



Grenoble INP - Ensimag
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Rapport de stage assistant ingénieur

Effectué à l'Institut Fourier

Implémentation de nouvelles commandes de calcul pour les probabilités et les statistiques pour le logiciel Giac/Xcas

ETTAHRI Hicham

2^e année - Filière IF

01 juin 2026 - 24 juillet 2026 (8 semaines)

Institut Fourier, Université Grenoble Alpes / CNRS
100 rue des maths - BP 74
38402 SAINT MARTIN D'HÈRES

Responsable de stage
PARISSE Bernard

Tuteur de l'école
MARANINCHI Florence

Table des matières

1	Introduction	1
2	Contexte	1
2.1	Le projet Giac/Xcas	1
2.2	Déroulement du stage	2
3	Problématique, existant et solution retenue	2
3.1	Le problème à résoudre	2
3.2	Objectifs	3
3.3	État de l’art	3
3.4	Contraintes fixées par la structure d’accueil	4
3.5	La solution retenue	4
4	Architecture et mise en œuvre	4
4.1	Vision haut niveau	4
4.2	Intégrer une commande dans le moteur	5
4.3	Analyse en composantes principales	5
4.4	Régression linéaire multiple	7
4.5	Simulation de processus stochastiques	7
4.6	Black-Scholes : dériver la formule plutôt que recopier les Greeks	9
4.7	Monte-Carlo	9
4.8	Estimation des paramètres	10
5	Validation par tests de régression	10
5.1	Le dispositif existant	10
5.2	Le fichier de tests du module	11
5.3	Cas des commandes aléatoires	12
6	Résultats	12
6.1	Atteinte des objectifs	12
6.2	Efficacité de la réduction de variance	13
6.3	Regard critique	14
7	Bilan personnel	14
8	Conclusion	15
	Références	16

A	Liste des commandes ajoutées	16
B	Documentation technique rédigée pendant le stage	16
B.1	Installation et développement	16
B.2	Documentation, tests de régression et menus	18
B.3	Déboguer avec gdb	20

Table des figures

1	Logo du projet Giac/Xcas	2
2	Chaîne ACP sur un tableau 8×5	6
3	Sortie de la commande <code>mlr</code>	7
4	Trajectoire simulée par le schéma de Milstein	8
5	Quarante trajectoires d'un processus d'Ornstein-Uhlenbeck	9
6	Estimation standard et antithétique d'une espérance	13
7	Prix d'un call par Monte-Carlo, standard et antithétique	13

Liste des tableaux

1	Les cinq blocs fonctionnels	3
2	Fichiers d'un groupe de tests de régression	11
3	Extrait des tests portant sur des commandes aléatoires	12
4	Couverture fonctionnelle	12
5	Gain des variables antithétiques	13
6	Liste des 36 commandes ajoutées	16

1 Introduction

Un système de calcul formel (Computer Algebra System) manipule des expressions mathématiques de façon exacte : il dérive, intègre, factorise, diagonalise, au lieu de seulement calculer des valeurs approchées. Giac/Xcas, développé à l'Institut Fourier, est un CAS libre très utilisé dans l'enseignement des mathématiques en France. Son catalogue en algèbre et en analyse est comparable à celui des logiciels commerciaux, mais son offre en probabilités et statistiques s'arrêtait aux lois usuelles et aux indicateurs d'une seule variable. Ni analyse en composantes principales, ni régression multiple, ni simulation de processus aléatoires, ni méthode de Monte-Carlo. Un étudiant devait donc quitter Xcas pour R ou Python dès qu'il dépassait le calcul d'une moyenne ou d'une variance.

Ce constat pose deux questions. Peut-on ajouter à Giac un ensemble cohérent de commandes de statistique multivariée et de calcul stochastique, fiables et utilisables directement depuis l'interface ? Et qu'apporte le fait de les écrire dans un système de calcul formel plutôt que d'utiliser une bibliothèque numérique existante ?

L'objectif du stage était de concevoir, implémenter, documenter et tester des commandes couvrant cinq domaines : l'analyse en composantes principales, la régression linéaire multiple avec inférence, la simulation de processus stochastiques, l'évaluation d'options européennes, l'estimation Monte-Carlo, et l'estimation des paramètres d'une équation différentielle stochastique.

2 Contexte

L'Institut Fourier est un des laboratoires de mathématiques de l'Université Grenoble Alpes, situé à Saint-Martin-d'Hères. Il est dédié aux mathématiques fondamentales, à la différence du Laboratoire Jean Kuntzmann (LJK) qui rassemble les mathématiques appliquées. Les deux sont des unités mixtes de recherche avec le CNRS. Les équipes de l'Institut Fourier sont réparties en thèmes, dont la physique mathématique, à laquelle appartient Bernard Parisse, l'encadrant de stage.

Le cadre n'est donc pas un service de développement logiciel mais un laboratoire de mathématiques, autour d'un logiciel maintenu aujourd'hui par une seule personne.

2.1 Le projet Giac/Xcas

Giac/Xcas est un système de calcul formel libre sous licence GPL 3, développé par Bernard Parisse, enseignant-chercheur à Grenoble depuis 1992. Le projet vient de programmes écrits en 1993 pour les calculatrices HP48, puis d'une année passée chez Hewlett-Packard en 1998-1999 sur le moteur de calcul formel de la HP49G. Après le retrait du constructeur du secteur des calculatrices graphiques, le développement d'un CAS libre pour ordinateur a été lancé en 2000 avec Renée De Graeve.

Le projet comporte deux niveaux :

- **Giac** est le moteur de calcul, une bibliothèque C++ qui contient les algorithmes et l'analyseur syntaxique du langage ;
- **Xcas** est une interface graphique au-dessus de ce moteur ; **icas** en est la version en

ligne de commande.

Une commande ajoutée dans Giac devient donc disponible dans toutes les interfaces. La bibliothèque dépasse largement le logiciel Xcas : c'est le moteur de calcul formel de GeoGebra, des calculatrices HP Prime et du logiciel PocketCAS, et elle est portée sur des calculatrices Casio, Texas Instruments et NumWorks. Giac/Xcas est un paquet fourni par les principales distributions Linux et figure parmi les modules optionnels de SageMath.

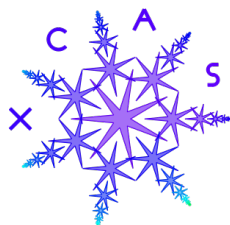


Figure 1 – Logo du projet Giac/Xcas.

2.2 Déroulement du stage

Le stage a duré huit semaines, sur place. Le sujet a été proposé sous une forme ouverte : étendre les capacités du logiciel en probabilités et statistiques. Le découpage en cinq blocs, la liste des commandes et leurs signatures ont été construits au fil de l'avancement.

L'encadrement a été rapproché : des échanges quasi quotidiens pendant les premières semaines, consacrées à l'appropriation du code et de la chaîne de compilation. Le fonctionnement est ensuite devenu plus autonome, bloc par bloc, mais les échanges sont restés réguliers : ils servaient à valider l'avancement et le code produit, et à répondre aux questions techniques rencontrées.

Le logiciel est maintenu par une seule personne, donc il n'y a pas de revue de code par des collègues. Ce rôle est tenu par la documentation et les tests : ce sont eux qui permettront à un futur contributeur de comprendre et de modifier le code. Cette contrainte a orienté une partie du travail, dont la rédaction de trois guides.

3 Problématique, existant et solution retenue

3.1 Le problème à résoudre

Avant le stage, Giac offrait un socle utilisable : statistiques descriptives d'une variable (`mean`, `stddev`, `variance`, `median`, `quantile`), covariance et corrélation de deux variables, régression linéaire simple à une variable (`linear_regression`), et les lois de probabilité usuelles avec leurs densités, quantiles et générateurs. À cela s'ajoutaient des briques d'algèbre linéaire réutilisables : diagonalisation (`egv`), décomposition QR (`qr`), dérivation symbolique (`derive`), méthode de Newton (`newton`).

