RESET
Recherches en sciences sociales sur Internet

11 | 2022
Codes. L'informatique comme elle s'écrit

Éléments pour une sociologie de l'activité de programmation

Inscriptions provisoires, chaînes de référence et indexations

Elements for a sociology of programming activities: Provisional inscriptions, chains of reference, and indexations

Florian Jaton

Édition électronique
URL : https://journals.openedition.org/reset/3829
DOI : 10.4000/reset.3829
ISSN : 2264-6221

Éditeur
Association Recherches en sciences sociales sur Internet

Référence électronique
Florian Jaton, « Éléments pour une sociologie de l'activité de programmation », *RESET* [En ligne], 11 | 2022, mis en ligne le 31 mars 2022, consulté le 06 octobre 2022. URL : http://journals.openedition.org/reset/3829 ; DOI : https://doi.org/10.4000/reset.3829

Éléments pour une sociologie de l'activité de programmation

Inscriptions provisoires, chaînes de référence et indexations

Elements for a sociology of programming activities: Provisional inscriptions, chains of reference, and indexations

Florian Jaton

1. Introduction

1 Depuis au moins une vingtaine d'années, les études critiques sur les effets sociaux des méthodes informatiques de calcul – souvent appelées « algorithmes » – se sont multipliées[1]. Les monographies de Bucher (2018), Noble (2016), O'Neil (2016) ou Steiner (2012) ont, parmi d'autres, mis en exergue les processus de discrimination et de mise en invisibilité subrepticement induits par le recours généralisé à des dispositifs algorithmiques. Ces analyses, importantes, ont contribué à problématiser l'objet « algorithme » et ont participé à élargir l'horizon du débat public quant aux technologies de l'information et de la communication (Crawford and Calo, 2016).

2 Pour autant, l'analyse des effets sociaux des algorithmes – toute importante qu'elle soit – tend parfois à instaurer une distance problématique avec son objet d'étude : à force de considérer les algorithmes de loin et à l'aune de leurs effets, ceux-ci paraissent à la fois puissants et intangibles (Pasquale, 2016). C'est ce que Ziewitz nomme le « drame algorithmique » (2016) : un discours critique désarmant qui répète à l'envi que les algorithmes sont d'autant plus puissants qu'ils sont impénétrables, ce qui les rend, à leur tour, d'autant plus puissants.

3 En réaction à cette ligne de recherche critique et au drame algorithmique qu'il tend à instaurer, plusieurs enquêtes récentes ont fait le pari d'investir les lieux où les algorithmes sont façonnés, rendus publics et maintenus (Bechmann et Bowker, 2019 ; Cardon, 2015 ; Grosman et Reigeluth ; Henriksen et Bechmann, 2020 ; Mackenzie, 2017). L'objectif plus ou moins affiché de cette ligne analytique est le suivant : au lieu de se

limiter à l'étude des effets des algorithmes, il est maintenant temps d'enquêter sur les *causes de ces effets* pour mieux les changer. Parmi ces enquêtes, hétérogènes, certaines se sont rattachées aux ethnographies de laboratoire telles que définies au sein des *Science & Technology Studies* (Jaton, 2017 ; 2019). Enquêter sur les algorithmes revient dès lors à enquêter sur un travail spécialisé, mais ordinaire et situé, dont les produits sont de *nouveaux algorithmes*, publiés notamment dans des articles académiques ou des actes de conférence et réutilisés, parfois, par des acteurs publics ou privés[2].

4 Documenter le façonnage des algorithmes en recourant au genre analytique de l'ethnographie de laboratoire – parmi d'autres possibles – implique de s'engager dans des situations différenciées (Jaton, 2021a). Discussions de couloir, séminaires, pauses café, réunions de projets : autant de *cours d'actions*[3] qu'il faut bien, d'une manière ou d'une autre, faire entrer dans des carnets de notes et logiciels d'enquête afin d'en extraire des propositions empiriquement fondées à même de constituer, si tout se passe bien, un compte-rendu réaliste de ce qui se fait à certains endroits, à certains moments. Jusque là, *business as usual* : l'enquête s'inscrit dans le « carré de l'activité » (Licoppe, 2008) et peut se construire sur de nombreux précédents (e.g., Latour et Woolgar, 2005 ; Lynch, 1985 ; Vinck, 1999).

5 La situation se complique lorsque l'ethnographe remarque, assez vite, qu'une partie importante du quotidien des *computer scientists* consiste à rédiger des listes numérotées de symboles permettant à leurs ordinateurs d'opérer des transformations précises sur des données numériques. Si iel souhaite rendre compte de la manufacture des algorithmes d'une façon un tant soit peu réaliste, l'ethnographe se doit ainsi également de documenter les *situations de programmation* qui se déroulent sur son terrain d'enquête.

6 À partir de là, deux problèmes se posent. Le premier, méthodologique, est la difficulté à saisir ce qui se passe dans ces situations : particulièrement emmêlées et solliciteuses, celles-ci se prêtent souvent mal à l'interrogation intrusive et à la prise de notes détaillées. Le deuxième problème, connexe, est la quasi-absence d'enquêtes s'étant risquées à ce type d'exercice. S'il existe de nombreux travaux d'inspiration sociologique sur l'organisation du travail logiciel (e.g., Brooks, 1975 ; Kidder, 2000 ; Rosenberg, 2008) et sur les communautés de programmeur·euse·x·s (e.g., Alcaras, 2020 ; Couture, 2019 ; Demazière et al. 2007), très rares sont ceux qui ont tenté de suivre empiriquement les cours d'action prenant part à l'activité de programmation.

7 Il y a bien quelques exceptions, à commencer par le travail de Flor et Hutchins (1991) sur l'activité à la fois située et distribuée de maintenance logicielle. Mais si cette étude fournit des outils précieux pour enquêter sur les pratiques effectives de programmation – dont une représentation graphique qui a fortement inspiré le formalisme que j'introduis en section 2 –, elle n'a hélas pas fait l'objet d'une démarche systématique, ses deux auteurs ayant ensuite bifurqué vers d'autres thématiques de recherche. Il faut également noter l'apport ethnométhodologique de Button et Sharrock (1995) sur le travail d'écriture et de lecture du code informatique. Mais si cette enquête, dont l'accès reste difficile, a pu poser les bases d'une sociologie des activités effectives de codage – notamment en proposant de ralentir l'analyse et de se focaliser sur les micro-évènements tels que vécus dans l'espace de travail des programmeur·euse·x·s –, sa démarche et ses propositions n'ont pas été, à ma connaissance, portées plus loin. Enfin, il y a l'article de Mackenzie et Monk (2004) : une enquête ethnographique sur l'*Extreme Programming* (XP), une approche de conception

logicielle caractérisée par son emphase sur l'épure et le travail en binôme et qui tranche avec certaines approches hiérarchiques traditionnelles en ingénierie logicielle. Mais là encore, si cette démarche analytique aboutit à des propositions notables – notamment le fait que le XP contribue à réincarner les pratiques individuelles de programmation en tant que processus de collaboration et d'organisation – il semble qu'elle a été le résultat d'une collaboration conjoncturelle et non pas le point de départ d'une ligne de recherche systématique.[4]

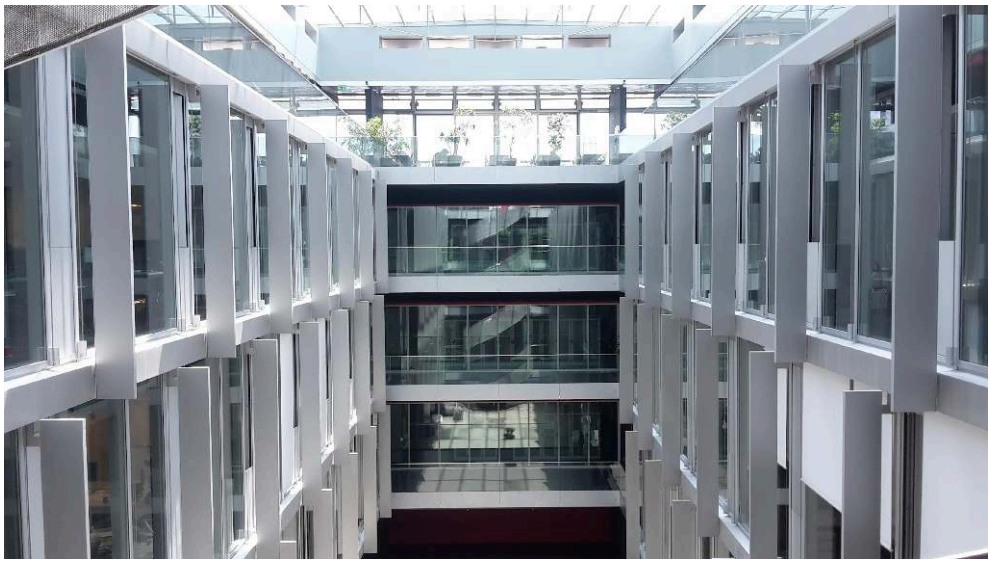
8 Les raisons d'une telle retenue vis-à-vis de l'étude ethnographique de l'activité de programmation sont encore à explorer[5]. Toujours est-il que pour qui souhaite acquérir une compréhension détaillée de cette pratique située, tout reste à faire, ou presque. Prenant acte de la situation, cet article propose quelques outils et analyses pour tenter de la changer. Dans un premier temps, il expose une technique d'investigation, ainsi qu'un formalisme, permettant de rester au plus près du déroulement séquentiel des situations de programmation. Dans un deuxième temps, il analyse des matériaux récoltés selon cette méthode d'enquête – nommée ici *sociologie de l'activité de programmation* – et suggère qu'une part importante des pratiques de codage consiste, parfois, à aligner des inscriptions pour renseigner l'état d'entités distantes (interpréteurs, compilateurs, processeurs) et, en retour, indexer un emplacement au sein d'un document numéroté. L'article finit par discuter les perspectives nouvelles sur l'étude sociale du code induites par cette démarche analytique.

2. Présentation des matériaux

2.1. Le Lab

9 Les matériaux suivants sont issus d'une enquête ethnographique menée entre 2013 et 2016 au sein d'un laboratoire d'informatique – nommé ici le Lab – spécialisé dans le traitement de l'image digitale. Le Lab faisait partie de la faculté d'informatique d'une grande université européenne et comportait entre dix et vingt collaborateur·rice·x·s (étudiant·e·x·s en doctorat pour la plupart), en fonction des projets de recherche alloués par ses partenaires publics et privés. Les bureaux du Lab étaient alignés le long d'une des quatre ailes du troisième étage d'un bâtiment iconique inauguré en 2004 (voir illustrations 2.1, 2.2 et 2.3). Grâce au soutien de sa directrice, j'ai pu bénéficier d'une bourse doctorale interdisciplinaire me donnant le statut de membre à part entière du Lab pour une durée de 28 mois. Le but affiché de ce projet de thèse en étude sociale des sciences et des techniques était de documenter certaines des pratiques par lesquelles de nouveaux algorithmes en traitement de l'image en venaient à être conçus et publiés. La démarche analytique de l'enquête était résolument ethnographique et s'appuyait, pour beaucoup, sur les travaux classiques de Lynch (1985), Latour et Woolgar (2005), Suchman (2006) et Vinck (1999).

Illustration 2.1. Le bâtiment principal de la faculté d'informatique du Lab.



De part et d'autre du patio central, des rangées de bureaux et de salles de séminaires. Au centre de l'image, dans des salles climatisées, trois batteries de serveurs. Au dernier étage, illuminé, on aperçoit l'entrée de la cafétéria de la faculté.

