Comment ne pas mettre

Les points sur les ȷ

Thierry Bouche & Bernard Desruisseaux

Institut Fourier
Laboratoire de mathématiques pures
Université Joseph Fourier – Grenoble

Recherche et développement
Corporate Software & Technologies
Montréal, Canada

Résumé. Les mathématiciens sont friands d'accents bizarres, qu'ils posent sur toutes les lettres de l'alphabet. Pour ce faire ils prennent le caractère de base et placent l'accent de leur choix au-dessus grâce à la primitive \accent de TEX. Malheureusement, il existe une lettre fournie uniquement en version «accentuée» par les fondeurs pour qui la typographie mathématique n'est pas essentielle : c'est le *j*. Nous discutons ici les diverses possibilités dont on dispose pour remplacer ce satané point par d'autres accents qui amusent beaucoup plus les mathématiciens.

1. Le problème

Notre problème tient en un mot : comment imprimer ĵ, ȷ̈, voire $\vec{\jmath}$ (plutôt que $\hat{\jmath}$, $\ddot{\jmath}$ ou $\vec{j}$) dès lors que la fonte utilisée n'est pas membre du club très fermé des fontes conçues pour TEX (c.f. [1]) ?

Notre approche sera essentiellement théorique, mais nous présenterons différentes solutions, étant entendu que la solution idéale serait que les dessinateurs de caractère prennent conscience des manques du jeu «standard» de glyphes actuellement proposés.

D'autre part notons que ce problème n'est pas *absolument* gratuit : on pourrait par exemple adapter nos méthodes pour construire des glyphes prévus (ou non...) dans le codage T1 de TEX, d'une façon plus satisfaisante[1].

1. On peut rêver d'un Ś dont l'accent serait celui du É plutôt que celui du é, comme ceci : Ś, ou d'une ligature fȷ utile pour les langues scandinaves.

2. Une couleur peut en cacher une autre

Grâce à la possibilité de contrôler la couleur depuis TEX [2] on peut simplement masquer un caractère ou une de ses parties en l'imprimant en blanc après l'avoir imprimé en noir. Ceci nécessite d'avoir une idée précise de l'ordre dans lequel TEX imprime les éléments de caractères, d'autant plus qu'on a des surprises : la primitive \accent imprime d'abord l'accent, *puis* le glyphe de base. Un moyen de cacher le point du j est donc d'imprimer un j, puis un rectangle blanc par dessus son point (les caractéristiques de ce rectangle sont faciles à déterminer : il doit couvrir une zone de même largeur que le caractère j, débutant au dessus du ı sans point, et s'achevant à la hauteur de j ; il est de plus facile avec l'interface de LATEX de faire croire à TEX que ce ȷ occupe une boite de la hauteur de celle de ı. Deux problèmes se posent immédiatement. Le premier est qu'un tel ȷ sans point est destiné à supporter des accents, mais que le rectangle blanc va cacher l'accent en même temps que le point s'il vient après (nous avons vu que les accents sont imprimés en premier par TEX) : il est donc nécessaire de modifier les macros d'accentuation de telle sorte que l'accent soit imprimé en dernier, ce qui n'est pas très compliqué. Le second problème est plus fâcheux : il n'existe pas de concept de «couleur de fond en cours d'utilisation» avec l'extension color, si bien qu'il n'est pas possible de cacher le point avec un rectangle de la couleur utilisée au moment de l'impression si jamais cette dernière n'est pas blanche (exemple typique : fonds grisés ou colorés pour mettre en valeur des énoncés importants) — nous introduisons la macro \couleurfond pour pallier à la main ce léger défaut. Il est néanmoins raisonnable de considérer cette «solution» comme inutilisable [3], ce qui ne nous empêchera pas de donner ci-dessous le code correspondant, et une figure illustrant la construction.

