Gathering Software Metrics from Software Version Control Systems and Automated Build Systems

Mark Cheung, TJHSST 2010
TJHSST Computer Systems Lab, 2009-2010
Mentor: Michael Ihde
Northrop Grumman Information Systems

Abstract: The purpose of this project is to develop tools that support the automated production of software metrics from version control systems (such as CVS and Subversion) and build systems (such as CruiseControl). The project includes code metrics programs for X-Midas and Java that generate Logical and Physical Source Lines of Code. The project will include the development of tools that assist in generating reports and performing analysis of the results.

Keywords: SLOC, LSLOC, PSLOC, CruiseControl, automated, Subversion,

Introduction: Software metrics are tools that can be used to measure in quantity the software development and its specification. It is often used to estimate the cost and resource requirements (such as time), the productivity, the data collection, the code quality, and the performance. The metrics are either direct or indirect. The direct metric depends only on the attribute it is measuring while an indirect metric makes inferences based on a measure of other attributes.

Background:
Source Lines of Code (SLOC): Source Lines of Code is a code metrics for counting the number of lines for a set of programs and thereby estimating the amount of effort required. There are various types of counting that can be done, including total lines, non-blank lines, physical lines, logical lines, and tokens. SLOC measures are divided into two major types: physical and logical. Physical SLOC measures the code directly; it includes the blank lines, comments lines, and other logical and style conventions. In contrast, Logical SLOC accounts for code conventions such as the closing and ending braces of a for-loop (in this case it would be 1 Logical SLOC and 2 Physical SLOC). Consequently, SLOC for one language should not be used for another due to the syntactical differences among them.

Disadvantages:
Vary by language: Different programming language has different syntax, structures, and expressions. A simple example is the comment delimiter. C, C++, and Java, uses “//” to symbolize comment until the end of line while Python, Maple, and Perl uses “#” to represent so. Certain type names in languages are also different, especially strings: C uses “char[]” while Java, C# uses “String” to represent String type name. Expression of for loop in Java and C is something like “for(int counter=1; counter<=10; counter++),” but in x-Midas and Ruby, there
needs to be an “end” statement after the recursion. The point here is that different language has different productivity and quality. C++, Java, Visual Basic, and Python are credited with higher productivity, reliability, and simplicity than low-level languages such as C. On average, a line of python is as expressive as six lines of C. Programmers can save time and effort when they use a simpler language. Table 1 shows ratios of High-Level-Language statements to equivalent C Code. The higher the ratio, the more each line of code in the language accomplishes.

Besides syntax, software effort is also affected by the programmer’s familiarity with the language. The linguists Sapir and Whorf hypothesize a correlation between a language and the ability to think certain thoughts. According to Sapir and Whorf, in order to think a thought, you first need to understand the words for expressing it (Whorf, 1956, as cited in McConnel, 2004, p. 63). As McConnel explains, the same could be said about programming languages (p.63). The expressions of a programming languages influence the thought process. In summary, using a higher-level language can reduce the SLOC, but sometimes it may be more challenging to use higher-level language due to unfamiliarity and thus requires more effort.

**COnstructive COst Model (COCOMO’81):** COCOMO, first published by Dr. Barry Boehm in 1981, takes the Code Metrics and computes the size, effort (time), and cost of a program. It allows programmers to estimate the development schedule in Person-Months and plan accordingly. The model classified the classes and projects into three modes: organic, semidetached, and embedded. Organic projects are familiar projects coded by a small team with good experience. They are typically within stable development environment and consist of no more than a hundred thousand SLOCs (100 KSLOC) Embedded project are less comprehensible projects requires operating within “tight” constraints such as hardware and software limitations, regulations (deadline), or operational procedures. There may be high cost on changing parts and projects are usually no on well known fields. Semidetached mode is the intermediate stage of development between the embedded and organic projects. Developers have a reasonable understanding of the system and the team usually comprised of both experienced and inexperienced developers, or even developers specialized for some project aspects.

