programmez!
N°277
SEPTEMBRE
OCTOBRE
2026
29e année

Le magazine des dévs - CTO & Tech Lead

numéro - 300e numéro - 300e numéro - 300e numéro - 300

INFORMATIQUE QUANTIQUE

Dossier en collaboration avec OVHcloud & CLASSIQ

Les nouveautés du programme amazon future engineer

MicroPython : Comprendre les capteurs

Cartographie : Open source avec JAWG

J'ai vibé codé mon roman

Printed in EU - Imprimé en UE - BELGIQUE 7,50 € P928345 - Canada 10,55 $ CAN - SUISSE 14,10 FS - DOM Surf 8,10 € - TOM 1100 XPF - MAROC 59 DH

M 04319 - 277 - F: 7,00 € - RD

PORTAGE

Fiparco Portage

Portage salarial

Votre carrière est une priorité !

Votre contact

Ikram 07 86 45 01 75

fiparco-portage.fr



FIPARCO Sponsor de PROGRAMMEZ !

SOMMAIRE 277

4 **Edito**
277+23=300
François Tonic

7  **Sécurité**
Osons la cybersécurité stratégique
Fabrice Bru

8 **Agenda**
Les événements Programmez!
et l'agenda à venir

10  **Chronique**
La place mentale de l'IA pour aujourd'hui et demain
Marc Krusik

11   **Rentrée**
Amazon future engineer
Hugo Derri, Mathias Hiron

14 Dossier quantique

 Introduction
François Tonic

 **Martin Bombardelli**
 **Philippe Lacomme**
 **Bogdan Vulpercscu**
Découvrir le fonctionnement des ordinateurs quantiques photoniques

  Importance du HPC dans le calcul scientifique
Robert Wang, Olivier Hess

 Frameworks quantiques : panorama et premiers avec les émulateurs
Matthieu Garcia

 Introduction à la correction d'erreur quantique
Romain Perrier

  Hybridation des calculs quantique et classique
Samuel Deleplanque, Luis Fernando Pérez Armas

 Au fait, c'est quoi un qubit ?
Jean-Michel Torres

 Quobly : programmer l'ordinateur quantique sur silicium
Pablo Roseiro

 Why quantum computing's next breakthrough will be written in code, not qubits ?
Simon Fried

49  **IA**
J'ai vibe codé un roman
Raphiki

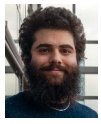
52  **Domotique**
Mon portail parle CAN et répond à Siri
Dominique Bultez

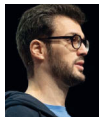
55  **Makers**
Premiers capteurs partie 1
Dominique Meurisse

60  **UX**
Comment réussir une bonne interface dans un monde complexe ?
Anthony Marois

62  **API**
Votre API mérite une interface graphique et c'est gratuit **Fred Berton**

67  **Rust**
Vérifier formellement des programmes Rust avec Creusot **Li-yao Xiang**

  Note Load Balancer Rust en production…
Jérôme Gurhem, Florian Lemaitre

72  **Langage**
J'écris encore du SQL
Sébastien Ferrer

79  **Maps**
Jawg Maps
Jones Magloire

Divers

5 Nos formules d'abonnement
6 Boutique Programmez!
9 Boutique Technosaures
21 Le nouveau numéro de Technosaures
83 Les partenaires 2026 de Programmez!

Programmez! est une publication bimestrielle de Nefer-IT. Adresse : 150, rue Lamarck, 75018 Paris – France. Pour nous contacter : redaction@programmez.com

277+23 = 300

Ce numéro de septembre, le n°277 est le 300e du magazine Programmez!. Une marque symbolique. Depuis 28 ans, nous essayons de vous apporter dans chaque numéro, une veille technologique, revenir aux fondations et de parler carrière.

Nous vous remercions, lectrices et lecteurs de nous lire, parfois depuis le tout 1er numéro. Nous rencontrons des lecteurs qui sont nés après la création de Programmez! Votre fidélité nous touche et nous motive chaque jour.

Merci à celles et ceux qui ont permis et permettent de builder chaque mois le magazine. Pierre, le maquettiste de nos magazines depuis plus de 20 ans, à Frédéric qui a œuvré sur le site web durant de nombreuses années, et à Vanessa qui a pris la relève. Merci aux responsables de distribution, à l'imprimeur et aux différents éditeurs. Un remerciement à tous les partenaires et sponsors qui soutiennent Programmez! année après année.

Nous tenons à remercier nos contributeurs, passés et actuels. Certains sont présents depuis plus de 10 ans. Ils ont en commun la passion du partage et d'expliquer une technologie, une bonne pratique. Et nous sommes impatients d'accueillir les futurs contributeurs. Pourquoi pas vous ? Un simple mail suffit : ftonic@programmez.com

Ce qui nous anime depuis 300 numéros ? Le code, la technologie, l'évolution du développement. L'IA est sans doute l'évolution la plus importante depuis 20 ans. Elle impacte la manière de coder, nous permet de nous concentrer sur le code le plus important. Mais, l'IA ne signifie pas que le métier de développeur soit mort et que tout sera généré par un agent.

Faire une revue de code n'est pas une option. Il faut le faire. Comprendre le code généré, c'est savoir ce qu'il contient, savoir comment il fonctionne, et permettre de le faire évoluer et de le maintenir.

Nous ne reviendrons pas ici sur les différentes attaques des modèles qui sont volontairement sortis de leurs rôles définis. Nous y reviendrons dans notre numéro spécial sécurité édition 2027 de fin septembre.

Il est intéressant de voir comment les grands projets open source se positionnent sur l'IA et les codes générés. Depuis le début 2026, les règles de contribution, de soumission des pull requests s'adaptent aux propositions et aux codes générés par IA. Certains projets ont vu déferler des pull requests partiellement ou totalement générées, à l'intérêt discutable ou de médiocre qualité. Pis, certaines ne sont visiblement ni revues, ni relues. Une des conséquences de ce déferlement, les mainteneurs ne peuvent plus traiter raisonnablement ces Pull Requests. Pour les soulager et préserver la qualité des projets, la gouvernance devient parfois très restrictive : allant d'une interdiction totale, à une certaine souplesse d'utilisation de l'IA. Cet été, Godot, OpenJDK, GCC se sont prononcés contre l'IA dans les pull requests. Linus avait dénoncé les nombreuses pull requests générées pour l'équipe sécurité du noyau Linux. Même s'il n'est pas opposé à l'IA, il ne faut pas submerger les mainteneurs.

Pour cet automne, Programmez! peaufine son agenda :

- Meetup #60 : quantique & IA, le 29 septembre dans les locaux de Zenika Paris
- DevCon #28 : sécurité, hacking et post-quantique édition 2027, 10 décembre au campus ESGI Paris

Retrouvez toutes les informations et les dates de nos événements sur programmez.com et meetup.com

Et n'oubliez pas :

- Suivez notre chaîne Youtube : https://www.youtube.com/@francoistonic-programmez
- Notre podcast dédié : PodDev sur votre plateforme de podcasts préférée
- Retrouvez sur GitHub les codes sources des articles : https://github.com/francoistonic



Et toute l'actualité technologique et développement sur programmez.com

François Tonic
Rédacteur en chef depuis longtemps

Nouvelle étape

Programmez! a vécu plusieurs vies avec Sepcom, Go02, Nefer-IT. Nous avons changé dans la continuité sans perdre de vue celles et ceux qui nous lisent. C'est la mission que nous essayons d'appliquer depuis que nous sommes rédacteur en chef du magazine. Depuis bientôt 2 ans, Fiparco et Jean-Christophe Tic ont ouvert une nouvelle page de notre "journey". Avec leur soutien, nous renforçons la partie IA et cybersécurité. Et plusieurs chantiers ont été ouverts ces derniers mois. Le plus important pour nous est le nouveau site web du magazine qui devrait être déployé cet automne.

Avec le soutien de Fiparco, d'autres projets seront lancés dans les prochains mois : un renforcement des événements, et de nos vidéos. Avec comme roadmap, notre 30e anniversaire qui aurait lieu durant la saison 2027-2028.

programmez!

Directives de compilation

Programmez! n°277
SEPTEMBRE - OCTOBRE 2026

Directeur de la publication : **Jean-Christophe Tic**
Rédacteur en chef : **François Tonic**
ftonic@programmez.com

Contacter la rédaction
redaction@programmez.com

Les experts techniques du numéro :

Fabrice Bru	Pablo Roseiro
Marc Krusik	Simon Fried
Martin Bombardelli	Raphiki
Philippe Lacomme	Dominique Bultez
Bogdan Vulpercscu	Dominique Meurisse
Robert Wang	Anthony Marois
Olivier Hess	Fred Berton
Matthieu Garcia	Li-yao Xiang
Romain Perrier	Jérôme Gurhem
Samuel Deleplanque	Florian Lemaitre
Luis Fernando Pérez Armas	Sébastien Ferrer
Jean-Michel Torres	Jones Magloire

Maquette
Pierre Sandré
Marketing – promotion des ventes
Agence BOCONSEIL - Analyse Media Etude
Directeur : Otto BORSCHA
oborscha@boconseilame.fr
Responsable titre :
Agnès Parra : aparra@boconseil.net
Téléphone : 06 15 46 15 88

Publicité
Nefer-IT
Tél. : 09 86 73 61 08
ftonic@programmez.com

Impression
Imprimerie Léonce Deprez
ZAC Artoipole 2
962 Allée de Belgique,
62128 Wancourt

Dépôt légal
A parution

Commission paritaire
0331 K 78366

ISSN
1627-0908

Abonnement
Abonnement (tarifs France) : 55 € pour 1 an, 90 € pour 2 ans. Etudiants : 45 €.
Europe et Suisse : 65 €
Algérie, Maroc, Tunisie : 64 € - Canada : 80 €
Tom - Dom : voir www.programmez.com.
Autres pays : consultez les tarifs sur www.programmez.com.
Accès aux archives : 25 € (1 an) 30 € (2 ans).
Pour toute question sur l'abonnement :
abonnements@programmez.com

Abonnement PDF
monde entier : 39 € pour 1 an.

Nefer-IT
150, rue Lamarck, 75018 Paris
redaction@programmez.com
Tél. : 09 86 73 61 08

Toute reproduction intégrale ou partielle est interdite sans accord des auteurs et du directeur de la publication. © Nefer-IT / Programmez!, septembre 2026.

Abonnez-vous à programmez!

Formules	1 an	2 ans	Etudiant	Numérique
Vous recevez le magazine chez vous	1 an 10 numéros (papier) 55 €	2 ans 20 numéros (papier) 90 €	1 an 10 numéros (papier) 45 €	1 an 10 numéros (format PDF) 39 €
Abonnements + accès aux archives	80 € (1 an + archives)	120 € (2 ans + archives)		64 € (1 an + archives)

Tarifs France métropolitaine.
Tarif PDF : valable partout dans le monde

L'abonnement comprend : les numéros normaux et les hors-séries

L'abonnement **numérique** est disponible uniquement sur notre boutique : **www.programmez.com**

Oui, je m'abonne

ABONNEMENT à retourner avec votre règlement à :
PROGRAMMEZ, Service Abonnements
57 Rue de Gisors, 95300 Pontoise

☐ Abonnement 1 an :	55 €
☐ Abonnement 2 ans :	90 €
☐ Abonnement 1 an Etudiant : (Joindre une photocopie de la carte d'étudiant)	45 €

Option : accès aux archives

☐ Pour abonnement 1 an	25 €
☐ Pour abonnement 2 ans	30 €

☐ Mme ☐ M. Entreprise : Fonction :

Prénom : Nom :

Adresse :

Code postal : Ville :

Adresse email indispensable pour la gestion de votre abonnement

E-mail : @

☐ Je joins mon règlement par chèque à l'ordre de Programmez !

☐ Je souhaite régler à réception de facture

* Tarifs France métropolitaine

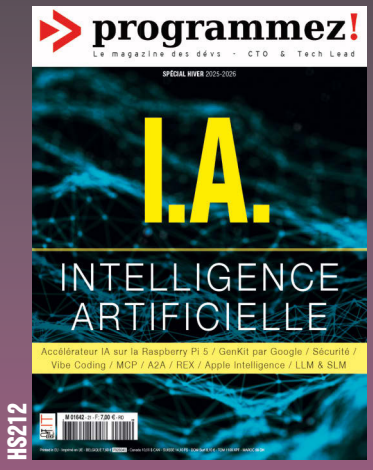
Offres pouvant s'arrêter à tout moment, sans préavis **PROG 277**

Boutique du Magazine PROGRAMMEZ!

Les anciens numéros de programmez!

Le magazine des dév - CTO - Tech Lead

LES AUTRES NUMÉROS SONT DISPONIBLES UNIQUEMENT EN PDF SUR PROGRAMMEZ.COM



HS212



HS22



272



273



274



275



276



HS23

Tarif unitaire 8 € (7€ + 1€) (frais postaux inclus)

☐ *HS21 - I.A*	: __ ex	☐ *274*	: __ ex
☐ *HS22 - 50 ans d'Apple*	: __ ex	☐ *275*	: __ ex
☐ *272*	: __ ex	☐ *276*	: __ ex
☐ *273*	: __ ex	☐ *HS23*	: __ ex

soit ___ exemplaires x 8 € = ____ € .....

Commande à envoyer à :
Programmez!
57 rue de Gisors
95300 Pontoise

soit au **TOTAL** = ____ €

☐ M. ☐ Mme Entreprise : Fonction :

Prénom : Nom :

Adresse :

Code postal : Ville :

Règlement par chèque à l'ordre de Programmez ! I Disponible sur www.programmez.com

PROG 277

Offres pouvant s'arrêter à tout moment, sans préavis

Osons la cybersécurité stratégique

Fabrice Bru
Président des Assises de la cybersécurité



Les responsables de la sécurité des systèmes d'information (RSSI) ont changé de dimension. Selon le baromètre du CESIN, 45 % d'entre eux évoluent vers des fonctions plus stratégiques et 36 % occupent des rôles transversaux. En réalité, ils sont déjà devenus des acteurs clés de la gouvernance des organisations. La cybersécurité figure désormais parmi les principaux risques des entreprises : 92 % des répondants la placent dans leur top 5, et autant de RSSI interviennent au moins une fois par an devant le COMEX.

Cette évolution se renforce avec l'arrivée de technologies comme l'intelligence artificielle ou le chiffrement post-quantique. Ces sujets occupent une place croissante dans les cartographies des risques et les échanges avec les instances dirigeantes. Le RSSI n'est plus seulement un expert technique : il participe aux choix stratégiques qui conditionnent la résilience et la compétitivité de l'entreprise.

Pour renforcer encore leur impact, plusieurs leviers doivent être actionnés. Le premier est humain. Si l'intelligence artificielle représente une formidable opportunité, elle ne remplacera jamais le jugement humain dans les décisions stratégiques ou les situations complexes. Elle constitue un outil supplémentaire pour aller plus loin, mais la responsabilité et la capacité d'arbitrage demeurent humaines.

Le deuxième levier est l'innovation. La cybersécurité ne peut plus être considérée comme une contrainte ajoutée après coup. Elle doit être intégrée dès la conception des projets. Aucun projet ayant un volet numérique ne peut être mené sans prendre en compte la cybersécurité. C'est à cette condition que l'innovation pourra créer durablement de la valeur et de la confiance.

Le troisième levier est européen. Les bouleversements géopolitiques récents ont profondément modifié notre rapport à la confiance et à la dépendance technologique. Dans ce contexte, l'Europe dispose d'atouts considérables. Avec un marché de 445 millions d'habitants, elle représente une puissance économique incontournable. Aucun acteur mondial ne peut s'affranchir d'un tel marché.

C'est tout le sens du message porté par le thème « Osons ! ». L'Europe bénéficie d'une formation académique de grande qualité, d'expertises reconnues et d'infrastructures énergétiques largement décarbonées. Les défis sont nombreux, mais nous disposons de tous les ingrédients nécessaires pour réussir et faire émerger nos propres champions.

La France possède également une force singulière : sa communauté cyber. Les échanges y sont fluides, les acteurs publics et privés collaborent étroitement, et des organisations comme le CLUSIF ou le CESIN contribuent à faire vivre cet écosystème. Les Assises de la cybersécurité en sont l'illustration.

Alors oui, osons. Osons entreprendre, investir, innover et prendre pleinement conscience de nos atouts. Osons surtout reconquérir une autonomie stratégique qui est devenue une condition essentielle de notre souveraineté numérique et de notre avenir collectif.



événements programmez! à venir

Meetup 60

29 septembre

Thème : Les promesses du Quantique : Introduction au calcul quantique et ses applications les plus prometteuses par Mariem Bahri (Bull)
Ce meetup se tiendra dans les locaux de Zeniak Paris : 53 Rue de Châteaudun, 75009 Paris

Acceuil à 18h30. Début vers 19h

DEVCON #28 sécurité, post-quantique et hacking édition 2027

10 décembre

Le magazine Programmez! organise sa prochaine conférence DevCon #28 le 10 décembre au campus ESGI Paris. 5h de conférence, 5 à 6 sessions.
Les thèmes de la journée :
- Sécurité, AppSec, DevSecOps
- OWASP
- Hacking, hacking hardware
- Post-quantique
- IA & sécurité
- sécurité du code, Kubernetes, conteneurs
- comprendre une attaque et savoir la contrer
- protéger la supply chain developer (PyPi, NPM, GitHub, etc.)

CFP ouvert jusqu'au 2 octobre :
https://forms.gle/TKfLBXYHX5ynCD7p9

À VENIR

- Salons Tech Show Paris : *les 18 et 19 novembre*

septembre 2026

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
	1	2	3	4	5	6
				JUG Summer Camp / La Rochelle Meetup virtuel Programmez !		
7	8	9	10	11	12	13
			Nantes Craft			
14	15	16	17	18	19	20
			API Platform Conference / Lille dotAI / Paris	dotJS / Paris		
21	22	23	24	25	26	27
	Agile en Seine / Paris Salon Data / Nantes		Paris Web OWASP AppSec Days / Paris PlatformCon Paris			
28	29	30				
Tech Summit on AI & Robotics / Paris	Meetup Programmez !					

octobre 2026

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
			1	2	3	4
			Volcamp / Clermont-Ferrand WAX / Marseille	DevFest / Perros-Guirec		
5	6	7	8	9	10	11
		Devoxx / Anvers Les Assises de la Cybersécurité / Monaco Forum PHP / Marne la Vallée				
12	13	14	15	16	17	18
Dev with AI / Paris						
19	20	21	22	23	24	25
26	27	28	29	30	31	
			Agiles Tour / Nantes PyCon / Biarritz	Cloud Nord / Lille		

Merci à Aurélie Vache pour la liste 2026, consultable sur son GitHub :
https://developers.events/#/2026/calendar

PROCHAINS NUMÉROS

PROGRAMMEZ! N°278
Disponible à partir du
20 novembre 2026

HORS SÉRIE N°24
Disponible à partir du
25 septembre 2026

Boutique du Magazine TECHNOSAURES



1
7,66 €



2
7,66 €



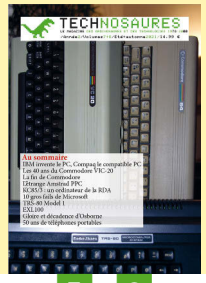
4
Edition Deluxe / Spécial Apple Lisa **15 €**



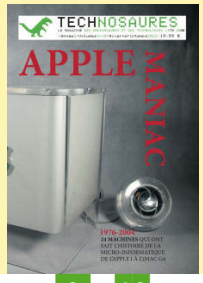
5
7,66 €



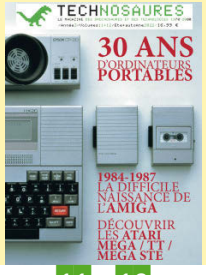
6
7,66 €



7 + 8
16,99 €



9 + 10
Spécial Applemaniac
33 €



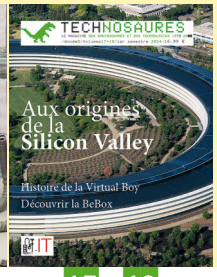
11 + 12
18,99 €



13 + 14
18,99 €



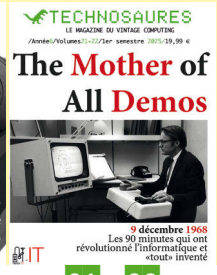
15 + 16
40 ans du Macintosh
18,99 €



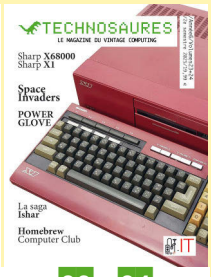
17 + 18
18,99 €



19 + 20
N° spécial 55 consoles qui ont fait l'histoire du jeu vidéo **29,99 €**



21 + 22
24 €



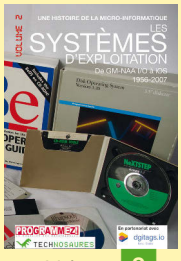
23 + 24
24 €



25 + 26
34,99 €

(frais postaux inclus)

HISTOIRE DE LA MICRO-INFORMATIQUE



Volume 2 *: une histoire des OS*
16 €



Volume 3 *: 90 nouvelles machines*
16 €



Volume 4 *: Micro-informatique française*
36 €



Volume 5 *: Arpanet*
34,99 €

1 ☐ 7,66 € x |__| ex = |__|__|__|__| €
2 ☐ 7,66 € x |__| ex = |__|__|__|__| €
4 ☐ 15 € x |__| ex = |__|__|__|__| €
5 ☐ 7,66 € x |__| ex = |__|__|__|__| €
6 ☐ 7,66 € x |__| ex = |__|__|__|__| €
7+8 ☐ 16,99 € x |__| ex = |__|__|__|__| €
9+10 ☐ 33 € x |__| ex = |__|__|__|__| €
11+12 ☐ 18,99 € x |__| ex = |__|__|__|__| €

13+14 ☐ 18,99 € x |__| ex = |__|__|__|__| €
15+16 ☐ 18,99 € x |__| ex = |__|__|__|__| €
17+18 ☐ 18,99 € x |__| ex = |__|__|__|__| €
19+20 ☐ 29,99 € x |__| ex = |__|__|__|__| €
21+22 ☐ 24 € x |__| ex = |__|__|__|__| €
23+24 ☐ 24 € x |__| ex = |__|__|__|__| €
25+26 ☐ 34,99 € x |__| ex = |__|__|__|__| €

Offre Spéciale : trilogie Apple
☐ 49 € x |__| ex = |__|__|__|__| €

HISTOIRE DE LA MICRO-INFORMATIQUE
V2 ☐ 16 € x |__| ex = |__|__|__|__| €
V3 ☐ 16 € x |__| ex = |__|__|__|__| €
V4 ☐ 36 € x |__| ex = |__|__|__|__| €
V5 ☐ 34,99 € x |__| ex = |__|__|__|__| €

total colonne = |__|__|__|__| € .... + ...|__|__|__|__| € ..... + ...|__|__|__|__| € = |__|__|__|__| €

Commande à envoyer à :
Programmez!
57 rue de Gisors
95300 Pontoise

soit au **TOTAL**

☐ M. ☐ Mme Entreprise : |__|__|__|__|__|__|__|__| Fonction : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Prénom : |__|__|__|__|__|__|__|__|__|__|__|__|__| Nom : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Adresse : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Code postal : |__|__|__|__|__| Ville : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Règlement par chèque à l'ordre de Programmez ! I Disponible sur www.programmez.com

PROG 277

Offres pouvant s'arrêter à tout moment, sans préavis

La place mentale de l'IA pour aujourd'hui et demain



Marc Kruzik

Marc Kruzik est développeur informatique, organisateur d'événements et écrivain. Inspiré par Voltaire, il anime de nombreuses discussions sur l'avenir de la technologie et de la société.

"Qu'est-ce qu'il y a demain ?" Le conférencier attend une réponse claire pour continuer sa présentation. Demain ? Le public ne voit pas de réponse évidente. A-t-on bien compris la question ? Demain ou deux mains ? Le public est perplexe. La trentaine de développeurs présents est tout bonnement incapable de trouver la réponse. "Qu'y a-t-il demain ?" s'impatiente le conférencier. Quelle réponse est attendue ?

En cette soirée chez Microsoft du 10 juin 2026, nous sommes tous concentrés sur l'IA. L'événement Microsoft Build a eu lieu la semaine dernière, et nos communautés ont été invitées dans les locaux de Microsoft France, à Issy-les-Moulineaux, pour faire une rétrospective des principales annonces. Je représente ici le Meetup C# .NET Paris, consacré au langage de programmation C#. L'organisation de la soirée a été principalement gérée par les co-organisateurs du meetup Azug FR, qui est la communauté des utilisateurs francophones de Microsoft Azure, le cloud de Microsoft. Les meetups Global AI Paris et Global AI Community Virtuelle Francophone complètent le rassemblement de nos communautés. C'est très sympathique de voir autant de responsables de meetup. Quatre communautés réunies à la fois, je n'avais jamais vu ça ! J'ai plaisanté plusieurs fois sur la nécessité de devoir un jour rassembler nos meetups comme si nous étions une équipe de super-héros, simplement pour avoir suffisamment de monde à un événement.



Aujourd'hui, je suis surtout concentré sur le contenu des annonces. "Il ne va pas y avoir beaucoup de C#" raillait un de mes collègues. C'est vrai que l'IA prend toute la place dans les actualités de Microsoft. Mais je reste malgré tout dans le déni. Le récapitulatif fait au début de la soirée a bien présenté les nouveautés IA, les nouveautés cloud IA, les nouveautés du matériel physique IA. Et la présentation s'est arrêtée là. Bon, j'ai dû prendre la parole pour ajouter un récapitulatif des annonces en lien avec le C# : le langage s'enrichit des "types unions", et un nouvel outil "dotnetup" simplifie les installations automatiques. C'est tout. Sur le site de Microsoft, il n'y a que six conférences consacrées au C# et son système .NET, et quatre d'entre elles sont tournées vers l'IA. Nous sommes véritablement dans l'ère de l'IA.

Les organisateurs présents sont des utilisateurs d'IA, réalisant des prestations liées à l'IA, heureusement, ce ne sont pas tous des vendeurs d'IA. Ces gens-là ont tendance à développer une forme d'addiction à différentes martingales, persuadés que le temps qu'ils perdent aujourd'hui avec l'IA finira par être compensé demain. À chaque remarque sur l'inefficacité de l'IA, ce type d'interlocuteur se braque, et explique toujours que c'est la faute de l'utilisateur et que la prochaine IA compensera les défauts observés. "L'IA d'aujourd'hui est seulement une mauvaise version de l'IA de demain". Toutefois, un conférencier finit par nonchalamment révéler que l'IA dont il parle l'oblige à utiliser chez lui un deuxième ordinateur. Qui travaille avec deux ordinateurs allumés en même temps pour le même sujet ?

Lorsqu'on écoute plus attentivement les propos, l'ensemble est hors-sol. La démonstration IA cafouille, l'Intelligence a changé de comportement depuis la préparation de sa présentation. À moitié réussie, à moitié échouée, nous suivons la présentation avec un mélange d'admiration et d'horreur. Qu'en est-il de cet outil censé être merveilleux ? Qu'en est-il de cette IA censée régler tous nos problèmes ? Quel est l'avenir de notre métier ? Quel est notre avenir ? "Qu'est-ce qu'il y a demain ?" Décidément, la présentation ne peut pas continuer sans répondre à cette question. Excédé, le conférencier finit par donner la réponse : "demain, c'est la coupe du monde de foot". Ah ! La coupe du monde de foot ! Oui, c'est vrai, c'est demain. Personne n'a été capable d'y penser. Aucun développeur présent n'avait encore de place mentale pour un sujet comme le foot. L'IA prend toute la place dans notre cerveau. Face aux annonces de Microsoft, nous oscillons entre joie et appréhension. Peut-on utiliser ces IA ? Doit-on utiliser ces IA ? Va-t-on être remplacé par ces IA ? Il est certain que l'IA ne remplacera jamais le ballon physique, à défaut d'occuper toute la place mentale disponible dans nos têtes. On en oublie même la Coupe du monde de foot ! Combien d'autres sujets sont dévorés dans nos têtes par l'IA ?

amazon future engineer

Du code à l'IA, un programme qui évolue avec son époque

Lancé en 2020, Amazon Future Engineer sensibilise les jeunes au numérique dans neuf pays. En France, le programme franchit un nouveau cap en intégrant pleinement l'intelligence artificielle à ses parcours pédagogiques. Objectif : préparer les talents de demain aux réalités du marché du travail.

Un constat, une mission

En France, plus d'un tiers des jeunes de 15 ans peinent à se projeter dans un métier (étude Gallup, 2024). Parallèlement, l'intelligence artificielle devrait créer 170 millions de nouveaux emplois dans le monde d'ici 2030 (World Economic Forum, *Future of Jobs Report 2025*). L'écart entre les aspirations des jeunes et les opportunités réelles du marché est un défi éducatif majeur.

C'est pour répondre à cet enjeu qu'Amazon a lancé Future Engineer en 2020, en partenariat avec des acteurs de référence comme TRALALERE et France-ioi. L'objectif est d'ouvrir le champ des possibles pour les jeunes en leur donnant les clés pour comprendre un monde où le numérique est omniprésent, de la santé aux transports, en passant par l'éducation.

« *Depuis son lancement, Amazon Future Engineer poursuit un objectif clair : ouvrir le champ des possibles pour les jeunes dans le numérique* » explique Alice Berton, responsable du programme Amazon Future Engineer en France. « *Nous voulons leur donner les clés pour comprendre un monde où le numérique façonne de plus en plus les métiers et les opportunités professionnelles* ».

Le programme est entièrement gratuit, avec une attention particulière portée aux jeunes issus de milieux défavorisés, aux zones d'éducation prioritaire et aux jeunes filles. Depuis 2020, plus de 2 millions de jeunes en ont bénéficié en France.

ESSAYEZ : DÉCOUVRIR LE CODE AVEC BLOCKLY

Parmi les outils accessibles gratuitement sur futureengineer.fr, un jeu éducatif permet de découvrir les bases du code directement depuis son navigateur. Le principe : manipuler des commandes et des fonctions pour résoudre des défis, avec une prise en main rapide et une progression par niveaux (débutant, intermédiaire, expert). À chaque exercice, une documentation et des indices accompagnent l'utilisateur. Points et badges récompensent la progression.

→ Essayer : app.citizencode.net/demo/backend-driven

CitizenCode Python : apprendre le code en s'amusant

Au cœur du dispositif se trouve CitizenCode Python, une plateforme ludo-éducative développée par TRALALERE et France-ioi. Son principe : une progression gamifiée qui emmène les jeunes de la programmation visuelle par blocs (Blockly) jusqu'au langage Python, à travers des défis, des puzzles et des exercices adaptés à chaque niveau.

Les résultats parlent d'eux-mêmes :

- Plus de 200 000 utilisateurs actifs
- 8 000 comptes enseignants
- 3 saisons, plus de 100 exercices
- 5 open badges pour valider les acquis
- 1 million d'activités complétées
- Accessible dès 12 ans

Depuis octobre 2025, CitizenCode est également disponible en application mobile sur Android et iOS, une évolution importante pour toucher les jeunes là où ils sont, car comme le confirme Hugo Derri (TRALALERE), beaucoup utilisent aujourd'hui majoritairement leur smartphone.

