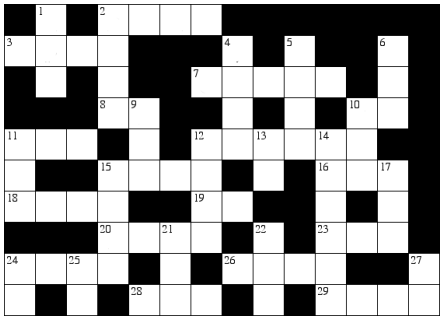
Langages croisés



HORIZONTALEMENT. **2**. sh + AWK + sed. « So far we've managed to avoid turning [it] into APL. :-) » — Larry Wall. **3**. Langage de programmation algébrique fonctionnel. Pierre précieuse. **7**. Langage compilé à structure de blocs, conçu, utilisé, et normalisé pour la conception de grands systèmes de télécommunications robustes. Supporté par GCC. **8**. Métalangage inventé par Robin Milner pour écrire le prouveur de théorèmes LCF. **10**. L'interprète de commande Unix **11**. « Within [it], there is a much smaller and cleaner language struggling to get out. » — Bjarne Stroustrup. **12**. Des règles, des objets et des relations. Langage de choix pour attaquer les NP-complets. **15**. N'a pas de rapport avec Javascript. **16**. Première personne à programmer, comtesse. **18**. Pour faire courir la tortue. **19**. Puissant langage de macros lexicales. **20**. Langage de haut niveau basé sur le concept d'ensembles. Inspirateur distant de Python. **23**. La prothèse du C. **24**. Objectif parenthèses. **26**. Fils direct de Simula, de mêmes parents. Introduit le concept de *pattern*. **28**. L'inventeur de l'encapsulation, des itérateurs, précurseur des structures génériques.

VERTICALEMENT. **1**. « [It] is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past : it creates a new generation of coding bums. » — Edsger Dijkstra. Nécessite un clavier spécial. Calculer les nombres premiers de 1 à R dans ce langage : $(\sim\ R \in R \circ . \times R)/R \rightarrow 1 \downarrow \iota R$. **4**. Langage de description de matériel (circuit électroniques etc.). Acronyme. **5**. Le langage universel vu par IBM dans les années 1960. A servi à écrire MULTICS. « Using [it] must be like flying a plane with 7,000 buttons, switches, and handles to manipulate in the cockpit. » — Edsger Dijkstra. **6**. David Korn répond à Steve Bourne. **9**. Langage interprété commun dans les jeux. **13**. Précurseur de FORTRAN par l'amiral Grace Murray Hopper. 1 m^2 de papier. **14**. Un petit saligaud de Pascal avec cinq politiques de passage d'arguments : `in` (par valeur), out (par résultat), `in-out` (par valeur-résultat), `var` (par référence), `name` (par nom). **17**. Quand M$ fait travailler les serveurs. **21**. Interpréteur à louer pour gros projets. La syntaxe de Tk. **22**. TchouC. **25**. Mélange magicien de paradigmes de programmation. Du Mozart. **27**. John Backus invente la programmation fonctionnelle. **29**. « (It) is the red pill. » — John Fraser. Lots of Insipid and Stupid (not *that* slow) Parentheses.

En bref

Les publications (disponibles sur publis.lrde.epita.fr)

- **Fast and Exact Discrete Image Restoration Based on Total Variation and on Its Extensions to Levelable Potentials.**[7] par Jérôme Darbon and Marc Sigelle accepté à *SIAM Conference on Imaging Science* . - 22 février

Les projets

- Transformers 0.4[8], publié fin 2005 par Olivier Gournet, fournit un nouveau moteur de désambiguïsation exploitant les grammaires attribuées, une couverture complète du langage C ISO 99, ainsi qu'une extension du C avec pré- et post-conditions.
- Sylvain Peyronnet et Johan Oudinet travaillent sur l'analyse statistique de la structure du web, dans le but de pouvoir améliorer l'indexation de pages et la lutte contre le web spam. De nombreuses études ont déjà été faites sur le sujet dont une récemment par Google. Une partie des résultats (un crawl de plusieurs dizaines de millions de pages) est déjà publiée.[9] D'autres résultats viendront bientôt.

