

Hiding Sensitive Information Using PDF Steganography

Ryan Klemm and Bo Chen

Department of Computer Science, Michigan Technological University, Michigan, United States
 {rcklemm,bchen}@mtu.edu

Abstract. The use of steganography to transmit secret data is becoming increasingly common in security products and malware today. Despite being extremely popular, PDF files are not often the focus of steganography research, as most applications utilize digital image, audio, and video files as their cover data. However, the PDF file format is promising for usage in medium-capacity steganography applications. In this paper, we present a novel PDF steganography algorithm based upon least-significant bit insertion into the real-valued operands of PDF stream operators. Where prior research has only considered a small subset of these operators, we take an extensive look at all the possible operators defined in the Adobe PDF standard to evaluate their usability in our steganography algorithm. We also provide a case study which embeds malware into a given cover PDF document.

1 Introduction

Steganography is the practice of hiding data inside of a cover media. This cover media is generally a benign file that appears normal in the context of the communication channel that messages are being passed over. The goal of hiding data with steganography is to make it appear as though no hidden information is being transferred at all, so any changes to the cover file must not be obvious to a viewer. The cover media for a steganographic message can be any sort of data, but is most commonly a digital media file such as an image, audio, or video file. This research, however, specifically focuses on hiding sensitive information into the PDF file format, a widely used file format in business, education, and government.

1.1 PDF Steganography

A variety of methods have been developed in order to hide data inside of PDF documents. Three of the major categories of PDF steganography are described in the following sections. This research will primarily focus on using “Operator Value” of the PDF documents for information hiding via steganography.

Data Outside of View. One of the simplest forms of PDF file steganography is placing data in areas of the file in which it will not be rendered when opened by a PDF reader. This could be as simple as placing hidden data within a PDF stream comment or after the `%%EOF` line of the file. More complex strategies include placing hidden data inside an object that is not linked to the main object tree within the PDF – for example, a page object that is not linked into the document’s page catalogue [4]. The main drawback to this steganography method is that it can be easily detected and defeated.

Between-Character, Between-Word, Between-Line. Similar to other forms of text-file steganography, whitespace is a vector for hiding data within PDF files. One proposed strategy is to switch between Unix (`\n`) and Windows (`\r\n`) line breaks within the file to encode one bit per line [9]. Other strategies use zero-width characters between characters or words within the text of the PDF, or change the justification of the text to encode data within the size of the whitespace present on the page [3].

Operator Values. In order to place text, draw vector graphics, and color the PDF document, PDF files utilize operators to define the layout of a document. Many of these operators take floating-point numerical arguments, which can be changed by very small amounts [1], [2]. When these operator arguments (operands) are very slightly changed, almost no visual effects occur within the viewed PDF file [11]. Thus, these slight changes can be used as an effective vector for steganography in PDF documents. Only a small subset of the operators defined by the PDF standard have been studied in existing work in steganography, namely TJ, Tc, Td, and Tw [8].

1.2 Our Research

We have proposed a new strategy for embedding sensitive data into PDF documents through imperceptible modifications to the PDF stream operator values. Compared to existing PDF steganography techniques, ours can achieve a higher carrying capacity and embedding rate, while minimizing the visual impacts to the PDF document. Mostly importantly, our work looks through all the possible operators defined in the Adobe PDF standard, identifying all those unique operators which may be utilized to hide sensitive information in a steganographic manner.

2 Background Information

2.1 Structure of a PDF File

A PDF file can be thought of as a linked tree of objects, each of which is responsible for one piece of the document. For example, a two-page document will have a root object and two page parent objects, each of which is connected to multiple text, image, etc. objects that render the actual visual parts of the page.

In the file itself, these objects are arranged in a list, with links made by objects referencing the numerical IDs of other objects. Each object consists of an ID, a list of key/value properties that define its type and connections to other objects, and then an optional data stream that holds the actual content such as the text or image data. Below is an example of an object holding a text stream, which displays “ABC”:

```
8 0 obj
<<
/Length 72155
>>
stream
BT
/F1 12 Tf
288 720 Td
(ABC) Tj
ET
endstream
endobj
```

This example shows a PDF object with ID 8, enclosed between the keywords `obj` and `endobj`. The stream’s properties are stored in between the ‘<<’ and ‘>>’. Following its properties, the object’s data stream lies between the keywords `stream` and `endstream`. Each line of this stream contains an operator which controls some aspect of the positioning, size, or content of the text displayed [1], [2].

2.2 PDF Stream Operators

The method for hiding information inside PDF documents utilizes the stream operators that control the display of the PDF. These operators consist of a string identifier that is preceded by some number of operands, which can either be strings or numerical. For example, the ‘Tf’ operator defines the font and size for the following text:

/F1 12 Tf (1)

While the number of operands can vary between operators, almost all operators have a defined and consistent number of operands. ‘Tf’ is always preceded by a string that decides the ID of the font to use, and a floating-point precision number that defines the font size. So, the above example denotes the usage of font #1 at 12-point size. The core of the steganography method comes from the floating-point precision of these operands. In this example, the sequence:

/F1 12.01 Tf (2)

is also a valid use of the operator, and produces text that is essentially visually identical, since 0.01 points of font size corresponds to negligible height on a printed page. Therefore, replacing sequence (1) with sequence (2) inside of a PDF document has no discernible effect on the visual appearance of the document [1], [2]. As will be shown later, sequences (1) and (2) are able to represent different parts of a steganographic hidden message.

Note on ISO PDF Standards The PDF file format was formalized as ISO 32000 in 2008, which defined version 1.7 of PDF documents. Most PDF files in circulation today have versions 1.1-1.7. The steganography method and discussion of the PDF file format in this document are based off of the version 1.7 standard [1]. However, in 2020, Adobe & ISO updated the standard to define PDF version 2.0. While this standard adds some extra features to the PDF file format, it is fully backwards-compatible with older PDF versions:

“ISO 32000-1 defines a PDF version matching 1.7. This document [ISO 32000-2] defines a new version of PDF designated 2.0. This document is also suitable for interpretation of files made to conform to any of the previous Adobe PDF specifications 1.0 through 1.7 and ISO 32000-1” [2].

Therefore, all methods described in this document will continue to be viable for steganography in PDF 2.0 files. There will also be a discussion of if any of the changes made to the standard can be used to expand the steganography method.

2.3 PDF Management Software

A variety of software exists to manage PDF files. The primary functionality that is necessary for this paper’s steganography method is the ability to compress and decompress PDF streams. In most generated and distributed PDF files, the object streams are encoded with lossless compression in order to reduce the overall size of the PDF file. However, in order to effectively read and write to the streams, they must be decompressed before being operated on by the steganography code. Once the edits are made, the streams can be compressed once again. The following two sections show how to compress and decompress PDF streams in different Linux tools for PDF file management [6], [7].

pdftk Decompression: `pdftk <in_file> output <out_file> uncompress`

Compression: `pdftk <in_file> output <out_file> compress`

qpdf Decompression: `qpdf <in_file> --stream-data=uncompress <out_file>`

Compression: `qpdf <in_file> --stream-data=compress <out_file>`

3 Related Work

Wang et al. [11] proposed a similar steganographic method to the one that we are proposing. However, they only considered a small subset of the possible operators (Tm, MediaBox). The paper proposed multiple ways to hide data in numerical operands: 1) Adding trailing zeros to all floating-point operands. Thus, to hide the decimal data ‘5’, 5 zeroes would be added to an operand. For example, 12.1 would change to 12.100000. The 5 could then be translated to some binary value. This method is guaranteed to not produce any visual artifacts, since the numerical value of the operands do not change at all. However, it will significantly increase the file size, since each 0 adds 1 character (1 byte) to the file. 2) Adding 2 trailing zeros to a floating-point operand, then appending the numerical value that you wish to encode. For example, to hide the decimal data ‘123’, 12.1 would change to 12.100123. The 123 could then be translated to some binary value. This

method slightly improves the issue of increasing the file size. It has the potential to introduce visual artifacts if used with extremely sensitive operators. However, the two trailing zeroes make the percentage magnitude of the numerical change too small to be visually noticeable in almost any case. Either of these methods could be directly reused on any of the 31 operators that we elaborate in this paper that take floating-point operands.

Sofian et al. [10] proposed a least-significant-bit insertion steganography method on the integer numerical operands to the TJ operator. Considering 98 as a numerical operand, $98_{10} = 01100010_2$. Currently, the least-significant-bit encodes a binary 0. To instead encode a binary 1, the number would be changed to $01100011_2 = 99_{10}$. Thus to encode a 1 instead of a 0, the operand 98 would need to be changed to 99. Since the integer operands in the TJ operator represent thousandths of a unit in text coordinates, a change from 99 to 98 is visually imperceptible [10]. Important to note is that encoding 99 as opposed to 98 does not require any extra file space. Therefore, this method can hide data without increasing the size of the cover PDF. In its current state, this steganography method cannot be generally applied to all 31 operators that this paper is concerned with. While a change such as 98 to 99 may be acceptable for the TJ operator, it may produce visual artifacts if used in a different operator.

4 Our Proposed PDF Steganography Method

We propose a new PDF steganography method by evaluating all the stream operators defined by the PDF document standard, identifying those which may be leveraged to embed sensitive information. Also, the LSB insertion is the most space efficient way to embed data, while adding trailing digits past the decimal point is the most stable way to avoid visual artifacts after steganography.

Which Operators can be Used? To achieve the highest possible capacity with PDF steganography, the exact operands that the method will work on must be defined. The PDF document standard (Version 3.0), published by Adobe, defines 73 different stream operators. Among them, 32 operators (Table 1) accept numerical operators that can be changed with floating-point precision. These 32 operators may be used for PDF steganography, as they take floating-point precision operands. The other 41 operators either have no operands, take string operands, or integer numerical operands that cannot be changed without significantly impacting their visual effect. Therefore, they are very unlikely to be utilized for our purpose.

Bit Embedding. Consider a case where making a change from 98 to 99 may cause visual artifacts. 98 could then instead be represented by 98.0. To do LSB insertion, consider $980_{10} = 001111010100_2$. This currently encodes a binary 0. To instead encode a 1, flip the LSB: $001111010101_2 = 981_{10}$. Then, place the decimal point back to its previous location, to reach 98.1. A change from 98.0 to 98.1 may not cause the same visual artifacts as a change from 98 to 99.