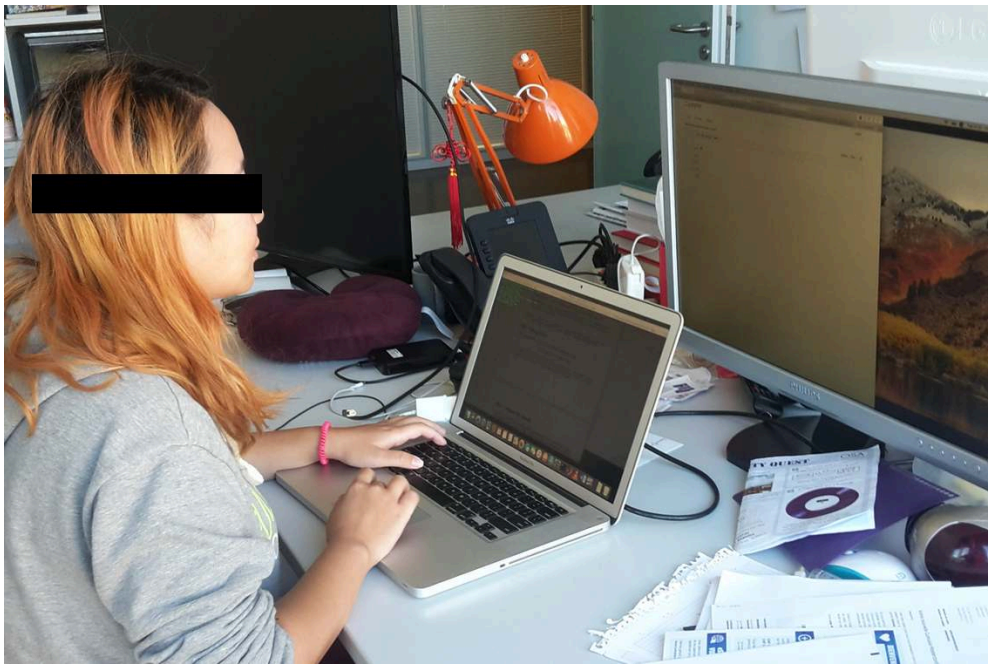
Source : auteur.

Illustration 2.2. Le hall du Lab.

Sur la gauche, derrière des portes fermées, la cafétéria privée et la salle de séminaire du Lab. À droite, sept bureaux occupés la plupart du temps par deux chercheur-euse-x·s.

Source : auteur.

Illustration 2.3. Dans l'un des bureaux du Lab.



Les chercheur·euse·x·s étaient généralement face à face, bien que derrière un à trois larges moniteurs. Source : auteur.

10 La publication d'un nouvel algorithme de traitement d'image est un processus tourbillonnaire qui implique des activités et situations variées. Mais parallèlement à la collecte et à la problématisation de données numériques, à la mise en place de bases de données référentielles[6], à la formulation mathématique des relations entre les éléments de ces bases de données[7] et aux nombreuses réunions, discussions informelles, séminaires et conférences qui aident à la réalisation et la coordination de tous ces cours d'action, il y a des situations plus ou moins longues de programmation durant lesquelles des listes d'instructions numérotées sont rédigées afin de rendre une machine informatique capable de traiter correctement des ensembles de données numériques. Ce sont ces cours d'action – qui ont un début, une fin et qui ressemblent parfois à ce qui peut se passer dans d'autres situations professionnelles connexes (ex. ingénierie, physique, sciences sociales, finance) – dont je vais rendre compte, partiellement, dans cette étude de cas.

11 Le problème qui est rapidement apparu lors de mon travail ethnographique au sein du Lab a été de savoir comment documenter les situations de programmation. Tout d'abord, comme les lignes de code écrites durant ces situations étaient passablement énigmatiques (je n'avais, à cette époque, pas encore de formation rigoureuse en informatique), il m'était difficile d'avoir une emprise sur ce qui se passait. La configuration de ces signes cryptiques sur les moniteurs de mes informateur·rice·x·s ne cessait jamais de changer ; de nouveaux symboles étaient ajoutés, d'autres effacés, d'autres corrigés. De plus, ces situations semblaient souvent engageantes pour les personnes concernées, ce qui m'empêchait de les interrompre pour leur poser des questions sur ce qu'elles étaient en train de faire. Durant ces moments qui pouvaient être particulièrement intenses, j'étais clairement l'élément de trop.

12 Afin de pallier ces problèmes méthodologiques, j'ai conçu mon propre projet de traitement d'image avec l'aide des membres du Lab. Après plusieurs séminaires de laboratoire, il a été décidé collectivement que je devais tenter de concevoir un modèle de prétraitement qui trierait automatiquement des ensembles de photographies numériques. Le but était de repérer certaines configurations de pixels qui feraient écho à d'autres processus de segmentation spécifiques qui étaient alors en cours de développement au sein du Lab. Ce modeste projet était explicitement conçu pour me forcer à acquérir les bases de plusieurs langages de programmation et à me familiariser avec la discipline du traitement du signal. Élément important : le projet comprenait également une « clause d'aide » qui me permettait de demander assistance aux membres du Lab lorsque j'étais dans une impasse de programmation. Cette méthode, clairement *ad hoc*, s'est avérée particulièrement utile. Elle a d'abord permis de me familiariser avec plusieurs langages de programmation[8] ; peu à peu, tous ces signes énigmatiques ont commencé à prendre sens. Elle a également rendu les membres du Lab plus à l'aise durant les situations de programmation que je tentais de documenter. Comme le projet avait été conçu collectivement et pouvait, potentiellement, être utilisé pour de futurs algorithmes et articles, les membres du Lab l'ont trouvé relativement pertinent. Et comme lesdites « séances d'aide » ne concernaient pas directement leurs propres projets, iels se sont également senti·e·x·s plus à l'aise lorsque je prenais des notes à leur côté et leur posais des questions alors qu'iels programmaient. Enfin, et c'est peut-être le plus important, cette méthode m'a permis de mieux équiper et documenter les situations de programmation : en plus de la tenue de notes décrivant les faits et gestes de la personnes qui programmait à côté de moi, je pouvais également enregistrer les moniteurs de mon ordinateur – un écran principal de 28 pouces relié à un écran secondaire de 15 pouces – ainsi que nos discussions. Pour les huit séances d'aide dont j'ai eu besoin pour mener à bien ce projet, j'ai ainsi obtenu des descriptions manuscrites, des enregistrements vidéo d'écrans et des enregistrements audio que j'ai pu ensuite analyser en profondeur (moyennant, il est vrai, un gros travail de transcription[9]).

13 Bien qu'instructifs à bien des égards, les matériaux recueillis lors de ces séances d'aide ont des limites. Comme les « scripts » – de petits programmes, sans lien avec la notion développée par Akrich (1989) – étaient principalement destinés à mon usage personnel, ils n'ont pas été directement conçus pour circuler au sein d'une communauté professionnelle de programmeur·euse·x·s comme c'est généralement le cas dans les entreprises logicielles et les projets libres ou *open-source*. En ce sens, des thématiques importantes telles que la production *in situ* d'intelligibilité lors de la consultation du code source n'ont pas pu être étudiées comme l'ont fait Button et Sharrock (1995) dans leur article pionnier, mais isolé, sur les pratiques de programmation. Néanmoins, comme nous le verrons plus loin, certaines de mes propositions pourraient bien leur faire écho.

2.2. PARSE

14 Les matériaux suivants sont tirés d'une séance d'aide au cours de laquelle GY – une étudiante en doctorat du Lab – écrivait un court programme que j'appellerai désormais PARSE et qui traitait des données que j'avais précédemment collectées – en m'inspirant des travaux en cours au sein du Lab – grâce à une épreuve de *crowdsourcing* en passant par la plateforme de l'entreprise *clickworker.com Inc*[10]. L'épreuve de *crowdsourcing* était

divisée en dix sessions. Pour chaque session, vingt à trente travailleur·euse·x·s européen·ne·x·s du clic devaient considérer cinquante photographies de paysages, de visages, d'oiseaux, de bâtiments, etc. Le contenu de ces photographies, extraites aléatoirement du site Flickr (en en passant toutefois par leur interface de programmation), était extrêmement varié. Pour chaque photographie, chaque travailleur·euse·x devait dessiner un ou plusieurs rectangles autour des parties de l'image qui attiraient son attention en premier lieu. Avant de passer à l'image suivante, chaque travailleur·euse·x devait également évaluer, de 1 à 7, la difficulté de ce choix d'étiquetage. Après les dix sessions, 254 travailleur·euse·x·s différent·e·x·s avaient chacun·e·x étiqueté cinquante images, pour un total de cinq cents images. Le numéro d'identification unique des images traitées, l'identifiant de la session, les coordonnées des rectangles dessinés et les notes attribuées à chaque tâche d'étiquetage ont ensuite été recueillis par une application web spécialement conçue pour le projet, puis rassemblées dans des fichiers texte (.txt) organisés comme dans la figure 2.1. Le contenu de ces fichiers .txt ainsi que les photographies utilisées pour l'épreuve de *crowdsourcing* constituaient les données sur lesquelles PARSE était amené à travailler.

Figure 2.1. Extrait d'un fichier .txt nommé « worker_05Waldave56jm9815.txt » tel que généré par l'application web à la fin de chaque session de *crowdsourcing*.

```
16714267603_cd60601b7f_b.jpg  1 startX:25px startY:32px width:450px height:361px
16705290404_d8de298f0e_b.jpg  5 startX:430px startY:76px width:260px height:414px startX:234px startY:227px width:189px height:216px
```

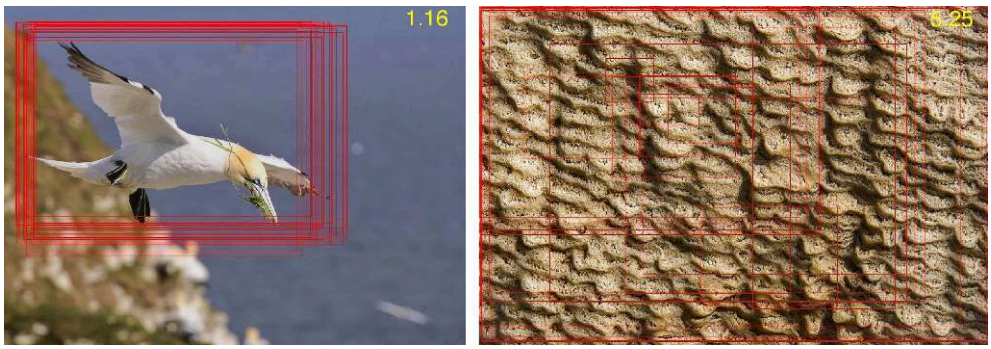
Le nom du fichier (« worker_05Waldave56jm9815.txt ») correspond à l'identifiant donné au·à la travailleur·euse·x par l'application web. Seules deux lignes du fichier sont présentées ici. Le premier élément de chaque ligne est une chaîne de caractères qui se termine par « .jpg » ; il correspond à l'identifiant (ID) de l'image traitée. Le deuxième élément de chaque ligne correspond à la note donnée à la tâche par lae travailleur·euse·x. Les éléments qui suivent correspondent aux coordonnées du ou des rectangles dessinés par lae travailleur·euse·x. Chaque rectangle est défini par quatre valeurs numériques faisant partie de l'espace de coordonnées de l'image traitée. La première valeur de chaque rectangle (« startX : *n*px ») correspond à la coordonnée horizontale de l'image. La deuxième valeur (« startY : *n*px ») correspond à la coordonnée verticale de l'image. La troisième valeur (« width : *n*px ») correspond à la largeur, en nombre de pixels, du rectangle dessiné. La quatrième valeur (« height : *n*px ») correspond à la hauteur, en nombre de pixels, du rectangle dessiné. Prises ensemble, ces quatre valeurs permettent de reconstruire le(s) rectangle(s) dessiné(s) par lae travailleur·euse·x.

15 Bien qu'explicitement conçu pour m'aider à documenter des situations de programmation nécessaires au façonnage de nouveaux algorithmes au sein du Lab, ce petit projet avait également un objectif en termes de traitement d'image. Cet objectif, secondaire, était de formuler des correspondances entre le contenu des photographies du corpus – en termes d'arrangement des valeurs pixelliques – et les rectangles et notes fournis par les travailleur·euse·x·s. En bref, l'hypothèse était que les photographies difficiles à étiqueter (notes élevées) et dont les rectangles étaient dispersés, n'étaient pas divisibles en parties plus petites. Inversement, dans le cas de photographies simples à étiqueter (notes basses) et dont les rectangles se recoupent, subdiviser leur contenu en sous-parties était opportun (voir figure 2.2). Être en mesure de trier automatiquement les images photographiques dont le contenu pouvait ou non être divisé en parties plus petites pouvait alors aider à l'élaboration de schémas de compression avec perte, eux-mêmes basés sur des procédures de segmentation. En ce sens, la méthode de calcul que j'essayais de définir pouvait éventuellement servir d'étape de prétraitement pour d'autres algorithmes de segmentation/compression plus complexes que les membres du Lab étaient alors en train de développer. En tous les cas,

avant de proposer une telle méthode, de nombreux programmes intermédiaires – y compris PARSE – se devaient d'être assemblés.

