

4.2.2.3 Triangular

The triangular sequence is a sequence of numbers where the number of objects could make an equilateral triangle. This may be defined as a sequence where n is multiplied by $(n+1)$ and the product is divided by 2 as seen in Eq. 6. Figure 4 shows a plot of the triangular sequence in black. The arithmetic sequence is plotted in blue to illustrate the difference in growth rates. This sequence was implemented to provide a middle ground between the plodding arithmetic sequence and the rocketing geometric sequence. The triangular class is fully documented in Appendix J.

$$s_n = \frac{n(n+1)}{2}. \tag{6}$$

Testing for overflow in the triangular sequence may be done by solving for n using the quadratic equation and replacing s_n with DUB_MAX to discover the largest acceptable value for n as in Eq. 8.

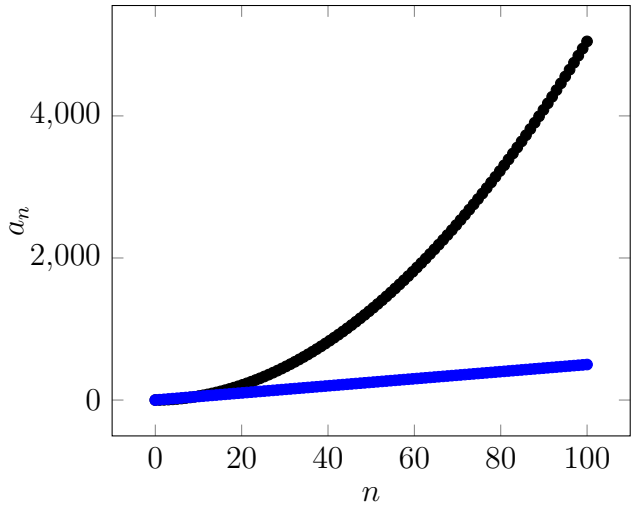

Fig. 4 Triangular and arithmetic sequences

$$0 = n^2 + n - 2DUB\_MAX. \tag{7}$$

In this Eq. 7 $a = 1, b = 1$, and $c = -2DUB\_MAX$ plugging that into the quadratic equation and discounting the negative root gives us

$$n = \frac{-1 + \sqrt{1 + 8DUB\_MAX}}{2}. \tag{8}$$

4.2.2.4 Square

The square sequence is a sequence of numbers where n is raised to the second power as seen in Eq. 9. Figure 5 shows a plot of the square sequence. The triangular sequence is plotted in blue to illustrate the difference in growth rates. This sequence was implemented to provide a middle ground between the plodding arithmetic sequence and the rocketing geometric sequence. It grows more rapidly than the triangular sequence but not as rapidly as the cube sequence. The square class is fully documented in Appendix I.

$$s_n = n^2. \tag{9}$$

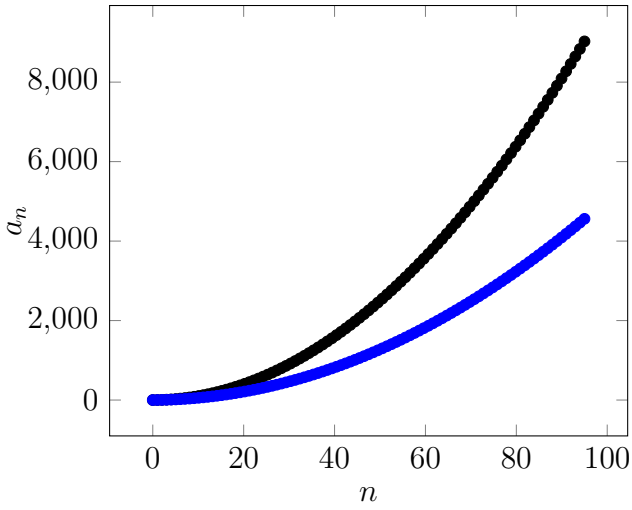

Fig. 5 Square and triangular sequences

Testing for overflow in the Square sequence may be done by solving for n and replacing s_n with DUB_MAX to discover the largest acceptable value for n as in Eq. 10.

$$n = \sqrt{DUB\_MAX}. \tag{10}$$

4.2.2.5 Cube

The cube sequence is a sequence of numbers where n is raised to the third power as seen in Eq. 11. Figure 6 shows a plot of the cube sequence. The square sequence is plotted in blue to illustrate the difference in growth rates. This sequence was implemented to provide a middle ground between the plodding arithmetic sequence and the rocketing geometric sequence. It grows more rapidly than the square sequence but not as rapidly at the Fibonacci sequence. The cube class is fully documented in Appendix F.

$$s_n = n^3. \tag{11}$$

Testing for overflow in the cube sequence may be done by solving for n and replacing s_n with DUB_MAX to discover the largest acceptable value for n as in Eq. 12.

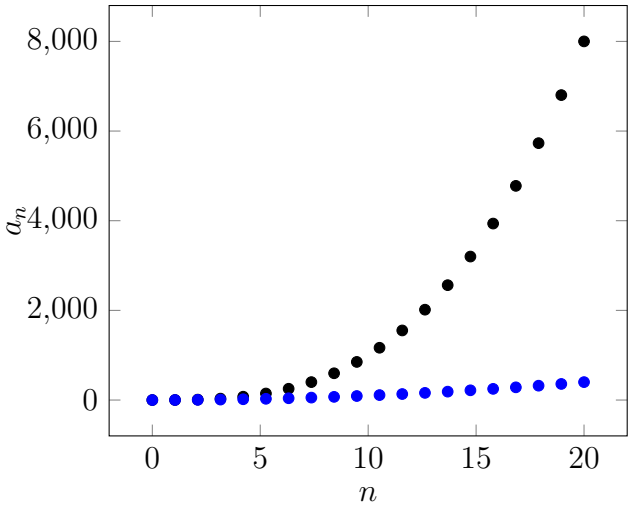

Fig. 6 Cube and square sequences

$$n = \sqrt[3]{DUB\_MAX}. \tag{12}$$

4.2.2.6 Fibonacci

The Fibonacci sequence implements the Fibonacci numbers where the first Fibonacci number, f_1, is 1, the second Fibonacci number, f_2, is 1, and each following Fibonacci number is the sum of the 2 proceeding Fibonacci numbers as seen in Eq. 13. Figure 7 shows a plot of the square sequence. The cube sequence is plotted in blue to illustrate the difference in growth rates. This sequence was implemented to provide a middle ground between the plodding arithmetic sequence and the rocketing geometric sequence. It grows more rapidly than the cube sequence but not as rapidly at the geometric sequence. It also has the property of starting out more slowly than the cube sequence before rocketing past it. The Fibonacci class is fully documented in Appendix G.

$$s_n - s_{n-2} + s_{n-1}. \tag{13}$$

The computation of Fibonacci numbers is a common exercise in computer science used to teach recursion. Fibonacci numbers may also be computed using Binet's formula as in Eq. 14 or 15.

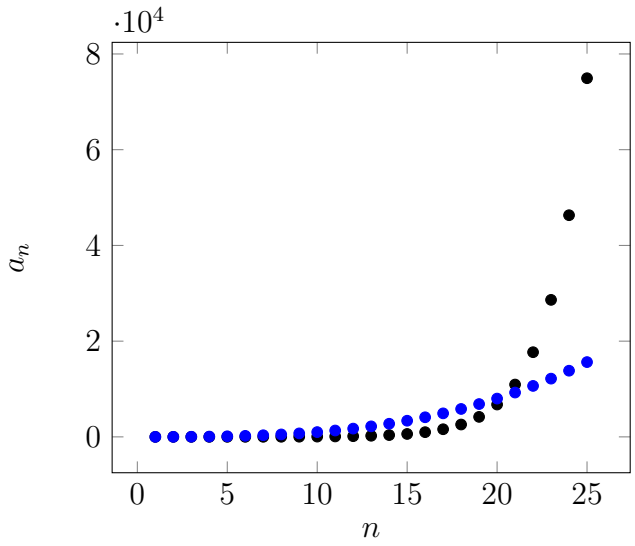

Fig. 7 Fibonacci and cube sequences

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}. \tag{14}$$

$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}. \tag{15}$$

Computing the Fibonacci sequence sequentially is fast and easy; however, computing it recursively can be expensive. Binet's formula is attractive because it reduces computing an arbitrary Fibonacci number to a small set of calculations. The problem with this method is that the term $(1+\sqrt{5})^n$ will overflow long before the Fibonacci number itself would overflow. One of the reasons that computing a Fibonacci number recursively is so resource intensive is that the same Fibonacci number is calculated several times. This could be addressed by maintaining a cache of previously computed Fibonacci numbers. It turns out to be fast and simple to initialize this cache with the numbers in the Fibonacci sequence when the sequence object is created. This also allows us to discover the maximum allowable Fibonacci number by using the Eq. 16.

$$DUB\_MAX - f_{n-2} < f_{n-1}. \tag{16}$$

4.2.2.7 Geometric

The geometric sequence is a sequence of numbers where there is a common ration between the terms as seen in Eq. 17. Figure 8 shows a plot of the geometric sequence where $a = 1$ and $r = 2$. It grows rapidly. The Fibonacci sequence is plotted in blue to illustrate the difference in growth rates. The geometric class is fully documented in Appendix H.

$$s_n = ar^{n-1}. \tag{17}$$

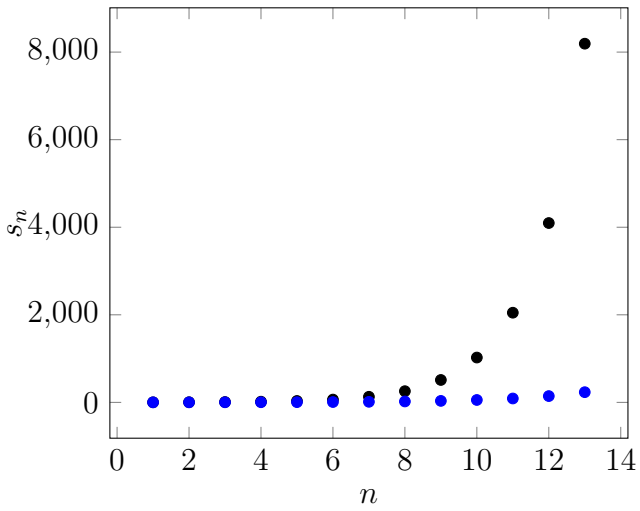

Fig. 8 Geometric and Fibonacci sequences

Testing for overflow in the geometric sequence may be done by solving for n and replacing s_n with DUB_MAX to discover the largest acceptable value for n as in Eq. 18.

$$n = \frac{log(\frac{s_n}{a})}{log(r)} + 1. \tag{18}$$

4.2.3 Main Function

The main program invokes the configuration module to collect the configuration information from the configuration files, environment, and command line. It then uses that information to create and configure a sequence object. This object is used to generate the numbers in the sequence. These numbers are then written to stan-

dard output separated by spaces or new lines based upon the value of that configuration item. The user interface for the sequence program is fully documented in Appendix A.

5. Results

In Section 5 we will revisit the requirements to demonstrate that each has been fulfilled. In Section 5.1.2.6 we will evaluate the `sequence` program's effectiveness in expanding and simplifying the scripts for experimental control. In Section 5.1.2.6 we will walk through of a case study using these mathematical sequences to explore the impact of packet loss on network intrustion detection. In Section 5.3.7 we will further demonstrate the use of the sequence program with some additional examples.