Overall, an LSB insertion of the lowest n bits of a number will change the decimal value of that number by a maximum of $2^n - 1$. Perhaps more relevant to PDF operator steganography is the actual percentage change to a numerical value if a certain bit insertion is applied. There is no closed formula for this, as it will vary on a case-by-case basis depending on the bit pattern of the existing numerical value and the bit pattern of the bits that need to be inserted. This percentage change should roughly correspond to how visually noticeable the changes in operand values are. It is crucial that we know the percentage cutoff for each viable steganography operator. This is the purpose of Section 10.

Once the percentage cutoffs are known, the actual steganography algorithm can be carried out: (Let the current operand value be v , the maximum allowable percentage change be p , the number of bits hidden per operand be n , and the next n bits of the hidden message be S)

1. Calculate O as the integer formed by the digits of v when the decimal point is removed. Keep track of the number of digits before the decimal point, as the decimal point must be returned to that position after the LSB insertion is carried out.
2. If the least significant n bits of O exactly match S , then no change is required. Simply move on to the next relevant operand value. There is a roughly $\frac{1}{2^n}$ chance of this happening for each operand. In this case, n bits of data are hidden without adding any size to the cover file.

Table 1: PDF operators that may be used for PDF steganography

Operator	PDF 2.0 Std. Table	PDF 1.7 Std. Table
c	Table 58	Table 59
v	Table 58	Table 59
y	Table 58	Table 59
l	Table 58	Table 59
m	Table 58	Table 59
re	Table 58	Table 59
cm	Table 56	Table 57
i	Table 56	Table 57
M	Table 56	Table 57
w	Table 56	Table 57
G	Table 73	Table 74
g	Table 73	Table 74
K	Table 73	Table 74
k	Table 73	Table 74
RG	Table 73	Table 74
rg	Table 73	Table 74
sc	Table 73	Table 74
SC	Table 73	Table 74
scn	Table 73	Table 74
SCN	Table 73	Table 74
Tc	Table 103	Table 105
Td	Table 106	Table 108
TD	Table 106	Table 108
Tf	Table 103	Table 105
TL	Table 103	Table 105
Tm	Table 106	Table 108
Ts	Table 103	Table 105
Tw	Table 103	Table 105
Tz	Table 103	Table 105
TJ	Table 107	Table 109
d0	Table 111	Table 113
d1	Table 111	Table 113

3. If the least significant n bits of O do not exactly match S , determine the value of O_S , formed by inserting S into the least significant n bits of O . If $|O_S - O| \leq p \cdot O$, then just replace O with O_S .
4. If $|O_S - O| > p \cdot O$, extend O one more digit past the decimal place, so $O_{ext} = 10 \cdot O$. Repeat steps 3 and 4 as many times as necessary until a small enough percentage change has been made. Note that for each application of step 4, one byte is added to the size of the cover file.
5. Once the final value of O_S is reached, replace the decimal point to form the final operator value.

Using this algorithm, the values of p and n can be easily customized per operator. As long as the recipient knows the value of n used for each operator, they can extract the message. Recommended settings can be found in Table 2. We have further justified the choices made in the ‘Percentage per Operand’ field of Table 2 in Appendix A.

Bit Extraction. Bit extraction is a significantly simpler process than bit embedding. Given an operator value v and the number of embedded bits n , the extraction is straightforward:

1. Form the mask m as the binary string of n 1’s.
2. Calculate O as the integer formed by the digits of v with the decimal point removed.
3. Calculate the next n binary digits of the hidden message by extracting the least significant n bits of O

Table 2: Minimum percentage change necessary for usable operators

Operator	Num (Usable) Operands	Percentage per Operand	Reliability
c	6	1	good
v	4	1	good
y	4	1	good
l	2	0.05	good
m	2	0.05	good
re	4	0.2	good
cm	6	0.1, 0.05	good
i	1	–	good
M	1	–	good
w	1	1	good
G	1	5	good
g	1	5	good
K	4	5	good
k	4	5	good
RG	3	5	good
rg	3	5	good
sc	1, 3, 4	5	good
SC	1, 3, 4	5	good
scn	1, 3, 4	5	good
SCN	1, 3, 4	5	good
Tc	1	1	low
Td	2	2	good
TD	2	2	good
Tf	1	0.5	good
TL	1	2	good
Tm	6	5-10, 2	good
Ts	1	5-10	good
Tw	1	1	low
Tz	1	0.5	good
TJ	0+	15	good
d0	1	1	good
d1	5	1, 1-2	good

Regex for Detecting Operators in a PDF. The most effective way of locating the operator sequences of a PDF document within code is through the use of regex masks. The regex mask for all operators except for TJ is as follows:

`(?:[\d\.\-]+\s+){a, b}op[\\s]`

Where a and b are the minimum and maximum number of operands for the given operator with code op .

The regex mask for operator TJ is as follows:

`\[. + ? \] \s* ? TJ`

The above two regex masks pull out the full operator sequence from the PDF file streams. To grab the individual numerical operands, a further regex operation is necessary. The regex mask for extracting operands from all operators except for TJ is as follows:

`[\d\.\-]+`

The regex mask for extracting operands from operator TJ is as follows:

`[\d\.\-]+(?:![^(\)*\>])(?![^\<]*\>)`

5 Case Study: Leveraging PDF steganography to Hide Malware

To show how to use the proposed PDF steganography method for practical purposes, we have implemented the proposed design in Python (the code is available in [5]) and performed a case study in which we embed a malware sample into a given cover PDF. We start by obtaining our cover PDF and malware sample:

Cover PDF – Adobe PDF Standard Specification, Version 1.7 – Source: https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/PDF32000_2008.pdf

Malware Sample – Petya Ransomware Trojan (Windows) – Source: <https://github.com/ytisf/theZoo/tree/master/malware/Binaries/Trojan.Ransom.Petya>

Information on the malware sample file is shown in the below image:

```
rklemm@rklemm-XPS:~/Downloads/petya-sample $ ls -l trojan-petya
-rw-r--r-- 1 rklemm rklemm 335872 Dec  6 2021 trojan-petya
rklemm@rklemm-XPS:~/Downloads/petya-sample $ file trojan-petya
trojan-petya: PE32 executable (GUI) Intel 80386, for MS Windows
```

Before we can operate on the PDF document, we must decompress the streams using one of the methods discussed in Section 2.4:

```
$ qpdf PDF32000_2008.pdf --stream-data=uncompress cover.pdf
```

This yields the file `cover.pdf`, which has all object stream data in cleartext.

Before attempting to embed the data, it is important to check the actual capacity of the cover PDF document. With our steganography Python script, that is done with the following command:

```
$ python3 full_pdf_steg.py stat cover.pdf
```

Which gives the results shown:

```
[+] Calculating Allowable Message Size
[-] Message Size Calculated
464007 bytes are hideable in this file with current config
```

Our malware sample has a size of 335,872 bytes, which is clearly smaller than the 464,007 byte carrying capacity of the cover PDF. Thus, we can continue on with the process. The command to embed the hidden data within the PDF is the following:

```
$ python3 full_pdf_steg.py embed cover.pdf steg.pdf trojan-petya
```

The results of this command are shown below:

```
[+] Embedding Message
[+] Calculating Allowable Message Size
[-] Message Size Calculated
Size of message: 335872
```

```
Added 1444290 more bytes
[-] Message Embedded
Message successfully embedded
```

After the steganography algorithm finishes processing the PDF file, the changes made in the document are almost imperceptible. To show an example, a page before (left) and after (right) data hiding is shown below:

Contents	Page
Foreword.....	vi
Introduction.....	vii
1 Scope.....	1
2 Conformance.....	1
2.1 General.....	1
2.2 Conforming readers.....	1
2.3 Conforming writers.....	1
2.4 Conforming products.....	2
3 Normative references.....	2
4 Terms and definitions.....	6
5 Notation.....	10
6 Version Designations.....	10
7 Syntax.....	11
7.1 General.....	11
7.2 Lexical Conventions.....	11
7.3 Objects.....	13
7.4 Filters.....	22
7.5 File Structure.....	38
7.6 Encryption.....	55
7.7 Document Structure.....	70
7.8 Content Streams and Resources.....	81
7.9 Common Data Structures.....	84
7.10 Functions.....	92
7.11 File Specifications.....	99
7.12 Extensions Dictionary.....	108
8 Graphics.....	110
8.1 General.....	110
8.2 Graphics Objects.....	110
8.3 Coordinate Systems.....	114
8.4 Graphics State.....	121
8.5 Path Construction and Painting.....	131
8.6 Colour Spaces.....	138
8.7 Patterns.....	173
8.8 External Objects.....	201
8.9 Images.....	203
8.10 Form XObjects.....	217
8.11 Optional Content.....	222
9 Text.....	237
9.1 General.....	237
9.2 Organization and Use of Fonts.....	237
9.3 Text State Parameters and Operators.....	243
9.4 Text Objects.....	248
9.5 Introduction to Font Data Structures.....	253
9.6 Simple Fonts.....	254
9.7 Composite Fonts.....	267
9.8 Font Descriptors.....	281
9.9 Embedded Font Programs.....	288
9.10 Extraction of Text Content.....	292
10 Rendering.....	296

© Adobe Systems Incorporated 2008 – All rights reserved

iii

Contents	Page
Foreword.....	vi
Introduction.....	vii
1 Scope.....	1
2 Conformance.....	1
2.1 General.....	1
2.2 Conforming readers.....	1
2.3 Conforming writers.....	1
2.4 Conforming products.....	2
3 Normative references.....	2
4 Terms and definitions.....	6
5 Notation.....	10
6 Version Designations.....	10
7 Syntax.....	11
7.1 General.....	11
7.2 Lexical Conventions.....	11
7.3 Objects.....	13
7.4 Filters.....	22
7.5 File Structure.....	38
7.6 Encryption.....	55
7.7 Document Structure.....	70
7.8 Content Streams and Resources.....	81
7.9 Common Data Structures.....	84
7.10 Functions.....	92
7.11 File Specifications.....	99
7.12 Extensions Dictionary.....	108
8 Graphics.....	110
8.1 General.....	110
8.2 Graphics Objects.....	110
8.3 Coordinate Systems.....	114
8.4 Graphics State.....	121
8.5 Path Construction and Painting.....	131
8.6 Colour Spaces.....	138
8.7 Patterns.....	173
8.8 External Objects.....	201
8.9 Images.....	203
8.10 Form XObjects.....	217
8.11 Optional Content.....	222
9 Text.....	237
9.1 General.....	237
9.2 Organization and Use of Fonts.....	237
9.3 Text State Parameters and Operators.....	243
9.4 Text Objects.....	248
9.5 Introduction to Font Data Structures.....	253
9.6 Simple Fonts.....	254
9.7 Composite Fonts.....	267
9.8 Font Descriptors.....	281
9.9 Embedded Font Programs.....	288
9.10 Extraction of Text Content.....	292
10 Rendering.....	296

© Adobe Systems Incorporated 2008 – All rights reserved

iii

The above command creates the file `steg.pdf`, which has the malware steganographically encoded into it. To check that our method is reliable, we can now extract the malware sample back out of `steg.pdf`. This is performed with the following command:

```
$ python3 full_pdf_steg.py extract steg.pdf trojan-petya-out
```

The results of this command are shown below:

```
[+] Extracting Message
Size of extracted message: 335872
[-] Message Extracted
Message successfully extracted
```

We see that the size of the extracted file matches the size of our original malware sample. To further confirm that the embedded and extracted malware samples are the same, we could run an `md5sum` command on both files and confirm that the hashes are equal.