16 La conception de l'application web qui a rendu possible l'épreuve de *crowdsourcing* et la collecte de données a nécessité la réalisation de nombreux programmes. Il a d'abord fallu concevoir un programme Python de collecte automatisée (*web scraping*) afin de pouvoir parcourir et télécharger des images hétérogènes, en haute définition et sous licence *Creative-Commons*, mises à disposition par l'interface de programmation du site internet Flickr. Ensuite, plusieurs programmes utilisant les langages de programmation JavaScript et PHP ont dû être assemblés afin de permettre aux travailleur·euse·x·s du clic d'interagir avec un nombre spécifique d'images et de stocker leurs identifiants, étiquettes et notes dans des fichiers .txt. Des programmes Matlab ont ensuite été nécessaires pour lire le contenu textuel et numérique de tous les fichiers .txt et les réorganiser dans l'environnement logiciel Matlab. En raison de son agilité à concevoir des problèmes d'algèbre linéaire, Matlab est largement utilisé à des fins de recherche et d'industrie en informatique, en ingénierie et en économie. Connu pour être bien adapté au calcul matriciel, Matlab est notoirement inadapté à la réorganisation de données .txt en matrices et tableaux, processus généralement appelé *parsing* par les membres du Lab. Comme pour la plupart des autres programmes essentiels à la réalisation de ce projet, une séance d'aide a été nécessaire pour m'aider à assembler le programme de réorganisation permettant d'obtenir des vues telles que celles présentées dans la figure 2.2. C'est la formation de ce programme de réorganisation – nommé ici PARSE – que nous allons suivre dans la suite de cet article.

Figure 2.2. Deux vues sur les données collectées lors de l'épreuve de *crowdsourcing*.



Les deux vues ont été générées par un programme Matlab qui a analysé les données des fichiers .txt et les a associées aux images .jpg correspondantes. Sur la gauche, les travailleur·euse·x·s ont étiqueté approximativement la même partie de l'image et ont donné une note faible à cette tâche d'étiquetage (moyenne de 1,16). On pourrait alors supposer qu'il ferait sens de subdiviser le contenu de cette photographie en de plus petites parties (dans ce cas, l'oiseau et le reste). À droite, la situation inverse : les travailleur·euse·x·s ont étiqueté l'image de façon presque aléatoire et ont donné une note élevée à cette tâche d'étiquetage (5,25 en moyenne). On pourrait ainsi supposer – moyennant une série de postulats – que subdiviser le contenu de cette image en de plus petites parties ne ferait pas forcément sens. Détenir ce type d'idée générale – et la formaliser informatiquement – sur le contenu pixellique de certaines photographies pouvait servir d'appui initial pour le développement d'algorithmes de compression avec perte, eux-mêmes basés sur des modèles de segmentation automatique.

17 Les spécifications de PARSE peuvent être résumées ainsi : pour des raisons que je ne pourrai pas détailler ici, PARSE devait être capable de lire chaque ligne de chaque fichier .txt de façon à inscrire son contenu dans une base de données Matlab dont les différents tableaux, sous-tableaux et cellules devaient être définis par certaines

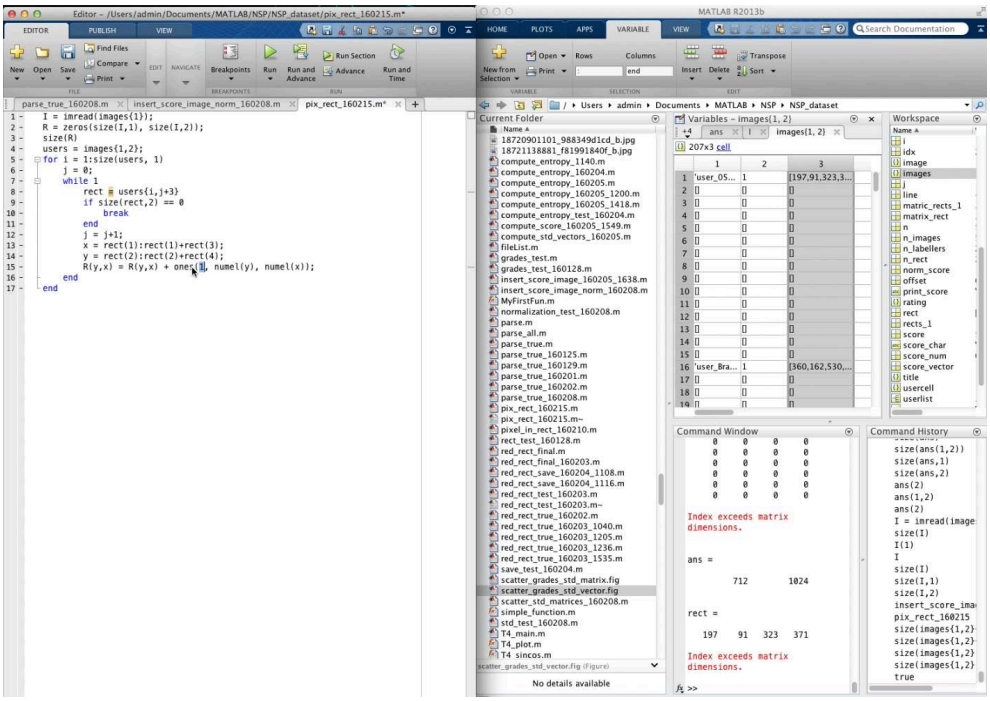
propriétés de la structuration des données .txt, comme par exemple leur nom ou la présence de tabs et de retour à la ligne. Plus précisément, ce que je nomme ailleurs le *scénario* de PARSE (Jaton, 2021a : pp. 181-195) – lui-même issu d'une histoire assignable – était : pour chaque fichier .txt, PARSE se doit d'extraire chaque ligne et classer leur contenu afin de faire correspondre chaque photographie utilisée lors de l'épreuve de *crowdsourcing* à un groupe de travailleur·euse·x·s, un groupe de notes, et un groupe de rectangles (eux-mêmes définis par des ensembles de quatre valeurs *x*, *y*, *width*, *height*). Le but de PARSE était ainsi d'organiser les données récoltées lors de l'épreuve de *crowdsourcing* afin de faciliter leur visualisation et analyse. La figure 2.2 ci-dessus est ainsi basée sur la réorganisation des données telle que produite par PARSE.

18 Je reviendrai dans la prochaine section sur certains objectifs et spécifications de PARSE. Le plus important à ce stade est de comprendre que PARSE a été conçu dans l'environnement logiciel et de programmation Matlab. Comme d'autres langages de programmation populaires dits « de haut niveau »[11] tels que Python ou C++, Matlab est généralement utilisé conjointement avec un environnement de développement intégré (EDI) qui comprend des fonctionnalités de visualisation et d'organisation de fichiers (voir figure 2.3). Mais contrairement à Python ou C++, le langage et l'EDI de Matlab sont détenus, maintenus et distribués sous licence propriétaire par la société *The MathWorks, Inc*. Au moment de mon enquête, le fait que Matlab soit propriétaire était critiqué par un nombre croissant de membres du Lab qui avaient tendance à préférer C++ (pour ses performances) ou Python (pour son statut open source et son active communauté de développeur·euse·s). Cependant, notamment en raison de sa structure interne conçue intrinsèquement pour le traitement des matrices, Matlab était, et est toujours, fréquemment utilisé dans le domaine du traitement de l'image.

2.3. Une proposition de formalisme

19 Pour des raisons de lisibilité, mon suivi de la formation pratique de PARSE ne portera que sur l'Éditeur et la Commande de l'EDI de Matlab. Dans les sections suivantes, le contenu de la figure 2.3 sera ainsi présenté comme dans le script 2.1.

Figure 2.3. Capture d'écran de l'environnement de développement intégré (EDI) de Matlab.



La fenêtre en bas à droite est l'Historique de Commande (*Command History*) qui permet de visualiser les dernières opérations effectuées. La fenêtre en haut à droite est l'Espace de Travail (*Workspace*) qui rassemble toutes les variables que lae programmeur·euse·x crée pendant sa session. À gauche de l'espace de travail, la fenêtre des Variables permet à lae programmeur·euse·x de visualiser les variables créées à l'intérieur de tableurs. Dans cette capture d'écran, la variable "images" est en cours de visualisation. En dessous se trouve la Fenêtre de Commande (*Command Window*) qui montre les résultats des opérations effectuées. Dans cette capture d'écran, la fenêtre affiche, entre autres, le message d'erreur « Index exceeds matrix dimensions ». La longue fenêtre au milieu de la capture d'écran est la fenêtre des Dossiers Courants (*Current Folder*) qui montre le contenu du dossier auquel Matlab est en train d'accéder. Sur la gauche, l'Éditeur (*Editor*) est la fenêtre qui permet à lae programmeur·euse·x d'écrire des programmes Matlab – également appelés scripts – c'est-à-dire des listes numérotées d'instructions écrites dans le langage de programmation Matlab. Lorsque lae programmeur·euse·x clique sur l'icône Exécuter (*Run*, en haut au milieu de l'Éditeur) ou utilise un raccourci clavier équivalent, les résultats du script sont imprimés dans la Commande. Dans cette capture d'écran, l'exécution du script fait, entre autres, apparaître l'inscription rouge "Index exceeds matrix dimensions" dans la Commande. La disposition spatiale de ces différentes fenêtres peut être modifiée selon les préférences de lae programmeur·euse·x.

Script 2.1. EDI de Matlab simplifié, tel qu'il sera présenté pour le reste de l'analyse.

```
1.  I = imread(images{1});
2.  R = zeros(size(I,1), size(I,2));
3.  size(R)
4.  users = images {1,2};
5.  for i = 1:size(users), 1)
6.     j = 0;
7.     while 1
8.         rect = users{i,j+3};
9.         if size(rect,2) == 0
10.             break
11.        end
12.        j = j+1;
13.        x = rect(1):rect(1)+rect(3);
14.        y = rect(2):rect(2)+rect(4);
15.        R(y,x) = R(y,x) + ones(numel(y),
           numel(x));
16.    end
17. end
```

```
ans =
  1024   712

rect=
  197  91  323  371


Index exceeds matrix
dimensions
```

Afin de rendre plus lisible le suivi des séquences de programmation, seuls les contenus de l'Éditeur (cellule de gauche) et de la Commande (cellule de droite) seront affichés. Ici, le script simplifié exprime une partie du contenu de la figure 2.3.

20 Même si PARSE était l'un des plus petits programmes du projet, je ne pourrai pas, et de loin, suivre l'ensemble de son processus de formation. Au lieu de rendre compte de l'ensemble de l'épisode de programmation qui a fait advenir PARSE, je me concentrerai uniquement sur une séquence spécifique qui me semble particulièrement instructive. Mon suivi de la séquence de programmation sera chronologique : il commencera au temps 0 (T0) et se terminera au temps 5 (T5). Pour autant, l'échantillonnage de chaque T ne suivra pas une période fixe, mais plutôt les modifications apportées à l'Éditeur et à la Commande. Supposons par exemple que le script 2.1 soit la première expression de PARSE au sein de la séquence de programmation que nous suivons (T0). Dès que la programmeuse apportera des modifications dans l'Éditeur et dans la Commande, ces modifications seront documentées et mises en évidence comme dans le script 2.2.

Script 2.2. Éditeur et Commande au T1, lorsque modifiés par la programmeuse.

```
1.  I = imread(images{1});
2.  R = zeros(size(I,1), size(I,2));
3.  size(R)
4.  users = images {1,2};
5.  for i = 1:size(users), 1)
6.     j = 0;
7.     while 1
8.         1
9.         rect = users{i,j+3};
10.        2
11.        if size(rect,2) == 0
12.            break
13.        end
14.        j = j+1;
15.        x = rect(1):rect(1)+rect(3);
16.        y = rect(2):rect(2)+rect(4);
17.        R(y,x) = R(y,x) + ones(numel(y),
           numel(x));
18.    end
19. end
```

```
ans =
  1024   712

ans =
  1

rect =
  197  91  323  371

ans =
  2

ans =
  1

Index exceeds matrix
dimensions
```

Dans le titre de la légende, le terme "T1" indique qu'il s'agit de la première modification du script au sein de séquence de programmation en train d'être suivie. Les instructions qui ont été ajoutées dans l'Éditeur sont surlignées en gris. Le contenu de la Commande est mis à jour.

