McCabe Complexity Metrics

Thomas McCabe defined a set of metrics to characterize the complexity of a software module's internal control flow logic. Glen Myers suggested a revision to the calculation in An Extension to the Cyclomatic Measure of Program Complexity, SIGPLAN Notices, Oct 1977.

**Cyclomatic Complexity (v(G))**
- Measures - The complexity of a module's decision structure.
- Calculation - Count of linearly independent paths through a function or set of functions. An Imagix 4D option controls whether the original McCabe calculation or the Myers revision is used.
- Use - Higher cyclomatic complexities correlate with greater testing and maintenance requirements. Complexities above 20 also correspond to higher error rates.
- Extensions - For collections of functions (classes, files and directories), the complexities of the individual functions they contain are used to determine the Total, Average and Maximum cyclomatic complexity. Decision Density is the cyclomatic complexity per line of source code.

**Essential Complexity (ev(G))**
- Measures - The degree to which a module contains unstructured constructs.
- Calculation - Cyclomatic complexity of a module's remaining decision structure, once it has been reduced by fusing primitive, well-structured regions. In Imagix 4D, an option controls whether return statements are treated as structured or unstructured constructs.
- Use - This metric measures the degree of structuredness and the quality of the code. It is used to predict maintenance effort and to help in the modularization process.
- Extensions - Essential Density is the ratio of essential complexity to the overall cyclomatic complexity, indicating what portion of the cyclomatic complexity is a result of the module's structure.

Further descriptions of these metrics, including examples and explanations of their implications for testing, are available in Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.

Woodward, Hennell and Hedley Knots Metric

Following McCabe's publication of the Cyclomatic Complexity metric, some in the field, such as Myers and Hansen, continued and refined McCabe's focus on token counts. Others, including Woodward et al, looked to measure the internal structure of a module. In 1979, the Knots metrics was published as A Measure of Control Flow Complexity in Program Text.

**Knots (k)**
- Measures - The complexity and unstructuredness of a module's control flow.
- Calculation - Count of the intersections among the control flow paths through a function.
- Use - In pointing out areas of higher unstructuredness and the resultant challenges in understanding and testing, higher knots values may suggest the need to reorder or restructure, rather than just repartition, a module.

Welker Maintainability Index

The combination of multiple metrics into a single indicator of relative maintainability of a body of source code was originally defined by Oman and Hagemeister. Empirical studies by others led to a series of variants, and calculation of the maintainability index evolved over several years. Welker's 1995 proposal with Oman has evolved slightly into today's most widely used definition.

**Maintainability Index (MI)**
- Measures - The relative maintainability of a body of source code.
- Calculation - Complex, experience-based composite that combines measurements of the McCabe cyclomatic complexity, Halstead volume, lines of code and the comment density of the code.
• Use - Higher maintenance index values correspond to lower maintenance efforts. Under the standard Welker definition, software having no maintenance burden has an index of 171. Some variants normalize this to 100.

Chidamber and Kemerer Object Oriented Metrics

As a way to characterize the "object-orientatedness" of a design, a suite of metrics was proposed by Shyam Chidamber and Chris Kemerer in their 1994 paper, *A Metrics Suite for Object-Oriented Design*. The six metrics are:

**Weighted Methods Per Class (WMC)**
- Measures - Some aspect of the scope of the methods making up a class.
- Calculation - Summation of the weighted methods of the class. Chidamber and Kemerer do not specify what weighting to apply to each method. Imagix 4D applies each module's Cyclomatic Complexity as the weight, and thus WMC becomes a measure of the complexity of the overall decision structure within the methods making up a class. Applying a unitary weight would result in WMC reflecting a simple method count for the class; in Imagix 4D, this count is available as the Number of Member Methods.
- Use - Higher WMC values correlate with increased development, testing and maintenance efforts. As a result of inheritance, the testing and maintenance efforts for the derived classes also increase as a result of higher WMC for a base class.