The `steg.pdf` file still has uncompressed object streams, and therefore is a much larger file than the original `PDF32000_2008.pdf` file. To recover disk space, we can (lossless) re-compress the PDF with the following command, as discussed in Section 2.4:

```
$ qpdf steg.pdf --stream-data=compress compressed_steg.pdf
```

The sizes of all relevant files are shown in the table below:

File Name	Compressed Streams	File Size (bytes)
<code>PDF32000_2008.pdf</code>	Y	22,491,828
<code>cover.pdf</code>	N	30,840,706
<code>steg.pdf</code>	N	32,284,996
<code>compressed_steg.pdf</code>	Y	22,909,867
<code>trojan-petya</code>	N/A	335,872

The following calculations summarize the results of this case study.

Steg message capacity = 464,007 bytes

Net Increase in PDF size (decompressed streams) = 1,444,290 bytes

Net Increase in PDF size (compressed streams) = 418,039 bytes

Rate (steg message capacity / compressed cover size) = 2.06%

Rate (steg message size / decompressed size increase) = 23.25%

Rate (steg message size / compressed size increase) = 80.34%

Rate (decompressed size increase / decompressed cover size) = 4.68%

Rate (compressed size increase / compressed cover size) = 1.86%

Overall, this sample run of the PDF steganography method successfully embedded a 335 KB malware sample into a 22.5 MB PDF file. The resulting PDF file has a size of 22.9 MB, which is an net size increase of less than 2%. The steganographic encoding very efficiently uses the additional size that it adds to the PDF file.

6 Conclusion

By utilizing an adaptable strategy of least-significant bit insertion on string representations of floating-point operands to perform imperceptible changes to PDF stream operators, this paper presents a novel PDF steganography method. Our method has a higher carrying capacity than previous PDF operator-based methods due to the use of all viable operators found within a PDF file. Our case study of embedding malware into a given PDF cover file justifies effectiveness of our proposed approach.

Acknowledgments. This work was supported by US National Science Foundation under grant number CNS-1928349, CNS-2225424, and 2043022-DGE.

References

- [1] *Document management — Portable document format — Part 1: PDF 1.7*. Standard. Geneva, CH: International Organization for Standardization, July 2008.
- [2] *Document management — Portable document format — Part 2: PDF 2.0*. Standard. Geneva, CH: International Organization for Standardization, Dec. 2020.
- [3] Behrooz Khosravi et al. “A New Method for Pdf Steganography in Justified Texts”. In: *J. Inf. Secur. Appl.* 45.C (Apr. 2019), pp. 61–70. ISSN: 2214-2126. DOI: 10.1016/j.jisa.2019.01.003. URL: <https://doi.org/10.1016/j.jisa.2019.01.003>.
- [4] Katarzyna Koptyra and Marek R. Ogiela. “Distributed Steganography in PDF Files—Secrets Hidden in Modified Pages”. In: *Entropy* 22.6 (May 2020), p. 600. ISSN: 1099-4300. DOI: 10.3390/e22060600. URL: <http://dx.doi.org/10.3390/e22060600>.
- [5] *PDFSteg*. 2024. URL: <https://snp.cs.mtu.edu/pdfsteg.html> (visited on 05/01/2024).
- [6] *PDFtk - The PDF Toolkit*. 2020. URL: <https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/> (visited on 01/30/2023).
- [7] *QPDF: A Content-Preserving PDF Transformation System*. 2023. URL: <https://qpdf.sourceforge.io/> (visited on 01/30/2023).
- [8] Alexander V. Sergeev and Pavel B. Khorev. “Analysis of Methods for Hiding Information in PDF Documents and Opportunities for Their Progress”. In: *2020 International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE)*. 2020, pp. 1–4. DOI: 10.1109/REEPE49198.2020.9059117.

- [9] Thomas Sloan and Julio Hernandez-Castro. “Dismantling OpenPuff PDF steganography”. In: *Digital Investigation* 25 (2018), pp. 90–96. ISSN: 1742-2876. DOI: <https://doi.org/10.1016/j.diin.2018.03.003>. URL: <https://www.sciencedirect.com/science/article/pii/S1742287617303286>.
- [10] Naldiyo Sofian, Arya Wicaksana, and Seng Hansun. “LSB Steganography and AES Encryption for Multiple PDF Documents”. In: *2019 5th International Conference on New Media Studies (CON-MEDIA)*. 2019, pp. 100–105. DOI: 10.1109/CONMEDIA46929.2019.8981842.
- [11] Jiun-Tsung Wang and Wen-Hsiang Tsai. “DATA HIDING IN PDF FILES AND APPLICATIONS BY IMPERCEIVABLE MODIFICATIONS OF PDF OBJECT PARAMETERS”. In: 2008.

A Evaluating All Possible Operators for PDF Steganography

Note that *the quoted descriptions of operators in below sections are pulled directly from the PDF standard.*

A.1 Graphics Drawing Operators

The following operators control graphics that are drawn onto a PDF document. They generally take operands in units of the coordinate space of the PDF document.

c, **v**, **y** operators draw cubic Bézier curves based on the coordinate parameters given. The difference between the variants determines which points are used as endpoints, and which points are used as control points.

l, **m** operators move the ‘current point’ of the PDF graphics. **m** skips directly to the new point, while **l** draws a straight line from the previous point to the new point. **m** is often used to start a sequence of graphics operations.

re operator draws a rectangle.

cm operator specifies the matrix transform to apply to all following graphics.

i, **M**, **w** each use one operand to set a single graphics parameter.

Overall, the study of using this class of operators for steganography shows that graphics coordinates should likely not be changed by integral amounts, which impacts operators **c**, **v**, **y**, **l**, **m**, **re**.

c Operator Takes 6 floating-point operands: **x1 y1 x2 y2 x3 y3**.

Table 58: “Append a cubic Bézier curve to the current path. The curve shall extend from the current point to the point (x3 , y3), using (x1 , y1) and (x2 , y2) as the Bézier control points (see 8.5.2.2, “Cubic Bézier Curves”). The new current point shall be (x3 , y3).” [2]

Figure 1 shows test runs of the **c** operator, with differing percentages of change in the coordinates. It appears that a percentage change of 0.1% in the operator values causes the least amount of visible change. This makes sense, since it corresponds to non-integer change in the PDF coordinate space. For any operators that require the specification of coordinate points, it appears that taking changes (at least) one place past the decimal point is the safest option.

Cutoff percentage will be equivalent to **v** and **y** operators, since this is just a different ordering of the same operation.

v Operator Takes 4 floating-point operands: **x2 y2 x3 y3**

Table 58: “Append a cubic Bézier curve to the current path. The curve shall extend from the current point to the point (x3 , y3), using the current point and (x2 , y2) as the Bézier control points (see 8.5.2.2, “Cubic Bézier Curves”). The new current point shall be (x3 , y3).” [2]

Cutoff percentage will be equivalent to **c** and **y** operators, since this is just a different ordering of the same operation.

y Operator Takes 4 floating-point operands: **x1 y1 x3 y3**

Table 58: “Append a cubic Bézier curve to the current path. The curve shall extend from the current point to the point (x3 , y3), using (x1 , y1) and (x3 , y3) as the Bézier control points (see 8.5.2.2, “Cubic Bézier Curves”). The new current point shall be (x3 , y3).” [2]

Cutoff percentage will be equivalent to **c** and **v** operators, since this is just a different ordering of the same operation.