Macro \dotlessj cachant le point à l'aide d'un rectangle blanc, utilisée pour la Fig. 1 qui montre dans l'ordre : `j`, `\dotlessj`, `\vec\dotlessj`.

```
\newbox\dot@j
\newlength\dot@length
\newcommand\couleurfond[1]{\definecolor{dot-color}{cmyk}{#1}}
\couleurfond{0,0,0,0}
\newcommand\dotlessj{%
\sbox\dot@j{j\/}%
\settoheight\dot@length{j}%
\addtolength\dot@length{-1ex}%
\raisebox{0pt}[\dp\dot@j][1ex]{j}%
\kern-\wd\dot@j\raisebox{0pt}[\dp\dot@j][1ex]{%
{\color{dot-color}
```

2. Nous nous appuierons ici sur l'extension color de David Carlisle.

3. Remarquons cependant que c'est la plus indépendante des drivers puisque les bonnes extensions graphiques génèrent le code approprié en fonction du driver utilisé. Ainsi, c'est la seule solution qui marcherait même pour des fontes au format bitmap, metafont ou true type.

```
\rule[1.1ex]{\wd\dot@j}{\dot@length}}}%
```

FIG. 1 – *Construction ne reposant que sur des macros TEX.*

À compléter par une modification de la définition de \vec, qui permet d'obtenir le dernier exemple de la FIG. 1 :

```
\let\vec@@ori\vec
\renewcommand\vec[1]{\ifx#1\jmath
\text{\rlap{\it\dotlessj}}\vec{\phantom{\dotlessj}
\else
\vec@@ori{#1}}}
```

3. Un petit clip peut cacher un grand flop

Comme la précédente solution n'est pas conceptuellement très satisfaisante, et que POSTSCRIPT est un langage de programmation aux ressources variées, l'étape suivante est, en se limitant désormais délibérément aux fontes et aux imprimantes POSTSCRIPT, d'utiliser les possibilités offertes par ce langage. L'idée qui vient immédiatement à l'esprit est qu'il suffit de ne pas imprimer ce qui dépasse au dessus de la ligne des bas de casse pour se débarrasser du point du j, comme de tout accent d'ailleurs. Ceci est permis par l'opérateur clip qui permet de n'imprimer (temporairement) que la portion de graphique contenue à l'intérieur du domaine délimité par un contour quelconque appelé le clippath. Il suffit par conséquent de déterminer un chemin adéquat, puis d'imprimer le j présent dans toute fonte qui se respecte pour obtenir le ȷ sans point tant désiré. Les problèmes évoqués plus haut devraient alors disparaître, car le glyphe imprimé n'empiétera pas hors du contour voulu.

Il est relativement facile de réaliser ce programme en POSTSCRIPT pur : on utilise un algorithme pour obtenir la *BoundingBox*[4] des caractères *j*, *ı* respectivement, à l'aide de laquelle on peut fabriquer la *BoundingBox* du caractère *ȷ* virtuel, et l'utiliser comme chemin de découpe. La FIG. 2 donne une idée des étapes de ladite construction. Comme l'a déjà fait remarquer S. RAHTZ [3] il est possible d'inclure directement des special POSTSCRIPT à l'aide de fontes virtuelles, donc de définir un caractère \j valide du point de vue de TEX (métriques déduites de celles des caractères *ı*, *j*) et interprété correctement par tout logiciel POSTSCRIPT.

4. On appelle *BoundingBox* d'un caractère les coordonnées caractérisant le plus petit rectangle contenant le dessin de ce caractère.

FIG. 2 – *Construction en pur* POSTSCRIPT.

Pour ce faire, le plus simple serait d'ajouter la définition suivante au fichier latin.mtx du logiciel fontinst [3, 2] :

```
\setglyph{dotlessj}
   \glyphspecial{ps: gsave newpath 0 0 moveto (\string\31) true charpath
   flattenpath pathbbox /IHeight exch def pop pop pop grestore gsave
   newpath 0 0 moveto (\string\152) true charpath flattenpath pathbbox
   pop
   exch /JDepth exch def /JRight exch def /JLeft exch def grestore
   gsave newpath}
   \push
   \moveup{\neg{\depth{j}}}
   \glyphspecial{ps: JLeft JDepth rmoveto JLeft neg JRight add 0 rlineto
      0 JDepth neg IHeight add rlineto JLeft neg JRight add neg 0
      rlineto 0 JDepth neg IHeight add neg rlineto closepath gsave
    stroke grestore clip}
   \pop
   \glyph{j}{1000}
   \glyphspecial{ps: grestore}
         \resetdepth{\depth{j}}
         \resetwidth{\width{j}}
         \resetitalic{\italic{dotlessi}}
         \resetheight{\height{dotlessi}}
\endsetglyph
```