5.1 Requirements Revisited

5.1.1 Functional Requirements

5.1.1.1 Overflow

The `nextN` member function of the sequence class is used by all of the child sequence tests for overflow in `n`. This may be seen in the unit test results presented in Appendix K. The `next` member function implemented in the sequence class and overridden by each of the child sequence tests for overflow in the result.

Each child of the Sequence class tests for overflow when they override the `next` method.

This may be seen in the unit test results presented in Appendix K.

5.1.1.2 Sequences

The sequence module properly implements the required mathematical sequences. This may be seen in the unit test results presented in Appendix K.

5.1.1.3 Command Line Interface (CLI)

The `sequence` program implements the required command line interface. This may be seen in the user manual page presented in Appendix A.

5.1.1.4 Output

The `sequence` program implements output where the values are separated by spaces and by new lines. The interface to select the desired behavior may be found in Appendix A.

5.1.2 Nonfunctional Requirements

5.1.2.1 Online Manual

The online manual page describing the `sequence` program may be seen in Appendix A. The online manual pages describing the configuration module may be found in Appendixes B and C. The online manual page describing the sequence module may be found in Appendixes D, E, F, G, H, I, and J.

5.1.2.2 Developer's Guide

A draft developer's guide has been generated using the Doxygen tool.

5.1.2.3 Report

This document is the report required by Requirement 3.2.3.

5.1.2.4 Extensible

The sequence module employs inheritance and implements the factory pattern to allow new mathematical sequences to be implemented with a minimum amount of effort. This effort involves defining the `name` and `description` attributes, implementing the `next` function, and registering the new sequence in the static `createSequence` function. For the 6 mathematical sequences that we implmented, this amounted to an averaged of 49 lines of actual code.

5.1.2.5 Portable

Employing the GNU autotools[14] suite allows the program to be easily configured and compiled on different systems. Currently, versions 5 and 6 of Red Hat Enterprise Linux and CentOS 7 have been tested.

5.1.2.6 Tested

Unit tests were written for all of the key functionalities using the CUTE[21] unit testing framework. The results may be seen in Appendix K.

5.2 Evaluation

The primary purpose of implementing the `sequence` program was to simplify

the scripts that control the experiments in the Packet Loss Study. We used the `estimator` script as an example. The `estimator` script takes command line arguments to control the experiment, then the name of the ruleset and the name of the dataset. This information is used to compute how long it will take to run a particular experiment.

Initially, the `estimator` script used shell functions and the `expr` and `bc` commands to calculate the speed of each run. It implemented the arithmetic, geometric, triangular, square, and cube sequences. It only stepped forward, and the only overflow testing was that done by either `expr` or `bc`. This script contained 336 lines of code. Of those lines, 93 were comments, 31 were blank, 86 were used for processing the command line arguments, 54 were used for computing the sequences, 11 were used to query the dataset database, 9 were used to compute estimate, and 42 were used to provide the output. The new `estimator` script, which uses the `sequence` program, is 223 lines of code. Of those 223 lines, 78 are comments or blank lines. A reduction of 67 lines of shell code may seem insignificant; however, it also includes new functionality like the Fibonacci sequence, the ability to fall backwards, and proper handling of overflow. Further, the `estimator` script is only one script used to control these experiments. This same code would need to be added and maintained in the `sendpcap` and `receivepcap` scripts.

5.3 Case Study

We will demonstrate the use of mathematical sequences to control experiments by reviewing one of the experiments used in our exploration of the impact of packet loss.[10] In this experiment, we used the mathematical sequences to control the speed multiplier as we replay network traffic. In the experimental environment seen in Fig. 9, Albus and Severus each have Gigabit Ethernet network interface cards that are directly connected to each other using CAT-5e cable. Albus will replay the four_hours sample from the Defense Advanced Research Projects Agency (DARPA) 1998 dataset[30] using Tcpreplay.[3] Severus will capture the data using Snort[31] configured with a rule set tailored to malicious traffic seen in 1998 time frame. The sendpcap script will control the experiment from Albus. The receivepcap script will control the experiment from Severus. The PLR and ALR will be extracted from the output of Snort with the extract shell script.

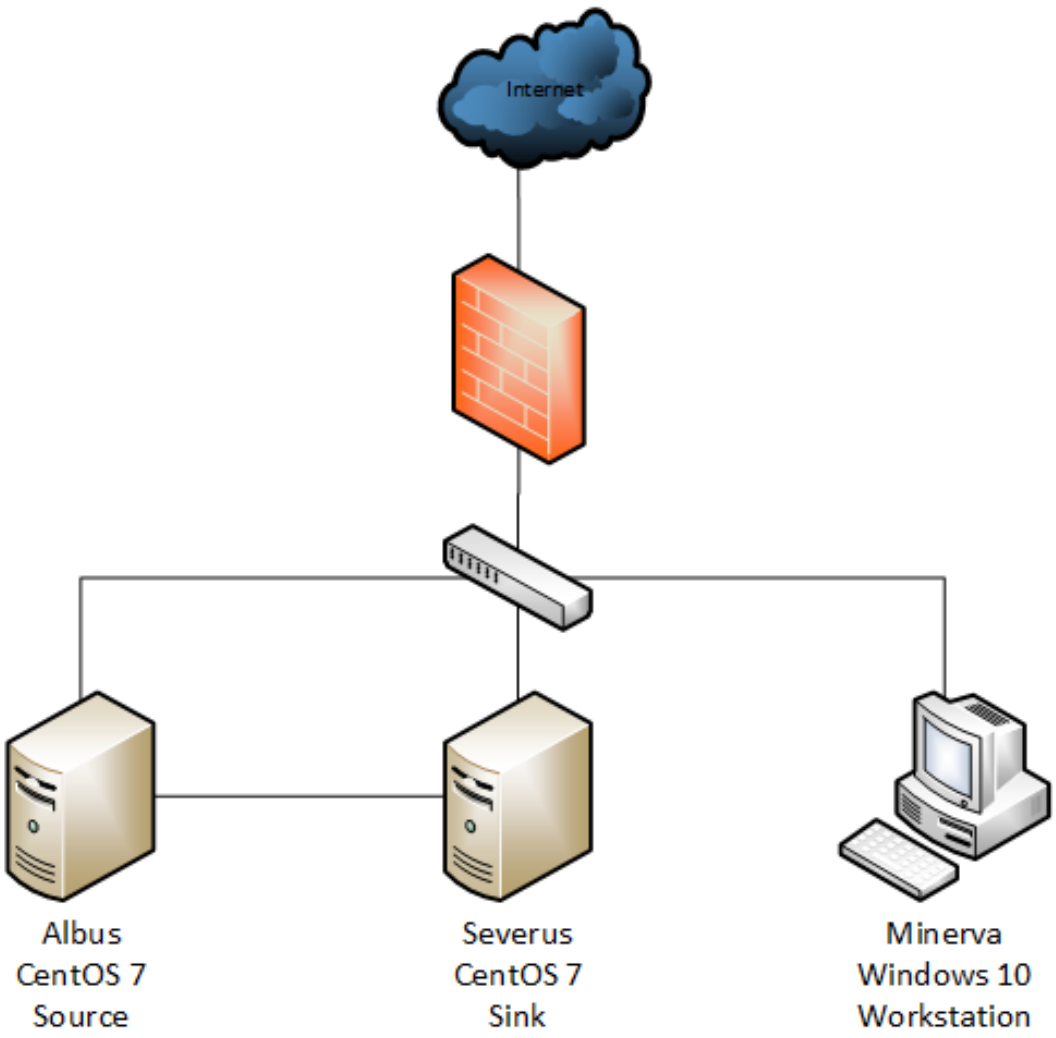

Fig. 9 Experimental environment 4

5.3.1 Geometric

We begin our exploration by conducting an experiment using the geometric arithmetic sequence. As we observed earlier, this sequence grows very rapidly, allowing us to gain an overview of the region to be explored. Theoretically, no matter how many iterations are run, this experiment should be completed in about 8 h. We chose to run an experiments with n falling from 60 to 0. This experiment completed in 40 h, 41 min, and 42 s for the following reasons. The four_hours dataset is actually 17 h 20 min and 57 s long. We added a 30-s lag time to allow Snort[31] to complete reading the configuration files. During our experimental exploration of sensor-level packet loss,[10] we discovered that it can take tcpreplay longer than the orginal elasp-

time divided by the speed factor. We added a 150-s minimum wait time to account for this. This run covered replay speeds from 1 to 576,460,752,303,423,488 times the original speed. The plot of PLR versus ALR may be seen in Fig. 10. The raw data is contained in Appendix L. Although this quickly provided an overview, there are not enough data points between PLRs of 12% and 35% for us to be certain that we know what is happening inside that range.

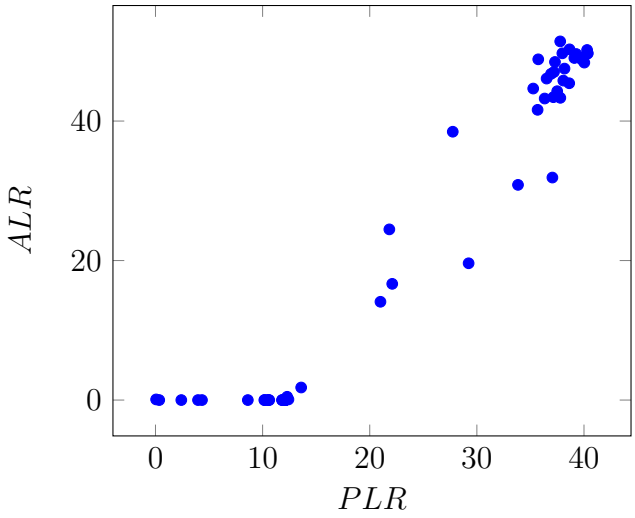

Fig. 10 PLR vs. the ALR using the geometric sequence

5.3.2 Fibonacci

We used the Fibonacci mathematical sequence to drill down into this region. Examining the raw data from the geometric experiment in Appendix L, we start to lose about 12% of the packets when the data is replayed at 262,144 times the original speed. We start hitting the 40% range around 4,000,000 times the original speed. We configured the experiment to use the 10^{th} through the 100^{th} Fibonacci number. This would cover replay speeds from 89 times the original speed to 354,224,848,179,261,997,056 times the original speed. We set the lag time to 30 s and the minimum wait time to 150 s. The elapsed time for this experiment was 5 h 29 s. The results of this experiment have been plotted in Fig. 11. This plot provides a more detailed view of the relationship than we saw using the geometric sequence; however, we would still like more data points to understand exactly what is happening.