**Basic COCOMO’81:**

\[
\text{Effort Applied} = a_b(\text{KSLOC})^b \quad (\text{mm man-months})
\]

\[
\text{Development Time} = c_b(\text{Effort Applied})^d \quad (\text{months})
\]

\[
\text{People required} = \frac{\text{Effort Applied}}{\text{Development Time}} \quad \text{(count)}
\]

<table>
<thead>
<tr>
<th>Software project</th>
<th>(a_b)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organic</td>
<td>2.4</td>
<td>1.05</td>
<td>2.5</td>
<td>0.38</td>
</tr>
<tr>
<td>Semi-detached</td>
<td>3.0</td>
<td>1.12</td>
<td>2.5</td>
<td>0.35</td>
</tr>
<tr>
<td>Embedded</td>
<td>3.6</td>
<td>1.20</td>
<td>2.5</td>
<td>0.32</td>
</tr>
</tbody>
</table>

\[
\text{Effort Applied} \quad \text{ACT (Annual Change Traffic)} = \text{ACT} \times \text{Effort Applied}^d
\]
In a year, 40 KSLOC project that increased by 2 thousands lines with modified 3 thousands modified lines. ACT= (3000+2000 )/ 40000=0.125

**Intermediate Constructive Cost Model (Intermediate COCOMO’81):** Like Basic COCOMO’81, Intermediate COCOMO calculate the development effort as a function of the code metrics. However, it also considers a set of “cost drivers” concerning assessments of the software, hardware, developers’ attributes, and project attributes. Each driver has been estimated with the following grades: VL, L, N, H, N, H, VH, and EH. See table 1 in appendix for more detail.

<table>
<thead>
<tr>
<th>Software project</th>
<th>( a_a )</th>
<th>( b_b )</th>
<th>( c_c )</th>
<th>( d_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organic</td>
<td>3.2</td>
<td>1.05</td>
<td>2.5</td>
<td>0.38</td>
</tr>
<tr>
<td>Semi-detached</td>
<td>3.0</td>
<td>1.12</td>
<td>2.5</td>
<td>0.35</td>
</tr>
<tr>
<td>Embedded</td>
<td>2.8</td>
<td>1.20</td>
<td>2.5</td>
<td>0.32</td>
</tr>
</tbody>
</table>

\( C = \text{Effort Adjustment factor (multiplying values of cost drivers)} \) (see table 1 for a list of cost drivers)

\[
\text{Effort Applied} = a_a(\text{KSLOC})^b_b \cdot C \text{ (mm man-months)}
\]

**Example:**

Cost Drivers:

- Software reliability is low \( \rightarrow 0.88 \)
- Low database size \( \rightarrow 0.94 \)
- Very high execution time constraint \( \rightarrow 1.30 \)
- High Programmer capability \( \rightarrow 0.86 \)
- Other cost drivers assumed nominal \( \rightarrow 1 \)

\( C = 0.88 \times 0.94 \times 1.30 \times 0.86 \times 1 \approx 0.93 \)

(Wheeler SLOCCount)

**Constructive Cost Model detailed model:** ?

**Constructive Cost Model II:** COCOMO II is the ?

**Costar 7.0:** Costar is an extension of COCOMO that estimate a project’s duration, staffing levels, effort and cost that provide 20% within the actual results 70% of the time. In addition to providing COCOMO, Costar 7.0 takes into account 32 cost drivers. With Costar 7.0, the user can make exchanges with “what-if” analysis to find the optimal project plan. (more?)
**Eclipse Modeling Framework Project (EMF):** EMF is a modeling framework and code generation facility. Given the definition of the model in XMI (XML Metadata Interchange), EMF can create a working set of Java classes for it. The generated classes and methods are tagged in Javadoc comment with @generated. It is difficult to measure the effort required to generate EMF code because the XMI for the EMF may take 30 minutes to a few days, and the results may be hundreds or thousands of lines of code.

**Direct Measure:** execution speed, memory size, defects reported (bugs)

**Indirect Measure:** functionality, quality, complexity, efficiency, reliability and maintainability.

**Relevant/Similar Projects:** StatSVN/StatCVS, USC CodeCount, Sonar, Checkstyle 5.0, Hackystat

**StatSVN:** StatSVN is open source software based on StatCVS that retrieves information from Subversion repository (Eclipse) and generates data on the development status. At Northrop Grumman, this is hooked to the Hobgoblin Metrics Collection Server. StatSVN counts the number of files, average file size, code per directory, lines of code (LOC), and Churn per day. StatSVN also produces a repo heatmap, which is an applet that shows all files in a hierarchical manner (see below). The tags of the rectangles are lists of directories and the bigger the directories, the larger the rectangles. The color shows change in LOC ( LGPL).