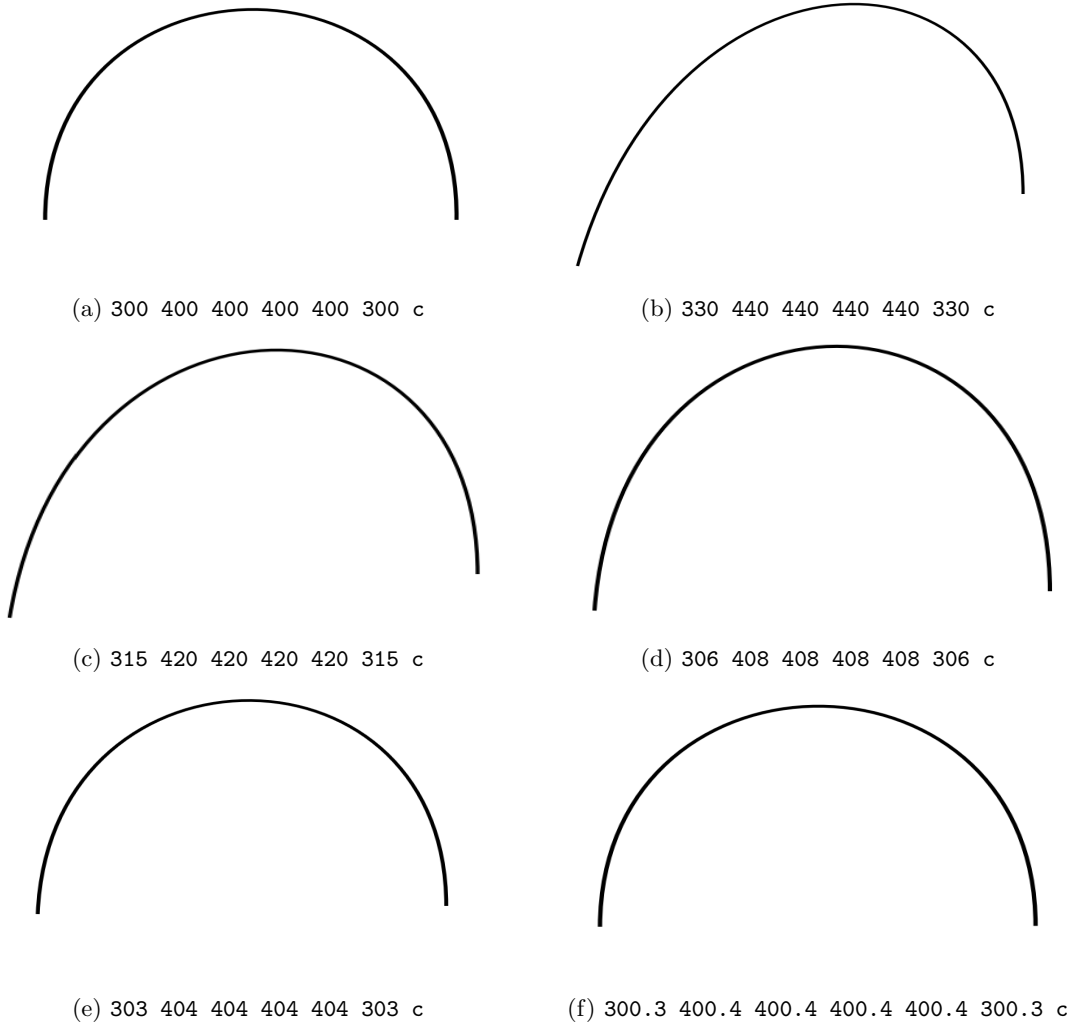


Fig. 1: c Operator - control, 10%, 5%, 2% 1%, 0.1% (1%)

l Operator Takes 2 floating-point operands: **x y**

Table 58: “Append a straight line segment from the current point to the point (x, y). The new current point shall be (x, y).”

m Operator Takes 2 floating-point operands: **x y**

Table 58: “Begin a new subpath by moving the current point to coordinates (x, y), omitting any connecting line segment. If the previous path construction operator in the current path was also m, the new m overrides it; no vestige of the previous m operation remains in the path.” [2]

The **m** operator places a point for PDF graphics control. Thus, it will impact the starting point of the next graphics operator. Since it acts on coordinates in graphics space, it’s cutoff shall be similar to **c**, **l**, **re** operators. We will place the cutoff at 0.05%.

re Operator Takes 4 floating-point operand: **x y width height**

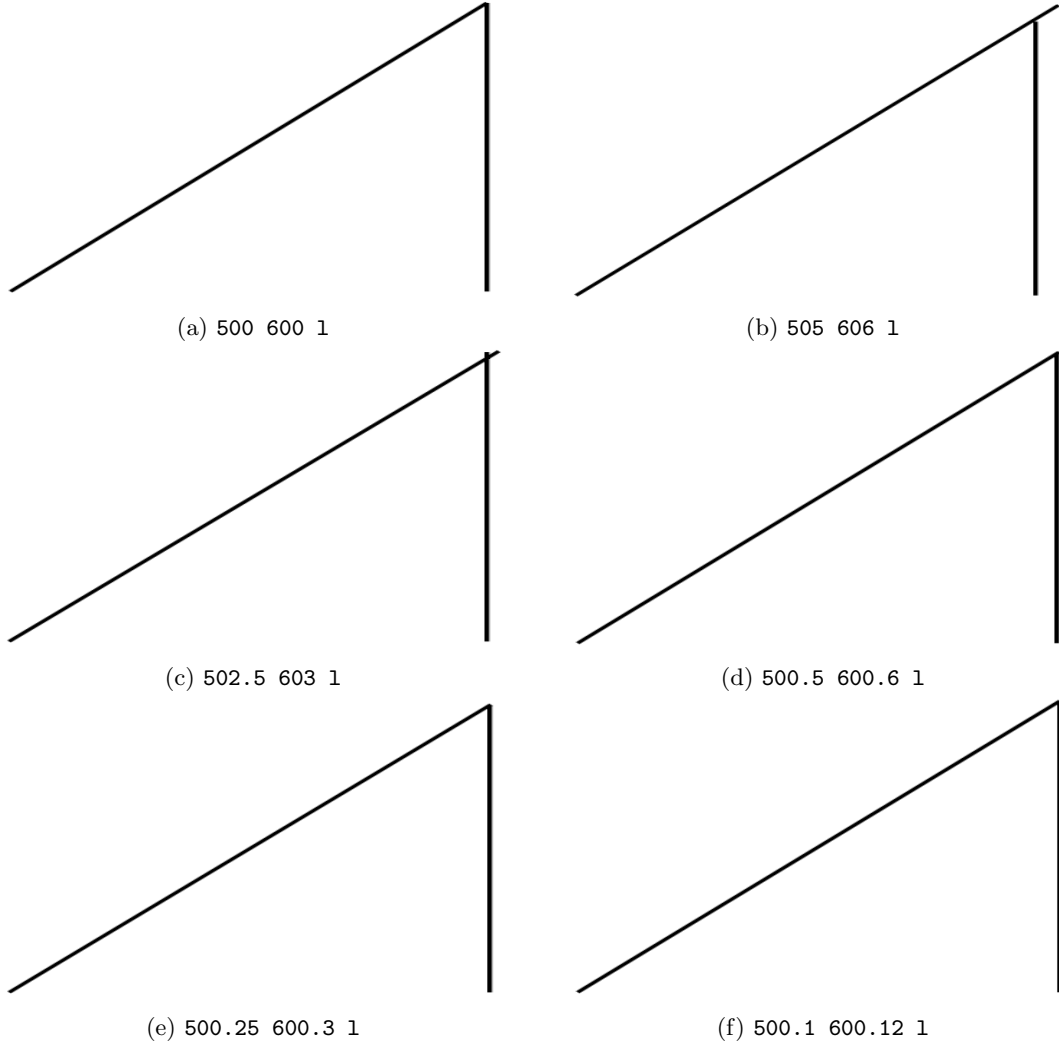


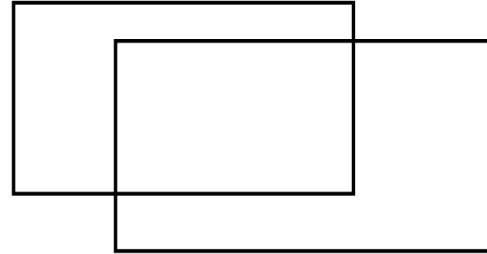
Fig. 2: l Operator - control, 1%, 0.5%, 0.1% 0.05%, 0.02% (0.05%)

Table 58: “Append a rectangle to the current path as a complete subpath, with lower-left corner (x, y) and dimensions width and height in user space.” [2]

This operator is equivalent to drawing 4 lines of a rectangle with the l operator.



(a) 300 675 100 50 re
300 600 100 50 re

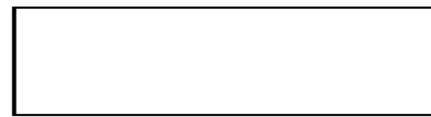
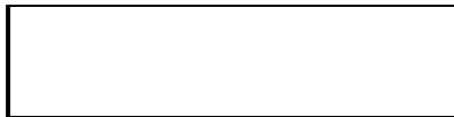


(b) 300 675 100 50 re
330 660 110 55 re



(c) 300 675 100 50 re
303 606 101 50.5 re

(d) 300 675 100 50 re
301.5 603 100.5 50.25 re



(e) 300 675 100 50 re
300.6 601.2 100.2 50.1 re

(f) 300 675 100 50 re
300.3 600.6 100.1 50.05 re

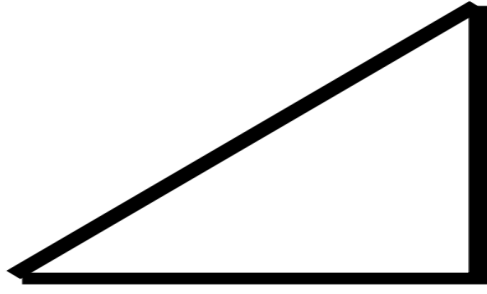
Fig. 3: re Operator - control, 10%, 1%, 0.5%, 0.2% 0.1% (0.2%)

cm Operator Takes 6 floating-point operands: **a b c d e f**.

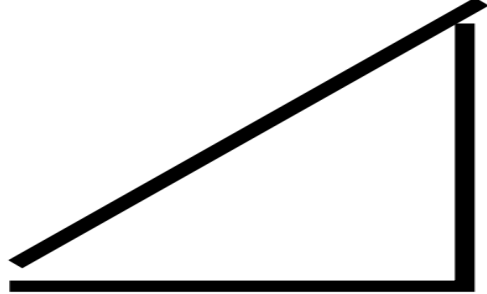
Table 56: “Modify the current transformation matrix (CTM) by concatenating the specified matrix (see 8.3.2, “Coordinate Spaces”). Although the operands specify a matrix, they shall be written as six separate numbers, not as an array.” [2]

Operands **e** and **f** simply shift the graphics space simply shift the graphics space by **e** units in the x direction and **f** units in the y direction. This makes those operands very similar in effect and units to the **m** operator, so we will place the cutoff for operands **e** and **f** at 0.1%.

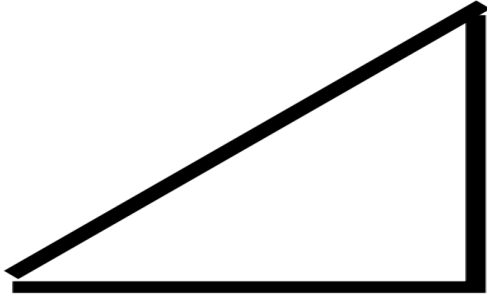
Operands **a**, **b**, **c**, **d** are more complex, as they apply a matrix transformation to the graphics space. Just modifying **a** and **d** will create stretching or compressing in the horizontal and vertical directions. Just modifying **b** and **c** will create skew of the vertical and horizontal axes. Modifying all 4 will cause a combination of both effects, including rotation of the coordinate space.