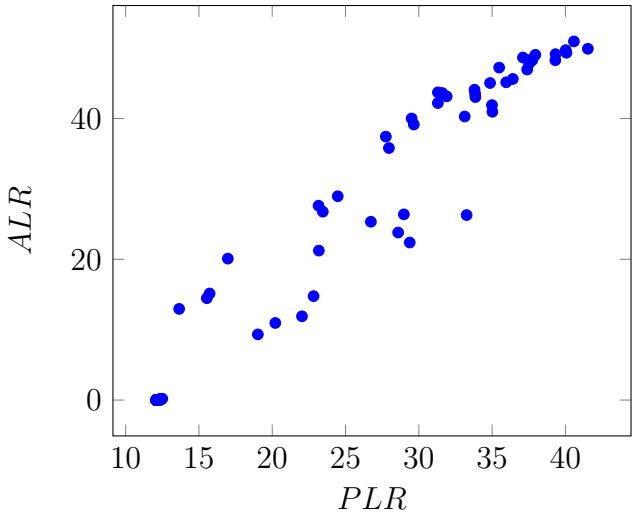

Fig. 11 PLR vs. the ALR using the Fibonacci sequence

5.3.3 Cube

A similiar examination of the data from the Fibonacci experiment leads us to select $n = 60$ through 160 for the next experiment using the cube sequence. This gives us a speed range of 233,428 to 4,096,000 times the original speed of the network traffic. After reviewing the output of the previous experiments, we discovered that at these speeds a lag time of 24 s and a minimum wait time of 36 s were sufficient. The DARPA 98 four_hour data set is 62,457 s long. Replaying this traffic at 233,428 times the initial speed should take place in less than a second. Since the `expr` command we use to compute the wait time only works with integers, the computation produces a wait time of 0 s. This is just as well because the `sleep`[32] command that we use to wait only supports sleeping for whole seconds. This means that the total wait time is completely composed of the lag time and the minimum wait time. Using the lag time of 24 s and minimum wait value of 36 s, each iteration will complete in 1 min, and the entire experiment will complete in 100 min. The results may be seen in Fig. 12. These 100 data points provide a clear picture of the relationship between 12% PLR and 45% PLR. If our goal were simply to study this relationship, we could stop here; however, we will continue to demostrate the use of the other sequences.

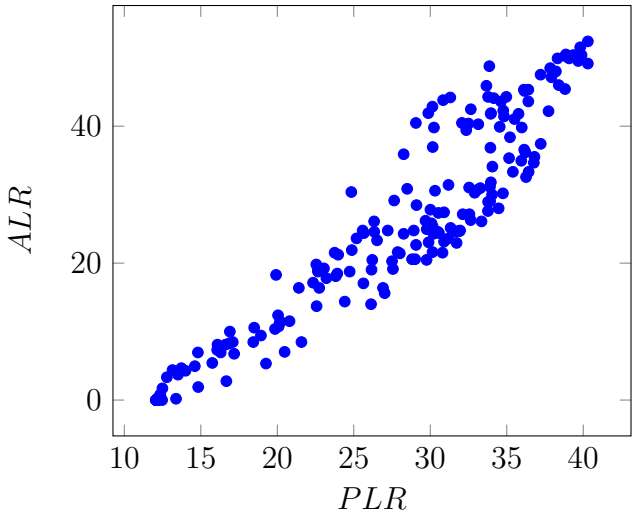

Fig. 12 PLR vs. the ALR using the cube sequence

5.3.4 Square

After examining the results from the cube run, we selected n from 1220 to 1000. This gives us a speed range of 1,000,000 to 1,488,400 times the original speed of the network traffic. This run took 3 h and 40 min. The results may be seen in Fig. 13.

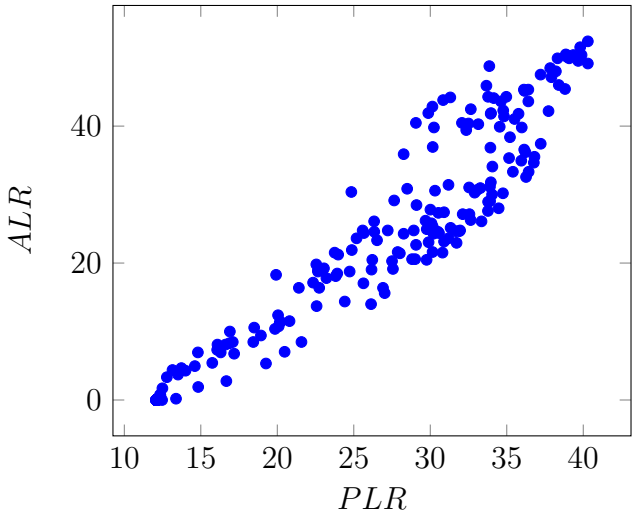

Fig. 13 PLR vs. the ALR using the square sequence

5.3.5 Triangular

After examining the results from the square run, we selected n from 1732 to 732. This gives us a speed range of 501,501 to 1,500,778 times the original speed of the network traffic. This run took 12 h and 12 min. The results may be seen in Fig. 14.

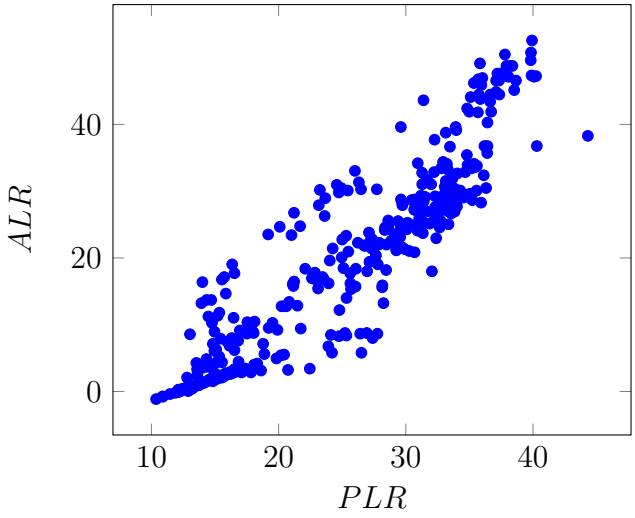

Fig. 14 PLR vs. the ALR using the triangular sequence

5.3.6 Arithmetic

After examining the results from the triangular run, we selected n from 0 to 202. We set the first value or a to 500,000 and the common difference to 50,000. This gives us a speed range of 500,000 to 1,050,000 times the original speed of the network traffic. This run took 20 h and 12 min. The results may be seen in Fig. 15.

5.3.7 Summary

This experiment demonstrates the power of the different sequences to provide a rapid overview of the data space or an intense exploration of a limit domain. The variation in the supported mathematical sequences allows the investigator to make trade-offs between the duration of the experiment and density of the domain. Table 3 illustrates some of these trade-offs. Notice that the geometric sequences allow a vast domain to be explored sparsely, the arithmetic sequence allows a domain to be explored intensely, and the others provide a rich middle ground.

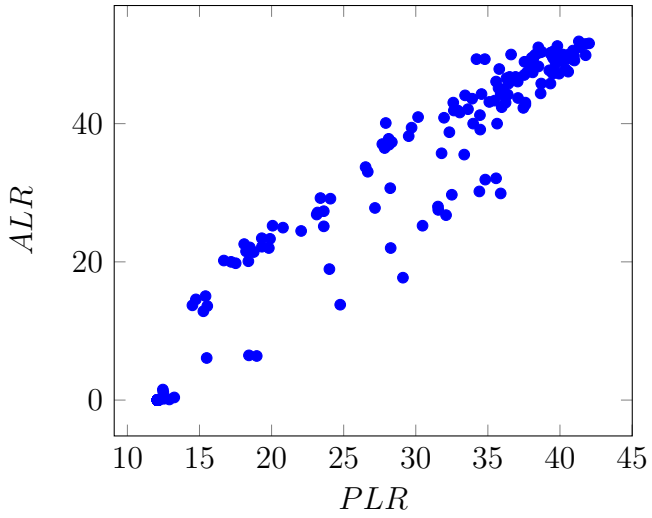

Fig. 15 PLR vs. the ALR using the arithmetic sequence

Table 3 Summary of experiments

Sequence	Range	Interations	Time
Geometric	1–576,460,752,303,423,488	60	40:41:42
Fibonacci	89–354,224,848,179,261,997,056	50	05:00:29
Cube	233,428–4,096,000	100	01:40:00
Square	1,000,000–1,488,400	220	03:40:00
Triangular	501,501–1,500,778	1000	12:12:00
Arithmetic	500,000–1,050,000	200	20:12:00

5.4 Examples

To further illustrate the utility of the sequence command, we will provide brief examples of how it has been integrated into other experimental control scripts. The `dropit` script was used in our theoretical exploration of the impact of packet loss[1] to control experiments using the packet dropper. The `entropit` script was used in our study of the use of entropy in lossy network traffic compression for network intrusion detection applications[33] to control the `entropinator`. The `snapit` script was used to control the snap length setting of `tcpdump` in our experiment to discover the effect of compressing network traffic using snap length for network intrusion detection applications.

5.4.1 Dropit

In the Introduction we mentioned the packet dropper appliction that we developed for our theoretical exploration of the impact of packet loss.[1] The `dropit` script that

automated experiments using the packet dropper could be modified to use the sequence program to control the drop rate. The command line to produce 20 iterations at 5% intervals from 0% to 95% can be seen in Listing 4. Running an experiment with 50 iterations at 2% intervals would be a simple as changing the count to 50 and the difference to 2.

Listing 4 Packet dropper example

```
$ dropit \
        --algorithm chance --limit 20 --difference 5 \
        --ruleset Circa2000 --type arithmetic\
        darpa98fourhour.pcap
```

5.4.2 Entropit

The entropy values of data sliced into 8-bit chunks run from 0 to 8. Since higher values closer to 8 indicate that the traffic is either compressed or encrypted, we wanted to remove packets with an entropy value higher than a certain threshold, and pass the remaining packets through Snort[31] to discover the impact that had on network intrusion detection. The `entropit` script automated an experiment to run 40 iterations where the threshold would move from 8 to 4 increments of 0.1. If the sequence program were incorporated into this script, this experiment could be run using the command line seen in Listing 5.

Listing 5 Entropit dropper example

```
$ entropit \
        --limit 40 --difference -0.1 --first 8 \
        --ruleset Circa2000 --type arithmetic \
        darpa98fourhour.pcap
```

5.4.3 Snapit

To explore the effect of truncating packets to compress network traffic on network intrusion detection, we added a snap length option to the `pcapcat` program. The `snapit` shell script was written to automate these experiments. The largest packet in the DARPA 98 four_hour data set is 1,514 bytes. The smallest packet is 42 bytes. The difference is 1,472 bytes. We can cover this range by conducting an experiment with 31 iterations ranging from 1542 to 42 with the `snapit` command seen in Listing 6.

Listing 6 Snapit dropper example

```
$ snapit \
        --difference 50 \
        --falling \
        --first 42\
        --limit 31 \
        --ruleset Circa2000 \
        --start 31 \
        --type arithmetic \
        darpa98fourhour.pcap
```

6. Conclusion

The goal of the sequence program is to provide researchers with a method to set the values of the control variable in an experiment-controlling script that would be more flexible and easier than the previous methods. The requirements for flexibility and maintainability are enumerated in Section 3. Table 4 provides a matrix that associates the definition of the requirement in Section 3 with the approach for meeting it in Section 4 and the results of implementing it in Section 5. Table 4 illustrates that the sequence program has satisfied all of the requirements. A comparison to the previous method is provided in Section 5.1.2.6. Athough there are more lines of C++ code than the shell code it replaced, the C++ code is more robust, more capable, better documented, and should be easier to maintain. Also, the C++ code will only need to be maintained in one place where the shell code would have had to be maintained in several. The case study using the packet loss study experiment in Section 5.1.2.6 and the examples provided in Section 5.3.7 demonstrate the use and utility of using mathematical sequences to control experiments.