StatSVN.

Below is an example output StatSVN using SCLC (Source Code Line Counter) for Northrop Grumman “P” Project.

→more explanation at home

<table>
<thead>
<tr>
<th>Lines</th>
<th>Blank</th>
<th>Cmnts</th>
<th>NCSL</th>
<th>AGSL</th>
<th>AESL</th>
</tr>
</thead>
<tbody>
<tr>
<td>44066</td>
<td>5564</td>
<td>1137</td>
<td>37365</td>
<td>0</td>
<td>822030</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>1100</td>
</tr>
<tr>
<td>188</td>
<td>23</td>
<td>33</td>
<td>132</td>
<td>0</td>
<td>1980</td>
</tr>
<tr>
<td>177632</td>
<td>16864</td>
<td>90787</td>
<td>70438</td>
<td>49619</td>
<td>422628</td>
</tr>
<tr>
<td>221936</td>
<td>22451</td>
<td>91957</td>
<td>107985</td>
<td>49619</td>
<td>1247738</td>
</tr>
</tbody>
</table>
USC CodeCount (Northrop Grumman Ultimate Code Line Accumulator Tool) is a C language toolset that produces software metrics with two possible Source Lines of Code (SLOC)—physical and logical SLOC. Physical SLOCs is the sum of the program’s source code including the commented lines while logical SLOC is the total number of statements statements that should be counted for less than its number of lines (e.g. “if” and “endif” are redundancy and should be counted as just one logical SLOC). The USC CodeCount supports several programming languages—C/C++, C#, Java, JavaScript, MUL, Pearl, SQL, and XML. It generates a report in .dat format that includes the total lines, total blank lines, total embedded comments, total compiler directives lines, total data declaration lines, total execution instructions lines of each file. It also reports the Physical and Logical SLOC and their ratio.

Sonar: Sonar is a code quality management platform that can collect, analyze, and report metrics on code. It calculates LOC, total classes, comments, duplications, and violations (a total of 169 rules). Projects at risk (due to duplications, violations, complexities, comments), can be easily detected with Sonar. Sonar currently supports Java and PL/SQL languages, but its algorithms are extensible to cover other languages. See below for a snapshot of a generated report.

Checkstyle 5.0: Checkstyle is development tool that makes automation of checking Java code to adhere to a coding standard. Its standard checks include AnnotationUseStyle, ModifierOrder, GenericWhitespace, BooleanExpressionComplexity, and many more.

Hackystat: Hackystat is an open source project for “collection, analysis, visualization, interpretation, annotation, and dissemination” of software metrics. It allows users to attach sensors to their code, and the sensor would collect the data about development and forward it to the repository called Hackystat SensorBase. Hackystat Sensorbase allows developers to share data in the same way as sharing journal entries on blog.

Project Components:
Software:

**Eclipse**: Eclipse is a multi-language IDE with various components (plugins) that allow for development of Java, C/C++, Python, Perl, web applications etc. It is soft coded; the plug-ins it employed provide all of its functionality including the runtime system. Eclipse open source has more than 60 different projects, which are organized into seven categories: enterprise development, embedded and device development, Rich Client Platform, Rich Internet Applications, Application Frameworks, Application Lifecycle Management (ALM), and Service Oriented Architecture (SOA). I am using Eclipse C/C++ Development Tooling (CDT), RCP/Plug-in Development Environment (PDE), and Rich Ajax Platform (RAP).

**Eclipse Rich Client Platform (Eclipse RCP)**: Eclipse RCP is an open source Java-based development platform composed of a minimal set of plug-ins for building a platform application. It is portable in that the components are Java based and widgets have native implementations. By using RCP, developers can make use of the existing codebase for speed purpose. It also allows developers to write in C++ the GUI development in Java. Each of the bundled features can easily be implemented Equinox OSGi standard bunding.

**Eclipse Rich Ajax Platform (Eclipse RAP)**: Eclipse RAP is very similar to Eclipse RCP. However, instead of Standard Widget Toolkit, it implements SWT API with RWT. This allows rendering of widgets on a web-enable application from a single code base and reuse of code and development tools.