21 Entre les différents T, les paroles et les actions de la programmeuse (dans cet exemple DF mais plus tard dans le texte, GY) et de l'ethnographe (FJ) seront retranscrites. Pour que les choses restent lisibles, j'omettrai certaines petites actions, comme les fautes de frappe ou les rapides hésitations. À la suite de T1 (script 2.2), la séquence de programmation se déroulerait, par exemple, comme suit:

> DF: OK. C'est là [DF pointe son doigt vers la ligne 9 du script 3.2 – T1]. Tu vois ? [DF place son curseur sur la ligne 9]. Il me donne « `1` », ensuite « `rect` », ensuite « `2` », ensuite « `1` » et s'arrête. C'est ce « `j+3` » qui devient trop grand après le premier rectangle. Il prend le premier rectangle et si le deuxième rectangle est plus grand, il n'arrive pas à incrémenter.
> FJ : Ah oui, ça bloque au deuxième.
> [DF supprime les lignes 3, 8 et 10 ; il inscrit « `;` » à la fin de la ligne 9]
> [DF exécute le programme]
> [script 2.3. – T2]

Script 2.3. Éditeur et Commande au T2.

```
1.  I = imread(images{1});
2.  R = zeros(size(I,1), size(I,2));

3.  users = images {1,2};
4.  for i = 1:size(users), 1)
5.     j = 0;
6.     while 1

7.        rect = users{i,j+3}

8.        if size(rect,2) == 0
9.           break
10.       end
11.       j = j+1;
12.       x = rect(1):rect(1)+rect(3);
13.       y = rect(2):rect(2)+rect(4);
14.       R(y,x) = R(y,x) + ones(numel(y),
          numel(x));
15.    end
16. end
```

```
Index exceeds matrix
dimensions
```

```
~~3.  size(R)~~
~~8.  1~~
~~10.  2~~
```

Dans le titre de la légende, le terme T2 indique qu'il s'agit du deuxième changement de la séquence de programmation. Les emplacements des modifications de l'Éditeur (ici, des suppressions de symboles) sont surlignés en gris. Le contenu de la Commande est mis à jour. De plus, les instructions qui ont été supprimées sont indiquées en texte barré dans la cellule du bas. Les numéros de ligne des instructions supprimées sont ceux de T*n*-1 (ici T1).

> DF : Ok. Maintenant on doit juste changer deux-trois trucs.

22 Ici et là, j'interviendrai également pour clarifier les choses et analyser ce qui se passe. Avant de commencer à suivre notre séquence de codage au prisme d'une sociologie de l'activité de programmation, il est important de garder à l'esprit qu'il n'est pas nécessaire de comprendre tout ce qui est dit dans les transcriptions ni tous les éléments de chaque T. Ce qui est important dans cette analyse approfondie des pratiques de programmation ce sont les *différences relatives* entre chaque T. C'est en se concentrant sur ces petites différences et leur progressive accumulation que nous parviendrons à saisir, peut-être, certains enjeux de ces cours d'action dont les produits ne cessent d'irriguer nos sociétés informatisées.

23 Je dois mentionner une dernière chose avant de plonger au cœur des pratiques effectives de programmation. On pourra objecter que cette étude de cas, et les propositions qui en découlent, ne sont pas représentatives des pratiques de programmation *en général*. À cela, je réponds que la représentativité n'est pas en jeu ici. La représentativité est un concept statistique puissant et valide quand les limites d'une population sont clairement définies. Les habitants d'une ville, les cellules d'un tissu, les mots d'un livre : cette classe d'éléments peut être reliée à un ensemble coûteux et équipé (les frontières administratives et géographiques d'une ville, les limites physiques d'un échantillon, la couverture d'un livre) qui définit, ensuite, un territoire et une population. Dans ces cas spécifiques, mais somme toute assez rares et souvent controversés, le concept de représentativité peut être utilisé pour extraire des résultats statistiquement significatifs.

24 Mais lorsqu'il n'y a pas de territoire, pas d'ensemble, la notion même de représentativité perd son sens. Qu'est-ce que la programmation ? Qui sont les programmeur·euse·x·s lorsqu'iels programment ? Quelle est la teneur de leurs expériences ? Nous n'en savons pratiquement rien. Et c'est peut-être là que l'ethnographie peut tendre à des considérations statistiques : l'exploration détaillées de territoires non définis – ou définis de façons problématiques – peut fournir des prises à la conception ultérieure de limites capables, elles, d'être explorées statistiquement. Et si je pense que la jeune artiste qui écrit un programme JavaScript pour animer les menus de son site web personnel, l'ingénieure de Boeing qui travaille sur la dernière mise à jour d'Ada pour les modules de pressurisation de cabine, ou l'informaticienne en traitement du signal qui réorganise ses fichiers .txt avec l'EDI de Matlab *diffèrent* à bien des égards – elles ont des problèmes, des affects, des environnements et des équipements différents –, je pense également que (presque) aucune de ces situations n'a encore été explorées micro-sociologiquement. Il faut bien commencer quelque part. Cette étude de cas en appelle ainsi d'autres, d'où l'aspect tout à fait exploratoire de ses propositions.

3. « Ici, il faut juste changer le '`line`' de '`rect`' »

25 Concentrons-nous sur PARSE. Sur la base de ce que j'ai présenté dans la section précédente, je vais maintenant documenter une courte séquence de programmation qui a duré moins de trois minutes en temps réel. Je resterai aussi proche que possible des matériaux formatés (mais empiriques) en recourant au formalisme introduit ci-dessus ainsi qu'à quelques concepts développés au sein des études sociales sur les sciences et les techniques. Mon but sera de suggérer qu'un ensemble de pratiques terriblement importantes pour les programmeur·euse·x·s a trait à la prolifération et l'alignement d'inscriptions en vue de paver un accès vers une entité éloignée et, consécutivement, indexer un emplacement au sein d'une liste numérotée. Mon espoir est que cette proposition deviendra plus claire au fur et à mesure que la section avancera.

3.1. Initier une enquête

26 Commençons *in medias res*, avec les scripts 3.1 et 3.2 :

[script 3.1 – T0]

Script 3.1. Éditeur et Commande au T0

```
1.  f = fopen('user_05Waldave56jm9815.txt');
2.
3.  images = cell(1);
4.  images{1} = cell(1);
5.  line = fgetl(f)
6.
7.  while ischar(line)
8.    elements = strsplit(line);
9.    rating = elements(2);
10.   images{1}{1,2} = sscanf(rating{1}, '%1');
11.   rect = line(4:2:10);
12.   coords = [];
13.
14.   for coord = rect
15.       rect = [coords sscanf(coord{1}, '%ipx')];
16.   end
17.
18.   images{1}{1,3} = coords;
19.   line = fgetl(f);
20.
21. end
```

```
>>
```

GY : On va déjà voir ce que ça donne.
[GY exécute le script]
[script 3.2 – T1]

Script 3.2. Éditeur et Commande au T1.

```
1.  f = fopen('user_05Waldave56jm9815.txt');
2.
3.  images = cell(1);
4.  images{1} = cell(1);
5.  line = fgetl(f)
6.
7.  while ischar(line)
8.    elements = strsplit(line);
9.    rating = elements(2);
10.   images{1}{1,2} = sscanf(rating{1}, '%1');
11.   rect = line(4:2:10);
12.   coords = [];
13.
14.   for coord = rect
15.       rect = [coords sscanf(coord{1}, '%ipx')];
16.   end
17.
18.   images{1}{1,3} = coords;
19.   line = fgetl(f);
20.
21. end
```

```
Cell contents reference
from a non-cell array
object

Error in parse(line 15)
rect = [coords
sscanf(coord{1},
'%ipx')];
```

GY : Et il coince, forcément. Je vais juste vérifier la ligne.

27 Que se passe-t-il entre le T0 et le T1 ? Après que GY ait exécuté le script, une inscription rouge apparaît dans la Commande, indiquant que « `Cell contents reference from a non-cell array object Error in parse(line 15) rect = [coords sscanf(c{1}, '%ipx')];` »[12]. D'où vient ce texte ? Qui l'a écrit ? Pour mieux comprendre l'origine de cette notification, il est nécessaire de présenter un acteur important de cette séquence de programmation : l'Interpréteur (INT). Pour que les vingt et une lignes de code de l'Éditeur ordonnent effectivement au *hardware* de l'ordinateur d'opérer sur le contenu des fichiers .txt, de nombreuses étapes doivent être franchies. Pour le cas qui nous intéresse ici, seule la toute première étape est importante, celle qui consiste à traduire chaque ligne de code en autre chose – dans ce cas précis, une série de sous-programmes en code machine qui, à leur tour, génèreront

des impulsions électriques à l'origine des calculs effectifs sur les données. L'une des entités responsables de cette traduction complexe est INT. Chaque fois que GY exécute le script, INT est enrôlé pour traduire le contenu de l'Éditeur, octet par octet. Nous n'avons pas besoin de savoir exactement ce que fait INT pendant ses processus de traduction : même pour GY, le fonctionnement précis de INT reste obscur. Ici, il nous suffit d'apprécier quatre caractéristiques de INT :

1. INT a sa propre trajectoire qui n'est tout à fait comprise par quasiment personne : des équipes hautement spécialisées travaillant pour la société *The MathWorks, Inc.*, éditrice de Matlab, l'ont façonné et continuent de le maintenir encore aujourd'hui. En ce sens – du moins du point de vue de GY lorsqu'elle est en train de programmer –, INT peut être considéré comme une entité *qui prend le risque d'exister* (James [1912] 2003 ; Latour 2013), tout comme un chat ou un éléphant de mer.
2. INT traduit une ligne de l'Éditeur après l'autre[13].
3. Dès que INT parvient à traduire une ligne de code de l'Éditeur, si cette ligne ordonne l'impression d'une inscription, INT imprime cette inscription dans la Commande.
4. Dès que INT ne parvient pas à traduire une ligne, il imprime une inscription rouge dans la Commande et, souvent, s'arrête. Parfois, cette inscription indique le numéro d'une ligne de l'Éditeur communément appelée le « point de rupture » (*breakpoint*), endroit et moment où INT s'est immobilisé.

28 GY est bien consciente – comme nous le sommes maintenant – que toute inscription rouge dans la Commande indique que INT n'a pas pu traduire la totalité des lignes du script. De plus, il suffit à GY de rapidement parcourir l'inscription pour voir qu'elle indique la ligne 15 du script, moment et endroit où INT a cessé ses opérations de traductions. Il s'agit donc pour GY de « vérifier cette ligne ».

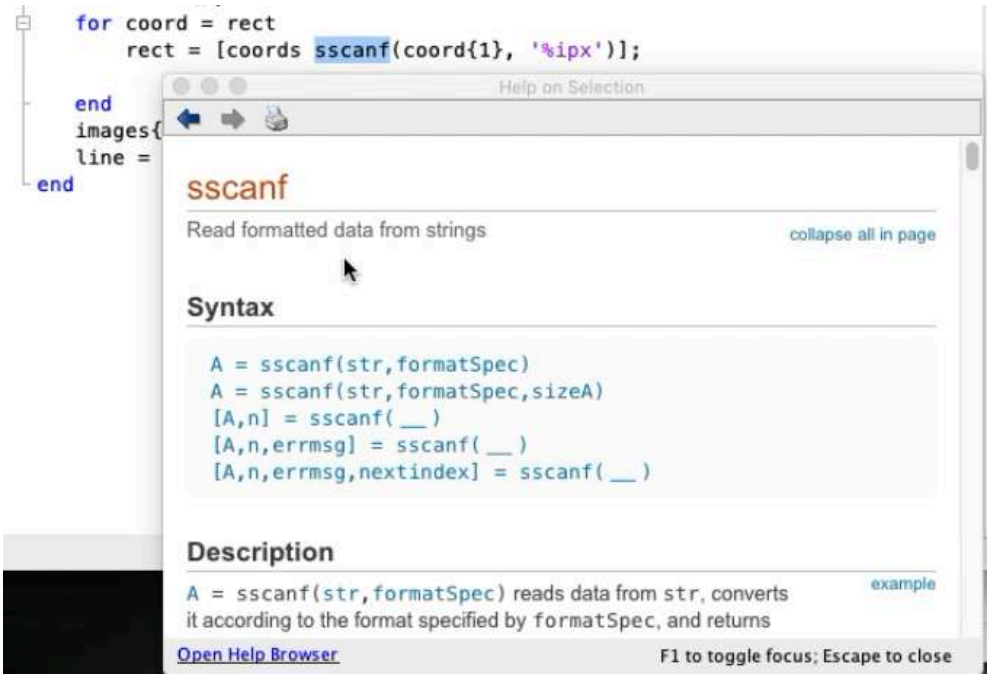
29 Déjà à ce stade, une observation s'impose. L'arrêt de INT dans sa traduction de PARSE est un phénomène inopiné : même s'il ne surprend pas GY – car cela coince « forcément » –, le surgissement de l'inscription rouge, expression de INT stoppé dans sa trajectoire, est un élément perturbateur qui force GY à s'engager dans une voie parallèle. À partir de T1 et de l'apparition de l'inscription rouge dans la Commande, il ne s'agit plus de façonner PARSE, mais bien de se saisir du phénomène qui empêche la poursuite de son façonnage. Le moment est donc charnière : si GY n'enquête pas sur ce qui affecte négativement INT, la formation de PARSE ne peut pas continuer.

