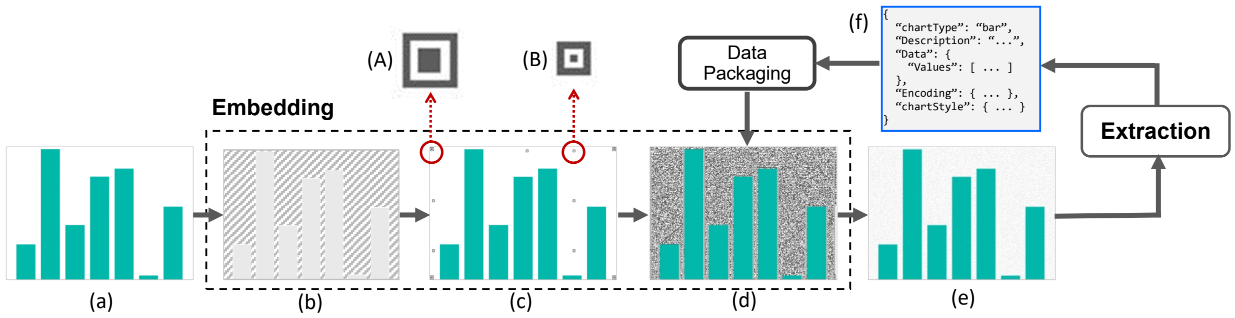
# Chartem: Reviving Chart Images with Data Embedding

Jiayun Fu, Bin Zhu, Weiwei Cui, Song Ge, Yun Wang, Haidong Zhang, He Huang,
Yuanyuan Tang, Dongmei Zhang, and Xiaojing Ma



```
{
  “chartType”: “bar”,
  “Description”: “...”,
  “Data”: {
    “Values”: [ ... ]
  },
  “Encoding”: { ... },
  “chartStyle”: { ... }
}
```

Fig. 1. Chartem Overview. Chartem embeds a piece of application-dependent digital information such as a chart specification (f) into a chart image (a). It detects background regions of the chart (b), embeds coarse marks (A), fine marks (B), and packaged information into the background regions (c and d), and adjusts the opacity of the embedded patterns to produce an information-embedded chart image (e), which can be used as a regular chart since the embedding patterns are generally subtle. However, when needed, Chartem can recover the embedded information (f) for different uses by determining and decoding the background patterns.

**Abstract**— In practice, charts are widely stored as bitmap images. Although easily consumed by humans, they are not convenient for other uses. For example, changing the chart style or type or a data value in a chart image practically requires creating a completely new chart, which is often a time-consuming and error-prone process. To assist these tasks, many approaches have been proposed to automatically extract information from chart images with computer vision and machine learning techniques. Although they have achieved promising preliminary results, there are still a lot of challenges to overcome in terms of robustness and accuracy. In this paper, we propose a novel alternative approach called Chartem to address this issue directly from the root. Specifically, we design a data-embedding schema to encode a significant amount of information into the background of a chart image without interfering human perception of the chart. The embedded information, when extracted from the image, can enable a variety of visualization applications to reuse or repurpose chart images. To evaluate the effectiveness of Chartem, we conduct a user study and performance experiments on Chartem embedding and extraction algorithms. We further present several prototype applications to demonstrate the utility of Chartem.

**Index Terms**—Chart embedding, background embedding, data embedding, chart image, chart reuse.

## 1 Introduction

As an effective and efficient means to convey quantitative information [39], charts have become an increasingly pervasive type of content widely adopted in newspapers, textbooks, websites, academic papers, etc. Nowadays, there are many tools, such as Excel, Tableau, and Power BI, to help users convert data into charts or graphs effortlessly. During the authoring process, a chart object is often created to maintain relationships between data and visual elements. After the authoring process, it is common to save the created chart as a bitmap image, for easy typesetting or sharing. In many cases, the resulting image is then disconnected from its chart object and becomes the only representation available for the underlying data. This may cause several issues in the long run. First, since the carried information and visual style are locked in a chart image, it is hard to reuse or repurpose the chart in the future. For example, if Alice wants to change the chart type or style for a different story or document, she often has to do it manually as a chart image is generally not machine readable. To assist with this task, many image-recognition-based techniques have been proposed to automatically recover data and visual design information from chart images [5, 6, 10, 11, 20, 21, 25]. However, this is still a relatively new research direction, and a robust solution that can accurately recover the full information of a chart image has not been accomplished yet due to diversity and complexity of chart content. Second, in many cases, the message conveyed by a chart is distilled from a bigger dataset via a series of aggregation and filtering operations. If a user likes to perform a different analysis on the same underlying dataset for a different purpose, it will be impossible since the information carried by the chart is limited and the original dataset is completely lost after the conversion.

In this paper, we present a novel approach to solving the above issues and unlocking chart images with more potential. Our solution, called *Chartem*, embeds the chart data or arbitrary data into a chart image when it is published. Once embedded, the information becomes an intrinsic part of the chart image. It can be retrieved, when needed, for further processing. For instance, if the information is the chart data, it can be directly used to generate a chart with a different style or type, or be analyzed for a different insight, etc. An overview of this process is shown in Fig. 1. Unlike image-recognition-based techniques, the integrity of extracted information is verified in our method to guarantee completeness and accuracy of the restored chart object. As a result, our proposed method unlocks chart data accurately, which would open a great opportunity for reusing underlying data of chart images.

There are alternative information-carrying solutions, such as overlaying information on a chart image (e.g., QR code) or piggybacking the information into chart files. Our data-embedding approach has two advantages over them. First, our embedding patterns are not apparent or intrusive during chart reading, thus the same user experience of normal charts is preserved. Second, the embedded information stays with the

- *J. Fu, Y. Tang, and X. Ma are with Natl. Eng. Res. Ctr. for Big Data Tech. and Sys., Big Data Sec. Eng. Res. Ctr., School of Cyber Science and Technology, Huazhong Univ. of Science and Technology, Wuhan, China. E-mails: {fujiayun, tangyuanyuan, lindahust}@hust.edu.cn. This work was done when J. Fu and Y. Tang were interns at Microsoft Research Asia.*
- *B. Zhu, W. Cui, S. Ge, Y. Wang, H. Zhang, H. Huang, and D. Zhang are with Microsoft Research Asia. E-mails: {binzhu, weiweicu, songge, wangyun, haizhang, rayhuang, dongmeiz}@microsoft.com.*

*Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx*

chart even after the chart image is format-converted or screenshot.

Image data embeddings have been extensively studied for natural images. These methods have been built on the continuous-tone characteristics of natural images. Unlike natural images, chart images typically comprise homogeneous color regions, which makes natural-image-based embedding techniques ineffective for chart images.

In Chartem, we present a novel data-embedding scheme specifically designed for chart images. It embeds arbitrary data into background regions of a chart image by slightly modifying the pixel values to form deliberated patterns to encode the data. Embedding patterns are faintly visible and do not incur adverse impact on chart reading. Chartem adopts a data segmentation mechanism as well as a design of synchronization marks to enhance the robustness of data extraction. The embedded data can be accurately extracted even if an embedded chart image undergoes typical image manipulations such as resizing, screenshot, compression, rotations, and brightness variations.

We present a user study to understand the impact of embedding patterns on human perception of charts and experimental evaluation of Chartem's performance. In illustrating potential utilization of Chartem, we present a prototype application as an Excel add-in to demonstrate how Chartem can be integrated into a chart authoring system to generate chart images with embedded data, followed by two prototype applications to show that Chartem can enable scenarios such as redesigning charts and reading charts to people with vision impairment.

## 2 Related Work

### 2.1 Parsing Chart Images

A huge amount of information is locked inside chart images and inaccessible to machines and visually impaired people [6,20]. To overcome this limit, many computational methods have been proposed to interpret chart images based on OCR and image recognition techniques. From a given bitmap chart image, these methods attempt to extract information including chart type, underlying data, visual encodings, etc.

ReVision [25] first uses a SVM model to detect chart type, then applies image processing techniques to locate the marks and recover data from bar and pie chart images. FigureSeer [26] trains a CNN network for chart type classification and uses legend information for more accurate data extraction. DVQA [11] employs a deep dual-network model to directly parse the data from bar chart images. Scatteract [7] automatically restores the numerical values of data points from images of scatter plots. More recently, Choi et al. [6] built a DNN-based automatic pipeline to extract data from chart images for reading to visually impaired people. Apart from chart data, there are research works focusing on chart design aspects. For example, Poco and Heer [20] proposed a multi-stage pipeline, which combines ML and heuristics techniques, to automatically infer a visual encoding specification from a chart image. Poco et al. [21] sought to extract color mappings from chart images. Besides images of standard charts, researchers have also investigated computational methods to parse images of infographics [2,5,14]. Due to diversity and complexity of chart content, all these techniques support only a limited number of chart types (e.g., bar, pie, line, scatter charts). Moreover, they often cannot achieve sufficient accuracy of data extraction, especially when a chart has overlapped visual entities. Although some techniques, such as ChartSense [10] and iVoLVER [16], adopt a mixed-initiative approach to improving data extraction accuracy with human interactions, they are not suitable for applications that require fully automated processing.