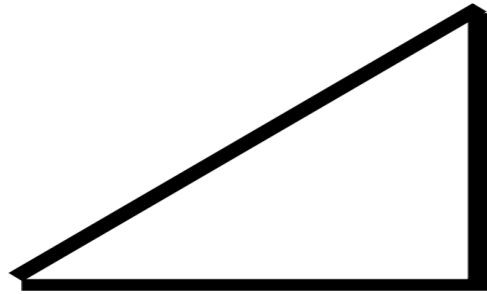
(a) 0.5 0.866 -0.866 0.5 0 0 cm



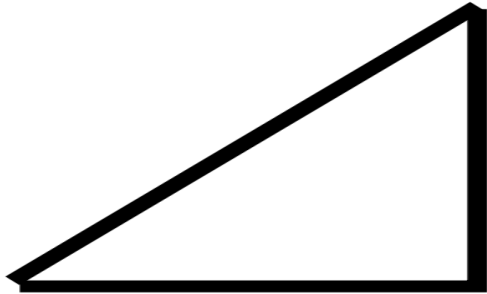
(b) 0.505 0.874 -0.874 0.505 0 0 cm



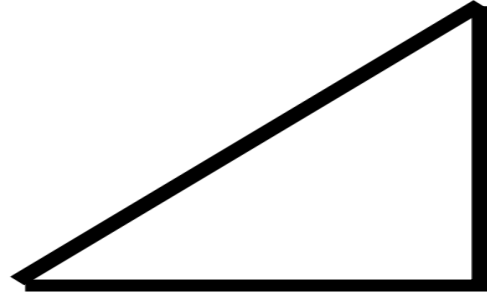
(c) 0.5025 0.8703 -0.8703 0.5025 0 0 cm



(d) 0.501 0.8677 -0.8677 0.501 0 0 cm



(e) 0.5005 0.86687 -0.86687 0.5005 0 0 cm



(f) 0.50025 0.86643 -0.86643 0.50025 0 0 cm

Fig. 4: cm Operator - control, 1%, 0.5%, 0.2% 0.1%, 0.05% (0.1%)

i Operator (?) Takes 1 floating-point operand: **flatness**

Table 56: “Set the flatness tolerance in the graphics state (see 10.6.2, ”Flatness Tolerance”). flatness is a number in the range 0 to 100; a value of 0 shall specify the output device’s default flatness tolerance.” [2]

“The flatness tolerance controls the maximum permitted distance in device pixels between the mathematically correct path and an approximation constructed from straight line segments, as shown in Figure 54. Flatness may be specified as the operand of the i operator (see Table 57) or as the value of the FL entry in a graphics state parameter dictionary (see Table 58). It shall be a positive number.”

M Operator (?) Takes 1 floating-point operand: `miterLimit`

Table 56: “Set the miter limit in the graphics state (see 8.4.3.5, ”Miter Limit”).” [2]

The formula given in the standard for miter limit is given as: $M = \frac{1}{\sin(\frac{j}{2})}$, where j is the angle between two meeting lines. If the calculated value for two arbitrary meeting lines is greater than the provided miter limit, then the point where the lines meet is turned into a bevel.

From the above formula, we can calculate the angle from the miter limit as follows: $j = 2 \cdot \arcsin(\frac{1}{M})$

Section 8.4.3.5: “A miter limit of 1.414 converts miters to bevels for j less than 90 degrees, a limit of 2.0 converts them for j less than 60 degrees, and a limit of 10.0 converts them for j less than approximately 11.5 degrees.” [2]

The following table looks at the percentage change in miter limit vs percentage change in the miter cutoff angle, for initial miter limits of 1.414, 2, and 10.

Miter Limit	Angle	% change	Miter Limit	% change	Angle
1.414	90.02	0			
1.555	80.04	10			11
1.485	84.66	5			6
1.442	87.81	2			2.5
1.428	88.9	1			1.25
2	60	0			0
2.2	54.07	10			10
2.1	56.87	5			5.2
2.04	58.71	2			2.2
2.02	59.35	1			1.1
10.0	11.48	0			0
11.0	10.43	10			9.1
10.5	10.93	5			4.79
10.2	11.25	2			2
10.1	11.36	1			1.05

It can be seen from the table that the percentage change in the miter limit correlates roughly linearly with the change in the cutoff angle between miters and bevels. This is likely due to the fact that the sine function can be accurately modelled as a linear function for small changes. Thus, by determining how much we can allow our angle cutoff to change, we can apply the same cutoff to the miter limit operand.

w Operator Takes 1 floating-point operand: `lineWidth`

Table 56: “Set the line width in the graphics state (see 8.4.3.2, ”Line Width”).” [2]

A.2 Color Control Operators

The following operators control the color of text and graphics in a PDF document. They generally take operands in units of color intensity in some color space. Operators specified by capital letters control colors for ‘stroking’ operations, which draw the outline of shapes. Operators specified by lowercase letters control the colors for ‘fill’ operations, which draw the inside of shapes.

Overall, we can assign a uniform percentage cutoff of 5% to all the color operators. A 5% change in color intensity is very difficult to see visually. This lines up with existing methods for image steganography that handle changes in color.

G Operator Takes 1 floating-point operand: `gray`

Table 73: “Set the stroking colour space to DeviceGray (or the DefaultGray colour space; see 8.6.5.6, ”Default Colour Spaces”) and set the gray level to use for stroking operations. `gray` shall be a number between 0.0 (black) and 1.0 (white).” [2]

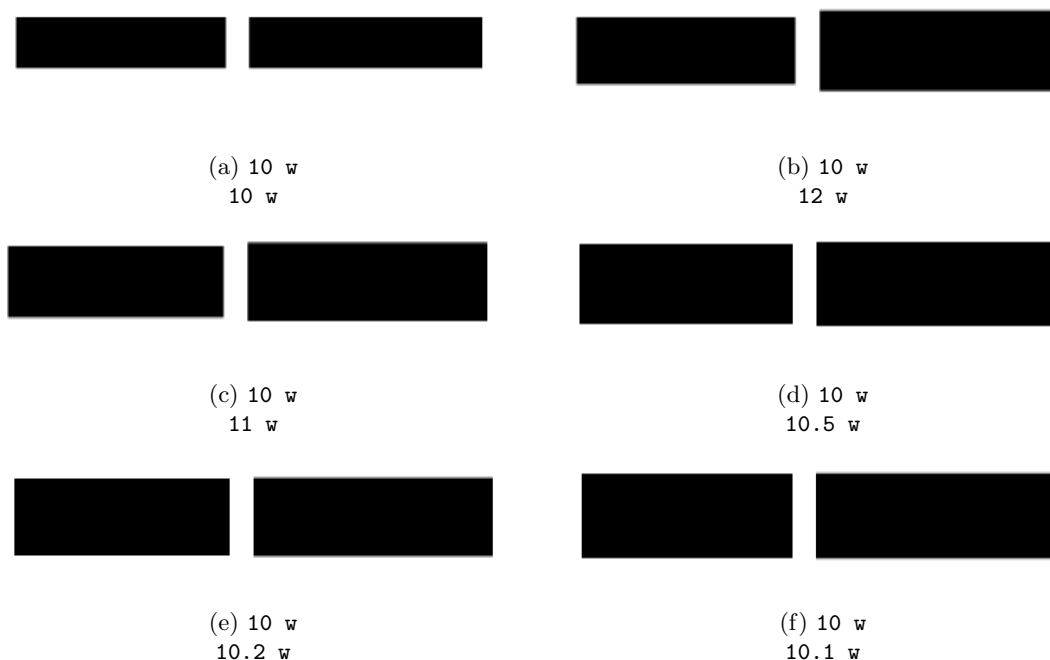


Fig. 5: w Operator - control, 20%, 10%, 5%, 2% 1% (1%)

g Operator Takes 1 floating-point operand: **gray**

Table 73: “Same as G but used for nonstroking operations.” [2]

K Operator Takes 4 floating-point operands: **c m y k**

Table 73: “Set the stroking colour space to DeviceCMYK (or the DefaultCMYK colour space; see 8.6.5.6, ”Default Colour Spaces”) and set the colour to use for stroking operations. Each operand shall be a number between 0.0 (zero concentration) and 1.0 (maximum concentration). The behaviour of this operator is affected by the overprint mode (see 8.6.7, ”Overprint Control”).” [2]

k Operator Takes 4 floating-point operands: **c m y k**

Table 73: “Same as K but used for nonstroking operations.” [2]

RG Operator Takes 3 floating-point operands: **r g b**

Table 73: “Set the stroking colour space to DeviceRGB (or the DefaultRGB colour space; see 8.6.5.6, ”Default Colour Spaces”) and set the colour to use for stroking operations. Each operand shall be a number between 0.0 (minimum intensity) and 1.0 (maximum intensity).” [2]

rg Operator Takes 3 floating-point operands: **r g b**

Table 73: “Same as RG but used for nonstroking operations.” [2]

sc Operator Takes 1+ floating-point operands: $c_1 \dots c_n$

Table 73: “(PDF 1.1) Same as SC but used for nonstroking operations.” [2]

Thus when $n = 1$, **sc** family of operators are equivalent to **G** and **g** operators.
When $n = 3$, **sc** family of operators are equivalent to **RG** and **rg** operators.