3.2. Instaurer un quasi-laboratoire

30 Poursuivons :

> GY : C'est peut-être cette fonction-là.
> [GY surligne la fonction « `sscanf` » à la ligne 15]
> FJ : Ah « `sscanf` ».
> GY : Ouais. Je dois avouer que je me souviens plus très bien.
> [GY clique droit sur « `sscanf` » à la ligne 15 et sélectionne le menu « *Help on Selection* »]
> [figure 3.1]

Figure 3.1. Capture d'écran du menu « *Help on Selection* » tel qu'appelé par GY au T1.



31 Bien que ne parvenant pas à traduire l'entier du script, INT a été en mesure d'imprimer un point de rupture indiquant la ligne 15. Afin de mieux cerner le problème qui affecte la trajectoire de INT, GY commence donc par examiner cette ligne. Très vite, son attention se centre sur le seul élément capable de perturber PARSE à cet endroit-ci : la fonction « `sscanf` » – partie intégrante du langage Matlab – qui permet de lire des chaînes de données (par exemple des caractères ou des nombres entiers) et les convertir en un autre format (ici, le format « `i` » pour *integer*). Mais similairement aux autres fonctions intégrées de Matlab, « sscanf » ne peut être reconnue et traduite par INT que si elle est exprimée dans une syntaxe particulière. Et afin de vérifier l'exactitude de cette syntaxe, GY en appelle au menu « *Aide* » (*Help on Selection*) qui affiche toutes les spécificités de « `sscanf` ». Cette nouvelle inscription, qui vient s'additionner à l'inscription rouge au sein de la Commande, permet à GY d'effectuer une vérification visuelle de « `sscanf` » dans PARSE. En se référant à son curseur, elle devient en mesure d'aligner l'entrée certifiée du dictionnaire Matlab à propos de « `sscanf` » sur sa propre utilisation de la fonction à la ligne 15 :

GY : Non, c'est juste. C'est pas là.
[GY ferme la fenêtre « *Help on Selection* »]
FJ : Tu dis, ligne 15 ?
GY : Oui, c'est ce qu'il dit mais ça doit venir de plus haut.

32 L'alignement entre le menu d'aide et le script – lui-même dérivé de l'inscription rouge initiale émise par INT – permet à GY d'affirmer que ce qui affecte INT ne provient pas directement de la ligne 15, dont la syntaxe est correcte. Le phénomène problématique devient peut-être manifeste pour INT, en tant que symptôme, à cet endroit du script, mais la cause de ce phénomène réside ailleurs. Mais où ? Pour l'heure, il est difficile de le savoir. D'où la démarche de GY qui va consister, dès lors, à produire la connaissance de cet endroit :

[GY déplace le curseur vers la ligne 8]
GY : Je veux juste être sûre du fichier.
[GY supprime « `;` » à la fin de la ligne 8]

[GY exécute le script – script 3.3 – T2]

Script 3.3. Éditeur et Commande au T2.

```
1.  f = fopen('user_ 05Waldave56jm9815.txt');
2.
3.  images = cell(1);
4.  images{1} = cell(1);
5.  line = fgetl(f)
6.
7.  while ischar(line)
8.    elements = strsplit(line)
9.    rating = elements(2);
10.   images{1}{1,2} = sscanf(rating{1}, '%1');
11.   rect = line(4:2:10);
12.   coords = [];
13.
14.   for coord = rect
15.       rect = [coords sscanf(coord{1}, '%ipx')];
16.   end
17.
18.   images{1}{1,3} = coords;
19.   line = fgetl(f);
20.
21. end
```

```
elements =
1×10 cell array
Columns 1 through 5
{'16711005783_277…'}
{'3'} {'startX:'}
{'289px'} {'startY:'}
Columns 6 through 10
{'168px'}
{'width:'}
{'618px'}
{'height:'}
{'460px'}

Cell contents reference
from a non-cell array
object

Error in parse(line 15)
rect = [coords
sscanf(coord{1},
'%ipx')];
```

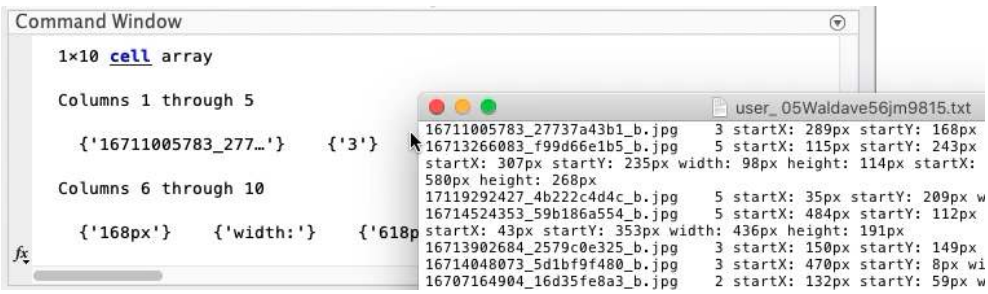
```
8. ;
```

GY : Ok. Maintenant.
[GY clique droit sur le fichier « user_ 05Waldave56jm9815.txt » du dossier courant ; sélectionne l'option « Open Outside Matlab »]

33 En supprimant le caractère « ; » à la fin de la ligne 8, GY ordonne à INT d'imprimer le résultat de la fonction « `strsplit` » [14]. S'ensuit l'apparition d'une série d'inscriptions qui ne sont pas rouges et qui peuvent donc être considérées comme des traductions véritables de cette ligne de code. Cela va de soi pour GY : des années de pratique de Matlab lui permettent d'être certaine que cette nouvelle inscription est une expression non problématique de INT. Mais cette nouvelle inscription exprime-t-elle la fonction « `strsplit` » du bon fichier .txt ? Si ce n'est pas le cas, c'est l'ensemble du script qui devra être reconsidéré. Pour vérifier que INT traite effectivement le bon fichier .txt, GY utilise une autre inscription, le fichier .txt lui-même, accessible depuis la fenêtre du dossier courant de l'EDI de Matlab :

[GY déplace le fichier .txt sur la droite de l'Éditeur, à hauteur de la Commande]
[figure 3.2]

Figure 3.2. Capture d'écran de la Commande de Matlab au T2 alignée au fichier « user_ 05Waldave56jm9815.txt ».



GY : Donc c'est bien ça. On est juste. C'est clairement après.
FJ : Mais avant la 15 ?
GY : Oui, je pense qu'il itère pas sur le bon truc.

34 Le rapprochement de deux nouvelles inscriptions – celle émanant de la ligne 8 et celle affichée par le fichier .txt – permet à GY d'affirmer que le problème de INT se trouve *après* la ligne 8. Une fois jointe à l'affirmation précédente, qui établissait le point de rupture de INT à la ligne 15, il s'ensuit que ce qui affecte la trajectoire de INT au point de le rendre incapable de traduire PARSE se trouve entre les lignes 9 et 14. Cela ne suffit évidemment pas à rendre le script opérationnel, mais permet tout de même à GY de resserrer son enquête.

35 Ce qui se passe au T2 est extrêmement simple, mais n'en reste pas moins profond et constitue un bel exemple du processus que je souhaite mettre en évidence. En imprimant la première ligne du fichier saisi par PARSE et en la comparant avec la première ligne du fichier .txt, GY – bien aidée par les 28 pouces de son écran principal – devient en mesure d'opérer une relation – ici une relation d'*identité* – dont le résultat la rapproche de l'identification du phénomène qui cause l'arrêt de INT à la ligne 15. Cette pratique qui consiste à produire, saisir et aligner des inscriptions au sein d'un même espace graphique afin d'identifier l'origine d'un phénomène est, je crois, au cœur de l'activité de programmation. Comme je le dis ailleurs (Jaton, 2021a : 135-195), ce n'est clairement pas le seul registre d'actions qui se déploie lors des situations de codage. Mais dans certains cas spécifiques, lorsqu'une entité véloce telle que INT est bloquée dans sa trajectoire, empêchant ainsi le traitement des données au moyen d'impulsions électriques, la production, l'empilement et l'alignement d'inscriptions sont cruciaux.

36 À ce stade un peu plus avancé de l'analyse, une deuxième observation s'impose. Au T0, et précédemment, GY inscrivait une série de symboles au sein d'une liste numérotée capable elle-même, en en passant par quantité d'autres entités, de traiter des données numériques (en l'occurrence des fichiers .txt et .jpg). L'espace de travail de GY, vertical (ses écrans) comme horizontal (sa table de travail), opérait alors comme un établi soutenant l'aboutissement d'un artefact technique : un script capable de réorganiser le contenu de fichiers .txt au sein d'une base de données Matlab personnalisée. Mais à partir de T1, et d'autant plus à T2, l'espace de travail se métamorphose. Il cesse, temporairement, d'être un atelier et devient, temporairement aussi, un *laboratoire* où se déroulent des expérimentations, modestes, qui tendent vers un autre horizon. Il ne s'agit plus de façonner un artefact technique mais bien d'isoler *les origines d'un phénomène problématique*. L'enjeu n'est plus le même : il ne s'agit plus d'articuler des astuces (fonctions `for`, `sscanf`, ou `strsplit`) par des détours successifs mais bien de renseigner l'état d'une entité lointaine, en l'occurrence INT, dont les rapports constitutifs sont à la fois si petits (il tient tout entier dans un ordinateur portable), si longs (son code source fait des centaines de milliers de lignes) et si rapides (il opère à une vitesse proche de celle de la lumière) que seul le ralentissement opéré par une expérimentation est capable d'identifier et, pour ainsi dire, d'*épingler*. Pour le dire en un langage plus philosophique, le mode de véridiction (Latour, 2012) qui engage GY, l'ethnographe, l'espace de travail, l'ordinateur et le script change : il passe subrepticement (car précipitamment) d'un registre de façonnage technique à un registre de connaissance certifiée.

3.3. Indexer un emplacement

37 Équipés de ces propositions, continuons notre suivi de PARSE :

[GY ferme le fichier .txt]

[GY ajoute « ; » à la fin de la ligne 8 ; déplace le curseur à la ligne 15]
GY : Donc là.
[GY remonte le curseur à la ligne 12]
GY : Ah mais c'est peut-être le type de « coords ».
FJ : Le type matrice ?
GY : Ouais mais c'est justement pas un cell.
[à la ligne 12, GY remplace « [] » par « {} »]

38 Après avoir défini la limite supérieure de la zone problématique de PARSE (la ligne 8), GY inspecte les lignes au-dessus de la ligne 15. Très vite, elle est prise d'un doute : le type de la variable « coords » (à l'intérieur de laquelle PARSE doit ensuite itérer les éléments de « rect »), tel que défini à la ligne 12, est peut-être erroné. En l'état, il s'agit d'une matrice – c'est-à-dire d'un tableau organisant des entiers – alors qu'il faudrait peut-être que ce soit une cellule (« cell ») – c'est-à-dire un tableau pouvant également contenir des données textuelles. Cela ferait d'ailleurs écho aux termes « cell » et « non-cell array » mentionnés par INT en précision de son point de rupture (cf. scripts 3.2 et 3.3). Mais est-ce vraiment là que se situe le problème de INT ? Afin de vérifier ce doute, GY remplace « [] » (qui définit une matrice) par « {} » (qui définit une cellule) et exécute le script :

[GY exécute le script]
[script 3.4 – T3]

Script 3.4. Éditeur et Commande au T3.

```
1.  f = fopen('user_05Waldave56jm9815.txt');
2.
3.  images = cell(1);
4.  images{1} = cell(1);
5.  line = fgetl(f)
6.
7.  while ischar(line)
8.    elements = strsplit(line);
9.    rating = elements(2);
10.   images{1}{1,2} = sscanf(rating{1}, '%1');
11.   rect = line(4:2:10);
12.   coords = {};
13.
14.   for coord = rect
15.       rect = [coords sscanf(coord{1}, '%ipx')];
16.   end
17.
18.   images{1}{1,3} = coords;
19.   line = fgetl(f);
20.
21. end
```

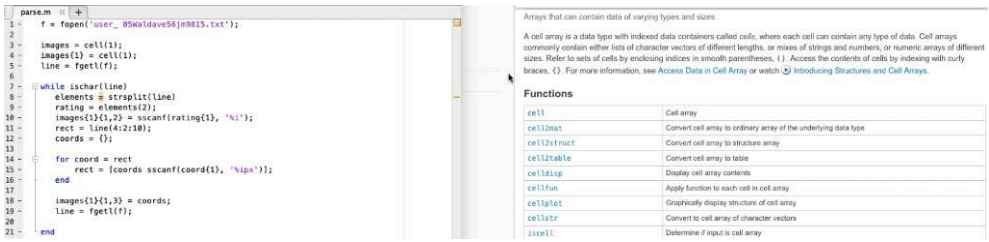
```
Cell contents reference
from a non-cell array
object

Error in parse(line 15)
rect = [coords
sscanf(coord{1},
'%ipx')];
```

```
12.   {}
```

GY : Ah non.
FJ : Ça change rien.
GY : Arg, je me rappelle plus bien comment ça marche avec *Matlab*. Faut juste que je checke.
[GY ouvre Firefox ; cherche « *Matlab cell array* » sur Google ; ouvre le premier lien affiché ; rapproche le navigateur de l'Éditeur]
[figure 3.3]

Figure 3.3. Capture d'écran de l'Éditeur au T3 aligné au navigateur web.