Le manque commençait dès que l'on dépassait ce socle : tout ce qui met en jeu plusieurs variables. Aucune des opérations suivantes ne tenait en une commande : réduire la dimension d'un tableau de données, ajuster une réponse sur plusieurs variables explicatives et savoir lesquelles sont significatives, simuler une trajectoire aléatoire, estimer une

espérance par simulation avec un intervalle de confiance, retrouver les paramètres d'un modèle aléatoire à partir de données observées.

Il fallait donc fournir ces objets dans le langage du CAS, pour qu'un utilisateur puisse enchaîner simulation, estimation et tracé dans la même session.

3.2 Objectifs

Le travail a été organisé en cinq blocs. Pour chacun, l'objectif était de livrer deux niveaux d'interface : des commandes élémentaires pour construire son propre calcul, et une commande de synthèse pour obtenir un résultat immédiat.

Bloc	Objectif	Commande de synthèse
1. ACP	Réduire la dimension d'un tableau	<code>pca(X,q)</code> : scores, nombre de composantes, variance cumulée
2. Régression multiple	Ajuster $Y = X\beta + \varepsilon$ et fournir l'inférence	<code>mlr(X,y)</code> : coefficients, prédictions, résidus, R^2 , variances, intervalles de confiance
3. Processus stochastiques	Simuler des trajectoires en temps continu	<code>gbm</code> , <code>ornstein_uhlenbeck</code> , <code>cir</code> , <code>sde_euler</code> et <code>sde_milstein</code>
4. Black-Scholes	Évaluer une option européenne et ses sensibilités	<code>bs_greeks</code> : prix et les cinq Greeks
5. Monte-Carlo et estimation	Estimer une espérance avec incertitude ; retrouver les paramètres d'une EDS	<code>mc_estimate</code> , <code>fit_gbm</code> , <code>fit_ou</code>

Table 1 – Les cinq blocs et leur commande de synthèse.

3.3 État de l'art

Trois familles de solutions existantes ont été examinées.

Bibliothèques numériques spécialisées. R (`prcomp`, `lm`), Python (NumPy, statsmodels, scikit-learn) et MATLAB couvrent tout le périmètre visé, avec des solutions validées. Leurs limites tiennent au contexte, pas à la qualité : elles sont purement numériques, donc incapables de dériver une formule ou de renvoyer une valeur propre exacte ; elles imposent de sortir de Xcas et de ses commandes de tracé et de calcul formel.

CAS propriétaires. Maple et Mathematica proposent des modules de finance et de processus stochastiques intégrés à leur langage. Ils montrent que ces commandes ont leur place dans un CAS, mais ce sont des logiciels fermés et payants.

Conclusion. La solution consiste donc à écrire des commandes natives, à partir des primitives de Giac, exactes quand c'est possible et numériques sinon. C'est ce dernier point qui les distingue d'une bibliothèque numérique : sur des données entières,

`pca_eigenvalues([[4,2],[2,3]])` renvoie $\left(\frac{7+\sqrt{17}}{2}, \frac{7-\sqrt{17}}{2}\right)$ et non deux flottants, et `pca_explained_ratio([6,4,1])` renvoie les fractions 6/11, 4/11, 1/11.

3.4 Contraintes fixées par la structure d'accueil

1. **API du moteur** : Toute commande est une fonction `gen f(const gen &args, GIAC_CONTEXT)`, accompagnée d'un triplet d'enregistrement qui l'inscrit dans la table des commandes.
2. **Conventions internes** : Validation des arguments avant tout calcul, messages d'erreur en français.
3. **Livrable complet** : Une commande est livrée lorsqu'elle a une section dans le manuel `doc/fr/cascmd_fr.tex`, une entrée en français et en anglais dans `doc/aide_cas`, une entrée de menu dans `doc/fr/xcasmenu` et un test de régression dans `check/`.

3.5 La solution retenue

La solution est un module de 36 commandes publiques ajoutées au fichier `src/proba.cc`, organisé selon trois principes.

Une **architecture en trois couches** sépare les fonctions utilitaires internes, les commandes élémentaires et les commandes de synthèse.

Un **format de sortie uniforme** : les commandes de synthèse renvoient une liste de paires `["nom", valeur]`. L'affichage est un tableau lisible, mais l'objet reste une structure de données que l'utilisateur peut indexer pour récupérer une grandeur.

La **réutilisation des primitives du CAS** plutôt que leur réimplémentation : `egv` pour la diagonalisation, `qr` pour la décomposition orthogonale, `derive` pour les dérivations, les lois de Giac pour les tirages aléatoires.

4 Architecture et mise en œuvre

4.1 Vision haut niveau

Une commande tapée par l'utilisateur est analysée par le parseur de Giac, qui la reconnaît grâce à la table des commandes et transmet ses arguments, empaquetés dans un seul objet `gen`, à la fonction C++ correspondante. Ces fonctions forment trois couches :

La **couche utilitaire** n'est pas exposée à l'utilisateur. Elle regroupe les opérations répétées dans tout le module : lecture et validation d'un réel ou d'un entier strictement positif, vérification qu'un argument est une matrice, tirage d'un vecteur de variables $\mathcal{N}(0, 1)$, mise en forme des trajectoires, construction du tableau de résultats. Ces fonctions sont déclarées `static`, donc invisibles hors du fichier.

La **couche des commandes élémentaires** expose chaque étape du calcul : `cov_matrix`, `pca_eigenvalues`, `mlr_beta`, `bs_delta`, `mc_estimate`. Ce découpage permet à un enseignant de dérouler la chaîne étape par étape devant des étudiants, au lieu de présenter une boîte noire.

La **couche de synthèse** enchaîne ces étapes et renvoie le tableau nommé : `pca`, `mlr`, `bs_greeks` etc.

Le moteur reçoit la commande de l'utilisateur et appelle la fonction C++ correspondante. Les commandes de synthèse appellent les commandes élémentaires, qui appellent la couche utilitaire.

4.2 Intégrer une commande dans le moteur

Pour ajouter une commande, il faut écrire une fonction et l'inscrire dans la table des commandes du langage, sinon le parseur ne la reconnaît pas. Le mécanisme est un triplet d'enregistrement, écrit pour chacune des 36 commandes et complété par une déclaration `extern` dans l'en-tête `src/proba.h`.

```
gen _bs_price(const gen & args, GIAC_CONTEXT){
  /* validation des arguments puis calcul */
}
static const char _bs_price_s []="bs_price";
static define_unary_function_eval(_bs_price, &_bs_price, _bs_price_s);
define_unary_function_ptr5(at_bs_price, alias_at_bs_price, &_bs_price, 0, true);
```

Listing 1 – Triplet d'enregistrement d'une commande, ici `bs_price`.

Une recompilation complète de Giac prend plusieurs dizaines de minutes, ce qui est incompatible avec un cycle de développement normal. Deux scripts sont fournis par B. Parisse (`mklocal` et `mkxlocal`) qui ne recompilent que les fichiers objets modifiés et refont uniquement l'édition de liens.

4.3 Analyse en composantes principales

L'ACP réduit un tableau $X \in \mathbb{R}^{n \times p}$ à une matrice $Z \in \mathbb{R}^{n \times q}$ avec $q \leq p$, en projetant les données centrées sur les q premiers vecteurs propres de la matrice de covariance Σ . Avec la décomposition spectrale $\Sigma = VDV^T$, $D = \text{diag}(\lambda_1, \dots, \lambda_p)$ et $\lambda_1 \geq \dots \geq \lambda_p$, la matrice de projection W est formée des q premières colonnes de V , et $Z = X_c W$.

Le calcul repose sur `egv`. La difficulté est que les vecteurs propres ne sont pas uniques. Si v est vecteur propre de Σ , $-v$ l'est aussi, et rien ne privilégie l'un ou l'autre. Une diagonalisation peut donc renvoyer V ou une matrice obtenue en changeant le signe de certaines colonnes. Le signe des scores devient alors arbitraire, deux versions du logiciel peuvent donner des sorties opposées, et aucun test par comparaison exacte n'est possible.