Eclipse RCP and RAP Comparison:

---

Languages:

- **C** - The language of the modified USC CodeCount uses.
- **Java** –
- **Midas**-
- **Python**-

Methods:

1) Evaluate the Northrop Grumman Ultimate Code Line Accumulator Tool (based on USC CodeCount) (done)
2) Extend CodeCount language support as necessary with C: (i.e. X-MIDAS and NextMIDAS scripting languages, Java, Python) (done)
3) Enhance CodeCount as necessary to support counting of auto-generated source code (done)
3.5) compare StatSVN with Logical SLOC

4) Select/Develop tool to perform automatic production of metrics and store them in a database (Hackystat)

5) Develop single-source plugins for Eclipse RAP and Eclipse RCP with Java to support report generation and analysis of metrics database.

6) Extend metrics collection to include code-quality, code-reuse, code-churn data, build failures, etc.

Method (in more detail later):

Java:

- Build upon existing USC Java Codecount
- modified methods labeled with @generated NOT or methods without this tag will be ignored during regeneration.

```
extern void generatedcheck(char line[], int line_length, bool_type *generated)
```

X-Midas Macro Script:

Macro syntax

```c
#define DIRR_NAME_LIST \
  "* SUBROUTINE   "\n  "* PROCEDURE    "\n  "* STARTMACRO   "\n  "* ENDMACRO     "\n  "* PIPE ON      "\n  "* PIPE OFF     "\n  "* XPIPE ON     "\n  "* XPIPE OFF    "\n  "* local        "\n  "
#define CONTROL_STATEMENTS_LIST \
  "* ELSE         "\n  "* LOOP         "\n  "* BREAK        "\n  "* IF           "\n  "* GOTO         "\n  "* LABEL        "\n  "* CALL         "\n  "* RETURN       "\n  "* LOCAL        "\n  "* WHILE        "\n  "* ENDIF        "\n  "* ELSEIF       "\n  "
extern void Ampersand(char line[], int line_length, bool_type *found_ptr)
```

Results:

Logical +Physical SLOC Comparison:

| Total | Blank | Comments | Compiler Cont | Comm | Number | File SLOC |
Sd350 and Midastest result (Midas):

Number of files successfully accessed........................ 2 out of 2

Number of files with:
- Commands > 100 = 0
- Data Declarations > 100 = 0
- Percentage of Comments to SLOC < 60.0 % = 2 Ave. Percentage of Comments to Logical SLOC = 30.4

Total occurrences of these Midas Keywords:

<table>
<thead>
<tr>
<th>Compiler Directives</th>
<th>Data Keywords</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBROUTINE.........</td>
<td>ELSE............</td>
<td>1 CALC</td>
</tr>
<tr>
<td>PROCEDURE.........</td>
<td>LOOP............</td>
<td>0 CONSTANT</td>
</tr>
<tr>
<td>STARTMACRO.........</td>
<td>BREAK...........</td>
<td>0 FASTFILTER</td>
</tr>
<tr>
<td>ENDMACRO.........</td>
<td>IF..............</td>
<td>2 MARRAY</td>
</tr>
<tr>
<td>PIPE ON...........</td>
<td>GOTO............</td>
<td>0 MFIT</td>
</tr>
<tr>
<td>PIPE OFF...........</td>
<td>LABEL...........</td>
<td>0 RQFSHIFT</td>
</tr>
<tr>
<td>XPIPE ON...........</td>
<td>CALL............</td>
<td>0 P_START</td>
</tr>
<tr>
<td>XPIPE OFF..........</td>
<td>RETURN..........</td>
<td>0 SMOOTH</td>
</tr>
<tr>
<td>local...............</td>
<td>LOCAL...........</td>
<td>0 STATUS</td>
</tr>
<tr>
<td>WHILE.............</td>
<td>WAVEFORM</td>
<td></td>
</tr>
<tr>
<td>ENDIF............</td>
<td>2 WAVEFORM</td>
<td></td>
</tr>
<tr>
<td>ELSEIF...........</td>
<td>1 STATISTICS</td>
<td></td>
</tr>
<tr>
<td>XRTDISPLAY</td>
<td>XRTPLOT</td>
<td></td>
</tr>
<tr>
<td>XRTRASTER</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Java CodeCount : Northrop Grumman Project:

Number of files successfully accessed........................ 682 out of 682

Ratio of Physical to Logical SLOC............................ 1.53

Number of files with:
- Executable Instructions > 100 = 157
- Data Declarations > 100 = 7
- Percentage of Comments to SLOC < 60.0 % = 88 Ave. Percentage of Comments to Physical SLOC = 127.8

Total occurrences of these Java Keywords:

<table>
<thead>
<tr>
<th>Compiler Directives</th>
<th>Data Keywords</th>
<th>Executable Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>import.............</td>
<td>abstract......</td>
<td>5 goto.............</td>
</tr>
<tr>
<td>const...............</td>
<td>0</td>
<td>2628</td>
</tr>
</tbody>
</table>
Discussion:

Comparison between SCLC and CodeCount:

<table>
<thead>
<tr>
<th>SCLC (StatSVN)</th>
<th>CodeCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines Blank Cmnts NCSL AGSL AESL</td>
<td>Total Blank Comments Compiler Data Exec Number</td>
</tr>
<tr>
<td>=== ==== === === === ===========</td>
<td>File SLOC Lines Lines Whole Embedded Direct Decl Instr of Files Type</td>
</tr>
<tr>
<td>44066 5564 1137 37365 0 822030</td>
<td>17632 16803 89498 0 7632 12743 49650 682</td>
</tr>
<tr>
<td>HTML (451 files) CSS (2 files) shell (11 files) Java (682 files)</td>
<td>45776 Physical 70025 43620 114980 43620 Logical 17632 16803 89498 0 7632 4680 33464 682</td>
</tr>
<tr>
<td>177632 16864 90787 70438 49619 422628</td>
<td>114980 9882 61285 341 0 8381 35239 441</td>
</tr>
<tr>
<td>Java (682 files)</td>
<td>114980 9882 61285 341 0 2808 22003 441</td>
</tr>
<tr>
<td>221936 22451 91957 107985 49619 1247738</td>
<td>114980 9882 61285 341 0 2808 22003 441</td>
</tr>
<tr>
<td>***** TOTAL *****</td>
<td>42811 Logical</td>
</tr>
<tr>
<td>(1146 files)</td>
<td></td>
</tr>
</tbody>
</table>

EMF COCOMO: Demonstrate how failing to account for auto-generated code in EMF throws the COCOMO model off:

References


**Acknowledgements:**

Mr. Michael Ihde, my mentor, Northrop Grumman, Eclipse Community, BIRT-Exchange Community, TJHSST community, etc.
Appendix

Appendix (Cost Drivers)

<table>
<thead>
<tr>
<th>Language</th>
<th>Level Relative to C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>C++</td>
<td>2.5</td>
</tr>
<tr>
<td>Fortran 95</td>
<td>2</td>
</tr>
<tr>
<td>Java</td>
<td>2.5</td>
</tr>
<tr>
<td>Perl</td>
<td>6</td>
</tr>
<tr>
<td>Python</td>
<td>6</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>6</td>
</tr>
<tr>
<td>Microsoft Visual Basic</td>
<td>4.5</td>
</tr>
</tbody>
</table>


Table 2:

<table>
<thead>
<tr>
<th>Cost Drivers</th>
<th>Ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Very Low</td>
</tr>
<tr>
<td>RELY</td>
<td>Required software reliability</td>
</tr>
<tr>
<td>DATA</td>
<td>Database size</td>
</tr>
<tr>
<td>CPLX</td>
<td>Product complexity</td>
</tr>
<tr>
<td>TIME</td>
<td>Execution time constraint</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>STOR</td>
<td>Main storage constraint</td>
</tr>
<tr>
<td>VIRT</td>
<td>Virtual machine (HW and OS) volatility</td>
</tr>
<tr>
<td>TURN</td>
<td>Computer turnaround time</td>
</tr>
<tr>
<td>ACAP</td>
<td>Analyst capability</td>
</tr>
<tr>
<td>AEXP</td>
<td>Applications experience</td>
</tr>
<tr>
<td>PCAP</td>
<td>Programmer capability</td>
</tr>
<tr>
<td>VEXP</td>
<td>Virtual machine experience</td>
</tr>
<tr>
<td>LEXP</td>
<td>Programming language experience</td>
</tr>
<tr>
<td>MODP</td>
<td>Use of &quot;modern&quot; programming practices (e.g. structured programming)</td>
</tr>
<tr>
<td>TOOL</td>
<td>Use of software tools</td>
</tr>
</tbody>
</table>
## SCED Required development schedule

| SCED | 1.23 (75% of nominal) | 1.08 (85% of nominal) | 1.00 (nominal) | 1.04 (130% of nominal) | 1.10 (160% of nominal) |