While sharing the same goal with these techniques, our work takes a completely different approach to unlocking chart images with more potential. Specifically, Chartem embeds data into background regions of a chart image. The embedded data can be extracted to support further processing. Compared with prior works on parsing content of chart images, our solution has several advantages. First, our solution is more robust and accurate as desired information is directly embedded into chart images. Second, since our solution does not depend on interpreting visual elements to decode information, it can be easily applied to different chart types. Third, the embedded data can be any digital information even not being presented on charts, so our solution can enable richer chart reuse applications.

### 2.2 Data Embedding and Watermarking

Both watermarking and data embedding embed information into a host signal, typically audio, image, or video [28]. While they share many common properties and requirements, watermarking and data embedding are targeted for different applications. Watermarking is generally used for tracking and copyright protection. A small number of bits are embedded, but the embedded data has to be very robust against all possible perceptual-quality-preserving manipulations including intentional attacks. Data embedding, on the other hand, generally embeds as much information as possible into a host signal, and the embedded data only has to survive the processing needed in its targeted applications. As a special type of data embedding, steganography aims to conceal the presence of a hidden message in a host signal [4]. Embedding in steganography should be imperceptible and undetectable.

Data embedding embeds information into a host signal by modifying selected features of the host signal, while watermarking can either embed watermark in or superimpose an additive spread spectrum watermark on the host signal. We focus on embedding techniques used in watermarking and data embedding. Features selected to carry information can be pixels in the spatial domain or coefficients in a transform domain such as in the frequency domain or a wavelet-transform domain. Spatial-domain embedding is generally for host images, wherein the least significant bits (LSBs) of pixels are modified to carry information [3,40] or pixels are modified in pairs, with each pair carrying one bit information [38]. Image steganography typically embeds in the spatial domain too, e.g., HUGO [19]. Transform-domain embedding, on the other hand, can be applied to audios [34,35], images [8,12,31,32,45], and videos [29,30,33,45], wherein middle and/or high frequencies in the frequency domain [8, 12, 30–35] or a wavelet-transform domain [29,45] are modified. Spatial-domain embedding is generally less robust than transform-domain embedding.

In addition to the above traditional approaches, deep learning has also been used for data embedding. An embedding network and an extraction network can be trained simultaneously to hide an image [1] or arbitrary data [43] into a host image for image steganography. They can hide a large amount of data into a host image, but small perturbations of a common image manipulation such as JPEG compression, screenshot, resizing, or rotation would render the hidden data unextractable. By incorporating, during training, various perturbations that an image may go through, hidden data is still extractable after JPEG compression and cropping [44], displaying and photographing [42], or printing and photographing [36], but the embedding capacity is significantly reduced, e.g., 56 bits for [36]. The perceptual quality of images produced by these deep-learning-based methods is generally good, but the embedding residual can be perceptible in large low-frequency regions of a host image [36], and a sharp edge can be found blurred.

All the aforementioned watermarking and data-embedding methods are designed for natural host signals, e.g., natural or continuous-tone images. Unlike natural images, synthetic images such as chart images typically comprise homogeneous color components. Spatial-domain-embedding methods used for natural images are generally ineffective for synthetic images since data embedding makes a homogeneous color region no longer homogeneous after embedding, resulting in perceptible embedding residual. Transform-domain-embedding methods are also ineffective since a homogeneous region has only energy around zero frequency. There is no middle or high frequency that can be modified to carry information. To address the unique characteristics of synthetic images, Masry [15] proposed a watermarking scheme for map and chart images by modifying boundaries of homogeneous color components. Designed for watermarking applications, this method can embed only limited information, and thus is ineffective for data-embedding applications that we focus on.

### 2.3 QR Code

Since its introduction in 1994 by Masahiro Hara [23], the QR code [41] has been widely used to carry information for various applications. The QR code is a machine-readable 2D barcode of black and white cells. Inspired by the QR code design, Chartem has borrowed some designs from the QR code, such as coarse and fine marks. On the other hand, our scheme differs from QR codes in several critical ways: our information carrying patterns are chart-dependent, faintly visible, and interleaved with foreground regions that vary from one chart to another. These key differences demand a very different approach.

## 3 Chartem

Chartem consists of two parts: an embedder to embed information into chart images, and an extractor to extract embedded information from chart images. Like traditional data embedding, Chartem faces three main conflicting requirements or challenges:

**Perceptual quality.** A chart embedded with information should not interfere with normal consumption of the chart.

**Capacity.** All desired information should be able to be embedded into a host chart image.

**Robustness.** The embedded information should be correctly extracted after targeted processing and distortions.

These requirements are inversely related. It is generally more robust when perceptual quality is lowered or capacity is reduced.

In viewing a chart, human's attention is typically focused on foreground components of the chart. To ensure perceptual quality, Chartem modifies only background pixels to carry information while keeping foreground components unchanged. To improve robustness, Chartem packages input data into segments. Each segment can independently determine if its extracted data is correct and complete. To balance capacity and robustness, Chartem uses fountain codes [13] to generate extra segments to embed whenever there is extra capacity. Any set of recovered segments with the count equal to the number of segments the input data is partitioned into can virtually recover the whole embedded data. A chart image the extractor receives may have a different shape or size from its original chart image, unknown to the extractor. Chartem embeds two sets of synchronization marks, or simply marks, for the extractor to register a received chart image to its original chart image.

### 3.1 Chartem Embedder

Fig. 1 includes a flowchart of the embedding process: Chartem detects the background of a chart image, embeds coarse and fine synchronization marks and bits of segments generated from packaging input data and fountain coding, and adjusts the visibility of generated embedded patterns via a weight. The resulting data-carrying chart image has the same size as and looks nearly identical to the original chart image. These processing steps will be described in detail in the following subsections.

#### 3.1.1 Background Detection

Background locations can be either passed to Chartem from a chart creation tool or detected by Chartem. For good usability, foreground components in a chart are typically visually distinctive from the background. This distinctiveness and the characteristics of chart images are exploited to detect the background of a chart image:

First Chartem groups pixels into clusters by classifying a new pixel into the cluster closest to it if the distance is within a threshold, determined a priori by the expected spread of background pixel values, or otherwise a new cluster. Each cluster maintains a histogram of its pixels and a reference value equal to the center of the histogram bin with the highest count. The distance of a pixel to a cluster is defined as the distance of the pixel's value to the reference value of the cluster, and the distance between two clusters is defined as the distance of their reference values. The reference value of a cluster is updated whenever the cluster adds a fixed number of new pixels.

Then Chartem labels one cluster as background and remaining clusters as foreground based on the cluster's size, spatial shape and location in the image, and distances to other clusters. The foreground is structurally dilated. Isolated foreground pixels and small background regions are removed. The resulting background is the background to find.

#### 3.1.2 Coarse and Fine Synchronization Marks

After data embedding, a chart image may undergo size or shape changes such as scaling. The extractor needs to register a received chart image to its original size and shape before correctly extracting the embedded

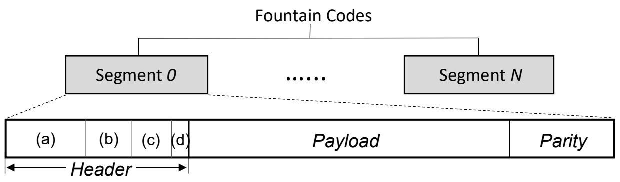


Fig. 2. The structure of a segment. Header consists of a marker (a), a segment ID (b), CRC (c), and a status bit (d).

data. A general approach like in QR codes is to insert specially designed marks at a preset distance to register a received image to the desired size and shape. Unfortunately, this approach does not work for Chartem since marks can be inserted only into background regions of a chart image. Background regions vary from one chart to another. A fixed location may not be available for all chart images to embed a mark.

To address this problem, Chartem embeds two sets of synchronization marks: coarse marks for rough and fine marks for accurate estimation of transformation parameters. These transformation parameters are used to register a received chart image to its original shape and size. Both marks have unique patterns that can be easily identified. Fig. 1(A) and (B) show the coarse and fine mark patterns that Chartem uses, respectively. The coarse mark is a pattern of 9 by 9 logical bits, while the fine mark is a pattern of 7 by 7 logical bits. Their center ratios along both directions are 1:1:1:3:1:1:1 and 1:1:1:1:1:1:1, respectively.

In its basic setting, Chartem inserts at least three and up to four coarse marks at the corners of a rectangle within which data embedding occurs. The rectangle can contain foreground components. In such a setting, coarse marks indicate a bounding box of data-embedding regions. This setting is not necessary since Chartem uses start and end blocks to indicate where data blocks are located (Section 3.1.5). In a general setting, the rectangle can be anywhere in a chart image, as long as at least three coarse marks can be embedded at its corners. The rectangle should be large enough to reduce potential estimation errors at the extractor.