Une convention de normalisation a donc été fixée : dans chaque colonne, la composante de plus grande valeur absolue est rendue positive. Cette règle est déterministe, indépendante de l'implémentation de `egv`, et laisse la matrice orthogonale. Les colonnes sont triées par valeur propre décroissante et normalisées.

La figure 2 montre la chaîne complète sur huit observations et cinq variables. Les deux premières composantes expliquent $64,3\% + 28,0\% = 92,3\%$ de la variance : `pca_select_q(S, 0.90)` renvoie $q = 2$, et `pca_scores` projette les huit observations dans l'espace vectoriel correspondant.

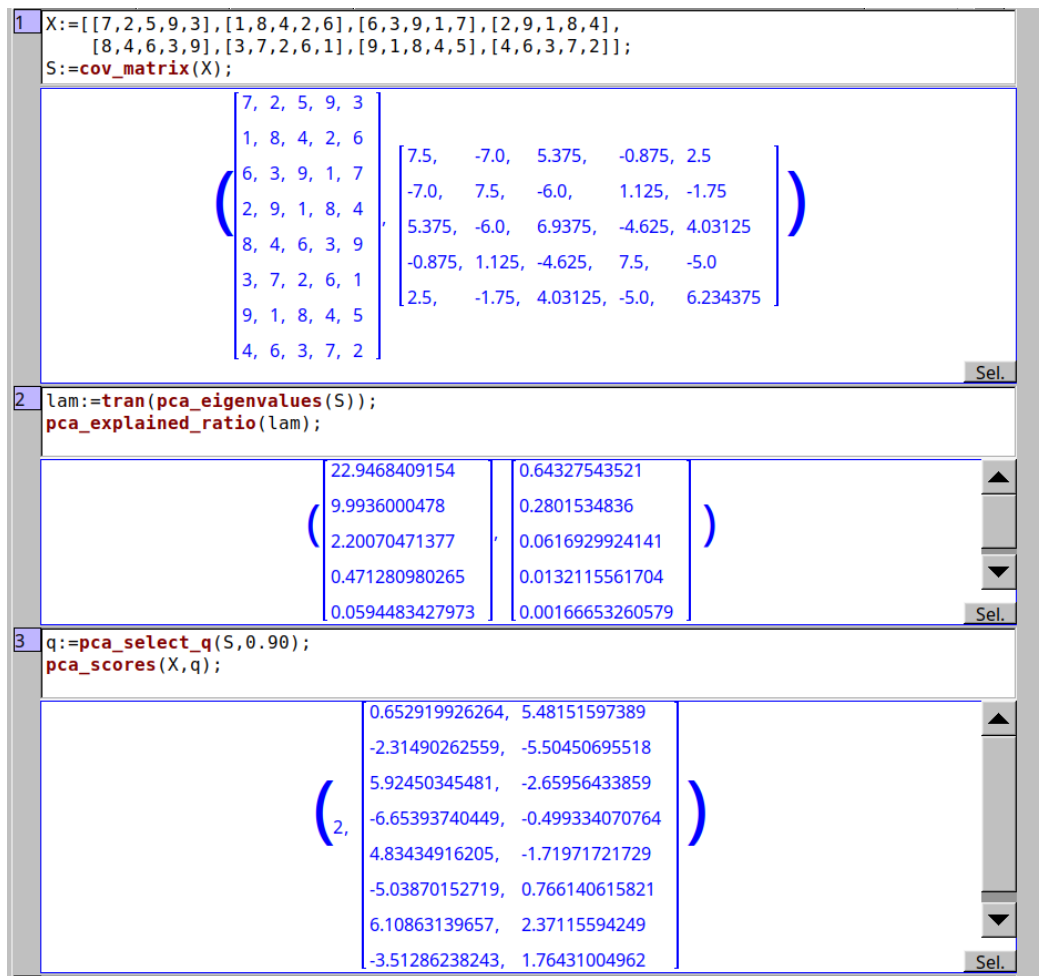


Figure 2 – Chaîne ACP sur un tableau 8×5 dans une session Xcas. La matrice de covariance S (5×5), les cinq valeurs propres triées avec les ratios de variance expliquée, la sélection automatique de q au seuil de 90 % et la matrice des scores Z (8×2).

4.4 Régression linéaire multiple

Le modèle est $Y = X\beta + \varepsilon$ avec $\varepsilon \sim \mathcal{N}(0, \sigma^2 I_n)$, et l'estimateur des moindres carrés s'écrit $\hat{\beta} = (X^T X)^{-1} X^T y$, X étant augmentée d'une colonne de 1. Deux points ont été traités au-delà de cette formule.

Décomposition QR : Former puis inverser $X^T X$ fait perdre des chiffres significatifs sur les coefficients. La décomposition QR réduite $X_{\text{aug}} = Q_1 R_1$ évite de passer par ce produit : $\hat{\beta}$ s'obtient en résolvant $R_1 \hat{\beta} = Q_1^T y$, avec R_1 triangulaire. Elle est proposée par un argument optionnel `use_qr=1` plutôt qu'imposée. Si $X^T X$ est singulière, la commande renvoie un message d'erreur qui signale la multicolinéarité au lieu d'un résultat aberrant.

Inférence : La variance résiduelle non biaisée $\hat{\sigma}^2 = \text{SS}_{\text{res}}/(n - p - 1)$ donne l'écart-type de chaque coefficient, $\text{se}(\hat{\beta}_i) = \hat{\sigma} \sqrt{[(X^T X)^{-1}]_{ii}}$, d'où l'intervalle de confiance à 95 % $\hat{\beta}_i \pm t_{n-p-1, 0.975} \cdot \text{se}(\hat{\beta}_i)$. Le quantile est celui de la loi de Student et non de la loi normale, parce que σ est inconnu et remplacé par son estimation. En mode QR, l'égalité $(X^T X)^{-1} = (R_1^T R_1)^{-1}$ donne les mêmes quantités sans reformer $X^T X$.

Un intervalle contenant 0 signale un coefficient non significatif au seuil de 5 %. La figure 3 le montre : sur sept observations et trois variables, le R^2 vaut 0,815 mais un seul des quatre intervalles exclut zéro. Le modèle explique une bonne part de la variance sans qu'on puisse l'attribuer à la plupart des variables, car il ne reste que trois degrés de liberté et les intervalles sont donc très larges.

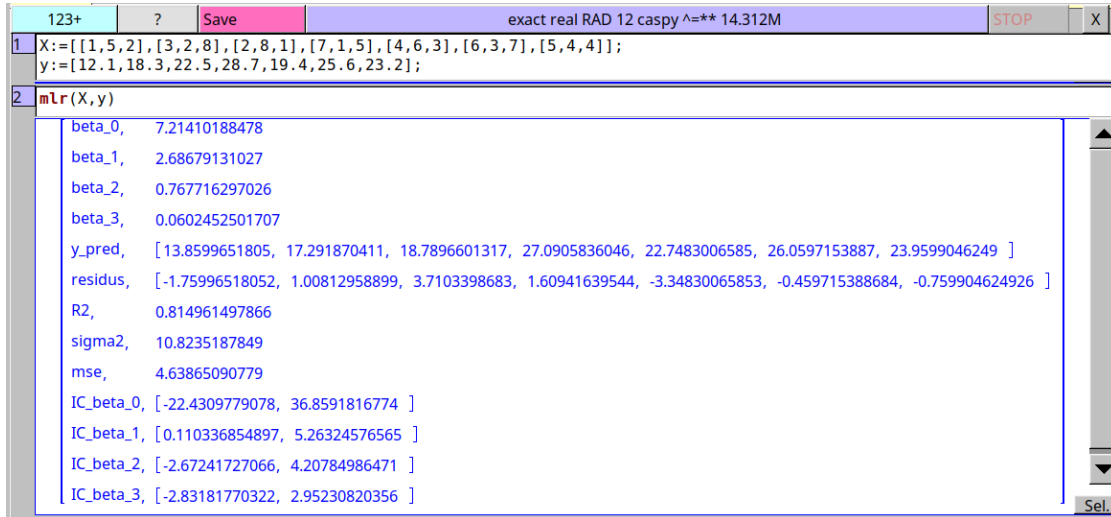


Figure 3 – Sortie de `mlr` sur $n = 7$ observations et $p = 3$ variables : coefficients, prédictions, résidus, R^2 , variance résiduelle non biaisée, MSE et un intervalle de confiance à 95 % par coefficient.

