Eurographics Conference on Visualization (EuroVis) 2013
B. Preim, P. Rheingans, and H. Theisel
(Guest Editors)

Volume 32 (2013), Number 3

Complexity Plots

Jeyarajan Thiyagalingam, Simon Walton, Brian Duffy, Anne Trefethen, and Min Chen

University of Oxford

Abstract

In this paper, we present a novel visualization technique for assisting the observation and analysis of algorithmic complexity. In comparison with conventional line graphs, this new technique is not sensitive to the units of measurement, allowing multivariate data series of different physical qualities (e.g., time, space and energy) to be juxtaposed together conveniently and consistently. It supports multivariate visualization as well as uncertainty visualization. It enables users to focus on algorithm categorization by complexity classes, while reducing visual impact caused by constants and algorithmic components that are insignificant to complexity analysis. It provides an effective means for observing the algorithmic complexity of programs with a mixture of algorithms and black-box software through visualization. Through two case studies, we demonstrate the effectiveness of complexity plots in complexity analysis in research, education and application.

1. Introduction

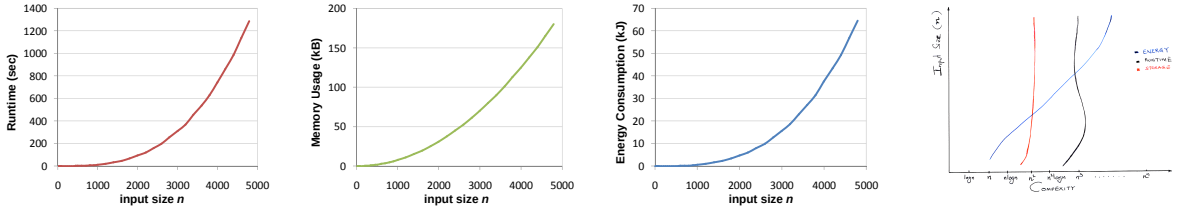
The history of the *line graph* can be traced back to at least a millennium ago [Fun36]. Between the 18th and 19th centuries, the introduction of the line graph by William Playfair, together with the bar chart, pie chart and circle graph (also by Playfair), scatter plot (by Francis Galton), and histogram (by Karl Pearson), led to the establishment of *statistics graphics* [Cle85]. The 18th century also saw the invention of graph papers [FD05], making line graphs an indispensable tool in almost every domain of study. Since then, many extensions and variations have been introduced.

In computing, one often visualizes the performance of an algorithm or a piece of software using line graphs. One of the most fundamental measures in computer science is *algorithmic complexity*, typically in terms of computation time or memory usage. As shown in Figure 1, for example, one can display the CPU runtime, maximum space requirement and energy usage of a piece of software as *dependent variables* on the y-axis against the input data size as an *independent variable* on the x-axis. For complexity analysis however, the line graphs exhibit three shortcomings:

- *Sensitivity to measurement units.* Although the three graphs in Figure 1*(a–c)* share the same independent variable, it is difficult to combine them into a single plot without the risk of misinterpreting the scaling of each dependent variable. Complexity is in fact *unitless*. It would be desirable for us to observe the complexities of time, space and energy in the same plot.
- *Non-uniform information loss.* The normal line graph is optimized for conveying information about a linear algorithm [CJ10]. Visualizing the dependent variables of algorithms in higher complexity classes often results in rapidly ascending lines, making observation and analysis difficult. This is especially true for a mixture of data from different complexity classes. Although using a logarithmic axis can alleviate some problems, it does not address the issue fundamentally, as illustrated in Figure 2. It is highly desirable to be able to juxtapose the performance of algorithms in different complexity classes.
- *Lack of means for assisting in complexity observation.* Many data processing programs are composed of a number of algorithms and sub-algorithms. Many software systems may appear as black-boxes with unknown algorithms inside. With line graphs, one may display a data series in conjunction with a few hypothesized complexity curves as seen in Figure 2. However, as the data series of a black-box can be affected by algorithmic components of lower complexity (including the constant offset), it is often challenging to correlate such a data series with any particular complexity class. Hence, it is desirable to support the complexity analysis of such 'hard-to-categorize' programs or black-box software through visualization.

In this paper, we propose a new form of line graph for supporting complexity analysis. We refer to this new visual representation as the **complexity plot**.

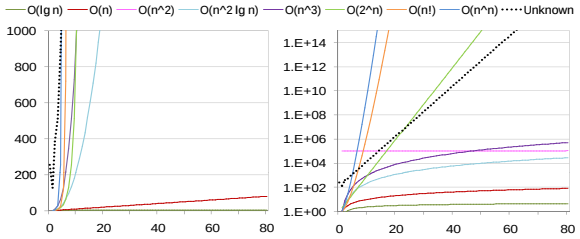
Our main contributions are:



(a) Runtime (b) Memory (c) Energy (d) Proposed

Figure 1: Visualizing the performance of an algorithm against different input sizes using conventional line graphs, including *(a)* runtime, *(b)* memory usage and *(c)* energy measurements. *(d)* Shows a design sketch of a complexity plot with complexity classes as the x-axis, and input size n as the y-axis.

- We have formulated a spatial mapping from the conventional line graph to the complexity plot, and present a practical solution for its implementation (Section 3).
- We have examined a number of design options for multivariate visualization and uncertainty visualization using complexity plots (Section 4).
- We have conducted two case studies (Section 5). The first focuses on the analysis of a collection of algorithms commonly taught in computer science, demonstrating the effectiveness of complexity plots in observing and comparing the performance of different algorithms including black-box programs. The second focuses on energy usage in computing, demonstrating the potential application of this form of visualization in an emerging topic of research analyzing the environmental impact of computation and communication.



(a) Linear (b) Logarithmic

Figure 2: A set of algorithmic measures are shown in a linear line graph *(a)* and a logarithmic plot *(b)*. The complexity of the 9th algorithm is unknown (black dotted line) and is difficult to estimate in either *(a)* or *(b)*. As the $O(n^2)$ (pink line) has a very large constant component, it is not shown in *(a)*, and its complexity cannot be observed in *(b)* easily.

2. Related Work

2.1. Complexity Analysis

In computer science, complexity analysis is concerned with the theoretical estimation of an algorithm's resources. Although the exact resource requirements such as runtime and storage may vary between platforms, the aim is to express the resource requirements as a function of a *single variable* – namely the input size n, capturing the *order of growth*, in particular for large values of n ($n \to \infty$). Such an order is commonly denoted by the Big O notation, which was inspired by the Landau notation [Bac94] used for analyzing the limiting behavior of functions of single variables.