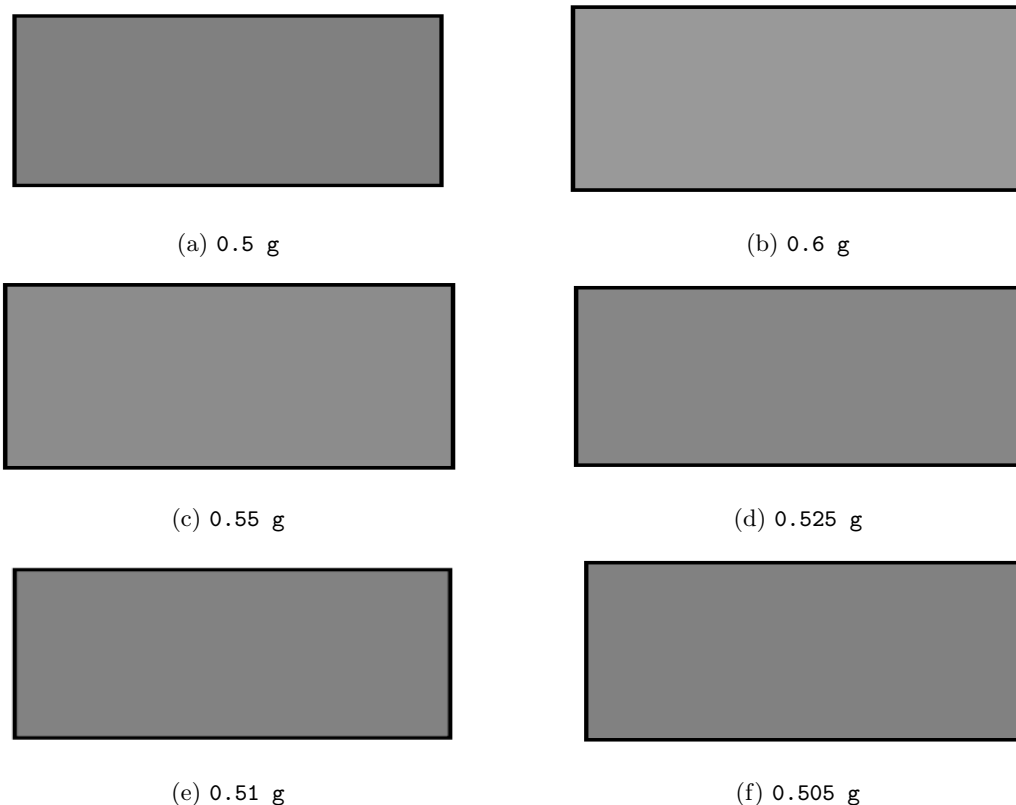


Fig. 6: g Operator - control, 20%, 10%, 5%, 2%, 1% (5%)

When $n = 4$, **sc** family of operators are equivalent to **K** and **k** operators.

Thus, since the cutoff for all operators **G**, **g**, **RG**, **rg**, **K**, **k** is 5%, the cutoff for all operators in the **sc** family must be 5%.

SC Operator Takes 1+ floating-point operands: $c_1 \dots c_n$

Table 73: “(PDF 1.1) Set the colour to use for stroking operations in a device, CIE- based (other than ICCBased), or Indexed colour space. The number of operands required and their interpretation depends on the current stroking colour space:

For DeviceGray, CalGray, and Indexed colour spaces, one operand shall be required ($n = 1$).

For DeviceRGB, CalRGB, and Lab colour spaces, three operands shall be required ($n = 3$).

For DeviceCMYK, four operands shall be required ($n = 4$).” [2]

scn Operator Takes 1+ floating-point operands: $c_1 \dots c_n$

Table 73: “(PDF 1.2) Same as SCN but used for nonstroking operations.” [2]

SCN Operator Takes 1+ floating-point operands: $c_1 \dots c_n$

Table 73: “(PDF 1.2) Same as SC but also supports Pattern, Separation, DeviceN and ICCBased colour spaces.” [2]

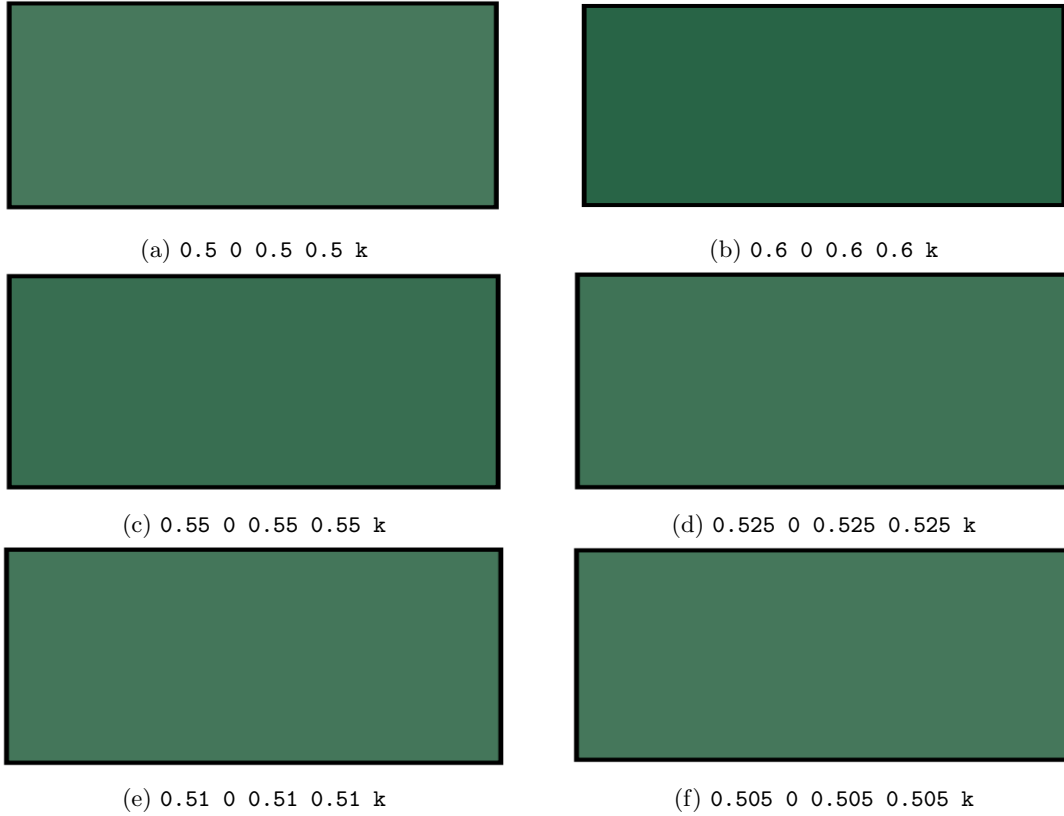


Fig. 7: k Operator - control, 20%, 10%, 5%, 2%, 1% (5%)

A.3 Text Control Operators

The following operators control how text is placed inside of a PDF document. Many take units in terms of ‘text units’, either scaled by some value or unscaled. In determining perceptibly of changes, it is important to consider both the vertical and horizontal changes in text scaling.

Tc Operator Takes 1 floating-point operand: `charSpace`

Table 103: “Set the character spacing, T_c , to `charSpace`, which shall be a number expressed in unscaled text space units. Character spacing shall be used by the Tj, TJ, and ’ operators. Initial value: 0.” [2]



(a) 0 0.5 0.5 rg



(b) 0 0.6 0.6 rg



(c) 0 0.55 0.55 rg



(d) 0 0.525 0.525 rg



(e) 0 0.51 0.51 rg



(f) 0 0.505 0.505 rg

Fig. 8: rg Operator - control, 20%, 10%, 5%, 2%, 1% (5%)

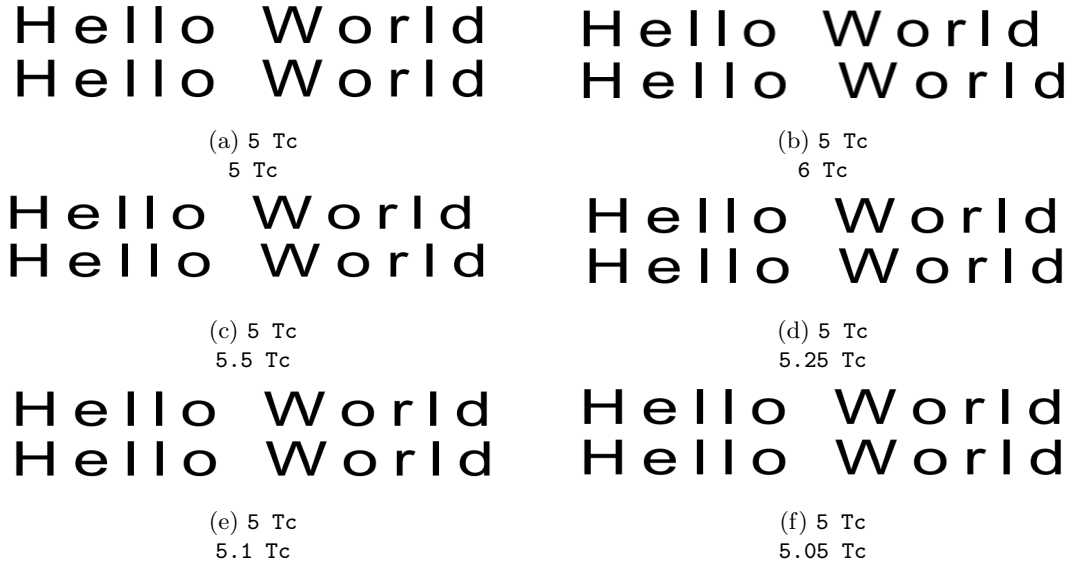


Fig. 9: Tc Operator - control, 20%, 10%, 5%, 2%, 1% (1%, less stable)

Td Operator Takes 1 floating-point operand: t_x t_y

Table 106: “Move to the start of the next line, offset from the start of the current line by (t_x, t_y) . t_x and t_y shall denote numbers expressed in unscaled text space units.” [2]

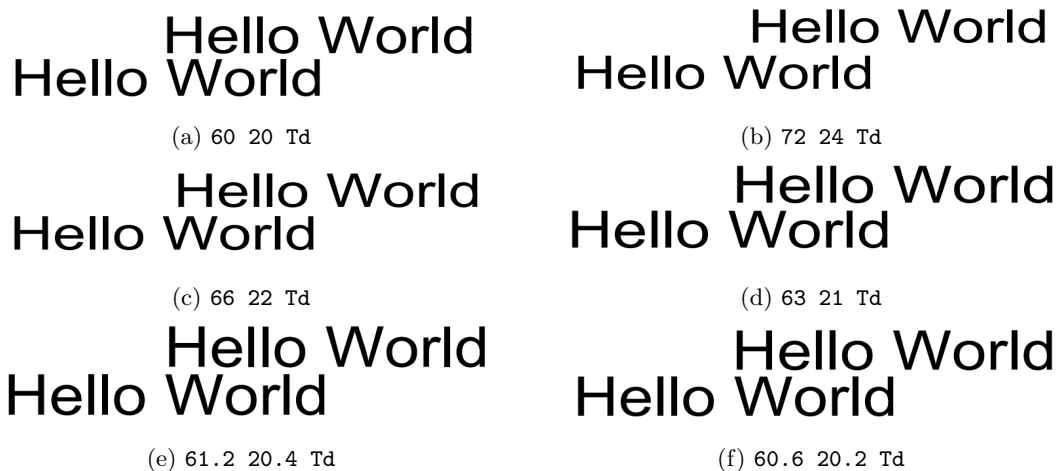


Fig. 10: Td Operator - control, 20%, 10%, 5%, 2%, 1% (2%)