[GY inspecte la documentation Matlab sur « *Cell Arrays* »]
GY : Non non, c'était juste. C'est ailleurs. Là on veut itérer dans une matrice

39 L'expérience de GY issue de son doute quant au type de la variable « `coords` » n'est pas concluante. Si la ligne 12 paraissait suspecte, ça n'était finalement dû qu'à son rapport aujourd'hui distant avec Matlab car elle lui préfère désormais Python pour le prototypage rapide et C++ pour l'implémentation logicielle. À cet endroit-là du script, il est important que « `coords` » soit une matrice, car ce tableau sera ensuite rempli de nombres entiers, grâce précisément à la fonction « `for` » qui suit immédiatement. Mais cette certitude quant à l'adéquation de la ligne 12 n'était soutenue, dans un premier temps, par aucune inscription : au T3, après avoir exécuté le script, aucuneaucune nouvelle inscription n'est générée, et les souvenirs épars de GY ne lui permettent pas d'affirmer quoi que ce soit quant à l'incertitude entre tableau et matrice. GY se tourne alors vers un autre dispositif, en l'occurrence un navigateur web donnant accès à un moteur de recherche bien connu, afin de produire une nouvelle série d'inscriptions alignées, à leur tour (en exploitant la largeur de l'écran principal de DF), à la ligne 12 de PARSE. De cette confrontation, il résulte que non, définitivement, la ligne 12 est correcte.

40 A priori peu concluante, cette épreuve confirmatrice permet tout de même à GY d'acquérir une autre certitude quant au phénomène qui affecte négativement INT. En vertu de cette rapide expérimentation, elle-même issue des autres rapides expérimentations déployées au T1 et au T2, GY sait maintenant que le problème se situe *ailleurs* (c'est-à-dire au-dessus). En vertu de l'alignement des *inscriptions provisoires* (Denis, 2017 : 34) produites depuis le T1, la cause du phénomène problématique se trouve dans l'espace délimité par les lignes 9 et 11.

41 Poursuivons :

[GY remplace « `{}` » par « `[]` » à la ligne 12]
GY : Je vais juste voir.
[GY supprime « `;` » à la ligne 11]
[GY exécute le script]
[script 3.5 – T4]

Script 3.5. Éditeur et Commande au T4.

```
1.  f = fopen('user_05Waldave56jm9815.txt');
2.
3.  images = cell(1);
4.  images{1} = cell(1);
5.  line = fgetl(f)
6.
7.  while ischar(line)
8.    elements = strsplit(line);
9.    rating = elements(2);
10.   images{1}{1,2} = sscanf(rating{1}, '%1');
11.   rect = line(4:2:10)
12.   coords = [];
13.
14.   for coord = rect
15.       rect = [coords sscanf(coord{1}, '%ipx')];
16.   end
17.
18.   images{1}{1,3} = coords;
19.   line = fgetl(f);
20.
21. end
```

```
Rect =
1058

Cell contents reference
from a non-cell array
object

Error in parse(line 15)
rect = [coords
sscanf(coord{1},
'%ipx')];
```

```
11.   ;
```

GY : Quoi ?
FJ : Ouais, pourquoi quatre [1058] ?
GY : C'est bizarre.

42 En poursuivant sa revue des lignes potentiellement problématiques, dont les limites ont été délimitées dès le T2, GY en vient à vérifier la ligne 11, objet central de l'itération régulée par la fonction « `for` » trois lignes plus loin. L'inscription qui en résulte, « `1058` », apparaît immédiatement problématique. Rappelons que le but de PARSE est de créer une base de données constituée, entre autres, des éléments inscrits dans les fichiers .txt. Parmi ces éléments, les coordonnées des rectangles inscrits par les travailleur·euse·x·s du clic revêtent une importance cruciale, car ce sont eux qui, pour beaucoup, attestent ou non de la pertinence de subdiviser le contenu des images (en fonction, bien sûr, des nombreux postulats du projet). Or il est établi depuis le début de la situation de programmation que ces rectangles sont définis par des ensembles de quatre valeurs : 1) Un point de départ sur l'axe x de l'image ; 2) Un point de départ sur l'axe y de l'image ; 3) Une largeur (en nombre de pixels) ; 4) Une longueur (en nombre de pixels). Il est ainsi curieux, même pour l'ethnographe, que la valeur de « `rect` » telle qu'exprimée par INT soit « `1058` », qui renverrait à un rectangle absurde commençant au point (1;0) de l'image et d'une surface de 5 sur 8 pixels. À ce stade, il n'est pas évident de savoir d'où dérive l'inscription « `1058` ». Mais compte tenu des spécifications de PARSE, cette inscription est « bizarre » et mérite d'être étudiée davantage :

GY : Je vais juste vérifier « `line` ».
[GY crée une nouvelle ligne 11 ; inscrit « `line` »]
[GY exécute le script]
[script 3.6 – T5]

Script 3.6. Éditeur et Commande au T5.

```
1.  f = fopen('user_05Waldave56jm9815.txt');
2.
3.  images = cell(1);
4.  images{1} = cell(1);
5.  line = fgetl(f)
6.
7.  while ischar(line)
8.    elements = strsplit(line);
9.    rating = elements(2);
10.   images{1}{1,2} = sscanf(rating{1}, '%1');
11.   line
12.   rect = line(4:2:10)
13.   coords = [];
14.
15.   for coord = rect
16.       rect = [coords sscanf(coord{1}, '%ipx')];
17.   end
18.
19.   images{1}{1,3} = coords;
20.   line = fgetl(f);
21.
22. end
```

```
Line =
16711005783_27737a43b1
_b.jpg  3  startX: 289
px startY: 168px width:
618px height: 460px

Rect =
1058

Cell contents reference
from a non-cell array
object

Error in parse(line 16)
rect = [coords
sscanf(coord{1},
'%ipx')];
```

[GY inspecte la Commande]
GY : Ah d'accord. C'est pas « `line` » qu'il doit prendre.
FJ : Mais comment il arrive à « `1058` » ?
GY : En fait il prend « 4 », chaque « 2 », jusqu'à « `10` », de cette chaîne. Et c'est pour ça qu'il peut pas itérer après. Ici, il faut juste changer le « `line` » de « `rect` ».

43 L'expérience est concluante. Grâce à la nouvelle inscription, le phénomène problématique qui affecte la trajectoire de INT est identifié : à la ligne 12 (précédemment 11), « `rect` » est incorrectement défini et c'est la raison pour laquelle INT « ne peut pas itérer après ». Mais comment GY en arrive-t-elle à cette conclusion qui pointe vers une erreur de définition de « `rect` » ? Si l'on observe le contenu de la Commande au T5, comme le fait GY, nous pouvons voir que INT commence par imprimer le contenu de la variable « `line` », comme GY le lui demande à la ligne 11. Ensuite, INT imprime le contenu de « `rect` » tel que défini à la ligne 12. Et c'est par ce biais que l'inscription « `1058` » revêt du sens, car une fois alignée avec le résultat de la ligne 11 au sein de la commande, il apparaît – aux yeux entraînés de GY – que INT extrait le quatrième caractère de « `line` » (« `1` »), une fois sur deux (« `05` »), jusqu'au dixième élément (« `8` »). Si cette fonction était effectivement adaptée pour extraire les éléments de « `rect` », elle aboutit à la définition d'une variable problématique quand elle est apposée à « `line` ». Le résultat de cette erreur, qu'il s'agira ensuite de corriger, est que INT ne peut pas faire d'itération en ligne 16 (précédemment ligne 15) puisque la base de cette itération est une chaîne de quatre caractères « `1058` ». INT imprime donc un message d'erreur en rouge dont la cause véritable – nous le savons maintenant – se trouve en fait à la ligne 12 (et non pas à la ligne 16 comme indiqué par le point de rupture).

44 À ce stade, il est important de rappeler que cette dernière inscription – bien que cruciale – n'a pas permis à elle seule la constitution d'un lien entre l'inscription rouge de INT et la ligne 12. C'est l'empilement et l'alignement des inscriptions précédentes qui ont progressivement conduit à la formation de cette dernière inscription. L'ensemble du processus d'alignement a permis à GY d'identifier la provenance du phénomène qui affecte négativement INT : il ne peut pas itérer, car la variable « `rect` » est inadéquatement définie à la ligne 12. C'est bien la chaîne d'inscriptions

ainsi constituées – en moins de trois minutes – par GY qui l'a rendue capable de produire cette connaissance sur l'état de INT. Au sein de son quasi-laboratoire, qui est aux sciences institutionnalisées ce que le garage d'un·e bricoleur·se·x est aux usines Renault, elle est parvenue à former une connaissance certifiée. Grâce à ce travail scriptural, GY est devenue une « sujette connaissant » et INT un « objet connu ».

45 Cette réflexion quant à l'ancrage scriptural qui permet à GY de connaître précisément une partie, certes infime, de INT se raccroche à une autre, celle-ci développée dans la littérature en *Science and Technology Studies* (STS [15]). Au cours des quarante dernières années, de nombreuses études sur la construction des connaissances certifiées ont effet souligné la centralité des documents (Latour et Woolgar, 1986), diagrammes (Netz, 2003), graphiques (Dennis 1989 ; Gooday, 1990) et notes (Lynch 1985 ; Garfinkel 1981) que je rassemble ici, à la suite de Latour (2012), sous le terme générique « inscriptions ». D'autres études importantes ont également mis en exergue les instruments, expériences et standards nécessaires afin de produire, confronter, et articuler ces inscriptions (Hacking, 1983 ; Knorr-Cetina et Mulkay, 1983 ; Collins 1975 ; Dear, 1987 ; Gooding et al.,1989). D'autres études ont encore souligné l'importance de la manipulation et la diffusion de ces inscriptions (Latour, 1987 ; Knorr-Cetina, 1999) qui, à force d'être comparées, confrontées, et articulées – en bref *alignées* – finissent parfois par former ce que Latour (1999) appelle des « chaînes de référence » : des accès matériels plus ou moins solidifiés qui permettent de documenter, lorsque tout est en place, une partie du comportement d'une entité distante (une planète, un virus, une particule). Par-delà leurs désaccords, notamment concernant les notions de « nature » et de « société », ces études classiques présentent les connaissances certifiées comme construites et objectives. Grâce aux pratiques scientifiques et aux fragiles institutions qui soutiennent l'expression de ces pratiques, la construction des connaissances est objective.

46 Comme cette courte séquence semble l'indiquer, les pratiques de programmation peuvent parfois – pas toujours – se rattacher timidement à la construction de connaissances certifiées. En effet, la production d'inscriptions – en en passant par des expériences équipées –, leur comparaison et leur alignement en vue de produire d'autres d'inscriptions fait écho, bien que lâchement, à ce qui a pu être observé dans certains laboratoires de science expérimentale, ou même certaines administrations (Weller, 2012). Peu à peu, grâce aux manipulations, comparaisons et alignements d'inscriptions (« je vais juste vérifier » ; « c'est peut-être là » ; « je vais juste voir » ; « il faut que je checke »), une fragile voie d'accès est mise en place qui peut permettre la caractérisation d'un phénomène engageant une entité distante. Dans le cas de la programmation, cette entité distante peut varier : il peut s'agir, par exemple, d'un interpréteur Matlab, d'un compilateur C ou d'un microprocesseur Intel. Mais en tous les cas, la caractéristique commune à ces différentes entités est l'incroyable rapidité de leurs rapports constitutifs. Comment, en effet, avoir une emprise sur un interpréteur, un compilateur ou – pire – un processeur qui exécute des milliards d'opérations par seconde ? Une fois assemblées, ces entités sont pratiquement insaisissables, d'où l'adéquation du mode de véridiction scientifique pour mieux comprendre ce qui les affecte. Et c'est ce régime d'énonciation particulier, qui permet de produire une information certifiée sur l'état d'une entité éloignée, qui se laisse parfois furtivement apercevoir lorsque l'on s'engage dans une sociologie de l'activité de programmation.