Future work involves reviewing the code and the comments in the code to refine the developer's guide and prepare the entire package for public release. It is possible that the square and cube sequences could be more generalized into 2^{nd} and 3^{rd} order polynomials; however, the available sequence options are more than enough to meet the current requirement.

Table 4 Traceability matrix

Requirement	Approach	Result
Overflow 3.1.1	4.2.2	5.1.1.1
Sequences 3.1.2	4.2.2	5.1.1.2
CLI 3.1.3	4.2.1	5.1.1.3
Output 3.1.4	4.2.3	5.1.1.4
Manual 3.2.1	4.1.3, 4.2.1	5.1.2.1
Guide 3.2.2	4.1.3	5.1.2.2
Report 3.2.3	N/A	5.1.2.3
Extensible 3.2.4	4.2.2	5.1.2.4
Portable 3.2.5	4.1.2	5.1.2.5
Tested 3.2.6	4.1.4	5.1.2.6

7. References

1. Smith S, Hammell R, Parker T, Marvel L. A theoretical exploration of the impact of packet loss on network intrusion detection. International Journal of Networked and Distributed Computing. 2016;4(1):1–10.

2. Smith SC, Hammell RJ. An experimental exploration of the impact of sensor-level packet loss on network intrusion detection. In: Military Communications Conference, MILCOM 2016 IEEE; p. 752–757.

3. Turner A, Bing M. Tcpreplay: Pcap editing and replay tools for *nix. Boston (MA): 2005 [accessed 2015 Mar]. http://tcpreplay.synfin.net.

4. Parker M, Youngman J, Eggert P. expr(1) Linux user's manual. 6.9 ed. Raleigh (NC): Red Hat, Inc.; 2017 Feb.

5. Fox B, Ramey C. bash(1) Linux user's manual. 6.9 ed. Raleigh (NC): Red Hat, Inc.; 2016 Jun.

6. Nelson PA. bc(1) Linux user's manual. 6.9 ed. Raleigh (NC): Red Hat, Inc.; 2016 Jun.

7. Wray J, Mateo C, Parker T, Ritchey R, Smith S. X-wray stats and performance explorer. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2014 Apr. Report No.: ARL-TR-6895.

8. Smith SC, Wong KW, Hammell RJ, Mateo CJ. An experimental exploration of the impact of network-level packet loss on network intrusion detection. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2015 Aug. Report No.: ARL-TR-7371.

9. Smith SC, Hammell RJ, Wong KW, Mateo CJ. An experimental exploration of the impact of host-level packet loss on network intrusion detection. In: Cybersecurity Symposium (CYBERSEC); 2016. p. 13–19.

10. Smith SC, Hammell RJ, Wong KW, Mateo CJ. An experimental exploration of the impact of multi-level packet loss on network intrusion detection. In: 2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA); 2016. p. 23–30.

11. Van Heesch D. Doxygen: source code documentation generator tool.; 2008 [accessed 2018 Jan]. URL: http://www.doxygen.org.

12. International standard - information technology portable operating system interface (posix)base specifications, issue 7. ISO/IEC/IEEE 9945:2009(E). 2009;1–3880.

13. Zadok E. Overhauling amd for the'00s: a case study of gnu autotools. In: USENIX Annual Technical Conference, FREENIX Track; 2002 Jun; Monterey, CA. p. 287–297.

14. MacKenzie D, Elliston B, Demaille A. GNU autoconf creating automatic configuration scripts. 2.69 ed. Boston (MA): Free Software Foundation, Inc; 2012.

15. MacKenzie D, Tromey T, Duret-Lutz A, Wildenhues R, Lattarini S. GNU automake. 1.15.1 ed. Boston (MA): Free Software Foundation, Inc; 2017.

16. Matzigkeit G, Oliva A, Tanner T, Vaughan GV. GNU libtool. 2.4.6 ed. Boston (MA): Free Software Foundation, Inc; 2015.

17. Miller P. Recursive make considered harmful. AUUGN Journal of AUUG Inc. 1998;19(1):14–25.

18. Ossanna JF. Nroff/troff user's manual. Citeseer; 1976.

19. Welsh M. Writing man pages using groff. Linux Journal. 1995;1995(18es):3.

20. Ritchie O, Thompson K. The unix time-sharing system. The Bell System Technical Journal. 1978;57(6):1905–1929.

21. Sommerlad P, Graf E. Cute: C++ unit testing easier. In: Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications companion; p. 783–784.

22. Murphy GC, Kersten M, Findlater L. How are Java software developers using the elipse IDE? IEEE software. 2006;23(4):76–83.

23. eclipse. Ottawa, Canada: Eclipse Foundation, Inc.; 2018 [www.eclipse.org.

24. Rozenthal G. Unit test framework: user's guide. Onancock (VA): 2007 [accessed 2015 Mar]. http://www.boost.org/doc/libs/1_56_0/libs/test/doc/html/utf/user-guide.html.

25. Feathers M, Lepilleur B. Cppunit cookbook. Fairfax (VA): Sourceforge; 2002 [accessed 2018 Jan]. http://cppunit.sourceforge.net/doc/1.8.0/cppunit_cookbook.html.

26. Sen A. A quick introduction to the google C++ testing framework. IBM DeveloperWorks. 2010;20.

27. Schilli M. A perl toolbox for regression tests. Linux Magazine. 2005;61.

28. Testing with C++. [date unknown; accessed 2018 Jan]. http://testanything.org/testing-with-tap/c-plus-plus.html.

29. Pajuelo I. Eclox. Ottawa, Canada: Eclipse Foundation, Inc.; 2018 [accessed 2018 July 10].https://marketplace.eclipse.org/content/eclox.

30. Lippmann RP, Fried DJ, Graf I, Haines JW, Kendall KR, McClung D, Weber D, Webster SE, Wyschogrod D, Cunningham RK, Zissman MA. Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation. In: Proceedings of the DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00; Vol. 2; 2000; Hilton Head, SC. p. 12–26.

31. Roesch M. Snort: lightweight intrusion detection for networks. In: Proceedings of the 13th System Administration Conference (LISA '99); Vol. 99; 1999 Nov 7–12; Seattle, WA. p. 229–238.

32. Meyering J, Eggert P. Sleep delay for a specified amount of time. 6.9 ed. Raleigh (NC): Red Hat, Inc.; 2017 Feb.

33. Smith S, Neyens S, Hammell R II. The use of entropy in lossy network traffic compression for network intrusion detection applications. In: ICMLG 2017 5th International Conference on Management Leadership and Governance; p. 352.

Appendix A. Sequence User's Manual Page

NAME

sequence − prints numbers in a sequence.

SYNOPSIS

sequence [**-1efhv**] [**-a** first] [**-d** difference] [**-l** limit] [**-r** ratio] [**-s** start] [**-t** type] arguments ...

DESCRIPTION

Sequence computes the numbers in a sequence based upon the type of sequence requested and the values of key constants like the first term, common difference, and common ratio. Each number in the sequence is printed on standard output.

OPTIONS

−1 −−oneline

By default sequence prints each of the numbers on the same line. This works well for feeding a shell **for** loop. This option will tell sequence to print each number on a different line. This works better for piping the output into a shell **while** loop.

−a −−first

Set the first term in the sequence usually the value of S when n equals one or S_0.

−c −−config

Set the name of a configuration file.

−d −−difference

Set the common difference usually expressed as the variable d.

−e −−enumerate

Enumerate the supported sequences.

−f −−falling

Decrement instead of increment n.

−h −−help

Print usage and exit.

−l −−limit

The maximum value of n.

−r −−ratio

Set the common ratio usually expressed as the variable r.

−s −−start

Set the initial value of n.

−t −−type

Set the type of sequence (e.g. arithmetic, geometric, etc.) See the DISCUSSION section for a full list and descriptions of the available sequences.

−v −−version

Print version and exits.

DEFINITIONS

The following variables are used in the equations to describe each type of sequence. **Note:** not all variables have meaning in all types of sequences. Check the sequence type definition to see if a particular variable applies.

first Represented by the variable a this is the first number in the sequence or S_1.

difference

Represented by the variable d this is the common difference.

ratio Represented by the variable r this is the common ratio.

number

Represented by the variable *n* this is the count of the elements in the sequence.

sequence

Represented by the variable *s* this is the result of the sequence such that s_n is the n^{th} number in the sequence.

DISCUSSION

This program implements the following sequences:

simple The simple sequence takes the starting value of *n* and increments it by 1 until the limit has been reached.

$$s_n = n$$

arithmetic

The arithmetic sequence is any sequence of numbers where there is a common difference between the terms. An arithmetic sequence as defined by the following equation:

$$s_n = a + d(n-1)$$

geometric

The geometric sequence is a sequence of numbers where there is a common ration between the terms. A geometric sequence as defined by the following equation:

$$s_n = ar^{n-1}$$

triangular

The Triangular sequence is a sequence of numbers where that number of object could make an equalateral triangle. This may be defined as sequence where *n* is multiplied by *n + 1* and the product is divided by *2*. This is illustrated by the following equation:

$$s_n = \frac{n(n+1)}{2}$$

square The Square sequence type implements a square sequence where *n* is raised to the second power. This is illustrated in the following equation:

$$s_n = n^2$$

cube The Cquare sequence type implements a cube sequence where *n* is raised to the third power. This is illustrated in the following equation:

$$s_n = n^3$$

fibonacci

The Fibonacci sequence types implements the Fibonacci numbers where the first Fibonacci numbers are *1*, and each following Fibonacci number is the sum of the two proceeding Fibonacci numbers. This is illustrated in the following equation:

$$s_n = s_{n-2} + s_{n-1}$$

ENVIRONMENT

Sequence recognizes the following environment variables:

SEQUENCE_DIFFERENCE

Set the common difference usually expressed as the variable *d*.

SEQUENCE_FALLING

If this envirnment variable is set to anything, sequence will decrement instead of increment *n*.

SEQUENCE_FIRST
Set the first term in the sequence usually the value of S when n equals zero or S_0.

SEQUENCE_LIMIT
The maximum value of n.

SEQUENCE_RATIO
Set the common ratio usually expressed as the variable r.

SEQUENCE_START
Set the initial value of n.

SEQUENCE_TYPE
Set the type of sequence (e.g. arithmetic, geometric, etc.) See the DEFINITIONS section for a full list and descriptions of the available sequences.

CONFIG FILE

Sequence reads the .sequencerc file in the user's home directory for configuration information. Configuration files may also be specified on the command line. Each configuration file is formatted such that the variables name is specified then one or more characters from the seperation string may appear followed by the value or the variable. The seperations string contains space, tab, equal, and colon. Each of the following lines will properly set the sequence type to "simple":

type = simple
type=simple
type simple
type:simple
type: simple

CAVEAT(S)

Sequence enforces the priority of the environment over the configuration file, and the priority of the command line over the environment. In order to do this, if a file is specified on the command line, it will process that file and then reread the environment and the command line. This also means that at most one configuration file may be specified on the command line. The configuration file option may appear multiples times on the command line, but only the last configruation file will be used.