TD Operator Takes 1 floating-point operand: t_x t_y

Table 106: “Move to the start of the next line, offset from the start of the current line by (t_x, t_y) . As a side effect, this operator shall set the leading parameter in the text state.” [2]

According to the PDF standard, **x y TD** is equivalent to:

-y TL
x y Td [2]



Fig. 11: TD Operator - control, 20%, 10%, 5%, 2%, 1% (2%)

Tf Operator Takes 1 floating-point operand: **size**

Table 103: “Set the text font, T_f , to font and the text font size, T_{fs} , to size. font shall be the name of a font resource in the Font subdictionary of the current resource dictionary; size shall be a number representing a scale factor. There is no initial value for either font or size; they shall be specified explicitly by using Tf before any text is shown.” [2]



Fig. 12: Tf Operator - control, 10%, 5%, 2%, 1%, 0.5%, 0.2%, 0.1% (0.5%)

TL Operator Takes 1 floating-point operand: **leading**

Table 103: “Set the text leading, T_l , to leading, which shall be a number expressed in unscaled text space units. Text leading shall be used only by the T*, ', and ” operators. Initial value: 0.” [2]



Fig. 13: Tl Operator - control, 10%, 5%, 2%, 1%, 0.5% (2%)

Tm Operator Takes 1 floating-point operand: **a b c d e f**

Table 106: “Set the text matrix, T_m , and the text line matrix, T_{lm} :

The operands shall all be numbers, and the initial value for T_m and T_{lm} shall be the identity matrix, $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Although the operands specify a matrix, they shall be passed to Tm as six separate numbers, not as an array.

The matrix specified by the operands shall not be concatenated onto the current text matrix, but shall replace it.” [2]

The Tm operator is similar to the cm graphics operator. Operands **e** and **f** move the location where the next piece of text will be drawn. Therefore, those two operands have the same effect as the Td operator, and should have the same percentage cutoff of 2%. Looking at operands **a**, **b**, **c**, **d** shows that their percentage cutoff should be



Fig. 14: Tm Operator - control, 20%, 10%, 5%, 2%, 1% (5-10%)

Ts Operator Takes 1 floating-point operand: `rise`

Table 103: “Set the text rise, T_{rise} , to rise, which shall be a number expressed in unscaled text space units. Initial value: 0.” [2]

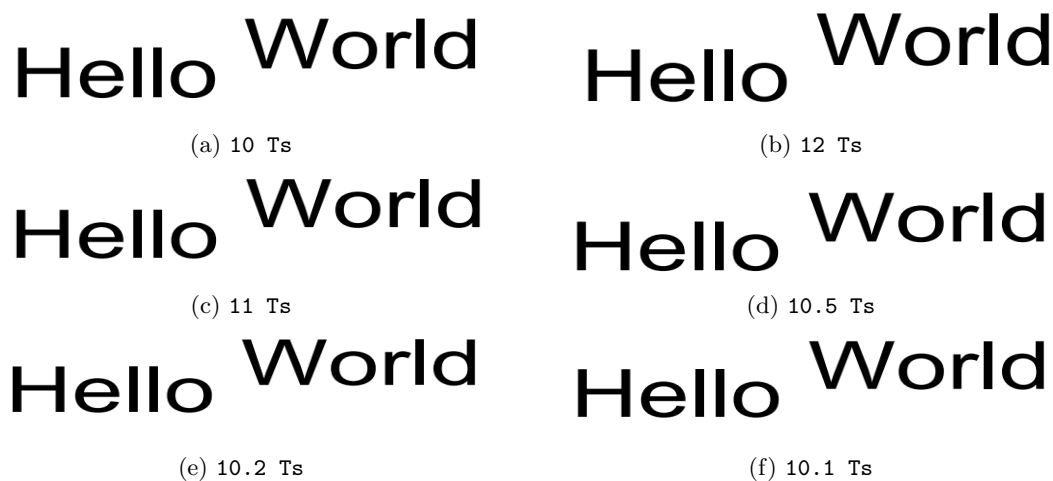


Fig. 15: Ts Operator - control, 20%, 10%, 5%, 2%, 1% (5-10%)

Tw Operator Takes 1 floating-point operand: `wordSpace`

Table 103: “Set the word spacing, T_w , to wordSpace, which shall be a number expressed in unscaled text space units. Word spacing shall be used by the Tj, TJ, and ' operators. Initial value: 0.” [2]

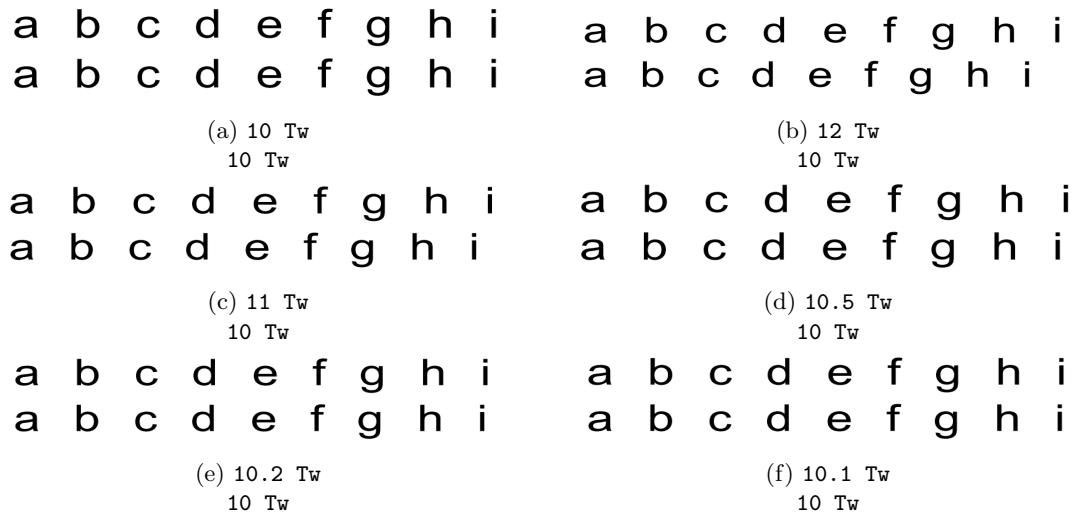


Fig. 16: Tw Operator - control, 20%, 10%, 5%, 2%, 1% (1%, less stable?)

Tz Operator Takes 1 floating-point operand: **scale**

Table 103: “Set the horizontal scaling, T_h , to $(\text{scale} \div 100)$. scale shall be a number specifying the percentage of the normal width. Initial value: 100 (normal width).” [2]

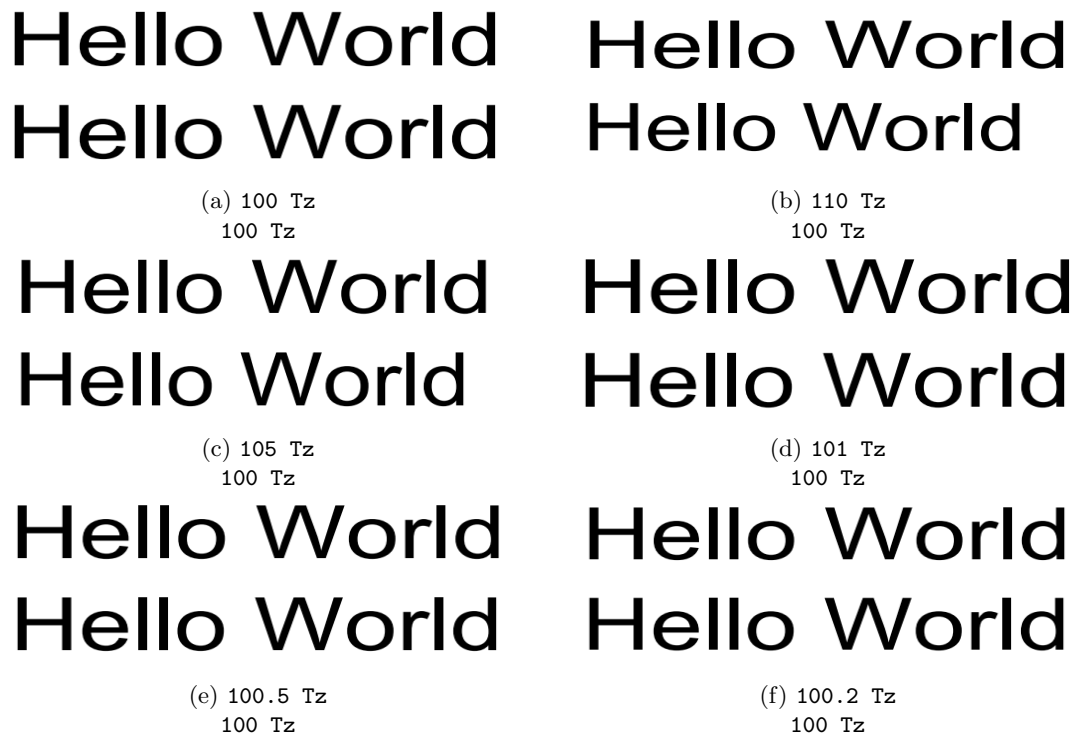


Fig. 17: Tz Operator - control, 10%, 5%, 1%, 0.5%, 0.2% (0.5%)

TJ Operator Takes an array of string and numerical operands. The numerical operands have been proven in previous work (Papers 9.1.1, 9.1.2) to be viable for steganography. Since the units are scaled to thousandths of text space units, a very significant percentage change is necessary to create a noticeable visual change. Extensive study has been done on using this operator for steganography. While our method will not likely yield improved embedding rates for the TJ operator in particular, .