Complexity analysis is mostly accomplished by manual study of the algorithms concerned – typically for average, best and worst behavior cases [CSRL01, MB05, Knu97]. Methods for automated estimation of complexity fall into two main categories: source-code analysis (e.g., [LM88]) or fitting curves to profile data (e.g., [GAW07]). Our work uses the latter approach to support visualization, focusing on the visual analysis of complexity, and it differs from a large collection of work on algorithm animation (e.g., [Sta90]), and schematic visual languages for algorithm visualization (e.g., [PSB92]).

2.2. Statistics Graphics – Line Graphs

During the past two to three centuries, many extensions to and variations of the basic line graph were developed. To address space scalability issues, dynamic approaches to querying time series data were developed [HS04, ARH12]. Javed *et al.* [JME10] studied several techniques for comparing data series including line graphs [Pla86], small multiple graphs [Tuf01], stacked graphs [BW08], horizon graphs [SMY*05] and braided graphs [JME10].

More abstract visual representations have been proposed to handle multivariate data, Harve *et al.* [HHWL02] for example, developed the ThemeRiver plot. Weber *et al.* [WAM01] presented the spiral graph, which supports better identification of periodic patterns in time series data. An extensive review of spatio-temporal visualization was given by Andrienko *et al.* [AAG03]. Borgo *et al.* [BPC*10] conducted a study on the impact of different tasks when using pixel-based visualization for time-varying data. Harris [Har99] described a full range of charts, graphs, maps, diagrams, and tables used daily to manage, analyze, and communicate information. Tufte [Tuf01] described a wide range of visualization techniques for time-varying data throughout the ages, from phase diagrams to cyclograms. Aigner *et*

al. [AMST11] gave a comprehensive guide to the current state of the art in visualization of time-oriented data.

3. Spatial Mapping

Let $\mathscr{A} : \mathbb{N}^+ \to \mathbb{R}$ be a function for measuring a dynamic property of an algorithm in relation to an input size $n \in \mathbb{N}^+$, where $\mathbb{N}^+$ is the set of all positive integers. There are many such properties that one may wish to measure: for example, CPU time, wall time, maximum memory usage, output size, energy usage, number of page faults and so on. As a general definition, we consider each measured quantity as a real value $v \in \mathbb{R}$, though in many cases it is usually a non-negative real value.

Although one can reason about the complexity of some properties of an algorithm by examining the corresponding implementation, the approach is applicable to simple algorithms. For a complicated algorithm or a piece of 'black box' software, and for some platform-dependent properties, one has to estimate the complexity by observing the data captured at runtime. Given a series of measured values of $\mathscr{A}$: $V = \{v_1, v_2, \ldots, v_i, \ldots, v_m\}$, one typically plots the values using a line graph as exemplified in Figure 1*(a–c)*. In order to achieve the design of a complexity plot as sketched out in Figure 1*(d)*, we need to transform the series $V = \{v_1, v_2, \ldots, v_m\}$ to a new series $C = \{c_1, c_2, \ldots, c_m\}$.

Note that we change the direction of plotting from aligning the independent variable $i = 1, 2, \ldots, m$, along the x-axis in a line graph to the y-axis in a complexity plot. Meanwhile, the data values of the dependent variable C are mapped to the x-axis instead of y. This ordering of axes metaphorically suggests the *order of growth* as the definition of complexity, though there is no fundamental reason for preventing one from swapping the two axes.

In the following subsections we outline the geometric transformation from a line graph to a complexity plot, discuss the accuracy of complexity estimators, the choice of *signature baselines* and the effect of estimation window size.

3.1. Spatial Transformation

Let $\phi : \mathbb{R}^k \to \mathbb{R}$ be a theoretically-ideal function that is able to determine the complexity of a given data series $V = \{v_1, v_2, \ldots, v_m\}$ representing the measurements of an algorithm taken for different input sizes that increase linearly at regular intervals. ϕ returns a real value that correctly determines the order of growth. For example, consider eight different algorithms in complexity classes $\Theta(\lg n)$, $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^2 \lg n)$, $\Theta(n^3)$, $\Theta(2^n)$, $\Theta(n!)$, and $\Theta(n^n)$ respectively. $V_1, V_2, \ldots, V_8$ are the measures of a dynamic property of these algorithms. We have $\phi(V_1) < \phi(V_2) < \ldots < \phi(V_8)$.

We can thus establish a spatial reference for these eight complexity classes; for instance, s =10, 20, 30, 40, 50, 60, 70, 80 along the x-axis as illustrated in Figure 1*(d)*. We call s_i the *spatial signature* of complexity class i. Since function ϕ maintains the correct order of these complexity classes, we can spatially transform $\phi(V_1) < \phi(V_2) < \ldots < \phi(V_8)$ to $s = 10, 20, 30, 40, 50, 60, 70, 80$ by using seven piecewise linear mappings, such that $\phi(V_i) \to s_i$.

Given a data series V of unknown complexity, we can apply ϕ to V. As long as $\phi(V_1) \leq \phi(V) \leq \phi(V_8)$, we can determine a pair of $\phi(V_i), \phi(V_{i+1})$ and compute the spatial mapping of V on the x-axis as:

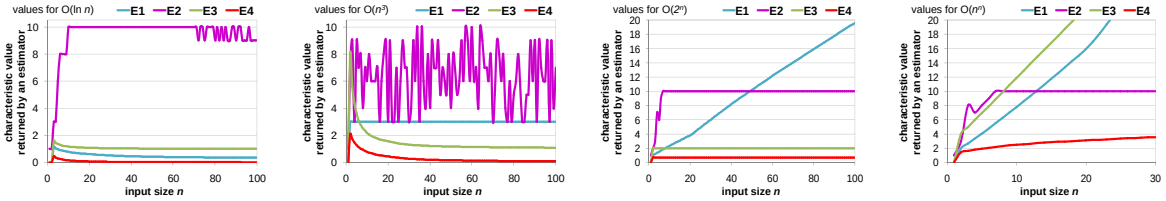
$$x = (s_{i+1} - s_i) \frac{\phi(V) - \phi(V_i)}{\phi(V_{i+1}) - \phi(V_i)} + s_i$$

In practice, such a theoretically-ideal function is yet to be found or may not exist at all. We hence have to use an estimator $\hat{\phi}$ or sometimes a few estimators. Similar to the ideal function, each estimator $\hat{\phi}$ takes a data series and returns a real value. Given a set of known complexity classes to be used as signatures on the x-axis, we can simply compute a data series for each class, such as $1, 4, 9, 16, \ldots, \beta^2$ for $\Theta(n^2)$. This data series can then be used to invoke $\hat{\phi}$. In order for $\hat{\phi}$ to determine the order of these complexity classes correctly, β cannot be too small. If the β is too small, the correct ordering of a complexity class relative to another cannot be guaranteed for all values. Hence the selection of β, which is referred to as a *signature baseline*, affects the correctness of the plot as well as spatial interpolation (see Section 3.3 for further details). In addition, given an unknown data series V with m values, one may wish to apply $\hat{\phi}$ to a subset of V with η consecutive values. η is referred to as the *window size* of $\hat{\phi}$. Note that η does not affect the application of $\hat{\phi}$ to the signature classes.