Les gens Après avoir réalisé son stage de Master au LRDE, puis une année de vacation, Roland Levillain est depuis janvier 2006 un permanent du labo. Il y est ingénieur de recherche, et travaille particulièrement sur Olena et Tiger.

[7] http://publis.lrde.epita.fr/200605-SIAM
[8] Transformers 0.4, http://transformers.lrde.epita.fr/TransformersRelease04.
[9] http://sylvain.berbiqui.org/web-statistics-fr/

L'air de rien
N° 2

L'aléastriel du **L**aboratoire de **R**echerche et de **D**éveloppement de l'**E**PITA[1] Numéro 2, Mars 2006

Edito

L'air de rien II : le retour...
par *Didier Verna*

Vous avez aimé le premier numéro ? Alors vous aimerez peut-être celui-ci ! Au sommaire, le premier opus d'une série intitulée « La minute Lispienne » : une collection d'articles destinés à mieux faire connaître ce langage étonnant qu'est LISP, ainsi qu'à casser les trop nombreuses idées reçues dont il est victime.

Cette minute Lispienne (à ne pas mettre entre parenthèses) est suivie de deux articles présentant des applications distinctes de traitement d'images : le débruitage et l'élimination d'objets.

Si vous arrivez jusque là (il est interdit de commencer la lecture par la dernière page), vous aurez le droit de vous détendre avec une grille de « langages croisés » suivie des incontournables potins du labo...

Bonne lecture !

La (1re) minute Lispienne

La légende de la lenteur
par *Didier Verna*

Parmi les trop nombreux mythes et légendes qui courent sur LISP, sa prétendue lenteur est probablement la pire de toutes les afflictions. Cette idée reçue, aujourd'hui colportée par des gens qui ne savent pas de quoi ils parlent, ou qui ont juste « entendu dire que ... » trouve sa source dans des vérités qui datent de la naissance du langage, mais qui sont caduques depuis plus de 20 ans.

LISP fut effectivement un langage lent... à ses débuts. Plusieurs raisons à cela : par conception, premièrement, le langage était entièrement dédié au traitement des listes (d'où son nom : LISt Processing). La liste était la seule structure de données disponible, la glu universelle et unique pour représenter tout agrégat d'objets (les fonctions y compris). Or, la manipulation permanente de listes implique un déréférencement massif de pointeurs, ce qui est très coûteux. Deuxièmement, LISP était un langage dynamiquement typé : les objets LISP (et non pas les variables) portaient en eux-mêmes leur information de type, ce qui impliquait une recherche, voire une vérification de type à l'exécution ; processus non moins coûteux. En corollaire dramatique à cet état de fait, notons que la représentation unifiée de tout « objet » LISP, aux fins de typage dynamique, empêche de représenter les nombres dans le format natif de l'architecture matérielle sous-jacente, et rend impossible l'utilisation directe des instructions matérielles, car les opérateurs arithmétiques doivent être polymorphes. Par conséquent, les performances de LISP sur du calcul numérique étaient tout à fait déplorables. Si l'on rajoute à cela le fait que les premiers LISP étaient seulement interprétés, l'on comprend bien que la légende de sa lenteur a pris racine dans des faits bien réels et avérés.

Mais voilà plus de 20 ans que la situation a changé du tout au tout, notamment grâce à l'effort de standardisation de COMMON LISP, dont l'initiation remonte au début des années 1980. Tout d'abord,

[1] L'air de rien, http://publis.lrde.epita.fr/LrdeBulletin.