4. Discussion et conclusion

47 L'invitation à se demander comment les programmes informatiques sont façonnés en situation ne constitue pas la voie privilégiée par les études sociales du code informatique. Peut-être est-ce dû à des difficultés méthodologiques ? Dans la perspective d'une sociologie de l'activité de programmation, le présent article entend poser quelques bases, fragiles mais réelles, à une telle exploration. À titre de conclusion, et au terme de ce plongeon au cœur du code, je voudrais insister sur trois résultats provisoires.

48 Premièrement, déplier le travail concret d'écriture des programmes et, ce faisant, ralentir son déploiement met à jour un moment particulier, parmi certainement beaucoup d'autres, de l'activité de programmation. Ce moment se caractérise par une déviation initiale : un phénomène problématique, souvent représenté par une inscription rouge, empêche le traitement effectif de données numériques et force les programmeur·euse·x·s à monter une série d'expérimentations – et donc à temporairement interrompre le façonnage de leur artefact technique – pour mieux identifier la source de cette interruption. Petits bouts de code, comparaisons furtives avec de la documentation officielle, consultation de forums web spécialisés : autant d'*inscriptions provisoires* qu'il s'agit progressivement d'aligner les unes aux autres au sein d'un même espace graphique. Si tout se passe bien, au bout d'un moment plus ou moins long, engageant plus ou moins d'acteurs, ce laboratoire improvisé parvient à assembler une chaîne de référence suffisamment longue et robuste pour renseigner l'état de l'entité problématique (interpréteur, compilateur, processeur). Et c'est précisément cette information certifiée qui va permettre d'indexer le phénomène problématique à un emplacement au sein d'une liste numérotée, indexation qui marque la fin, temporaire, de ces furtifs moments quasi-scientifiques.

49 Deuxièmement, comme c'est le cas des pratiques de laboratoire au sein des sciences expérimentales institutionnalisées, les pratiques de programmation – du moins lorsqu'elles visent à renseigner l'état d'une entité lointaine – tendent à écarter les instruments qui ont permis la caractérisation du phénomène étudié. Dans les deux cas, lorsque la source d'un phénomène a été identifiée grâce à un environnement expérimental spécifique et certifié, les pratiques, instruments et expériences qui ont permis la formation de la chaîne de référence sont généralement mis de côté (Latour et Woolgar, 1986 : 105-155). Cette tendance au sein des sciences expérimentales peut rendre son histoire difficile à mener ; lorsque les faits établis sont purifiés des échafauds qui leur ont précédemment permis d'être assemblés et solidifiés, grande peut être la tentation de partir des faits établis et d'extrapoler à rebours (Collins 1975). Pour saisir empiriquement la pratique des sciences, il est donc crucial de considérer les faits scientifiques comme des conséquences de processus spécifiques plutôt que comme des causes d'événements ultérieurs (Bloor, 1981). Dans une moindre mesure, il en va parfois de même pour la programmation informatique. Lorsque le phénomène engageant l'entité distante est caractérisé ; lorsque l'emplacement problématique au sein du script est identifié, la plupart des instruments (petits bouts de code, données traitées, documentation, forum web) sont mis de côté et rapidement oubliés. À la fin de l'épisode de programmation, lorsque le script est fonctionnel et exécute la tâche souhaitée, beaucoup de ces *objets intermédiaires* (Vinck, 2011) sont éclipsés. Par conséquent, si l'on se base uniquement sur des scripts ou des programmes terminés

afin d'étudier les pratiques de programmation, le risque est grand d'invisibiliser encore davantage ce qui a été précédemment nécessaire à la formation de ces scripts ou programmes[16].

50 Enfin, cette posture analytique suggère d'explorer plus à fond deux pistes de recherche. Dans le cas de la programmation informatique, on peut en effet imaginer différentes expressions de ces pratiques d'alignement. Même si je présume que ces expressions consistent, en partie, à former des chaînes de référence pour accéder à des entités distantes et pointer vers des lignes de listes numérotées, elles peuvent se déployer dans des temporalités et des espaces différents de ceux de GY. Si l'on considère par exemple le *program testing* – un travail industriel important qui consiste à détecter et à documenter les erreurs d'exécution afin de modifier les lignes de code correspondantes – ce travail peut être tout à fait distribué dans l'espace et le temps (Parrington et Roper, 1989 ; Myers et al., 2011). Les rapports d'erreurs (*error reports*) que nous rencontrons souvent lorsqu'un de nos logiciels plante pour de mystérieuses raisons sont également d'autres indications de cette nécessité d'alignement d'inscriptions, car ces rapports visent précisément à documenter à quel moment et après quelles opérations le programme a fatalement affecté l'interpréteur, le compilateur ou le processeur. Ces rapports servent ainsi de premières inscriptions qui, à leur tour, seront articulées à d'autres, puis à d'autres, jusqu'à indiquer finalement, parfois, l'origine du phénomène au sein du code source du programme. Bien que différents en termes d'extension et de main-d'œuvre, ces processus de *program testing* et de *error reporting* peuvent également être lus, peut-être, en fonction du mode de véridiction scientifique consistant, en partie, à aligner des inscriptions en vue de produire des chaînes de référence. S'ensuit une série de pistes de recherche pouvant porter sur ces entreprises collectives, pour l'heure assez peu documentées, visant à continuellement actualiser les connaissances sur les interpréteurs, compilateurs ou processeurs dans leurs rapports avec les nouveaux produits proposés par les fabricants de matériel informatique[17].

51 L'importance du régime d'action consistant à aligner des inscriptions afin d'identifier des emplacements au sein de listes numérotées peut également expliquer, au moins en partie, l'obsession des programmeur·euse·x·s professionnel·le·s pour l'intelligibilité des programmes. Ce sujet a été documenté par Button et Sharrock (1995) dans leur admirable mais solitaire étude. Comme ils l'ont montré, rendre un programme intelligible pour les autres programmeur·euse·x·s implique un travail de *mise en convention* des variables et des fonctions pour rendre la structure du programme lisible en tant que document organisé et référencé ; un travail de *formatage* des différentes fonctions et paramètres du code afin de le rendre consultable à partir de son organisation visuelle ; et un travail de *commentaires* au moyen de phrases explicatives que les symboles initiaux (« % » pour le cas de Matlab) permettent aux interpréteurs ou aux compilateurs de ne pas prendre en compte durant leurs opérations de traduction. Si la séquence de programmation que nous venons de suivre ne porte pas directement sur ces trois aspects (le formatage, la mise en page et le commentaire), elle précise néanmoins que ces pratiques tendent vers des moments futurs où leurs produits pourront opérer comme points de repère directement mobilisables dans la constitution de chaînes de référence. Ces marqueurs fonctionnent ainsi comme infrastructure référentielle capable d'accélérer le travail d'alignement en cas de future affection négative d'un interpréteur ou d'un compilateur, ce qui se produit sans cesse dans les entreprises logicielles responsables de la maintenance de programmes longs et complexes. S'ensuit dès lors une autre ligne de recherche (connexe) quant aux

modalités de maintenance du code informatique et des différents ressorts de sa mise en (in)visibilité.

52 La présente contribution, qui se veut surtout exploratoire, en appelle ainsi d'autres en vue de former, peut-être, une sociologie systématique de l'activité de programmation ; une démarche capable de seconder, à son niveau, les nombreuses entreprises déjà en cours dans le domaine des études sociales du code informatique.

BIBLIOGRAPHIE

Akrich Madeleine (1989). « La construction d'un système socio-technique. Esquisse pour une anthropologie des techniques », *Anthropologie et Sociétés*, 13 (2), pp. 31-54.

Alcaras Gabriel (2020). « Des biens industriels publics. Genèse de l'insertion des logiciels libres dans la Silicon Valley. », *Sociologie du travail*, 62 (3).

Anderson Tim (2020) « 'It's Really Hard to Find Maintainers...' Linus Torvalds Ponders the Future of Linux. » *The Register*, Juin 2020. Accessible à l'adresse : https://www.theregister.com/2020/06/30/hard_to_find_linux_maintainers_says_torvalds/

Bechmann Anja & Bowker Geoffrey C. (2019). « Unsupervised by any other name: Hidden layers of knowledge production in artificial intelligence on social media », *Big Data & Society*, 6 (1).

Bloor David (1981). « The Strengths of the Strong Programme », *Philosophy of the Social Sciences*, 11 (2), pp. 199-213.

Brooks Frederick (1975). *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass, Addison-Wesley Professional.

Bucher Taina (2018). *If...Then: Algorithmic Power and Politics*. New York, Oxford University Press.

Button Graham & Sharrock Wes (1995). « The mundane work of writing and reading computer programs », in Have Paul T. & Psathas George (dir.), *Situated order: Studies in the social organization of talk and embodied activities*. Washinton, DC, University Press of America, pp. 231-58.

Cardon Dominique (2015). A *quoi rêvent les algorithmes. Nos vies à l'heure des big data*. Paris, Le Seuil.

Casilli Antonio (2019). *En attendant les robots*. Paris, Le Seuil.

Collins Harry (1975). « The Seven Sexes: A Study in the Sociology of a Phenomenon, or the Replication of Experiments in Physics », *Sociology*, 9 (2), pp. 205.

Couture Stéphane (2019). « The Ambiguous Boundaries of Computer Source Code and Some of Its Political Consequences », in *digitalSTS*. Princeton N.J., Princeton University Press, pp. 136-156.

Crawford Kate & Calo Ryan (2016). « There is a blind spot in AI research », *Nature*, 538 (7625), pp. 311-313.

Dear Peter (1987). « Jesuit Mathematical Science and the Reconstitution of Experience in the Early Seventeenth Century », *Studies in History and Philosophy of Science Part A*, 18 (2), pp. 133–175.

Dear Peter & Jasanoff Sheila (2010). « Dismantling boundaries in science and technology studies », *Isis; an International Review Devoted to the History of Science and Its Cultural Influences*, 101 (4), pp. 759-774.

Demazière Didier, Horn François & Zune Marc (2007). « The Functioning of a Free Software Community: Entanglement of Three Regulation Modes – Control, Autonomous and Ditributed », *Science Studies*, 20 (2), pp. 34-54.

Denis Jérôme (2018). *Le travail invisible des données: Elements pour une sociologie des infrastructures scripturales*. Paris, Presses des Mines.

Dennis M.A. (1989). « Graphic understanding: instruments and interpretation in Robert Hooke's Micrographia », *Science in Context*, 3 (2), pp. 309-364.

Flor Nick V. & Hutchins Edwin L. (1991). « Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance », in Koenemann-Belliveau Jurgen, Moher Thomas, & Robertson Scott P. (dir.), *Empirical Studies of Programmers: Fourth Workshop*. Norwood, N.J, Ablex Publishing Corp., pp. 36-62.

Fuller Matthew (Ed.) (2008). *Software Studies – A Lexicon*. Cambridge, Mass, MIT Press.

Garfinkel Harold (1981). « The Work of a Discovering Science Constructed with Materials From the Optically Discovered Pulsar », *Philosophy of the Social Sciences*, 11 (2), pp. 131.

Gooday Graeme (1990). « Precision Measurement and the Genesis of Physics Teaching Laboratories in Victorian Britain », *The British Journal for the History of Science*, 23 (1), pp. 25-51.

Gooding David, Pinch Trevor & Schaffer Simon (1989). *The Uses of Experiment: Studies in the Natural Sciences*. Cambridge University Press.

Gray Mary L. & Suri Siddharth (2019). *Ghost Work: How to Stop Silicon Valley from Building a New Global Underclass*. Boston, Houghton Mifflin Harcourt.

Grosman Jérémy & Reigeluth Tyler (2019). « Perspectives on algorithmic normativities: engineers, objects, activities », *Big Data & Society*, 6 (2), pp. 2053951719858742.

Hacking Ian (1983). *Representing and Intervening: Introductory Topics in the Philosophy of Natural Science*. Cambridge Cambridgeshire ; New York, Cambridge University Press.