3.2. Estimators $\hat{\phi}$

Given m measurements $V = \{v_1, v_2, \ldots, v_m\}$ against m incremental input sizes at regular intervals, we can use a regression or curve fitting technique to find a function $f(i)$ such as $f(i) \approx v_i$. Some parametric property of $f(i)$ may be used to characterize the *order of growth* of $f(i)$. The use of this regression or curve fitting technique and the return of the parametric property gives an estimator $\hat{\phi}$.

There are many regression and curve fitting techniques in the literature (e.g., [PTVF07]). For this work, we give special attention to two groups of complexity classes, $O(n^k)$ and $O(k^n)$. The former would cover a large spectrum of functions $f(n)$ in the form of $n, n^2, n^3, \ldots, n^n$, while the latter covers $f(n)$ in the form of $1, 2^n, 3^n, \ldots, n^n$. We consider four different estimators – two for each group. An estimator is normally expected to be more accurate for functions that are close to its underlying model. Nevertheless, these estimators can be applied to data series in other complexity classes. It would be ideal to have an estimator with an underlying model matching with the complexity class of the data series. In practice, it is not possible to know the complexity class of



(a) $O(\ln n)$ Algorithm (b) $O(n^3)$ Algorithm (c) $O(2^n)$ Algorithm (d) $O(n^n)$ Algorithm

Figure 3: The convergence rate of four main estimators on four different algorithms. The x-axis shows the problem (input) size, and the y-axis shows the output values returned by different estimators.

every data series beforehand, nor to have an infinite number of estimators to accommodate every underlying model.

3.2.1. Estimators with Underlying Model n^k

Power Estimator (E1): assumes the following power function relates a dependent variable $\mathbf{v}$ to an independent variable $\mathbf{i}$, where A is a constant:

$$\mathbf{v} = A\mathbf{i}^k \tag{1}$$

The logarithmic form of the function is:

$$\ln(\mathbf{v}) = k\ln(\mathbf{i}) + \ln(A) \tag{2}$$

We can estimate k using linear regression:

$$\sum_{i=1}^{m} \| \ln(v_i) - k\ln(i) \|^2$$

Applying least squares fitting over the entire m measurements, we obtain k as:

$$k = \frac{m\sum_i(\ln i \ln v_i) - \sum_i(\ln i)\sum_i(\ln v_i)}{m\sum_i(\ln i)^2 - (\sum_i \ln i)^2} \tag{3}$$

The estimator (E1) returns k as the characteristic property, such that $\hat{\phi}_1(V) = k$.

Polynomial Estimator (E2): assumes that dependent variable $\mathbf{v}$ is a polynomial function of the independent variable $\mathbf{i}$ where ε is the unobserved random error:

$$\mathbf{v} = a_q\mathbf{i}^q + a_{q-1}\mathbf{i}^{q-1} + \cdots + a_0 + \varepsilon \tag{4}$$

Since complexity is asymptotic, the highest order (assuming a positive coefficient) is sufficient to determine a suitable complexity class. However, one of the major drawbacks of polynomial regression is the input order q has to be specified in advance. This limitation can be overcome by exhaustively evaluating Equation 4 for $q = 1 \ldots Q$ (for some $Q \in \mathbb{N}^+$) such that the overall sum of squares of residual error is minimized. In our implementation, $Q = 10$.

3.2.2. Estimators with Underlying Model k^n

Base Estimator (E3): assumes the underlying model $\mathbf{v} = Ak^{\mathbf{i}}$. The logarithmic form of the model is:

$$\ln(\mathbf{v}) = \ln(k)\mathbf{i} + \ln(A) \tag{5}$$

where A is a constant. Applying linear regression, we estimate the base k of the power function and return it as the output of the estimator, such that $\hat{\phi}_3(V) = k$.

Exponential Estimator (E4): assumes an exponential power-based model $\mathbf{v} = Ae^{B\mathbf{i}}$. Its logarithmic form is:

$$\ln(\mathbf{v}) = B\mathbf{i} + \ln(A) \tag{6}$$

Applying linear regression, we compute an optimal value for B, which becomes the output of the estimator $\hat{\phi}_4(V)$. Although e^B in Equation 6 is the same as k in Equation 5, different optimization processes may yield different results. We have hence experimented with both E3 and E4.

Figure 3 shows a comparison of these estimators using ideal functions (e.g., $f(n) = n^2$ for $O(n^2)$) in several known complexity classes. Since the polynomial estimator relies on minimum residual errors, the outputs varied rather considerably. In most cases, all other estimators outperformed the polynomial estimator. For this reason, we limit our studies to the three best-performing estimators.

3.3. Choice of Signature Baseline β

Users can select an arbitrary set of complexity classes to form the x-axis according to their needs and the expected complexity ranges of the data series. Let $O(f(n))$ and $O(g(n))$ be two different complexity classes. Their spatial signatures on the x-axis are s_f and s_g respectively. The special signatures are similar to the tick positions in a conventional line graph, and one can extend a set of vertical grid lines along the x-axis to represent them. Firstly, we must ensure that $s_f > s_g$ if $O(f(n))$ is more complex than $O(g(n))$, and vice versa. Secondly, we must ensure that for any estimator $\hat{\phi}$ used for this complexity plot, it will return the characteristic properties (e.g., k for E1 and B for E4) for $f(n)$ and $g(n)$ in the same order. In other words, if V_f and V_g are the two representative data series for $f(n)$ and $g(n)$ respectively, we must have $\hat{\phi}(V_f) > \hat{\phi}(V_g)$ when $O(f(n))$ is more complex than $O(g(n))$.