les listes sont aujourd'hui tombées en obsolescence. COMMON LISP dispose de nombreuses glus alternatives (vecteurs, tableaux, structures, tables de hash, *etc.*) dont la manipulation est bien plus efficace. Les seules traces de listes qui subsistent encore se trouvent dans des programmes mal écrits, ou bien quand les performances ne sont pas un problème, ou encore dans les phases initiales (prototypage rapide) d'un programme en cours de développement. Deuxièmement, LISP n'est plus un langage typé dynamiquement. Ou plus exactement, il ne l'est plus *nécessairement*. Bien sûr, vous pouvez continuer à manipuler des « objets » LISP dans le vague. Mais si vous connaissez le type des objets manipulés, vous pouvez les déclarer explicitement, auquel cas le typage devient statique. Dès lors, selon que vous choisissez de privilégier la sûreté du code ou son efficacité, les déclarations de type serviront d'assertions (vérification de type statique, ou dynamique le cas échéant) ou permettront d'optimiser le code en utilisant les meilleures représentations possibles pour chaque objet de type connu, voire même des représentations natives quand c'est possible. Ce processus, appliqué aux valeurs numériques, est connu sous le terme d'*inlining*. Pour terminer, passons vite sur le fait que LISP peut aujourd'hui être compilé aussi bien qu'interprété.

Alors de grâce, comparons ce qui est comparable : pour un programme LISP bien écrit (entendre en particulier : utilisant des structures de données appropriées), correctement typé et bien sûr compilé, les performances obtenues sont comparables à celle du C, et ce, même pour du calcul numérique.

Optimisation Exacte

par *Jérôme Darbon*

1- gauche : image originale, droite : image bruitée

De nombreux problèmes de débruitage et de restauration d'image peuvent s'exprimer comme une minimisation d'une énergie. Cette dernière est composée de deux termes : le premier mesure la fidélité de l'image débruitée par rapport à l'image originale observée, tandis que le second terme correspond à un *a priori* sur l'image recherchée.

Cette formulation énergétique est bien adaptée à la résolution de ce type de problème : en effet, on montre facilement que ce cadre énergétique est équivalent à une formulation probabiliste du problème. Cette dernière permet de prendre en compte facilement le bruit qui corrompt l'image ; rappelons qu'un bruit est souvent modélisé par un processus aléatoire dont on ne connaît que la distribution de probabilité. Par conséquent, la fidélité à l'image originale, à un pixel donné, est mesurée par cette probabilité. La figure 1 présente une image originale ainsi que sa version corrompue par un bruit impulsif de 70% : autrement dit, soit un pixel garde sa valeur (avec une probabilité 0,3) soit il est corrompu (avec une probabilité de 0,7), et ce pixel prend une valeur aléatoire tirée uniformément sur l'intervalle discret $\{0, \ldots, 255\}$.

Trouver un bon *a priori* sur les images naturelles demeure un sujet de recherche. Un modèle très souvent utilisé est celui de la Variation Totale. Cette dernière présente l'intérêt de bien préserver les contours, contrairement à bien d'autres modèles qui ont tendance à les lisser et donc de considérer des images qui sont floues.

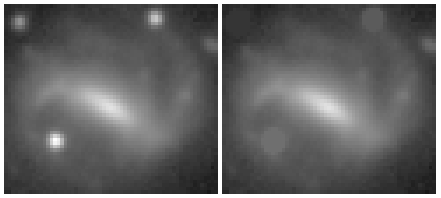
2- Résulats : notre méthode et par filtrage médian

La minimisation de cette énergie est un problème délicat. En effet, elle possède plusieurs minima locaux et il est difficile de trouver celui qui est minimum. En revanche, des techniques récentes permettent de reformuler ce problème de minimisation d'énergie comme un problème de coupure minimale dans un graphe. Ce dernier problème, qui est un problème de combinatoire, a été largement étudié dans le cadre de la théorie des graphes et nous disposons d'algorithmes efficaces pour le résoudre. En résumé, afin de débruiter l'image, nous construisons un graphe tel que sa coupure minimale produise un minimiseur de l'énergie associé à la restauration de l'image. Ce minimiseur est une image, qui est présentée sur la figure 2. A titre de comparaison, nous présentons le résultat obtenu par un filtrage médian. Ce dernier consiste à remplacer la valeur du pixel par la valeur médiane calculée sur une fenêtre centrée sur ce pixel.

Ce travail a été réalisé en collaboration avec Marc Sigelle de Télécom Paris et a mené à deux publications dans une revue scientifique internationale[2].

