

Radim Hatlapatka; Petr Sojka  
PDF Enhancements Tools for a Digital Library

In: Petr Sojka (ed.): Towards a Digital Mathematics Library. Paris, France, July 7-8th, 2010.  
Masaryk University Press, Brno, Czech Republic, 2010. pp. 45--55.

Persistent URL: <http://dml.cz/dmlcz/702572>

## Terms of use:

© Masaryk University, 2010

Institute of Mathematics of the Academy of Sciences of the Czech Republic provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.



This paper has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* <http://project.dml.cz>

# PDF Enhancements Tools for a Digital Library

## pdfJbIm and pdfsign

Radim Hatlapatka and Petr Sojka

Masaryk University, Faculty of Informatics  
Botanická 68a, 602 00 Brno, Czech Republic  
208155@mail.muni.cz, sojka@fi.muni.cz

**Abstract.** This paper describes several innovative PDF document enhancements and tools that can be used when building a digital library. The main result presented in this paper is the PDF re-compression tool, developed using the `jbig2enc` encoder called `pdfJbIm`. This re-compression tool enables the size of the original bitonal PDFs to be, on average, downsized by one third. Some modifications to the `jbig2enc` encoder that increase the compression ratio even further are also described here. Together with another program, the `pdfsizeopt.py` by Péter Szabó, we have managed to decrease PDF storage size to such an extent that the transmission needs of a digital library were significantly reduced. We report the storage saving results that we have achieved on The Czech Digital Mathematics Library DML-CZ — we have downsized the PDF corpus to 43% of its original size. We also describe `pdfsign` tool for batch digital signature stamping of PDF documents.

**Key words:** `jbig2enc`, JBIG2, PDF size optimization, compression, DML, digital signature, JB2, DjVu, `pdfJbIm`, `pdfsign`, DML-CZ, EuDML

*Smaller is faster and safer too.* (Stephen Adams, Google)

## 1 Motivation

PDF is the most frequently used format for digital libraries today. Although storage size is not the most important aspect when storing data, it must be taken into account when delivering hundreds of thousands of documents to users. PDF allows any digital objects in a document to be compressed by different methods, supported in PDF format evolution.

The standard applications used to generate PDFs are usually inadequate to the optimal-size generation task. Only during PDF *postprocessing*, global optimizations like choosing the most appropriate compression and object decomposition might be performed. Postprocessing is the best way of optimizing the size and the page-at-time rendering for in-browser PDF rendering and for delivering PDFs from a digital library.

During the course of The Czech Digital Mathematical Library project [1], we started to experiment with different ways of PDF optimization and delivery. As most of the files are two-layer PDFs with scanned bitonal bitmap in front

and the OCR recognized text behind, the recent JBIG2 standard, developed for one-bit depth scanned images, emerged as the most suitable for storing this kind of digitized data. As we also wanted our data to be distinguishable from other versions claiming the same content, we decided to digitally sign every PDF that resulted from our project.

The JBIG2 standard and its formats are described in Section 2. Section 4 on page 50 presents our application `pdfJbIm` to enhance the compression ratio achieved by the improved leading edge software which is then compared with `pdfsizeopt.py` program in Section 5 on page 51. The main results are discussed in Section 6 on page 51. An approach to using digital signature support in PDFs and a `pdfsign` tool which supports this is described in Section 7. We conclude with a summary and ideas for future work in the final Section 8 on page 54.

*Not only are PDF files that contain JBIG2 compressed information easier to send and share, but they are easier to store, they display rapidly online, and they are OCR ready.*  
(Franc Gagnon, 2010)

## 2 Introduction to JBIG2

JBIG2 is a standard for the compression of bitonal images developed by Joint Bi-level Image Experts Group. These are images that consists of two colors only (usually black and white). The main area of such images is scanned text. JBIG2 was published in 2000 as an international standard ITU T.88 [17] and one year later as ISO/IEC 14492 [6]. It typically generates files that are three to five times smaller than Fax Group 4 and two to four times smaller than JBIG1, which was the previous standard released by Joint Bi-level Image Experts Group) [12]. JBIG2 also supports lossy compression that increases the compression ratio several times without any noticeable visual difference when compared with lossless mode. Lossy compression without noticeable lossiness is called *perceptually lossless coding*. Scanned text often contains flyspecks (tiny pieces of dirt) and perceptually lossless coding can help get rid of the flyspecks and thus increase the quality of the output image.