**Depth of Inheritance Tree (DIT)**
- Measures - The inheritance upon which a class was built.
- Calculation - Maximum of number of levels in each of the class’s inheritance paths.
- Use - While reuse potential goes up with the number of ancestors, so does design complexity, due to more methods and classes being involved. Studies have found that higher DIT counts correspond with greater error density and lower quality.

**Number of Children (NOC)**
- Measures - How widely a class is reused to build other classes.
- Calculation - Count of the classes that are directly derived from a specified class.
- Use - Larger NOC counts point to greater reuse of the class, and with it a need for increased testing. High NOCs may also flag a misuse of subclassing.

**Coupling Between Object Classes (CBO)**
- Measures - How dependent a class is on other classes.
- Calculation - Count of the external classes whose members are called, set, read or used as types by members of the current class. In Imagix 4D, options control whether inherited members are included as part of the current class when calculating coupling and whether uses from external classes is included along with uses into external classes.
- Use - Excessive coupling limits the availability of a class for reuse, and also results in greater testing and maintenance efforts.

**Response For a Class (RFC)**
- Measures - The overall complexity of the calling hierarchy of the methods making up a class.
- Calculation - Count of the methods of a class and the methods that they directly call.
- Use - Larger RFC counts correlate with increased testing requirements.

**Lack of Cohesion in Methods (LCOM)**
- Measures - How widely member variables are used for sharing data between member functions.
- Calculation - Count of the pairs of class methods that don’t access any of the same class variables reduced by the number of pairs that do. In Imagix 4D, an option controls whether inherited member variables are considered to be in the set of class variables.
- Alternatives - A second Imagix 4D option enables the use of alternative Cohesion calculations. Li and Henry proposed an alternative measure, re-defined by Hitz and Montazeri, that counts the number of disconnected method / variable groupings. A third measure, from Henderson-Sellers, indicates how widely a class’s member functions access
its member variables, without consideration of which specific functions and variables are involved.

- Use - A higher LCOM indicates lower cohesion. This correlates with weaker encapsulation, and is an indicator that the class is a candidate for disaggregation into subclasses.

**Halstead Complexity Metrics**

Halstead complexity metrics were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module to measure a program module’s complexity directly from source code. Among the earliest software metrics, they are strong indicators of code complexity. Because they are applied to code, they are most often used as a maintenance metric. There is evidence that Halstead measures are also useful during development, to assess code quality in computationally-dense applications. Because maintainability should be a concern during development, the Halstead measures should be considered for use during code development to follow complexity trends. Halstead measures were introduced in 1977 and have been used and experimented with extensively since that time. They are one of the oldest measures of program complexity.

**2 Number of Operators and Operands**

Halstead’s metrics is based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand.

Then is counted

- number of unique (distinct) operators (n1)
- number of unique (distinct) operands (n2)
- total number of operators (N1)
- total number of operands (N2).

The number of unique operators and operands (n1 and n2) as well as the total number of operators and operands (N1 and N2) are calculated by collecting the frequencies of each operator and operand token of the source program.

Other Halstead measures are derived from these four quantities with certain fixed formulas as described later.

The classification rules of CMT++ are determined so that frequent language constructs give intuitively sensible operator and operand counts.

**2.1 Operands**

Tokens of the following categories are all counted as **operands** by CMT++:

**IDENTIFIER** all identifiers that are not reserved words

**TYPENAME**

**TYPESPEC** (type specifiers) Reserved words that specify type: *bool, char, double, float, int, long, short, signed, unsigned, void*. This class also includes some compiler
specific nonstandard keywords.

**CONSTANT**
Character, numeric or string constants.

### 2.2 Operators

Tokens of the following categories are all counted as **operators** by CMT++ preprocessor directives (however, the tokens `asm` and `this` are counted as operands):

- **SCSPEC** (storage class specifiers) Reserved words that specify storage class: `auto`, `extern`, `inlin`, `register`, `static`, `typedef`, `virtual`, `mtuable`.

- **TYPE_QUAL** (type qualifiers) Reserved words that qualify type: `const`, `friend`, `volatile`.