4.5 Simulation de processus stochastiques

Une équation différentielle stochastique décrit un processus soumis à une tendance déterministe et à un bruit brownien :

$$dX_t = b(t, X_t) dt + \sigma(t, X_t) dW_t.$$

Le module fournit six commandes de simulation : quatre modèles classiques et deux schémas génériques.

Simulation exacte quand une solution explicite existe : Le mouvement brownien géométrique, qui modélise le prix d'un actif risqué, admet la solution $S_t = S_0 \exp\left((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t\right)$. Le processus d'Ornstein-Uhlenbeck a une transition gaussienne explicite. Dans ces deux cas la simulation est exacte, car il existe un schéma exact.

Schéma adapté sinon : Le modèle de Cox-Ingersoll-Ross, $dR_t = a(b - R_t)dt + \sigma\sqrt{R_t}dW_t$, n'a pas de forme connue. La simulation utilise un schéma d'Euler, où $\sqrt{R_t}$ est remplacé par $\sqrt{\max(R_t, 0)}$.

Milstein et le calcul formel : Les commandes `sde_euler` et `sde_milstein` acceptent une EDS quelconque, la dérive b et la diffusion σ étant fournies comme expressions en t et x . Le schéma d'Euler-Maruyama,

$$X_{k+1} = X_k + b(t_k, X_k)\Delta t + \sigma(t_k, X_k)\Delta W_k,$$

converge à l'ordre 1/2. Le schéma de Milstein atteint l'ordre 1 en ajoutant un terme correctif :

$$X_{k+1} = X_k + b\Delta t + \sigma\Delta W_k + \frac{1}{2}\sigma\partial_x\sigma\left(\Delta W_k^2 - \Delta t\right).$$

Ce terme fait intervenir $\partial_x\sigma$. Dans une bibliothèque numérique, l'utilisateur devrait fournir cette dérivée, ou bien l'implémentation l'approcherait par différences finies, ce qui réintroduit une erreur là où l'on cherchait à en gagner. Ici σ est une expression symbolique : un appel à `derive` suffit. La précision supplémentaire du schéma de Milstein est donc obtenue sans rien demander à l'utilisateur, parce que le code s'exécute dans un système de calcul formel. La figure 4 montre une trajectoire de $dX_t = \sin(X_t)dt + \cos(X_t)dW_t$, qui est une équation sans solution explicite dont les deux coefficients sont passés comme expressions libres.

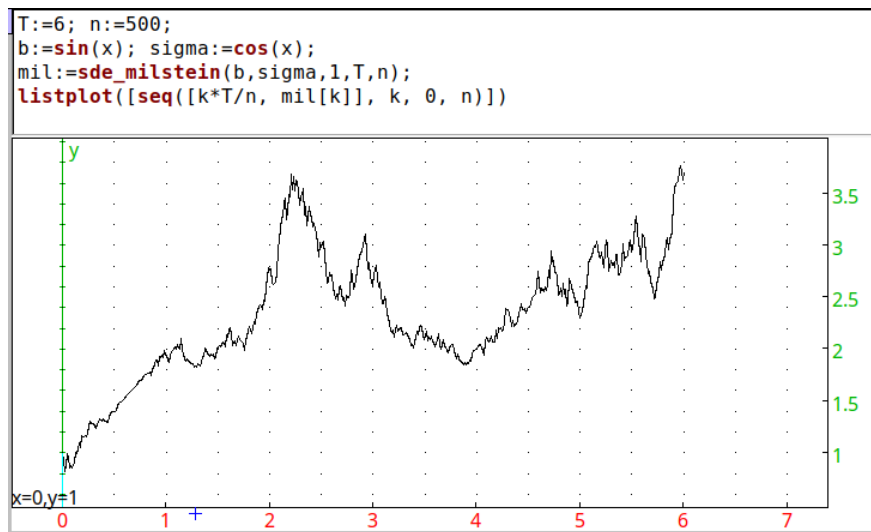


Figure 4 – Une trajectoire de $dX_t = \sin(X_t)dt + \cos(X_t)dW_t$ sur $[0, 6]$ à 500 pas, simulée par `sde_milstein`.

La figure 5 montre le retour à la moyenne du processus d'Ornstein-Uhlenbeck sur quarante trajectoires indépendantes partant du même point. Le nuage se resserre autour du niveau d'équilibre $\theta = 2$. La possibilité de renvoyer m trajectoires en une commande, sous forme de matrice, a été ajoutée pour ce type de visualisation.

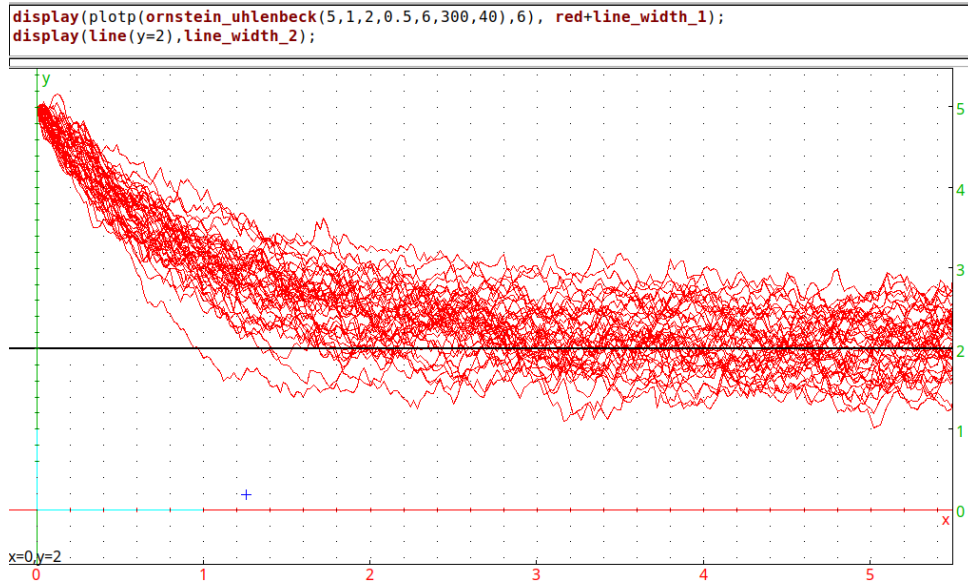


Figure 5 – 40 trajectoires indépendantes d'un processus d'Ornstein-Uhlenbeck ($X_0 = 5$, $\alpha = 1$, $\theta = 2$, $\sigma = 0,5$) sur $[0, 6]$.

4.6 Black-Scholes : dériver la formule plutôt que recopier les Greeks

Le prix d'une option européenne est l'espérance actualisée du flux terminal sous la probabilité risque-neutre, $\text{prix} = e^{-rT} \mathbb{E}_{\mathbb{Q}}[h(S_T)]$, ce qui donne pour un call $C = S N(d_1) - K e^{-rT} N(d_2)$. Les *Greeks* sont les sensibilités de ce prix aux paramètres du marché : $\Delta = \partial C / \partial S$, $\Gamma = \partial^2 C / \partial S^2$, $\nu = \partial C / \partial \sigma$, $\Theta = \partial C / \partial t$, $\rho = \partial C / \partial r$.

Le prix est construit une fois comme expression symbolique, et chaque Greek s'obtient en dérivant cette expression, puis en l'évaluant numériquement. La fonction de répartition normale vient de la fonction d'erreur, $N(x) = (1 + \text{erf}(x/\sqrt{2}))/2$, qui est dérivable par Giac.