Given a complexity class $O(f(n))$, we will use its ideal function $f(n)$ to generate its representative data series. For example, $f(n) = n\ln n$ for $O(n\ln n)$, and $f(n) = 2^n$ for $O(2^n)$. However, the correct order of $\hat{\phi}(V_f)$ vs. $\hat{\phi}(V_g)$ depends on the estimator's convergence speed, the size of V_f

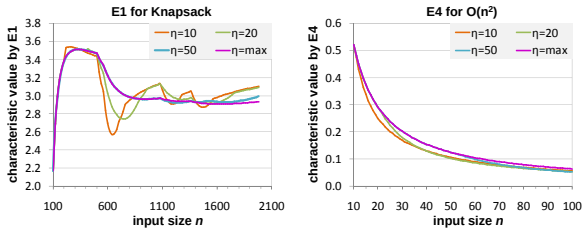
and V_g, as well as the two complexity classes. For example, it is not difficult to observe that for $O(n^2)$ vs. $O(2^n)$, the above mentioned estimators can do well with V_f and V_g of a relatively small size. On the other hand, for $O(n^{100})$ vs. $O(2^n)$, one would need larger V_f and V_g.

The *signature baseline* β is the size of V_f and V_g that ensures $\hat{\phi}(V_f)$ and $\hat{\phi}(V_g)$ return correct ordering of $O(f(n))$ and $O(g(n))$. For a complexity plot, β is the input size that ensures all complexity classes selected for the x-axis will be ordered correctly. As β appears as a horizontal line at value β on the y-axis, it is referred to as a signature baseline, suggesting that on and above this line, the estimator(s) used for the plot impose a correct ordering on the spatial signatures across the x-axis. For a collection of commonly-used complexity classes $O_1, O_2, \ldots, O_h$, we can predetermine all $\beta_{i,j}, i, j = 1, 2, \ldots, h; i \neq j$ using the estimator(s), and store them in a visualization system. When a user selects a subset of complexity classes, the system can automatically choose the largest $\beta_{i,j}$ value for this subset. Alternatively, one can use a crude estimation of $\hat{\beta}$ by simply ensuring $f(\hat{\beta})$ and $g(\hat{\beta})$ have the same ordering as $O(f(n))$ and $O(g(n))$.

3.4. Effect of Window Size η

Given an ideal function for a complexity class, an estimator normally performs more accurately for a larger data series than a smaller one. In practice, the accuracy can also be affected by the noise in the data series as well as its distribution. For example, when the input size is small, the measurements on many algorithms do not give an accurate reflection of the complexity as some constant or low complexity components may dominate the measurements. Moreover, almost all estimators will perform slightly faster with a smaller data series. We thus introduce a sliding window parameter η, with which an estimator $\hat{\phi}$ is swept over m data points, taking up to η measurements at a time. In other words, to compute the new data series $C = \{c_1, c_2, \ldots, c_m\}$ from the original data series V, the estimator $\hat{\phi}$ would use $[v_1, \ldots, v_\eta]$ for estimating c_η; $[v_2, \ldots, v_{\eta+1}]$ for $c_{\eta+1}$ and so forth. For $c_i, i = 3, 4, \ldots, \eta - 1$, $\hat{\phi}$ uses all data points before i, i.e., $[v_1, \ldots, v_i]$. Since **E1**, **E3**, and **E4** require a minimum of three data points, i starts from 3 (and thus the minimum η is also 3). We also have a special window *max*, which uses all available data points $[v_1, v_2, \ldots, v_\eta]$ to estimate c_η.

Using a window-based estimation, the plot can convey local variations around different input sizes, though it may take slightly longer to converge. However, setting $\eta (\geq 3)$ too small, an estimator converges more slowly. From our experience, we found that $\eta > 10$ often leads to better results. In our study, we used $\eta = 20$. Figure 4 shows two sets of examples, demonstrating the effects of η on two different estimators applied to two different algorithms.



(a) E1 (b) E4

Figure 4: The convergence patterns of two estimators, E1 and E4, when they are applied to the Knapsack algorithm and an $O(n^2)$ algorithm respectively. We show the variation with four different windows sizes $\eta = 10, 20, 50, max$, where *max* is the special maximum window size.

4. Visual Design

In designing the visual elements of the complexity plot, we first considered its functional requirements. The plot must convey the following information:

- The complexities of one or more algorithms or measurements on a single plot;
- As above, but using multiple complexity estimators $\hat{\phi}_1, \hat{\phi}_2, \ldots \hat{\phi}_h$ should multiple estimators be required;
- Per estimator, the uncertainty of the estimation along each data series;
- Per data series, the varying window size of the estimator.

In the following two subsections, we describe firstly the various visual components of the complexity plot, and then discuss our methods for estimating uncertainty.

4.1. Visual Components of Complexity Plot

Figure 5 shows a set of designs that address our requirements. In all cases, the horizontal axis denotes the complexity classes and the vertical axis denotes the increasing input size. Each line moves from the bottom (small input size) to the top, indicating the estimation of complexity converging to a stable characteristic value. If the value is close to a known complexity class (a vertical grey line), the complexity of the algorithm concerned is likely to fall into this class. If it sits between two known classes, it likely has a complexity in between, and can be further analyzed visually by choosing other known classes between the current two bounding classes. The figure also shows a general trend of convergence. Most lines have unreliable estimation below the signature baseline and become more accurate with increasing input sizes. For example, in Figure 5*(a)*, the complexity class of Alg_c is near $\Omega(n^2 \ln n)$ for small input sizes; however, as the input size grows, the complexity converges to $\Omega(n^2)$.