Other reserved words of C++: `asm`, `break`, `case`, `class`, `continue`, `default`, `delete`, `do`, `else`, `enum`, `for`, `goto`, `if`, `new`, `operator`, `private`, `protected`, `public`, `return`, `sizeof`, `struct`, `switch`, `this`, `union`, `while`, `namespace`, `using`, `try`, `catch`, `throw`, `const_cast`, `static_cast`, `dynamic_cast`, `reinterpret_cast`, `typeid`, `template`, `explicit`, `true`, `false`, `typename`. This class also includes some compiler specific nonstandard keywords.

- **OPERATOR**

```
! != %= && | &= ( ) * *= += , - -- -= -> . ... / /= : :: < << <<= <= == >= >>= >>>= ? [
] ^ ^= { } | |= ~
```

The following control structures `case ...`, `for (...)`, `if (...)`, `switch (...)`, `while for (...)`, and `catch (...)` are treated in a special way. The colon and the parentheses are considered to be a part of the constructs. The case and the colon or the `for (...)`, `if (...)`, `switch (...)`, `while for (...)`, and `catch (...)` and the parentheses are counted together as one operator.

### 2.3 Other

**COMMENTS**
The comments delimited by `/*` and `*/` or `//` and newline do not belong to the set of C++ tokens but they are counted by CMT++.

### 3 Halstead’s Measure derived from the unique and total number of operands and operators

The number of unique operators and operands (`n1` and `n2`) as well as the total number of operators and operands (`N1` and `N2`) are calculated by collecting the frequencies of each operator and operand token of the source program.

All other Halstead’s measures are derived from these four quantities using the following set of formulas.
3.1 Program length (N)

The program length (N) is the sum of the total number of operators and operands in the program:

\[ N = N_1 + N_2 \]

3.2 Vocabulary size (n)

The vocabulary size (n) is the sum of the number of unique operators and operands:

\[ n = n_1 + n_2 \]

3.3 Program volume (V)

The program volume (V) is the information contents of the program, measured in mathematical bits. It is calculated as the program length times the 2-base logarithm of the vocabulary size (n):

\[ V = N \times \log_2(n) \]

Halstead's volume (V) describes the size of the implementation of an algorithm. The computation of V is based on the number of operations performed and operands handled in the algorithm. Therefore V is less sensitive to code layout than the lines-of-code measures.

The volume of a function should be at least 20 and at most 1000. The volume of a parameterless one-line function that is not empty; is about 20. A volume greater than 1000 tells that the function probably does too many things.

The volume of a file should be at least 100 and at most 8000. These limits are based on volumes measured for files whose LOCpro and y(G) are near their recommended limits. The limits of volume can be used for double-checking.

3.4 Difficulty level (D)

The difficulty level or error proneness (D) of the program is proportional to the number of unique operators in the program. D is also proportional to the ration between the total number of operands and the number of unique operands (i.e. if the same operands are used many times in the program, it is more prone to errors).

\[ D = \left( \frac{n_1}{2} \right) \times \left( \frac{N_2}{n_2} \right) \]

3.5 Program level (L)

The program level (L) is the inverse of the error proneness of the program. I.e. a low level program is more prone to errors than a high level program.

\[ L = \frac{1}{D} \]
3.6 Effort to implement (E)

The effort to implement (E) or understand a program is proportional to the volume and to the difficulty level of the program.

\[ E = V \times D \]

3.7 Time to implement (T)

The time to implement or understand a program (T) is proportional to the effort. Empirical experiments can be used for calibrating this quantity. Halstead has found that dividing the effort by 18 give an approximation for the time in seconds.

\[ T = \frac{E}{18} \]

3.8 Number of delivered bugs (B)

The number of delivered bugs (B) correlates with the overall complexity of the software. Halstead gives the following formula for B:

\[ B = \left( \frac{E^{(2/3)}}{3000} \right) \quad \text{** stands for "to the exponent"} \]

Halstead’s delivered bugs (B) is an estimate for the number of errors in the implementation. Delivered bugs in a file should be less than 2. Experiences have shown that, when programming with C or C++, a source file almost always contains more errors than B suggests. The number of defects tends to grow more rapidly than B.