4.7 Monte-Carlo

La méthode de Monte-Carlo estime $I = \mathbb{E}[g(X)]$ par la moyenne empirique $\hat{I}_N = \frac{1}{N} \sum_i g(X_i)$. Le théorème central limite donne l'incertitude : $\hat{I}_N \pm 1,96 \hat{\sigma} / \sqrt{N}$. La convergence en $1/\sqrt{N}$ est lente (diviser l'erreur par dix demande cent fois plus de tirages).

Le choix d'architecture de ce bloc est la séparation entre le noyau statistique et les applications. La commande `mc_estimate` ne tire rien et ne connaît aucune loi : elle reçoit une liste de réalisations et renvoie la moyenne, le rayon de l'intervalle de confiance, ses deux bornes, l'erreur standard et le nombre de tirages. Les commandes applicatives (`mc_mean`,

`mc_bs_price` et leurs variantes) produisent les Y_i puis appellent la même fonction de mise en forme.

Variables antithétiques : Au lieu de N tirages indépendants, on tire N paires en utilisant chaque Z_i et son opposé $-Z_i$, et on moyenne $Y_i = (g(Z_i) + g(-Z_i))/2$. Comme $-Z$ suit la même loi que Z , l'estimateur reste sans biais. Sa variance vaut

$$\text{Var}(Y_i) = \frac{\sigma^2}{2}(1 + \rho), \quad \rho = \text{Corr}(g(Z), g(-Z)),$$

contre $\sigma^2/2$ pour deux tirages indépendants. Le gain est acquis si et seulement si $\rho < 0$, c'est-à-dire pour g monotone. Pour g paire, $\rho = +1$ et l'erreur augmente d'un facteur $\sqrt{2}$. Cette condition est documentée dans le manuel. Les gains mesurés sont donnés en section 6.

4.8 Estimation des paramètres

Les blocs précédents vont du modèle vers les données. Le dernier fait l'inverse : à partir d'une trajectoire observée, retrouver les paramètres qui l'ont engendrée.

Pour le mouvement brownien géométrique, la solution exacte implique que les log-rendements $r_k = \ln(S_{k+1}/S_k)$ sont indépendants et gaussiens, de moyenne $(\mu - \frac{1}{2}\sigma^2)\Delta$ et de variance $\sigma^2\Delta$. Estimer (μ, σ) revient donc à estimer la moyenne et la variance d'un échantillon gaussien, puis à inverser ces relations.

Pour le processus d'Ornstein-Uhlenbeck, la transition gaussienne $X_{k+1} = a + bX_k + \varepsilon_k$ fait de l'estimation une régression linéaire d'un pas sur le précédent, suivie de l'inversion $\hat{\alpha} = -\ln \hat{b}/\Delta$ et $\hat{\theta} = \hat{a}/(1 - \hat{b})$.

5 Validation par tests de régression

5.1 Le dispositif existant

Giac dispose d'un dispositif de tests de régression mis en place par Bernard Parisse pour l'ensemble du logiciel. Il est rassemblé dans le répertoire `check/` et couvre les grands domaines du moteur : intégration, algèbre linéaire, arithmétique, équations différentielles, etc. Le principe est simple : figer les sorties connues correctes, puis vérifier après chaque modification du code qu'elles n'ont pas changé.

Un groupe de tests correspond à trois fichiers, décrits dans le tableau 2. Un quatrième fichier, `XXX.tst`, est produit à l'exécution.

Le script est simple. Il passe le fichier d'entrée dans la version en ligne de commande du logiciel, redirige la sortie dans `XXX.tst`, et compare ligne à ligne avec `diff`.

```
#!/bin/sh
export LANG=en_US.UTF8
../src/icas testproba > proba.tst
diff proba.tst proba.out
```

Listing 2 – Le script `check/chk_proba`.

Fichier	Rôle	Contenu
testXXX	entrée	Une commande Xcas par ligne. Écrit par le développeur.
XXX.out	référence	Les sorties correspondantes, vérifiées à la main une fois pour toutes. Sert de référence pour toutes les exécutions suivantes.
chk_XXX	script	Exécute l'entrée dans <code>icas</code> et compare le résultat à la référence.
XXX.tst	sortie	Produit à chaque exécution du script, comparé à <code>XXX.out</code> , puis écrasé à l'exécution suivante.

Table 2 – Les fichiers d'un groupe de tests de régression dans `check/`

Un `diff` vide signifie qu'aucune sortie n'a changé : les tests passent. Toute ligne affichée signale une différence entre le comportement actuel et la référence, et donc l'impact d'une modification. Ajouter le nom du script dans `check/Makefile.am` le fait lancer automatiquement par `make check`, avec tous les autres groupes du logiciel.

5.2 Le fichier de tests du module

La première ligne de tout fichier d'entrée est la même dans l'ensemble du projet. Les lignes suivantes sont des appels élémentaires.

```
cas_setup(0,0,0,1,0,[1e-10,1e-17],12,[1,50,0,25],0,0,0),xcas_mode(0);
cov_matrix([[1,2],[3,4],[5,6]]);
pca_eigenvalues([[4,2],[2,3]]);
pca_select_q([[2,1,1],[1,2,1],[1,1,2]],0.80);
mlr_beta([[1],[2],[3],[4],[5]],[2.1,3.9,6.2,7.8,10.1]);
mlr_r2([[1],[2],[3],[4],[5]],[2.1,3.9,6.2,7.8,10.1]);
bs_price(100,100,0.05,0.2,1,"call");
bs_greeks(100,100,0.05,0.2,1,"call");
gbm(100,0.05,0,1,4)[4]-100*exp(0.05);
mc_estimate([2,4,4,4,5,5,7,9])[0][1]-5;
```

Listing 3 – Extrait de `check/testproba`.

Le fichier de référence contient les sorties correspondantes, dans le même ordre :

```
[2.66666666667,2.66666666667],[2.66666666667,2.66666666667]],
[(sqrt(17)+7)/2,(-sqrt(17)+7)/2],
2,
[0.05,1.99],
0.997305328901,
10.4505835722,
vector[["prix",10.4505835722],["delta",0.636830651176],["gamma",0.0187620173458],
["vega",37.5240346917],["theta",-6.41402754644],["rho",53.2324815454]],
4.54747350886e-13,
0.0,
```

Listing 4 – Extrait de `check/proba.out`.

La mise en place suit toujours le même ordre : écrire le fichier d'entrée, lancer le

script une première fois pour produire `proba.tst`, vérifier à la main que chaque sortie est correcte, puis enregistrer ce fichier comme `proba.out`. C'est la seule étape où la correction des résultats est réellement contrôlée ; toutes les exécutions suivantes ne font que détecter les changements. La section `proba` comporte 115 appels et couvre les 36 commandes.

5.3 Cas des commandes aléatoires

Deux moyens ont été utilisés. D'une part l'annulation de l'aléa : à diffusion nulle, les schémas exacts redonnent la solution analytique, et l'écart mesuré vaut au plus $4,5 \times 10^{-13}$. D'autre part les invariants, vrais quelles que soient les valeurs tirées : parité call-put, différence des deltas, taille et noms des sorties. Le tableau 3 en donne des exemples.

Test	Attendu	Obtenu
<code>gbm(100,0.05,0,1,4) [4]</code> $-100e^{0.05}$	0	$4,5 \times 10^{-13}$
<code>ornstein_uhlenbeck(3,1,5,0,1,4) [4]</code> $-(5 + (3 - 5)e^{-1})$	0	0
<code>sde_milstein(-x,0,1,1,4) [4]</code> $-0,75^4$	0	0
$C - P$ contre $S - Ke^{-rT}$	égalité	4,87705754993 des deux côtés
$\Delta_{\text{call}} - \Delta_{\text{put}}$	1	1,0
$\Gamma_{\text{call}} - \Gamma_{\text{put}}$	0	0,0
<code>mc_estimate([1,1,1,1])</code> : rayon de l'IC	0	0
<code>mc_mean_antithetic(x,x,1000)</code> : estimation	0	0

Table 3 – Tests portant sur des commandes aléatoires.

6 Résultats

6.1 Atteinte des objectifs

Les cinq blocs prévus ont été livrés, chacun avec la chaîne complète exigée : code, documentation dans le manuel, entrée de menu et tests de régression.