To embed fine marks, Chartem determines a grid of cell size $h \times v$ logical pixels and its bias so that more fine marks can be embedded at grid intersections in the background, where $h, v \in A$, and $A$ is a set of admissible values for a grid. Chartem selects $A = \{56, 63\}$, which is designed to uniquely determine, after image registration with coarse marks, $h$ or $v$ of the grid used in a received chart image from two detected fine marks up to 4 cells apart. Chartem requires inserting at least three fine marks not aligned along a horizontal or vertical line. More embedded fine marks improve robustness since the extractor may miss some fine marks. In Fig. 1(c), six fine marks are embedded with a grid of $56 \times 56$ logical pixels: one circled by the right red circle, one on its left and two below it, and two more below the left red circle.

The synchronization marks are embedded into the background of a chart image first. They are embedded in the same way as embedding data, which is described in Section 3.1.5. Data is embedded into remaining background regions.

#### 3.1.3 Data Packaging

Information to be embedded can be arbitrary data. Input data is first compressed losslessly to reduce its capacity requirement, and then prefixed with 2 bytes to represent the length of compressed data. The prefixed input data is then partitioned and packaged into segments. Each segment can be extracted and checked correctness independently. Such data packaging prevents error propagation from one segment to another and thus improves the robustness of extraction.

A chart image may have extra capacity after embedding the segments constructed from input data. In this case, we use fountain codes [13] to generate an arbitrary number of segments to exhaust all the embedding capacity of the chart image. Fountain codes are a class of erasure codes that can generate a potentially unlimited number of segments from a set of source segments such that the source segments can be fully recovered from any subset of segments with the size equal to or slightly larger than the number of the source segments. The fountain coding [37] used in Chartem preserves input data. As a result, we refer to a segment constructed from input data as a *data segment* and a segment generated

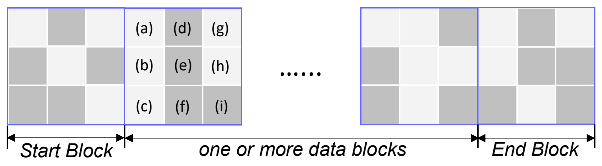


Fig. 3. A data block sequence: start and end blocks at both ends, and one or more data blocks in the middle. In a data block, there is a flip bit (a), and the remaining (b-i) bits are data bits.

from fountain codes as a *fountain segment* in this paper.

A segment consists of header, payload, and parity, as shown in Fig. 2. The header is composed of a marker to identify a segment header from a bitstream, a segment ID for the index of the segment among all distinct segments, a status bit to indicate if the segment is a data or fountain segment, and a CRC (Cyclic Redundancy Check) code to check if there is any error in the combination of the segment ID, the status, and the payload. Payload contains data from user data for a data segment or data generated by fountain codes for a fountain segment. Parity contains error correction parity to correct errors in the combination of the payload and the header excluding the marker in the header. In this setting, the total number of distinct segments is limited by the size of segment ID. When more segments can be embedded, Chartem reuses existing segments.

#### 3.1.4 Data Blocks

Bits in segments are partitioned into data blocks, which are then organized into block sequences. Each block sequence consists of a start block, an end block, and one or more data blocks between them, as shown in Fig. 3, and is embedded into a continuous background region not taken by any synchronization mark. Each block comprises $m \times n$ such as $3 \times 3$ logical bits. The start and end blocks are special blocks with fixed bit patterns. They are used to identify a block sequence.

A data block contains a flip bit to indicate if the bits in the block have been flipped or not, and the remaining bits are bits from segments. When adding a data block to a block sequence, Chartem checks if the start or end block is replicated. If replication occurs, the newly added data block is flipped to eliminate the replication. In this way, there is no replication of the start or end block in any block sequence.

#### 3.1.5 Embedding Data

To embed a block sequence, each logical bit in the block sequence is mapped into a preset block of, such as $p \times p$, pixels. The block size is determined by targeted applications. Mapping to a large block of pixels increases robustness at the cost of reduced capacity. Based on the logical bit value, Chartem assigns each pixel in the block a color value slightly above or below the local average of background pixels. The gap between pixels representing 0 and 1 of a logical bit is a fixed factor times a weight. By adjusting the weight, we can adjust the perceptual quality of data-embedded chart images. The larger the weight is, the more visible the embedded patterns are. If logical bits are randomly distributed, this embedding process preserves the local average of background pixels. When moving up or down a specified distance from a local average results in a value outside the valid value range of pixels, Chartem shifts the assigned value back to the valid range while preserving the gap between pixels representing 0 and 1. After such shifting, the local average after embedding is slightly shifted from the original local average.

In the implementation of Chartem, we convert a chart image into the YUV color space, and embed information into Y-component while leaving the UV components unchanged.

For each foreground component, we secure a buffer region of a fixed width around the border of the foreground component as a transition region. No data is embedded into any transition region. After embedding block sequences to all available background regions, unused background pixels, such as those in transition regions or background regions too small to embed a block sequence, are assigned values in the same way corresponding to random logical bit values except that Chartem ensures no replication of the start or end block.

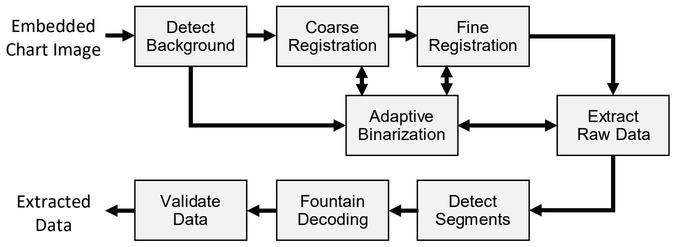


Fig. 4. A flowchart of the data extraction process.

### 3.2 Chartem Extractor

Fig. 4 shows a flowchart of the data extraction process: the extractor detects background regions in a chart image, locates coarse and fine marks to execute coarse and fine registrations of the chart image, locates pairs of start and end blocks to identify block sequences and extracts bits from their data blocks, detects and validates each segment, performs fountain decoding, and validates the extracted data. To facilitate detection of coarse and fine marks and bit extraction, adaptive binarization is used to enhance embedding patterns. These steps will be described in detail in following subsections.

#### 3.2.1 Detecting Background Regions

To detect background regions, the extractor applies the clustering method described in Section 3.1.1 to cluster pixels inside a sliding square window, typically of size in range $[31, 51]$, and then slides the window to cluster new pixels coming into the window. After clustering, it counts boundary pixels for each pair of clusters. If the maximum count normalized by the image size is above a preset threshold, we combine the two clusters of the pair into a single cluster and take it as a candidate for background. This occurs when the embedding weight is so large that pixels representing 0 and 1 are classified into two clusters. Since pixels carrying 0 and 1 interleave with each other, their clusters have significantly more boundary pixels than usual. The extractor then determines a cluster as the background and the remaining clusters as the foreground based on the cluster's size, spatial shape and location in the image, and distances to other clusters.

While embedder's background detection tries to exclude foreground pixels from the detected background to avoid touching foreground pixels during embedding, the extractor's background detection allows some foreground pixels in the detected background to avoid missing any embedded data. These foreground pixels and their impact will be removed in subsequent procedures.

#### 3.2.2 Adaptive Binarization

Binarization is needed in detecting coarse and fine marks and in extracting embedded bits. Chartem adopts an adaptive binarization scheme for potential lighting variation: It collects local distributions of background pixels to determine the size of an adapting window, with background pixels inside such an adaptive window being able to be considered as quasi-static. Then Chartem moves the window over the image: the window cannot cross any boundary of a large foreground region but can contain small foreground regions. At each position, Chartem collects the histogram of background pixels inside the window, removes pixels whose values significantly outside the expected range of background pixels, and applies the mode method [22] to determine a threshold, which is robust to foreground pixels not excluded yet as long as their ratio to the background pixels in the window is small. The threshold is used to binarize the background pixels at the nominal center of the window, i.e., the center of the window at the position it should be located if there were no large foreground regions in the chart image.

#### 3.2.3 Detection of Marks and Image Registration

Coarse marks and fine marks are detected in the same manner. After binarization of pixels in background regions, Chartem scans background pixels to search for patterns that each matches the ratio of the mark to be detected both horizontally and vertically across the center block of the mark, with the outermost ring of the pattern, used as a guarding buffer, excluded. For the coarse and fine marks shown in Fig. 1, the ideal ratio to match is 1:1:3:1:1 for the coarse mark and 1:1:1:1:1 for the fine mark. For each found pattern, Chartem checks if the exterior shape of each layer of the pattern is nearly a parallelogram. If they are, Chartem determines that the pattern is a mark to be detected.

Chartem detects coarse marks first. The center of an original coarse mark is black. Chartem first uses centers of detected coarse marks to determine black and white values of binarization. Then it uses detected coarse marks to roughly register the received chart image. Since coarse marks are arranged at corners of a rectangle at embedding, the detected coarse marks are used to determine a perspective transform to convert the received chart image into a rectangular shape. The horizontal and vertical scales are then estimated from each converted coarse mark, and their averages over detected coarse marks are calculated. The averaged horizontal and vertical scales are then combined with the perspective transform just applied to convert the received chart image into a chart image roughly like the original image.

Chartem then detects fine marks on the roughly registered chart image, determines the actual distances among detected fine marks, and uses them to evaluate a more accurate perspective transform to convert the received chart image into a chart image more accurately like the original image. To facilitate data extraction to be described next, Chartem actually registers a received image into a chart image of $k$ times the size of the original image, i.e., one original pixel is equivalent to $k \times k$ pixels in the registered image.