### 2.1 Basic Principles of JBIG2

The content of each page may be segmented into three regions — text, halftone and generic. Text regions contain text, halftone regions contain halftone images<sup>1</sup> and generic regions contain everything else. In some situations it is better to use text regions rather than generic ones and in other situations, the converse is true. JBIG2 encoder segments text regions into their constituent symbols. These symbols must then be encoded. JBIG2 uses modified versions of Arithmetic and Huffman coding. Huffman coding is used mostly by faxes because of its lower computation demands, even though Arithmetic coding does give slightly better results.

<sup>1</sup> More about halftone can be found at <http://en.wikipedia.org/wiki/Halftone>

```

2 0 obj << /DecodeParms
      << /JBIG2Globals 1 0 R >>
      /Width 3265 /BitsPerComponent 1 /Height 4911
      /Filter /JBIG2Decode
      /Subtype /Image
      /Length 4582
      /ColorSpace /DeviceGray
      /Type /XObject
    >>
  stream
  ...
endstream

```

**Fig. 1.** Example of storing JBIG2 image in PDF document

JBIG2 also supports the multi-page compression that symbol coding uses (coding of text regions). Any symbol that is frequently used on more than one page is stored in a global dictionary. They only need to be stored once, thereby reducing the space needed to store documents.

## 2.2 JBIG2 in PDF

Since PDF version 1.4 (2001, Acrobat 5, see 3<sup>rd</sup> edition of the PDF Reference book) support for JBIG2Decode filter [15] has been embedded. This allows using images compressed according to standard JBIG2. This support has allowed JBIG2 standard to quickly spread far and wide without placing any burden on the end user.

PDF discards headers and some other data from JBIG2 images and sends this information to the PDF dictionary associated with the image object stream as can be seen in Figure 1.

## 2.3 JB2 in DjVu

DjVu [4] is an open digital document format with advanced compression technology and high performance value. It was developed at AT&T Labs to solve the problem of transporting documents containing high resolution scanned images via the Internet. It became an alternative format to PDF. DjVu was initially considered to be vastly superior to PDF, but since PDF version 1.4 (support of JBIG2 in PDF) this is no longer the case.

DjVu encoded images consist of three parts—foreground image, background image and mask image. The first two are low-resolution colored images and the last is a high-resolution bi-level image. A mask image will show if a color from the background or the foreground image should be used. For compressing the foreground and the background image, a compression algorithm called IW44 is used. For compressing the mask image a method called JB2 is used.

The JB2 algorithm is a variation of AT&T's proposal to the upcoming JBIG2 standard. The basic ideas behind JB2 are very similar to JBIG2. The basic image is first segmented into individual marks (connected components of black pixels). The marks are clustered hierarchically based on similarity using an appropriate distance measure. While the image is coded for each mark (symbol), an identifying index and position relative to that of the previous mark is specified. Marks are coded using a statistical model and arithmetic coding. Some are coded directly and some are coded indirectly based on previously coded marks. If it is the first occurrence of the mark it is coded directly, the bitmap also being coded; if not, it is coded indirectly with only its position and reference being coded.

For the clustering and the conditional encoding of marks, an algorithm called "soft pattern matching" [11] is currently used.

Unlike OCR, JB2 coding solves the problem of substitutional errors. If we have an imperfectly scanned symbol (e.g. due noise), it can be improperly matched and treated as a totally different symbol. This is one of the reasons why no OCR engine has one hundred percent accuracy. In OCR, it is necessary to substitute a symbol, whereas in JB2 and JBIG2 we can easily place this in the dictionary as a new symbol instead of being uncertain if it is a correct substitution.

*The most simple methods are used that do not significantly sacrifice performance.*  
(Adam Langley)

### 3 Jbig2enc and its Improvement

Jbig2enc [13] is an open-source encoder developed with Google support by Adam Langley under an Apache License. It uses the Leptonica library [2] for image manipulation. Leptonica takes care of rendering text, comparison of symbol components, aligning, etc.

Jbig2enc supports only arithmetic coding and instead of halftone regions it uses generic regions. It has embedded support for creating an output in a format suitable for PDF documents.

If we use symbol coding, some kind of lossy coding with a vision to be perceptually lossless is always possible. This is because almost all symbols in a document vary slightly since they are not identical at the pixel level. For this purpose a thresholding value in jbig2enc is used — it says how similar two symbols must be to be considered equivalent. The default value used is 0.85, meaning that two symbols must be at least 85% the same to be considered equivalent. The maximum value allowed by the jbig2enc encoder is 0.9.<sup>2</sup>

#### 3.1 Modification of Jbig2enc

We are improving the perceptually lossless coding of the encoder, jbig2enc, which is a component of pdfJbIm (see Section 4).