Figure 5*(a)* shows an example of the most common use of this form of visualization: comparing the complexities of algorithms. A single estimator is used in this case. The x-axis

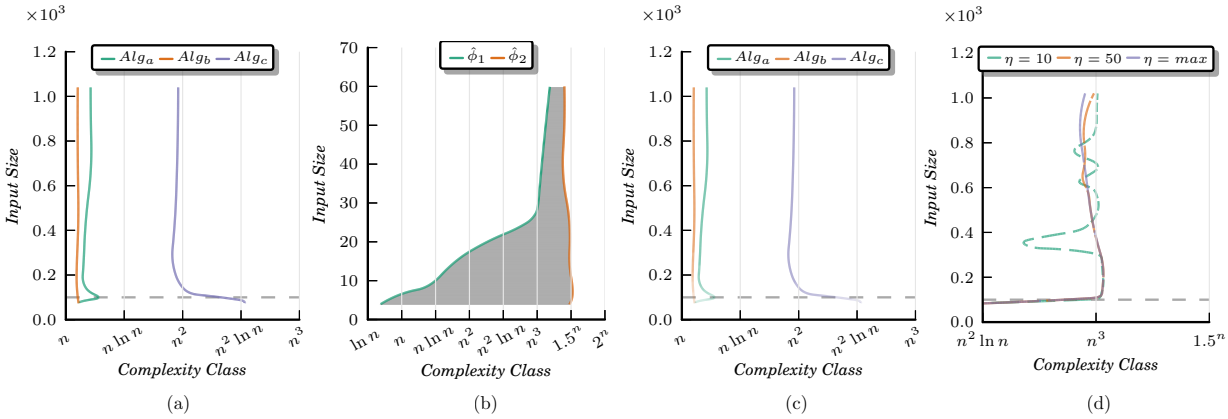


Figure 5: Elements of the complexity plot visual design. *(a)* Comparison of three algorithms' complexities. *(b)* Comparing the convergence of two estimators for an algorithm. *(c)* Conveying uncertainty in the visualization through opacity. *(d)* Comparing an algorithm plotted using estimations of varying window sizes.

is labelled with a range of ordered complexity classes. It is clear from this visualization that Alg_a is somewhere between linear and $\Theta(n \ln n)$ complexity, Alg_b belongs in $\Theta(n)$, and Alg_c is $\Theta(n^2)$. We use the traditional approach in line graphs to differentiate three different algorithms using color.

Figure 5*(b)* shows the case of measuring a single algorithm, but with two estimators ($\hat{\phi}_1$: green, $\hat{\phi}_2$: orange). We draw a translucent background to connect the lower and upper bound of the estimation, suggesting the grouping of the two sets of results as well as a degree of uncertainty.

Figure 5*(c)* is similar to *(a)*, showing three different algorithms plotted with a single estimator. We compute the uncertainty in estimating each data series, and map the level of uncertainty to the opacity of the line. We will detail the uncertainty estimation in Section 4.2.

In Figure 5*(d)*, we convey varying window sizes using different lengths of line stipple. Using three different window sizes ($\eta = 10$, $\eta = 50$, $\eta =$ maximum window size), the same estimator was applied to a single data series. Shorter line stipple indicates smaller window size. When the line is solid, the window size is set to its maximum. From this figure, we can observe that the smallest window size seems to be rather unstable due to localization, while the maximum window size has a smoothing effect.

Figures 5*(b)–(d)* all convey types of uncertainty, but using different visual mappings. In addition, the signature baseline also conveys a type of uncertainty below the line. The designs are consistent with the findings in the literature. For example, representing uncertainty of a line-based data series via stipple patterns is considered a natural choice, with the degree of stipple usually interpreted as the amount of "distrust" [GS06]. Blurring, translucency, reduced intensity and sketchiness are other alternatives [BBIF12].

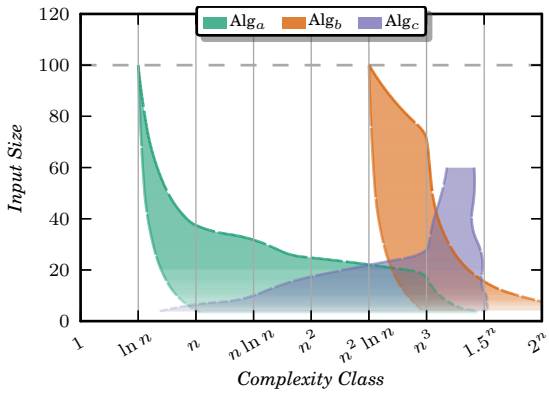
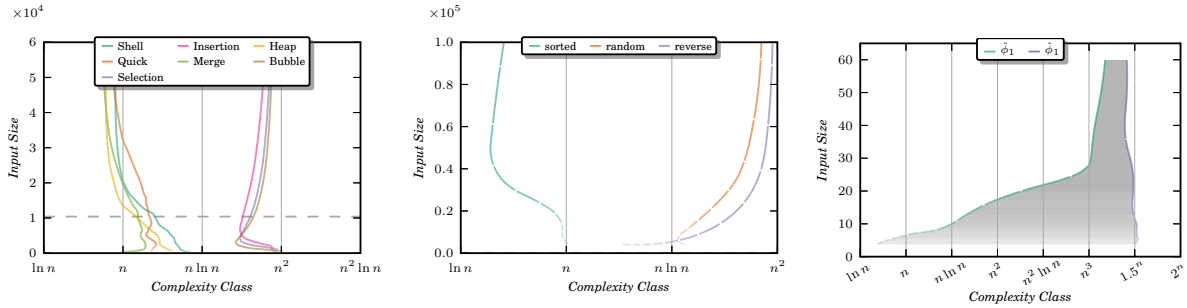


Figure 6: Our final visual design, showing three algorithms (green, orange, purple) analyzed by two estimators each.

One small challenge is to display multiple algorithms with multiple estimators. Figure 6 shows such an example with three algorithms, each with two estimators. Similar to Figure 5*(b)*, each algorithm's estimation boundary is filled using a color assigned to that algorithm's data series, with the opacity set to the average of all estimators' opacities at each input size, multiplied by 0.5 to permit alpha blending at intersections.

4.2. Computing Uncertainty in our Visual Design

As discussed previously, uncertainty may arise from different sources. Some visual mappings are designed explicitly for a specific type of uncertainty, for example, the signature baseline for the area below, the translucent background for multiple estimators, and the line stipple for window sizes. Nevertheless, the opacity of the lines in Figure 5*(c)* is more generic. Here we consider two types of uncertainty that one may wish to display with this visual mapping.



(a) Sorting Algorithms (b) Differing Ordering of Data (c) Knapsack Algorithm

Figure 7: Complexity plots for a variety of algorithms. *(a)* A number of sorting algorithms with different runtime complexities. *(b)* Complexity of insertion sort with different data ordering. *(c)* Complexity of the brute-force Knapsack algorithm.