Bloc	Commandes
ACP	8
Régression multiple	8
Processus stochastiques	6
Black-Scholes	7
Monte-Carlo	5
Estimation d'EDS	2
Total	36

Table 4 – Nombre de commandes par bloc.

Deux livrables non prévus s'y sont ajoutés : l'intégration dans l'interface avec des entrées d'aide et une arborescence de menus qui rendent les commandes trouvables sans en connaître le nom ; et trois guides pour les contributeurs suivants (section B).

6.2 Efficacité de la réduction de variance

Le protocole est une comparaison à coût égal : l'estimateur standard sur $N = 100\,000$ tirages contre l'estimateur antithétique sur 50 000 paires, soit également 100 000 évaluations. Les résultats sont donnés figures 6 et 7 et résumés dans le tableau 5.

<pre>mc_mean(exp(x),x,normald,[0,1],100000); mc_mean_antithetic(exp(x),x,50000);</pre>			
(	estimation,	1.6521948389	)
	rayon_ic,	0.0136484816837	
	ic_bas,	1.63854635721	
	ic_haut,	1.66584332058	
	erreur_std,	0.00696351106312	
	n,	100000	
	estimation,	1.6475919317	
	rayon_ic,	0.0104976668641	
	ic_bas,	1.63709426483	
	ic_haut,	1.65808959856	
	erreur_std,	0.00535595248166	
	n,	50000	
Sel.			

Figure 6 – Estimation de $\mathbb{E}[e^Z]$, $Z \sim \mathcal{N}(0,1)$, de valeur exacte $e^{1/2} \approx 1,6487$. À gauche l'estimateur standard sur 100 000 tirages, à droite l'antithétique sur 50 000 paires.

S:=100; K:=100; r:=0.05; sigma:=0.2; T:=1;

bs_price(S,K,r,sigma,T);

mc_bs_price(S,K,r,sigma,T,100000);

mc_bs_price_antithetic(S,K,r,sigma,T,50000);

estimation, 10.4738366988

rayon_ic, 0.0916105510393

ic_bas, 10.3822261478

ic_haut, 10.5654472499

erreur_std, 0.0467400770609

n, 100000

estimation, 10.4614261773

rayon_ic, 0.0646203453843

ic_bas, 10.3968058319

ic_haut, 10.5260465226

erreur_std, 0.0329695639716

n, 50000

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

10.4505835722,

),

Figure 7 – Prix d'un call européen ($S_0 = K = 100$, $r = 0,05$, $\sigma = 0,2$, $T = 1$). La valeur analytique 10,4505835722 est à gauche, puis l'estimation standard et l'estimation antithétique à coût égal.

Cas	Valeur exacte	SE standard	SE antithétique	Gain en variance
$\mathbb{E}[e^Z]$	1,648721	$6,96 \times 10^{-3}$	$5,36 \times 10^{-3}$	$\div 1,69$
Call Black-Scholes	10,450584	$4,67 \times 10^{-2}$	$3,30 \times 10^{-2}$	$\div 2,01$

Table 5 – Gain des variables antithétiques à coût égal (100 000 évaluations dans les deux colonnes).

6.3 Regard critique

Plusieurs limites sont à signaler :

Le calcul numérique est fait en double précision alors que Giac permet mieux. Le moteur sait manipuler des flottants à précision arbitraire. Le choix est cohérent avec les contraintes de portabilité et suffisant pour l’usage visé.

Tous les intervalles de confiance sont asymptotiques. Ceux de Monte-Carlo reposent sur le théorème central limite, ceux de l’estimation d’EDS sur la normalité asymptotique du maximum de vraisemblance. Ils ne valent donc que pour un grand nombre de tirages ou une trajectoire longue.

La reproductibilité n’est que globale. Le générateur pseudo-aléatoire de Giac se contrôle par `srand`, qui agit sur toute la session. Il n’y a pas de graine locale à un appel, ce qui empêche de rejouer une trajectoire précise.

Le périmètre financier est étroit. Seules les options européennes vanille sont traitées, alors que la simulation permettrait d’évaluer des options dépendant du chemin.

Le coût des Greeks n’est pas optimisé. La dérivation symbolique est refaite à chaque appel. Négligeable pour un calcul isolé, cela devient inutilement coûteux pour construire une grille de prix ; un cache des expressions dérivées serait le premier gain à réaliser.

Aucune mesure de performance comparative n’a été faite. Le module a été validé sur son exactitude, pas sur sa vitesse, et il n’existe pas de comparaison chiffrée avec R ou Python.

7 Bilan personnel

Le langage et le type `gen` : La difficulté initiale a été le modèle de programmation de Giac. Toutes les données du moteur sont représentées par un seul type `gen`, dont le contenu réel n’est connu qu’à l’exécution. Le compilateur ne vérifie donc pas la forme des arguments : une commande appelée avec un mauvais type compile et échoue à l’exécution. La validation des arguments devient alors une partie substantielle de chaque fonction, ce qui explique le poids de la couche utilitaire. Ce déplacement de la sécurité du compilateur vers le code est ce que j’ai le plus appris sur le plan de la conception.

La chaîne de compilation : Le premier obstacle réel du stage était de compiler Giac et de faire primer ma version locale sur le paquet installé par le système. Installer les dépendances, comprendre `configure` et `make`, découvrir `mklocal` et `mkxlocal` qui ramènent le cycle de compilation à quelques secondes : cette étape a pris une part notable des premières semaines. Aucun document ne décrivait ce chemin, j’ai donc rédigé un guide d’installation et de développement pour les contributeurs suivants. Il couvre les dépendances, la construction, les deux scripts, l’ajout d’une fonction dans un fichier existant ou dans un nouveau module, et les erreurs les plus fréquentes.

Les tests : Mes premiers tests comparaient directement des trajectoires simulées, et le `diff` n’était jamais vide. J’ai d’abord soupçonné mon code alors que c’était la conception

du test qui était fautive. J'en retiens une règle : tester un composant non déterministe consiste à identifier ce qui ne dépend pas du hasard dans son comportement.

Le débogage : Les objets `gen` s'affichent mal avec les commandes usuelles de `gdb`, tant qu'on ne connaît pas la commande spécifique fournie par le projet. L'apprentissage des points d'arrêt, du parcours de la pile d'appels et de `valgrind` pour les erreurs mémoire non reproductibles a donné lieu à un deuxième guide, consacré au débogage. Un troisième guide documente toute la chaîne : documentation, tests de régression et menus.

Compétences relationnelles : Travailler avec un mainteneur unique demande une adaptation particulière. Beaucoup de conventions du projet ne sont écrites nulle part, elles se lisent dans le code existant ou s'obtiennent en posant la bonne question. J'ai appris à préparer mes questions, c'est-à-dire à cerner ce que je ne comprenais pas avant de solliciter du temps, et à considérer que respecter le style du projet passe avant mes préférences. J'ai aussi appris à écrire pour un lecteur absent : mes guides et mes sections de manuel s'adressent à des personnes que je ne rencontrerai pas.

8 Conclusion

Le stage partait de deux questions. La première était de savoir s'il était possible d'ajouter à Giac/Xcas un ensemble cohérent de commandes de statistique multivariée et de calcul stochastique. La réponse est oui : 36 commandes ont été ajoutées au moteur, couvrant l'analyse en composantes principales, la régression linéaire multiple avec inférence, la simulation de processus stochastiques, l'estimation Monte-Carlo avec réduction de variance et l'estimation des paramètres d'une EDS. Chacune est documentée dans le manuel, accessible depuis l'aide et les menus, et protégée par 115 tests de régression.

La seconde question était de savoir ce qu'apporte l'écriture de ces commandes dans un système de calcul formel. Trois éléments y répondent. Le schéma de Milstein atteint l'ordre 1 sans que l'utilisateur fournisse la dérivée de la diffusion, obtenue par dérivation symbolique. Les cinq *Greeks* sont dérivés de la formule de prix, ce qui garantit leur cohérence par construction. Sur des données exactes, les valeurs propres et ratios de variance expliquée sont renvoyés sous forme exacte. Aucun de ces trois points n'est accessible à une bibliothèque purement numérique.