When dynamic testing is concerned, the most important Halstead metric is the number of delivered bugs. The number of delivered bugs approximates the number of errors in a module. As a goal at least that many errors should be found from the module in its testing.

This is “Halstead’s Software Science”.

Example:

Take a program, for example this little bit of one.

```c
int f=1, n=7;
for (int i=1; i<=n; i+=1)
  f*=i;
```

An “operator” is a fixed symbol or reserved word in a language, and “operand” is everything else: variable names, function names, numeric constants, etc. Comments don’t count.
define n1 to be the number of unique operators, in the example there are 10:

```
int = , ; for ( <= += ) *=
```

define n2 to be the number of unique operands, in the example there are 5:

```
f 1 n 7 i
```

define N1 to be the number of operators, in the example there are 16:

```
int = , = ; for ( int = ; <= ; += ) *= ;
```

define N2 to be the number of operands, in the example there are 12:

```
f 1 n 7 i 1 i n i 1 f i
```

define N, the length of the program to be N1+N2 (example 28)

define V, the volume of the program, to be \( N \times \log_2(n1 + n2) \) (example 109.4)

define D, the difficulty of the program, to be \( (n1 \times N2) \div (n2 \times 2) \) (example 12)

define E, the effort required to write the program, to be \( D \times V \) (example 1312.8)

calculate B, the number of bugs you should expect in the program as \( E^{0.667\div3000} \) (example 0.04)

calculate T, the time it took to write the program as \( E\div18 \) seconds (example 73 seconds)

## Knot Count

For a programmer, to understand a given program, he typically draws arrows from the point of control transfer to its destination, helping him to create a mental picture of the program and the control transfer in it.

According to this metric the more interwined these arrows become the more complex the program. This notion is captured in the concept of a "Knot".
A knot is essentially the intersection of two such control transfer arrows. If each statement in the program is written one as separate line, this notion can be formalized as follows. A jump from line a to line b is represented by the pair (a, b).

Two jumps (a, b) and (p, q) give rise to a knot if either \( \min (a,b) < \min (p,q) < \max (a,b) \) and \( \max (p,q) > \max (a,b) \); or \( \min (a,b) < \max (p,q) < \max (a,b) \) and \( \min (p,q) < \min (a,b) \).

**What are Function Points? How are they Computed? Explain**

Function points are one of the most widely used measures of software size. The basis of function points is that the "functionality" of the system is; what the system performs, is the measure of the system size. In function points, the system functionally is calculated in terms of the number of function it implements, the number of inputs, the number of output etc.

Parameter that can be obtained after requirements analysis and that are independent of the specification (and implementation) language.

The original formulation for computing the function points uses the count of five different parameters, namely, external input types, and external output types, logical internal file type, external interface file types and external inquiry type. According to the function point approach these five parameters capture the entire functionality of a system.

However, two elements of the same type may differ in their complexity and hence should not contribute the same amount of the "functionality" of the system. To account for complexity, each parameter in its type is classified as simple, average, or complex. Each unique input (data or control) type that is given as input to application from outside is considered of external input type and is counted. The source of external input can be the user, or some other application, files.

An external input type is considered simple if it has a few data elements and affects only a few internal fields of the application. It is considered complex if it has many data items
and may have internal logical files that are needed for processing them. The complexity is average if it has many data items and many internal logical files are needed for processing them. The complexity is average if it is in between.

Similarly, each unique output that leaves the system boundary is counted as external output type. Reports or messages to the users or other applications are counted as external input types. The complexity criteria is similar to that of the external input type. For a report, if it contains a few columns it is considered simple, if it has multiple columns it is considered average, and if it contains complex structure of data and reference many files or production, it is considered complex.

Each application maintains information internally for performing its functions. Each logical group of data or control information that is generated, used and maintained by the application is counted as a logical internal file type. A logical internal file is simple if it contains a few record type, complex if it has many type, and average if it is in between.