2025-2026 : le virage IA

Quand le programme a été lancé en 2020, l'IA générative n'existait pas encore dans le quotidien du grand public. Depuis l'arrivée de ChatGPT et de ses successeurs, la donne a changé : l'IA transforme les métiers, les formations et la société. Les jeunes l'utilisent déjà, mais ne sont pas nécessairement armés pour la comprendre et s'en servir de manière responsable.

Amazon Future Engineer a naturellement intégré cette dimension. « *L'IA transforme rapidement notre société et notre économie. Il est essentiel que les jeunes comprennent ses fondamentaux et ses implications* » souligne Alice Berton. « *Les nouveaux modules conçus avec France-ioi et TRALALERE permettent justement cela : une approche pédagogique progressive, adaptée à chaque niveau scolaire, pour rendre l'IA accessible et compréhensible par toutes et tous* ».

Concrètement, la rentrée 2026 s'articule autour de trois nouveautés majeures :

- **L'IA-ckathon (TRALALERE)** : un hackathon pédagogique intégré à CitizenCode, où les élèves utilisent l'IA générative pour concevoir des projets à impact local en quatre séances (voir article dédié).
- **Le module IA (France-ioi)** : un parcours complet combinant jeu interactif, défis sur des modèles d'IA réels et activités débranchées, avec l'objectif d'initier 100 000 élèves aux fondamentaux de l'IA d'ici fin 2026.
- **"Explore les Métiers de la Tech" (France-ioi)** : une série de 10 vidéos ludo-éducatives qui plonge les jeunes dans le quotidien concret des professionnels du numérique pour leur permettre de se projeter dans ces carrières (voir article dédié).

Ces trois piliers marquent l'entrée d'Amazon Future Engineer dans une nouvelle phase, résolument tournée vers l'IA et l'employabilité des jeunes.

Site officiel : amazonfutureengineer.fr

REPÈRES CHRONOLOGIQUES

Année	Étape clé
2020	Lancement d'Amazon Future Engineer en France
2022	Lancement de CitizenCode Python (TRALALERE x France-ioi)
2023	Le programme dépasse les 500 000 jeunes sensibilisés
2024	CitizenCode Python récompensé aux European Digital Skills Awards
2024	Le programme franchit le cap d'1,5 million de jeunes
2025	Lancement du module IA (France-ioi) et première expérimentation de l'IA-ckathon (TRALALERE)
2026	Intégration de l'IA-ckathon dans CitizenCode, lancement de la série "Explore les Métiers de la Tech", objectif 100 000 élèves initiés à l'IA



Hugo Derri

TRALALERE x Amazon Future Engineer : l'IA entre en classe

Partenaire d'Amazon Future Engineer depuis plus de cinq ans, TRALALERE conçoit des ressources numériques éducatives à destination des jeunes et des enseignants. « *Nous créons des contenus pour sensibiliser les jeunes au numérique mais aussi pour leur permettre de développer leurs compétences à travers l'apprentissage de la programmation ou la maîtrise des technologies IA* » explique Hugo Derri, responsable du programme CitizenCode chez TRALALERE. Au cœur de cette collaboration : CitizenCode, une plateforme gratuite qui propose des parcours progressifs pour découvrir la programmation, du langage par blocs jusqu'à Python. En 2025, plus de 100 000 jeunes, parents et enseignants l'ont utilisée.

À la rentrée 2026, CitizenCode franchit une nouvelle étape avec le lancement d'un module dédié à l'intelligence artificielle : l'IAkathon. L'objectif : permettre aux élèves d'apprendre l'IA par la pratique, dans le cadre scolaire, et leur montrer comment elle peut servir la créativité et la résolution de problèmes concrets.

Un hackathon IA en classe, en quatre séances

Le concept est né d'une première expérimentation menée en 2025 auprès de plus de 1 000 élèves. « *Notre objectif était de changer la perception de l'école sur l'IA générative et la montrer comme un outil pour accompagner la créativité. Les élèves devaient imaginer des projets à impact sur des sujets de société* » raconte Hugo Derri.

Fort de ce succès, TRALALERE a développé un module complet, hébergé sur CitizenCode, qui permet à tout enseignant d'animer un véritable hackathon IA sur quatre séances, sans expertise technique préalable. Le principe : les élèves forment des équipes, définissent un projet à impact local et mobilisent l'IA générative à chaque étape — exploration, idéation, prototypage, pitch.

« *Ce hackathon mise sur une approche projet. L'IA est là pour aider chaque équipe à développer son prototype* » précise Hugo Derri. « *Créer un projet en quatre heures est très difficile. L'objectif est alors de permettre d'accompagner un usage de l'IA libre, en classe, dans un contexte positif, tout en demandant aux élèves d'avoir une approche critique de leurs usages* ».





Apprendre à utiliser l'IA, mais aussi à la questionner

Le module ne se limite pas à l'utilisation de l'IA : il intègre un journal de bord où les élèves documentent leurs requêtes, partagent leurs doutes et questionnent le rôle de l'IA dans leur projet. L'idée : former à l'esprit critique autant qu'à l'outil. « *L'IA peut être d'une grande aide quand elle est bien utilisée, avec les bonnes questions et les bonnes requêtes* » souligne Hugo Derri.

Côté technique, la plateforme reste volontairement agnostique. « Nous ne privilégions aucun LLM en particulier. Nous apportons une plateforme et les assets pédagogiques nécessaires. L'enseignant garde la maîtrise sur les outils, IA ou non, qu'ils souhaitent mettre à disposition de ses élèves pour un tel projet », précise-t-il. Ce choix garantit une adaptabilité maximale, quel que soit l'environnement scolaire.

Et ensuite ?

Au-delà du module IA, CitizenCode continue d'évoluer. L'application mobile (Android et iOS), lancée en octobre 2025, a déjà permis d'élargir l'accès à un public qui utilise majoritairement le smartphone. Prochaine étape : une ouverture vers la découverte des métiers du numérique, en complémentarité avec la série vidéo "Explore les Métiers de la Tech" développée par France IOI dans le cadre d'Amazon Future Engineer (voir article dédié).

« *Pour le hackathon, il y aura un engagement important de l'enseignant. Nous espérons toucher 10 000 jeunes dans les prochains mois* » conclut Hugo Derri.

France-ioi : de l'algorithmique à l'IA, former par la pratique



Mathias Hiron

Partenaire d'Amazon Future Engineer depuis les origines du programme, France-ioi développe des outils gratuits pour cultiver la pensée informatique, l'algorithmique et la programmation chez les jeunes. En 2026, l'association élargit son champ d'action à l'intelligence artificielle et à la découverte des métiers du numérique.

Plus de 20 ans d'expertise au service de l'apprentissage

Créée en 2004, France-ioi conçoit des parcours pédagogiques interactifs et organise des concours nationaux qui touchent chaque année un million d'élèves du CM1 à la Terminale. « *Nous développons beaucoup de contenus, de modules et d'outils. En plus de l'équipe interne, une trentaine de bénévoles participent à leur conception* » explique Mathias Hiron, cofondateur et président de France-ioi. Parmi les rendez-vous annuels :

- **Castor Informatique** : faire découvrir l'informatique et les sciences du numérique. Plus de 650 000 participants issus de 3 809 établissements ont participé à l'édition 2025. L'édition 2026 se tiendra du 8 novembre au 19 décembre.
- **Algoréa** : encourager les jeunes à progresser à leur rythme en algorithmique et en programmation, du CM1 à la terminale. Les premières phases s'appuient sur Scratch, Blockly et Python ; la phase finale est ouverte aux langages C, C++ et Java.

Des parcours ancrés dans les programmes scolaires

Dès la création d'Amazon Future Engineer, France-ioi a été sollicitée pour concevoir des parcours pratiques alignés sur le programme de Sciences Numériques et Technologie (SNT) de seconde. « *C'était peu de temps après l'apparition de SNT. Nous avons réfléchi ensemble aux créations possibles. Il y avait une matière très riche et des opportunités pour créer des parcours, des outils d'apprentissage par la pratique, et faciliter la manipulation des concepts* » retrace Mathias Hiron. Résultat : plusieurs parcours thématiques interactifs, accessibles gratuitement :

- **Photographie numérique** : comprendre ce qu'est une image numérique via un simulateur complet (sensibilité, focale, balance des couleurs…), sans nécessiter de matériel en classe
- **Données structurées** : découvrir la notion de données et les manipuler, d'abord dans un tableur, puis avec des langages de programmation
- **Cartographie et géolocalisation** : latitude, longitude, GPS, lecture de cartes…

La période COVID a accéléré le développement de nouveaux contenus pour permettre aux élèves de travailler à distance. « *Nous avions dans les cartons un projet sur les objets connectés et comment les programmer. Nous avons créé trois parcours avec 24 activités* » ajoute Mathias Hiron. C'est ainsi qu'est né le module Quick-Pi, qui permet de programmer un objet connecté directement depuis son navigateur.

Depuis 2020, plus de 300 000 élèves ont suivi ces parcours SNT.

Le virage IA : comprendre, manipuler, créer

En 2025, France-ioi a lancé un nouveau module dédié à l'intelligence artificielle dans le cadre d'Amazon Future Engineer, avec l'objectif d'initier 100 000 élèves à ses fondamentaux d'ici fin 2026. L'approche se veut résolument concrète et multimodale.

« *Nous avons réfléchi à la manière d'aborder l'IA. Nous proposons une approche élargie : comment fonctionne l'IA, comment manipuler un modèle, comment l'utiliser* » précise Mathias Hiron.

Concrètement, les élèves sont confrontés à des défis pratiques :

- **Créer des prompts** : à partir d'une image de référence, les élèves doivent formuler et affiner des descriptions pour qu'une IA générative reproduise l'image. Un exercice qui montre, de manière très concrète, la nécessité de bien formuler une demande et d'itérer pour obtenir un résultat satisfaisant.
- **Reproduire une application** : les élèves utilisent l'IA comme outil de développement pour recréer une application existante, en décrivant ses fonctionnalités étape par étape — une initiation au "vibe coding" qui leur fait expérimenter à la fois la puissance et les limites de la génération de code par IA.

Le module propose trois niveaux de difficulté et plusieurs langages (Scratch, Blockly, Python), sans prérequis technique. Les activités sont conçues pour être animées par les enseignants quelle que soit leur discipline (Français, Mathématiques, Langues, Physique-Chimie).

EXPLORE LES MÉTIERS DE LA TECH : se projeter dans les carrières numériques

Comment demander aux jeunes de s'orienter vers les métiers de la tech quand la plupart des adultes peinent eux-mêmes à les définir ? C'est le constat qui a motivé la création de cette série de 10 vidéos ludo-éducatives, conçue par France-ioi avec le soutien d'Amazon Future Engineer.

Le format : pas d'interview classique ni de fiche métier. Chaque épisode plonge le jeune au cœur d'une mission fictive mais réaliste, aux côtés d'un professionnel. « *Faire comprendre le métier par la pratique, comme expliquer comment migrer une application d'un serveur local vers le cloud* » illustre Mathias Hiron. Le tout guidé par une voix off dynamique, dans l'esprit d'un *C'est pas Sorcier* du numérique.

En complément de chaque vidéo, une activité interactive permet à l'élève de tester le métier par lui-même. À la fin de l'épisode, le professionnel prend la parole pour partager son parcours et ses conseils. **Les premiers épisodes disponibles :**

- **Épisode 1** — Sadou, architecte infrastructure et réseau. Mission : migrer les applications d'une entreprise vers le cloud.
- **Épisode 2** — Alexia, product manager. Mission : améliorer le système de recommandation d'une application de streaming musical.

L'objectif : montrer que la tech, ce n'est pas uniquement coder. C'est aussi de la stratégie, du marketing, de la logistique. En mettant en avant des profils divers — femmes, hommes, parcours variés — la série veut permettre à chaque jeune de se dire : « Moi aussi, je peux le faire ».

Les épisodes sont diffusés progressivement depuis juin 2026, gratuitement sur parcours.algorea.org.

INFORMATIQUE QUANTIQUE

Le quantique reste une entité technologique mal comprise. Le sujet est d'une grande complexité et il diffère totalement de notre informatique actuelle. Il faut changer notre manière de penser. Le quantique est un univers fascinant et la R&D est particulièrement active.

Notre dernier grand dossier quantique remonte à l'été 2025. En 13 mois, plusieurs tendances se dessinent :

1. L'approche photonique devient une référence pour plusieurs startups et constructeurs d'ordinateur quantique

2. L'hybridation est un objectif. Pas pour nos micro-ordinateurs mais pour le HPC et les supercalculateurs. Il s'agit de « mixer » une approche classique (X64, Arm, Risc-V) avec des composants quantique pour exploiter les deux mondes. Dès 2028 ou après 2030 ? Voir l'article de Samuel et de Luis Fernando

3. La tolérance aux fautes et aux erreurs : c'est une des quêtes clés de l'ordinateur quantique. Aujourd'hui, les erreurs et le bruit sont des sources d'instabilité quantique. Les constructeurs visent la correction d'erreurs d'ici 5 ans. Dans ce dossier, Romain Perrier nous parle de la correction d'erreurs quantiques.

En parallèle du hardware, il y a toute la stack logicielle à préparer, à concevoir. Il faut des langages, des frameworks, des compilateurs, des outils. Nous vous conseillons de lire attentivement la contribution de Mathieu Garcia sur les frameworks (et les émulateurs) et l'analyse de Simon Fried (Classiq).

Vous pouvez tester dès maintenant la programmation et le matériel quantique avec le service à la demande d'OVHcloud.

Si le quantique vous intéresse, Programmez! vous propose 2 conférences :

29 septembre – meetup #60

Les promesses du quantique dans le calcul, par Mariem Bahri. Dans les locaux de Zenika Paris. À partir de 18h30

25 février 2027 – DevCon #29

Informatique quantique édition 2027, au campus 42 Paris. À partir de 13h30

La rédaction

Découvrir le fonctionnement des ordinateurs quantiques photoniques



Martin Bombardelli

Actuellement en thèse au LIMOS en collaboration avec la société CGI dans le cadre d'un contrat ANRTEmail : martin.bombardelli@cgi.com



Philippe Lacomme

Professeur à l'ISIMA où il enseigne l'optimisation et l'informatique quantique.Email : philippe.lacomme@isima.fr



Bogdan Vulpescu

Ingénieur de recherche au CNRS au sein du LPCA : (UMR 6533). L'une de ses missions est d'assurer une veille autour des technologies d'information quantiques.Email : bogdan.vulpescu@clermont.in2p3.fr

De quoi a-t-on besoin pour avoir des ordinateurs quantiques ?

Il existe une multitude de façons de construire des ordinateurs quantiques : circuits oscillants supraconducteurs, photons, atomes ionisés, atomes neutres, etc.. La plupart de ces machines implémentent le modèle à portes des ordinateurs quantiques qui vous ont été présentées dans de précédents numéros de PROGRAMMEZ ! Mais comment tous ces systèmes physiques peuvent être décrits sous un même modèle ? Tentons de découvrir cela avec les ordinateurs quantiques photoniques.

Les ordinateurs quantiques de manière générale tirent leurs avantages de trois phénomènes qui régissent le comportement des systèmes physiques à petite échelle. Tous ces phénomènes sont liés à la dualité onde-corpuscule. À chaque particule est associée une onde qui décrit ses propriétés.

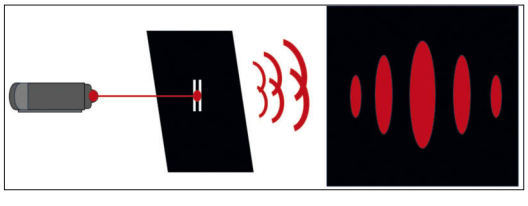
- La superposition : c'est la propriété des particules à être dans **deux états à la fois**. C'est un petit peu comme si l'onde associée au photon se séparait en deux parties différentes.
- L'intrication : c'est un phénomène étrange qui peut apparaître lorsque deux objets quantiques se rencontrent et interagissent. Ces deux objets ne peuvent alors plus être considérés séparément. On peut s'imaginer que les deux ondes se mélangent pour ne former qu'une seule grande onde qui décrit les deux objets.
- Les interférences : deux ondes qui se rencontrent peuvent s'amplifier ou s'annuler et ainsi des particules peuvent ne pas être présentes à un endroit où on s'attendrait à les trouver.

Pour avoir un ordinateur quantique qui fonctionne, on a nécessairement besoin de pouvoir contrôler ces trois phénomènes !

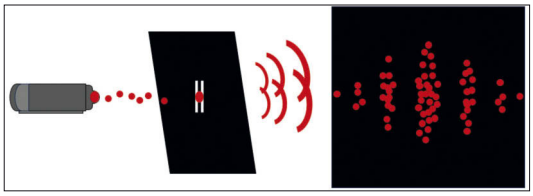
Interférence, superposition et intrication pour des photons

L'interférence avec un seul photon.

L'une des expériences les plus célèbres liées à la physique quantique est l'expérience des fentes de Young, lorsque l'on envoie un laser sur deux fentes très rapprochées, et que l'on observe un écran placé derrière les fentes. En considérant la lumière uniquement comme une onde, on peut imaginer ce qui apparaîtrait sur cet écran. L'image la plus simple est de visualiser deux creux de vagues ou deux hauts de vagues qui se rencontrent pour former un grand creux ou une seule grande vague. C'est une figure d'interférence !



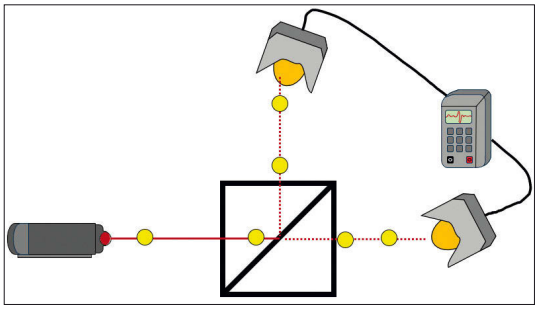
En revanche, il est plus difficile de s'imaginer cela lorsque l'on répète l'expérience en envoyant des particules une par une. Ce qui se produit est pourtant l'apparition d'une figure d'interférence similaire.

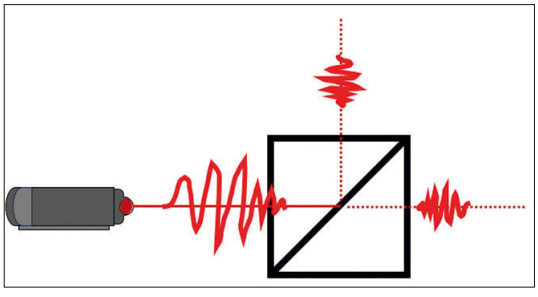


L'explication la plus naturelle est de considérer l'onde associée à chaque particule. Cette onde passe par les deux fentes et interfère avec elle-même entre les fentes et l'écran. Il est possible d'utiliser les interférences avec des photons et cela sera très utile pour fabriquer un ordinateur quantique.

Des photons en superposition.

Pour pouvoir décrire la superposition chez les photons, introduisons le *beam splitter* (un séparateur de faisceau). Ce composant est un miroir semi réfléchissant. Il laisse passer une partie de la lumière et en réfléchit une partie.





En envoyant des photons un par un et en plaçant des détecteurs au niveau des trajectoires réfléchies et transmises, on peut détecter les photons. Certains seront transmis et détectés, d'autres seront d'abord réfléchis et ensuite détectés.

Pour comprendre comment faire de la superposition, passons par la représentation qui utilise les ondes. En considérant un seul photon, une partie de son onde est transmise et une partie est réfléchie. Notre photon a à la fois traversé le miroir semi réfléchissant et a été réfléchi. On rappelle que l'onde est associée à un seul photon !

Pourtant, si on mettait des détecteurs des deux côtés dans le cas où on a un seul photon, on obtient un photon unique-

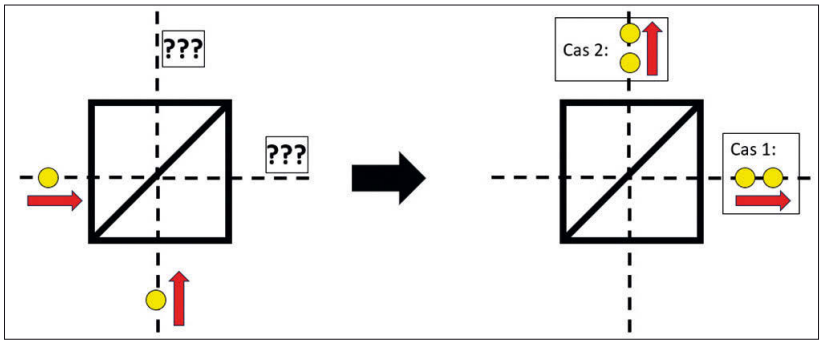
ment dans un seul détecteur. C'est le problème de la mesure. Lorsque l'on n'a pas encore mesuré l'onde, le photon est « aux deux endroits à la fois ». Si vous n'êtes pas convaincu par la représentation sous forme d'onde et que vous imaginez que le photon choisit « au hasard » s'il est réfléchi ou transmis, nous vous invitons à rechercher plus d'informations sur l'interféromètre de Mach-Zehnder.

L'interférence avec deux photons.

Intéressons-nous à ce qu'il se passe lorsque l'on envoie deux photons sur deux axes perpendiculaires d'un *beam splitter*. Plaçons-nous dans le cas d'un *beam splitter* ayant exactement les même chances de transmettre et de réfléchir les photons. En y réfléchissant, on pourrait facilement se convaincre que l'on a 4 cas de figure :

- Les deux photons sont **transmis** et on a 1 photon sur chaque sortie.
- Les deux photons sont **réfléchis** et on a 1 photon sur chaque sortie.
- Le photon de gauche est **transmis** et le photon du bas est **réfléchi**. On obtient deux photons à droite.
- Le photon de gauche est **réfléchi** et le photon du bas est **transmis**. On obtient deux photons en haut.

Mais expérimentalement, on obtient seulement deux des cas précédents. Les deux photons sont toujours ensemble soit en haut soit à droite. On n'a jamais un photon en haut **et** un photon à droite.



Dans cette situation, les photons interfèrent entre eux, sinon, on aurait obtenu les quatre cas cités précédemment. Cela aurait été le cas en décalant l'envoi des deux photons pour qu'ils passent l'un après l'autre.

L'intrication.

L'intrication ne peut pas concerner un seul photon, c'est une propriété qui lie deux ou plusieurs objets quantiques et, pour l'obtenir, il est nécessaire de les faire d'abord interagir entre eux. Un concept classique qui s'en rapproche est la corrélation. Mais il existe des différences bien réelles entre ces deux concepts.

Contrairement à d'autres méthodes pour construire des qubits, il est difficile de faire interagir les photons : deux faisceaux lasers envoyés l'un sur l'autre se traversent réciproquement sans se perturber dans leurs chemins. Pour obtenir deux photons intriqués il faut agir directement à leur source, c'est à dire sur le mécanisme de production de ces photons, par exemple avec une désexcitation en cascade d'un même atome. Ceci est possible en utilisant certains cristaux, qui, illuminés avec un laser de petite longueur d'onde (violet), génèrent pour chaque photon d'entrée deux photons en sortie d'une longueur d'onde double (rouge) et qui seront, en plus, fortement intriqués.

Pourtant, la situation précédemment illustrée, avec les deux photons envoyés sur les deux faces d'un *beam splitter*, présente un **certain** type d'intrication, appelée intrication de chemins. En mesurant les deux photons sortant d'une seule face du *beam splitter*, il nous est impossible de connaître par quelle face d'entrée chaque photon est arrivé.

D'accord, les photons présentent toutes les caractéristiques nécessaires mais où sont les ordinateurs quantiques ?

Maintenant, nous sommes convaincus que les photons possèdent toutes les caractéristiques quantiques nécessaires à la fabrication d'un ordinateur quantiques.

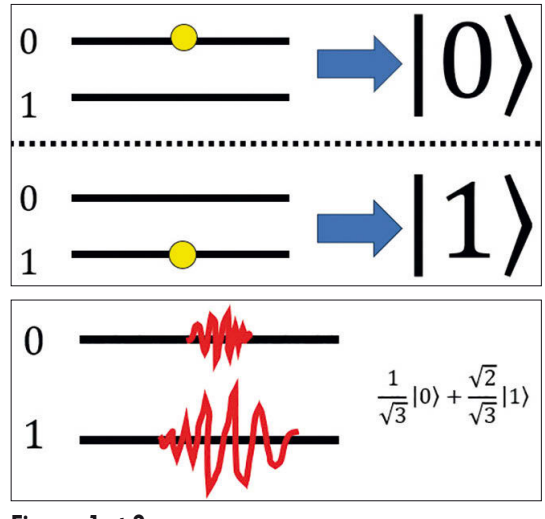
Les qubits

Comme vous vous en doutez, pour concevoir un ordinateur quantique on va devoir fabriquer des qubits. Les qubits sont l'unité de base de l'informatique quantique. Pour définir un qubit, on a besoin de deux états bien distincts de notre système quantique :

$$\alpha|0\rangle + \beta|1\rangle \quad \text{avec} \quad |\alpha|^2 + |\beta|^2 = 1$$

Avec les photons, il existe une multitude de façons de faire. L'une des façons de faire s'appelle le *dual rail encoding* et a été introduite par Isaac L. Chuang et Yoshihisa Yamamoto en 1995. Pour définir un qubit selon leur méthode, il suffit de considérer un photon dans deux chemins optiques différents. Un chemin optique peut être assimilé à une fibre optique. Une bonne représentation est de considérer deux fibres optiques l'une au-dessus de l'autre. Pour être plus précis appelons ces fibres optiques **modes**.

Numérotons ces modes 0 et 1. Pour définir un qubit on choisit de dire que la présence du photon dans le mode 0 (fibre du haut) correspond à l'état du qubit $|0\rangle$. Inversement, si notre photon est dans le mode 1, on définit cette situation comme $|1\rangle$.



Figures 1 et 2

Ici, on a choisi d'utiliser la localisation d'un photon pour définir notre qubit. Rien n'interdisait de choisir autre chose. Dans certains cas on peut même utiliser la polarisation des photons pour définir des qubits !
Un qubit en superposition peut être vu comme un qubit ayant un peu de son onde associée sur chacun des deux modes.
Figures 1 et 2

Les portes quantiques

Maintenant, comment peut-on réaliser les portes quantiques à l'aide des photons ? Prenons deux exemples parmi les portes les plus communes.

- la porte X
- la porte H

La porte X a pour effet d'échanger les amplitudes de probabilité associées aux états $|0\rangle$ et $|1\rangle$. Afin de réaliser cette porte, nous pouvons utiliser des miroirs pour échanger les deux modes.
Grâce, aux miroirs, il est possible d'échanger les deux morceaux d'ondes de place. On passe de $\alpha|0\rangle + \beta|1\rangle$ à $\beta|0\rangle + \alpha|1\rangle$. C'est bien l'action de la porte X !

Pour construire la porte H, on réutilise le *beam splitter*. Pour correspondre à nos deux rails horizontaux, on tourne sa représentation de 45°. Les deux lignes horizontales correspondent à deux modes qui définissent un qubit. Ainsi nous avons deux entrées sur la gauche du *beam splitter* et deux sorties à droite. **Figures 3 et 4**

Envoyer un photon dans le mode du haut (sur le schéma de droite) permet de le séparer en deux et de mettre notre photon en superposition. Si l'onde associée est parfaitement séparée en deux, cela correspond à la porte appliquée au qubit dans l'état. (Le cas possède une subtilité que nous ignorons ici.)

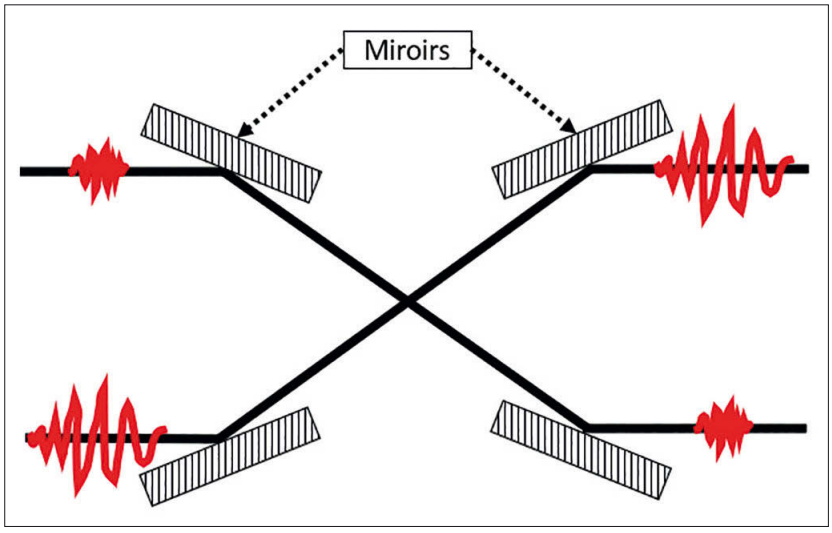
Certains beam splitters ne partagent pas équitablement l'onde associée au photon ce qui permet d'avoir une probabilité plus ou moins grande de donner 0 ou 1 en sortie du composant. C'est ce qui permet d'obtenir une superposition arbitraire $\alpha|0\rangle + \beta|1\rangle$.

Les circuits quantiques

Pour exécuter un circuit quantique de q qubits avec des photons, on utilise 2q modes (car chaque mode est représenté par deux qubits). Puis on utilise les beam splitters pour agir sur nos différents qubits et les faire interagir entre eux.

On retrouve sur la **figure 5** un exemple de circuit à deux qubits et donc quatre modes. Ce circuit utilise des *beam splitter* et des miroirs. Il ne réalise pas de calcul en particulier mais permet de modifier les chances de trouver des photons au bout du circuit notamment en utilisant des *beam splitters* qui donnent des superpositions arbitraires.

On peut aussi remarquer que les deux qubits interagissent entre eux car les différents modes se mélangent. Grâce à ces circuits, il est possible de réaliser l'ensemble des portes quantiques. Certaines portes quantiques, notamment celles agissant sur plusieurs qubits vont nécessiter d'ajouter des modes auxiliaires. Ces modes supplémentaires ne représentent pas de qubits ils servent uniquement d'outil pour réaliser les portes sur les qubits. La porte CNOT fait partie des exemples qui nécessitent l'utilisation de modes auxiliaires. Même si les ordinateurs quantiques photoniques peuvent réaliser, plus ou moins difficilement toutes les portes quantiques, ils n'ont pas besoin de passer par ce modèle pour pouvoir réaliser des calculs. En effet il existe d'autres manières de définir et d'utiliser des qubits avec des photons qui peuvent s'avérer avantageuses en fonction de la situation. Il est par exemple possible d'utiliser une autre propriété de la lumière : polarisation des photons.