<sup>2</sup> A value of 0.85 is usually more than sufficient but for very poor quality documents it might not be optimal.

At this point in its development we are trying to find accumulations of differences between pairs of symbols<sup>3</sup>. To obtain the differences between two symbols we apply an operation XOR and we get their XORed bitmap.

To find local differences we divided an XORed bitmap (represented as a matrix) into submatrixes<sup>4</sup> and we count foreground pixels of four adjacent submatrixes if they contain enough different pixels to form a line or a point. Four adjacent submatrixes are counted because an accumulation could be placed exactly on the border of the submatrix. We use different segmentation (different sizes of subimages) for each type of shape. This method enables us to find the accumulation of differences in shapes as points or lines, whether vertical, diagonal, or horizontal.

If we find an accumulation bigger than a counted threshold value (counted differently for each size of a symbol) we consider these two symbols as different. If we do not find such an accumulation of differences, we consider these symbols equivalent—all references to these templates are set to point only to one template and the second one is deleted.

This method for comparing symbols is run as an additional method after the Leptonica classifier. With this new algorithm we are able to increase the compression ratio of the encoder `jbig2enc` by about eight percent, even for images with relatively bad quality.

We are now working on embedding OCR tools and techniques that will enhance the comparison process of two symbols. It should allow us to decrease the size of the output image as much as possible to the size of the born-digital text.

As Figure 2 on the following page shows, the image before and after compression look the same at first sight but the size of the compressed image is less than 70% of the original one. But they are not exactly the same. There are slight differences which are shown in the third image in Figure 2 on the next page. Removing the slight differences between the same symbols would improve the quality of the output image.

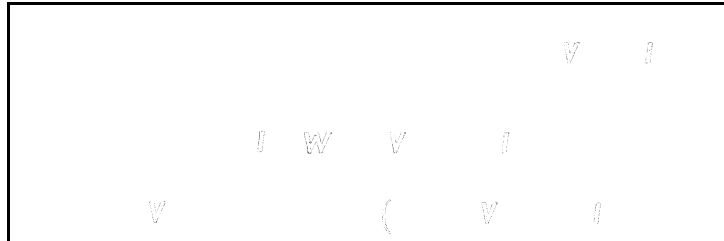
Running `jbig2enc` removes some of the differences between the same symbols which can make the output quality appear either better or worse. It is crucial that the right representative that will stay in the text is chosen. To guarantee the improvement of quality we always need to choose the best symbol from the equivalent ones. At this stage of development the first symbol is used as a representative symbol—it gives the same result as a random one (the quality remains mostly the same).

Embedding OCR tools could also help us to choose which symbol would be better as the representative one.

---

<sup>3</sup> Working only with templates (representative symbols) returned by Leptonica

<sup>4</sup> Subimages, bitmap segments

$$\begin{aligned}
 \mathbf{A} &= \left[ \lambda_1 \left( \mathbf{W} - \frac{u}{v} \mathbf{V} - \frac{kv - ul}{v} \mathbf{I} \right) + \lambda_2 \left( \frac{1}{v} \mathbf{V} - \frac{l}{v} \mathbf{I} \right) + \right. \\
 &\quad \left. + \lambda_3 \mathbf{I} \right] (\mathbf{W}^2 + \mathbf{V}^2 + m^2 \mathbf{I})^{-1} = \\
 &= (\lambda_1 \mathbf{V}_1 + \lambda_2 \mathbf{V}_2 + \lambda_3 \mathbf{I}) (\mathbf{W}^2 + \mathbf{V}^2 + m^2 \mathbf{I})^{-1} \\
 \mathbf{A} &= \left[ \lambda_1 \left( \mathbf{W} - \frac{u}{v} \mathbf{V} - \frac{kv - ul}{v} \mathbf{I} \right) + \lambda_2 \left( \frac{1}{v} \mathbf{V} - \frac{l}{v} \mathbf{I} \right) + \right. \\
 &\quad \left. + \lambda_3 \mathbf{I} \right] (\mathbf{W}^2 + \mathbf{V}^2 + m^2 \mathbf{I})^{-1} = \\
 &= (\lambda_1 \mathbf{V}_1 + \lambda_2 \mathbf{V}_2 + \lambda_3 \mathbf{I}) (\mathbf{W}^2 + \mathbf{V}^2 + m^2 \mathbf{I})^{-1}
 \end{aligned}$$