Once the counts for all five different types are known for all three different complexity classes, the raw or unadjusted function point can be computed as a weighted sum as follows:

\[
UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} C_{ij}
\]

Where \( i \) reflects the row and \( j \) reflects the column in Table, \( w_{ij} \) is the entry in the \( i \)th row and \( j \)th column of the table (i.e. it represents the contribution of an element of the type \( i \) and complexity \( j \)); and \( C_{ij} \) is the count of the number of elements of type \( i \) that have been classified as having the complexity corresponding to column \( j \).
Once the UFP is obtained, it is adjusted for the environment complexity. For this 14 different characteristics of the system are given. These are data communications, distributed processing, performance objective, operation configuration load, transaction rate, on line data entry, end user efficiency, on-line update, complex processing logic, re-usability, installation ease, operational ease, multiple sites, and desire to facilitate change. The degree of influence of each of these factors is taken to be from 0 to 5.

<table>
<thead>
<tr>
<th>Function type</th>
<th>Simple</th>
<th>Average</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>External input</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td><strong>External output</strong></td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Logical internal file</td>
<td>7</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>External interface file</td>
<td>5</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>External inquiry</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

representing the six different levels: not present (0), insignificant influence (1), modern influence (2), average influence (3), significant influence (4), and strong influence (5). The 14 degrees of influence for the system are then summed, giving total N (N ranges from 0 to 14*5 = 70). This N is used to obtain a complexity adjustment factor (CAF) as follows:

\[
CAF = 0.65 + 0.01 \times N.
\]

With this equation, the value of CAF ranges between 0.65 and 1.35. The delivered function points (DFP) are simply computed by multiplying the UFP by CAF. That is, Delivered Function Points = CAF * Unadjusted Function Points.

**Differentiate Between Process, Project and Products**
A software process as mentioned earlier, specifies a method of development software. A software project, on the other hand is a development project in which a software process is used. And software products are the outcomes of a software project.

Each software development project starts with some needs and (hopefully) ends with some software that satisfies those needs. A software process specifies the abstract set of activities that should be performed to go from user needs to final product. The actual act of executing the activities for some specific user needs is a software project. And all the outputs that are produced while the activities are being executed are the products.

**Differentiate Between Error, Fault and Failure**

The term error is used in two different ways. It refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value. That is error refers to the difference between the actual output of software and the correct output.

Fault is a condition that causes a system to fail in performing its required function. A fault is the basic reason for software malfunction and is synonymous with the commonly used term bug. Failure is the inability of a system or component to perform a required function according to its specifications. A software failure occurs if the behavior of the software is different from the specified behavior.

**What are the Matrices, Measurements and Models of Project Management Process**

For effective project monitoring, the information coming from the development process to the management process should be objective and quantitative data about the project. Software matrices are quantifiable measures that could be used to measure different characteristics of a software system or the software development process.

All engineering disciplines have matrices (such as matrices for weight, density, wavelength, and temperature) to quantify various characteristics of their products. A number of matrices have been proposed to quantify things like the size, complexity, and reliability of a software product.

Matrices provide the scale for quantifying qualities; actual measurement must be performed on a given software system in order to use matrices for quantifying characteristics of the given software. The Measurement method must be objective and should produce the same result independent of the measurer. Values for some matrices can be directly measured; others might have to be deduced by other measurement.
If a metric is not measured directly, we call the metric indirect. Some factors, like many software quality parameters, cannot be measured directly either because there are no means to measure the metric directly, or because the final product whose metric is of interest still does not exist. Similarly, the reliability of a software cannot be measured directly, even though precise definition and matrices for reliability exist. It has to be estimated from other measurements that are possible.

For estimating, models are needed. A model is a relationship of the predicted variable with other variables that can be measured. That is, if there is some metric of interest, which cannot be measured directly, then we build models to estimate the metric value based on the value of some other metrics that we can measure.

The model may be determined based on empirical data or it may be analytic. It should be clear that metrics, measurements, and models go together. Metrics provide a quantification of some property, measurements provide the actual value for the metrics, and models are needed to get the value for the metrics that cannot be measured directly.