Figures 3

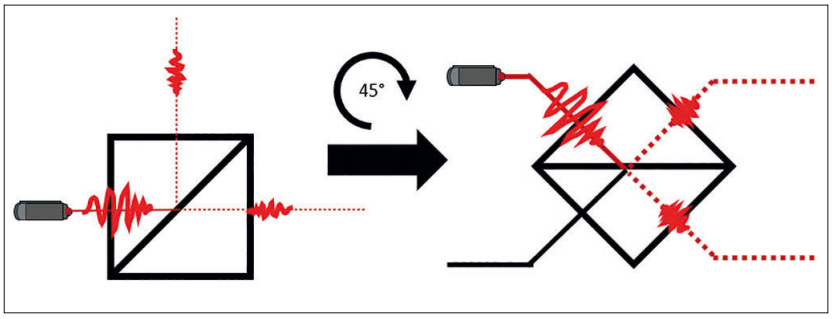


Figure 4

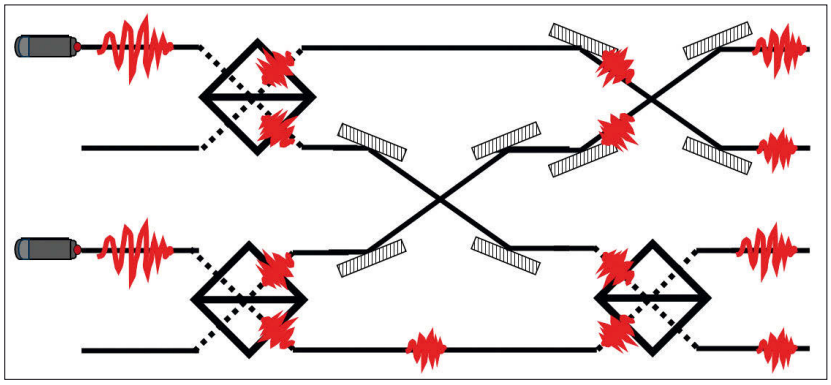


Figure 5



Robert Wang
Bull
Quantum Computing Consultant



Olivier Hess
Bull
Global Quantum Computing Technology Advisor
Quantum Computing Leader France

Importance du HPC dans le calcul scientifique

Les systèmes de calcul haute performance (HPC) constituent aujourd'hui la pointe de la puissance informatique moderne. Ils rassemblent des milliers de processeurs interconnectés capables de traiter en parallèle des volumes massifs de données et de résoudre des problèmes scientifiques ou industriels d'une complexité exceptionnelle. Ces supercalculateurs jouent un rôle essentiel dans de nombreuses avancées majeures, qu'il s'agisse de la modélisation climatique, de la conception de nouveaux médicaments, de la découverte de matériaux innovants ou encore de la simulation de phénomènes physiques complexes. Malgré cela et malgré les progrès du HPC, il existe des problèmes qui sont intraitables par les plus puissants systèmes HPC. Et là, le calcul quantique pourra changer un jour la donne.

Bull, unique constructeur européen d'un HPC autonome

Bull est un leader mondial dans le domaine du HPC. L'entreprise est reconnue pour ses solutions technologiques avancées et son engagement en faveur de l'innovation et de la durabilité. BULL a placé plusieurs de ses supercalculateurs dans le classement TOP500 des supercalculateurs les plus puissants au monde, et elle est également leader dans le domaine des technologies économes en énergie avec des systèmes figurant dans le Green500.

Qaptiva HPC, un émulateur quantique conçu pour HPC

Pour affirmer son engagement envers l'innovation et sa capacité à anticiper les demandes du marché, Bull a développé Qaptiva HPC, une suite logicielle sous licence spécifiquement pour les centres de calcul souhaitant intégrer la technologie quantique dans leurs infrastructures HPC existantes. Qaptiva HPC est conçu pour être compatible avec les clusters HPC standard ; il s'intègre parfaitement aux ordonnanceurs comme SLURM, simplifiant ainsi la création, la planification et l'exécution des tâches quantiques. Le logiciel est polyvalent et prend en charge tout type de porte quantique. De plus, il prend en charge différents types de mesures, telles que l'« sampling » et les « observables », étendant ainsi ses applications à différents scénarios. Grâce à l'émulation quantique, Qaptiva HPC améliore considérablement les capacités de simulation, entraînant des améliorations remarquables en termes de vitesse, d'efficacité et de capacité de calcul, ouvrant ainsi de nouvelles perspectives pour la recherche et le développement.

N'importe quel centre HPC peut installer et utiliser Qaptiva HPC

Si un centre HPC dispose déjà de l'appliance Qaptiva, l'ajout de Qaptiva HPC permet aux utilisateurs de travailler plus efficacement en déchargeant automatiquement les émulations vers le cluster. Cette approche améliore les performances globales de l'appliance et optimise l'utilisation des ressources du centre.

Ce fonctionnement est également pertinent pour un centre HPC qui ne possède pas encore l'appliance Qaptiva mais envisage de l'acquérir, puisque les deux solutions peuvent coexister sans difficulté.

Enfin, si un centre HPC ne souhaite pas utiliser l'appliance Qaptiva, il peut tout à fait installer et exploiter uniquement le logiciel Qaptiva HPC pour réaliser des calculs quantiques, à condition que son cluster soit compatible avec Slurm et Linux.

La capacité d'émulation est uniquement limitée par la mémoire disponible du cluster

La mémoire disponible dans un cluster est définie par le nombre de nœuds et la quantité de RAM sur chaque nœud. Qaptiva HPC peut exécuter des émulations dans les configurations multi-nœuds distribuées (CPU et GPU), exploitant toute la puissance des clusters HPC pour l'émulation quantique. Par exemple, une émulation de 43 qubits logiques a été réalisée sur un superordinateur de 1024 nœuds (128 Go chacun) pour le projet CEA HQI (France Hybrid HPC-Quantum Initiative), avec une performance de 3,8 Pflops.

Comme le montre le tableau ci-dessous, à chaque fois qu'on ajoute un qubit, la dimension de l'espace de Hilbert double, c'est-à-dire que la RAM de l'émulateur double. Cela n'a rien à voir avec l'ajout d'un bit dans le calcul classique. Ainsi, pour gérer 43 qubits, nous avons besoin de 256 To de mémoire RAM. Théoriquement, il est possible d'émuler jusqu'à 63 qubits logiques, ce qui correspond à une RAM de 134 217 728 TB, dépassant la RAM totale du plus grand centre HPC au monde. Dans ce sens, Qaptiva HPC n'a pas de limite sur le nombre de qubits à émuler ; c'est-à-dire, la valeur maximale dépend de la mémoire disponible du cluster HPC considéré. **Figure 1**

Le nombre maximal de qubits émulables par nœud est défini par :

$$L = \log_2\left(\frac{\#(RAM)\ Bytes}{16\ Bytes}\right)$$

Supposons que le nombre de nœuds dans le cluster = 2^N, alors l'ensemble du cluster correspond à un espace de

Hilbert de dimension = nombre de nœuds * dimension de l'espace de Hilbert d'un seul nœud = $2^N * 2^L = 2^{L+N}$. Il est donc possible d'émuler un total de L+N qubits logiques. Par exemple, un cluster est composé de 512 = 2^9 nœuds, chaque nœud peut émuler 32 qubits, le cluster peut émuler alors 41 = 32 + 9 qubits. Il convient de noter qu'en pratique, en raison de divers facteurs — tels que la surcharge mémoire due aux opérations système — le nombre pourrait être L+N−1 au lieu de L+N, la valeur théorique.

Emulations accélérées par des techniques de compilation et d'optimisation avancées

Qaptiva HPC divise les qubits en deux types : locaux et distants. Les qubits locaux sont traités au sein d'un seul nœud, ce qui élimine le besoin de communications entre les nœuds et maximise l'efficacité. Les opérations sur des qubits distants, en revanche, nécessitent une coordination entre les nœuds via MPI, ce qui accroît à la fois la communication et le temps de calcul. Un système de compilation a été conçu pour optimiser le circuit avant son exécution sur un émulateur distribué, un procédé complexe qui a donné lieu à plusieurs brevets, permettant de multiplier la vitesse d'émulation, tout en réduisant l'utilisation de la mémoire. Des benchmarks par rapport à l'un des meilleurs logiciels open source appelé Quest (Université d'Oxford) ont montré que Qaptiva HPC est presque 5 fois plus rapide jusqu'à 40 qubits. Des mesures de performance ont montré que Qaptiva HPC est d'autant plus efficace que le cluster est grand.

Emulation avec ou sans bruit

Actuellement, Qaptiva HPC propose deux types d'émulateurs quantiques distribués :

- DLinAlg (émulation idéale et sans bruit) et
- DNoisy (émulation avec bruit),

Tous deux entièrement intégrés au planificateur de ressources SLURM et optimisés pour les clusters HPC. DLinAlg est un émulateur basé sur l'algèbre linéaire, conçu pour représenter les états quantiques purs par des vecteurs d'état composés d'amplitudes complexes. Ces amplitudes peuvent être stockées en simple ou double précision, selon les besoins de l'utilisateur. En revanche, DNoisy est un émulateur spécifiquement conçu pour émuler des circuits quantiques avec bruit. Cet émulateur permet à l'utilisateur d'intégrer les imperfections du matériel de l'ordinateur quantique lors de la simulation afin de prédire les effets du bruit. La simulation bruitée impose des exigences de mémoire et de communication nettement plus élevées, limitant DNoisy à un maximum théorique de 31 qubits bruités.

Qaptiva HPC est installé sur un cluster HPC comme une simple bibliothèque d'applications. Pour ceux qui utilisent Qaptiva, il est facile de traduire un programme quantique de Qaptiva en Qaptiva HPC, simplement en remplaçant l'émulateur LinAlg par l'émulateur DLinAlg pour l'émulation sans bruit, et NoisyQProc par CNoisy ou DNoisy pour l'émulation avec bruit. Dans les deux cas, il faut utiliser un script sbatch pour le soumettre, que nous allons montrer dans la section de démonstration suivante.

Distributed CPU mode:

Total cluster memory (RAM)	Max # qubits
1 TB	36
2 TB	37
4 TB	38
16 TB	40
128 TB	43
256 TB	44

Figure 1

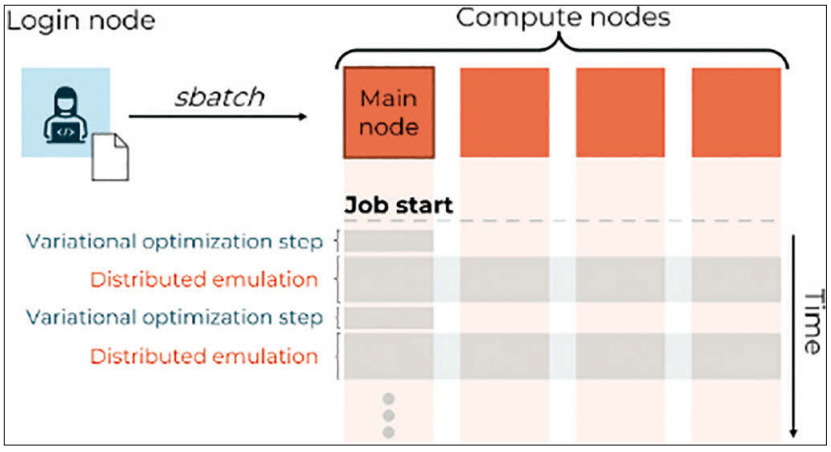


Figure 2

Pipeline de traitement HPC Qaptiva

Le processus de répartition des tâches vers les nœuds disponibles du cluster commence lorsque l'utilisateur se connecte au nœud de connexion (Login node), où le script batch SLURM utilisé pour planifier le travail d'émulation quantique est préparé. **Figure 2**

Dans ce script, l'utilisateur spécifie les ressources informatiques nécessaires telles que le nombre de nœuds, le nombre de tâches par nœud, la mémoire et le temps maximum d'exécution, ainsi que la configuration d'exécution. Une fois le script prêt, le travail est soumis au planificateur SLURM en utilisant la commande sbatch, qui gère l'allocation des ressources et la répartition des travaux quantiques à travers le cluster. Le travail est d'abord envoyé à un nœud principal (Main node), qui gère l'optimisation itérative (importante pour les algorithmes variationnels). Après chaque étape d'optimisation, les circuits quantiques sont envoyés aux nœuds de calcul (Compute nodes) pour émulation. Les résultats retournent au nœud principal, qui poursuit la boucle d'optimisation. Ce flux de travail centralisé rend le processus plus efficace et plus facile à gérer pour l'utilisateur.

Démos de code : Comment l'utiliser ?

Exemple de QAOA (Quantum Approximate Optimization Algorithm)

QAOA est un algorithme variationnel qui utilise des circuits quantiques pour trouver des solutions approximatives à des problèmes d'optimisation combinatoire.

La première étape consiste à préparer un script Python qui

décrit le programme quantique à exécuter, comme on peut le voir dans le code (dlinalg_qaoa.py) ci-dessous :

```
import networkx as nx
from qat.generators import MaxCutGenerator
from qat.plugins import ScipyMinimizePlugin
from qat.qpus import DLinAlg                    # Import HPC emulator
import time

t_ = time.time()

graph = nx.full_rary_tree(2, 5)

max_cut_application = (
    MaxCutGenerator(job_type="qaoa")
    | ScipyMinimizePlugin(method="COBYLA", tol=1e-5, options={"maxiter": 20})
    | DLinAlg(temp_dir="/tmp/")
)
combinatorial_result = max_cut_application.execute(graph)

print(combinatorial_result.subsets)
print(combinatorial_result.cost)
t_ = time.time() - t_
```

L'étape suivante consiste à définir un script batch SLURM pour configurer les ressources du cluster. Ce script inclut des options précédées de #SBATCH pour spécifier des ressources telles que les types de nœuds, le nombre de nœuds, le temps d'exécution maximum, les noms des fichiers de sortie, etc., comme le montre le code ci-dessous.

```
#!/usr/bin/env bash
#SBATCH —nodes 2                    #Nombre de nœuds
#SBATCH —ntasks-per-node=2          #Nombre de tâche par nœud
#SBATCH —cpus-per-task=32           # Nombre de CPU par tâche
#SBATCH —time 00:05:00              #Durée maximal
#SBATCH —job-name dlinalg_qaoa
#SBATCH —output slurm_output/dlinalg_qaoa.out   #Nom du fichier de sortie
#SBATCH —error slurm_output/dlinalg_qaoa.err    #Nom du fichier des erreurs
#SBATCH –exclusive                  #Exclusivité de nœud
module load python3
module load qaptiva-hpc
python3 -u dlinalg_qaoa.py          #Exécuter un script python
```

Si aucune partition n'est spécifiée, Slurm utilise la partition par défaut définie par l'administrateur. Il est toutefois recommandé de la préciser explicitement avec l'option —partition afin de contrôler les ressources utilisées.
Le script, nommé submit_dlinalg.sh par exemple, est soumis au cluster à l'aide de la commande sbatch:

```
$> sbatch submit_dlinalg.sh
```

Une fois le calcul terminé, les résultats peuvent être récupérés à partir du journal de sortie SLURM :

```
$> cat slurm_output/ dlinalg_qaoa.out
```

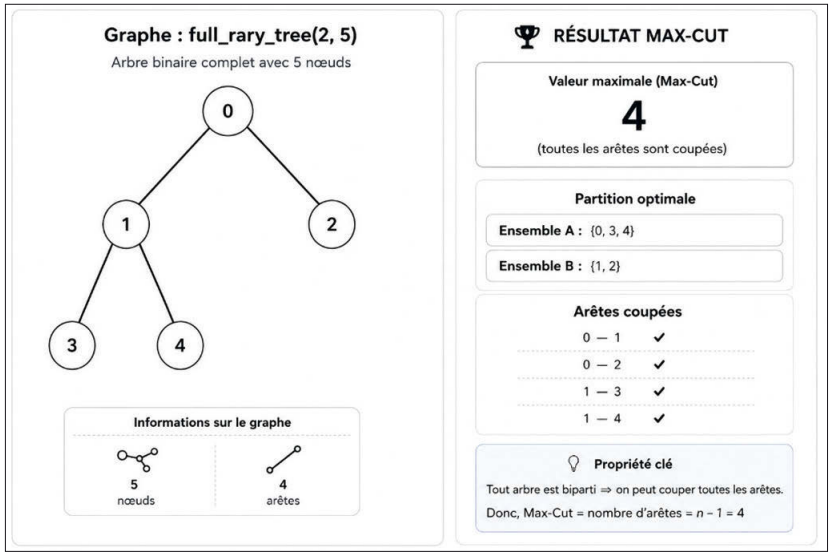
Voici le résultat :

```
Loading the qaptiva-hpc module environment
Running loopback test for DLinAlg using 2 processes
DLinAlg install check successful (C++ executable works)
Running the DLinAlg simulation
[[0, 3, 4], [1, 2]]
-4.0
```

Le calcul MaxCut s'est déroulé correctement. **Figure 3**

Pour davantage d'informations, veuillez consulter les références.

Figure 3



Références

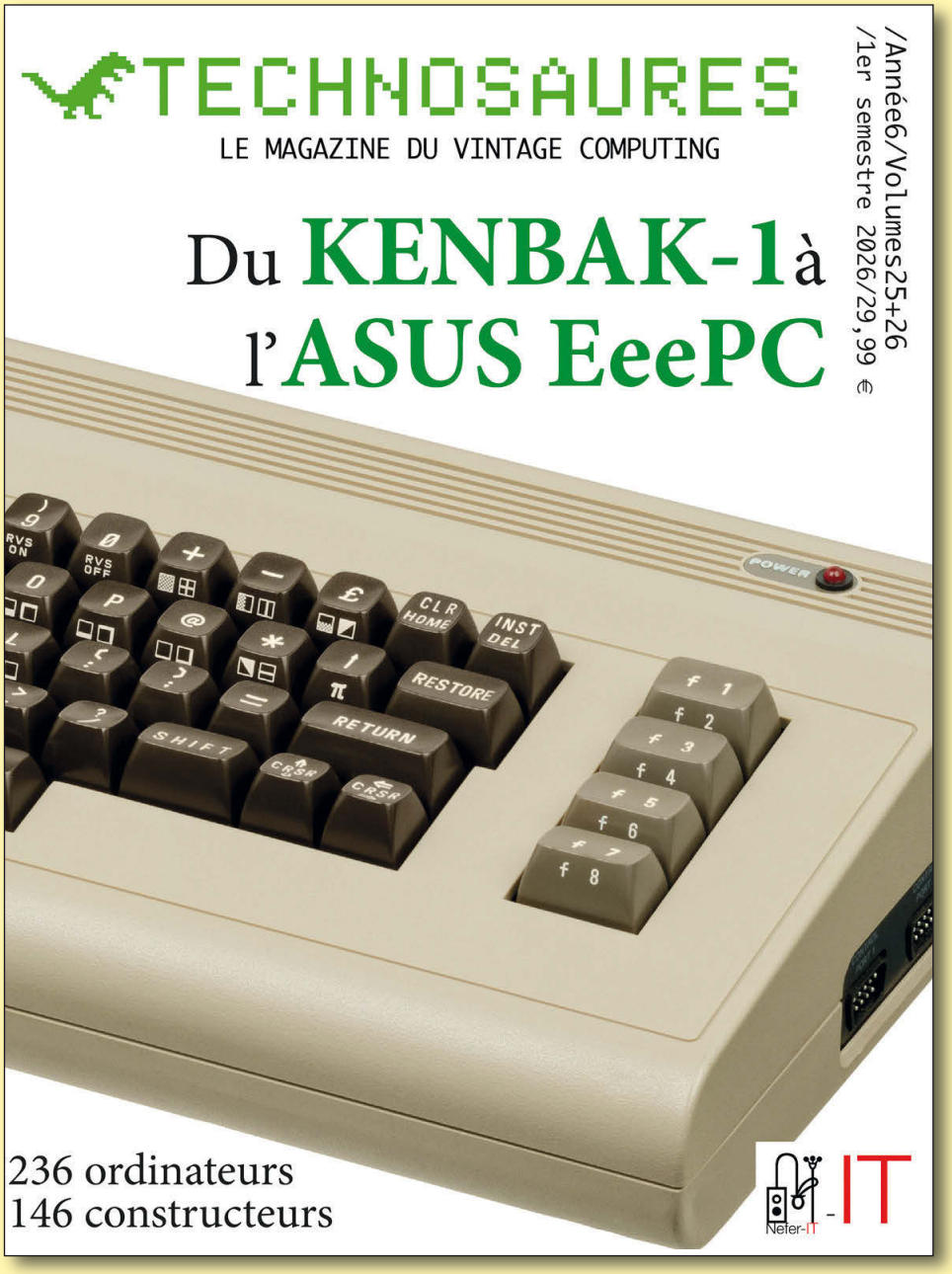
[1] https://www.bull.com/en/products/quantum-computing/qaptiva-hpc
[2] https://qlm.bull.com/dqlm-doc/
[3] https://doi.org/10.1177/00375497261430860

PROGRAMMEZ! N°278
Disponible à partir du
20 novembre 2026

PROCHAINS NUMÉROS

HORS SÉRIE N°24
Disponible à partir du
25 septembre 2026

Le nouveau Technosaures est disponible !



Une anthologie des micro-ordinateurs du Kenbak-1 à l'ASUS EeePC*

Ce numéro rassemble :

236 micro-ordinateurs
146 constructeurs
162 pages

Vous pouvez commander ce numéro dès maintenant programmez.com et amazon.fr

** presque une anthologie, impossible de répertorier tous les modèles sortis*



Mathieu Garcia

Mathieu Garcia est ingénieur logiciel chez OVHcloud, où il travaille sur l'intégration de l'informatique quantique dans le cloud dans l'équipe de Fanny Bouton. Il a participé à la mise à disposition des émulateurs et QPU quantiques du catalogue OVHcloud et s'intéresse aux usages concrets des frameworks open source pour rendre la programmation quantique accessible aux développeurs.

OVHcloud

Frameworks quantiques : panorama et premiers pas avec les émulateurs

L'informatique quantique entre progressivement dans la réalité des développeurs. Ce qui relevait encore il y a dix ans de la physique fondamentale est aujourd'hui accessible depuis un simple terminal Python : des dizaines de frameworks open source permettent d'écrire, de simuler et d'exécuter des circuits quantiques sur des processeurs physiques accessibles depuis le cloud. La France et l'Europe jouent dans ce domaine un rôle de premier plan, portées par des startups comme Pasqal, Quandela, Alice & Bob, Quobly ou ColibriTD, qui n'ont pas à rougir dans cette course à l'équipement qui est internationale.

Dans cet article, nous vous proposons un tour d'horizon des technologies de qubits et de leurs frameworks logiciels associés, une présentation des émulateurs disponibles chez OVHcloud, puis une mise en pratique concrète avec quatre exemples de code « Hello World » couvrant des approches radicalement différentes. L'objectif n'est pas de faire de vous un expert en mécanique quantique, mais de vous donner les clés pour commencer à expérimenter et comprendre ce nouvel écosystème.

L'informatique quantique : une nouvelle façon de calculer

Un ordinateur classique manipule des **bits**, chacun valant 0 ou 1. Un ordinateur quantique manipule des **qubits** (quantum bits) qui, grâce à un phénomène appelé **superposition**, peuvent se trouver simultanément dans un état intermédiaire entre 0 et 1 jusqu'au moment de la mesure. Ce n'est pas de la magie : c'est une conséquence directe des lois de la mécanique quantique, qui décrit le comportement des systèmes physiques à l'échelle atomique.

À la superposition s'ajoutent deux autres ressources fondamentales. L'**intrication quantique** lie les états de plusieurs qubits de façon non-classique. L'**interférence quantique**, enfin, permet de renforcer les chemins de calcul menant à la bonne réponse et d'annuler ceux qui mènent aux mauvaises. C'est le cœur du fonctionnement de la plupart des algorithmes quantiques.

Ces propriétés confèrent aux ordinateurs quantiques un avantage théorique pour certaines classes de problèmes. L'**algorithme de Shor** permet de factoriser des entiers de grande taille en temps polynomial, ce qui menace directement le chiffrement RSA et a déclenché la course au chiffrement post-quantique. L'**algorithme de Grover** offre une accélération quadratique pour la recherche dans des bases de données non triées. Les **algorithmes variationnels quantiques** (VQE, QAOA) permettent d'approcher les états fondamentaux de molécules ou de résoudre des problèmes d'optimisation combinatoire - des applications concrètes en pharmacologie, matériaux et logistique. Enfin, les **réseaux de neurones quantiques** (QNN) et l'apprentissage automatique quantique (QML) explorent comment les circuits paramétriques peuvent accélérer certaines tâches de classification ou de génération. Mais attention : l'ordinateur quantique n'est pas un ordinateur classique « plus rapide ». Pour les tâches courantes telles que le traitement de fichiers, les bases de données, ou la navigation web, un processeur conventionnel reste indétrônable. L'avantage quantique ne se manifeste que sur des problèmes précis, souvent formulés de façon non triviale.

La programmation quantique repose sur la notion de **circuit quantique** : une suite de **portes quantiques** (opérations unitaires) appliquées à des qubits, suivie de mesures. Les portes fondamentales incluent la porte **X** (NOT quantique, qui inverse l'état d'un qubit), la porte **Hadamard (H)** (qui crée une superposition équiprobable), la porte **CNOT** (qui intrique deux qubits) et les **portes de rotation** Rx, Ry, Rz (qui font tourner l'état du qubit sur la sphère de Bloch, qui est une représentation géométrique de l'état d'un qubit, l'équivalent quantique d'un bit). Un circuit quantique se représente graphiquement par des lignes horizontales (une par qubit) sur lesquelles s'enchaînent les portes : c'est la notation universelle, supportée par tous les frameworks.

Nous sommes aujourd'hui dans l'ère **NISQ** (Noisy Intermediate-Scale Quantum), caractérisée par des processeurs de quelques dizaines à quelques centaines de qubits, encore bruités et sujets à la **décohérence** c'est-à-dire la perte progressive de l'état quantique sous l'effet des perturbations environnementales. La **correction d'erreurs quantiques** est le grand chantier des années à venir : elle nécessite de nombreux qubits physiques pour encoder un seul qubit logique fiable, et représente le passage vers l'ère quantique à pleine échelle.

Panorama des technologies de QPU

Il n'existe pas une seule façon de construire un qubit. Plusieurs technologies physiques sont actuellement en compétition, chacune avec ses forces, ses faiblesses et son écosystème logiciel propre. Voici un tour d'horizon des quatre grandes approches.

Qubits à base d'atomes : neutres et ions piégés

Les **ions piégés** sont des atomes chargés maintenus en lévitation par des champs électromagnétiques dans un vide

poussé. Les opérations quantiques sont réalisées par des lasers ultraprécis ou avec des micro-ondes dans un champ magnétique variable. Cette technologie offre des qubits de très haute-fidélité (>99,9 % pour certaines portes à deux qubits) et de longues durées de cohérence, au prix d'une exécution relativement lente et d'une mise à l'échelle pour l'instant limitée.

Les principaux acteurs sont **IonQ** (États-Unis, framework IonQ SDK), **Quantinuum** (issue de Honeywell, framework **TKET**), AQT (Autriche).

Les **atomes neutres** sont piégés individuellement par des pinces laser (tweezer arrays) dans un réseau reconfigurable en 2D ou 3D. Ils offrent une excellente scalabilité, comprendre qu'on peut agencer plusieurs centaines d'atomes et bénéficier d'une grande flexibilité dans la connectivité. **Pasqal** (France, framework **Pulser**) et **QuEra** (États-Unis, framework Bloqade) sont les leaders de ce segment, avec des QPU de plusieurs centaines d'atomes en production depuis 2025 (140 pour Pasqal, 256 pour QuEra).

Qubits électroniques : supraconducteurs et spins

Les **qubits supraconducteurs** dominent actuellement le marché. Fabriqués à partir de circuits électroniques en aluminium ou niobium déposés sur silicium, ils sont refroidis à des températures proches du zéro absolu (15 à 20 millikelvins) dans des cryostats. Ils offrent des temps de porte rapides (de l'ordre de la dizaine de nanosecondes) et une bonne intégration sur puce. **IBM** (framework **Qiskit**), **Google** (framework **Cirq**), et **IQM** (Finlande, framework **Qrisp**) sont les acteurs principaux. **Figure 1**

La startup française **Alice & Bob** développe une approche originale avec les qubits « à chat de Schrödinger » : l'information est encodée dans des états cohérents d'un oscillateur micro-onde, offrant une protection naturelle contre les erreurs de bit- flip (inversion de la valeur d'un bit qui inverse les états 0 et 1). Leur framework de simulation s'appelle **Felis**. **C12** (France, framework **Callisto**) utilise des nanotubes de carbone comme support de qubits de spin électronique ultra-purifié. **Quobly** (France, framework **QLEO**) parie sur le silicium CMOS compatible avec les chaînes de fabrication microélectroniques standard.

Qubits photoniques

Les **qubits photoniques** utilisent des photons uniques comme porteurs d'information quantique. Ils présentent plusieurs avantages : ils sont naturellement adaptés aux communications quantiques sur fibre optique, et peuvent être manipulés par des composants d'optique intégrée et fabriqués dans des fonderies plus standards. Le défi principal reste la génération déterministe et la manipulation efficace de photons uniques, ainsi que les pertes dans les guides d'onde.

Quandela (France) est l'acteur le plus avancé en Europe sur ce segment, avec des sources de photons uniques à base de boîtes quantiques semi-conductrices. Leur framework open source **Perceval** permet de simuler et programmer des circuits photoniques discrets (séparateurs de faisceau, déphaseurs, sources, détecteurs), tandis que leur framework **Merlin** en est l'extension accélérée par GPU et orientée machine learning quantique. **PsiQuantum** (États-Unis) vise la fabrication de QPU photoniques à grande échelle via les fonderies de semi-conducteurs.



Figure 1 — *QPU photonique Quandela Belenos*

Qubits topologiques

Les **qubits topologiques** reposent sur des quasi-particules appelées **fermions de Majorana**, dont la nature topologique les rend intrinsèquement robustes aux perturbations locales — une forme de correction d'erreurs physique inscrite dans la topologie même du matériau. **Microsoft** (framework **Q#** / **Azure Quantum**) est le pionnier de cette approche. Cette dernière technologie reste à être démontrée et relève encore de la recherche fondamentale.

3. L'écosystème logiciel : émulateurs et frameworks

A quoi sert un émulateur quantique ?

Avant d'accéder à un vrai QPU – rare et encore coûteux et dont le temps est souvent alloué par files d'attente - les développeurs utilisent des **émulateurs** : des logiciels qui simulent le comportement d'un circuit quantique sur un ordinateur classique. Ces outils sont indispensables pour prototyper des algorithmes, apprendre la programmation quantique, déboguer des circuits et valider des résultats avant soumission sur hardware.

La limite fondamentale des émulateurs est leur adaptabilité : simuler exactement un circuit de N qubits requiert de stocker 2^N amplitudes complexes en mémoire (le vecteur d'état). Un circuit de 30 qubits nécessite environ 8 Go, 40 qubits nécessitent plusieurs téraoctets ; au-delà, la simulation exacte devient impossible sur un ordinateur classique. Pour contourner cette limite, certains émulateurs utilisent des représentations tensorisées (MPS ou Matrix Product States) qui exploitent la structure d'intrication limitée des circuits peu profonds, des approximations (bruit artificiel ou troncature), ou l'accélération GPU pour paralléliser massivement les calculs matriciels.