**Fig. 2.** Example of part of the page before (upper frame) and after (middle frame) compression by `jbig2enc`: the lower frame shows the differences

*I don't paint things. I only paint the difference between things.* (Henri Matisse)

#### 4 PdfJbIm

`PdfJbIm` [9,10] is a tool written in Java for re-compressing bitonal images placed in a PDF using symbol coding of the `jbig2enc` encoder. It has been developed for DML-CZ [8] and is still under development, with the goal being used in EuDML [14]. The main purpose is to decrease the size of PDF documents containing scanned text (mostly mathematical) and make it easier to transfer such documents via the Internet so that download time and costs can be reduced.

`PdfJbIm` replaces images with their re-compressed versions. It uses the `jbig2enc` encoder and two libraries for this purpose: `PDFBox` [7] and `iText` [5]. `PDFBox` is used to extract raw image data and convert them to suitable image format. `iText` is used for decrypting PDF if necessary and for replacing images with their re-compressed version. Information about images (their position and dimensions) are remembered during the process of extraction.

`jbig2enc` allows the use of symbol coding that is suitable for scanned text. If symbol coding is engaged then it segments an image containing text into

components (mostly one component contains exactly one symbol) and compare them. All the same components are put into a dictionary after which only references to the dictionary are used.

`pdfJbIm` uses the modified version of `jbig2enc` to achieve better results by enabling this option as an argument in a command line.

*Premature optimization is the root of all evil (or at least most of it) in programming.*  
(Donald Ervin Knuth)

## 5 Pdfsizeopt.py

`pdfsizeopt.py` [16] is a script written in Python (under GNU General Public License). It combines different Unix tools and scripts to optimize the size of PDF files without causing any damage. To optimize content streams the recommended procedure is to use the commercial optimizer PDF Enhancer or Adobe Acrobat first, then to run `pdfsizeopt.py` for optimizing mainly images and Type1 fonts. `pdfsizeopt.py` uses also `Multivalent tool.pdf.Compress` to do most of the remaining work. If `Multivalent` is installed, `pdfsizeopt.py` will run it automatically.

`pdfsizeopt.py` also removes duplicate and unused data, serializes strings more effectively, compresses streams by high-effort ZIP, removes page thumbnails since they can be created whenever they are needed. For these purposes it uses many different tools e.g. `ghostscript`, `pdftk`, `jbig2enc`, `sam2p`, `pngout`, `Multivalent` and `png22pnm`. For example it uses `ghostscript` to convert fonts to CFF (Compact Font Format—Type 2, Type 1C). Then it unifies subsets of the same fonts.

Optimized PDF files created by `pdfsizeopt.py` use some specifics that are used in PDF since version 1.5 which means that for viewing the PDFs is required Acrobat 6 or newer.

*We must develop knowledge optimization initiatives to leverage our key learnings.*  
(Scott Adams)

## 6 Combining Pdfsizeopt.py and PdfJbIm

To represent the results of these two optimizers we have applied them to the data from the DML-CZproject. We used PDF documents from the journal, *Applications of Mathematics*, which contains 19,690 pages in 1,799 papers. These documents are two-layer documents with OCR text for indexing and search (with possible errors) hidden behind the scanned bitmap.

In most documents the threshold value 0.85 is a sufficient guarantee of no visible loss but for documents of very poor quality, this does not suffice—we found some visible losses for some letters using the Cyrillic alphabet. For this reason we use a threshold value 0.9 which seems to be safe.

In Table 1 on the following page, we present the results of running the optimizer `pdfJbIm` and `pdfsizeopt.py`. To retrieve statistical data we use `pdfsizeopt.py` with the option `--stats` before and after running the optimizer



pipeline. The table shows how much is stored in each part of the PDF document as well the size of the whole document as *average* values that we have got from the optimized PDFs.

The most significant items for comparison are the rows *image objects* and *other objects*. This is because pdfJbIm only tries to reduce the size of images—it does not optimize other parts of PDF documents. As part of *image objects* is counted only size of objects specified in PDF as image objects without size of objects which they only are referencing to. Size of these kind of objects is counted in the section *other objects* (the size of a global dictionary is counted in this section as well).