RESTRICTIONS

For some sequences and for the intended application values of n less than zero are undefined and useless; therefore, n may not be less than zero. If the **falling** option is specified, it is not considered an error for n to be less than zero because having the program count down to zero and stop is legitimate behavior. If the **falling** options is not specificed, it is considered an overflow error.

EXAMPLE(S)

The following invocation demonstrates how sequence may be used with a shell for loop:

for multiplier **in** ` *sequence* −t simple −l 10`; **do**
tcpreplay-x $multiplier -r file.pcap -i em2
done

The following invocation demonstrates how sequence may be used with a shell while loop:

sequence −1 −t simple −l 10 |
while read *multiplier*; **do**
tcpreplay-x $multiplier -r file.pcap -i em2
done

FILES

The tilde character is often used to represent the home directory of the current user. In this section it will be used to represent the installation root for receivepcap and related programs.

~/.sequencerc
The default configuration file.

Appendix B. ConfigItem Programmer's Manual Page

NAME

ConfigItem − defines a configuration item.

SYNOPSIS

#include <**ConfigItem.h**>

Public Member Functions

ConfigItem()
virtual **~ConfigItem**()
std::string ***getName**()
void **setName**(const char *)
void **setName**(const std::string)
std::string ***getShortOption**()
void **setShortOption**(const char *)
void **setShortOption**(const std::string)
std::string ***getLongOption**()
void **setLongOption**(const char *)
void **setLongOption**(const std::string)
int **getHas_arg**()
void **setHas_arg**(int)
std::string ***getEnv**()
void **setEnv**(const char *)
void **setEnv**(const std::string)
std::string ***getDesc**()
void **setDesc**(const char *)
void **setDesc**(const std::string)
std::string ***getValue**()
void **setValue**(const char *)
void **setValue**(const std::string)

Private Attributes

std::string ***name**
std::string ***shortOption**
std::string ***longOption**
int **had_arg**
std::string ***environment**
std::string ***description**
std::string ***value**

DESCRIPTION

A Configuration consists of a list of configuration items. Configuration Items contain character strings for the name, shortOption, longOption, environment, description, and default value of the item. They also contain a value has_arg which indicates whether or not the item contains an argument. The class consists of getters and setters for each of these values.

Configuration Item Values

name Each CI has a unique name which is used to by the program to distinguish it. It is also the name that the CI takes in a configuration file. The name is set by the setName() and retrieved by the method getName().

shortOption

CIs may have a short option which is used by the command line. Short options are single letters and must be unique within a Configuration. They are specified on the command line by a single hyphen followed by the letter. The shortOption is set by setShortOption and retrieved by getShortOption.

longOption
CIs may have a long option which is used by the command line. Long options are single word. They must contain spaces, and they must be unique. They are specified on the command line by two hyphens followed by the word. The longOption is set by setLongOption and retrieved by getLongOption.

has_arg
CIs may have arguments. If a CI has no arguments has_arg should be set to 0. If the CI requires an argument then has_arg should be set to 1. If the CI has an optional argument then has_arg is set to 2. The has_arg is set by setHas_arg() and retrieved by getHas_arg().

environment
CIs may have an environment variable name which is used to pull the CI from the environment. The environment is set with setEvn() and retrieved with getEnv().

description
CIs may have a description. This is the text displayed in the usage message which informs the user what the CI does and how to set it from the command line. It is set with setDesc() and retrieved with getDesc().

value Each CI has a value. The default value is zero or NULL. A default value may be set by invoking setValue() before the loadConfig method from the Configuration is invoked. The value may be retrieved by calling getValue().

Configuration Item Methods

ConfigItem
This simple constructor creates a zero initialized ConfigItem.

˜ConfigItem
Destorys the CI freeing all of the dynamic memory it used.

getName
returns the name of the CI.

setName
set the name of the CI. This method comes in two version one taking a null terminated array of characters and the other taking string or string literal. Each creates a new string containing that value, and sets the name equal to a pointer to the new string. If the name is not NULL (i.e. it was already set to something else), the old string is deleted first. The value of the name of a configuration item maybe reset to zero by passing a NULL pointer to this method.

getShortOption
return a pointer to a string containing shortOption value.

setShortOption
set the value of the shortOption. This method comes in two version on taking a null terminated array of characters and the other taking string or string literal. Each creates a new string containing that value, and sets the shortOption value equal to a pointer to the new string. If the shortOption value is not NULL (i.e. it was already set to something else), the old string is deleted first. The value of the shortOption value maybe reset to zero by passing a NULL pointer to this method.

getLongOption
return a pointer to a string containing the longOption value.

setLongOption
set the value of the shortOption. This method comes in two version on taking a null terminated array of characters and the other taking string or string literal. Each creates a new string containing that value, and sets the longOption value equal to a pointer to the new string. If the longOption value is not NULL (i.e. it was already set to something else), the old string is deleted first. The value of the longOption value maybe reset to zero by passing a NULL pointer to this method.

getHas_arg
return the has_arg value.

setHas_arg
set the has_arg value.

getEvn return a pointer to a string containing the environment value.

setEvn set the environment value. This method comes in two version on taking a null terminated array of characters and the other taking string or string literal. Each creates a new string containing that value, and sets the environment value equal to a pointer to the new string. If the environment value is not NULL (i.e. it was already set to something else), the old string is deleted first. The value of the environment value maybe reset to zero by passing a NULL pointer to this method.

getDesc
return a pointer to a string containing the description value.

setDesc
set the description value. This method comes in two version on taking a null terminated array of characters and the other taking string or string literal. Each creates a new string containing that value, and sets the description value equal to a pointer to the new string. If the description value is not NULL (i.e. it was already set to something else), the old string is deleted first. The value of the description value maybe reset to zero by passing a NULL pointer to this method.

getValue
return a pointer to a string containing the value value.

setValue
set the default value. This method comes in two version on taking a null terminated array of characters and the other taking string or string literal. Each creates a new string containing that value, and sets the default value equal to a pointer to the new string. If the default value is not NULL (i.e. it was already set to something else), the old string is deleted first. The value of the default value maybe reset to zero by passing a NULL pointer to this method. It is expected that this default value will be replaced when the configuration is loaded.

AUTHOR

Sidney C. Smith
US Army Research Laboratory

SEE ALSO

getopt(3), **getopt_long**(3), **Configuration**(3)

Appendix C. Configuration Programmer's Manual Page

NAME

Configuration − class to contain, obtain, and query configuration items.

SYNOPSIS

#include <**cerrno**>
#include <**cstdio**>
#include <**exception**>
#include <**map**>

#include <**cstdlibexcept.h**>
#include <**ConfigItem.h**>
#include <**Configuration.h**>

Public Member Functions

virtual **˜Configuration** ()
std::string ***getConfig** (const char *)
const char ***getConfig (std::string *)**
void addConfig(ConfigItem *)
std::string *getArguments()
void setArguments(const char *)
void setArguments(std::string *)
std::string *getFileName()
void setFileName(char *)
void setFileName(std::string *)
FILE *getFile()
void setFile(FILE *)
std::string *getProgName()
void setProgName(const char *)
intgetOptindex()
void loadConfig(int, char **)
void printUsage(FILE *)

Static Public Member Functions

static Configuration ***getConfiguration**()

Protected Member Functions

void **loadConfigFile**()
void **loadConfigEnvironment**()
int **loadConfigCommandLine**(int, char**)

Private Member Functions

Configuration()

Private Attributes

std::string ***fn**
FILE ***fp**
itemMap_t **itemMap**
std::string ***progname**
std::string ***arguments**
int **optindex**

Static Private Attributes

static Configuration ***configuration**

DESCRIPTION

The *Configuration* object is used to gather into one place all of the outside configuration information about the program. This includes information specified in a configuration file, the environment, and the command line. A Configuration Item (CI) is created for each separate piece of information. The CI contains all of the information required to extract this information from the environment. These are then added to the Configuration with the addConfig() method. Once the Configuration object is configured and the CIs are added, the loadConfig() method is invoked, and then calls to the getConfig() method will a pointer to a C system string with the value of the CI.

The *Configuration* object contains the following data items. **Note:** all data items are private and may only be manipulated by the methods provided.

arguments

This variable is printed with the usage message and allows the programmer to specify how the system interprets arguments from the command line that are not considered options. Examples are "[files] ..." which means zero or more file names may be specified on the command line, and "from-file to-file" which means that two files must be provided. The option is set with setArguments() function and may be retrieved by invoking the getArguments() function.

fn This is the name of the configuration file. It is set by invoking the setFileName() method and retrieved by invoking the getFileName() method. The setFileName() method has the side effect of opening the file for reading and placing the FILE pointer into the fp data element.

fp This is the pointer to a C standard I/O FILE structure which is the configuration file opened for reading. The file pointer may be set indirectly by invoking setFileName() or directly by invoking setFile(). The file pointer for be retrieved by invoking the getFile() method.

optindex

After the loadConfig() method has been invoked, this contains the index into the argv array of the first non-option argument. It may be retrieved by invoking the getOptindex() method.

progname

This contains a pointer to the name of the program being configured. It may be set by invoking the setProgName() method. It may be retrieved by invoking the getProgName() method.

The printUsage() method may be invoked to print a usage message on the file pointer provided.

DISCUSSION

The format of the configuration file is very simple:

```
config item = value
```

The equal sign may be replaced by a space, tab, vertical bar, or colon. A more sophisticated algorithm could be implemented by a class inheriting Configuration and replacing the protected loadConfigFile() method.

CIs set in a command line overwrite CIs set in the configuration file. CIs set in the configuration file overwrite CIs set in environment.

EXAMPLE(S)

The following code implements the classic hello world program with a twist to illustrate the Configuration library.

```
#include
 <cerrno>
#include <cstdio>
#include <exception>
#include <map>
#include <string>
```

```
#include "cstdlibexcept.h"
#include "Configuration.h"

void
defineConfiguration() {

        Configuration *config = Configuration::getConfiguration();

        // Add -g --greeting option.

        ci = new ConfigItem();
        ci->setName( "greeting" );
        ci->setShortOption( "g" );
        ci->setLongOption( "greeting" );
        ci->setEnv( "HELLO_GREETING" );
        ci->setHas_arg( 1 );
        ci->setDesc( "set the greeting." );
        ci->setValue( "Hello" );
        config->addConfig( ci );

        // Add -w --who option.

        ci = new ConfigItem();
        ci->setName( "who" );
        ci->setShortOption( "w" );
        ci->setLongOption( "who" );
        ci->setEnv( "HELLO_WHO" );
        ci->setHas_arg( 1 );
        ci->setDesc( "set to whom we say hello." );
        ci->setValue( "World!" );
        config->addConfig( ci );
}

void
processConfiguration( int argc, char **argv ) {
        defineConfiguration();
        Configuration *config = Configuration::getConfiguration();
        config->setArguments( "[arguments] ...");
        char *progname = basename( argv[0] );
        config->setProgName( progname );
        char *home = getenv( "HOME" );
        char cfn[256];
        (void) sprintf( cfn, "%s/.%src", home, progname );
        try {
                config->setFileName( cfn );
        } catch ( CStdLibExcept *e ) {
                fprintf( stderr, "%s %s\n", "Warning", e->what() );
        }

        if (config->loadConfig( argc, argv ) < 0) {
                config->printUsage( stderr );
                return 0;
        }
        return config;
```

```
}

main( int argc, char *argv[] ) {
        Configuration *config = Configuration::getConfiguration();

        try {
                processConfiguration( argc, argv );
        } catch( std::exception *e ) {
                fprintf(                        stderr, "%s:  %s\n", argv[0], e->what() );
                return -1;
        } catch ( CStdLibExcept *e ) {
                fprintf( stderr, "Fatal %s\n", e->what() );
                return -1;
        }
        if (config == 0) return -1;
        const char *greeting = config->getConfig( "greeting" );
        const char *who = config->getConfig( "who" );
        printf( "%s %s", greeting, who );
        for (int        optindex = config->getOptindex(); optindex < argc; optindex++) {
                printf (" and %s", argv[optindex] );
        }
        printf( "\n" );
        return 0;
}
```

SEE ALSO

getopt(3), **getopt_long**(3), **configitem**(3)

DIAGNOSTICS

setFileName

Since setFileName has the side effect of opening the file for reading, it will throw a CStdLibExcept exception if something prevents it from successfully opening the file.

loadConfig

Upon successful completion loadConfig returns the index into argv[] of the first non-option argument. If there are errors processing the command line, loadConfig returns a negative number. It is necessary for loadConfig to allocate dynamic memory, if this fails it will throw a CStdLibExcept exception.

printUsage

If printUsage is passed a NULL file pointer, it will throw a CStdLibExcept exception.