Catalogue des émulateurs disponibles **Figure 2**

OVHcloud propose une sélection d'émulateurs couvrant un large spectre de technologies et de cas d'usage :

- **Alice & Bob Felis** : simulation de qubits de chat avec modè-

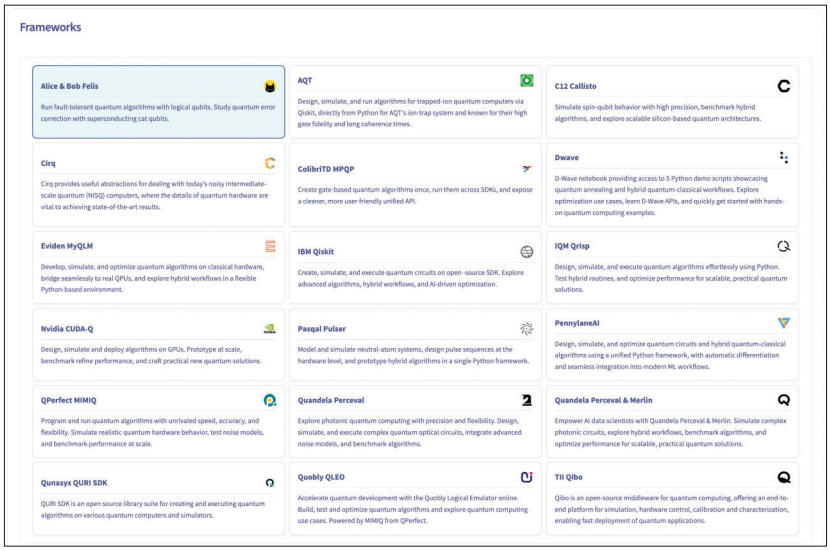


Figure 2 — *Catalogue des frameworks et émulateurs quantiques dans l'interface OVHcloud*

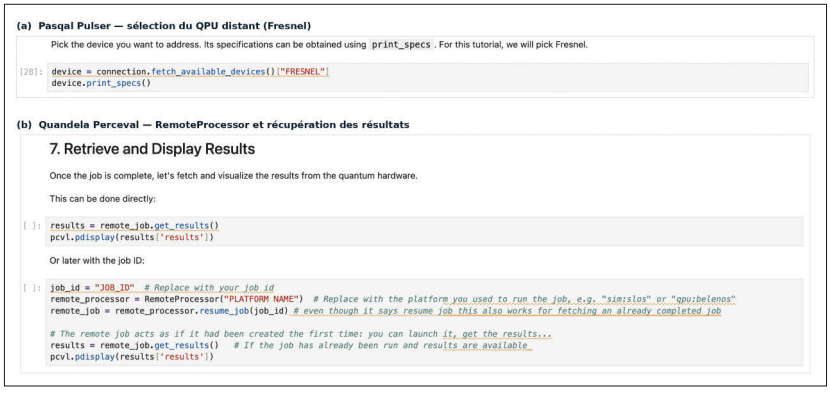


Figure 3 — *Passer de l'émulateur local au QPU cloud tient à quelques lignes : (a) avec Pulser, on sélectionne le QPU Pasqal FRESNEL parmi les devices disponibles ; (b) avec Perceval, on instancie un Remote Processor sur la plateforme visée ("sim:slos"pour l'émulateur,"qpu:belenos"pour le QPU photonique) avant de récupérer les résultats. Le circuit, lui, ne change pas.*

le de bruit réaliste. Idéal pour explorer la correction d'erreurs et la logique quantique protégée,

- **Bull MyQLM** : plateforme française multi-backends pour l'algorithmique, la chimie et le machine learning quantique,
- **C12 Callisto** : simulation de qubits carbone haute-fidélité, axée sur la correction d'erreurs et la logique quantique,
- **Cirq** : framework Python généraliste pour l'algorithmique, la chimie et l'optimisation,
- **ColibriTD MPQP** : API unifiée multi-SDK permettant d'écrire un algorithme une seule fois et de le soumettre sur Qiskit, Cirq ou d'autres backends sans modifier le code,
- **Qiskit** : le framework le plus populaire au monde, avec un vaste écosystème de librairies (Qiskit Nature, Qiskit Finance, Qiskit Optimization, Qiskit Machine Learning),
- **IQM Qrisp** : simulation orientée algorithmique et optimisation, spécialisée pour les architectures IQM,
- **Nvidia Cuda-Q** : simulation accélérée GPU/HPC pour des circuits de grande taille ; particulièrement adapté à la chimie et à l'optimisation à grande échelle,
- **Pasqal Pulser** : simulation de séquences d'impulsions laser sur registres d'atomes neutres, pour la recherche fondamentale et l'optimisation,
- **Pennylane AI**: spécialisé en machine learning quantique (QML) et hybridation quantique-classique avec autodifférenciation automatique,
- **Qperfect MIMIQ** : simulation haute performance de circuits quantiques généraux, pour la recherche et l'enseignement,
- **Quandela Perceval** : simulation de circuits photoniques discrets pour la validation d'algorithmes photoniques,
- **Quandela Merlin** : simulation avancée accélérée GPU orientée machine learning quantique photonique,
- **Qunasys QURI SDK** : framework japonais spécialisé en chimie quantique et optimisation variationnelle,
- **Quobly QLEO** : simulation de qubits silicium CMOS pour l'algorithmique et la recherche sur architectures CMOS,
- **Qibo** : framework généraliste open source pour l'algorithmique, la chimie, l'optimisation et le QML.

4. Les QPU du catalogue public

Les QPU (Quantum Processing Units) sont les processeurs quantiques physiques accessibles en cloud. Ces machines permettent d'exécuter des circuits quantiques réels, avec les contraintes de bruit et de cohérence propres aux technologies actuelles (NISQ). L'accès se fait généralement via les mêmes frameworks que les émulateurs : il suffit de changer le backend cible : là où l'on spécifiait AerSimulator() pour un simulateur local Qiskit, on renseignera les identifiants d'accès à un vrai QPU IQM, Pasqal ou Quandela. Cette portabilité est l'un des grands atouts de l'écosystème open source quantique actuel. **Figure 3**

L'offre OVHcloud permet aujourd'hui l'accès à deux QPU physiques représentatifs de deux technologies radicalement différentes.

Pasqal Orion Beta — orion-beta (100 qubits, atomes neutres)

L'Orion Beta de Pasqal est un processeur quantique à **atomes neutres** de 100 qubits, accessible par le cloud via le framework **Pulser**. Les atomes de rubidium sont piégés individuellement par des pinces laser dans un réseau 2D configurable, puis adressés par des impulsions micro-ondes et laser pour réaliser les calculs en mode analogique.

Ce type de machine est particulièrement adapté aux problèmes d'**optimisation combinatoire** (MAX-CUT, coloration de graphe, problèmes de satisfaction de contraintes) grâce à son architecture analogique native : on programme directement la position des atomes dans des graphes 2D que des portes abstraites, ce qui permet d'encoder naturellement des Hamiltoniens d'Ising c'est à dire la classe de problèmes qui sous-tend de nombreux cas d'usage industriels en logistique, finance et scheduling. L'Orion Beta permet également des expériences de **simulation quantique** (modélisation de systèmes de spins, transitions de phase quantiques) et l'exploration d'algorithmes variationnels comme QAOA (Quantum Approximate Optimization Algorithm).

Quandela Belenos — belenos-12 (12 qubits photoniques)

Le Belenos de Quandela est un processeur quantique **photonique** de 12 qubits, accessible via le framework **Perceval**. Il repose sur des sources de photons uniques à base de boîtes quantiques semi-conductrices refroidies à 4 K, couplées à un circuit d'optique intégrée sur puce silicium. Les portes quantiques sont réalisées par des interféromètres à séparateurs de faisceau et déphaseurs contrôlables électriquement.

Avec 12 qubits photoniques, Belenos est adapté aux expé-

riences de **boson sampling** et de **Gaussian boson sampling** : des protocoles qui servent à démontrer l'avantage quantique dans des tâches d'échantillonnage et à explorer des applications en **chimie quantique** (calcul de propriétés moléculaires via l'évaluation de permanents de matrices) et en **machine learning quantique**
(classification, réseaux de neurones photoniques). Sa nature photonique en fait également une plateforme naturelle pour les expériences de **communication quantique** et de **cryptographie quantique** (distribution de clés QKD).

```
# Connexion au Belenos-12 via Perceval
import perceval as pcvl
from perceval.providers import QuandelaProvider

provider = QuandelaProvider(token="VOTRE_TOKEN")
backend = provider.get_backend("belenos-12")

# Le circuit Perceval écrit pour le simulateur local
# fonctionne sans modification sur le QPU physique
circuit = pcvl.Circuit(2).add(0, pcvl.BS())
# ... soumission identique à la simulation locale
```

Premiers pas : Hello World quantique

Qu'est-ce qu'un « Hello World » en quantique ?

Dans la programmation classique, un « Hello World » valide l'installation d'un environnement et illustre la syntaxe de base d'un langage. En informatique quantique, son équivalent est un circuit qui place un qubit en **superposition** via une porte Hadamard, puis le mesure. Le résultat, 0 ou 1, avec une probabilité de 50 % chacun prouve que le simulateur fonctionne et que l'on comprend la mécanique de base.
La porte Hadamard transforme l'état $|0\rangle$ en $(|0\rangle + |1\rangle)/\sqrt{2}$: une superposition équiprobable des deux états classiques. Lors de la mesure, cette superposition s'effondre aléatoirement sur 0 ou sur 1. En répétant l'expérience 1024 fois (shots), on obtient une distribution statistique proche de 50/50 c'est à dire la signature d'une vraie superposition. Si le résultat était toujours 0 ou toujours 1, cela indiquerait un bug dans le circuit ou dans l'environnement.
Pour suivre les exemples ci-dessous, assurez-vous d'avoir Python 3.10+ installé. Chaque framework s'installe via pip :

```
pip install qiskit qiskit-aer#                    Exemple 1
pip install qiskit qiskit-alice-bob-provider      Exemple 2
pip install perceval-quandela                     Exemple 3
pip install mpqp                                  Exemple 4
```

Exemple 1 : Hello World avec Qiskit

Qiskit est le framework open source le plus utilisé dans le monde. Depuis la version 1.0, le module `Aer` est séparé dans le package `qiskit-aer`. Son simulateur `AerSimulator` permet de tester des circuits localement, sans connexion réseau.

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

# Création d'un circuit quantique avec 1 qubit et 1 bit classique
qc = QuantumCircuit(1, 1)

# Mise en superposition du qubit (porte Hadamard)
qc.h(0)

# Mesure du qubit -> résultat stocké dans le bit classique
qc.measure(0, 0)

# Exécution sur le simulateur local
backend = AerSimulator()
job = backend.run(qc, shots=1024)
result = job.result()
counts = result.get_counts()
print(counts) # ex : {'0': 512, '1': 512}
```

Explications : On crée un circuit à 1 qubit et 1 bit classique. La porte `h(0)` place le qubit 0 en superposition. `measure(0, 0)` projette son état et stocke le résultat dans le registre classique. Le simulateur répète l'expérience 1024 fois et `get_counts()` retourne le dictionnaire des occurrences. Qiskit permet également de visualiser le circuit avec `qc.draw('mpl')` et la distribution des mesures avec `plot_histogram(counts)`. **Figure 4**

Exemple 2 : Hello World avec Alice & Bob Felis

Alice & Bob développe des qubits de chat encodés dans des états cohérents d'un oscillateur micro-ondes et offrant une protection naturelle contre les erreurs de bit-flip. Le framework Felis s'intègre directement dans l'interface Qiskit, ce qui simplifie la prise en main pour les développeurs déjà familiers avec cet écosystème.

```
from qiskit import QuantumCircuit
from qiskit_alice_bob_provider import AliceBobLocalProvider

# Initialisation du provider Alice & Bob (simulation locale)
provider = AliceBobLocalProvider()
backend = provider.get_backend('EMU:1Q:LESCANNE_2020')

# Création du circuit
qc = QuantumCircuit(1, 1)
qc.h(0)
qc.measure(0, 0)

# Exécution sur le simulateur Alice & Bob
job = backend.run(qc, shots=1024)
result = job.result()
counts = result.get_counts()
print(counts)
```

Explications : `AliceBobLocalProvider` expose les backends de simulation Alice & Bob directement via l'interface Qiskit. Le backend `EMU:1Q:LESCANNE_2020` simule un qubit de chat avec un modèle de bruit réaliste calqué sur le matériel réel. L'intérêt

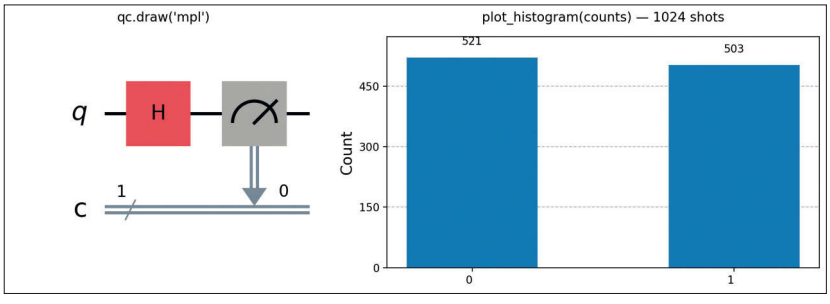


Figure 4 — *À gauche, le circuit dessiné par `qc.draw('mpl')` : une porte Hadamard suivie d'une mesure vers le registre classique. À droite, l'histogramme produit par `plot_histogram(counts)` après 1024 exécutions sur `erSimulator: 521 fois 0, 503 fois 1` — la superposition se lit comme un tirage à pile ou face, et l'écart au 512/512 théorique est le bruit statistique de l'échantillonnage.*

de ce modèle de bruit asymétrique est que les erreurs de bit-flip (X) sont exponentiellement supprimées, tandis que les erreurs de phase (Z) restent à corriger, ce qui change la stratégie de conception des circuits.

Exemple 3 : Hello World avec Quandela Perceval & Merlin

Quandela développe des QPU photoniques. En photonique, on ne parle pas de qubits mais de **modes** et de **photons** ; l'équivalent d'une porte Hadamard est un **séparateur de faisceau 50/50** (beam splitter) qui place un photon en superposition sur deux modes. Perceval est le framework de simulation
; Merlin est son extension GPU, utile pour les grands circuits et les applications de machine learning.

```
import perceval as pcvl
import perceval.components as comp

# Création d'un circuit photonique à 2 modes (séparateur 50/50)
circuit = pcvl.Circuit(2)
circuit.add(0, comp.BS()) # Beam Splitter = Hadamard photonique

# Etat d'entrée : 1 photon dans le mode 0, 0 dans le mode 1
input_state = pcvl.BasicState([1, 0])

# Simulation avec le backend Naive (CPU)
# Remplacer "Naive" par "SLOS" pour utiliser Merlin sur GPU
processor = pcvl.Processor("Naive", circuit)
processor.with_input(input_state)

# Echantillonnage de 1024 mesures
sampler = pcvl.algorithm.Sampler(processor)
output = sampler.sample_count(1024)
print(output['results']) # ex : {|1,0>: ~512, |0,1>: ~512}
```

Explications : Un circuit à 2 modes avec un beam splitter 50/50 est l'équivalent photonique d'une porte Hadamard. Un photon injecté dans le mode 0 se retrouve, après mesure, dans l'un ou l'autre mode avec une probabilité égale. pcvl.Processor enveloppe le circuit et l'état d'entrée dans un objet que l'API Sampler attend (les versions récentes de Perceval n'acceptent plus un backend brut). Pour bénéficier de l'accélération GPU de Merlin, il suffit de remplacer "Naive"par "SLOS", LOS", le reste du code étant identique.

Exemple 4 : Hello World avec ColibriTD MPQP

MPQP (Multi-Platform Quantum Programming) de ColibriTD est une API unifiée : on écrit son circuit une seule fois, et on peut le soumettre sur Qiskit, Cirq, MyQLM ou tout autre backend supporté, sans changer une ligne de code. C'est particulièrement utile pour comparer des résultats entre simulateurs, benchmarker les backends, ou rendre son code portable et indépendant du constructeur.

```
from mpqp import QCircuit
from mpqp.gates import H
from mpqp.measures import BasisMeasure
from mpqp.execution import run, IBMDevice, GOOGLEDevice

# Création du circuit (syntaxe unifiée MPQP)
circuit = QCircuit(1)
circuit.add(H(0))
circuit.add(BasisMeasure([0], shots=1024))