**Table 1.** Average sizes (in bytes) of each type of PDF objects stored in *multi-page*, two layered (bitmap + OCR below) PDF documents created by pdftk. PdfJbImuses modified jbig2enc with enabled symbol coding. The threshold value is set to 0.9. Pdfsizeopt.py uses Multivalent and generic coding of jbig2enc, and does not use pngout (has minimal effect if used or not because JBIG2 is the most common compression method used, not ZIP)

	Original PDF	After using pdfJbIm	After using pdfsizeopt.py	After using both
Total size (in kB)	1,424	1,128	733	618
Content objects (in kB)	55	55	55	55
Font data objects (in kB)	464	464	77	77
Header	15	15	15	15
Image objects (in kB)	770	415	584	411
Linearized Xref table	0	0	0	0
Other objects (in kB)	127	185	17	75
Separator data	25	23	23	23
Trailer	120	121	107	102
Wasted between objects	0	0	0	0
Xref table	7,937	7,957	497	523

Table 2 on the next page shows how much the size of PDF documents and the sizes of image data were reduced using pdfJbIm and pdfsizeopt.py in comparison with the original PDF. The results are represented as a percentage of the size of optimized PDFs in comparison with the original PDF file. It uses the same PDF corpus as in the previous table.

Image data in this case are counted together with the data of other objects because the global dictionary is stored as a separate object. The size of this object is counted in the section *other objects*. Unfortunately, the global dictionary is not the only thing stored as *other* object. To at least partly distinguish which part of the data stored in other objects is a global dictionary, we use size of *other objects* after running both optimizers for summarizing. For pdfsizeopt.py there is nothing more to optimize in the global dictionary data. By combining

**Table 2.** New size of PDF document in comparison with the original one (generated by FineReader and  $\text{T}_\text{E}\text{X}$ ).

	Original PDF	After using pdfJbIm	After using pdfsizeopt.py	After using both
Size of whole PDF (in %)	100	79.21	51.49	43.41
Size of image and other objects (in %)	62.96	34.44	42.21	34.12

these two values, the results adequately indicate how effective each optimizer is for reducing the size of the image data.

As the table shows the pdfJbIm gives significantly better results than pdfsizeopt.py for images stored in multi-page PDF documents. We can see that the best solution is to run pdfJbIm first and pdfsizeopt.py on the result. It is better to run pdfJbIm first because of the performance. In most cases, images are compressed by JBIG2 and if they are, pdfsizeopt.py will not have to try other types of compression methods.

Instead of using optimizers, it could seem to be more beneficial to use the DjVu format instead of PDF. DjVu give us smaller documents than unoptimized PDFs but not as small as the optimized ones.

For comparison, we present the results given by running pdf2djvu with the same corpus as in the tables above. By transforming PDF documents to DjVu using pdf2djvu, the size of each document was reduced on average to 28.05%. This is half the size reduction compared to running both PDF optimizers (pdfJbIm and pdfsizeopt.py).

*You can't trust code that you did not totally create yourself.*

(Ken Thompson's address on receipt of the 1983 Turing Award)

## 7 Digital Signature of PDF Documents

PDF allows signing the documents digitally using a public key infrastructure — PKI, an architecture to verify the identities of people, web sites and computer programs on the Internet. Even sites delivering scientific content may want to prove — for several reasons — that the delivered documents originated in a trustworthy chain from author via publisher to the digital library. Digital signatures *guarantee the identity* of the data provider, confirming *data integrity* and making the *authorship undeniable*. It is important in cases of plagiarism, and it may also increase the credit of a project site — it may even increase the ranking of the site by some search engines. All PDF documents of the DML-CZ project are digitally signed with the DML-CZ certificate, and score high in Google searches.

It is hardly economical to sign thousands of PDFs manually, thus a batch digital stamper program pdfsign has been developed [3]. It is implemented in Java using the iText library [5]. The hash function of class SHA-2 (SHA-512) is used when signing. Also, the mechanism of *Timestamp Authority* is used as we expect the documents to be valid for a long time.

`pdfsign` is applied at the very end of the PDF generation workflow, after all OCR, cover page typesetting and page merging, optimization and compression steps have been performed. Signing a document usually increases the PDF size by several bytes only, and does not depend on previously applied compression. For safety reasons, digital signatures are added on a different site from the one that houses the digital library.