Table 107: “Show one or more text strings, allowing individual glyph positioning. Each element of array shall be either a string or a number. If the element is a string, this operator shall show the string. If it is a number, the operator shall adjust the text position by that amount; that is, it shall translate the text matrix, T_m . The number shall be expressed in thousandths of a unit of text space (see 9.4.4, “Text Space Details”). This amount shall be subtracted from the current horizontal or vertical coordinate, depending on the writing mode. In the default coordinate system, a positive adjustment has the effect of moving the next glyph painted either to the left or down by the given amount”. [2]

A.4 Misc. Operators

Note: not deprecated, just very infrequently used in practice. But it is still in the new standard.

d0 Operator Takes 2 floating-point operands: w_x w_y

Table 111: “ w_x denotes the horizontal displacement in the glyph coordinate system; it shall be consistent with the corresponding width in the font’s Widths array. w_y shall be 0 (see 9.2.4, “Glyph Positioning and Metrics”).” [2]

Therefore, it effectively only has 1 operand, since w_y must be fixed to 0

He
He

(a) [(H)50(e)] TJ
[(H)50(e)] TJ

He
He

(c) [(H)50(e)] TJ
[(H)65(e)] TJ

He
He

(e) [(H)50(e)] TJ
[(H)57.5(e)] TJ

He
He

(b) [(H)50(e)] TJ
[(H)75(e)] TJ

He
He

(d) [(H)50(e)] TJ
[(H)60(e)] TJ

He
He

(f) [(H)50(e)] TJ
[(H)55(e)] TJ

Fig. 18: TJ Operator - control, 50%, 30%, 20%, 15%, 10% (20%)

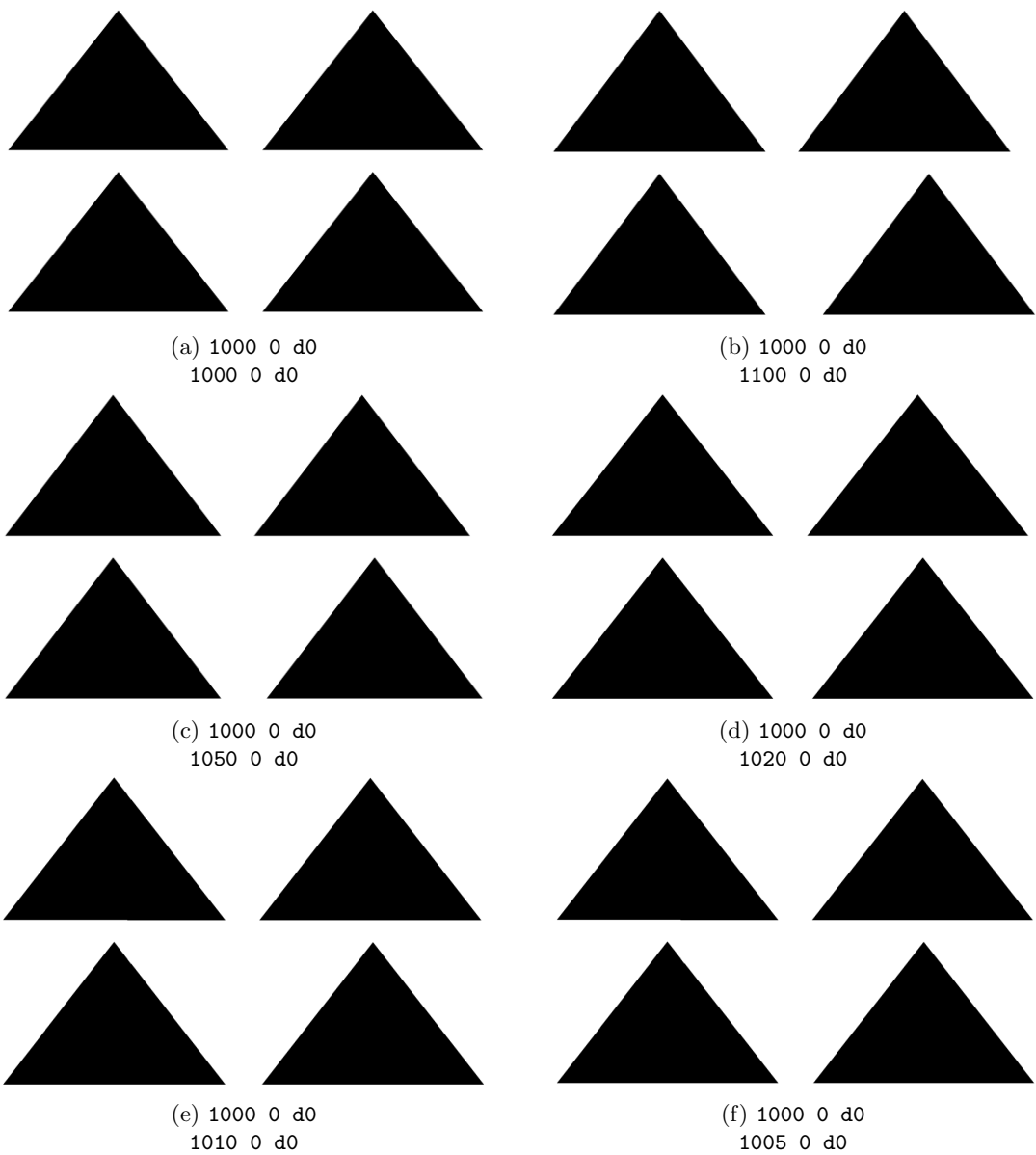


Fig. 19: d0 Operator - control, 10%, 5%, 2%, 1%, 0.5% (1%)

d1 Operator Takes 6 floating-point operands: w_x w_y ll_x ll_y ur_x ur_y

Table 111: “ w_x denotes the horizontal displacement in the glyph coordinate system; it shall be consistent with the corresponding width in the font’s Widths array. w_y shall be 0 (see 9.2.4, ”Glyph Positioning and Metrics”).

ll_x and ll_y denote the coordinates of the lower-left corner, and ur_x and ur_y denote the upper-right corner, of the glyph bounding box. The glyph bounding box is the smallest rectangle, oriented with the axes of the glyph coordinate system, that completely encloses all marks placed on the page as a result of executing the glyph’s description. The declared bounding box shall be correct—in other words, sufficiently large to enclose the entire glyph. If any marks fall outside this bounding box, the result is unpredictable.” [2]

Therefore, it effectively only has 5 operands, since w_y must be fixed to 0. The w_x and w_y operands are the same as in the d0 operator, so they have a percentage cutoff of 1%. The following diagrams check the percentage cutoffs for the ll_x ll_y ur_x ur_y operands. These operands just control the bounding box around the font glyphs, which appear to only impact the box shown when the ‘text’ is highlighted within a PDF.

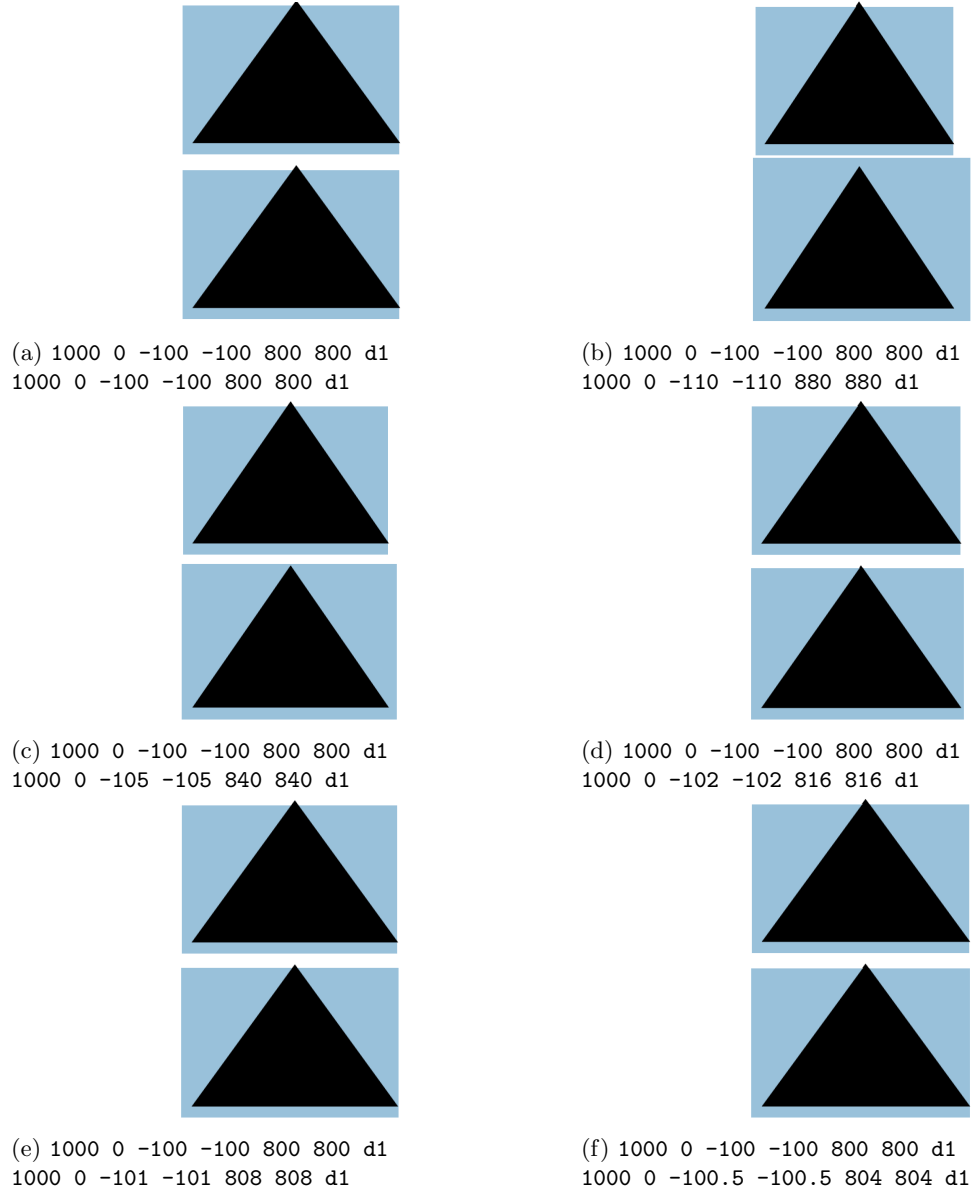


Fig. 20: d1 Operator - control, 10%, 5%, 2%, 1%, 0.5% (1-2%)