### 3.2.4 Data Extraction

After the fine registration, each logical bit corresponds roughly $kp \times kp$ pixels. Chartem first applies the adaptive binarization to convert background pixels of the registered image to bipolar -1 and 1. It then uses a template of $kp \times kp$ pixels, each of of value $1/(kp)^2$, as a matched filter to scan all background pixels. In an ideal case, the matched filter produces 1 or -1, corresponding to logical bit 1 and 0, respectively, when the template is aligned with a block of pixels that represents a logical bit, and the matched filter's output is a maximum (or minimum) along one direction, either horizontal or vertical direction, if the logical bit is 1 (or 0) and its two neighboring logical bits on both sides along the direction are both 0 (or 1). These facts are exploited to detect values of logical bits and align them horizontally and vertically.

More specifically, Chartem applies a preset threshold to find all locations whose absolute value of the matched filter output is larger than the threshold. These locations are candidates of logical bit blocks. Chartem then locate extremums (maximums or minimums) along both directions and also along a single direction. To determine rows of logical bits, Chartem locates rows with the number of extremums along both directions and along the vertical direction above a preset threshold. These rows are determined to be rows of logical bits. Chartem then extends from these determined rows to determine other rows of logical bits, based on the fact that the vertical distance of a row is about $rp$ pixels, fine-tuned with the locations of found extremums along both directions and along the vertical direction close to the row. Chartem determines locations of logical bits in each row in a similar way.

At the end of the above process, all logical bits are determined in background regions of the chart image. Then Chartem applies the patterns of start and end blocks to scan these logical bits to determine potential locations of start blocks and end blocks, and determine each matching pair of start block and end block that satisfies the conditions at embedding. Each pair determines a block sequence, wherein data blocks are determined, and raw bits are extracted.

### 3.2.5 Unpacking Raw Data

At the end of the last step, a stream of raw bits is obtained. Chartem then applies the marker of a segment header as a matched filter to scan the raw bit stream to locate positions whose output is above a preset threshold. These positions are potential locations of segments. For each potential segment, Chartem applies error correction to decode the segment and then checks if CRC is correct. If both are successful, Chartem determines that a segment is found.

Once all segments are determined, Chartem checks segments with the same segment ID. If two segments with the same segment ID have conflict, the one with the worse match of the header marker is dropped. Then Chartem determines the largest segment ID of data segments and the smallest segment ID of fountain segments to determine a potential range for the number of data segments. Chartem tries each value in the range to fountain-decode the payload data from survived segments. If the decoding is successful, the embedded data is successfully extracted, and the prefix length is used to determine the size of the input data. The extracted input data is then decompressed and output.

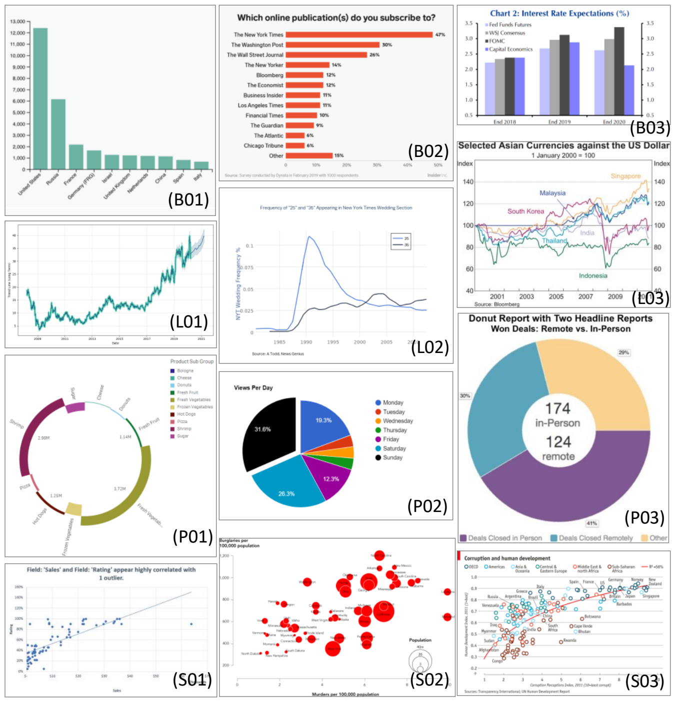


Fig. 5. Chart images used in our user study and experiments, including bar charts (B01-B03), line charts (L01-L03), pie charts (P01-P03), and scatter plots (S01-S03).

## 4 Evaluation

In our evaluation, we first carried out a user study to understand user perception of embedded patterns with different weights. We then conducted experiments to assess Chartem's performance on embedding capacity, extraction accuracy, and execution time. In addition, we present a set of example results as supplementary material, which demonstrate that Chartem can support a variety types of chart designs.

### 4.1 User Study

Chartem embeds data into background regions of a chart image by adjusting background pixel values to form certain patterns to carry desired information. As described in Section 3.1.5, Chartem uses a parameter, *weight*, to determine the value Chartem moves a background pixel from the local average to carry information. Weight determines the distortion our data embedding brings to a chart image or equivalently the visibility of our embedded patterns. The higher the weight is, the more visible the embedded patterns are to humans, and at the same time the more robust of the embedding. We conducted a user study to understand human's tolerance of embedding distortions and their impact on aesthetics.

We recruited 22 participants (12 males and 10 females, 21-62 years old, average age = 30.5) from a technology company and a university. The participants included undergraduate and graduate students, professors, data analysts, researchers, program managers, software engineers and salespersons. They were all general users who had more or less experience of reading chart images to understand data in their daily work and study. None of the participants reported vision impairment in viewing the content on chart images.

### 4.1.1 Stimuli and Procedure

We prepared a set of 12 chart images shown in Fig. 5 for this user study. These chart images were selected from the Internet to have a variety of chart designs. Specifically, they are of four chart types: bar charts, pie charts, line charts, and scatter plots. We consider these chart types because they are the most frequently used ones in real world [9]. For each chart type, we chose three chart images with different sizes and chart styles. For example, we selected both vertical and horizontal bar charts, included donut charts in addition to normal pie charts, and covered not only single-series line charts but also multiple-series ones.

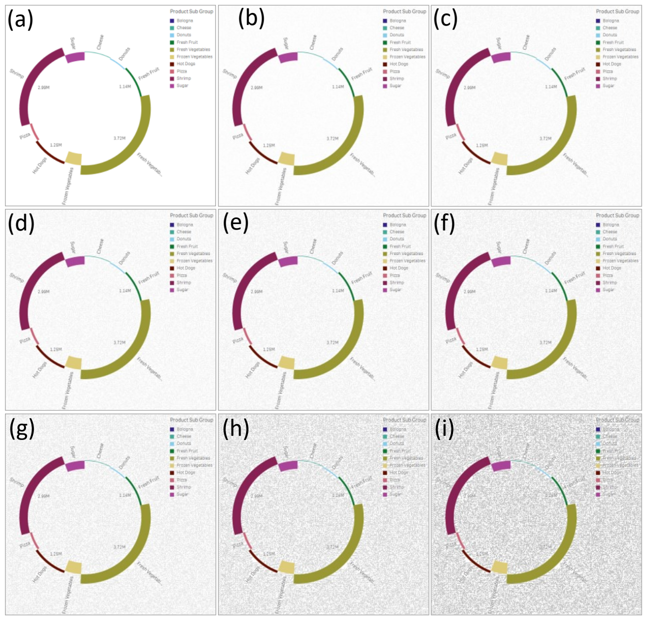


Fig. 6. Embedding results with different weights. (a): original chart image; (b)-(i): data-embedded chart images with weight = 6, 11, 17, 22, 27, 33, 63, and 93, respectively.

For each chart, we applied Chartem to create eight embedded chart images, each embedded with the same data but using a different embedding weight, as shown in Fig. 6 for a test pie chart. Specifically, we selected eight levels of weight (i.e., 6, 11, 17, 22, 27, 33, 63, and 93). These weights were selected using the following criterion: an added weight should have perceptual difference from its adjacent weight when examined closely. All embedded chart images, along with their original images, are included as supplemental material.

Participants performed 12 trials. In each trial, participants were first presented an original chart image, followed by eight embedded chart images with different embedding weights. We asked participants to rate each embedded chart image according to how much the embedding patterns on background impact the overall aesthetic of the chart. Participants responded using a 5-point Likert-scale ranging from "High Impact" to "No Impact At All". To avoid potential bias, the eight embedded chart images were shown in a random order.

### 4.1.2 User Study Results

We received a total of 2,112 ratings (22 *participants* $\times$ 12 *charts* $\times$ 8 *embedding wights*). Fig. 7 shows 95% confidence intervals of mean ratings for different chart types at each of the 8 embedding weights.