AUTHOR(S)

Sidney C. Smith, US Army Research Laboratory

Appendix D. Sequence Programmer's Manual Page

NAME

Sequence − Implements a generic sequence generator.

SYNOPSIS

`#include <Sequence.h>`

Inherited by **Arithmetic**, **Cube**, **Fibonacci**, **Geometric**, **Square**, and **Triangular**.

Public Member Functions

long **nextN** ()
virtual **~Sequence** ()
virtual double **next** ()
virtual std::string **toString** ()
double **getA** ()
void **setA** (double)
long **getN** ()
void **setN** (long)
double **getD** ()
double **getR** ()
void **setR** (double)
void **setFalling** (bool)
bool **isFalling** ()
std::string **getMyType** ()

Static Public Functions

static **Sequence** * **createSequence** (std::string)
static std::string **getName** ()
static std::string **getDescription** ()
static void **printSequences** (FILE *)

Protected Member Functions

Sequence ()

Protected Attributes

double **a**
long **n**
double **d**
double **r**
bool **falling**
std::string **myType**
std::string **myDesc**
static const std::string **name** = 'simple'
static const std::string **description**

DESCRIPTION

The **Sequence** class is designed to allow the user to configure the first term, common difference, and common ratio of a sequence then call the next method to get the next number in the sequence. The **Sequence** class is a parent class for a family of sequence generators. It is the home for the shared variables used by each class and the getters and setters that manipulate them. Each child class would then over ride the toString and next methods to implement each of the sequences. Since the **Sequence** class is a parent class, the only function implemented is to increment the number by one. The **Sequence** class implements the factory pattern where child objects are registered with the parent and the static method **createSequence()** is passed a string sequence name and returns a pointer to the correct object.

Constructor & Destructor Documentation

Sequence::Sequence () [protected]

Creates a new object of the **Sequence** class with all of the variables set to zero.

Returns:
new **Sequence** object.

References a, d, description, falling, myDesc, myType, n, name, and r.

Referenced by createSequence().

Sequence::~Sequence () `[virtual]`
Destory a **Sequence** object.

Member Function Documentation

Sequence * Sequence::createSequence (std::string sequence) `[static]`
Implements the factory pattern to create **Sequence** objects. The factory pattern allows the user or main program to be ignorant of the current set of implemented sub classes. This method takes a string that it will match against known sub classes to provide the user the correct object.

Parameters:
sequence - the name of the desired sequence.

Returns:
a pointer to the newly created object.

Exceptions:
std::invalid_argument when passed the name of unimplemented sequence.
References Sequence().

double Sequence::getA ()
Provides read access to the protected variable *a* which represents the first term or n_1.

Returns:
return the value of the first term.

References *a*.

double Sequence::getD ()
Provides read access to the protected variable *d* which represents the common difference.

Returns:
the common difference.
References *d*.

std::string Sequence::getDescription () `[static]`
Provides read access to the protected variable *description* which contains a description of the sequence.

Returns:
the description of the sequence.
Reimplemented in **Arithmetic**, **Cube**, **Fibonacci**, **Geometric**, **Square**, and **Triangular**.

References *description*.

Referenced by printSequences().

std::string Sequence::getMyType ()
Provides read access to protected variable *myType* which is set equal to the static variable *name* in the constructor.

Returns:
sequence type string.

References myType.

long Sequence::getN ()
Provide read access to the protected variable *n* which is used to track the number of current term in the series.

Returns:
the current term number.

References *n*.

std::string Sequence::getName () `[static]`

Provide read access to the protected static variable *name*. **Returns:**
the name of the sequence.

Reimplemented in **Arithmetic**, **Cube**, **Fibonacci**, **Geometric**, **Square**, and **Triangular**.

References *name*.

Referenced by printSequences().

double Sequence::getR ()

Provide read addess to the protected variable *r which represents the common ratio.*

Returns:
the common ratio.

bool Sequence::isFalling ()

Query the falling value. If falling is set to true then *n* will decrement. If falling is set to false then *n* will increment.

Returns:
the value of falling

double Sequence::next () `[virtual]`

This methods performs the prescribed calculates to compute the next number in the given sequence. It has the side of effect of either incrementing or decrementing
n based upon the value of *falling*.

Returns:
the next number in the sequence.

Exceptions:
overflow_error - if n would overflow.

Reimplemented in **Arithmetic**, **Cube**, **Fibonacci**, **Geometric**, **Square**, and **Triangular**. References n, and nextN().

long Sequence::nextN () `[protected]`

Based upon the value of falling *nextN* will either increment or decrement n by 1.

Returns:
the next value of n.

If falling is true **nextN()** will decrement *n*. If falling is false **nextN()** will increment *n*.

return the new value of n.

References falling, and n.

Referenced by Triangular::next(), Square::next(), next(), Geometric::next(), Fibonacci::next(), Cube::next(), and Arithmetic::next().

void Sequence::printSequences (FILE * fp) `[static]`

Print a list of the supported sequences and descriptions on the given file pointer.

Parameters:
fp - C Standard Library file pointer.

Exceptions:
CStdLibExcept - if there is an error writing to the file pointer.

References Fibonacci::getDescription(), Cube::getDescription(), Square::getDescription(), Triangular::getDescription(), Geometric::getDescription(), Arithmetic::getDescription(), getDescription(), Fibonacci::getName(), Cube::getName(), Square::getName(), Triangular::getName(), Geometric::getName(), Arithmetic::getName(), and getName().

Referenced by main().

void Sequence::setA (double newA)
Provide write access to the protected variable *a* which represents the value of n_1.

Parameters:
newA - the new value of the first term.

References *a*.

void Sequence::setD (double newD)
Provide write access to the protected varaible *d* which represents the common difference.

Parameters:
newD - the new common difference.

References *d*.

void Sequence::setFalling (bool newFalling)
Provide write access to the protected variable *falling*. If falling is set to true then *n* will decrement. If falling is set to false then *n* will increment.

Parameters:
newFalling - the boolean value of falling.

References *falling*.

void Sequence::setN (long newN)
Provide write access to the variable *n* which is used to track the number of current term in the series.

Parameters:
newN the new term number.

Exceptions:
std::invalid_argument if n is negative.

References *n*.

void Sequence::setR (double newR)
Provide write access to the protected variable *r* which represents the common ratio.

Parameters:
newR - the new common ratio.

References *r*.

std::string Sequence::toString ()
Provide read access to the protected variable *description* which describes the sequence.

return a string describing sequence.

References *myDesc*.

Member Data Documentation

double Sequence::a `[protected]` **first term**
Referenced by Arithmetic::Arithmetic(), Cube::Cube(), Fibonacci::Fibonacci(), Geometric::Geometric(), getA(), Geometric::next(), Arithmetic::next(), Sequence(), setA(), Square::Square(), and Triangular::Triangular().

double Sequence::d `[protected]` **common difference**
Referenced by Arithmetic::Arithmetic(), Cube::Cube(), Fibonacci::Fibonacci(), Geometric::Geometric(), getD(), Arithmetic::next(), Sequence(), setD(), Square::Square(), and Triangular::Triangular().

const std::string Sequence::description `[static, protected]` **Initial value:**
'The identity sequence where S_n = n'
Description of the **Sequence**

Reimplemented in **Arithmetic**, **Cube**, **Fibonacci**, **Geometric**, **Square**, and **Triangular**.

Appendix E. Arithmetic Programmer's Manual Page

NAME

Arithmetic − Implements an arithmetic sequence generator.

SYNOPSIS

`#include <Arithmetic.h>`

Inherits **Sequence**.

Public Member Functions

virtual **~Arithmetic** ()
double **next** ()
static std::string **getName** ()
static std::string **getDescription** ()

Protected Static Attributes

static const std::string **name**
static const std::string **description**

DESCRIPTION

The **Arithmetic** class is designed to allow the user to configure the first term, common difference of a sequence then call the **next()** method to get the next number in the sequence. It implements the following equation where s_n is the n^{th} number in the sequence, a is the first term in the sequence, and d is the common difference.

$$s_n = a + d(n-1)$$

Constructor & Destructor Documentation

Arithmetic::Arithmetic ()

Construct an object in the **Arithmetic** class.

Returns:

a pointer to an object in the **Arithmetic** class.

References Sequence::a, Sequence::d, description, Sequence::myDesc, Sequence::myType, Sequence::n, name, and Sequence::r.

Arithmetic::~Arithmetic () `[virtual]`

Destroy an object in the **Arithmetic** class.

Member Function Documentation

std::string Arithmetic::getDescription () `[static]`

Provides read access to the protected *description* attribute.

Returns:

a std::string containing the description of the sequence.

Reimplemented from **Sequence**.

References description.

Referenced by Sequence::printSequences().

std::string Arithmetic::getName () `[static]`

Provides read access to the protected *name* attribute.

Returns:

a std::string containing the name of the sequence.

Reimplemented from **Sequence**.

References *name*.

Referenced by Sequence::printSequences().

double Arithmetic::next () `[virtual]`

Compute and return the next number in the arithmetic sequence defined by the first term, and common difference contained in the object.

Returns:

the next value in the arithmetic sequence.

Reimplemented from **Sequence**.

References Sequence::a, Sequence::d, Sequence::n, and Sequence::nextN().

Member Data Documentation

const std::string Arithmetic::description `[static, protected]`

Provides a place where a description of the sequence may be defined.

Reimplemented from **Sequence**.

Referenced by Arithmetic(), and getDescription().

const std::string Arithmetic::name = 'arithmetic' `[static, protected]`

Provides a place where the name of the sequence may be defined.

Reimplemented from **Sequence**.

Referenced by Arithmetic(), and getName().

DIAGNOSTICS

The *Arithmetic* sequence throws std::overflow_error if either *n* or the next number in the sequence would overflow the data types used to store them. To test for overflow first we compute the smallest value of *d* that could cause overflow. If *d* is smaller than this then there is no valid value of *n* that could cause overflow. If we didn't do this test first we could overflow computing maxn. Now it is safe to compute *maxn* and test if the current *n* is larger than *maxn*.