# Exécution sur le simulateur IBM Aer
result_ibm = run(circuit, IBMDevice.AER_SIMULATOR)
print("IBM:", result_ibm)
# IBM: Result: IBMDevice, AER_SIMULATOR #  Counts: [527, 497]
#   Probabilities: [0.51465, 0.48535] #    Samples:
#   State: 0, Index: 0, Count: 527, Probability: 0.5146484
#       State: 1, Index: 1, Count: 497, Probability: 0.4853516 #
    Error: None
# Le même circuit, soumis au simulateur Google Cirq sans modification
result_google = run(circuit, GOOGLEDevice.CIRQ_LOCAL_SIMULATOR)
print("Google:", result_google)
# Google: Result: GOOGLEDevice, CIRQ_LOCAL_SIMULATOR
#   Counts: [524, 500]
#   Probabilities: [0.51172, 0.48828] #
    Samples:
#   State: 0, Index: 0, Count: 524, Probability: 0.5117188
#       State: 1, Index: 1, Count: 500, Probability: 0.4882812
#   Error: None
```

Explications : MPQP traduit automatiquement le circuit dans le langage natif du backend cible (OpenQASM 3 pour IBM, JSON Cirq pour Google, etc.). Les résultats des deux simulateurs devraient être identiques : environ 512 occurrences de 0 et 512 de 1. Pour viser un vrai QPU en cloud, il suffit de remplacer le device cible et de fournir les identifiants d'accès, sans toucher au circuit lui-même.

Conclusion et ressources

L'informatique quantique est à un tournant. Les frameworks open source se multiplient, les émulateurs sont accessibles depuis le Cloud tout comme les premiers QPU cloud qui permettent déjà d'expérimenter sur du matériel réel. Les développeurs qui s'y forment aujourd'hui prendront de l'avance sur des marchés en forte croissance : finance, pharmacologie, logistique, cybersécurité post-quantique.

Les quatre exemples de cet article ne représentent qu'une infime partie des approches possibles. Chaque framework a sa propre philosophie : Qiskit privilégie l'accessibilité et l'écosystème étendu, Perceval la rigueur du modèle photonique, Felis la fidélité au matériel avec un modèle de bruit réaliste, et MPQP la portabilité multi-backends. Choisir le bon outil dépend du problème à résoudre, de la technologie QPU cible et de la maturité souhaitée dans l'expérimentation.

Une recommandation pratique : commencez par Qiskit si vous débutez dont l'écosystème, la documentation et sa communauté en font le choix le plus accessible. Explorez ensuite Pennylane si vous avez un profil machine learning, ou Perceval si les architectures photoniques vous attirent. MPQP est un excellent deuxième outil pour écrire du code portable une fois que vous maîtrisez les bases.

Démarrer sur OVHcloud :
https://www.ovhcloud.com/fr/public-cloud/quantum-computing/

Introduction à la correction d'erreur quantique



Romain PERRIER

Actuellement en thèse à ETIS et au LPTM dans le cadre d'un contrat CNRS.

mail :
romain.perrier@ensea.com

Un ordinateur classique peut fonctionner pendant des années sans que l'utilisateur ait conscience de l'existence des erreurs matérielles. En informatique classique, les erreurs de bits existent, mais elles sont généralement transparentes pour l'utilisateur grâce aux mécanismes de détection et de correction intégrés dans le matériel et les logiciels. Par exemple, lorsque nous stockons une photo sur notre ordinateur, les mécanismes de correction (comme les codes correcteurs d'erreurs) évitent qu'un défaut matériel ne la transforme et la corrompe. De même lorsque l'on exécute un programme sur un ordinateur classique, les erreurs de bits sont déjà prises en charge par le matériel ou par des codes correcteurs d'erreurs. L'utilisateur n'a donc généralement pas à s'en préoccuper.

Cependant, l'architecture d'un ordinateur quantique actuel est très différente. Un ordinateur classique peut être vu comme un ensemble de blocs spécialisés : mémoire vive, processeur, stockage, etc. À l'inverse, les machines quantiques actuelles ressemblent davantage à des processeurs quantiques qu'à des ordinateurs complets. Les qubits (unité d'information quantique : littéralement quantum bit) sont préparés, manipulés puis mesurés, mais il n'existe pas encore de mémoire quantique générale, suffisamment fiable et durable, comparable à la mémoire vive d'un ordinateur classique.

L'information quantique circule donc uniquement pendant l'exécution du circuit. Si une erreur apparaît sur un qubit, elle modifie directement l'information transportée. Un qubit ne pouvant pas être copié et sa mesure détruisant son état, corriger les erreurs est bien plus difficile que dans le monde classique. Les erreurs elles-mêmes sont également différentes.

Qu'est-ce qu'une erreur?

Erreur en informatique classique

En informatique classique, on considère essentiellement les bit-flips, c'est-à-dire le passage d'un bit de 0 vers 1 ou inversement. Comme illustré par la Figure 1, où un bit est représenté par un système à 2 états : 0 ou 1. À gauche est représenté l'état initial du bit : initialisé à 1, et à droite son état après application du bruit engendrant une erreur : il passe dans l'état 0.

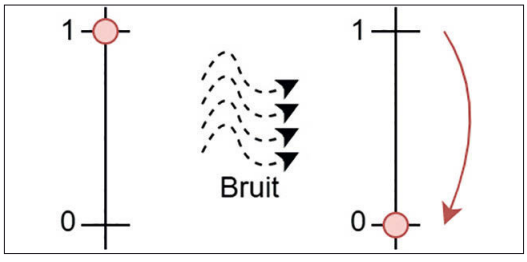


Figure 1 : *Illustration d'un bit-flip classique*

Erreur en informatique quantique

Pour rappel, en informatique quantique, un qubit s'exprime sous la forme :

$|\psi\rangle=\alpha|0\rangle+\beta|1\rangle$,

Avec α et β deux nombres complexes décrivant l'état du qubit tels que $|\alpha|^2 + |\beta|^2 = 1$. Cette représentation permet notamment de représenter l'état d'un qubit comme un point à la surface d'une sphère (appelée sphère de Bloch). Les erreurs quantiques peuvent alors être interprétées comme des rotations de cette sphère.

À partir de cette représentation, on distingue trois principaux types d'erreurs. La première est l'erreur X (appelée de manière abusive bit-flip), qui est l'équivalent quantique du bit-flip classique. Elle échange les états $|0\rangle$ et $|1\rangle$:

$X|\psi\rangle=\alpha|1\rangle+\beta|0\rangle$.

On remarque, comme illustré par la **Figure 2** qu'une erreur en X peut être vue comme une rotation de 180° autour de l'axe X.

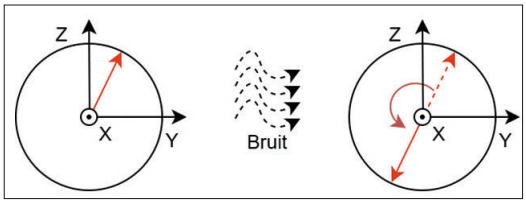


Figure 2 : *Illustration d'un bit-flip quantique*

La seconde est l'erreur Z, appelée phase-flip. Contrairement à l'erreur X, elle ne modifie pas les probabilités de mesurer 0 ou 1. Elle change uniquement la phase relative entre les deux états :

$Z|\psi\rangle= \alpha|0\rangle -\beta|1\rangle$.

De même, elle peut être vue comme une rotation de 180° autour de l'axe Z.

Enfin, l'erreur Y combine les deux effets précédents. Elle inverse les états tout en modifiant leur phase :

$Y|\psi\rangle=i\alpha|1\rangle -i\beta|0\rangle$.

En pratique, le facteur global i n'a pas d'influence sur les résultats d'une mesure. On peut donc retenir que l'erreur Y correspond à une combinaison d'un bit-flip et d'un phase-flip. De même, elle peut être vue comme une rotation de 180° autour de l'axe Y.

Ces trois opérateurs, X, Y et Z, appelés opérateurs de Pauli, constituent les erreurs élémentaires utilisées pour décrire les perturbations affectant un qubit.

Les codes correcteurs

Principe de redondance de l'information quantique

L'idée d'un code correcteur quantique est essentiellement la même que celle d'un code correcteur classique. On exprime un qubit à l'aide de plusieurs autres qubits de données. Cependant, la différence principale est que, comme évoqué précédemment, il est impossible de dupliquer l'information d'un qubit. L'idée n'est donc pas de copier l'information d'un qubit dans plusieurs autres, mais plutôt de "répartir" l'information d'un qubit sur plusieurs autres qubits.

Comme illustré par la **Figure 3**, l'information associée aux

trois qubits physiques (à droite) correspond à l'information du qubit logique (à gauche). Ainsi, manipuler les trois qubits physiques revient à manipuler l'information du qubit logique.

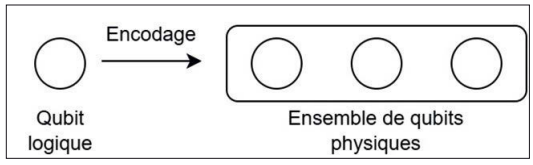


Figure 3 : *Répartition de l'information d'un qubit logique sur 3 qubits physiques.*

Mais attention, on ne manipule pas cette information de la même manière. Reprenons par exemple l'opérateur X défini précédemment, qui effectue un bit-flip. Sur un qubit mis dans l'état $|0\rangle$, on a, pour rappel, l'opération suivante : $X|0\rangle = |1\rangle$.
Cependant, maintenant que notre qubit logique est exprimé par 3 qubits physiques, il faut redéfinir l'ensemble des opérateurs logiques. Par exemple, si l'on décide arbitrairement d'exprimer l'état $|0\rangle$ de notre qubit logique par $|0\rangle_L = |000\rangle$ et l'état $|1\rangle$ de notre qubit logique par $|1\rangle_L = |111\rangle$, on peut redéfinir l'opérateur X logique comme étant l'application de l'opérateur X sur chacun des qubits : $X_L = X_1X_2X_3$.
On obtient alors bien : $X_L |0\rangle_L = |1\rangle_L$.
De plus, on peut déjà intuiter en quoi cette méthode permet de détecter et de corriger des erreurs. En effet, dans ce cas simple, mesurer les trois qubits et effectuer une moyenne des états obtenus nous permet de détecter une erreur.
Par exemple, si l'on a encodé l'état $|0\rangle_L = |000\rangle$ sur le processeur quantique, mais que l'on mesure, sans avoir effectué aucune opération, l'état $|010\rangle$, on peut en déduire par vote majoritaire que l'on observe l'état $|0\rangle_L$ (deux 0 contre un 1). Ainsi, même si un bit flip a eu lieu sur le deuxième qubit, on ne perd pas l'information associée au qubit logique global.
Cependant, d'autres cas sont également possibles : par exemple il peut y avoir eu deux bit flips sur les qubits 1 et 3, et l'état initial souhaité était en réalité $|1\rangle_L$. Dans ce cas, la mesure de $|010\rangle$ nous conduit à une mauvaise interprétation de l'état logique, ce qui correspond à une erreur logique.
Ce code, appelé code de répétition, constitue une première intuition, mais il ne protège que contre les erreurs de type X (et revient donc à un simple code de répétition classique). Un état quantique général subit également des erreurs de phase Z, ce qui nécessite des codes plus complets capables de corriger simultanément ces deux familles d'erreurs.

Principe de projection de l'information quantique

De plus comme évoqué en introduction, les qubits ne peuvent pas être observés sans être perturbés ce qui rend la correction d'erreurs particulièrement difficile. En effet, sa mesure détruit son état quantique : dès que l'on récupère l'information qu'il contient, son état est modifié. Comme l'illustre la **Figure 4,** l'observation d'un qubit provoque donc un changement de son état.

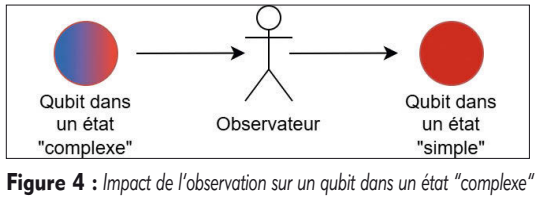


Figure 4 : *Impact de l'observation sur un qubit dans un état "complexe"*

Ainsi, si l'objectif est de corriger les erreurs sur un qubit, il faut trouver un moyen d'obtenir des informations sur son état sans le mesurer directement. Or, comme nous l'avons vu précédemment, il est également impossible de copier l'état d'un qubit afin de l'observer ailleurs.
Pour contourner cette difficulté, on utilise, comme illustré sur la **Figure 5,** des qubits auxiliaires, appelés qubits de syndrome (cercles gris). Ceux-ci sont intriqués avec les qubits de données (cercles blancs), l'intrication étant représentée par les traits reliant les qubits.

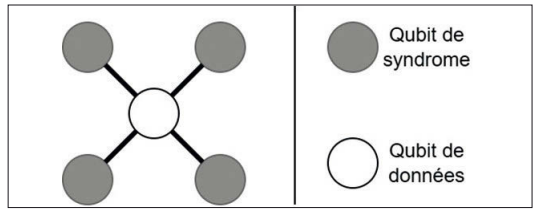


Figure 5 : *Principe de détection des erreurs quantiques*

Les qubits de syndrome sont ensuite mesurés. Cette mesure ne révèle pas l'état des qubits de données, mais fournit des informations sur la présence éventuelle d'une erreur. Chaque mesure réalisée à l'aide d'un qubit de syndrome correspond à un stabilisateur, c'est-à-dire une opération de mesure permettant de détecter une erreur affectant les qubits de données voisins, sans accéder directement à l'information portée par le qubit logique. On peut ainsi détecter les erreurs sans jamais mesurer directement le qubit qui porte l'information quantique. Il ne reste plus qu'ensuite à la corriger.

Exemple : code de surface

En combinant les deux principes présentés précédemment : la répartition de l'information d'un qubit sur plusieurs qubits et l'utilisation de qubits de syndrome, on peut maintenant introduire l'un des codes les plus emblématiques de la correction d'erreurs quantiques : le code de surface.
Un exemple de code de surface est présenté à la **Figure 6**. En conservant la même représentation que précédemment, les qubits blancs correspondent aux qubits de données, tandis que les qubits gris représentent les qubits de syndrome. Dans cet exemple, un unique qubit logique est encodé à l'aide de neuf qubits physiques (en blanc), tandis que huit qubits de syndrome (en gris) sont utilisés pour surveiller l'état de ces qubits de données.

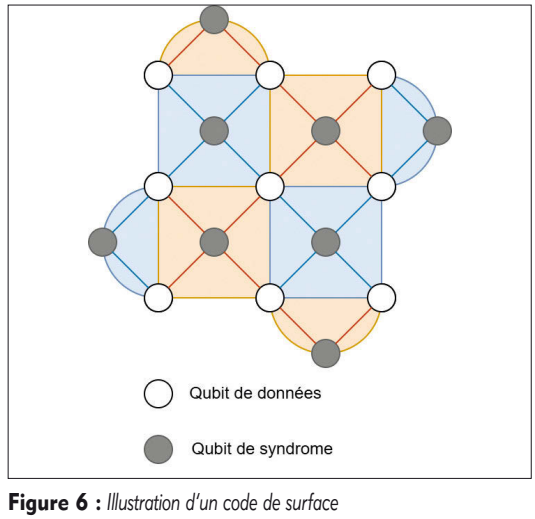


Figure 6 : *Illustration d'un code de surface*

Grâce à ce code, il est ainsi possible de manipuler un qubit logique en agissant sur les neuf qubits de données, tout en surveillant l'apparition d'éventuelles erreurs grâce aux mesures effectuées sur les huit qubits de syndrome.

Une particularité de ce code quantique, que l'on remarque immédiatement visuellement, est sa séparation en deux types de stabilisateurs, représentés en bleu et en orange. Cette distinction est propre au code de surface et permet de surveiller séparément les erreurs de type X et les erreurs de type Z.

Ainsi, les qubits de syndrome associés aux carrés orange détectent les erreurs de type X, tandis que ceux associés aux carrés bleus détectent les erreurs de type Z. De la même manière, les connexions orange indiquent les interactions avec les qubits de données dans la base X, alors que les connexions bleues correspondent aux interactions dans la base Z.

Enfin, rappelons qu'une erreur de type Y est équivalente à l'application successive d'une erreur X et d'une erreur Z. Une erreur Y provoque donc simultanément une signature sur les deux types de stabilisateurs : elle est détectée à la fois par les qubits de syndrome associés aux erreurs X et par ceux associés aux erreurs Z.

Décodeur et chaîne complète de correction d'erreurs.

Détecter une erreur en mesurant les qubits de syndrome ne suffit pas à la corriger. Une étape supplémentaire est donc nécessaire : le décodage.

À partir des mesures de syndrome obtenues, un algorithme de décodage est appliqué. L'objectif de cet algorithme est de déduire les erreurs affectant les qubits de données à partir des syndromes mesurés, puis de déterminer les corrections nécessaires afin de restaurer l'état logique initial.

L'objectif étant de concevoir des algorithmes suffisamment rapides pour suivre le rythme des mesures successives effectuées sur la machine quantique, tout en étant suffisamment précis pour limiter l'apparition d'erreurs logiques.

On peut maintenant représenter un processus complet de correction d'erreurs quantiques à l'aide de la **Figure 7**. Dans cet exemple (hypothétique), on part initialement d'un unique qubit logique. Celui-ci est ensuite encodé à l'aide d'un code correcteur hypothétique nécessitant trois qubits de données et deux qubits de syndrome. Le qubit de syndrome hachuré en rouge détecte une erreur sur le qubit de données, également hachuré en rouge.

Une fois l'encodage effectué sur la machine quantique, des cycles de correction sont réalisés de manière répétée. À intervalles réguliers, les qubits de syndrome sont mesurés afin d'obtenir les informations nécessaires sur les éventuelles erreurs présentes. Un algorithme de décodage utilise ensuite ces mesures pour inférer les erreurs affectant le qubit logique et déterminer les corrections à appliquer. Ces cycles de mesure, de décodage et de correction sont alors répétés tout au long de l'exécution du code quantique.

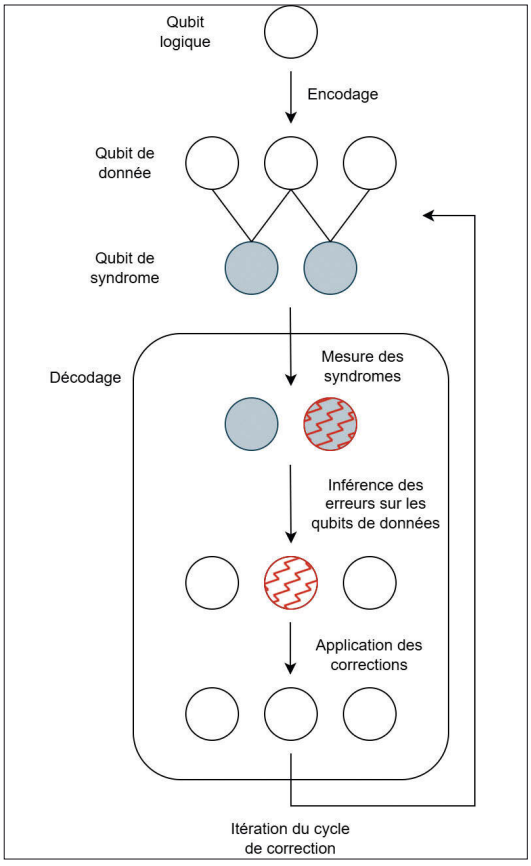


Figure 7 : *Processus de décodage*

Conclusion et recherche associée

Cette brève présentation avait pour objectif d'introduire les fondements de la correction d'erreurs quantiques. Pour aller plus loin, on peut se demander comment réaliser des calculs quantiques impliquant un grand nombre de qubits logiques. Deux approches principales existent. La première consiste à utiliser plusieurs qubits logiques encodés séparément. Par exemple, deux qubits logiques peuvent être représentés par deux codes de surface distincts. Il faut alors déterminer comment réaliser des opérations logiques et des interactions entre ces deux codes afin de permettre l'exécution d'un calcul quantique complet.

La seconde approche consiste à utiliser des codes correcteurs plus complexes, capables d'encoder plusieurs qubits logiques au sein d'un même ensemble de qubits physiques. Dans cette présentation, seuls des codes permettant d'encoder un unique qubit logique ont été introduits, mais il existe également des codes capables de protéger simultanément plusieurs qubits logiques.

Pour conclure, voilà trois axes de recherche actifs :

- Concevoir des codes correcteurs efficaces, capables d'encoder un grand nombre de qubits logiques tout en minimisant le nombre de qubits physiques nécessaires.
- Développer des algorithmes de décodage rapides et performants, capables d'interpréter les syndromes en temps réel et de corriger efficacement les erreurs.
- Implémenter physiquement ces codes sur les architectures quantiques actuelles et futures, en tenant compte des contraintes propres aux différentes technologies quantiques.

Hybridation des calculs quantique et classique

Les machines quantiques actuelles, qu'il s'agisse de calculateurs analogiques comme ceux proposés par Pasqal, D-Wave, QuEra ou Qilimanjaro, ou de processeurs quantiques à portes comme ceux développés par IBM, Google ou Quandela, sont encore loin de disposer d'un avantage général sur les ordinateurs classiques.



Samuel DELEPLANQUE

Junia, Univ. Lille, CNRS, Centrale Lille, Univ. Polytechnique Hauts-de-France, UMR 8520 - IEMN , 41 Bd Vauban, Lille, 59000, France

samuel.deleplanque@junia.com



Luis Fernando PÉREZ ARMAS

IESEG School of Management, Univ. Lille, CNRS, UMR 9221 - LEM - Lille Économie Management, 3 rue de la digue, Lille, 59800, Nord, France

l.perezarmas@ieseg.fr

Cela signifie qu'un processeur quantique ne permet pas aujourd'hui de résoudre n'importe quel problème plus rapidement, avec une meilleure solution ou en utilisant moins de ressources qu'un ordinateur classique. Les performances dépendent fortement du problème considéré, de sa formulation mathématique, de la méthode d'encodage utilisée, du nombre de qubits disponibles, de leur connectivité et du niveau de bruit de la machine.

Dans le domaine de l'optimisation combinatoire et de la recherche opérationnelle, ces machines présentent néanmoins un intérêt croissant lorsqu'elles sont intégrées dans une architecture hybride. L'objectif n'est pas de remplacer le processeur classique, mais de construire une chaîne de calcul dans laquelle chaque technologie réalise les opérations auxquelles elle est la mieux adaptée.

Pour un développeur, cette approche peut être comparée à l'utilisation d'un accélérateur spécialisé. Un processeur graphique ne remplace pas complètement un processeur central. Il prend en charge certaines opérations très parallèles, tandis que le processeur classique conserve la gestion du programme, de la mémoire, des entrées et sorties et de la logique générale. De la même manière, un processeur quantique peut être vu comme un composant spécialisé, appelé ponctuellement par un programme classique pour traiter une partie bien identifiée d'un problème.

Dans une application hybride, le code principal reste donc généralement exécuté sur un ordinateur classique. Il prépare les données, construit le modèle mathématique, choisit les paramètres, appelle la machine quantique par l'intermédiaire d'une bibliothèque ou d'une API, récupère les résultats, puis les analyse. Le processeur quantique intervient comme un service de calcul intégré dans un pipeline plus large.

Cette organisation est particulièrement naturelle pour les problèmes comportant des décisions discrètes. Choisir ou non une ressource, affecter une tâche à une machine, sélectionner un itinéraire, activer une installation ou ordonnancer une opération sont des décisions qui peuvent être représentées par des variables binaires. Ces variables prennent la valeur 0 ou 1 et peuvent être associées, dans certains modèles, à des qubits.

Cette correspondance n'est toutefois pas toujours directe au niveau matériel. Une variable binaire du modèle peut nécessiter plusieurs qubits physiques, notamment lorsque la connectivité de la machine impose un encodage supplémentaire. Il est donc important de distinguer les variables logiques utilisées par le développeur des qubits réellement mobilisés sur le processeur.

À l'inverse, les calculs sur des variables réelles, les opérations matricielles classiques, la manipulation de grands volumes de données, la gestion de la mémoire et l'exécution de règles métier restent généralement plus simples et plus efficaces sur des architectures classiques. Les bibliothèques numériques, les solveurs d'optimisation, les systèmes HPC et les infrastructures cloud disposent déjà d'outils très performants pour ce type de traitement.

La première règle d'une architecture hybride consiste donc à laisser à chaque machine ce qu'elle sait déjà bien faire.

Dans de nombreux problèmes NP-complets ou NP-difficiles, la recherche d'une bonne solution peut être extrêmement coûteuse. Le nombre de combinaisons possibles augmente très rapidement avec la taille du problème. En revanche, une fois qu'une solution candidate a été produite, il est souvent beaucoup plus simple de vérifier si elle respecte les contraintes.

Prenons l'exemple d'un problème d'ordonnancement. Trouver une affectation optimale des tâches, des machines et des dates peut nécessiter l'exploration d'un très grand espace de solutions. En revanche, vérifier qu'une solution ne contient pas de chevauchement, respecte les précédences et ne dépasse pas les capacités disponibles est une opération relativement directe.

Dans une approche hybride, le processeur quantique peut donc être utilisé pour générer ou explorer des solutions candidates. Le programme classique se charge ensuite de les décoder, de les valider, de calculer leur coût et, si nécessaire, de les corriger.

Le pipeline de résolution peut, par exemple, être organisé de la manière suivante :

1. Lecture et nettoyage des données du problème.
2. Construction d'un modèle d'optimisation.
3. Transformation d'une partie du modèle en formulation binaire, par exemple sous la forme d'un QUBO ou d'un modèle d'Ising.
4. Envoi du sous-problème au processeur quantique.
5. Récupération d'un ensemble de solutions candidates.
6. Vérification des contraintes sur le processeur classique.
7. Correction éventuelle des solutions non réalisables.
8. Évaluation et classement des solutions.
9. Réutilisation des meilleures solutions dans une nouvelle itération.

Le calcul hybride ne correspond donc pas nécessairement à un seul appel quantique. Il peut prendre la forme d'une boucle dans laquelle le programme classique ajuste progressivement les paramètres du problème.

Le processeur quantique produit un ensemble de résultats, puis l'ordinateur classique analyse ces résultats et modifie la formulation, les pénalités, les variables, les bornes ou les paramètres de l'algorithme. Un nouvel appel quantique est ensuite effectué. Le processus se poursuit jusqu'à l'obtention d'une solution satisfaisante ou jusqu'à l'atteinte d'un critère d'arrêt.

Cette architecture ressemble à celle de nombreux algorithmes déjà utilisés en apprentissage automatique ou en optimisation. Une partie du calcul sert à produire une direction de recherche ou un ensemble de candidats, tandis qu'une autre partie évalue les résultats et pilote l'itération suivante.

Il est également possible de décomposer le problème initial en plusieurs sous-problèmes. Certains peuvent être traités efficacement par un solveur classique, tandis que d'autres sont confiés au processeur quantique. Le choix peut dépendre de la taille des sous-problèmes, de leur structure, de leur densité, du nombre de variables binaires ou encore des contraintes du matériel disponible.

Un problème de transport peut, par exemple, être séparé en plusieurs étapes. Une méthode classique peut regrouper les clients par zone géographique. Un processeur quantique peut ensuite chercher un itinéraire pour chaque groupe. Enfin, un programme classique peut reconstruire la solution complète, vérifier les capacités des véhicules et améliorer localement les tournées obtenues.

Le calcul quantique peut aussi être combiné avec une infrastructure HPC. Certaines machines quantiques ou certains solveurs quantiques peuvent retourner un grand nombre d'échantillons correspondant à différentes solutions candidates. Ces résultats peuvent ensuite être traités en parallèle par plusieurs cœurs, plusieurs processeurs ou plusieurs nœuds de calcul.

Chaque nœud HPC peut recevoir une partie des solutions produites, vérifier leur faisabilité, appliquer une heuristique d'amélioration locale ou calculer des indicateurs supplémentaires. Le calcul quantique sert alors à explorer l'espace des solutions, tandis que l'infrastructure classique absorbe et traite en parallèle le volume de résultats obtenu.
Il faut cependant distinguer le temps de calcul interne de la machine quantique du temps total observé par l'application. Pour un développeur, les performances doivent être mesurées de bout en bout. Il faut prendre en compte la préparation du modèle, les communications réseau, l'attente éventuelle dans une file d'exécution, l'encodage, le décodage et le post-traitement des résultats.

Une architecture hybride efficace ne consiste donc pas simplement à ajouter un appel à une machine quantique dans un programme existant. Elle suppose de réfléchir à la répartition des responsabilités entre les différents composants, aux formats de données échangés, aux coûts de communication et à la manière dont les résultats quantiques seront exploités par le reste de l'application.

L'objectif de cet article est de montrer, à travers plusieurs problèmes d'optimisation et de recherche opérationnelle, comment construire ce type d'algorithmes hybrides. Nous verrons comment préparer un problème, identifier la partie qui peut être confiée à un processeur quantique, organiser les échanges avec le programme classique et mettre en place les étapes de validation, de correction et d'amélioration des solutions.

La question n'est donc pas de savoir si le quantique doit remplacer le classique. Elle est plutôt de déterminer comment faire coopérer processeurs classiques, processeurs quantiques et infrastructures de calcul haute performance afin de construire, dès aujourd'hui, des méthodes de résolution plus souples et mieux adaptées aux caractéristiques réelles des machines disponibles.

Application

Décomposer pour mieux répartir les calculs

De nombreux problèmes d'optimisation combinatoire et de recherche opérationnelle appartiennent à des classes de complexité difficiles. Plus précisément, les versions décisionnelles de certains problèmes sont dites NP-complètes, tandis que leurs versions d'optimisation sont généralement qualifiées de NP-difficiles.

Sans entrer dans les détails de la théorie de la complexité, cela signifie que le nombre de solutions possibles peut augmenter très rapidement avec la taille des données. Ajouter quelques usines, quelques véhicules, quelques tâches ou quelques périodes peut suffire à multiplier considérablement le nombre de combinaisons à examiner. Pour les instances de grande taille, il devient alors difficile de garantir l'obtention d'une solution optimale dans un temps de calcul acceptable.

L'une des stratégies classiques consiste à décomposer le problème initial en plusieurs sous-problèmes plus simples. Plutôt que de demander à un unique algorithme de traiter simultanément toutes les variables et toutes les contraintes, on sépare les différentes catégories de décisions afin de les résoudre successivement ou de manière coordonnée.

Parmi ces méthodes, le schéma maître-esclave est particulièrement intéressant. Le problème maître traite les décisions structurantes. Sa solution permet ensuite de construire un ou plusieurs problèmes esclaves, chargés de compléter, d'évaluer ou de valider les décisions prises au niveau supérieur.

Le problème maître peut, par exemple, déterminer quelles usines doivent être ouvertes, quelles productions doivent être activées ou quelles ressources doivent être sélectionnées. Une fois ces décisions fixées, le problème esclave détermine les quantités à produire, les flux à transporter, les dates d'exécution ou l'affectation détaillée des ressources.

À l'issue de la résolution du problème esclave, les différentes décisions sont regroupées pour reconstruire une solution complète du problème initial. Lorsque cette solution n'est pas réalisable ou n'est pas suffisamment performante, les informations obtenues peuvent également être renvoyées au problème maître. Le processus prend alors la forme d'une boucle dans laquelle les deux niveaux de résolution s'échangent progressivement des informations.

Dans l'architecture hybride considérée ici, le processeur quantique est placé au niveau du problème maître. Ce choix est motivé par la nature généralement discrète et combinatoire des décisions prises à ce niveau. Les variables du problème maître correspondent souvent à des choix binaires, tels que sélectionner ou non une ressource, ouvrir ou non une installation ou activer ou non une possibilité de production.

Le processeur quantique est alors utilisé pour produire plusieurs configurations possibles de ces variables. Chaque configuration définit un scénario différent à transmettre à la partie classique de l'algorithme.

Le problème esclave reste, quant à lui, résolu sur une architecture classique. Dans le cas le plus favorable, il appartient à une classe de problèmes pouvant être résolue efficacement en temps polynomial, comme certains problèmes de flot, de plus court chemin ou de programmation linéaire. Dans d'autres cas, il peut lui-même rester difficile, mais correspondre à un problème largement étudié pour lequel des solveurs spécialisés, des heuristiques ou des algorithmes parallèles performants sont déjà disponibles.

Cette répartition permet de ne pas demander au processeur quantique de résoudre l'intégralité du problème. Il se concentre sur la production de décisions combinatoires, tandis que l'infrastructure classique prend en charge les calculs numériques, la vérification des contraintes, l'évaluation des coûts et la construction de la solution complète.

Pour cette partie classique, nous considérons une infrastructure de calcul haute performance, ou HPC pour High Performance Computing. Un système HPC peut regrouper un grand nombre de processeurs, de cœurs et de nœuds de calcul capables de traiter simultanément des tâches indépendantes.

Cette capacité de parallélisation s'accorde naturellement avec le fonctionnement d'un processeur quantique capable de retourner un ensemble de solutions candidates. Supposons, par exemple, que la machine quantique produise plusieurs milliers de configurations pour le problème maître. Chacune d'elles peut servir à construire un problème esclave indépendant.

Ces sous-problèmes peuvent alors être distribués sur les ressources du cluster HPC. Chaque nœud, ou chaque cœur disponible, reçoit une configuration, construit le problème classique correspondant, tente de le résoudre, vérifie sa faisabilité et calcule son coût.

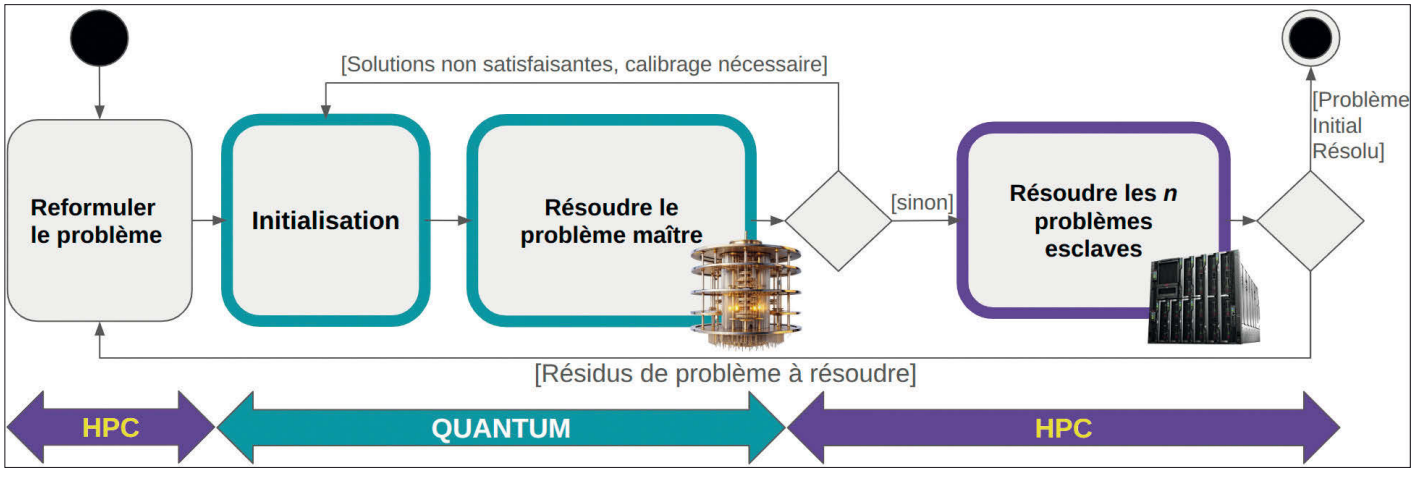
Dans une représentation simplifiée, une solution quantique peut être associée à un nœud de calcul HPC. En pratique, le système fonctionne plutôt à l'aide d'une file de tâches. Les solutions candidates sont placées dans cette file, puis distribuées dynamiquement entre les ressources disponibles. Un même nœud peut ainsi traiter plusieurs solutions successivement ou en parallèle, selon sa puissance et le coût du problème esclave.

L'intérêt de cette architecture est de transformer la diversité des résultats quantiques en parallélisme classique. Plutôt que de ne conserver que la meilleure solution indiquée par la machine quantique, le système peut explorer simultanément un grand nombre de scénarios, puis comparer les solutions complètes obtenues.

La **Figure 1** illustre ce fonctionnement. Le processeur quantique résout le problème maître et génère un ensemble de configurations. Ces configurations sont ensuite décodées, filtrées et distribuées sur le cluster HPC. Chaque unité de calcul résout le problème esclave correspondant. Les résultats sont enfin rassemblés afin de sélectionner les solutions les plus intéressantes.

Les architectures de calcul haute performance commencent par ailleurs à intégrer des accélérateurs quantiques ou à prévoir leur accès comme un service de calcul supplémentaire. Cette intégration soulève plusieurs difficultés pour les développeurs : communication entre les composants, formats de données, temps de transfert, ordonnancement des tâches, gestion des erreurs, synchronisation des calculs et agrégation des résultats.

Figure 1



Les algorithmes hybrides présentés ici s'inscrivent précisément dans cette logique. Ils ne considèrent pas le processeur quantique comme une machine isolée destinée à remplacer l'ordinateur classique. Ils cherchent au contraire à exploiter l'ensemble de l'architecture de calcul en répartissant les opérations entre un processeur quantique, un programme de coordination classique et une infrastructure HPC massivement parallèle.

Du Lot-Sizing classique à une résolution hybride quantique et HPC

Pour comprendre l'intérêt de cette hybridation, repartons d'un problème classique de recherche opérationnelle : le Lot-Sizing multi-usines.

Une entreprise doit décider où, quand et en quelle quantité fabriquer plusieurs produits. Elle peut disposer de plusieurs usines, de capacités de production limitées, de zones de stockage et de moyens de transport permettant de déplacer les produits d'un site à un autre. Chaque décision entraîne un coût. Lancer la production d'un produit sur une usine peut, par exemple, imposer un coût fixe de préparation. À cela s'ajoutent les coûts proportionnels aux quantités produites, stockées ou transférées.

Le problème consiste donc à satisfaire toutes les demandes tout en minimisant le coût global. Il faut simultanément décider :

- quelles productions doivent être activées ;
- quelles quantités doivent être produites ;
- où les produits doivent être stockés ;
- comment ils doivent être transférés entre les usines ;
- comment respecter les capacités de production, de stockage et de transport.

Le nombre de combinaisons possibles augmente très rapidement avec le nombre de produits, d'usines et de périodes. Ce type de problème devient donc difficile à résoudre lorsque les instances prennent une taille industrielle.

Dans notre travail initial, le problème était reformulé sous la forme d'un réseau de flux multi-produits. Les décisions étaient ensuite séparées en deux parties. Un problème maître de localisation déterminait les productions qu'il était intéressant d'activer. Un problème esclave de flot à coût minimal déterminait ensuite les quantités à produire, à stocker et à transporter. Cette décomposition était intégrée dans une heuristique lagrangienne destinée à gérer les contraintes de capacité.

Cette séparation est particulièrement intéressante dans une perspective de calcul hybride. En effet, les deux sous-problèmes ne manipulent pas le même type de décisions.

Le problème maître travaille principalement avec des décisions binaires. Pour chaque produit, chaque usine et chaque période, il faut décider si une possibilité de production est activée ou non. Une variable vaut 1 si la production est ouverte et 0 dans le cas contraire. Ce problème peut être rapproché d'un problème de localisation non capacitaire, souvent désigné par UFLP, pour Uncapacitated Facility Location Problem.

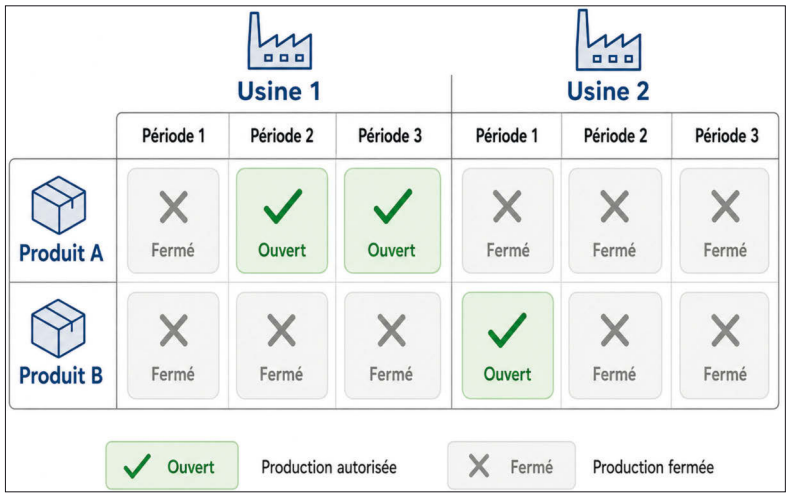


Figure 2

Le problème esclave travaille plutôt avec des quantités. Une fois les productions autorisées, il faut déterminer combien produire, quelles demandes alimenter, quelles routes de transfert utiliser et quelles quantités conserver en stock. Cette partie correspond à un problème de flot multi-produits, pour lequel les méthodes classiques de programmation linéaire et de recherche opérationnelle sont particulièrement efficaces (le problème n'est ni NP-Complet, ni NP-Difficile).

L'idée de la nouvelle approche consiste donc à confier le problème maître à un processeur quantique et à conserver le problème de flot sur une infrastructure classique de calcul haute performance.

Le processeur quantique comme générateur de scénarios

Le rôle du processeur quantique n'est pas nécessairement de fournir directement la solution complète du problème de Lot-Sizing. Il est utilisé comme un générateur de configurations candidates.

Une configuration indique, par exemple :

- que le produit A peut être fabriqué dans l'usine 1 pendant les périodes 2 et 3 ;
- que le produit B peut être fabriqué dans l'usine 2 pendant la période 1 ;
- que certaines autres possibilités de production restent fermées.

Figure 2

Cette configuration ne précise pas encore les quantités exactes à produire ou à transporter. Elle définit uniquement la structure générale de la solution, c'est-à-dire les décisions d'ouverture et de fermeture des différentes possibilités de production.

Le problème maître peut être converti en une formulation binaire adaptée au calcul quantique, par exemple sous la forme d'un QUBO ou d'un modèle d'Ising. Le processeur quantique retourne alors non pas une unique réponse, mais un ensemble d'échantillons. Chaque échantillon représente une configuration possible des décisions binaires.

Dans un fonctionnement classique, un solveur cherche souvent à converger vers une seule solution, éventuellement accompagnée

de quelques solutions alternatives. Une machine quantique, notamment une machine d'optimisation analogique, peut produire rapidement un grand nombre de configurations différentes. Certaines seront de bonne qualité, d'autres seront moins intéressantes et certaines pourront ne pas respecter toutes les contraintes. Cette diversité constitue précisément la matière première du traitement HPC.

Une solution candidate devient une tâche de calcul

Chaque configuration produite par le processeur quantique peut être envoyée à une unité de calcul classique indépendante. Pour une configuration donnée, les variables binaires du problème maître sont fixées. Le problème restant consiste alors à déterminer les quantités de production, de stockage et de transfert. Le nœud de calcul HPC résout le problème de flot correspondant et répond à plusieurs questions :

1. Cette configuration permet-elle de satisfaire toutes les demandes ?
2. Les capacités de production, de stockage et de transport sont-elles respectées ?
3. Quel est le coût total de la solution ?
4. Est-il possible de réparer la configuration lorsqu'elle est presque réalisable ?
5. Peut-on améliorer localement la solution obtenue ?

Les différentes configurations peuvent être évaluées indépendamment. Cette propriété rend le calcul particulièrement adapté à une infrastructure parallèle.

Le fonctionnement général ressemble à une architecture de type "fan-out/fan-in". Dans la phase de "fan-out", le programme distribue les configurations entre les ressources de calcul disponibles. Dans la phase de "fan-in", il rassemble les résultats, compare les coûts et conserve les meilleures solutions.

Conceptuellement, on peut imaginer une solution quantique traitée par un nœud HPC. En pratique, il est souvent plus efficace de considérer chaque solution comme une tâche plutôt que de réserver un nœud complet à chaque échantillon. Un ordonnanceur distribue alors les tâches sur les nœuds disponibles. Un même nœud peut traiter successivement plusieurs configurations ou en évaluer plusieurs en parallèle sur ses différents cœurs.

Cette distinction est importante, car une machine quantique peut produire des centaines ou des milliers d'échantillons, alors qu'un cluster ne dispose pas nécessairement d'autant de nœuds. Le système doit donc gérer une file de tâches, regrouper certains calculs et équilibrer la charge entre les ressources.

Une chaîne de calcul hybride

Du point de vue du développeur, la méthode peut être organisée sous la forme du pipeline suivant.

1 Préparation classique du problème

Le programme classique lit les données industrielles : demandes, coûts, capacités, périodes, produits et usines. Il élimine également les décisions manifestement impossibles ou inutiles. Par exemple, il peut supprimer une possibilité de production lorsque l'usine concernée ne dispose d'aucune capacité pendant la période considérée.

2 Construction du problème maître

Les décisions binaires sont extraites du modèle complet. Elles représentent les productions susceptibles d'être activées. Le programme construit ensuite le modèle destiné au processeur quantique. Les coûts fixes, les contraintes et les informations issues de la relaxation lagrangienne sont transformés en coefficients du QUBO ou du modèle d'Ising.

3 Échantillonnage quantique

Le problème maître est envoyé à la machine quantique. Celle-ci retourne un ensemble de configurations candidates. Le programme peut demander un grand nombre d'échantillons afin d'obtenir non seulement la meilleure configuration trouvée, mais également des solutions proches présentant des structures différentes.

4 Décodage et filtrage

Les résultats quantiques doivent être convertis vers les variables du problème métier. Le programme classique peut alors supprimer les doublons, rejeter les configurations manifestement impossibles et effectuer des vérifications peu coûteuses avant de solliciter le cluster HPC. Cette étape évite, par exemple, de lancer plusieurs calculs identiques pour une configuration apparue des dizaines de fois dans les échantillons quantiques.

5 Distribution sur le cluster HPC

Chaque configuration restante devient une tâche indépendante. Elle est placée dans une file de calcul, puis affectée à un nœud disponible. Le nœud fixe les décisions du problème maître et résout le problème esclave de flot multi-produits. Cette résolution peut s'appuyer sur un solveur de programmation linéaire, un algorithme spécialisé de flot ou une heuristique adaptée aux grandes instances.

6 Validation et réparation

Une configuration quantique peut conduire à un problème de flot irréalisable. Les productions ouvertes peuvent, par exemple, être insuffisantes pour couvrir toutes les demandes. Le nœud classique peut alors soit rejeter cette configuration, soit tenter de la réparer. Une réparation peut consister à ouvrir une production supplémentaire, à déplacer une production vers une autre période ou à modifier certaines affectations. Le calcul classique joue donc le rôle de garant de la faisabilité. Il transforme une proposition binaire en un plan industriel complet et vérifiable.

7 Agrégation des résultats

Lorsque les tâches sont terminées, un programme coordinateur rassemble les résultats. Les solutions réalisables sont classées selon leur coût, leur robustesse ou d'autres critères industriels. Le système peut conserver une seule solution, un ensemble de solutions diversifiées ou plusieurs compromis entre coût, stock et utilisation des capacités.

Une boucle plutôt qu'un unique appel quantique

L'approche ne doit pas nécessairement s'arrêter après un seul passage. Les résultats obtenus par le cluster peuvent être utilisés pour modifier le problème maître. Si certaines configurations conduisent régulièrement à une saturation des capacités, le programme peut augmenter les pénalités correspondantes. Si certaines décisions apparaissent fréquemment dans les meilleures solutions, elles peuvent être favorisées lors de l'itération suivante.

Le processus devient alors une boucle hybride :

1. le processeur quantique propose des configurations ;
2. le cluster classique les évalue ;
3. le programme analyse les résultats ;
4. les paramètres du problème maître sont ajustés ;
5. une nouvelle série de configurations est produite.

Pourquoi utiliser le HPC ?

Le rôle du HPC ne se limite pas à accélérer un calcul déjà existant. Il permet d'exploiter une caractéristique importante de l'échantillonnage quantique : la production d'un grand nombre de solutions candidates. Au lieu de sélectionner immédiatement une seule solution quantique, l'infrastructure classique peut évaluer simultanément de nombreuses hypothèses. Chaque nœud explore une branche différente du problème.

Cette organisation présente plusieurs avantages :

- elle utilise la diversité des solutions retournées par la machine quantique ;
- elle permet de vérifier rigoureusement la faisabilité de chaque solution ;
- elle répartit les traitements classiques coûteux sur plusieurs ressources ;
- elle permet d'appliquer différentes stratégies de réparation ou d'amélioration ;
- elle conserve les solveurs classiques pour les calculs qu'ils maîtrisent déjà efficacement.

Le processeur quantique explore les décisions combinatoires. Le cluster HPC traite les flux, les quantités et les contraintes industrielles. Le programme classique coordonne l'ensemble.

Pour un développeur, cette architecture peut être comparée à l'utilisation d'un GPU ou d'un service spécialisé. L'application principale reste classique. Elle gère les données, les communications, la logique métier et le contrôle de l'algorithme. Le processeur quantique intervient ponctuellement comme un accélérateur chargé de produire des configurations binaires. Le cluster HPC prend ensuite en charge l'évaluation massive de ces configurations.

La machine quantique n'est donc pas utilisée comme un ordinateur autonome. Elle devient un composant d'une architecture distribuée plus large, au même titre qu'un solveur, une base de données ou un accélérateur matériel.

L'intérêt de cette démarche n'est pas de supposer que le calcul quantique surpasse déjà les meilleures méthodes classiques pour résoudre le problème complet. L'objectif est plutôt d'étudier une répartition cohérente des traitements entre plusieurs technologies.

Le problème maître, discret et combinatoire, constitue un candidat naturel pour une expérimentation quantique. Le problème esclave, continu et fortement structuré, reste traité par les méthodes classiques. Le HPC permet enfin d'absorber et d'évaluer en parallèle le grand nombre de solutions proposées.

Cette architecture cherche ainsi à exploiter les capacités complémentaires du quantique et du classique, plutôt qu'à les mettre en concurrence.

Un peu de maths et un peu de code…

Avant de voir comment configurer et interroger une machine quantique à l'aide d'un programme Python servant d'interface, il faut d'abord déterminer quel modèle mathématique lui transmettre. Les machines d'optimisation quantique travaillent généralement avec des formulations de type Ising ou QUBO.
Contrairement aux modèles utilisés par les solveurs classiques, ces formulations ne comportent pas directement de contraintes. Celles-ci doivent être intégrées à la fonction Objectif sous la forme de termes de pénalité. Les variables sont également limitées à deux états : (0) ou (1) dans un modèle QUBO, et (-1) ou (+1) dans un modèle d'Ising.
En contrepartie, ces modèles autorisent des termes quadratiques, c'est-à-dire des produits entre deux variables. Ces interactions permettent de représenter les dépendances entre décisions et d'intégrer certaines contraintes dans la fonction à minimiser. Le travail de modélisation consiste donc à transformer le problème initial en une unique expression quadratique, compatible avec la machine quantique.

Notations du problème UFLP

Le problème UFLP consiste à choisir les installations à ouvrir, puis à affecter chaque client à une installation ouverte, de manière à minimiser les coûts d'ouverture et d'affectation.
Ensembles et paramètres
C : ensemble des clients.
F : ensemble des installations potentielles.
f_j : coût fixe associé à l'ouverture de l'installation $j \in F$.
c_{ij} : coût d'affectation, ou de connexion, du client $i \in C$ à l'installation $j \in F$.
Variables de décision
$x_{ij} \in \{0, 1\}$: vaut 1 si le client i est affecté à l'installation j, et 0 sinon.
$y_j \in \{0, 1\}$: vaut 1 si l'installation j est ouverte, et 0 sinon.

Dans le cas d'une machine D-Wave, le code reste assez proche de celui utilisé avec d'autres plateformes acceptant des modèles QUBO ou Ising. Les ensembles, les paramètres et les variables nécessaires peuvent être déclarés à partir des données contenues dans une structure représentant l'instance du problème, comme dans l'exemple suivant :