Convergence Rate Let $C = \{c_1, c_2, \ldots, c_m\}$ be a new data series generated by an estimator $\hat{\phi}$ for incremental input sizes at regular intervals. The convergence rate μ of the new data series is defined as:

$$\mu_i = \frac{|c_i - c_{i-1}|}{|c_{i-1} - c_{i-2}|}$$

As we do not actually know what the value C will converge to, μ_i is an estimation. As long as $\mu_i < 1$ it is converging. We define estimated uncertainty ξ_i as a function of μ_i:

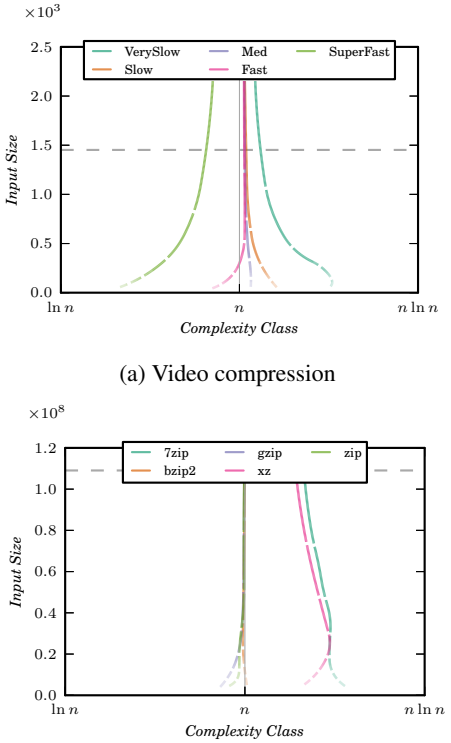
$$\xi_i = \begin{cases} 0.5\mu_i & \mu_i \leq 2 \\ 1 & \mu_i > 2 \end{cases}$$

Deviation due to Multiple Estimators Sometimes it may not be desirable to display the estimations by different estimators explicitly – especially to avoid cluttering. One may wish to map the deviation caused by multiple estimators to the opacity of the line. The deviation can take many forms, including the extent between the lower and upper bounds of the estimation, standard deviation, and mean squared error.

Ideally one would wish to have many visual channels for mapping different types of uncertainty. However, the availability of visual channels may not scale with the variety of uncertainty in the data. On balance, we decided to use opacity as the generic visual component for uncertainty, while maintaining specific semantic representations of the signature baseline, translucent background and stipple lines.

5. Case Studies

We demonstrate the utility and effectiveness of the complexity plot using two case studies in the subsections below. The first study focuses on the application of our technique to complexity analysis, whilst the second incorporates multivariate analysis of algorithms in the context of runtime, memory usage and energy consumption. Both of these studies demonstrate the power of the complexity plot in capturing the growth rate of different algorithms and software.



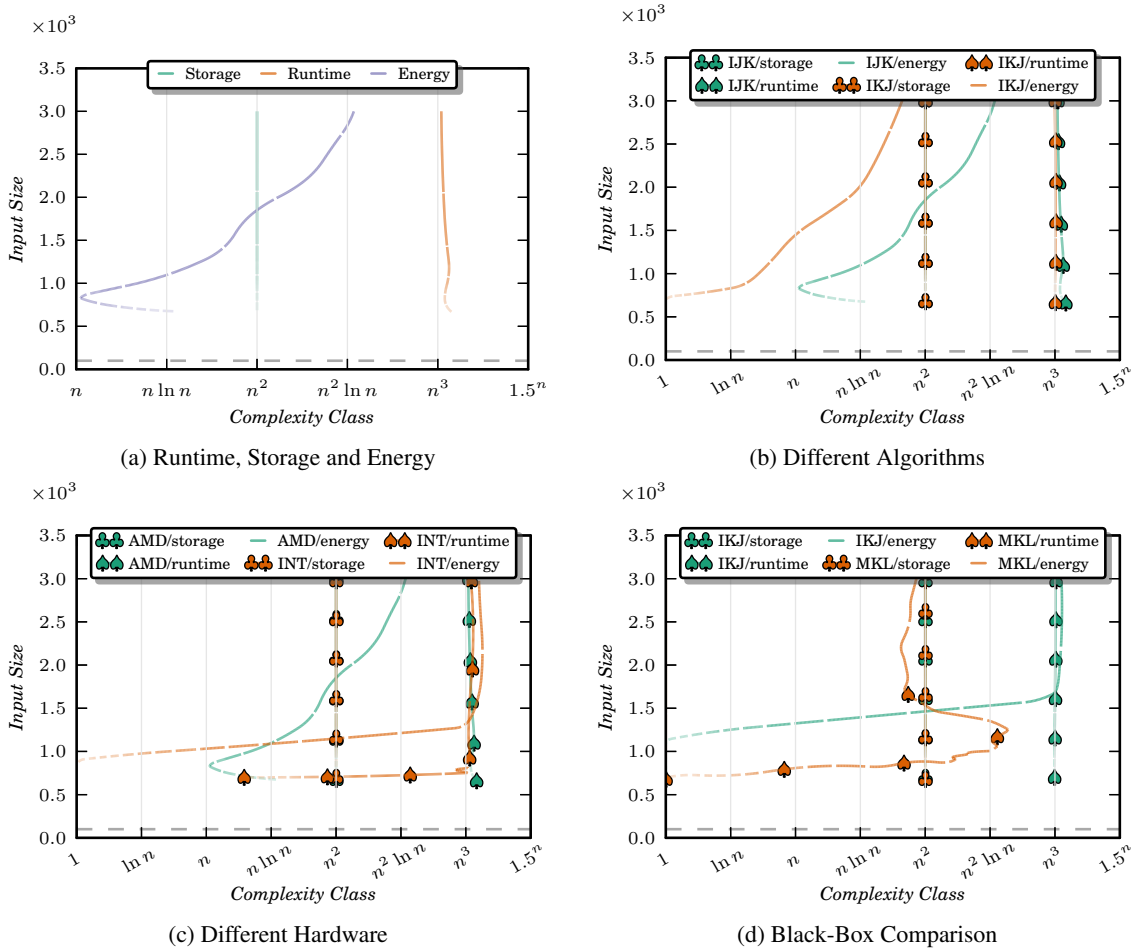
(a) Video compression

(b) File compression

Figure 8: Comparing *(a)* varying quality settings for an *ffmpeg* encode; *(b)* a set of lossless file compression algorithms.