AUTHOR

Sidney C. Smith
US Army Research Laboratory

Appendix F. Cube Programmer's Manual Page

NAME

Cube − Implements the **Cube** sequence generator.

SYNOPSIS

`#include <Cube.h>`

Inherits **Sequence**

Public Member Functions

Cube ()

virtual **˜Cube** ()

double **next** ()

static std::string **getName** ()

static std::string **getDescription** ()

Protected Static Attributes

static const std::string **name**

static const std::string **description**

DESCRIPTION

The **Cube** class is designed to allow the user call the **next()** method to get the next number in the sequence. It implements the following equation where S_n is the n^{th} number in the sequence.

$$s_n = n^3$$

Constructor & Destructor Documentation

Cube::Cube ()

Cube - construct an object in the **Cube** class. Description of the **Sequence**

Returns:

a pointer to an object of the **Cube** class.

References Sequence::a, Sequence::d, description, Sequence::falling, Sequence::myDesc, Sequence::myType, Sequence::n, name, and Sequence::r.

Cube::˜Cube () `[virtual]`

Destroy an object of the **Cube** class.

Member Function Documentation

std::string Cube::getDescription () `[static]`

Provide read access to the *description* attribute

Returns:

a std::string containing the description of the sequence.

Reimplemented from **Sequence**.

References description.

Referenced by Sequence::printSequences().

std::string Cube::getName () `[static]`

Provide read access to the *name* attribute.

Returns:

a std::string containing the name of the sequence.

Reimplemented from **Sequence**.

References name.

Referenced by Sequence::printSequences().

double Cube::next () [virtual]

Compute the next value in the sequence.

Returns:

the next number in the sequence.

Reimplemented from **Sequence**.

References Sequence::n, and Sequence::nextN().

Member Data Documentation

const std::string Cube::description [static, protected]

The description of the sequence.

Reimplemented from **Sequence**.

Referenced by Cube(), and getDescription().

const std::string Cube::name [static, protected]

The name of the sequence.

Reimplemented from **Sequence**.

Referenced by Cube(), and getName().

DIAGNOSTICS

The *Cube* sequence throws std::overflow_error if either *n* or the next number in the sequence would overflow the data types used to store them.

AUTHOR

Sidney C. Smith
US Army Research Laboratory

Appendix G. Fibonacci Programmer's Manual Page

NAME

Fibonacci − Implements an **Fibonacci** sequence generator.

SYNOPSIS

`#include <Fibonacci.h>`

Inherits **Sequence**.

Public Member Functions

Fibonacci ()

virtual ~**Fibonacci** ()

double **next** ()

Static Public Member Functions

static std::string **getName** ()

static std::string **getDescription** ()

Protected Attributes

std::vector< double > **FibVector**

long **maxn**

Static Protected Attributes

static const std::string **name**

static const std::string **description**"

DESCRIPTION

The **Fibonacci** class is designed to implement the **Fibonacci** sequence where the first Fibonacci number, f_1, is 1 and the second Fibonacci number, f_2, is 1 and each following Fibonacci number is the sum of the two proceeding Fibonacci numbers.

$$f_n = f_{n-2} + f_{n-1}$$

Constructor & Destructor Documentation

Fibonacci::Fibonacci ()

Constructs an object in the **Fibonacci** class. Computing Fibonacci numbers in sequence is cheap and easy. Computing them recursively is very expensive. Using Binet's method isn't that computationally expensive; however, for a 64 bit double, it overflows at the 604th **Fibonacci** number. This is significantly lower than the 1476th Fibonacci number that a 64 bit double is able to store. The simplest way to solve this problem is to precompute all of the allowable Fibonacci numbers into a vector while at the same time computing the maximum value of n.

Returns:

a pointer to an object of the **Fibonacci** class.

References Sequence::a, Sequence::d, description, FibVector, maxn, Sequence::myDesc, Sequence::myType, Sequence::n, name, and Sequence::r.

Member Function Documentation

double Fibonacci::next () `[virtual]`

Return the next Fibonacci number.

Reimplemented from **Sequence**.

References FibVector, maxn, Sequence::n, and Sequence::nextN().

Member Data Documentation

const std::string Fibonacci::description `[static, protected]`

Description of the **Sequence**.

Reimplemented from **Sequence**.

Referenced by Fibonacci(), and getDescription().

std::vector<double> Fibonacci::FibVector `[protected]`
A vector of Fibonacci numbers.

Referenced by Fibonacci(), and next().

long Fibonacci::maxn `[protected]`
The largest allowable Fibonacci number.

Referenced by Fibonacci(), and next().

const std::string Fibonacci::name [static, protected]
"Name of the sequence.

Reimplemented from **Sequence**.

Referenced by Fibonacci(), and getName().

DIAGNOSTICS

The *Fibonacci* sequence throws std::overflow_error if either *n* or the next number in the sequence would overflow the data types used to store them.

AUTHOR

Sidney C. Smith
US Army Research Laboratory

Appendix H. Geometric Programmer's Manual Page

NAME

Geometric − Implements a geometric sequence.

SYNOPSIS

`#include <Geometric.h>`

Inherits **Sequence**.

Public Member Functions

Geometric ()

virtual **~Geometric** ()

double **next** ()

static std::string **getName** ()

static std::string **getDescription** ()

Protected Static Attributes

static const std::string **name**.

static const std::string **description**

Detailed Description

The **Geometric** class is designed to allow the user to configure the first term and common ratio of a sequence then call the **next()** method to get the next number in the sequence. It implements the following equation where s_n is the n^{th} number in the sequence, a is the first term in the sequence, and r is the common ratio.

$$s_n = ar^n$$

Constructor & Destructor Documentation

Geometric::Geometric ()

Constructs an object of the **Geometric** class.

Returns:

a pointer to an object of the **Geometric** class.

References Sequence::a, Sequence::d, description, Sequence::myDesc, Sequence::myType, Sequence::n, name, and Sequence::r.

Geometric::~Geometric () `[virtual]`

Destroy an object in the **Geometric** class.

Member Function Documentation

std::string Geometric::getDescription () `[static]`

Provides read access to the protected *description* attribute.

Returns:

a std::string containing the description of the sequence.

Reimplemented from **Sequence**.

References description.

Referenced by Sequence::printSequences().

std::string Geometric::getName () `[static]`

Provide read access to the protected *name* attribute.

Returns:

a std::string containing the name of the sequence.

Reimplemented from **Sequence**.

References name.

Referenced by Sequence::printSequences().

double Geometric::next () `[virtual]`

Compute next number in the sequence.

Returns:
the next number in a geometric sequence

Exceptions:
std::overflow_error if either *n* or the result would overflow.

Reimplemented from **Sequence**.

References Sequence::a, Sequence::n, Sequence::nextN(), and Sequence::r.

Member Data Documentation

const std::string Geometric::description `[static, protected]`

Name of the **Sequence**

Reimplemented from **Sequence**.

Referenced by Geometric(), and getDescription().

const std::string Geometric::name `[static, protected]`

Reimplemented from **Sequence**.

Referenced by Geometric(), and getName().

DIAGNOSTICS

The **next** function will throw std::overflow_error if either *n* or the next number in the sequence would overflow the data types used to store them. This method tests for overflow by taking the log base r of DBL_MAX divided by *a* to compute the maximum value of *n* that will not overflow the result as seen in the equation below. It compares the current value of n to maxn and throw an overflow_error if n is larger than maxn.

$$\max = \log_r\left(\frac{DBL\_MAX}{a}\right)$$

Since C++ does not have log function with an arbitrary base we will make use of the fact that log base r of x is equal to the log base y of x divided by the log base y of r where y can be anything as see below.XE

$$\log_{r(x)} = \frac{log(x)}{log(r)}$$

This allows us to compute *maxn* with the following equation:

$$maxn = \frac{\log\left(\frac{DBL\_MAX}{a}\right)}{log(r)} - 1$$

We reduce *n* by one because the log base 2 of *DBL_MAX* will produce an *n* that will overflow when 2 is raised to the n^{th} power.

AUTHOR

Sidney C. Smith
US Army Research Laboratory

Appendix I. Square Programmer's Manual Page

NAME

Square − Implements a square mathematical sequence generator.

SYNOPSIS

`#include <Square.h>`

Inherits **Sequence**.

Public Member Functions

Square () virtual **~Square** ()

double **next** ()

Static Public Member Functions

static std::string **getName** ()

static std::string **getDescription** ()

Static Protected Attributes

static const std::string **name**

static const std::string **description**

DESCRIPTION

The **Square** class is designed to allow the user to call the **next()** method to get the next number in the sequence. It implements the following equation where s_n is the n^{th} number in the sequence.

$$s_n = n^2$$

Constructor & Destructor Documentation

Square::Square ()

Construct an object in the **Square** class.

Returns:

a pointer to an object in the **Square** class.

References Sequence::a, Sequence::d, description, Sequence::falling, Sequence::myDesc, Sequence::myType, Sequence::n, name, and Sequence::r.

Square::~Square () `[virtual]`

~Square - destroy an object in the **Square** class.

Member Function Documentation

std::string Square::getDescription () `[static]`

Provides read access to the protected *description* attribute.

Returns:

a std:: string contain the description of the sequence.

Reimplemented from **Sequence**.

References description.

Referenced by Sequence::printSequences().

std::string Square::getName () `[static]`

Provide read access to the protected variable *name*.

Returns:

a std::string containing the name of the sequence.

Reimplemented from **Sequence**.

References name.

Referenced by Sequence::printSequences().

double Square::next () `[virtual]`

Compute the next number in the square sequence.

Returns:

the next number in the square sequence.

Reimplemented from **Sequence**.

References Sequence::n, and Sequence::nextN().

Member Data Documentation

const std::string Square::description

The description of the **Sequence**.

Reimplemented from **Sequence**.

Referenced by getDescription(), and Square().

const std::string Square::name

The name of the sequence.

Reimplemented from **Sequence**.

Referenced by getName(), and Square().

DIAGNOSTICS

The member function **next**() will throw a std::overflow_error exception if either *n* or the next number in the sequence would overflow the data types used to store them.

AUTHOR

Sidney C. Smith
US Army Research Laboratory

Appendix J. Triangular Programmer's Manual Page

NAME

Triangular − Implements a triangular mathematical sequence.

SYNOPSIS

`#include <Triangular.h>`

Inherits **Sequence**.

Public Member Functions

Triangular ()

virtual ~**Triangular** ()

double **next** ()

static std::string **getName** ()

static std::string **getDescription** ()

Protected Member Attributes

static const std::string **name**

static const std::string **description**

DESCRIPTION

The **Triangular** class implements a triangular mathematical sequence allowing allow the user to call the **next()** method to get the next number in the sequence. It implements the following equation where s_n is the n^{th} number in the sequence.

$$s_n = \frac{n(n+1)}{2}$$

Constructor & Destructor Documentation

Triangular::Triangular ()

Constructs an object of the **Triangular** class.

Returns:

a pointer to an object of the **Triangular** class.

References Sequence::a, Sequence::d, description, Sequence::myDesc, Sequence::myType, Sequence::n, name, and Sequence::r.