Les résultats quantitatifs les plus nets concernent la réduction de variance, qui divise la variance de l'estimateur par 1,7 à 2,0 à coût égal, et la validation par cas dégénérés, avec un écart maximal de $4,5 \times 10^{-13}$ aux solutions analytiques.

Les prolongements découlent des limites identifiées : la mise en cache des expressions dérivées, d'autres techniques de réduction de variance comme les variables de contrôle, et l'exploitation de la précision arbitraire du moteur. Les trois guides rédigés pendant le stage visent à rendre ces prolongements possibles par quelqu'un d'autre, ce qui est la condition d'existence d'une contribution à un logiciel libre.

Références

1. Ensimag, *Introduction to Statistical Learning and Applications* (ISLA), cours de 2^e année. Analyse en composantes principales, régression linéaire multiple.
2. Ensimag, *Introduction aux produits dérivés* (IPD), cours de 2^e année. Modèle de Black-Scholes, valorisation risque-neutre, sensibilités.
3. Ensimag, *Principes et Méthodes Statistiques* (PMS), cours de 1^{re} année. Estimation, intervalles de confiance, maximum de vraisemblance.
4. B. Parisse, R. De Graeve, *Manuel de référence de Giac/Xcas*, fichier `doc/fr/cascmd_fr.tex` des sources du logiciel.
5. Page du projet Giac/Xcas, Institut Fourier, Université Grenoble Alpes.

A Liste des commandes ajoutées

Bloc	Commandes
ACP	<code>cov_matrix</code> , <code>pca_eigenvalues</code> , <code>pca_explained_ratio</code> , <code>pca_V</code> , <code>pca_loadings</code> , <code>pca_scores</code> , <code>pca_select_q</code> , <code>pca</code>
Régression	<code>mlr_beta</code> , <code>mlr_predict</code> , <code>mlr_residuals</code> , <code>mlr_r2</code> , <code>mlr_sigma2</code> , <code>mlr_mse</code> , <code>mlr_ic</code> , <code>mlr</code>
Simulation	<code>brownian</code> , <code>gbm</code> , <code>ornstein_uhlenbeck</code> , <code>cir</code> , <code>sde_euler</code> , <code>sde_milstein</code>
Black-Scholes	<code>bs_price</code> , <code>bs_delta</code> , <code>bs_gamma</code> , <code>bs_vega</code> , <code>bs_theta</code> , <code>bs_rho</code> , <code>bs_greeks</code>
Monte-Carlo	<code>mc_estimate</code> , <code>mc_mean</code> , <code>mc_bs_price</code> , <code>mc_mean_antithetic</code> , <code>mc_bs_price_antithetic</code>
Estimation	<code>fit_gbm</code> , <code>fit_ou</code>

Table 6 – Les 36 commandes ajoutées au fichier `src/proba.cc`.

B Documentation technique rédigée pendant le stage

Trois guides destinés aux futurs contributeurs de Giac ont été rédigés pendant le stage. Ils sont reproduits ci-dessous.

B.1 Installation et développement

Ce document explique comment installer Giac/Xcas à partir des sources, le compiler, et y ajouter ses propres fonctions ou modules. Il s'adresse aux personnes qui veulent contribuer au code (nouveaux stagiaires, développeurs, etc.).

Système. On suppose une distribution Ubuntu ; pour une autre distribution, adapter les noms de paquets à son gestionnaire. Pour les utilisateurs de Windows, on peut se placer dans une machine virtuelle (VirtualBox...) ou utiliser WSL (Windows Subsystem for Linux).

Dépendances. Installer les bibliothèques de compilation :

```
sudo apt install gcc g++ libgmp-dev libmpfr-dev libmpfi-dev libpari-dev \
libgsf-dev libxext-dev libpng-dev libjpeg-dev libreadline-dev \
libncurses-dev mesa-common-dev libx11-dev libxt-dev libxft-dev \
libntlm-dev libgl1-mesa-dev libgl-dev libao-dev hevea debhelper \
libfltk1.3-dev libxinerama-dev libxcursor-dev libxfixes-dev \
libcliquer-dev libsamplerate0-dev libglpk-dev liblapack-dev \
libblas-dev libecm-dev libusb-1.0-0-dev
```

Le paquet nauty s'appelle `libnauty2-dev` sur les versions anciennes et `libnauty-dev` sur les récentes ; cette ligne installe celui qui existe :

```
sudo apt install libnauty2-dev || sudo apt install libnauty-dev
```

Si une édition de liens échoue plus tard sur une bibliothèque manquante, installer son paquet `-dev`.

Récupérer les sources. Télécharger et décompresser :

```
wget https://www-fourier.univ-grenoble-alpes.fr/~parisse/giac/giac_stable.tgz
tar xf giac_stable.tgz
cd giac-2.0.0
```

Compiler. `configure` adapte les Makefile à la machine, `make` construit :

```
./configure
make -j4
```

Retirer le paquet système pour que la version locale prime :

```
sudo dpkg --remove giac
```

Les scripts `mklocal` et `mkxlocal`. Recompiler tout Giac est long ; ces scripts ne recompilent que les `.o` modifiés puis refont uniquement l'édition de liens, ce qui rend chaque itération rapide. `mklocal` relie `icas` (console), `mkxlocal` relie `xcas` (graphique). Les placer dans `src/` et les rendre exécutables :

```
cd src
chmod +x mklocal mkxlocal
```

Ajouter une fonction dans un fichier existant (le plus simple, par exemple `proba.cc`). Dans `proba.cc` :

```
gen_hello_giac(const gen & args, GIAC_CONTEXT){
    return 1;
}
static const char _hello_giac_s[]="hello_giac";
static define_unary_function_eval(__hello_giac,&_hello_giac,_hello_giac_s);
define_unary_function_ptr5(at_hello_giac,alias_at_hello_giac,
                           &__hello_giac,0,true);
```

Dans `proba.h`, à l'intérieur de namespace `giac` :

```
gen_hello_giac(const gen & args, GIAC_CONTEXT);
extern const unary_function_ptr * const at_hello_giac;
```

Le trio d'enregistrement et le `extern` sont obligatoires; le `true` final inscrit le nom dans `Xcas.proba.o` étant déjà dans `mklocal`, il n'y a rien d'autre à câbler.

Ajouter un nouveau fichier `monmodule.cc` et `monmodule.h` dans `src/`. Reprendre en tête les `#include` et le namespace `giac { ... }` de `proba.cc`, y mettre les fonctions avec leur trio d'enregistrement, et déclarer les `extern` dans `monmodule.h`. Ensuite :

- ajouter `monmodule.o` dans `mklocal` et `mkxlocal`, aux deux endroits : la liste du `make -j2 ... monmodule.o ...` (pour le compiler) et la ligne d'édition de liens `c++ -o icas -g -I.. ... monmodule.o ...` (pour le relier);
- ajouter `monmodule.cc` et `monmodule.h` aux sources dans `src/Makefile.am`: le `.cc` dans les lignes contenant les fichiers `.cc`, le `.o` dans celles contenant les fichiers `.o`;
- depuis le dossier `giac-2.0.0`, au-dessus de `src/` : `autoreconf` puis `./configure`.

Compiler et tester. Après chaque modification :

```
./mklocal
./icas
```

Dans `icas`, taper `hello_giac()` donne 1. Pour la version graphique :

```
./mkxlocal
./xcas
```

Dépannage. Message « Makefile absent » : lancer `./configure`. Fichiers de construction obsolètes : `autoreconf` puis `./configure`. Erreur d'édition de liens `-lXXX` : installer le paquet `libXXX-dev`.

B.2 Documentation, tests de régression et menus

Ce document explique comment, une fois une fonction ajoutée, la documenter dans le manuel, écrire ses tests de régression, et l'insérer dans l'aide et les menus de Xcas. Emplacements, depuis `giac-2.0.0` : manuel et menus dans `doc/fr/`, aide dans `doc/`, tests dans `check/`.