Mais ceci a de gros inconvénients. En effet, le caractère *j* faisant partie du codage ASCII, il occupe la même place dans tous les codages raisonnables utilisés pour une fonte de texte. *A contrario*, le caractère *ı* ayant une utilité «uniquement typographique», il n'est en général pas codé (absent des codages d'échange usuels comme ASCII ou Isolatin, codé de 3 façons différentes dans le codage standard d'Adobe, et dans les deux standards de TEX : OT1, T1). Par conséquent, le code POSTSCRIPT littéral que nous présentons ci-dessus est très dépendant du contexte dans lequel il va réellement se trouver au sein du fichier POSTSCRIPT ultimement produit par dvips ou équivalent. Avec la mise en œuvre actuelle des fontes POSTSCRIPT pour dvips, les fontes de texte utilisent

le codage 8r dans lequel /dotlessi est codé à la place qu'il occupe en OT1 (code octal 152 ci-dessus, à comparer au code 365 en Adobe Standard Encoding), mais il est à peu près impossible de prédire la fonte courante en un point du fichier PostScript où rien n'est imprimé. Comme notre code ci-dessus doit *nécessairement* effectuer divers calculs *avant* d'imprimer le premier pixel, il est extrêmement fragile et s'est révélé inutilisable, par exemple pour obtenir un \jmath dans une fonte mathématique composite dont les lettres italiques proviennent d'une fonte PostScript standard.

Fig. 3 – *Construction relativement imprécise obtenue en injectant des commandes PostScript paramétrées par* fontinst.

A. Jeffrey, l'auteur de fontinst, nous a communiqué une méthode permettant de calculer le clippath par avance en utilisant la connaissance que fontinst a de la fonte considérée. Cette méthode permet de rendre la fonte virtuelle générée aussi indépendante que possible du pilote d'impression. Voici le code correspondant (après quelques retouches esthétiques) :

```
\setglyph{dotlessj}
 \push
   \glyphspecial{ps: gsave gsave}
   \moveup{\neg{\depth{j}}}
   \movert{\neg{\width{j}}}
   \glyphspecial{ps: currentpoint /JDepth exch def /JLeft exch def}
   \moveup{\add{\depth{j}}{\mul{1.1}{\height{dotlessi}}}}
   \movert{\mul{\width{j}}{3}}
   \glyphspecial{ps: currentpoint /IHeight exch def /JRight exch def
       grestore  JLeft JDepth moveto JLeft neg JRight add 0 rlineto
      0 JDepth neg IHeight add rlineto JLeft neg JRight add neg 0
      rlineto 0 JDepth neg IHeight add neg rlineto closepath  clip}
  \pop
  \glyph{j}{1000}
  \glyphspecial{ps: grestore}
         \resetdepth{\depth{j}}
         \resetwidth{\width{j}}
         \resetitalic{\italic{dotlessi}}
         \resetheight{\height{dotlessi}}
\endsetglyph
```

On remarque que le gros défaut de ce code est l'imprécision. En effet, fontinst (à l'image de TEX) ne permet pas d'accéder à la *BoundingBox* exacte qui peut être beaucoup plus large que l'espace occupée par le caractère (le *j* étant un exemple typique

de ce phénomène, son jambage empiétant franchement sur la gauche). D'où les coefficients multiplicatifs qui marchent empiriquement, mais limitent la généricité de cette approche.

4. Comment ajouter des caractères à une fonte

Ce problème de point sur le j est un grain de sable que l'on peut déplacer dans la «chaîne graphique», du source LaTeX à la fonte elle-même, en passant par l'approche hybride à base de fontes virtuelles que nous venons de voir. Nous concluons cet article par la présentation d'une méthode qui permet de corriger les «oublis» du dessinateur de caractère en ajoutant automatiquement à toute fonte PostScript vectorielle les caractères qui peuvent être reconstitués algorithmiquement. Avec cette approche, la fonte considérée, une fois recodée en 8r, possède le caractère /dotlessj, et il n'est plus nécessaire de toucher à l'interface ni de modifier des macros TeX. En fonction de l'installation utilisée, on peut cependant avoir des problèmes à la prévisualisation de documents utilisant ce caractère. Notons toutefois que, si l'on dispose de ghostscript, il est possible d'obtenir une image *bitmap* du caractère ainsi construit.

Références

[1] Thierry Bouche. *Sur la diversité des fontes mathématiques*, Cahiers GUTenberg n° 25, 1–24, Décembre 1996.

[2] Alan Jeffrey. *The fontinst package*, documentation accompagnant le logiciel, Juin 1994.

[3] Sebastian Rahtz.?? TUGboat 1995?