Triangular::~Triangular () `[virtual]`

Destroy an object of the **Triangular** class.

Definition at line 102 of file Triangular.cpp.

Member Function Documentation

std::string Triangular::getDescription () `[static]`

Provides read access to the protected *description* attribute.

Returns:

a std::string containing a description of the sequence.

Reimplemented from **Sequence**.

References description.

Referenced by Sequence::printSequences().

std::string Triangular::getName () `[static]`

Provide read addess to the *name* attribute.

Returns:

a std::string containing the name of the sequence.

Reimplemented from **Sequence**.

References name.

Referenced by Sequence::printSequences().

double Triangular::next () `[virtual]`

Compute the next number in the trianular mathematical sequence.

Returns: a double containing the next number in the sequence.

Reimplemented from **Sequence**.

References Sequence::n, and Sequence::nextN().

Member Data Documentation

const std::string Triangular::description `[static, protected]`

Name of the **Sequence**

Reimplemented from **Sequence**.

Referenced by getDescription(), and Triangular().

const std::string Triangular::name `[static, protected]`

The name of the sequence.

Reimplemented from **Sequence**.

Referenced by getName(), and Triangular().

DIAGNOSTICS

The **next**() function sequence throws std::overflow_error if either *n* or the next number in the sequence would overflow the data types used to store them.

AUTHOR

Sidney C. Smith
US Army Research Laboratory

Appendix K. CUTE Test Results

```
#beginning CUTE_CStdLibExceptTest 9

#starting CStdlibExcept_test::baseCStdlibExceptConstructorTest

#success CStdlibExcept_test::baseCStdlibExceptConstructorTest OK

#starting CStdlibExcept_test::halfCStdlibExceptConstructorTest

#success CStdlibExcept_test::halfCStdlibExceptConstructorTest OK

#starting CStdlibExcept_test::fullCStdlibExceptConstructorTest

#success CStdlibExcept_test::fullCStdlibExceptConstructorTest OK

#starting CStdlibExcept_test::getFileTest

#success CStdlibExcept_test::getFileTest OK

#starting CStdlibExcept_test::getLineTest

#success CStdlibExcept_test::getLineTest OK

#starting CStdlibExcept_test::getUsrmsgTest

#success CStdlibExcept_test::getUsrmsgTest OK

#starting CStdlibExcept_test::getErrorTest

#success CStdlibExcept_test::getErrorTest OK

#starting CStdlibExcept_test::getErrmsgTest

#success CStdlibExcept_test::getErrmsgTest OK
```

```
#starting CStdlibExcept_test::whatTest

#success CStdlibExcept_test::whatTest OK

#ending CUTE_CStdLibExceptTest

#beginning CUTE_ConfigItemTest 14

#starting ConfigItemTest

#success ConfigItemTest OK

#starting getNameTest

#success getNameTest OK

#starting getNameStringTest

#success getNameStringTest OK

#starting getShortOptionTest

#success getShortOptionTest OK

#starting getShortOptionStringTest

#success getShortOptionStringTest OK

#starting getLongOptionTest

#success getLongOptionTest OK

#starting getLongOptionStringTest
```

```
#success getLongOptionStringTest OK

#starting getHas_argTest

#success getHas_argTest OK

#starting getEnvironmentTest

#success getEnvironmentTest OK

#starting getEnvironmentStringTest

#success getEnvironmentStringTest OK

#starting getDescriptionTest

#success getDescriptionTest OK

#starting getDescriptionStringTest

#success getDescriptionStringTest OK

#starting getValueTest

#success getValueTest OK

#starting getValueStringTest

#success getValueStringTest OK

#ending CUTE_ConfigItemTest

#beginning CUTE_ConfigureTest 6

#starting getArgumentsTest
```

```
#success getArgumentsTest OK

#starting getArgumentsStringTest

#success getArgumentsStringTest OK

#starting getFileNameTest

#success getFileNameTest OK

#starting getFileNameStringTest

#success getFileNameStringTest OK

#starting getProgNameTest

#success getProgNameTest OK

#starting getProgNameStringTest

#success getProgNameStringTest OK

#ending CUTE_ConfigureTest

#beginning CUTE_SequenceTestSuite 12

#starting sequenceConstructorTest

#success sequenceConstructorTest OK

#starting sequenceSetATest

#success sequenceSetATest OK
```

```
#starting sequenceSetDTest

#success sequenceSetDTest OK

#starting sequenceSetNTest

#success sequenceSetNTest OK

#starting sequenceSetRTest

#success sequenceSetRTest OK

#starting sequenceSetFallingTest

#success sequenceSetFallingTest OK

#starting sequenceNextTest

#success sequenceNextTest OK

#starting sequenceFallingNextTest

#success sequenceFallingNextTest OK

#starting sequenceGetNameTest

#success sequenceGetNameTest OK

#starting sequenceGetDescriptionTest

#success sequenceGetDescriptionTest OK

#starting squencePrintSequencesTest

#success squencePrintSequencesTest OK
```

```
#starting createSequenceTest

#success createSequenceTest OK

#ending CUTE_SequenceTestSuite

#beginning CUTE_SequenceTests 7

#starting processConfigurationDefaultTest

#success processConfigurationDefaultTest OK

#starting processConfigurationShortOptionTest

#success processConfigurationShortOptionTest OK

#starting processConfigurationLongOptionTest

#success processConfigurationLongOptionTest OK

#starting processConfigurationOneOptionTest

#success processConfigurationOneOptionTest OK

#starting processConfigurationLogfileTest

#success processConfigurationLogfileTest OK

#starting processConfigurationEnvironmentTest

#success processConfigurationEnvironmentTest OK

#starting processConfigurationPrecedenceTest
```

```
#success processConfigurationPrecedenceTest OK

#ending CUTE_SequenceTests

#beginning CuteArithmeticTest 7

#starting arithmeticConstructorTest

#success arithmeticConstructorTest OK

#starting arithmeticNextAscendingTest

#success arithmeticNextAscendingTest OK

#starting arithmeticNextDescendingTest

#success arithmeticNextDescendingTest OK

#starting arithmeticNextNoverflowTest

#success arithmeticNextNoverflowTest OK

#starting arithmeticNextOverflowTest

#success arithmeticNextOverflowTest OK

#starting arithmeticNextNegativeDOverflowTest

#success arithmeticNextNegativeDOverflowTest OK

#starting arithmeticNextFallingTest

#success arithmeticNextFallingTest OK

#ending CuteArithmeticTest
```

```
#beginning CuteGeometricTest 4
#starting GeometricConstructorTest
#success GeometricConstructorTest OK
#starting GeometricNextTest
#success GeometricNextTest OK
#starting GeometricFallingNextTest
#success GeometricFallingNextTest OK
#starting GeometricNextOverflowTest
#success GeometricNextOverflowTest OK
#ending CuteGeometricTest
#beginning CuteTriangularTest 4
#starting TriangularConstructorTest
#success TriangularConstructorTest OK
#starting TriangularNextTest
#success TriangularNextTest OK
#starting TriangularFallingNextTest
#success TriangularFallingNextTest OK
```

```
#starting TriangularNextOverflowTest

#success TriangularNextOverflowTest OK

#ending CuteTriangularTest

#beginning CuteSquareTest 4

#starting SquareConstructorTest

#success SquareConstructorTest OK

#starting SquareNextAscendingTest

#success SquareNextAscendingTest OK

#starting SquareNextDescendingTest

#success SquareNextDescendingTest OK

#starting SquareNextOverflowTest

#success SquareNextOverflowTest OK

#ending CuteSquareTest

#beginning CuteCubeTest 4

#starting CubeConstructorTest

#success CubeConstructorTest OK

#starting CubeNextAscendingTest

#success CubeNextAscendingTest OK
```

```
#starting CubeNextDescendingTest

#success CubeNextDescendingTest OK

#starting CubeNextOverflowTest

#success CubeNextOverflowTest OK

#ending CuteCubeTest

#beginning CuteFibonacciTest 4

#starting FibonacciConstructorTest

#success FibonacciConstructorTest OK

#starting FibonacciNextAscendingTest

#success FibonacciNextAscendingTest OK

#starting FibonacciNextDescendingTest

#success FibonacciNextDescendingTest OK

#starting FibonacciNextOverflowTest

#success FibonacciNextOverflowTest OK

#ending CuteFibonacciTest
```

Appendix L. Raw Data from EE4D984HC2000R20

Table L-1: Raw Data from EE4D984HC2000R20

Speed	PLR	ALR
1x	4.34	0.00
2x	0.06	0.09
4x	0.35	0.00
8x	2.41	0.00
16x	3.97	0.00
32x	8.62	0.00
64x	10.16	0.00
128x	10.20	0.00
256x	10.52	0.00
512x	10.52	0.00
1024x	10.62	0.00
2048x	11.81	0.00
4096x	11.84	0.00
8192x	11.94	0.00
16384x	12.07	0.00
32768x	12.06	0.00
65536x	12.10	0.00
131072x	12.11	0.00
262144x	12.12	0.00
524288x	12.09	0.00
1048576x	29.23	19.61
2097152x	12.08	0.00
4194304x	22.10	16.66
8388608x	36.51	46.09
16777216x	33.84	30.85
33554432x	38.65	50.28
67108864x	12.29	0.47
134217728x	40.29	50.19
268435456x	39.77	48.85
536870912x	37.98	49.71
1073741824x	37.50	44.28

Continued on next page

Continued from previous page

Speed	PLR	ALR
2147483648x	12.07	0.00
4294967296x	12.08	0.00
8589934592x	12.08	0.00
17179869184x	12.41	0.09
34359738368x	12.12	0.00
68719476736x	13.60	1.80
137438953472x	35.67	41.61
274877906944x	35.73	48.85
549755813888x	36.33	43.23
2199023255552x	37.20	47.04
4398046511104x	37.79	43.33
8796093022208x	39.10	49.04
17592186044416x	39.28	49.61
35184372088832x	38.62	45.42
70368744177664x	38.18	47.52
140737488355328x	35.26	44.66
281474976710656x	27.75	38.47
562949953421312x	36.92	46.76
1125899906842624x	37.78	51.42
2251799813685248x	37.05	31.90
4503599627370496x	38.07	45.80
9007199254740992x	37.14	43.42
18014398509481984x	21.83	24.47
36028797018963968x	40.26	49.61
72057594037927936x	37.29	48.47
144115188075855872x	21.00	14.09
288230376151711744x	40.03	48.38
576460752303423488x	40.36	49.71

List of Symbols, Abbreviations, and Acronyms

ACRONYMS:

ALR : alert loss rate

ARL : US Army Research Laboratory

CLI : command line interface

DARPA : Defense Advanced Research Projects Agency

IDE : integrated development environment

PLR : packet loss rate

SDE : software development environment

MATHEMATICAL SYMBOLS:

a: the first number in a sequence

d: the common difference

DUB_MAX: the largest value that a variable of the type long may have in C++ before it overflows

$LONG\_MAX$: the largest value that a variable of the type long may have in C++ before it overflows

n: the position in the sequence

r: the common ratio

s_n: the n^{th} number in a sequence

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIR ARL IMAL HRA RECORDS MGMT RDRL DCL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
1 (PDF)	DIR USARL RDRL CIN S S SMITH