5.1. Case Study 1 – Algorithms

Our study consists of a suite of algorithms and a set of black-box software. Algorithms were selected from a number of sources in the literature [CSRL01, Dro06, Baa88] and implemented in C++. Visualizations corresponding to these algorithms are shown in Figure 7. Figure 7*(a)* compares the runtime complexities of a number of sorting algorithms for randomized input: quicksort, insertion, shell, merge, heap,



(a) Runtime, Storage and Energy

(b) Different Algorithms

(c) Different Hardware

(d) Black-Box Comparison

Figure 9: Different complexities compared for various cases. *(a)* Combined visualization of storage, runtime and energy complexities. *(b)* Comparison of runtime and energy complexities for MMIJK and MMIKJ versions. *(c)* Different complexities of MMIJK compared across two different hardware platforms. *(d)* Comparing runtime/energy complexities of IKJ against a highly-optimized vendor library (IMKL).

selection and bubble sort. From Figure 7, it is easy to observe the complexity of these algorithms. Notably, the algorithms perform uniformly better than the average case complexity predicted by the Big O classification. We augment this analysis by showing different orderings of input elements for insertion sort in Figure 7*(b)*: sorted (green), randomized (orange), and reverse-order (purple). Insertion sort gives worst-case performance of $O(n^2)$ for input that is in reverse (descending) order. The corresponding best, average and worst case complexities for insertion sort clearly emerge as the curves in the complexity plot conform to the expected classes.

Figure 7*(c)* examines an *NP-Complete* problem, namely the Knapsack problem [Baa88], where for a given set of items the algorithm fills a knapsack of a fixed capacity with a subset of items that meets its capacity. Once a valid subset of items is found, the algorithm terminates. A *brute force* approach tries every subset, yielding a worst case complexity of $O(2^n)$. The figure shows the complexities from two different estimators. We observed the outputs converge to a slightly lower indicator close to $O(1.5^n)$ instead of the expected complexity of $O(2^n)$.

We used a number of Unix utilities as 'black-box' software to test the complexity plot. Figure 8*(a)* shows the results for a set of *ffmpeg* encodes of a video using the *H*.264 codec with a number of preset quality categories. Figure 8*(b)* shows the complexities of a number of file compression tools, namely `rar`, `bzip2`, `gzip`, `zip`, `7z` and `xz`.

5.2. Case Study 2 – Energy

Computing plays a vital role across all scientific disciplines, and the benefits of large scale computation are undeniable. These benefits however come with costs. In this case study,

we consider the energy cost. In recent years, more emphasis has been put on the importance of 'green computing' [Mur08]. Strategies for minimizing energy consumption of computing through software are considered to be a viable option [LdSS*10] and are an emerging topic of research.

Quantifying energy consumption of hardware through sensor devices is relatively easy. Analyzing software and algorithms is not. Many factors may affect the energy consumption of software. Recent research [CM08] has attempted to tackle this problem, but the proposed techniques rely on analyzing correlations among different factors. In addition, there is a need to measure and analyze energy consumption at different scales of input size for comparison with measurements of runtime and storage cost. Although one may analyse and predict algorithmic complexity through asymptotic analysis, energy is hardware-dependant and one has to rely on captured data. The complexity plot is thus useful in this application.

Figure 9 demonstrates the effectiveness of the complexity plot for real-world data. Figure 9*(a)* employs a complexity plot for the same data visualized in three separate plots in Figure 1. Comparing the rates of growth of different properties of an algorithm now becomes easier. This example illustrates a common misconception: that energy complexity is mostly correlated with runtime complexity. It is clear from Figure 9*(a)* that this is false. We will see in later examples that energy consumption is more closely dependant on algorithmic design and underlying hardware and configurations.

Figure 9*(b)* shows two algorithms implementing matrix multiplication. The complexity plot reveals significantly different energy signatures despite having the same runtime complexity. The first algorithm, MMIJK, (acronym referring to the loop iteration ordering) is a naïve implementation resulting in the worst-case exploitation of the spatial locality of array elements. This leads to excessive use of cache memories. In contrast, the second algorithm, MMIKJ, has improved temporal and spatial locality and is more cache-friendly. The difference in the energy signatures of MMIJK and MMIKJ is primarily due to memory access patterns affecting the amount of data transfer.

In addition to implementation, architecture also impacts the overall energy signature of an algorithm. Figure 9*(c)* shows the energy, runtime and storage complexities of MMIJK on two different processor architectures, AMD Magny Cours (6128 model) and Intel Xeon (E5-2680 model). Because the complexity plot classifies algorithms by growth rate, we can easily compare the algorithm implementations on different processors and/or architecture types. For MMJIK, runtime and storage map to the expected complexity classes. For energy we observe distinctly different signatures on each processor architecture, confirming the hypothesis that energy complexity is hardware dependant.

Optimizing implementations to match architectural peculiarities is known to yield better performance but the effect on algorithmic and energy complexity has not been fully understood. To demonstrate this, we used DGEMM matrix multiplication implemented in the Intel Math Kernel Library (IMKL). IMKL exploits all available architectural features, such as fused-multiply-add operations, vectorization units, registers, cores, and performs an extreme degree of tiling for better cache memory locality. Figure 9*(d)* clearly shows the effect of performing architecture-specific optimizations on the energy signature of an algorithm.

6. Discussions and Conclusions

In this paper, we have proposed a new form of line graphs specifically designed for supporting complexity analysis. This novel visualization technique, referred to as the *complexity plot*, allows more intuitive and scalable observations of the complexity of any measured data series that represents a dynamic property of an algorithm in relation to input size. Because it is unitless, it is ideal for comparative visualization of related measurements of different properties. In this work, we have formulated the mathematical transformation from conventional line graphs to the complexity plots. We have provided a practical solution for creating such a plot using estimators, and a set of visual designs for visualizing the uncertainty associated with the estimation process. We have demonstrated the utility and effectiveness of the complexity plot through two case studies. The application to the analysis of energy consumption has resulted in new insights.

As a novel concept, it has a number of limitations. Firstly, it is relatively easy for a viewer to mistake the x-axis as linearly scaled. This is inherently related to the fundamental understanding of the relationship between any two complexity classes. In most cases, there is no linear scaling. This issue is very similar to the risk of misinterpretation with a logarithmic plot, and can be addressed through education and training. It will also be useful to conduct empirical studies in the future to study the cognitive load in visualizing algorithmic complexity. Another limitation is the use of estimators, which result in several different types of uncertainty. However, similar types of uncertainty exist widely in a variety of numerical and statistical methods, such as regression, correlation, clustering, multi-dimensional reduction, and so on. Our solution to this limitation is to be open about the uncertainty and to use visualization to depict the uncertainty. This is a visualization designed for our own discipline; namely computer science. We hope that it will lead to many improvements and extensions in the future.

Acknowledgements

The authors wish to thank the partial support from several funding bodies, including James Martin Foundation, and UK EPSRC. This publication is based on work supported by Award No. KUK-C1-013-04, made by King Abdullah University of Science and Technology (KAUST).