When weight increases, we observe a decreasing trend of ratings for all the chart types, which verifies our hypothesis that a higher weight leads to a lower acceptance by users. Specifically, the two highest weights (i.e., 93 and 63) are not acceptable by participants with ratings about 2.0 or below, while the three lowest weights (i.e., 6, 11, and 17) all receive a rating above 3.5, indicating that they are well accepted by participants. The other three weights (i.e., 22, 27, and 33) sit in the marginal zone. The ratings on a same weight vary slightly with different chart types. Generally, the ratings for pie and bar charts are higher than line and scatter charts. This rating difference can be

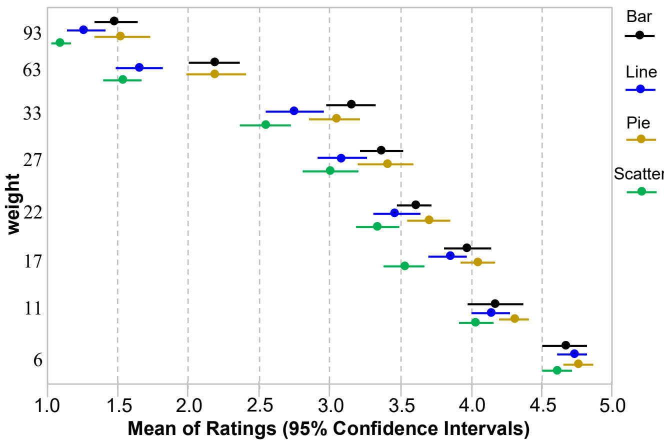


Fig. 7. Mean ratings and 95% confidence intervals per embedding wight and per chart type, calculated via bootstrap (1 = High Impact; 2 = Impact; 3 = Neural; 4 = Not Impact; 5 = Not Impact At All).

explained by the distinct characteristics of different chart types: pie and bar charts typically comprise large foreground components that quickly attract human's attention and are easier to understand, resulting in less attention paid to the background when they are viewed. Line charts and scatter plots, on the other hand, typically comprise smaller and scattered foreground components that interleave much more extensively with the background. A reader generally needs to pay more attention to identify foreground components and understand the content of a chart of these types, making embedding patterns more distractive.

The rating results show that when an appropriate weight is applied, embedding distortions are totally acceptable to users, and do not impact their effectiveness of reading chart images. Based on the results, we further adopt 17, the highest among all acceptable weights, as the default weight value used for Chartem data embedding. With this setting, we hope to achieve a good balance that embedded patterns are easy to detect by machines while remaining non-intrusive to readers.

## 4.2 Evaluation of Robustness, Capacity, and Runtime

We have implemented Chartem in C++ based on OpenCV [18] to evaluate the robustness, capacity, and runtime on the same set of chart images used for the user study. Although these test charts all have a popular white background, Chartem works on any chart with a flat background of any color distinctly different from foreground components. We include embedding examples of chart images with color background in the supplemental material. The results and conclusions obtained in this subsection are generally valid.

In our experiments, the default weight obtained from the user study described in Section 4.1.2 was used, and the block size for each logic bit described in Section 3.1.5 was set to $2 \times 2$, i.e., $p = 2$. In addition, the header marker was set to 16 bits, Wirehair [37] was used as fountain codes to recover data segments from extracted segments, and the Reed-Solomon error correction [24] was selected for error correction within a segment, with 5 bits for a symbol, 21 data symbols, and 10 parity symbols. With this setting, the error correction can correct 5 erroneous symbols or 10 erasure symbols, payload is of 11 bytes (i.e. $21 \times 5 - 8 - 8 - 1 = 88$ bits), a segment has 171 bits (i.e., $21 \times 5 + 10 \times 5 + 16 = 171$), and the combination of ID and CRC is 16 bits in total. If we use 8 bits for segment ID and 8 bits for CRC, Chartem supports 256 distinct segments, and the maximum number of bits for input data (after compression) is thus 2814 ($= 256 \times 11 - 2$) bytes. If larger input data needs to be supported, we can increase the size of segment ID, at the cost of reduced error detection capacity by CRC. For example, if we use 10 bits for segment ID and thus 6 bits for CRC, 1024 distinct segments can be supported, and input data in this case can be up to 56318 bytes.

### 4.2.1 Robustness

We first conducted experiments to evaluate Chartem's robustness after typical operations on chart images, including scaling, screenshot,

Table 1. Precision and recall in percentage (%) of test chart images (Fig. 5) after randomly scaling up and then screenshotting (S+S) and rotations

| Chart | S+S | | Rotating 30° | | Rotating −30° | |
|---|---|---|---|---|---|---|
| | Precision | Recall | Precision | Recall | Precision | Recall |
| B01 | 100.0 | 99.96 | 99.73 | 98.33 | 99.77 | 99.77 |
| B02 | 100.0 | 99.97 | 98.25 | 94.21 | 99.72 | 97.03 |
| B03 | 100.0 | 99.51 | 99.67 | 99.81 | 99.78 | 99.78 |
| L01 | 100.0 | 99.96 | 99.59 | 99.60 | 99.71 | 99.71 |
| L02 | 100.0 | 99.10 | 99.66 | 99.07 | 99.67 | 99.60 |
| L03 | 100.0 | 99.36 | 99.39 | 98.67 | 99.34 | 99.30 |
| P01 | 100.0 | 99.48 | 99.64 | 98.89 | 99.63 | 99.63 |
| P02 | 100.0 | 99.94 | 99.72 | 99.75 | 99.73 | 99.73 |
| P03 | 100.0 | 100.0 | 99.63 | 99.63 | 99.66 | 99.66 |
| S01 | 100.0 | 99.98 | 99.27 | 99.27 | 99.35 | 99.35 |
| S02 | 100.0 | 99.99 | 99.56 | 99.16 | 99.41 | 95.88 |
| S03 | 100.0 | 99.90 | 99.60 | 99.60 | 98.41 | 99.48 |

Table 2. Precision and recall in percentage (%) of test chart images (Fig. 5) after JPEG compression with different quality-factor values

| Chart | 80 (default) | | 70 | | 60 | |
|---|---|---|---|---|---|---|
| | Precision | Recall | Precision | Recall | Precision | Recall |
| B01 | 99.16 | 97.18 | 97.30 | 92.29 | 95.07 | 88.89 |
| B02 | 99.18 | 98.00 | 97.92 | 97.22 | 95.96 | 86.91 |
| B03 | 99.10 | 95.09 | 97.08 | 91.88 | 94.97 | 81.58 |
| L01 | 99.19 | 98.21 | 97.39 | 91.20 | 95.31 | 81.08 |
| L02 | 99.24 | 97.66 | 97.56 | 92.58 | 95.22 | 81.59 |
| L03 | 99.20 | 97.38 | 97.66 | 87.15 | 95.74 | 78.35 |
| P01 | 99.23 | 99.07 | 97.76 | 94.02 | 95.66 | 84.41 |
| P02 | 99.28 | 98.92 | 97.66 | 89.71 | 95.87 | 84.85 |
| P03 | 99.20 | 94.26 | 97.48 | 93.83 | 95.14 | 87.60 |
| S01 | 99.06 | 98.41 | 97.35 | 91.94 | 95.96 | 83.67 |
| S02 | 99.25 | 96.59 | 97.56 | 92.05 | 98.25 | 82.36 |
| S03 | 99.12 | 96.43 | 97.85 | 92.72 | 95.97 | 83.13 |

rotating, JPEG compression, and brightness variations. In these experiments, we generated random bits to embed without using any error or erasure correction or data packaging (i.e., all 171 bits in each segment were randomly generated) and then compared extracted bits with the embedded bits to determine correctly extracted bits. We use recall and precision to measure Chartem's robustness. Recall is defined as the number of correctly extracted bits divided by the total number of embedded bits. Precision is defined as the number of correctly extracted bits divided by the total number of extracted bits. The total number of bits embedded into each chart image is listed as raw capacity in Table 3 and will be described in Section 4.2.2.

A data-embedded chart image was first saved into the PNG format. Then we extracted the embedded bits from the saved image. This was to test the robustness when a data-embedded chart image has not gone through any distortion yet. All the test chart images got 100.0% for both precision and recall.

The next experiment was to randomly scale up a data-embedded image, screenshot the scaled image using Snipping Tool in Windows [17], and extract the embedded bits from the captured image. This was to mimic a typical process in which a user captures a digitally published chart image, which may be scaled during the capturing process or after being published. Columns 2 and 3 of Table 1 show the obtained precision and recall for the test images. They all have 100.0% precision, and their recalls are close to 100.0%.

To test robustness against rotations, an image was rotated at an angle either anti-clockwise (a positive angle) or clockwise (a negative angle), displayed on a screen, and screenshot. Extraction was then applied to the screenshot image. Columns 4 to 7 of Table 1 show the results for each test chart image after rotating $\pm 30°$. Both precision and recall are close to 100.0% for each test chart image.

To test robustness against JPEG compression, we used popular image viewer software IrfanView [27] to convert an image into a JPEG compressed image at different quality-factor values. Table 2 shows the resulting precision and recall for each test chart image after JPEG compression with the quality factor set to 80 (IrfanView's default value), 70, and 60. We can see from the table that the precision remains at about 95% or above while the recall decreases to around 80% or below when the JPEG compression's quality factor is lowered from the default 80 to 60. If the block size of a logic bit is increased from the current $2 \times 2$ to $4 \times 4$, at the cost of reduced capacity, the precision and recall are both above 90% for each test chart image except images L03 and S03 even when the quality factor decreases to 25. At block size $4 \times 4$, images L03 and S03 cannot embed any data since neither one has a sufficiently large background region to embed a single block sequence.