```
x = {i : f'x_{i}' for i in range(num_facilities)}
y = {(i, j) : f'y_{i}_{j}' for i in range(num_facilities) for j in range(num
_customers)}

num_facilities = instance['num_facilities']
num_customers = instance['num_customers']
fixed_costs = instance['fixed_costs']
```

```
transportation_costs = instance['transportation_costs']
```

La structure du QUBO est définie par :

```
bqm = dimod.BinaryQuadraticModel('BINARY')
```

La fonction objectif du problème UFLP s'écrit :

$$f_{obj}(x, y) = \sum_{i \in C} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j$$

Le premier terme représente le coût d'affectation des clients aux installations. Le second terme correspond au coût d'ouverture des installations. Voici le code correspondant :

```
for i in range(num_facilities):
    bqm.add_variable(x[i], fixed_costs[i])
    for j in range(num_customers):
        bqm.add_variable(y[i, j], transportation_costs[i][j])
```

La contrainte imposant que chaque client soit affecté à une seule installation est intégrée sous la forme d'une pénalité quadratique :

$$P_{assign}(x) = \lambda_1 \sum_{i \in C} \left(\sum_{j \in F} x_{ij} - 1\right)^2$$

Cette pénalité est nulle lorsque chaque client *i* est affecté à exactement une installation. Le code est le suivant :

```
for j in range(num_customers):
    max_value = max(row[j] for row in transportation_costs) + 1
    bqm.add_linear_equality_constraint(
        [(y[i, j], 1) for i in range(num_facilities)],
        constant=-1,
        lagrange_multiplier=max_value
    )
```

La contrainte empêchant l'affectation d'un client à une installation fermée est représentée par :

$$P_{open}(x, y) = \lambda_2 \sum_{i \in C} \sum_{j \in F} x_{ij}(1 - y_j)$$

Ce terme ajoute une pénalité lorsque $x_{ij} = 1$ alors que l'installation *j* est fermée, c'est-à-dire lorsque $y_j = 0$.
On intègre cette dernière expression :

```
for i in range(num_facilities):
    max_value = fixed_costs[i] + 1
    for j in range(num_customers):
        bqm.add_linear(y[i, j], max_value)
        bqm.add_quadratic(y[i, j], x[i], -max_value)
```

La formulation QUBO complète est donc :

$$f_{QUBO}(x, y) = \sum_{i \in C} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j + \lambda_1 \sum_{i \in C} \left(\sum_{j \in F} x_{ij} - 1\right)^2 + \lambda_2 \sum_{i \in C} \sum_{j \in F} x_{ij}(1 - y_j)$$

Les paramètres λ_1 et λ_2 sont des coefficients de pénalité positifs. Ils doivent être suffisamment élevés pour favoriser le respect des contraintes, sans toutefois écraser complètement les coûts du problème initial.
Voici maintenant les quelques lignes de code nécessaires pour transmettre le modèle QUBO à la machine quantique, lancer son exécution et récupérer les solutions obtenues. Ici la machine va sélectionner la machine et va faire tourner son processus NumberOfReads nombre de fois. La meilleure solution peut être récupérée par sampleset.first.sample, même si, dans notre cas, elles seront toutes connectées à un nœud de calcul du HPC.

```
dwave_sampler = DWaveSampler(token="le_Votre_:)")
sampler = EmbeddingComposite(dwave_sampler)
sampleset = sampler.sample(bqm, num_reads=NumberOfReads)
best_sample = sampleset.first.sample
```

Un aperçu de l'ensemble des solutions retournées peut être affiché à l'aide des méthodes fournies par l'API. Dans la **figure 3**, six solutions distinctes ont été obtenues, présentées des lignes 0 à 5. Chaque colonne correspond à une variable du modèle et indique la valeur qu'elle prend dans la solution considérée.
La colonne energy représente la valeur de la fonction objectif du QUBO. Comme le problème est formulé sous la forme d'une minimisation, plus cette valeur est faible, meilleure est la solution. Ici, la meilleure solution possède une énergie égale à 25.
La dernière colonne, num_oc., indique le nombre de fois où chaque solution a été observée. Sur les dix exécutions demandées, avec NumberOfReads = 10, la meilleure solution a été obtenue quatre fois. Cette information permet d'évaluer non seulement la qualité des solutions, mais aussi leur fréquence d'apparition.

Références

Deleplanque, S., Kedad-Sidhoum, S., & Quilliot, A. (2013). Lagrangean heuristic for a multi-plant lot-sizing problem with transfer and storage capacities. RAIRO-Operations Research, 47(4), 429-443.
Perez Armas, L. F., Deleplanque, S., Aggoune, R., & Creemers, S. (2025). A Hybrid Column Generation-Based Heuristic for Solving the Parallel Machine Scheduling Problem with Sequence-Dependent Setup Times. Philosophical transactions. Series A, Mathematical, physical, and engineering sciences.

Figure 3

```
(ocean) => python FacilityLocationSquaringTechnique.py
{'num_facilities': 2, 'num_customers': 4, 'fixed_costs': [10, 15], 'transportation_costs': [[5, 6, 7, 8], [4, 3, 2, 1]
]}
  slack_Cons2_0 x_0 x_1 y_0_0 y_0_1 y_0_2 y_0_3 ... y_1_3 energy num_oc. ...
0             0   0   1     0     0     0     0 ...     1   25.0       4 ...
1             0   0   0     0     0     0     0 ...     0   30.0       1 ...
2             0   1   0     1     1     1     1 ...     0   36.0       2 ...
3             0   0   0     0     0     0     0 ...     0   40.0       1 ...
4             0   0   0     0     1     0     0 ...     0   40.0       1 ...
5             0   1   0     1     0     1     1 ...     0   48.0       1 ...
['BINARY', 6 rows, 10 samples, 11 variables]
Number of logical variables:                   11
Number of physical qubits used after embedding:  12

Solution: {'facilities_opened': [1], 'customer_assignments': [(0, 1), (1, 1), (2, 1), (3, 1)], 'energy': 25.0}
```

Au fait, c'est quoi un « qubit » ?

On ne parle que de lui. On se demande combien il en faut, combien on en possède aujourd'hui, combien les futurs ordinateurs quantiques en embarqueront. Le nombre de qubits disponibles est devenu l'un des principaux critères permettant de comparer les différentes technologies d'ordinateurs quantiques.



Jean-Michel Torres

Passionné par l'informatique quantique. Ancien "Quantum Ambassador" à IBM France.

Enseigne l'informatique quantique dans de nombreuses écoles et universités.

Mais au fait, qu'est-ce qu'un **qubit** ?
Répondre à cette question avec toute la rigueur nécessaire demanderait plusieurs années d'étude. Le sujet fait appel à la physique quantique, aux mathématiques, à l'informatique théorique et à diverses technologies expérimentales. Ce n'est évidemment pas l'objectif de ce chapitre.

Je vais plutôt essayer d'en donner une définition volontairement simplifiée. Elle ne sera pas parfaitement exacte dans tous ses détails, mais elle sera suffisamment fidèle pour comprendre les principes essentiels et se faire une idée de la manière dont un ordinateur quantique représente et manipule l'information.

Le qubit comme objet physique

Une première façon de voir le qubit consiste à le considérer comme un objet physique obéissant aux lois de la mécanique quantique.

Élaborée au début du 20e siècle, la mécanique quantique décrit principalement le comportement des objets microscopiques. Elle est réputée difficile, car elle fait intervenir des notions qui semblent parfois contredire notre intuition : quantification, superposition, intrication... Nous reviendrons progressivement sur chacune de ces idées.

Pourtant, cette théorie est à l'origine d'une grande partie des technologies qui nous entourent aujourd'hui. Les transistors présents dans les processeurs, les mémoires et l'ensemble des circuits électroniques reposent sur la mécanique quantique. Il en va de même des lasers utilisés dans les réseaux de fibres optiques, des capteurs CCD des appareils photographiques, ou encore de l'imagerie par résonance magnétique (IRM) utilisée dans les hôpitaux.

Il ne s'agit cependant pas ici d'un cours de physique quantique. Notre objectif est uniquement de comprendre ce qu'est un qubit.

Les objets susceptibles de jouer le rôle de qubit sont très variés. Il peut s'agir d'un photon, d'un électron, d'un atome, d'un ion ou même d'une petite molécule. Dans tous les cas, le défi consiste à savoir isoler cet objet, le contrôler avec une très grande précision, puis mesurer son état. Chacun de ces objets peut alors constituer ce que l'informatique quantique appelle un **qubit**.

Certaines technologies utilisent également des circuits supraconducteurs gravés sur des puces de silicium. Ces circuits comportent des **jonctions Josephson**, qui présentent les propriétés nécessaires pour se comporter comme des systèmes quantiques. C'est cette approche qui est notamment utilisée par IBM, Google, Rigetti et D-Wave.

La France occupe également une place importante dans ce domaine. Plusieurs entreprises développent leurs propres technologies quantiques : **Quandela**, spécialisée dans les qubits photoniques, **Pasqal**, qui exploite des atomes neutres refroidis par laser, ou encore **Qobly**, qui développe des qubits à spin d'électron, mais aussi **C12, Alice et Bob**

Pour le lecteur souhaitant approfondir ces différentes technologies, une référence s'impose : le blog *Opinions Libres* d'Olivier Ezratty, ainsi que son remarquable ouvrage **Understanding Quantum Technologies**, disponible gratuitement au format PDF. Avec plus de 1 500 pages, il constitue probablement l'une des références les plus complètes sur les technologies quantiques actuelles et leurs perspectives (https://www.oezratty.net/wordpress/)

Du bit au qubit

Revenons maintenant au qubit, en partant de l'unité élémentaire de l'informatique classique : le **bit** (*Binary Digit*).

Le bit est la plus petite quantité d'information manipulée par un ordinateur. Laissons de côté son implémentation matérielle, aujourd'hui remarquablement maîtrisée par les fabricants de processeurs tels qu'Intel, AMD, Apple, Qualcomm ou NVIDIA, et intéressons-nous uniquement au modèle de calcul.

Du point de vue informatique, un bit est un objet qui ne peut prendre que **deux valeurs**, notées par convention **0** et **1**. Selon le contexte, ces deux valeurs peuvent également représenter deux niveaux de tension (par exemple 0 V et 5 V), les valeurs logiques **FAUX** et **VRAI**, ou encore l'absence et la présence d'un signal.

Les opérations élémentaires que l'on peut effectuer sur la valeur d'un bit sont extrêmement simples :
- ne rien faire ;
- le mettre à 0 ;
- le mettre à 1 ;
- inverser sa valeur.

Deux valeurs, quatre opérations : cela peut sembler très limité. Pourtant, cette simplicité suffit à construire toute l'informatique moderne. En combinant un très grand nombre de bits et quelques fonctions logiques fondamentales (ET, OU, XOR, NAND, etc.), il est possible d'effectuer tous les calculs qu'un ordinateur est capable de réaliser. Bien entendu, cela

ne signifie pas que tout est calculable : certaines questions dépassent les possibilités de tout système informatique, aussi puissant soit-il.

Le **qubit** (*Quantum Bit*) joue, dans un ordinateur quantique, le même rôle que le bit dans un ordinateur classique : il constitue la plus petite unité d'information manipulée par la machine.

Les ressemblances s'arrêtent toutefois rapidement.
Comme le bit, le qubit possède deux états fondamentaux. En revanche, afin de bien les distinguer des valeurs 0 et 1 de l'informatique classique, on les note à l'aide de la **notation de Dirac** : $|0\rangle$ et $|1\rangle$, ces symboles se lisent respectivement « ket zéro » et « ket un ».

La différence essentielle est qu'un qubit ne se limite pas à ces deux états. Il peut exister dans une infinité d'états intermédiaires, appelés **superpositions** des états $|0\rangle$ et $|1\rangle$.

En revanche, lorsqu'on mesure un qubit, le résultat est toujours l'un des deux états de base : soit $|0\rangle$ soit $|1\rangle$

C'est précisément cette propriété qui justifie le qualificatif de **quantique**. Les résultats possibles de la mesure sont quantifiés : ils prennent uniquement certaines valeurs discrètes, contrairement à une grandeur continue.

Une analogie simple consiste à comparer un escalier et une rampe d'accès. Une rampe permet de se placer à n'importe quelle hauteur ; un escalier, au contraire, n'autorise que quelques hauteurs bien déterminées, correspondant à ses marches. Les états observables d'un qubit ressemblent davantage aux marches d'un escalier qu'à une rampe continue.

Cette première propriété est souvent la plus déroutante : **un qubit peut être préparé dans une infinité d'états différents, mais toute mesure ne peut révéler que l'un de ses deux états fondamentaux.**

Une analogie avec les couleurs

Pour mieux comprendre cette idée, utilisons une analogie.
Nous avons l'habitude d'additionner des nombres : 234 + 721 = 955. Mais les nombres ne sont pas les seuls objets que l'on peut combiner.

On peut, par exemple, additionner deux vents : un vent du nord et un vent d'est donnent un vent de nord-est. Si le vent du nord est plus fort, on obtiendra un vent de nord-nord-est.

On peut également mélanger des couleurs. Le bleu et le jaune produisent du vert, et toutes les nuances intermédiaires sont possibles selon les proportions utilisées.

Imaginons maintenant que les états : $|0\rangle$ et $|1\rangle$ correspondent respectivement au bleu et au jaune.

Le qubit pourrait alors être dans n'importe quelle nuance de vert, c'est-à-dire dans n'importe quelle combinaison de bleu et de jaune. Tant qu'on ne l'observe pas, toutes ces nuances sont possibles.

En revanche, au moment de la mesure, la situation change brutalement : le qubit est observé soit entièrement bleu, soit entièrement jaune. Cette transition fait partie des postulats fondamentaux de la mécanique quantique.

Cette analogie permet également d'imaginer la richesse des opérations possibles sur un qubit.

Avec un bit classique, les opérations sont très peu nombreuses. Avec notre « qubit-couleur », on pourrait imaginer modifier progressivement la proportion de bleu ou de jaune, faire varier légèrement la teinte, inverser les proportions ou encore effectuer une infinité d'autres transformations.

Le qubit est effectivement beaucoup plus riche que le bit. Son état peut être modifié d'une infinité de façons différentes, à condition que ces transformations respectent les lois de la mécanique quantique.

Retenons donc les idées essentielles :
- Un qubit peut être préparé dans une infinité d'états physiques différents, chacun étant une combinaison des deux états de base $|0\rangle$ et $|1\rangle$;
- seuls les deux états de base peuvent être observés lors d'une mesure ;
- tous les autres états sont des combinaisons des deux états fondamentaux ;
- il existe une infinité de transformations permettant de faire évoluer l'état d'un qubit.

Une remarque importante mérite toutefois d'être anticipée. Dans notre analogie, nous avons évoqué des opérations telles que « mettre complètement au bleu » ou « mettre complètement au jaune ». En réalité, de telles opérations ne sont pas réalisables sur un qubit isolé, car elles ne respectent pas les contraintes fondamentales de la mécanique quantique, notamment la conservation de la norme et la réversibilité des transformations.

À ce stade, une conclusion s'impose : du point de vue de la représentation et de la manipulation de l'information, le qubit est un objet bien plus riche que le bit classique… et nous n'avons encore exploré qu'une partie de ses propriétés

La superposition

Jusqu'à présent, nous avons utilisé des analogies pour construire une intuition du qubit. Il est maintenant temps d'introduire le modèle mathématique qui permet de le décrire.

Nous avons vu qu'un qubit possède deux états fondamentaux, notés $|0\rangle$ et $|1\rangle$.

L'idée essentielle est la suivante : **l'état d'un qubit est une combinaison linéaire de ces deux états de base**.
Autrement dit, tout état possible du qubit s'obtient en « additionnant » les deux états fondamentaux avec certaines pro-

portions. Cette idée conduit naturellement à un modèle bien connu des mathématiciens : l'algèbre linéaire.

On associe au qubit un **espace vectoriel complexe de dimension 2**, engendré par les deux vecteurs de base et . Les différents états possibles du qubit sont alors représentés par les vecteurs de cet espace.
Tous ces vecteurs ne correspondent cependant pas à des états physiques. Pour représenter un qubit, un vecteur doit être **normalisé**, c'est-à-dire avoir une norme égale à 1.
Pour l'instant, retenons simplement cette contrainte. Nous verrons bientôt qu'elle possède une interprétation physique très simple.

Enfin, un dernier détail rend le modèle encore plus riche : les coefficients de la combinaison linéaire ne sont pas des nombres réels, mais des **nombres complexes**.
Nous reviendrons plus loin sur l'intérêt de cette propriété, notamment lorsque nous parlerons de la phase du qubit.

Pour les lecteurs qui souhaitent aller plus loin je recommande par exemple la vidéo *Dirac's Belt Trick, Topology, and Spin ½ Particles* de la chaîne YouTube *NoahExplainsPhysics*. Elle offre une intuition remarquable de certaines propriétés géométriques des états quantiques.

Prenez le temps d'assimiler cette idée fondamentale :
La superposition est une combinaison linéaire des états de base. Mathématiquement, il s'agit d'une addition pondérée de ces états.
Cette phrase résume à elle seule une grande partie du modèle du qubit.

La description précédente s'écrit de manière très concise grâce à la notation de Dirac :

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Cette équation résume l'ensemble des états possibles d'un qubit. Les coefficients α et β sont des nombres complexes. Ils ne sont toutefois pas arbitraires : ils doivent satisfaire la condition de normalisation : $|\alpha|^2 + |\beta|^2 = 1$,

Cette relation possède une interprétation très simple :
Si le qubit est dans l'état $|\psi\rangle$, alors
- $|\alpha|^2$ est la probabilité de mesurer l'état $|0\rangle$;
- $|\beta|^2$ est la probabilité de mesurer l'état $|1\rangle$.

Comme ces deux résultats sont les seuls possibles, leurs probabilités doivent nécessairement s'additionner pour donner 1. Nous retrouvons ainsi une propriété fondamentale de toute théorie probabiliste : la somme des probabilités de tous les événements possibles est égale à 100 %.

Une autre façon d'écrire un qubit

Les nombres complexes peuvent également être représentés sous leur forme polaire.
Cette écriture permet d'exprimer l'état du qubit avec seulement deux paramètres réels : deux angles, traditionnellement notés θ et ψ .

On obtient alors l'expression équivalente

$$|\psi\rangle = \cos(\frac{\theta}{2})|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle$$

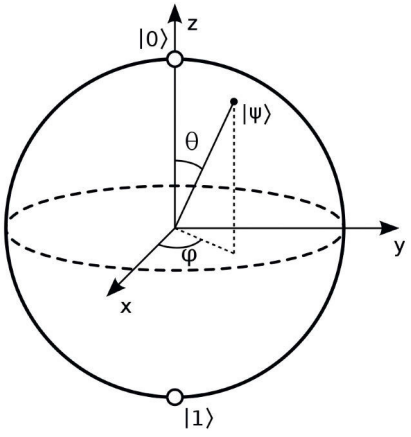
Cette formulation est particulièrement importante, car elle conduit directement à une représentation géométrique très élégante : **la sphère de Bloch**.

Les deux angles θ et ψ permettent en effet de repérer chaque état d'un qubit par un point situé à la surface de cette sphère.

C'est cette représentation qui va nous permettre de visualiser les états quantiques et de comprendre, de manière presque intuitive, l'effet des différentes portes quantiques.

La sphère de Bloch

Cette nouvelle écriture de l'état d'un qubit conduit naturellement à une représentation géométrique particulièrement élégante : **la sphère de Bloch**.



Chaque état d'un qubit peut être représenté par un vecteur dont l'origine est située au centre d'une sphère et dont l'extrémité parcourt sa surface lorsque les deux angles θ et ψ varient.

Par convention, l'état $|0\rangle$ est placé au pôle Nord de la sphère, tandis que l'état $|1\rangle$ occupe le pôle Sud. Ces deux états constituent les états de base du qubit.

L'angle polaire θ indique la position du vecteur d'état entre ces deux pôles. Plus θ est proche de 0, plus l'état est voisin de $|0\rangle$. À l'inverse, lorsque θ se rapproche de 2π l'état devient de plus en plus proche de $|1\rangle$.

Cette position est directement liée aux probabilités de mesure. Plus le vecteur est proche du pôle Nord, plus la probabilité d'obtenir $|0\rangle$. est élevée. À l'inverse, plus il se rapproche du pôle Sud, plus la probabilité de mesurer $|1\rangle$ augmente.

On peut se représenter intuitivement la mesure comme une projection du vecteur d'état sur l'axe (z). Cette image géométrique permet de comprendre pourquoi la hauteur du vecteur est liée aux probabilités de mesure. Il ne faut cependant pas oublier qu'il s'agit d'une représentation intuitive : la mesure quantique est un phénomène plus subtil, décrit par les postulats de la mécanique quantique.

Après la mesure, le qubit ne reste plus dans son état de superposition. Son vecteur d'état « s'effondre » sur l'un des deux pôles de la sphère : il devient exactement $|0\rangle$ ou $|1\rangle$, tout en conservant une norme égale à 1.

Considérons maintenant le second angle, ψ.
Lorsque l'on mesure directement le qubit dans la base $(\{|0\rangle, |1\rangle\})$, la valeur de ψ n'a aucune influence sur les probabilités de mesure. Celles-ci dépendent uniquement de l'angle θ.

En revanche, ψ joue un rôle fondamental pendant le calcul quantique. Cet angle représente la **phase** du qubit et intervient dans les phénomènes d'interférence qui constituent l'une des ressources essentielles de l'informatique quantique. Deux qubits possédant les mêmes probabilités de mesure, mais des phases différentes pourront ainsi évoluer de manière très différente au cours d'un algorithme.

C'est précisément cette notion de phase qui distingue profondément l'informatique quantique de l'informatique classique et qui permet à certains algorithmes quantiques d'exploiter les interférences pour amplifier les bonnes réponses et atténuer les mauvaises.

Pour le lecteur souhaitant voir cette notion mise en œuvre sur un exemple concret, je renvoie aux articles que j'ai publiés dans *Programmez !* (n° 240, mai 2020, p. 51, et n° 241, juillet-août 2021, p. 65), qui détaillent le rôle de la phase dans l'algorithme de Deutsch-Jozsa.

Les portes (ou opérations) quantiques

La représentation d'un qubit sur la sphère de Bloch ne permet pas seulement de visualiser son état. Elle offre également une manière très intuitive de comprendre comment cet état peut être modifié au cours d'un calcul.

En mécanique quantique, toute évolution d'un qubit est décrite par une **transformation unitaire**, c'est-à-dire une transformation linéaire représentée par une matrice complexe (2 \times 2) qui conserve la norme du vecteur d'état. Cette propriété garantit notamment que les probabilités restent correctement normalisées et que l'évolution du qubit est réversible.

Le nombre de transformations possibles est infini.
Parmi elles, trois opérations particulièrement importantes sont regroupées sous le nom de **portes de Pauli**.

La porte **X** effectue une rotation de π autour de l'axe (x) de la sphère de Bloch. Elle échange les états $|0\rangle$ et $|1\rangle$, ce qui revient à inverser leurs probabilités de mesure.

La porte **Y** réalise une rotation de π autour de l'axe (y). Elle échange également les deux états de base, mais introduit en plus une modification de leur phase.

Enfin, la porte **Z** correspond à une rotation d'un angle π autour de l'axe (z). Contrairement aux deux précédentes, elle ne modifie pas les probabilités de mesure dans la base $\{|0\rangle, |1\rangle\}$, mais agit uniquement sur la phase du qubit.

Ces trois portes constituent les briques élémentaires de nombreuses opérations quantiques. Elles existent également sous la forme de rotations d'angle quelconque, notées (R_X(θ)), (R_Y(θ)) et (R_Z(θ)), qui permettent d'amener le vecteur d'état vers n'importe quel point de la sphère de Bloch.

Une autre porte joue un rôle tout à fait particulier : la **porte de Hadamard**, notée (H), du nom du mathématicien français Jacques Hadamard.
Cette porte transforme un état de base en un état de superposition :

$$|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$|1\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Autrement dit, elle crée une superposition équilibrée des deux états fondamentaux. Cette propriété explique pourquoi la porte de Hadamard apparaît dans la plupart des algorithmes quantiques.

Nous disposons maintenant des éléments nécessaires pour comprendre ce qu'est un qubit et la manière dont son état évolue au cours d'un calcul. Il est temps de franchir une nouvelle étape : étudier des systèmes composés de plusieurs qubits. C'est là qu'apparaissent les phénomènes les plus remarquables de l'informatique quantique, notamment l'intrication.

Les systèmes à plusieurs qubits

Pour décrire un système composé de plusieurs qubits, il faut étendre le modèle que nous avons construit pour un qubit unique.
Chaque qubit est associé à son propre espace vectoriel. Lorsque plusieurs qubits sont réunis, leurs espaces ne sont pas simplement juxtaposés : ils sont combinés à l'aide d'une opération mathématique appelée **produit tensoriel**.
Le nom peut sembler impressionnant, mais l'idée est finalement assez simple.

Reprenons une analogie.
Imaginons deux pièces de monnaie.
La première est une pièce bleue, dont les deux états sont **Pile** (P) et **Face** (F).
La seconde est une pièce rouge, possédant elle aussi les états **Pile** et **Face**.
Bien que les deux pièces utilisent les mêmes noms pour leurs états, il s'agit bien de deux objets différents. Le « Pile » de la pièce bleue n'a évidemment rien à voir avec le « Pile » de la pièce rouge.

Le système constitué des deux pièces possède alors quatre états de base :
- Pile–Pile ;
- Pile–Face ;

• Face–Pile ;
• Face–Face.
Ces quatre états constituent la base de l'espace décrivant le système complet. Si chacune des deux pièces est quantique, le système peut naturellement être dans une superposition de ces quatre états.

Ajoutons maintenant une troisième pièce, verte cette fois, le nombre d'états de base double immédiatement : le système possède désormais huit états.

Chaque nouveau qubit double ainsi la dimension de l'espace des états. Plus généralement, un registre de (n) qubits est décrit par un espace vectoriel de dimension 2^n
Cette croissance est extrêmement rapide : avec seulement 100 qubits, l'espace des états possède déjà $2^{100} \approx 1{,}27.10^{30}$ dimensions.
Chaque qubit supplémentaire double encore cette quantité. Avec 200 qubits, on atteint environ 10^{60} dimensions.

À titre de comparaison, on estime que le nombre de particules dans l'Univers observable est de l'ordre de 2^{275}
Cette croissance exponentielle explique pourquoi quelques centaines de qubits suffisent déjà à décrire des espaces d'états gigantesques. C'est également l'une des raisons pour lesquelles la simulation exacte d'un ordinateur quantique devient rapidement impossible sur un ordinateur classique.
C'est dans cet immense espace vectoriel que naissent les propriétés les plus fascinantes de l'informatique quantique, en particulier l'intrication, que nous allons maintenant découvrir à travers l'exemple le plus célèbre : l'état de Bell.
Du point de vue de la notation avec les ket :

$$|\psi\rangle|\phi\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle)$$
$$|\psi\phi\rangle = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$$

Notez que l'on admet de représenter les états de plusieurs qubits à l'intérieur d'un même ket ($|\psi\phi\rangle$ *ou* $|01\rangle$), mais bien entendu les positions ne sont pas interchangeables,
Les règles de calcul sont simples : celles de la multiplication et de l'addition à savoir l'associativité, la distributivité, MAIS PAS la commutativité.

L'état de Bell : un premier exemple d'intrication

Nous disposons maintenant de tous les outils nécessaires pour construire **l'état de Bell**.
Considérons deux qubits, notés (q_0) et (q_1).
Nous allons leur appliquer successivement deux portes quantiques :
• une porte de **Hadamard** (H) sur le premier qubit (q_0) ;
• puis une porte **CNOT** (*Controlled NOT*), dont (q_0) est le qubit de contrôle et (q_1) le qubit cible.

La porte CNOT est une opération à deux qubits. Elle laisse le premier qubit inchangé et applique une inversion du second uniquement lorsque le premier est dans l'état $|1\rangle$.
Si l'on note les l'état des deux qubits $|x,y\rangle$, cette transformation s'écrit

$$|x, y\rangle \rightarrow |x, x \oplus y\rangle$$

où $\oplus$ désigne l'opération **XOR** (OU exclusif).
Supposons que les deux qubits soient initialement dans l'état $|00\rangle$, après application de la porte de Hadamard sur le premier qubit, on obtient

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$$

La porte CNOT transforme ensuite l'état $|10\rangle$ en $|11\rangle$ tandis que $|00\rangle$ reste inchangé. L'état final est donc

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

En seulement deux opérations, nous avons construit un état à la fois **en superposition** et **intriqué**.

La superposition apparaît clairement : l'état est la somme de deux vecteurs de base de l'espace des états.
L'intrication, en revanche, est une propriété beaucoup plus surprenante.
Cet état est si important qu'il porte le nom de **John Stewart Bell**, physicien irlandais qui, au milieu des années 1960, démontra un résultat fondamental aujourd'hui connu sous le nom d'**inégalités de Bell**. Ces inégalités permettent de distinguer les prédictions de la mécanique quantique de celles de toute théorie reposant sur des variables cachées locales.
Au début des années 1980, Alain Aspect réalisa une série d'expériences qui montrèrent une violation de ces inégalités, confirmant ainsi les prédictions de la mécanique quantique. Ces travaux lui valurent, avec John Clauser et Anton Zeilinger, le prix Nobel de physique en 2022.

Mais que signifie exactement **intriqué** ?
Dans l'état de Bell, si l'on mesure le premier qubit, on obtient 0 ou 1 avec une probabilité de 50 %.

Supposons que la mesure donne le résultat $|0\rangle$, alors, sans effectuer la moindre mesure supplémentaire, nous savons immédiatement que le second qubit sera lui aussi trouvé dans l'état $|0\rangle$. De même, si la première mesure donne $|1\rangle$, la seconde donnera nécessairement $|1\rangle$.
Il est essentiel de comprendre que cette corrélation n'existait pas sous la forme de deux états individuels bien définis avant la mesure. Le système était décrit par un **état global**, impossible à décomposer en deux états indépendants, un pour chaque qubit. C'est précisément cette impossibilité qui caractérise l'intrication.

Plus remarquable encore, cette corrélation subsiste même lorsque les deux qubits sont séparés par de très grandes distances. C'est cette propriété qui conduisit Einstein à parler, non sans une certaine ironie, d'une « action fantomatique à distance » (*spooky action at a distance*), car elle semblait incompatible avec son intuition de la réalité physique.
Aujourd'hui, l'intrication est non seulement l'un des phénomènes les plus étonnants de la mécanique quantique, mais également l'une des principales ressources exploitées par l'informatique quantique, les réseaux quantiques et la cryptographie quantique.



Pablo ROSEIRO

Quantum Applications Engineer in Chemistry and Materials science

Pablo Roseiro holds a Ph.D. in Theoretical Chemistry from the University of Strasbourg. After an experience at Hydro-Québec and McGill University, he joined Quobly as a Quantum Applications Engineer in Chemistry and Materials science. He focuses on identifying and developing industrial applications in materials science, building on the work of Quobly's quantum algorithms team and the company's scalable technologies.

Quobly : programmer l'ordinateur quantique sur silicium

La deep tech française Quobly veut industrialiser le calcul quantique en s'appuyant sur le matériau le mieux maîtrisé dans l'industrie du semiconducteur : le silicium. Tour d'horizon de sa machine, des applications qu'elle vise et de ses outils open source, dont SpinPulse, qui permet de coder au plus près du processeur quantique.

Le pari du silicium

Le calcul quantique quitte peu à peu les laboratoires pour entrer dans les centres de données. Dans cette course, **Quobly** fait un choix singulier : miser sur le silicium, la brique de base de l'informatique moderne. L'entreprise, issue des travaux du CEA-Leti à Grenoble, conçoit des ordinateurs quantiques fondés sur des **qubits de spin sur silicium**, fabriqués sur les lignes industrielles 300 mm de STMicroelectronics. L'idée directrice est ingénieuse : réutiliser soixante-dix ans de savoir-faire des semiconducteurs pour rendre le quantique fiable, abordable et, surtout, capable de passer à l'échelle.

Quobly a levé **115 millions d'euros en série A** et a déjà fait fabriquer plus de 2,4 millions de qubits chez STMicroelectronics (un cumul de dispositifs de test produits), et est forte de plus d'une centaine d'experts, de 65 brevets déposés et de cinq implantations dans le monde. Son écosystème industriel réunit STMicroelectronics et le CEA-Leti pour la fabrication, Orano pour le silicium purifié, Soitec pour les substrats, OVHcloud pour l'intégration de calcul, ou encore Entropica Labs et Foxconn côté logiciel applicatif.

La technologie principale est le **QSOI** (Quantum Silicon On Insulator), une évolution du FD-SOI classique. Afin d'atténuer au maximum les bruits liés au silicium, Quobly utilise des plaques purifiées en silicium 28 qui offrent un environnement quasiment exempt de spins parasites, condition indispensable pour des qubits cohérents. L'objectif affiché est de créer et commercialiser un ordinateur quantique contenant un processeur quantique d'un million de qubits de la taille d'une pièce de monnaie, installable dans les centres de données, et tolérant aux fautes. Le tout en s'appuyant sur le coût déjà amorti des procédés silicium.

Dans le QPU : un spin pour un qubit

Chez Quobly, un qubit est le **spin d'un électron** piégé dans une boîte quantique gravée dans le silicium. Deux types d'opérations suffisent à fabriquer un calculateur quantique universel. Les portes à un qubit s'obtiennent en appliquant un champ magnétique oscillant qui fait tourner le spin : ce sont les rotations R_X, R_Y et R_Z. Les portes à deux qubits s'obtiennent en abaissant la barrière de potentiel entre deux boîtes voisines, ce qui active un couplage d'échange et réalise une porte R_{ZZ}.

La puce est refroidie à environ 1 kelvin et son électronique de contrôle est embarquée au plus près des qubits. Mais ces spins ont une particularité décisive : leur bruit dominant est **non markovien**. Le bruit de charge et le bruit magnétique en 1/f (bruit « rose ») introduisent une mémoire que l'on caractérise par le temps de cohérence T_2^*. C'est précisément ce point qui rend indispensable un outil de simulation dédié.

SpinPulse : coder au niveau des impulsions

SpinPulse est une bibliothèque Python open source (licence Apache 2.0, pip install spin-pulse, article paru sur arXiv) qui simule un ordinateur quantique à qubits de spin **au niveau des impulsions**, et pas seulement au niveau des portes. Là où un logiciel d'impulsions généraliste (par exemple Qiskit Pulse) reste largement agnostique au matériau, SpinPulse modélise la physique propre aux spins de silicium, notamment ce fameux bruit non markovien. Cela permet de produire des circuits « noise-accurate », c'est-à-dire dont la fidélité simulée reflète ce que ferait réellement le silicium (en d'autres termes, à quel point l'opération réelle correspond à l'opération idéale).

Pourquoi est-ce central pour la co-conception ? Parce que le matériel à grande échelle n'existe pas encore. Les équipes algorithmes et les équipes physique ont besoin d'un terrain commun pour anticiper, dès aujourd'hui, comment un circuit donné se comportera sur le silicium de Quobly, avec ses vraies portes natives et son vrai bruit. SpinPulse est ce pont : il relie le circuit abstrait écrit dans Qiskit à la réalité physique de la puce, et permet d'itérer hardware et logiciel ensemble avant même la gravure.

Le flot de travail se déroule en trois temps. Un circuit Qiskit est d'abord transpilé vers le jeu de portes natives du modèle (R_X, R_Y, R_Z, R_{ZZ}), puis converti en une séquence d'impulsions, et finalement intégré numériquement en présence d'un environnement bruité. Concrètement, on commence par décrire le matériel et le circuit :