Similar program `JsignPdf` has been posted on <http://sourceforge.net/projects/jsignpdf/> recently. It uses the older hash function SHA-1, which NIST does not consider safe. `JsignPdf` does not provide support for batch processing of large volumes of PDFs as `pdfsign` does.

## 8 Conclusion and Future Work

We have shown how to significantly compress PDFs with bitonal scanned image—when running both optimizers (`pdfJbIm` and `pdfsizeopt.py`) we were able to reduce the size of PDF documents to less than half of the original size. We anticipate even better results when all the planned improvements in our `pdfJbIm` will be made. Improvements in PDF compression were orthogonal to the digital signature which is added by the `pdfsign` batch PDF stamper software to increase the trustworthiness of the content of a digital library.

We are now developing the integration of OCR tools to `jbig2enc` to improve the comparison process in symbol coding. This approach should help us to decide which symbol to choose as a representative one to further improve page rendering and to increase readability.

### Acknowledgement

We acknowledge the partial support of the European Commission, under the Grant Agreement #250,503 (Project EuDML) and the partial support of Masaryk University by grant for student research assistants #22,525/2010.

### References

1. Bartošek, M., Lhoták, M., Rákosník, J., Sojka, P., Šárfy, M.: DML-CZ: The Objectives and the First Steps. In: Borwein, J., Rocha, E.M., Rodrigues, J.F. (eds.) CMDE 2006: Communicating Mathematics in the Digital Era, pp. 69–79. A. K. Peters, MA, USA (2008)
2. Bloomberg, D.: Leptonica. [online] (2010), [cit. 2010-04-25], <http://www.leptonica.com/jbig2.html>
3. Bočák, P.: Digitálně podpísované PDF dokumenty (Bachelor thesis written in Czech, Digital signatures of PDF documents). Masaryk University, Faculty of Informatics (advisor Petr Sojka), Brno, Czech Republic (2008)
4. Bottou, L., Haffner, P., Howard, P.G., Simard, P., Bengio, Y., Le Cun, Y.: High Quality Document Image Compression with DjVu. *Journal of Electronic Imaging* 7(3), 410–425 (1998), <http://leon.bottou.org/papers/bottou-98>

5. Bruno, L.: IText PDF. [online] (2009), <http://www.itextpdf.com/>
6. Committee, J.: 14492 FCD. ISO/IEC JTC 1/SC 29/WG 1 (1999), <http://www.jpeg.org/public/fcd14492.pdf>
7. Foundation, T.A.S.: Apache PDFBox – Java PDF Library. [online] (2010), <http://pdfbox.apache.org/>
8. Hatlapatka, R.: JBIG2 komprese (Bachelor thesis written in Czech, JBIG2 compression). Masaryk University, Faculty of Informatics (advisor Petr Sojka), Brno, Czech Republic (2010)
9. Hatlapatka, R.: PDF Recompression using JBIG2. [online] (2010), <http://nlp.fi.muni.cz/projekty/eudml/pdfRecompression/>
10. Hatlapatka, R.: Source codes of pdfjblm. [online] (2010), <http://code.google.com/p/pdfrecompressor/>
11. Howard, P.: Text image compression using soft pattern matching. *Computer Journal* 40(2/3), 146–156 (1997)
12. ISO/IEC JTC1/SC29/WG1: JBIG Maui Meeting Press Release (December 1999), <http://www.jpeg.org/public/mauijbig.pdf>
13. Langley, A.: Homepage of jbig2enc encoder. [online], <http://github.com/ag1/jbig2enc>
14. Sylwestrzak, W., Borbinha, J., Bouche, T., Nowiński, A., Sojka, P.: EuDML—Towards the European Digital Mathematics Library. In: Sojka, P. (ed.) *Proceedings of DML 2010*. Masaryk University Press, Paris, France (Jul 2010)
15. Adobe Systems Incorporated: Adobe Systems Incorporated: PDF Reference, pp. 90–100. Adobe Systems Incorporated, sixth edn. (2006), [http://www.adobe.com/devnet/acrobat/pdfs/pdf\\_reference\\_1-7.pdf](http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference_1-7.pdf)
16. Szabó, P.: Optimizing PDF output size of  $\text{\TeX}$  documents. *TUGboat* 30(3), 112–130 (2009), [cit. 2010-04-26], <http://code.google.com/p/pdFSIZEOPT/>
17. Union, I.T.: ITU-T Recommendation T.88. ITU-T Recommendation T.88 (2000), <http://www.itu.int/rec/T-REC-T.88-200002-I/en>