References

[AAG03] ANDRIENKO N., ANDRIENKO G., GATALSKY P.: Exploratory Spatio-Temporal Visualization: An Analytical Review. *Journal of Visual Languages and Computing 14* (2003), 503–541. 2

[AMST11] AIGNER W., MIKSCH S., SCHUMANN H., TOMINSKI C.: *Visualization of Time-Oriented Data*. Human-Computer Interaction Series. Springer, 2011. 3

[ARH12] AIGNER W., RIND A., HOFFMANN S.: Comparative Evaluation of an Interactive Time-Series Visualization that Combines Quantitative Data with Qualitative Abstractions. *Computer Graphics Forum 31*, 3(pt.2) (2012), 995–1004. 2

[Baa88] BAASE S.: *Computer Algorithms, Introduction to Design and Analysis*, 2nd ed. Addison Wesley, 1988. 7, 8

[Bac94] BACHMANN P. G. H.: *Die analytische Zahlentheorie*. Teubner, Leipzig, 1894. 2

[BBIF12] BOUKHELIFA N., BEZERIANOS A., ISENBERG T., FEKETE J.-D.: Evaluating Sketchiness as a Visual Variable for the Depiction of Qualitative Uncertainty. *IEEE Transactions on Visualization and Computer Graphics 18* (2012), 2769–2778. 6

[BPC*10] BORGO R., PROCTOR K., CHEN M., JANICKE H., MURRAY T., I. T.: Evaluating the Impact of Task Demands and Block Resolution on the Effectiveness of Pixel-based Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16 (2010), 963–972. 2

[BW08] BYRON L., WATTENBERG M.: Stacked Graphs — Geometry & Aesthetics. *IEEE Transactions on Visualization and Computer Graphics 14* (2008), 1245–1252. 2

[CJ10] CHEN M., JAENICKE H.: An Information-theoretic Framework for Visualization. *IEEE Transactions on Visualization and Computer Graphics 16* (2010), 1206–1215. 1

[Cle85] CLEVELAND W. S.: *The Elements of Graphic Data*. Brooks Cole, 1985. 1

[CM08] CHO S., MELHEM R.: Corollaries to Amdahl's Law for Energy. *IEEE Comput. Archit. Lett. 7*, 1 (2008), 25–28. 9

[CSRL01] CORMEN T. H., STEIN C., RIVEST R. L., LEISERSON C. E.: *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001. 2, 7

[Dro06] DROZDEK A.: *Data Structures and Algorithms in C++*, 3 ed. Cengage Learning, 2006. 7

[FD05] FRIENDLY M., DENIS D.: The Early Origins and Development of the Scatterplot. *Journal of the History of the Behavioral Sciences 41*, 2 (2005), 103–130. 1

[Fun36] FUNKHOUSER H. G.: A Note on a Tenth Century Graph. *Osiris 1* (1936), 260–262. 1

[GAW07] GOLDSMITH S. F., AIKEN A. S., WILKERSON D. S.: Measuring Empirical Computational Complexity. In *Proc. 6th European Software Eng. Conf. and ACM SIGSOFT Symposium Foundations of Software Eng.* (2007), ACM, pp. 395–404. 2

[GS06] GRIETHE H., SCHUMANN H.: The Visualization of Uncertain Data: Methods and Problems. In *Proc. SimVis* (2006), pp. 143–156. 6

[Har99] HARRIS R. L.: *Information Graphics: A Comprehensive Illustrated Reference*. Oxford University Press, NY, 1999. 2

[HHWL02] HARVE S., HETZLER E., WHITNEY P., L. N.: Themeriver: Visualizing Thematic Changes in Large Document Collections. *IEEE Transactions on Visualization and Computer Graphics 8* (2002), 9–20. 2

[HS04] HOCHHEISER H., SHNEIDERMAN B.: Dynamic Query Tools for Time Series Data Sets: Timebox Widgets for Interactive Exploration. *Information Visualization 3*, 1 (2004), 1–18. 2

[JME10] JAVED W., MCDONNEL B., ELMQVIST N.: Graphical Perception of Multiple Time Series. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 927–934. 2

[Knu97] KNUTH D. E.: *The Art of Computer Programming, Vol. 1:Fundamental Algorithms*. Addison Wesley Longman, Redwood City, CA, USA, 1997. 2

[LdSS*10] LI D., DE SUPINSKI B., SCHULZ M., CAMERON K., NIKOLOPOULOS D.: Hybrid MPI/OpenMP Power-Aware Computing. In *Proc. IEEE Int. Symposium Parallel and Distributed Processing* (2010), pp. 1–12. 9

[LM88] LE MÉTAYER D.: ACE: An Automatic Complexity Evaluator. *ACM Trans. Program. Lang. Syst. 10*, 2 (1988), 248–266. 2

[MB05] MILLER R., BOXER L.: *Algorithms Sequential and Parallel: A Unified Approach*, 2nd ed. Charles River Media Inc., 2005. 2

[Mur08] MURUGESAN S.: Harnessing Green IT: Principles and Practices. *IT Professional 10*, 1 (2008), 24–33. 9

[Pla86] PLAYFAIR W.: *The Commercial and Political Atlas: Representing, by Means of Stained Copper-plate Charts, the Progress of the Commerce, Revenues, Expenditure and Debts of England during the Whole of the Eighteenth Century*, 1st ed. T. Burton for J. Wallis, London, 1786. 2

[PSB92] PRICE B. A., SMALL I. S., BAECKER R. M.: A taxonomy of software visualization. *Journal of Visual Languages and Computing 4* (1992), 211–266. 2

[PTVF07] PRESS W. H., TEUKOLSKY S. A., VETTERLING W. T., FLANNERY B. P.: *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, 2007. 3

[SMY*05] SAITO T., MIYAMURA H. N., YAMAMOTO M., SAITO H., HOSHIYA Y., KASEDA T.: Two-Tone Pseudo Coloring: Compact Visualization for One-Dimensional Data. In *Proc. IEEE Symp. Information Visualization* (2005), pp. 174–180. 2

[Sta90] STASKO J. T.: Tango: A Framework and System for Algorithm Animation. *Computer 23*, 9 (1990), 27–39. 2

[Tuf01] TUFTE E. R.: *The Visual Display of Quantitative Information*, 2nd ed. Graphics Press, 2001. 2

[WAM01] WEBER M., ALEXA M., MÜLLER W.: Visualizing time-series on spirals. In *Proc. IEEE Symposium Information Visualization* (2001), pp. 7–. 2