Ajouter de la documentation. Le manuel français est `doc/fr/cascmd_fr.tex`. On y ajoute une section : un `\section` avec le nom de la commande, une courte description, et des exemples au format « On tape / On obtient ». Ajouter aussi un `\index{}` avec le nom de la commande : cela indexe les pages pour l'affichage dans l'aide de Xcas, bouton « Plus de détails ». On peut mettre le nom en gras avec `|textbf`, par exemple `\index{ma_fonction|textbf}` :

```
\section{Titre : {\tt ma_fonction}}
\index{ma_fonction|textbf}
Description de la commande.
On tape :
\begin{center}{\tt ma_fonction(2,3)}\end{center}
On obtient :
\begin{center}{\tt 5}\end{center}
```

Le plus simple est de s'inspirer d'une commande existante dans le fichier. Compiler en deux passes :

```
cd doc/fr
pdflatex cascmd_fr.tex
pdflatex cascmd_fr.tex
```

Tests de régression : principe. Tout se passe dans `check/`. Pour un groupe `XXX` il y a trois fichiers : `testXXX` (les commandes), `XXX.out` (la référence, résultats connus corrects) et un script `chk_XXX`. Le script fait passer les commandes par `icas`, écrit la sortie ligne à ligne dans `XXX.tst`, puis la compare à `XXX.out` :

```
#!/bin/sh
export LANG=en_US.UTF8
../src/icas testXXX > XXX.tst
diff XXX.tst XXX.out
```

Un `diff` vide signifie aucun changement, donc tests corrects. Toute différence signale l'impact d'une modification.

Tests de régression : écrire. La première ligne est standard dans tous les fichiers de test :

```
cas_setup(0,0,0,1,0,[1e-10,1e-17],12,[1,50,0,25],0,0,0),xcas_mode(0);
```

Puis une commande par ligne, par exemple dans `testintegrate` :

```
integrate(1/(x**4-1)**10,x);
integrate((x**4+4*x**2+6*x+4)/(x+1)**2,x);
```

Mise en place : générer une première fois le `.tst`, vérifier à la main que les résultats sont corrects, et le sauvegarder en `.out`. Ensuite, si on modifie le code, `chk_XXX` montre exactement ce qui a changé par rapport à cette référence. Pour que le test soit lancé automatiquement par `make check`, ajouter `chk_XXX` dans `check/Makefile.am`.

Modifier l'aide (`doc/aide_cas`). Ce fichier alimente l'aide dans Xcas (touche F1) et la complétion. Une entrée par fonction :

```
# ma_fonction
1 Description en francais.
2 Description in English.
0 REAL,REAL
-1 fonction_liee_1
-2 fonction_liee_2
ma_fonction(2,3)
```

Soit : # suivi du nom; ligne 1 = description en français; 2 = en anglais; 0 = types d'arguments (REAL, INT, Vect, Expr, Str, entre crochets si optionnel); -1, -2, -3 = « voir aussi »; puis des exemples. Une ligne vide sépare les entrées.

Modifier les menus (`doc/fr/xcasmenu`). On modifie ici les menus en français; pour une autre langue, changer le répertoire `fr` (par exemple `en`). Une ligne par entrée, sous la forme `chemin/du/menu/nom_fonction` :

```
Cmds/Calculer/Diff int 1-d/integrate
Cmds/Proba stats/Calcul stochastique/gbm
```

Le chemin crée ou réutilise les sous-menus ; grouper ses fonctions par thème, dans un ordre logique.

Prise en compte de l'aide et des menus. Il suffit de les recopier dans l'installation de Giac, sans recompilation :

```
cp doc/aide_cas /usr/share/giac/doc/aide_cas
cp doc/fr/xcasmenu /usr/share/giac/doc/fr/xcasmenu
```

Relancer Xcas pour voir les changements.

Pour aller plus loin. On peut aussi ajouter :

- des sessions Xcas d'exemple via le fichier `doc/fr/xcasex` ;
- des traductions de noms de commandes via `doc/lang/keywords`, où `lang` vaut par exemple `fr` ou `de` (voir `doc/fr/keywords`) ;
- des traductions localisées de l'aide via `lang/aide_cas` (voir `doc/de/aide_cas`), ce qui évite de toucher au fichier principal `doc/aide_cas`.

B.3 Déboguer avec gdb

Ce document explique comment déboguer Giac/Xcas avec `gdb` : poser des points d'arrêt, exécuter pas à pas, inspecter les variables, en particulier les objets `gen`, et diagnostiquer un plantage. Les binaires locaux sont compilés avec les symboles de débogage (`-g`, déjà présent dans `mklocal`), donc utilisables directement avec `gdb`. L'intérêt est d'inspecter les variables en direct et de localiser un plantage, sans ajouter d'affichages ni recompiler.

En ligne de commande :

```
gdb -tui ./icas # -tui : affiche le code source en haut de l'écran
```

Commandes essentielles, dans `gdb` :

```
b _ma_fonction    # break : arrêt a l'entree de la fonction
b proba.cc:724    # ou a une ligne precise, sous la forme fichier:ligne
r                # run : lance icas. r fichier passe des arguments a icas
                  # (ex. un fichier contenant l'appel a deboguer).
                  # Apres recompilation, r seul relance avec les memes
                  # arguments : pas besoin de les retaper.
n / s            # next / step : ligne suivante / entrer dans l'appel
u                # until : continuer jusqu'a la fin d'une boucle
c                # continue
finish           # finir la fonction courante et voir sa valeur de retour
bt               # backtrace : pile d'appels
f N              # frame N : aller dans une fonction appelante pour y
                  # inspecter ses variables locales (avec p ou v)
p / v            # print une variable / v affiche un gen lisiblement
quit
```

Afficher un gen. Un objet `gen` s'affiche mal avec `print`. Deux fichiers interviennent. Le premier, `src/.gdbinit`, est chargé automatiquement par `gdb` au lancement depuis `src/` : il définit la commande `v`, qui affiche un `gen` lisiblement. Voici son contenu :

```
echo set print elements 0 for limitless printing
echo Defining v as print command for giac types\n
set print repeats 0
define v
print ($arg0).dbgprint()
end
echo set print thread-events off
set print thread-events off
```

Le second, `~/config/gdb/gdbinit`, autorise ce chargement automatique ; sans lui, `gdb` refuse de lire un `.gdbinit` local :

```
set auto-load safe-path /
```

On utilise ensuite `v` comme `print` :

```
v args                # affiche l'argument (un gen) de facon lisible
```

Exemple pratique.

```
$ gdb -tui ./icas
(gdb) b _mlr
Breakpoint 1 at 0xff68e: file proba.cc, line 724.
(gdb) r
[icas démarre ; on tape la commande :]
0>> mlr([[1,5,2],[3,2,8],[2,8,1],[7,1,5],[4,6,3],[6,3,7],[5,4,4]],
[12.1,18.3,22.5,28.7,19.4,25.6,23.2])
[l'exécution s'arrête au point d'arrêt, dans _mlr]
(gdb) n          4 fois : on arrive sur
                gen gX_aug = mlr_augment(gX, n, p);
(gdb) s          entre dans mlr_augment, pour l'inspecter
(gdb) finish     exécute mlr_augment jusqu'au bout et revient dans _mlr
(gdb) v gX_aug  affiche la matrice augmentée
```

Diagnostiquer un plantage. Lancer `gdb ./icas`, puis `run`, reproduire le plantage, puis `bt` pour voir où et pourquoi le code casse, `f N` pour remonter dans la frame de la fonction, et `v` ou `print` pour inspecter les variables à cet instant.

Crash aléatoire (mémoire). Les crashes aléatoires viennent souvent d'un problème d'allocation mémoire (accès hors limites, mémoire non initialisée...), difficile à reproduire sous `gdb`. `valgrind` les détecte :

```
valgrind ./icas
```

Il suffit ensuite de reproduire le problème, `valgrind` signale la ligne fautive et la nature de l'erreur mémoire.