Henriksen Anne & Bechmann Anja (2020). « Building truths in AI: Making predictive algorithms doable in healthcare », *Information, Communication & Society*, 23 (6), pp. 802-816.

Jaton Florian (2017). « We get the algorithms of our ground truths: Designing referential databases in digital image processing », *Social Studies of Science*, 47 (6), pp. 811-840.

Jaton Florian (2019). « « Pardonnez cette platitude » : de l'intérêt des ethnographies de laboratoire pour l'étude des processus algorithmiques », *Zilsel*, 5 (1), pp. 315-339.

Jaton Florian (2021a). *The Constitution of Algorithms: Ground-Truthing, Programming, Formulating*. Cambridge (MA), The MIT Press.

Jaton, Florian (2021b) « Assessing Biases, Relaxing Moralism: On Ground-Truthing Practices in Machine Learning Design and Application », *Big Data & Society*, 8 (1), pp. 20539517211013570.

Kidder Tracy (2000). *The Soul of a New Machine*. New York, Back Bay Books.

Knorr-Cetina Karin (1981). *The manufacture of knowledge: an essay on the constructivist and contextual nature of science*. Pergamon Press.

Knorr-Cetina Karin (1999). *Epistemic Cultures: How the Sciences Make Knowledge*. Cambridge, Mass, Harvard University Press.

Knorr-Cetina Karin & Mulkay Michael Joseph (1983). *Science Observed: Perspectives on the Social Study of Science*. Sage Publications.

Latour Bruno (1987). *Science in Action — How to Follow Scientists & Engineers Through Society*. Cambridge, Mass., Harvard University Press.

Latour Bruno (1999). *Pandora's Hope: Essays on the Reality of Science Studies*. Cambridge, Mass, Harvard University Press.

Latour Bruno (2012). *Enquête sur les modes d'existence : Une anthropologie des Modernes*. Paris, La Découverte.

Latour Bruno & Woolgar Steve (2005). *La vie de laboratoire : La production des faits scientifiques*. Paris, Editions La Découverte.

Licoppe, Christian (2008). « Dans le 'carré de l'activité' : perspectives internationales sur le travail et l'activité. », *Sociologie du travail*, 50 (3): 287–302.

Lynch Michael (1985). *Art and Artifact in Laboratory Science: A Study of Shop Work and Shop Talk in a Research Laboratory*. London ; Boston, Routledge Kegan & Paul.

Mackenzie Adrian (2017). *Machine Learners: Archaeology of a Data Practice*. Cambridge, MA, MIT Press.

MacKenzie Adrian & Monk Simon (2004). « From Cards to Code: How Extreme Programming Re-Embodies Programming as a Collective Practice », *Computer Supported Cooperative Work (CSCW)*, 13 (1), pp. 91-117.

Myers Glenford J., Sandler Corey & Badgett Tom (2011). *The Art of Software Testing*. Hoboken, N.J, Wiley.

Netz Reviel (2003). *The Shaping of Deduction in Greek Mathematics: A Study in Cognitive History*. Cambridge University Press.

Noble Safiya Umoja (2018). *Algorithms of Oppression: How Search Engines Reinforce Racism*. New York, New York University Press.

O'Neil Cathy (2016). *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. New York, Crown.

Parrington Norman & Roper Marc (1989). *Understanding Software Testing*. Chichester, West Sussex, England; New York, John Wiley & Sons Australia Ltd.

Pasquale Frank (2016). *The Black Box Society: The Secret Algorithms That Control Money and Information*. Cambridge, Massachusetts London, England, Harvard University Press.

Pütz, Ole (2021). "Managing exactness and vagueness in computer science work: Programming and self-repair in meetings", *Social Studies of Science*, 51 (6), pp. 938–961.

Rosenberg Scott (2008). *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software*. New York, Three Rivers Press.

Seifert Colleen M. & Hutchins Edwin L. (1992). « Error As Opportunity: Learning in a Cooperative Task », *Human-Computer Interaction*, 7 (4), pp. 409–435.

Steiner Christopher (2012). *Automate This: How Algorithms Came to Rule Our World*. New York, Portfolio Hardcover.

Suchman Lucy (2006). *Human-Machine Reconfigurations: Plans and Situated Actions*. Cambridge ; New York, Cambridge University Press.

Ullman Ellen (2012). *The Bug: A Novel*. New York, Picador.

Vinck (1999). *Ingénieurs au quotidien*. Grenoble, PUG.

Vinck Dominique (2011). « Taking intermediary objects and equipping work into account in the study of engineering practices », *Engineering Studies*, 3 (1), pp. 25-44.

Weller Jean-Marc (2012). « Comment ranger son bureau ? Le fonctionnaire, l'agriculteur, le droit et l'argent », *Reseaux*, 171 (1), pp. 67–101.

Zhou Minghui, Chen Qingying, Mockus Audris & Wu Fengguang (2017). « On the Scalability of Linux Kernel Maintainers' Work" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 27–37. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery.

Ziewitz Malte (2016). « Governing Algorithms Myth, Mess, and Methods », *Science, Technology & Human Values*, 41 (1), pp. 3-16.

NOTES

1. Cet article, qui a reçu le soutien du Fonds national suisse de la recherche scientifique (projet Sinergia 180350), reprend des éléments présentés dans Jaton (2021 : chapitres 1 et 4). Je remercie Gabriel Alcaras, Manuel Boutet et Antoine Larribeau pour la coordination de ce numéro spécial, ainsi que les trois évaluateur·rice·x·s anonymes pour leurs pertinentes remarques.

2. L'une des forces de l'industrie informatique est en effet la possibilité de capitaliser sur des publications académiques. Sur ce sujet, voir Jaton (2021a : 37-40).

3. Par le terme "cours d'action", j'entends ici l'ensemble des gestes, conversations, outillages, dispositifs, et documents par lesquelles une activité – souvent ordinaire – se déroule. Le terme est en tout point synonyme de "situation" ou de "pratique".

4. À noter que juste après la rédaction du présent article, un travail ethnométhodologique, de grande qualité, sur les pratiques de programmation a été proposé par Pütz (2021).

5. Je propose quelques éléments de réponse dans Jaton (2021a : 93-134).

6. Ces bases de données référentielles, centrales au travail de façonnage des algorithmes, sont souvent appelées *ground truths* au sein des communautés de recherche en traitement du signal et en science des données. La notion de *ground truth* – parfois traduite en « vérité terrain » – est issue de la discipline géographique. Sur ce sujet, voir Jaton (2017 ; 2021b) et Henriksen et Bechmann (2020).

7. Sur ces pratiques de formulation qui consistent, entre autres, à enrôler des énoncés mathématiques, voir Jaton (2021a : pp. 197-280).

8. Pour mener à bien ce projet, j'ai dû acquérir des compétences dans les langages de programmation Python, PHP, JavaScript et Matlab.

9. Les huit entretiens audiovisuels ont abouti à plus de mille pages de transcriptions détaillées.

10. Je traite, ailleurs, du *crowdsourcing* dans le domaine du traitement du signal (Jaton 2017 ; 2019 ; 2021a). Pour des analyses détaillées de ce que fait le *crowdsourcing* au capitalisme contemporain, voir Casilli (2019) et Gray et Suri (2019).

11. Les langages dits « de haut niveau » (*high-level programming languages*) sont considérés, le plus souvent, comme les langages qui en passent par un interpréteur ou un compilateur avant d'être exécutés. Matlab, Python, R ou C peuvent ainsi être considérés, selon cette définition, comme des langages de haut niveau.

12. Le message d'erreur « `Cell contents references from a non-cell array object` » est celui de la version 2015 de Matlab. Dans sa version actuelle – 2020 – le message d'erreur est plus spécifique : « `Brace indexing is not supported for variables of this type` ».

13. Cette traduction ligne par ligne correspond à ce qui est vécu par la programmeuse. Dans la trajectoire de INT, et de la plupart des autres interpréteurs, la liste numérotée de symboles est traduite en un arbre syntaxique qui, souvent, ne conserve pas l'organisation ligne par ligne de l'Éditeur.

14. Dans le langage de programmation Matlab, toute déclaration qui n'est pas conditionnelle et qui ne se termine pas par un point-virgule est, par défaut, imprimée par l'interpréteur dans la Commande. Matlab diffère en ceci de plusieurs autres langages de programmation de haut niveau pour lesquels les opérations d'impression doivent être spécifiées par une instruction (typiquement, l'instruction « `print` »).

15. Comme je le mentionne ailleurs (Jaton, 2019), en m'appuyant sur Dear et Jasanoff (2010 : 761), ce qui relie – lâchement – les chercheur·euse·x·s de cette communauté de recherche hétérogène labellisée STS est la conviction que « depuis les travaux de Ludwick Fleck, Thomas Khun, et David Bloor, la science n'est pas le froid royaume de l'empirisme logique et des principes premiers, que nous préexistons nos connaissances du monde (tout comme le monde préexiste nos connaissances), que la matérialité est centrale à la formation et à la mise à l'épreuve des vérités scientifiques, et que les sciences et les dynamiques des pratiques scientifiques et techniques sont un terrain fertile à l'analyse sociale, politique et éthique » (ma traduction).

16. Il peut s'agir ici d'une limite des *Software Studies*, comme par exemple introduites dans Fuller (2008). En considérant le code achevé, ces études tendent parfois à négliger les opérations pratiques qui ont conduit à l'achèvement du code. Le regard des *Software Studies* reste néanmoins important en ce qu'il permet d'apprécier certains effets culturels des produits logiciels, ce que ma méthode microsociologique n'est pas tout à fait en mesure de faire (du moins directement).

17. Une belle exploration de ce travail collectif de constitution de chaînes de référence se trouve dans le roman réaliste de Ullman (2012). Pour des exemples plus récents, voir les discussions plus ou moins spécialisées autour des *Linux Kernel maintainers*, par exemple Zhou et al. (2017) ou Anderson (2020).

RÉSUMÉS

Les multiples fonctionnements numériques – algorithmes inclus – qui participent à nos actions quotidiennes se doivent de passer, d'une manière ou d'une autre, par les mains expertes de programmeur·euse·x·s capables de traduire désirs, plans et intuitions en listes d'instructions exécutables par une machine informatique. Pour autant, ce travail concret de façonnage de programmes n'a que peu fait l'objet d'études de terrain détaillées. Si l'on compare les quelques rares enquêtes, importantes mais isolées, à la massivité de la programmation en tant qu'activité située dont les produits ne cessent d'irriguer nos sociétés informatisées, le contraste est inouï : tout reste à faire, ou presque. Prenant acte de la situation, cet article propose quelques outils et analyses pour tenter de la changer. Dans un premier temps, il expose une technique d'investigation, ainsi qu'un formalisme, permettant de rester au plus près du déroulement séquentiel des situations de programmation. Dans un deuxième temps, il analyse des matériaux récoltés selon cette méthode d'enquête – nommée ici sociologie de l'activité de programmation – et suggère qu'une part importante des pratiques de codage consiste, parfois, à aligner des inscriptions afin de renseigner l'état d'entités distantes (interpréteurs, compilateurs, processeurs) et, en retour, indexer un emplacement au sein d'un document numéroté. L'article finit par discuter les perspectives nouvelles sur l'étude sociale du code induites par cette démarche analytique.

The many digital functionings — including algorithms — that contribute to our daily lives must pass, in one way or another, through the expert hands of programmers capable of translating desires, plans, and intuitions into numbered lists of instructions that can be executed by a computing device. However, this concrete work of shaping computer programs has been the topic of few detailed field studies. If one compares the rare, but important, ethnographic works with the massivity of computer programming as a situated activity whose products are constantly irrigating our computerized societies, the contrast is astonishing: everything, almost, remains to be done. Taking note of the situation, this article proposes some tools and analyses to try to change it. First, the paper presents an investigation technique, as well as a formalism, that allows ethnographers to stay as close as possible to the sequential unfolding of computer programming situations. Secondly, the paper analyzes ethnographic materials collected according to this method of investigation — coined here sociology of computer programming activity — and suggests that an important part of coding practices consists, sometimes, in aligning inscriptions in order to inform the state of distant entities (interpreters, compilers, processors) and, in turn, to index a location within a numbered document. The paper ends by discussing the new perspectives on the social study of code induced by this down-to-earth analytical framework.

INDEX

Keywords : computer programming, code, algorithm, sociology, ethnography, activity, STS
Mots-clés : programmation, code, algorithme, sociologie, ethnographie, activité, STS

AUTEUR

FLORIAN JATON

STS Lab, Institut des Sciences Sociales, Université de Lausanne