Un philtre anti-poétique

par *Thierry Géraud*



Dans de nombreux problèmes de traitement d'images, une difficulté à surmonter est la présence d'objets indésirables dans les images. S'il s'agit de vos photos de vacances, cela peut se traduire par la présence en arrière-plan d'un chien s'abandonnant contre un arbre, gâchant ainsi le portrait de votre petit(e) ami(e). Pour le projet EFIGI, on souhaite étudier la « forme » des galaxies, et malheureusement, dans les images de galaxies se trouvent des étoiles. Si on ne les gomme pas, les résultats de l'étude seront faux. Si ce gommage n'est pas automatique alors, à raison d'une seconde pour le gommage manuel d'une étoile, il faudrait passer plusieurs jours pour nettoyer la masse de données acquise chaque jour ...

Dans les logiciels de traitement d'images que vous connaissez, tel The Gimp, vous avez la possibilité de caresser à la main le mulot pour modifier localement une image, ou *a contrario*, d'appliquer un « filtre » à toute l'image. Certains filtres vous permettent d'enlever du bruit, de contraster l'image, de rehausser les contours, *etc.* À y regarder de plus près, un inconvénient majeur des filtres est d'affecter *toute* l'image, y compris les endroits de l'image où il n'y a pas de bruit, où le contraste est suffisant, *etc.* Au bout du compte, améliorer l'image (ou lui enlever des défauts) entraîne des effets indésirables ailleurs. Avec une image de galaxie, nous sommes confrontés à ce même type de problème : nous voulons enlever les étoiles, mais surtout, ne pas toucher au reste de l'image.

Nous nous sommes alors intéressés à une famille de filtres : les « nivellements ». Ces filtres, qui appartiennent au domaine de la « morphologie mathématique », ont comme propriété sympathique de préserver les contours dans les images ; ils ne les rendent pas flous et ils ne les déplacent (déforment) pas. Pour expliquer simplement le fonctionnement de ces filtres, imaginez qu'une image est un paysage. Les lignes et les colonnes correspondent respectivement aux longitudes et latitudes, et les niveaux de gris à l'altitude. Une zone plutôt claire dans l'image correspond alors à une montagne et une zone plutôt foncée à une vallée. Les étoiles, dans les images du ciel, sont des montagnes aux formes très particulières : elles sont plutôt circulaires, leur étendue est plutôt faible (mais pas trop faible quand même) et l'altitude de leur sommet est toujours élevée. Pour éliminer les étoiles d'une image, nous allons procéder en plusieurs étapes. Tout d'abord, transformer la représentation de l'image : d'un tableau 2D de niveaux de gris, nous passons à un arbre où les nœuds terminaux représentent les bosses de l'image (des montagnes plus ou moins prononcées). Puis nous allons calculer pour chaque nœud un ensemble d'attributs : un critère de circularité, l'étendue, l'altitude du pic maximal, *etc.* L'application d'une règle simple sur ces attributs nous permet de sélectionner les montagnes dont la forme est caractéristique de la présence d'une étoile. Enfin, nous n'avons plus qu'à raser ces montagnes et à reconstituer l'image résultat.

Ne cherchez pas ce type de filtres dans The Gimp, vous ne les trouverez pas ; ils font encore partie du domaine de la recherche.

Ce travail de Christophe Berger et Nicolas Widynski[3] (ing2 du LRDE) a été réalisé avec Olena[4], encadré par Théo, et présenté cette année lors d'un séminaire du labo. Derrière ce travail se cachent en fait des mathématiques — la « morphomathématique[5] » —, de l'algorithmie — autour de l'algorithme d'Union-Find de Tarjan[6] —, et bien entendu, des réalisations informatiques.

[2] revue scientifique internationale, http://publis.lrde.epita.fr/2006XXX-JMIVb.
[3] travail de Christophe Berger et Nicolas Widynski, http://publis.lrde.epita.fr/Seminar-2005-07-13.
[4] Olena, http://olena.lrde.epita.fr.
[5] morphomathématique, http://www.cmla.ens-cachan.fr/Utilisateurs/vachier/TranspMorpho.pdf.
[6] algorithme d'Union-Find de Tarjan, http://publis.lrde.epita.fr/200504-ISMM.