### Source: Boehm

Appendix (Methods):

#### Summaries of Methods:

**extern void comment_processing**(bool_type *comment_flag, int line_length,
  char line[], int *comment_lines, int *e_comm_lines)

/*----------------------------------------------------------------*/
/*  Procedure will scan each line for the comment delimiter `!`. When code  */
/*  appears prior to comment delimiter, it indicates the comment is an   */
/*  embedded comment. Set 'comment_flag' to true for whole line comments. */
/*----------------------------------------------------------------*/

**extern void control_statements_processing**(bool_type *control_flag,
  int line_length, char line[], int *control_statements,
  target_tally_array_type local_control_tally,
  target_name_array_type control_names, int control_length,
  bool_type close_match, bool_type *found, int *line_loc, char exclude[],
  target_tally_array_type control_tally)

/*----------------------------------------------------------------*/
/*  Procedure will scan each line for the control statements, which are  */
/*  defined in the CONTROL_STATEMENTS_LIST. Declare SLOC to be a control  */
/*  statement and set control_flag accordingly.                      */
/*----------------------------------------------------------------*/

**extern void commands_processing**(char line[], int line_length, int *comm_lines)

/*----------------------------------------------------------------*/
/*  Procedure will scan each line for the commands, which are defined in */
/*  the CONTROL_STATEMENTS_LIST. Declare SLOC to be a control statement*/
/*  and increases comm_lines accordingly.                          */
/*----------------------------------------------------------------*/

**extern void compiler_directives_processing**(bool_type *directive_flag,
  int line_length, char line[], int *directive_lines,
  target_tally_array_type local_dirr_tally,
  target_name_array_type dirr_names, int dirr_length,
  bool_type close_match, bool_type *found, int *line_loc, char exclude[],
  target_tally_array_type dirr_tally)

/*----------------------------------------------------------------*/
/*  Procedure will scan each line for the compiler directives, which are*/
/*  defined in the DIRR_NAME_LIST. Declare SLOC to be a directive line */
/*  and set directive_flag accordingly.                            */
/*----------------------------------------------------------------*/
Appendix (files):

**Sd350:**

```
STARTMACRO/xpipe/setup\dec=0 N:nample
!
! macro for emulating SD350 spectrum analyzer
!
local SFREQ, CFREQ, FWIDTH, FS

if /c gt 0
    WAVEFORM/t1=1k/ctag=3:SFREQ _wave CF nsamples SIN
else dsdf
    WAVEFORM/t1=1k/ctag=3:SFREQ _wave SF nsamples SIN
endif
RQFSHIFT/rs/fft-1k/ctag(2:CFREQ, 3:FWIDTH)/nowarn/wb=3 _wave _waves 0e-3 ,, MFFT/rs/fftp=3 _waves _wavep 1k hann 0.0
STATUS/quiet _wavep ,, ,, FS
XRTPLOT/xs=3/mtag=X:SFREQ/xcnt=2/mark=3 _wavep(cl=/DEC*FS) 1e-10 .5 lo ,, 70
XRTRASTER/xs=2/mtag=X:CFREQ/lpb=2/stretch _wavep 0 0.25 lo mm 20
XRTDISPLAY/xs=4 SFREQ/"Sine Frequency" FWIDTH/"Filter Width"
!
ENDMACRO
```

**Macrotest:**

```
SMOOTH A: input A: output N: winlen
!
! smooths data file using rectangular filter
!
CALC/quiet P_START (1-winlen)/2
CONSTANT TEMPWORK REAL winlen 1/winlen 0
P_START 1 1
FASTFILTER input TEMPWORK output

STATISTICS file mean
IF mean GT 4.00
    MARRAY file file 0.5
ELSEIF mean LT 1.0
    MARRAY file file 2.0 ENDIF
ENDMACRO
```

**Appendix (Results):**