To test robustness against brightness variations, we conducted two experiments. In one experiment, we linearly compressed pixel values towards either 0 or 255 to leave enough room to respectively add 140 to or subtract 140 from each pixel to mimic brightening or darkening an image. We got 100.0% for both precision and recall for all the test chart images. In the other experiments, we compressed pixel values in the same way to leave a room of 140 to add to or subtract from each pixel, and then adjusted the value to add to or subtract from each pixel in a linear manner along either horizontal or vertical direction such that one side was 0 and the other side was 140. This was to mimic gradual brightness changes. All the test chart images got 100.0% for both precision and recall except S01 with 99.99% precision and 99.75% recall when the value subtracted from each pixel changed linearly along the vertical direction from 0 at the top and 140 at the bottom, and S02 with 100.0% precision and 98.58% recall when the value added to each pixel changed linearly along the vertical direction from 0 at the top and 140 at the bottom.

Table 3. The image size, background ratio, and embedding capacity for each test chart image (Fig. 5) (where S+S means randomly scaling up followed by screenshotting and JPEG is of the default quality)

| Chart | Size | Backgrd Ratio | Capacity | | |
|---|---|---|---|---|---|
| | | | Raw | Input Data (bytes) | |
| | (pixels) | (%) | (bits) | S+S | JPEG |
| B01 | $480 \times 480$ | 84.06 | 30911 | 1978 | 1945 |
| B02 | $750 \times 563$ | 81.42 | 46375 | 2979 | 2946 |
| B03 | $791 \times 444$ | 62.43 | 24871 | 1560 | 1505 |
| L01 | $1000 \times 600$ | 89.78 | 60023 | 3859 | 3738 |
| L02 | $700 \times 525$ | 94.44 | 46487 | 2935 | 2649 |
| L03 | $600 \times 467$ | 81.19 | 21575 | 1351 | 1307 |
| P01 | $619 \times 591$ | 90.54 | 53879 | 3430 | 3155 |
| P02 | $805 \times 511$ | 71.54 | 51511 | 3309 | 2968 |
| P03 | $721 \times 786$ | 46.88 | 34767 | 2231 | 2187 |
| S01 | $674 \times 424$ | 89.52 | 22383 | 1428 | 1406 |
| S02 | $900 \times 604$ | 88.48 | 75215 | 4827 | 4596 |
| S03 | $595 \times 404$ | 77.55 | 10879 | 691 | 647 |

### 4.2.2 Embedding Capacity

Another important performance metric is the amount of arbitrary binary data that can be embedded into a chart image, i.e., embedding capacity, which is inversely related to robustness studied in Section 4.2.1: increasing robustness generally reduces capacity, and vice versa. Embedding capacity of a chart image depends on the size and the distribution of foreground components of a host chart image. Table 3 shows the size, background ratio defined as the total number of background pixels divided by the image size, raw capacity, and input data capacity for scaling up and then screenshotting and JPEG at the default quality for each test chart image. *Raw capacity* is the total number of bits of all segments, including header and parity bits, embedded into a chart image, while *input data capacity* is the maximum number of bytes of arbitrary input data that can still be correctly extracted after the targeted processing. For the specific setting of the experiments described at the beginning of Section 4.2, a segment contains 171 raw bits but only 11 bytes of payload for input data. The latter is much smaller than the former due to error correction and header information of a segment.

The capacity in Table 3 is for arbitrary binary input data, which is after compressing user data in practical applications. The actual amount of user data that can be embedded into a chart image depends on the compressibility of the user data. For text input, lossless compression can typically reduce to half of the original size, and thus the amount of data to be embedded would be twice the capacity shown in Table 3.

As a rule of thumb, the larger the size of a chart image, the more data the chart image can host. For chart images of the same size, the higher ratio of background regions to the image size, the more embedded data. Since a block sequence has to be embedded into a continuous background region and a block sequence has a minimum of 3 blocks, start and end blocks and at least one data block, a chart image with large background regions can embed more data than a chart image with scattered small background regions when they have the same image size and background ratio.

Table 4. Execution time (s) for embedding: overall and major modules

| Chart | Backgrd Detection | Sync Marks | Data | Overall |
|---|---|---|---|---|
| B01 | 0.128 | 0.070 | 0.062 | 0.276 |
| B02 | 0.258 | 0.151 | 0.120 | 0.552 |
| B03 | 0.209 | 0.105 | 0.084 | 0.421 |
| L01 | 0.317 | 0.142 | 0.144 | 0.636 |
| L02 | 0.357 | 0.101 | 0.092 | 0.572 |
| L03 | 0.324 | 0.103 | 0.065 | 0.519 |
| P01 | 0.276 | 0.079 | 0.097 | 0.476 |
| P02 | 0.355 | 0.067 | 0.110 | 0.558 |
| P03 | 0.328 | 0.174 | 0.134 | 0.670 |
| S01 | 0.306 | 0.066 | 0.070 | 0.463 |
| S02 | 0.451 | 0.124 | 0.153 | 0.758 |
| S03 | 0.239 | 0.097 | 0.050 | 0.408 |

Table 5. Execution time (s) of overall and major modules for extracting data from chart images after scaling up by 20% and then screenshotting

| Chart | Backgrd Detection | Binarization | Sync Marks | Data | Overall |
|---|---|---|---|---|---|
| B01 | 0.068 | 0.337 | 0.352 | 0.241 | 1.020 |
| B02 | 0.243 | 0.611 | 0.595 | 0.396 | 1.876 |
| B03 | 0.121 | 0.382 | 0.348 | 0.300 | 1.174 |
| L01 | 0.172 | 0.925 | 0.900 | 0.635 | 2.666 |
| L02 | 0.278 | 0.597 | 0.675 | 0.392 | 1.968 |
| L03 | 0.421 | 0.410 | 0.413 | 0.257 | 1.529 |
| P01 | 0.109 | 0.562 | 0.627 | 0.392 | 1.714 |
| P02 | 0.249 | 0.502 | 0.553 | 0.385 | 1.716 |
| P03 | 0.143 | 0.354 | 0.335 | 0.433 | 1.294 |
| S01 | 0.334 | 0.454 | 0.388 | 0.296 | 1.498 |
| S02 | 0.402 | 0.832 | 0.902 | 0.587 | 2.754 |
| S03 | 0.435 | 0.353 | 0.217 | 0.207 | 1.234 |

#### 4.2.3 Runtime

To measure runtime, we ran Chartem with a single thread on an ASUS FL8000 laptop with Intel i7-8550U CPU @1.80GHz and 8GB memory running 64-bit Windows 10. Table 4 and Table 5 show the obtained overall runtime and its breakdown by major modules for both embedding and extraction, respectively. The second column in both tables is the runtime for background detection. In Table 4, the third column is the runtime for finding an embedding rectangle and embedding coarse and fine synchronization marks, and the fourth column for packaging and embedding data. The overall embedding time ranges from 0.276s to 0.758s for the test chart images. In Table 5, the third column is the runtime for the adaptive binarization, the fourth column for detecting coarse and fine synchronization marks and registering the image, and the fifth column for extracting and unpacking data. The overall extraction time ranges from 1.020s to 2.754s.

For both embedding and extraction, an image of a larger size and with more background pixels generally has a longer overall runtime. We note that the current implementation has not been optimized for execution time. There should be a significant room to speed up.

## 5 Sample Applications

In this section, we demonstrate three sample applications that leverage Chartem to create and consume charts with embedded information.

### 5.1 Chart Creation in Excel

To take advantage of Chartem's ability to embed and extract information, the first step is to build a convenient tool to help users embed information into normal charts. One obvious choice is to build a standalone tool to ask users to directly provide a chart image and embed information into it. However, we believe this is not ideal in terms of user experience, as users need to leverage a proprietary tool. Instead, we aim to integrate Chartem smoothly into the workflow of general users. As a result, we have built an Excel add-in for Chartem, as Excel is a powerful and widely used platform for analyzing data and creating chart visualizations.

Specifically, users may follow their normal workflow to analyze data and create chart visualizations accordingly with all the built-in functions in Excel. Once a chart is created (Fig. 8(a1)), users can directly click the *Chartem* button in the ribbon area. Then a side panel (Fig. 8(a2)) will appear to help users embed information into the chart. In this prototype application, we allow two types of information, both are optional. The first one is the data table itself. Since Excel maintains the data model that drives the chart visualization, we can directly collect the data table from the model, instead of parsing the chart image. The second one is a textual description. We allow users to directly provide it in the side panel. Finally, we allow users to customize the embedding weight, although a default value is provided. Once users complete the configuration, they can click the *Embed* button, and an embedded version of the chart is created and previewed in a pop-up dialog. Then, users can either save the chart as an image to the file system or copy it to the clipboard for use in other applications.

### 5.2 Customize Charts in PowerPoint

By design, PowerPoint is able to host charts generated by Excel. By doing so, backend chart models can be maintained and used to support a wide range of follow-up actions. For example, users can directly revise the backend data, so that the chart visualization can be updated automatically. In addition, users can also change the chart type or style to make it more consistent with the theme of presentation. However, in many cases when the chart visualization is imported as an image, all these possibilities are lost.

In the second sample application, we demonstrate how to leverage embedded information to empower chart images with the same flexibility in PowerPoint. Specifically, we have built a PowerPoint add-in to convert a chart image generated by the previous Excel add-in to an active chart object. For example, users can directly drag-and-drop a chart image into PowerPoint (Fig. 8(b1)). Then, to further customize the chart visualization, they can simply click the *Convert to Chart* button. Our backend service first tries to recover all essential information from the image, such as the backend data table and chart configurations. If all the information exist, our add-in will replace the inserted image with its equivalent of chart object (Fig. 8(b2)), so that users can take advantage of the built-in features to freely customize the chart as needed.

### 5.3 Voice-Over on Mobile Phones

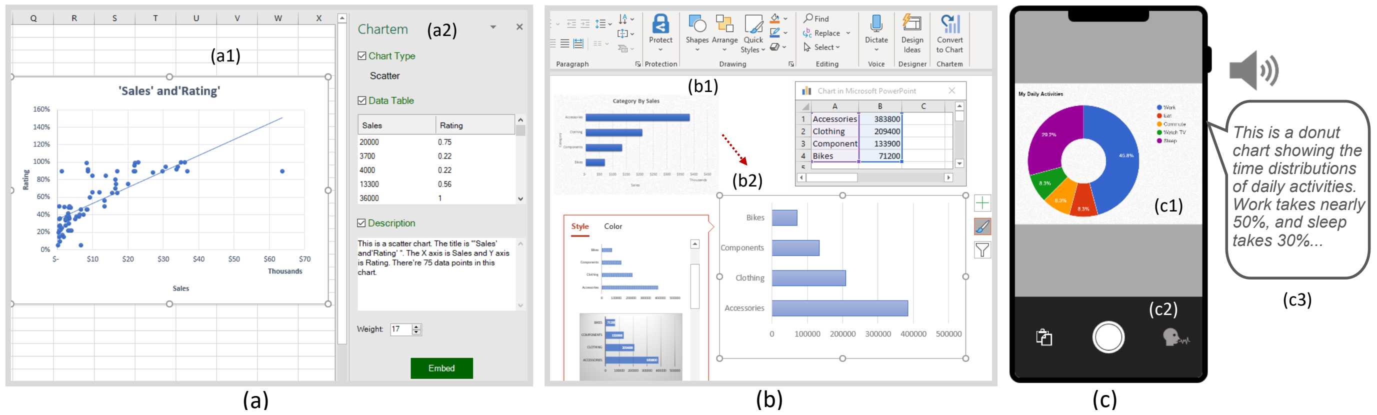
During the creation process, we allow users to embed a free-form text into a chart image, which can be used to serve different purposes. For example, we can integrate additional description to elaborate the chart or essential description to help visually impaired users.

A voice-over mobile app is illustrated in Fig. 8(c). In this hypothetical scenario, users may use the camera on a mobile phone to scan a chart image (Fig. 8(c1)). Then the mobile app will try to extract the textual information embedded in the image, and use a text-to-speech program to convert the description to an audio clip and play it back (Fig. 8(c2)). Currently, a desktop version of the application is implemented, in which a chart image is loaded as a image file instead of captured using cameras. However, we believe it is promising to directly recover embedded information using a camera and a mobile app.

## 6 Discussion

### 6.1 Machine Friendly Charts

As chart data is accumulating rapidly on the Internet, it has become an important topic for machines to interpret chart images. There are several intriguing motivations behind this idea. For example, users may need to reprocess the data behind a certain chart, restyle or index a chart, etc. However, since charts are originally designed to be read by humans, they are not easy for machines to interpret. Many sophisticated approaches have been proposed to involve computer vision and machine learning techniques. Although they have achieved promising preliminary results, there are still a lot of challenges to overcome in terms of robustness and accuracy.


(a1)
'Sales' and'Rating'
Sales
Rating
(a2)
(a)
(b1)
(b2)
(b)
(c1)
(c2)
(c3)
(c)
This is a donut chart showing the time distributions of daily activities. Work takes nearly 50%, and sleep takes 30%...

Chartem

☑ Chart Type

Scatter

☑ Data Table

| Sales | Rating |
|---|---|
| 20000 | 0.75 |
| 3700 | 0.22 |
| 4000 | 0.22 |
| 13300 | 0.56 |
| 36000 | 1 |

☑ Description

This is a scatter chart. The title is "'Sales' and'Rating' ". The X axis is Sales and Y axis is Rating. There're 75 data points in this chart.

Weight: 17

Embed

Chart in Microsoft PowerPoint

|  | A | B | C |
|---|---|---|---|
| 1 | Accessories | 383800 |  |
| 2 | Clothing | 209400 |  |
| 3 | Component | 133900 |  |
| 4 | Bikes | 71200 |  |

Fig. 8. Sample applications: (a) A Excel add-in to help users embed information into typical charts; (b) A PowerPoint add-in to help users convert chart images into chart objects; (c) A voice-over mobile app that reads embedded information.

In this work, we try to address this issue from the root, i.e., directly creating charts that are friendly to both humans and machines. Specifically, we do not aim to change human's reading experience. People can use charts in any ways that they are used to. On the other hand, we piggyback information that can be efficiently and accurately consumed by machines on top of chart images.

There are two unique advantages of this approach. First, since the extracted information can be guaranteed completeness and accuracy, it is more robust and direct compared with previous machine-learning-based approaches. Second, since machines do not rely on chart visuals to collect any information, this method works for different chart types. Therefore, it has the potential to be a new form of charts to replace existing chart visualizations, as it maintains the human experience while providing opportunities for more applications.

## 6.2 Opportunities for New Applications

As discussed in Section 1 and Section 2.2, there are several ways to embed information. For example, we may directly overlay information (e.g., QR code) on a chart, insert information into image files, encode information using the frequency domain, etc. Among all these candidates, we choose to embed information into the background area of a chart for two reasons. First, we aim to minimize interruptions to the reading experience. According to the user study reported in Section 4.1.2, most participants felt comfortable when reading the charts generated by our method, since chart backgrounds are generally not their foci and our embedding patterns are barely noticeable. Second, the information needs to be associated with chart images instead of files, since charts may be screenshot or saved in different formats.

In addition, since the embedded information is highly customizable, it can provide more flexibility to downstream applications. We have illustrated two examples in Section 5. However, we believe there are much more scenarios that can take advantage of this technique. For example, creators can embed encrypted confidential information into a chart. Then, only authorized users can use a mobile app to scan the chart and provide a password to decrypt the extracted information. We can also use the technique to enable AR-like experiences. For example, users may use a mobile device to see animations by pointing the camera at a chart with proper information embedded, which is a valuable complement to traditional static charts.

However, to make general users benefit from our approach, it is required to vastly distribute the chart embedder and extractor. Ideally, they can be integrated into mainstream software, as demonstrated in Section 5. Otherwise, charts with embedded information simply regress to normal charts without providing any benefits at all. Considering this situation, we believe machine-learning-based approaches are still prevalent and valuable for the foreseeable future.

## 6.3 Limitations

Chartem makes charts accessible to machines, which may greatly expand the scope of applications. At the same time, it is also subject to several restrictions and limitations.

The first limitation is about embedding capacity. Chartem requires a minimum background size to embed information. A valid embedding pattern includes at least three coarse marks, three fine marks, and a block sequence of a minimum of 3 blocks (for data, start, and end, respectively). This requires a minimum background size to embed. When a chart image does not meet this minimum requirement, no data can be embedded at all. In addition, if background regions are too small, it may also fail to embed all desired data. There are several ways to address the insufficient capability issue. For example, it is possible to combine the technique used in [15] by also embedding data at boundaries of foreground components to complement Chartem's background embedding. In addition, if the foreground also contains large areas of solid colors (e.g., in a typical treemap), we can also embed into foreground regions with a control of its embedding noise to make the perceptual quality acceptable for targeted applications. This foreground embedding complements Chartem's background embedding well and can significantly increase the embedding capacity. Finally, it is also possible to store the actual information in the cloud and embed the corresponding url address in the chart image instead. However, this approach requires Internet access when extracting information.

The second limitation is related to information robustness. Chartem embeds a bit by adjusting a block of pixels above or below the local average. Extracting the bit requires estimating the local average. To reduce estimation error, Chartem requires that background should be relatively smooth locally. If background of a chart image is not smooth, the estimated local average may be significantly affected by distortions to local pixels brought by an operation on the chart image such as scaling and thus inaccurate, hence damaging the information robustness. However, charts in the real world may have a much more complex or hostile background. For example, they may have noisy backgrounds or natural images as backgrounds, which are difficult for Chartem to embed information since Chartem is designed for charts with homogeneous backgrounds. For such complex backgrounds, traditional image-data-embedding methods can be adopted to embed information into background regions. Traditional image data embedding complements Chartem well for various backgrounds charts may use.

## 7 Conclusion

We presented a novel solution, Chartem, to unlock information locked in charts, typically published in bitmap images that are unfriendly to machines, and enrich chart applications. Chartem is based on data embedding: chart data and information and/or generic data that enriches user experiences can be embedded into background regions of a chart image. Foreground regions are untouched to ensure a good perceptual quality after embedding yet maintain a large capacity and good robustness. Our user study and performance experiments indicate that data-embedded chart images are well accepted and Chartem is robust with relatively high capacity. We presented several prototype applications to demonstrate the utility of Chartem. In addition to extracting chart data and information to revive a chart image, Chartem opens the door for many more potential applications around chart images.

## REFERENCES

[1] S. Baluja. Hiding images in plain sight: Deep steganography. In *Advances in Neural Information Processing Systems*, pp. 2069–2079, 2017.

[2] Z. Bylinskii, S. Alsheikh, S. Madan, A. Recasens, K. Zhong, H. Pfister, F. Durand, and A. Oliva. Understanding infographics through textual and visual tag prediction. *arXiv preprint arXiv:1709.09215*, 2017.

[3] M. U. Celik, G. Sharma, A. M. Tekalp, and E. Saber. Lossless generalized-lsb data embedding. *IEEE transactions on image processing*, 14(2):253–266, 2005.

[4] A. Cheddad, J. Condell, K. Curran, and P. Mc Kevitt. Digital image steganography: Survey and analysis of current methods. *Signal processing*, 90(3):727–752, 2010.

[5] Z. Chen, Y. Wang, Q. Wang, Y. Wang, and H. Qu. Towards automated infographic design: Deep learning-based auto-extraction of extensible timeline. *IEEE transactions on visualization and computer graphics*, 26(1):917–926, 2019.

[6] J. Choi, S. Jung, D. G. Park, J. Choo, and N. Elmqvist. Visualizing for the non-visual: Enabling the visually impaired to use visualization. In *Computer Graphics Forum*, vol. 38, pp. 249–260. Wiley Online Library, 2019.

[7] M. Cliche, D. Rosenberg, D. Madeka, and C. Yee. Scatteract: Automated extraction of data from scatter plots. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 135–150. Springer, 2017.

[8] J. Fridrich, M. Goljan, and R. Du. Lossless data embedding—new paradigm in digital watermarking. *EURASIP Journal on Advances in Signal Processing*, 2002(2):986842, 2002.

[9] J. Harper and M. Agrawala. Converting basic d3 charts into reusable style templates. *IEEE transactions on visualization and computer graphics*, 24(3):1274–1286, 2017.

[10] D. Jung, W. Kim, H. Song, J.-i. Hwang, B. Lee, B. Kim, and J. Seo. Chartsense: Interactive data extraction from chart images. In *Proceedings of the 2017 chi conference on human factors in computing systems*, pp. 6706–6717, 2017.

[11] K. Kafle, B. Price, S. Cohen, and C. Kanan. Dvqa: Understanding data visualizations via question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5648–5656, 2018.

[12] T.-H. Lan and A. H. Tewfik. A novel high-capacity data-embedding system. *IEEE Transactions on Image Processing*, 15(8):2431–2440, 2006.

[13] M. Luby. LT codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pp. 271–280, 2002.

[14] S. Madan, Z. Bylinskii, M. Tancik, A. Recasens, K. Zhong, S. Alsheikh, H. Pfister, A. Oliva, and F. Durand. Synthetically trained icon proposals for parsing and summarizing infographics. *arXiv preprint arXiv:1807.10441*, 2018.

[15] M. A. Masry. A watermarking algorithm for map and chart images. In *Security, Steganography, and Watermarking of Multimedia Contents VII*, vol. 5681, pp. 495–503. International Society for Optics and Photonics, 2005.

[16] G. G. Méndez, M. A. Nacenta, and S. Vandenheste. ivolver: Interactive visual language for visualization extraction and reconstruction. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 4073–4085, 2016.

[17] Microsoft. Use snipping tool to capture screenshots. `https://support.microsoft.com/en-us/help/13776/windows-10-use-snipping-tool-to-capture-screenshots`, 2019.

[18] OpenCV team. OpenCV. `https://opencv.org/`, 2019.

[19] T. Pevnỳ, T. Filler, and P. Bas. Using high-dimensional image models to perform highly undetectable steganography. In *International Workshop on Information Hiding*, pp. 161–177. Springer, 2010.

[20] J. Poco and J. Heer. Reverse-engineering visualizations: Recovering visual encodings from chart images. In *Computer Graphics Forum*, vol. 36, pp. 353–363. Wiley Online Library, 2017.

[21] J. Poco, A. Mayhua, and J. Heer. Extracting and retargeting color mappings from bitmap images of visualizations. *IEEE transactions on visualization and computer graphics*, 24(1):637–646, 2017.

[22] J. M. Prewitt and M. L. Mendelsohn. The analysis of cell images. *Annals of the New York Academy of Sciences*, 128(3):1035–1053, 1966.

[23] QR Code.com. History of QR code. `https://www.qrcode.com/en/history/`. Last accessed April 28, 2020.

[24] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[25] M. Savva, N. Kong, A. Chhajta, F.-F. Li, M. Agrawala, and J. Heer. Revision: Automated classification, analysis and redesign of chart images. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pp. 393–402, 2011.

[26] N. Siegel, Z. Horvitz, R. Levin, S. Divvala, and A. Farhadi. Figureseer: Parsing result-figures in research papers. In *European Conference on Computer Vision*, pp. 664–680. Springer, 2016.

[27] I. Skiljan. Irfanview graphic viewer, version 4.54. `https://www.irfanview.com/`, 2020.

[28] M. D. Swanson, M. Kobayashi, and A. H. Tewfik. Multimedia data-embedding and watermarking technologies. *Proceedings of the IEEE*, 86(6):1064–1087, 1998.

[29] M. D. Swanson, B. Zhu, B. Chau, and A. H. Tewfik. Multiresolution video watermarking using perceptual models and scene segmentation. In *Proceedings of International Conference on Image Processing*, vol. 2, pp. 558–561. IEEE, 1997.

[30] M. D. Swanson, B. Zhu, B. Chau, and A. H. Tewfik. Object-based transparent video watermarking. In *Proceedings of First Signal Processing Society Workshop on Multimedia Signal Processing*, pp. 369–374. IEEE, 1997.

[31] M. D. Swanson, B. Zhu, and A. H. Tewfik. Robust data hiding for images. In *1996 IEEE Digital Signal Processing Workshop Proceedings*, pp. 37–40. IEEE, 1996.

[32] M. D. Swanson, B. Zhu, and A. H. Tewfik. Transparent robust image watermarking. In *Proceedings of 3rd IEEE International Conference on Image Processing*, vol. 3, pp. 211–214. IEEE, 1996.

[33] M. D. Swanson, B. Zhu, and A. H. Tewfik. Data hiding for video-in-video. In *Proceedings of International Conference on Image Processing*, vol. 2, pp. 676–679. IEEE, 1997.

[34] M. D. Swanson, B. Zhu, and A. H. Tewfik. Audio watermarking and data embedding–current state of the art, challenges and future directions. In *Multimedia and Security Workshop at ACM Multimedia*, vol. 41. Citeseer, 1998.

[35] M. D. Swanson, B. Zhu, A. H. Tewfik, and L. Boney. Robust audio watermarking using perceptual masking. *Signal processing*, 66(3):337–355, 1998.

[36] M. Tancik, B. Mildenhall, and R. Ng. Stegastamp: Invisible hyperlinks in physical photographs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2117–2126, 2020.

[37] C. A. Taylor. Wirehair - fast and portable fountain codes in c. `https://github.com/catid/wirehair/`, 2019.

[38] J. Tian. Reversible data embedding using a difference expansion. *IEEE transactions on circuits and systems for video technology*, 13(8):890–896, 2003.

[39] E. R. Tufte. *The visual display of quantitative information*, vol. 2. Graphics press Cheshire, CT, 2001.

[40] R. G. Van Schyndel, A. Z. Tirkel, and C. F. Osborne. A digital watermark. In *Proceedings of 1st international conference on image processing*, vol. 2, pp. 86–90. IEEE, 1994.

[41] D. Wave. Information technology automatic identification and data capture techniques qr code bar code symbology specification. *International Organization for Standardization, ISO/IEC*, 18004, 2015.

[42] E. Wengrowski and K. Dana. Light field messaging with deep photographic steganography. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1515–1524, 2019.

[43] K. A. Zhang, A. Cuesta-Infante, L. Xu, and K. Veeramachaneni. Steganogan: High capacity image steganography with gans. *arXiv preprint arXiv:1901.03892*, 2019.

[44] J. Zhu, R. Kaplan, J. Johnson, and L. Fei-Fei. Hidden: Hiding data with deep networks. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 657–672, 2018.

[45] W. Zhu, Z. Xiong, and Y.-Q. Zhang. Multiresolution watermarking for images and video. *IEEE transactions on circuits and systems for video technology*, 9(4):545–550, 1999.