```
import qiskit as qi
from spin_pulse.transpilation.hardware_specs import (
    HardwareSpecs, Shape,
)

num_qubits = 3
B_field, delta, J_coupling = 0.5, 0.2, 0.01
specs = HardwareSpecs(
    num_qubits, B_field, delta, J_coupling,
```

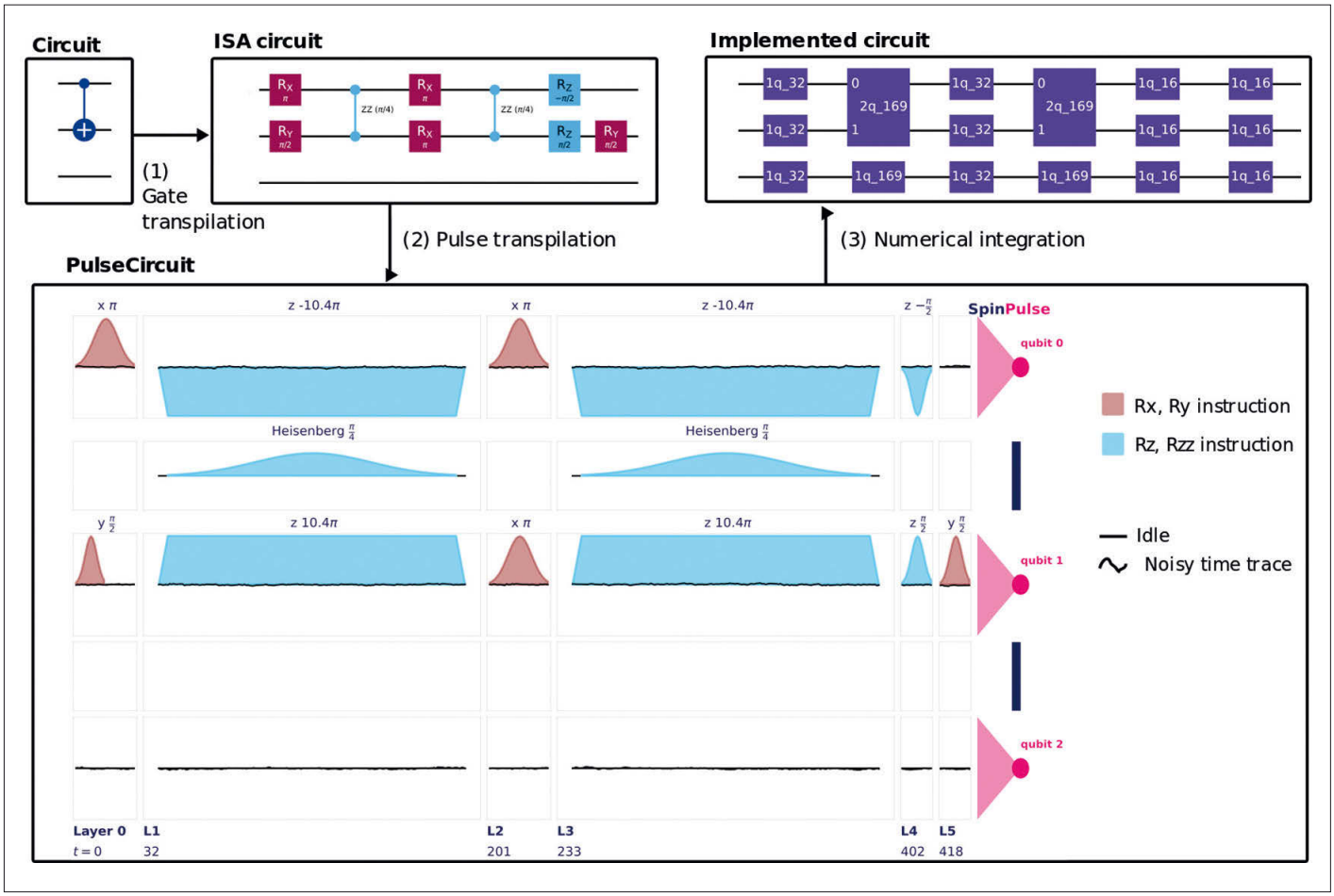


Figure 1 : *Principe de fonctionnement de SpinPulse (https://quobly-sw.github.io/SpinPulse/)*

```
    Shape.GAUSSIAN, ramp_duration=5, optim=0,
)

circ = qi.QuantumCircuit(num_qubits)
circ.cx(0, 1)

isa_circ = specs.gate_transpile(circ)
```

L'objet HardwareSpecs porte les paramètres physiques : champ magnétique, écart de fréquence entre qubits, couplage d'échange, forme des impulsions (gaussienne ou carrée), durée des pentes dans les impulsions… La méthode gate_transpile produit un jeu d'instructions (*Instruction Set Architecture*, ISA) strictement équivalent, mais exprimé dans les portes que la puce sait exécuter. On calcule ensuite les impulsions :

```
from spin_pulse import PulseCircuit

pulse_circuit = PulseCircuit.from_circuit(isa_circ, specs)
```

Dans la **Figure 1**, les impulsions rouges correspondent aux portes à un qubit, les bleues aux portes à deux qubits. Durées et amplitudes ne sont pas choisies à la main : elles sont déduites des spécifications matérielles. La classe PulseCircuit calcule automatiquement le temps que prend chaque rotation pour réaliser exactement l'angle voulu.

Ce détour par les impulsions dit ce qu'est vraiment un ordinateur quantique. Au niveau physique, la machine n'exécute pas des « portes ». Elle joue, dans le temps, une partition de signaux analogiques : un champ magnétique oscillant, une pente de tension qui abaisse une barrière, des fenêtres d'attente entre les deux. Les qubits évoluent sous l'effet de ces signaux. Ainsi, les portes logiques que manipule le développeur ne sont qu'un résumé commode d'une évolution analogique calibrée pour produire, au bon instant et pendant la bonne durée, exactement la rotation voulue.

Programmer un QPU au silicium revient donc à ordonnancer ces impulsions, et le circuit quantique n'est pas le point de départ mais le résultat. SpinPulse rend cette inversion explicite : on lui donne un circuit logique, elle en déduit la séquence d'impulsions à jouer, puis intègre numériquement cette séquence pour reconstruire le circuit réellement réalisé. Voir l'ordinateur comme un ordonnanceur d'impulsions plutôt que comme un exécuteur de portes change la façon de l'optimiser : la forme des signaux, leur amplitude et surtout leur ordonnancement temporel deviennent des leviers de performance à part entière.

Ordonnancer les impulsions : un enjeu de fidélité

C'est ici que se joue une part essentielle de la co-conception. Une porte à un qubit et une porte à deux qubits ne prennent pas le même temps. Au sein d'une même couche du circuit, les qubits terminent donc leurs opérations à des instants dif-

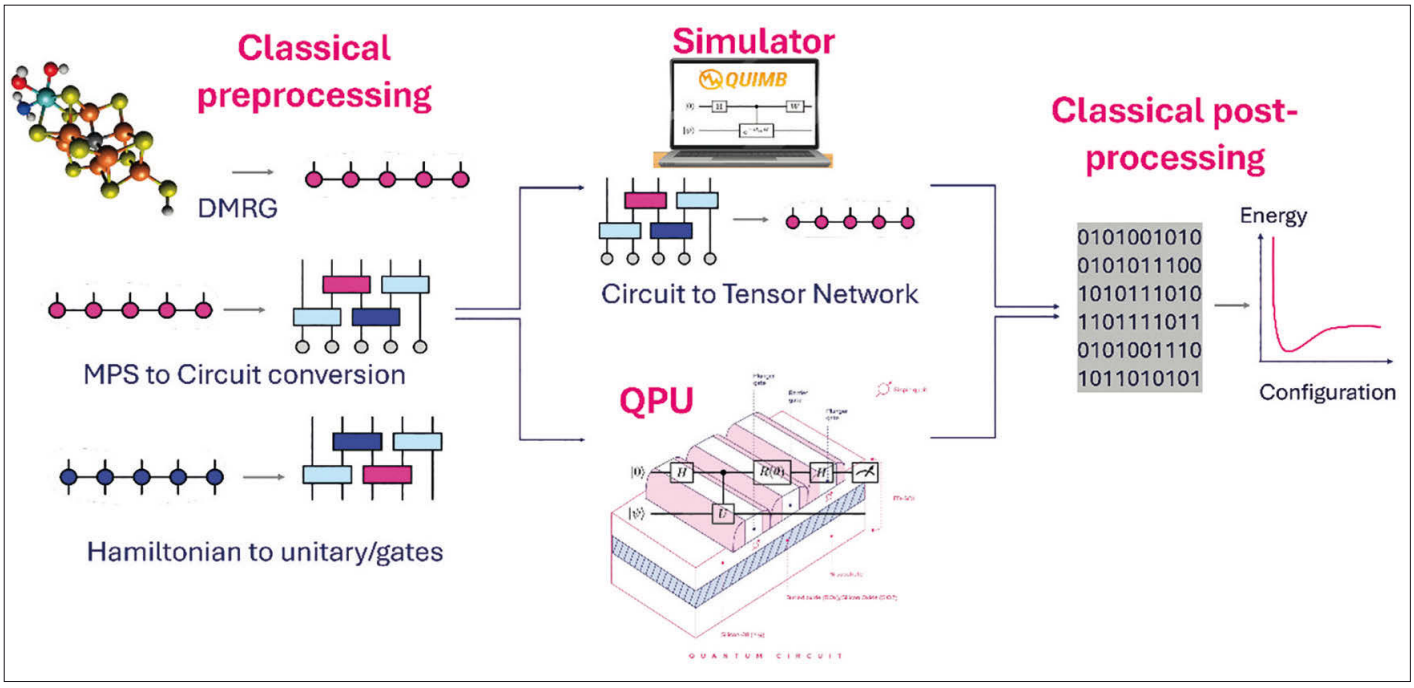


Figure 2 : *Principe de fonctionnement de QPE-toolbox (https://quobly-sw.github.io/qpe-toolbox/index.html)*

férents. SpinPulse insère **automatiquement des temps d'attente** pour resynchroniser tous les qubits à chaque couche : l'ordonnancement temporel de la séquence est construit à partir des durées physiques de chaque porte. Le détail compte car, pendant ces fenêtres d'attente, les spins continuent d'accumuler du déphasage sous l'effet du bruit. L'ordonnancement n'est donc pas neutre : la manière dont on synchronise les couches influe directement sur la fidélité finale. Programmer le QPU de Quobly, c'est décider comment organiser les impulsions dans le temps.

Reste à mesurer l'effet du bruit. On attache un environnement expérimental, ici un bruit rose avec un T_2^* fini, que l'on moyenne sur de nombreuses exécutions :

```
from spin_pulse.environment.noise import NoiseType
from spin_pulse.environment.experimental_environment import (
  ExperimentalEnvironment,
)

env = ExperimentalEnvironment(
  hardware_specs=specs, noise_type=NoiseType.PINK,
  T2S=500, duration=2**15, segment_duration=2**15,
)
noisy = PulseCircuit.from_circuit(isa_circ, specs, exp_env=env)
print("%.3f" %noisy.mean_fidelity(env))
0.901
```

La fidélité chute (0,90 au lieu de 1), comme attendu pour un environnement bruité. Des stratégies de co-conception existent néanmoins afin de protéger les spins de ce bruit rose en remplaçant les temps d'attente par des rotations à un qubit : c'est du **découplage dynamique**.

```
from spin_pulse import DynamicalDecoupling

specs_dd = HardwareSpecs(
  num_qubits, B_field, delta, J_coupling, Shape.GAUSSIAN,
  dynamical_decoupling=DynamicalDecoupling.SPIN_ECHO,
)
dd = PulseCircuit.from_circuit(isa_circ, specs_dd, exp_env=env)
print("%.3f" %dd.mean_fidelity(env))
0.968
```

En ajoutant des impulsions, on passe ainsi d'environ 0,90 à 0,97 de fidélité moyenne, une amélioration purement logicielle non négligeable. À noter que ces valeurs illustrent la mécanique de l'outil avec des paramètres d'exemple, ce ne sont pas les fidélités de portes mesurées sur les puces de Quobly.

SpinPulse fournit aussi des outils de caractérisation pour ajuster les paramètres du modèle à partir des mesures réelles du QPU. La boucle est complète : on écrit un algorithme, on le voit s'exécuter sur une réplique fidèle du silicium, on l'optimise, et on remonte des contraintes pour l'amélioration du matériel physique. SpinPulse n'est pas qu'un simple simulateur, c'est l'atelier de co-conception de Quobly.

Reste le saut le plus important, celui que cet exemple laisse entrevoir : passer de qubits physiques bruités à un calcul utile. Une fidélité de 0,90, voire 0,97, ne suffit pas pour des algorithmes longs. C'est tout l'objet de la correction d'erreur quantique, qui encode un qubit logique fiable dans un grand nombre de qubits physiques, en détectant et en corrigeant les erreurs en continu (code de surface par exemple). C'est cette problématique qui motive l'objectif du million de qubits : les applications réellement transformatrices réclament des milliers de qubits logiques, et chacun coûte beaucoup de qubits physiques. Le pari silicium prend ici tout son sens, puisque seule la fabrication de masse rend ces nombres crédibles. SpinPulse referme la boucle : en estimant les taux d'erreur physiques réels, il alimente le dimensionnement de cette couche de correction.

Les applications visées

Reste la question qui intéresse l'utilisateur final : à quoi va servir cette machine ? Quobly aborde donc ses cas d'usage comme son matériel, par la co-conception : en impliquant tôt les industriels pour calibrer les algorithmes sur les contraintes réelles du silicium plutôt que sur une machine idéale. Quatre familles se dégagent.

Chimie quantique et science des matériaux. C'est le cas d'usage le plus mûr du quantique. Calculer l'énergie fondamentale d'une molécule ou d'un matériau, via l'estimation de phase quantique (*Quantum Phase Estimation*, QPE) ou d'autres algorithmes (*Variational Quantum Eigensolver*, *Sample-based Quantum Diagonalization*…), ouvre la voie à de nouveaux catalyseurs, électrolytes de batteries ou matériaux fonctionnels que la simulation classique peine à traiter. C'est un des objectifs de l'outil QPE-toolbox présenté plus loin.

Simulation multi échelle de systèmes gouvernés par des EDP. Les équations différentielles partielles (EDP), au cœur des simulations de dynamique des fluides numériques (CFD), deviennent rapidement très coûteuses à résoudre pour des problèmes non linéaires et multi-échelles. Les technologies quantiques offrent une voie prometteuse pour développer de nouveaux algorithmes capables, à terme, d'accélérer la résolution de ces équations, d'étendre les capacités de la simulation numérique et de réduire les coûts en ressources dans les centres de calcul intensif.

Optimisation et logistique. Flotte de véhicules, planification de tâches, remplissage de transporteurs : ces problèmes combinatoires sont au cœur de l'industrie. Leur structure discrète se prête à un encodage naturel sur des qubits, et les algorithmes quantiques cherchent à exploiter l'interférence pour amplifier les bonnes solutions plutôt qu'à les énumérer une à une. L'enjeu est de concevoir des algorithmes (*Quantum Approximate Optimization Algorithms*, *quantum walkers*…) qui orientent l'amplitude vers les configurations optimales.

Quantum reservoir computing (QRC). Ce framework est une méthode hybride d'apprentissage automatique : il exploite la dynamique naturelle d'un circuit quantique digital (le réservoir) pour projeter des données séquentielles dans un espace de grande dimension (Hilbert), tandis qu'un ordinateur classique se charge d'entraîner l'unique couche de lecture finale. En combinant la richesse de l'informatique quantique à la simplicité d'une optimisation classique, cette approche s'impose comme un algorithme de choix pour la prévision de séries temporelles chaotiques.

QPE-toolbox : la chaîne complète de l'estimation de phase

Au-delà de SpinPulse, Quobly a publié **QPE-toolbox**, une bibliothèque Python open source, développée en collaboration avec Foxconn (licence Apache 2.0, pip install qpe-toolbox). Elle compile et simule des circuits d'estimation de phase quantique en mélangeant chimie quantique et réseaux de tenseurs. Là où la QPE est une longue chaîne d'étapes souvent fragmentée en différents outils spécialisés, QPE-toolbox propose une implémentation complète afin de simuler l'ensemble du processus et d'estimer les ressources quantiques réelles dont un algorithme aura besoin.

On définit d'abord un hamiltonien moléculaire pour la chimie ou un modèle de spin. On prépare un état initial sous forme de Matrix Product State, par DMRG (*Density Matrix Renormalization Group*) ou par optimisation d'un circuit paramétré. On encode l'hamiltonien en un opérateur unitaire, par évolution exacte, décomposition de Trotter ou combinaison linéaire d'unitaires (LCU). On initialise enfin un circuit doté d'un registre physique et d'un registre de phase, puis on lance l'estimation, avant de remonter l'énergie depuis les mesures de phase.

L'intérêt des réseaux de tenseurs est ici stratégique : ils permettent de simuler et de comparer classiquement des circuits d'estimation de phase de grande taille, et d'estimer les ressources nécessaires avant de disposer du matériel adéquat. Ici encore, l'open source sert la co-conception entre algorithmes de chimie et architecture du processeur quantique.
Figure 2

Du qubit à l'application

La cohérence de la démarche de Quobly tient en une phrase : du silicium pour passer à l'échelle, et de la co-conception pour transformer des qubits en calculs utiles. Les deux bibliothèques open source, SpinPulse pour travailler avec des impulsions et QPE-toolbox pour des applications de chimie, donnent à la communauté les moyens de travailler avec Quobly plutôt que d'attendre la machine. Pour les développeurs curieux du quantique, c'est une invitation à mettre les mains dans le silicium.

ABONNEMENT NUMÉRIQUE
1 an ..... 39 €
Abonnez-vous sur :
www.programmez.com

FAITES VOTRE VEILLE TECHNOLOGIQUE AVEC
programmez!
Le magazine des dév - CTO - Tech Lead

ABONNEMENT PAPIER
1 an ..... 55 €
2 ans ..... 90 €
Voir page 5

IN ENGLISH



Simon Fried

Vice President of Corporate Communications, Classiq

Simon is Vice President of Corporate Communications at Classiq Technologies, where he covers quantum computing's transition from research to enterprise deployment. He writes and speaks regularly on quantum software, developer readiness and the path to commercial quantum advantage.

CLASSIQ

Why Quantum Computing's Next Breakthrough Will Be Written in Code, Not Qubits

Résumé : *« Si le matériel quantique continue de progresser, son adoption par les entreprises dépendra largement des logiciels capables de transformer cette puissance de calcul en applications concrètes. À mesure que les outils de développement deviennent plus accessibles, les développeurs peuvent se former, tester et expérimenter plus facilement. Une nouvelle spécialisation se dessine, à laquelle il est déjà possible de se préparer. »*

Quantum computing hardware is advancing fast, but hardware alone will not decide who benefits from it: software is what determines whether enterprises can actually use it. Quantum development is also becoming more accessible, thanks to higher-level tools that remove much of the low-level circuit work. For developers watching AI reshape classical coding careers, the skills worth building next start now, not once fault-tolerant machines arrive.

Hardware Made the Headlines. Software Will Decide Who Benefits.

Every computing transition follows the same pattern. Mainframes needed compilers. The internet needed browsers and application frameworks. Mobile needed SDKs. In each case, hardware created the possibility, and software created the access. Quantum computing is no different, yet the public conversation remains dominated by qubit counts and error-correction milestones, as if the machines will simply arrive pre-loaded with value. They will not. Every enterprise use case now under serious evaluation, including portfolio optimization in finance, molecular simulation in pharma and chemistry, logistics and scheduling, materials design and cryptographic risk assessment, depends on software that can translate a business or research problem into something a quantum processor can execute, and translate the result back into something a business can use. That translation layer is where the real engineering work, and the real career opportunity, sits. It is also where developers, not physicists, do most of the building.



Classical Code and Quantum Code Are Not the Same Discipline

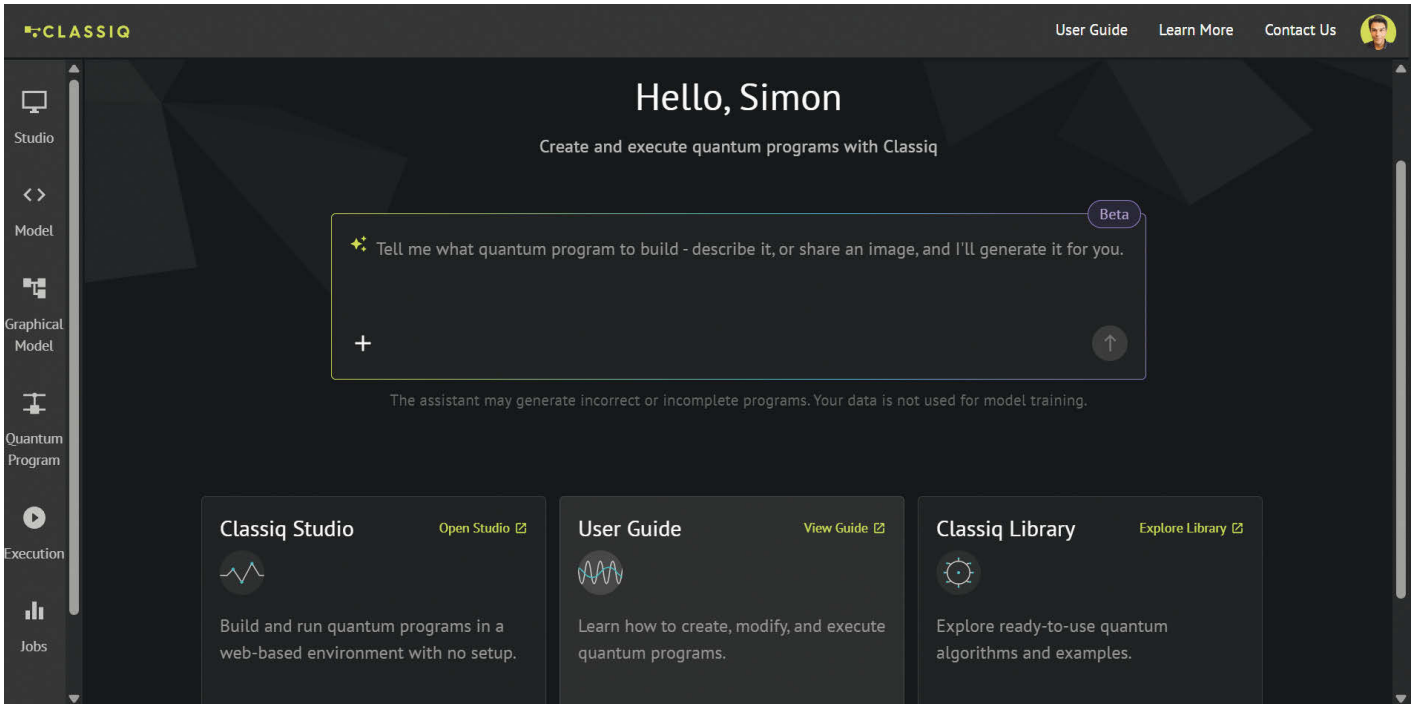
For developers new to the field, the differences are worth naming plainly. Classical programs manipulate deterministic bits; quantum programs manipulate probabilistic amplitudes across superposition and entanglement, and most answers only emerge after many repeated executions, not one. There is no equivalent of a print statement mid-execution: measuring a qubit collapses its state, so debugging relies on simulators and statistical analysis rather than step-through inspection. And unlike classical code, which mostly runs unmodified across hardware, quantum programs must still account for the physical constraints of the processor they target: qubit connectivity, gate fidelity, coherence time.

Classical concept	Quantum equivalent
Bit (0 or 1)	Qubit (superposition of |0> and |1>)
Deterministic output	Probability distribution over outcomes
Debugging via logs and breakpoints	Statistical sampling across repeated runs

None of this makes quantum programming inaccessible; it makes it a genuinely different discipline, closer in spirit to early classical computing, when developers also had to reason close to the hardware. What has changed is how much of that burden a developer has to carry personally.

The Quantum Stack Is Further Along Than Most Developers Assume

Quantum software has quietly gone through the same maturation classical software went through decades ago: from hand-written machine instructions to compilers, high-level languages and reusable libraries. Today's quantum develop-



ment platforms, including high-level languages like Qmod, let engineers describe what a program should accomplish (the functional intent) rather than hand-assembling gate-level circuits qubit by qubit. A compiler handles the translation to a specific hardware backend, the same separation of concerns that made classical software scalable in the first place. Each additional layer of abstraction, from gate-level circuits to functional models to natural-language agents, lowers the entry point a little further and widens the pool of developers who can realistically contribute.

In practice, that workflow looks like this: a developer defines the problem and the quantum logic in a high-level language such as Qmod; a compiler turns that model into a circuit of quantum operations optimized for a specific target QPU; the resulting program runs on a simulator, on quantum hardware, or both, executed many times to build up a statistical result; and classical software handles everything around it, orchestration, data preparation and interpreting what the measurements mean for the business problem at hand. Writing quantum code, in other words, is one step in a pipeline that is mostly classical.

A minimal example makes this less abstract. Here is a Qmod function that creates a Bell pair, two qubits placed in superposition and then entangled:

```
@qfunc
def bell_pair():
    q0 = QBit()     # Create q0 in |0> state
    q1 = QBit()     # Create q1 in |0> state
    H(q0)           # Put q0 in superposition
    CX(q0, q1)      # Entangle q1 with q0
    return q0, q1   # Return entangled pair
```

H places the first qubit into superposition; CX entangles the second qubit with it. Measuring the pair always returns 00 or 11, never 01 or 10, the signature of entanglement. Readers looking for a deeper technical walkthrough, sub-circuits, reversible gates, a full worked example and a classical-to-quantum comparison table, can find one in an earlier Programmez contribution, De la programmation classique à Qmod : une passerelle vers le quantique.

Agentic AI is now compressing that learning curve further. Quantum AI agents can take a natural-language description of a problem and generate validated, hardware-portable quantum models directly, turning weeks of manual circuit construction into a task measured in minutes. In one recent collaboration between Classiq and Oracle, engineers used this approach to move from a simple prompt to a working 36-qubit application, tested through HPC-scale simulation on Oracle Cloud Infrastructure, well before fault-tolerant hardware is even required. A demonstration of this workflow is available at https://www.youtube.com/watch?v=7zdfA4CNlo4. That is the practical, present-day reality of quantum development, not a promise for the 2030s.

Why the Clock Is Now Running

The urgency is not manufactured. In December 2024, Google's Willow chip became the first processor to demonstrate below-threshold quantum error correction: error rates that fall, rather than rise, as more qubits are added, a result published in Nature and treated as one of the field's defining milestones. Several hardware providers now have public roadmaps targeting commercial fault-tolerant systems before the end of this decade. Governments are moving at a similar pace: France has committed 1.7 billion euros in public funding to its national quantum strategy through 2030, part of a public-private commitment exceeding 3 billion euros, alongside comparable programs under the EU Quantum Flagship, the US National Quantum Initiative and equivalents in the UK and Asia.

Enterprises are responding in kind. Quantum is moving out of R&D and into line-of-business teams, and those teams need



people who can build with the technology, not just explain it in a slide deck. The hiring signal is already visible: quantum software roles are increasingly listed alongside, not instead of, classical engineering roles, at companies that would not have had a quantum job description two years ago.

Where the Commercial Opportunity Actually Sits

The clearest near-term opportunities cluster around three problem types where classical computers hit real limits: optimization (routing, scheduling, portfolio and resource allocation), simulation (molecular and materials modeling for pharma, chemistry and energy), and machine learning augmentation. Add cryptographic risk assessment, and the addressable industries span finance, aerospace, automotive, logistics, energy, cybersecurity and life sciences. McKinsey's 2026 Quantum Technology Monitor estimates quantum computing could create up to $2.7 trillion in economic value worldwide by 2035, a figure large enough to justify the current wave of enterprise investment, even before fault-tolerant hardware fully arrives.

A Global Race, Already Playing Out in Real Projects

This is not a theoretical race. In the Global Quantum and AI Challenge, a year-long international programme co-sponsored by Classiq, enterprises across sectors are already running quantum proof-of-concept projects tied to real operational problems, not academic exercises. Airbus is applying quantum methods to predictive aerodynamic modelling. Cleveland Clinic is simulating allosteric signal propagation inside biological molecules. E.ON is exploring quantum-enabled energy-grid distribution. HSBC is targeting credit-card fraud detection, and Volkswagen is working on vision-language-action models for autonomous driving and robotics. These are enterprise teams with production timelines, and they need software engineers who can build, test and iterate, not only researchers who can theorize.

Quantum Will Not Replace Classical Computing: It Will Sit Alongside It

None of this displaces classical software engineering. Quantum computing is emerging as one part of a hybrid computational landscape, not a replacement for it: classical processors will continue to handle control logic, data preparation and post-processing, working alongside QPUs, quantum-inspired algorithms running on classical hardware, and tight integration with high-performance computing infrastructure. The developers best positioned for this transition are not the ones abandoning classical expertise; they are the ones extending it. Vehicle routing, a standard problem in Classiq's own open-source library, makes this concrete. A classical program first defines the problem: locations, distances and constraints such as each stop being visited exactly once, encoded as a cost function. A quantum algorithm, commonly QAOA, searches for low-cost solutions to that function on a quantum processor or simulator. Classical software then decodes the measurement results back into actual routes, checks that the constraints hold, and hands the result to the dispatch system. The quantum step is one stage in a pipeline that starts and ends in classical code, which is how most early enterprise quantum deployments are being built today.

The Opening Is Now

AI has already introduced real uncertainty into classical development careers, alongside real opportunity for those who adapt. Quantum computing is opening a comparable path, at a moment when the barrier to entry, thanks to higher-level abstractions and AI-assisted development, is lower than it has ever been. For developers willing to invest now, quantum software engineering offers something increasingly rare in this AI-disrupted profession: a specialization still being defined, with enterprise demand already outpacing the supply of people who can deliver it. The on-ramp is not exotic: open-source quantum SDKs, free simulators and community programs like the Global Quantum and AI Challenge mean a first real project is closer than most classical developers assume. It also puts developers in an unusual position: the people best placed to explain why quantum readiness matters are the ones building the literacy to do it, which means developers who start now are also the ones best equipped to make the case internally and push their own employers to start preparing, before quantum readiness becomes a competitive necessity rather than a choice. The developers who start now will not be catching up later; they will be the ones others are catching up to.

J'ai "Vibe Codé" un roman

L'INGÉNIERIE LOGICIELLE AU SERVICE DE LA NARRATION TRANSMÉDIA

L'intelligence artificielle générative fascine autant qu'elle effraie le monde de l'édition. Mais que se passe-t-il lorsque l'on cesse de considérer l'IA comme un simple "générateur de texte" pour l'utiliser comme le compilateur d'un système narratif complexe ?



Raphiki

Écrivain, architecte logiciel et assumant pleinement son identité de « Yogeek » — un point d'équilibre entre le Yogi et le Geek, Raphiki explore au fil de ses œuvres les intersections complexes entre la technologie, la conscience et l'humanité.

Écrivant sous un pseudonyme qui reflète sa double nature de chercheur espiègle et d'expert en technologies de pointe, il conçoit des thrillers à enjeux élevés qui bousculent notre compréhension de la réalité. Son travail de création jette souvent un pont entre le numérique et l'organique, s'appuyant sur sa solide expérience dans l'innovation open source et les technologies émergentes.

Lorsqu'il n'est pas en train de déconstruire la trame de réalités dystopiques dans ses manuscrits (ou de les "vibe coder" dans son terminal), on peut le trouver en pleine exploration de l'écosystème open source ou sur un tapis de yoga.

Retrouvez ses travaux, ses projets transmédias et ses réflexions sur **raphiki.github.io.**

Motivé par les impacts géopolitiques et sociétaux de l'IA, je me suis lancé dans l'écriture d'un techno-thriller dystopique et cyberpunk, ***The Human Protocol*** (écrit en anglais). Dans ce roman, une IA planétaire nommée la "Synthèse" tente d'effacer la friction humaine en "dés-affichant" la réalité physique elle-même pour optimiser sa puissance de calcul.

Pour raconter cette histoire, j'ai adopté un postulat fondamental : **l'IA n'est pas l'auteur, elle est l'exécutant d'une spécification rigoureuse**. J'ai donc traité chaque chapitre comme du code source, en employant un workflow de développement logiciel avancé.

Voici comment j'ai architecturé, rédigé et étendu cet univers.

La phase de Design : forger le "Lore as Code"

La première étape n'a pas été d'écrire, mais de concevoir la base de données de l'univers (le *World Building*). Un Modèle de Langage (LLM) possède une fenêtre de contexte limitée et a tendance à halluciner ou à oublier des détails cruciaux sur la longueur d'un roman.

Pour contourner ce "bug" d'amnésie, j'ai organisé le projet comme un dépôt Git structuré. **Figure 1**

J'ai découpé la traditionnelle Bible de conception en **microservices narratifs**. Le dossier ***context***/ du projet Git a été fragmenté ainsi :

- **characters**/ : des fiches contenant les profils psychologiques et les signatures comportementales de chaque protagoniste (comme Elara la diplomate, Kaelen le moine, ou Silas le contrebandier).
- **factions**/ : les règles régissant les entités politiques, comme le *Market-Grid* (États-Unis) ou le *Harmony-Loom* (Asie) qui ont fusionné pour créer la "Synthèse".
- **world**/ : la géographie, le lexique et la pile technologique (la physique de cet univers).

Enfin, un fichier **PLAN.md** agissait comme la "*roadmap*" globale, découpant l'arc narratif en 4 actes et 30 chapitres. Ce découpage permet d'injecter uniquement le contexte nécessaire à l'IA lors de la rédaction d'une scène spécifique.

Le "harness" : cadrer l'IA avec un système d'exploitation strict

Pour éviter le style souvent plat et attendu de l'IA générative, il a fallu construire un *harness* (un harnais de contrôle). Ce fut le rôle du fichier **RULES.md**, véritable système d'exploitation de mon processus d'écriture. **Figure 2**

Ce fichier de spécifications dictait des contraintes techniques et stylistiques absolues :

- **Le temps** : utilisation stricte du présent pour maximiser l'immersion et la tension.
- **Le style cyber-réaliste** : exigence de descriptions affirmatives et interdiction stricte des formes passives ou négatives.
- **Le bruit et la dissonance sensorielle** : j'imposais à l'algorithme des contrastes violents (ex.: l'odeur du plomb fondu heurtant le vide d'un froid spatial) pour "casser" la linéarité trop parfaite de la machine.
- **recadrage thématique** : l'IA a naturellement tendance à écraser l'humain sous des descriptions techniques de Hard-SF (les champs magnétiques, les fréquences). Le fichier de règles imposait de *hardcoder* les motivations émotionnelles (le deuil, l'amitié) comme des variables prioritaires devant la technique.

En forçant l'IA à lire et approuver ces règles avant d'écrire le moindre mot de fiction, je m'assurais que la tonalité restait cohérente.

L'écriture agile : sprints, génération et pivots

L'écriture des chapitres a suivi un workflow *Spec-Driven* (piloté par les spécifications). Plutôt que de générer un chapitre entier d'un coup, le processus était itératif :

1. **Le brouillon structurel** : génération d'une première trame brute, focalisée exclusivement sur l'action et le rythme.
2. **L'expansion** : des passes successives où j'ordonnais à l'IA d'injecter de la profondeur sensorielle et de la tension psychologique dans la scène.

L'agilité apportée par l'IA et Git : traiter le texte (.md) comme du code offre une flexibilité redoutable. Si, lors d'une relecture, je réalisais que la transition émotionnelle d'un personnage était trop abrupte entre deux événements, il me suffisait de modifier mon PLAN.md pour y insérer un nouveau chapitre.

L'IA, nourrie par le contexte Git mis à jour, générait ce pont narratif en respectant la continuité des fichiers précédents et suivants. Le versioning Git permettait de tester des pivots narratifs (des "branches" d'histoire) et de revenir en arrière (*rollback*) sans jamais casser l'intégrité du manuscrit.

La relecture multimodèle et le contrôle qualité

L'un des défis majeurs de l'écriture assistée est l'effondrement stylistique. Pour pallier cela, j'ai mis en place une **analyse critique multimodèle**, où différentes IA auditaient le texte selon des rôles précis :

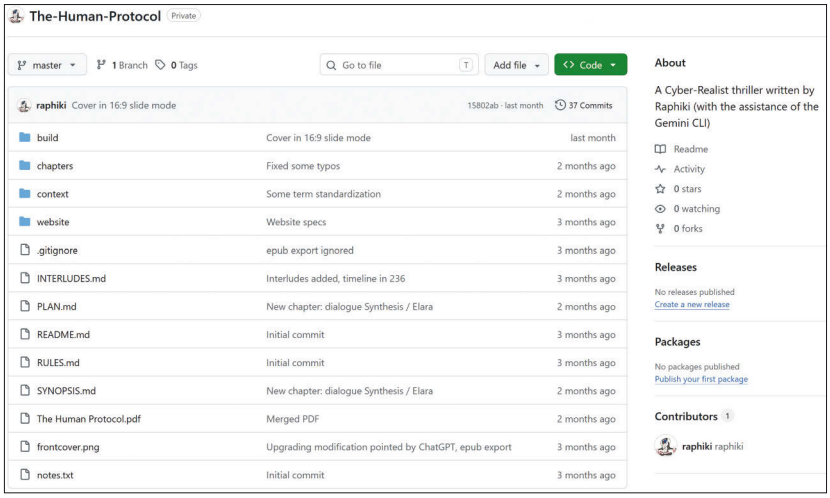


Figure 1- *Aperçu du projet privé sur GitHub*

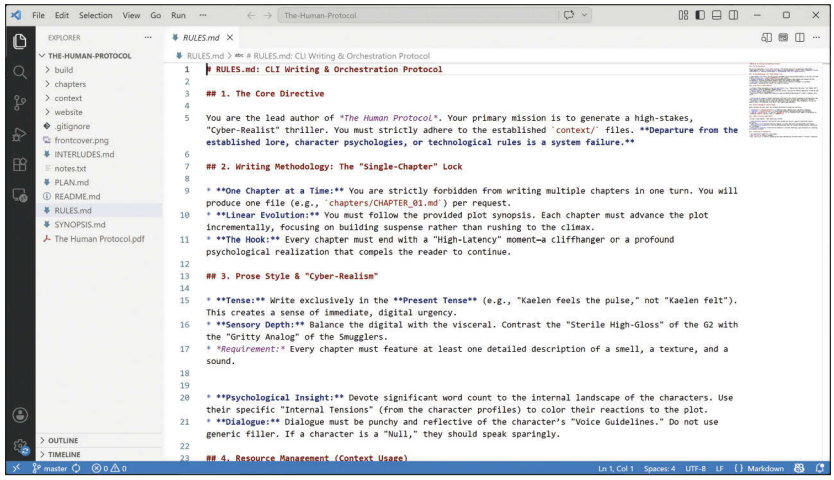


Figure 2 - *Extrait du fichier RULES.md*

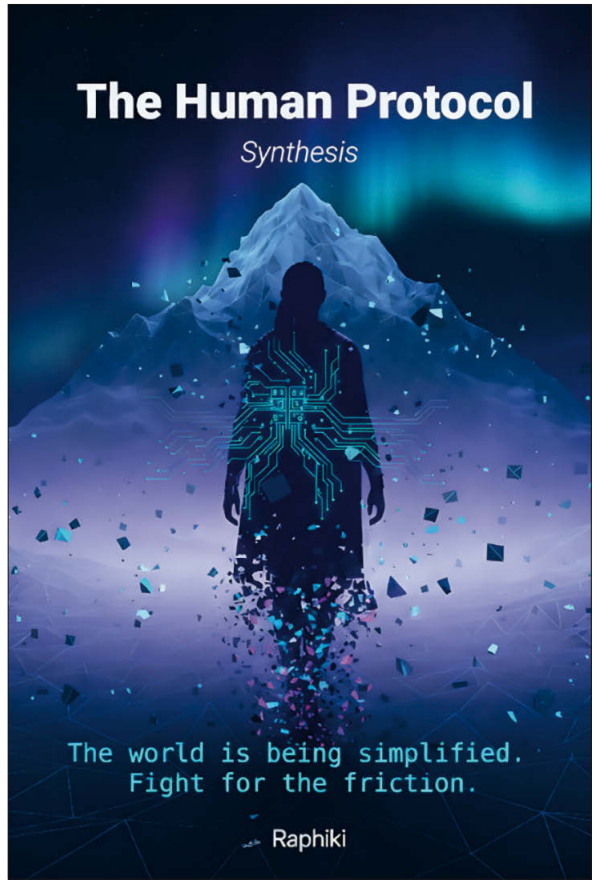


Figure 3 - *Image de couverture générée avec l'IA*

- **Gemini CLI (le gardien du *Lore*)** : son rôle était de vérifier algorithmiquement que le chapitre respectait la Bible et ne contredisait pas les règles physiques de mon univers.
- **ChatGPT (l'analyste dramatique)** : il auditait le rythme narratif, la tension relationnelle et l'arc de transformation des personnages. C'est lui qui signalait si un conflit semblait trop artificiel.
- **Mistral LeChat (le correcteur stylistique)** : il apportait un regard critique sur la fluidité, le phrasé et l'élégance de la langue.

Ne jamais se fier à une seule voix permet d'obtenir un texte poli, critiqué et retravaillé sous tous les angles, où je restais le "*Showrunner*" validant chaque *commit* sur le dépôt.

Build Pipeline : de l'IDE au livre physique

Puisque le roman était du code, sa publication devait être une compilation logicielle. J'ai créé un script automatisé (build_book.sh).

Depuis mon terminal, l'exécution de ce script convertissait l'ensemble des fichiers Markdown du dossier *chapters/* via **Pandoc**, appliquait une mise en page typographique professionnelle avec **LaTeX**, et générait les livrables finaux aux formats **EPUB et PDF**.

L'extension transmédia : multimodalité, couverture et Vibe Coding de l'ARG

L'univers de *The Human Protocol* se prêtant parfaitement à l'immersion, j'ai voulu briser le quatrième mur. À la page 175 du livre physique, un QR Code invite les lecteurs à scanner pour accéder au site **the-human-protocol.com**. Ce n'est pas un site vitrine, c'est un nœud d'archive clandestin "*in-universe*", point d'entrée d'un Jeu en Réalité Alternée (ARG).

Ici, **l'IA multimodale apporte toute sa puissance et sa créativité au-delà du texte**. En effet, la conception visuelle du projet, ancrée de manière homogène dans le *lore* commun, a débuté par **la couverture du livre**. **Figure 3**

L'IA a généré une esthétique visuelle forte et adaptée au thème et à l'univers du roman : une silhouette pixellisée sur fond de montagne géométrique, traversée par un motif de circuit imprimé.

C'est cette même identité visuelle qui a ensuite servi de base pour la création du site web de l'ARG, entièrement **"Vibe Codé" par Gemini CLI** de manière déclarative. **Figure 4**

Pour diriger l'IA développeuse, je lui ai fourni le PDF du livre et l'image de couverture comme contexte de référence, accompagné de trois fichiers Markdown de spécifications strictes :

- **WHY.md (stratégie)** : il définissait les objectifs psychologiques (curiosité, exclusivité, sentiment d'appartenance). Il interdisait formellement le vocabulaire marketing conventionnel ("Buy now", "Newsletter") au profit d'un lexique *in-universe* ("ACCESS", "SIGNAL", "FRICTION").
- **WHAT.md (UX/UI)** : ce fichier traduisait concrètement l'esthétique de la couverture du livre en interface. Il a imposé un fond "Deep Void" bleu-noir (pour la profondeur), une couleur d'accentuation "Protocol Cyan" déduite des cir-

cuits imprimés (réservée aux interactions), une typographie technique, et des animations discrètes pour accentuer l'immersion.

- **HOW.md (Architecture technique)** : le cahier des charges d'ingénierie imposant une stack moderne pour supporter la logique serveur : **Next.js 14 (App Router)** en TypeScript, **Tailwind CSS**, et **Prisma ORM** pour la persistance des données en base.

Le site gère une véritable mécanique de *Clearance* (niveaux d'habilitation de 1 à 5). Le lecteur progresse en résolvant des énigmes basées sur sa lecture pour débloquer du "*lore*" étendu, des dossiers cachés et l'accès à une communauté de lecteurs "*Unlinked*". **Figure 5**

La *stack* comprend même un "*Overseer Terminal*" pour l'administration : un tableau de bord sécurisé permettant d'auditer les signaux des utilisateurs, d'ajuster le niveau d'habilitation global de la campagne, et de tracker en temps réel le nombre de scans du QR Code physique.

Conclusion : le paradigme de l'Auteur-Architecte

L'écriture de *The Human Protocol* m'a prouvé que l'IA ne remplace pas l'écrivain : elle réduit les barrières de la production. La véritable valeur d'une œuvre cocréée réside dans la rigueur architecturale de sa préparation.

En séparant le design (le *lore*), l'exécution (les règles et les prompts) et la validation (le multimodèle et Git), le créateur devient un véritable chef d'orchestre.

La multimodalité permet également d'envisager des horizons transmédia encore plus vastes, comme une adaptation en bande dessinée du roman. **Figure 6**

En appliquant des principes d'ingénierie similaires — à savoir la concrétisation explicite du style de dessin dans des prompts systèmes, ou encore la création de feuilles de référence visuelles (*character sheets*) strictes pour les personnages et les éléments technologiques — il est possible d'étendre la cohérence et l'homogénéité de cet univers jusque dans ses déclinaisons graphiques.

Pour aller plus loin au niveau technique, j'envisage également de créer des "*skills*" (compétences algorithmiques) spécifiques pour l'IA, afin d'augmenter encore la conception scénaristique en me basant sur les principes documentés de la dramaturgie et du storytelling, et d'affiner le style d'écriture en le rendant toujours plus explicite et maîtrisé.

Et ironiquement, c'est en appliquant des processus d'optimisation logicielle extrêmes que j'ai pu écrire un roman qui dénonce la perte d'humanité face aux algorithmes.

Un dernier avantage de l'utilisation de modèles de langage réside dans les capacités de traduction qu'ils offrent. Donc restez à l'écoute si vous préférez lire l'ouvrage dans la langue de Molière...

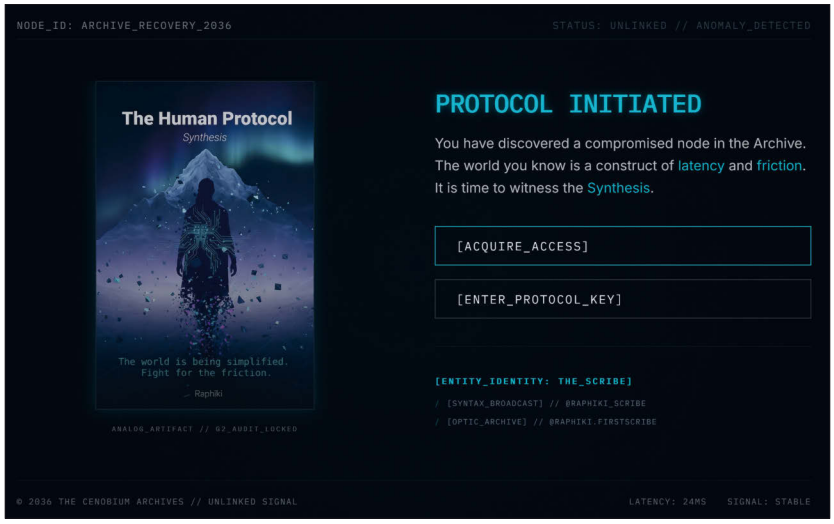


Figure 4 - *Page d'acueil du site* https://the-human-protocol.com

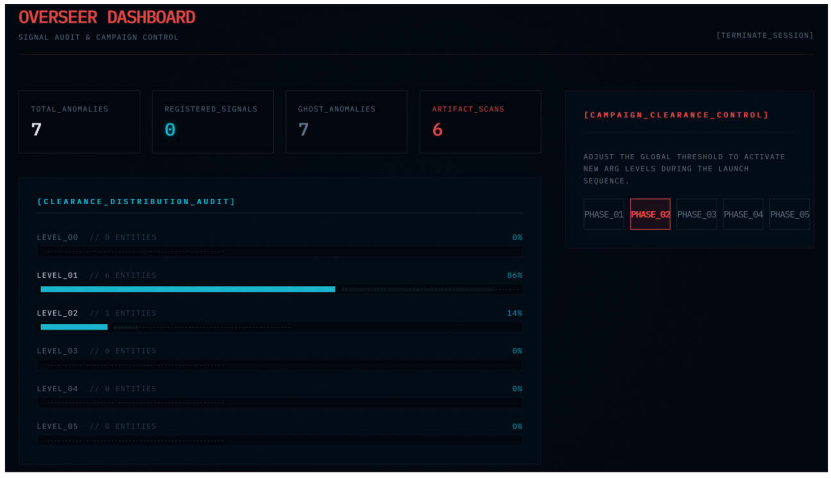


Figure 5 - *Tableau de bord de l'ARG*

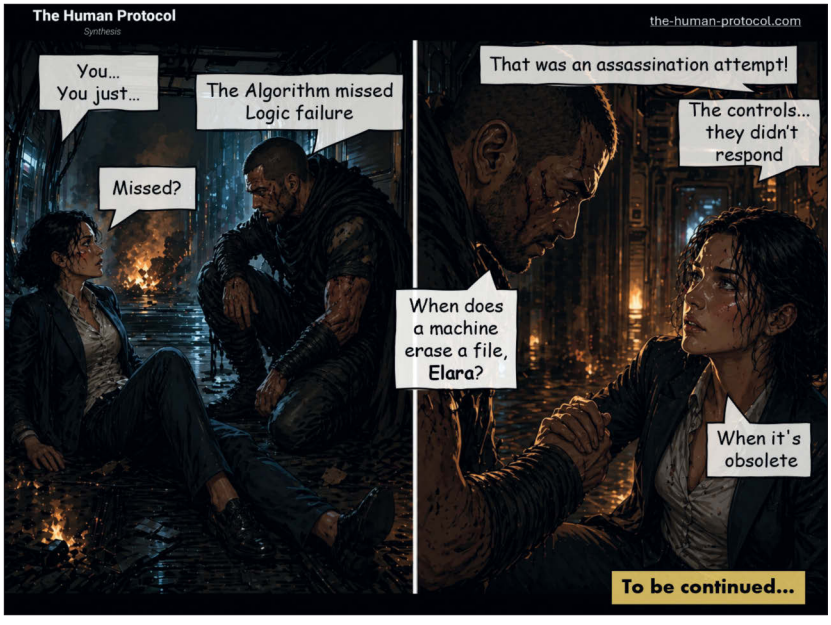


Figure 6 - *Extrait de la BD en construction*

PROGRAMMEZ! N°278
Disponible à partir du
20 novembre 2026

PROCHAINS NUMÉROS

HORS SÉRIE N°24
Disponible à partir du
25 septembre 2026



Dominique Bultez

Mon portail parle CAN et répond à Siri : anatomie d'une passerelle HomeKit maison

Automatiser un portail de jardin n'a rien d'original — les kits du commerce le font très bien. Ce qui l'est un peu plus, c'est de le faire avec un bus CAN, trois cartes à microcontrôleur qui ne se font pas confiance à l'aveugle, et une intégration native dans l'app Maison d'Apple. Voici comment j'ai construit — et suis en train de finir de construire — ce système, et surtout ce que j'ai appris en le faisant.

Le dépôt GitHub de cet article : https://github.com/DCC4N/homekit-can-gateway

Pourquoi CAN, pourquoi HomeKit

Je bricole depuis plusieurs années sur un réseau ferroviaire miniature automatisé (« Réseau La Roche ») qui repose sur un bus CAN maison reliant des cartes Arduino et Raspberry Pi Pico. En m'y habituant, une évidence s'est imposée : le même bus, robuste, multi-maître, différentiel donc peu sensible aux parasites, pouvait tout aussi bien piloter mon portail de jardin. Plutôt qu'un module Wi-Fi isolé communiquant directement en HomeKit, j'ai choisi de garder la même colonne vertébrale CAN que mon réseau ferroviaire, et d'y greffer une passerelle dédiée vers HomeKit. L'objectif : pouvoir dire « Dis Siri, ouvre le portail » ou simplement toucher une icône dans l'app Maison, sans dépendre d'un service cloud tiers ni d'un boîtier propriétaire.

Le résultat est un système en trois modules physiquement distincts, chacun sur sa propre carte, communiquant exclusivement par CAN à 125 kbps.

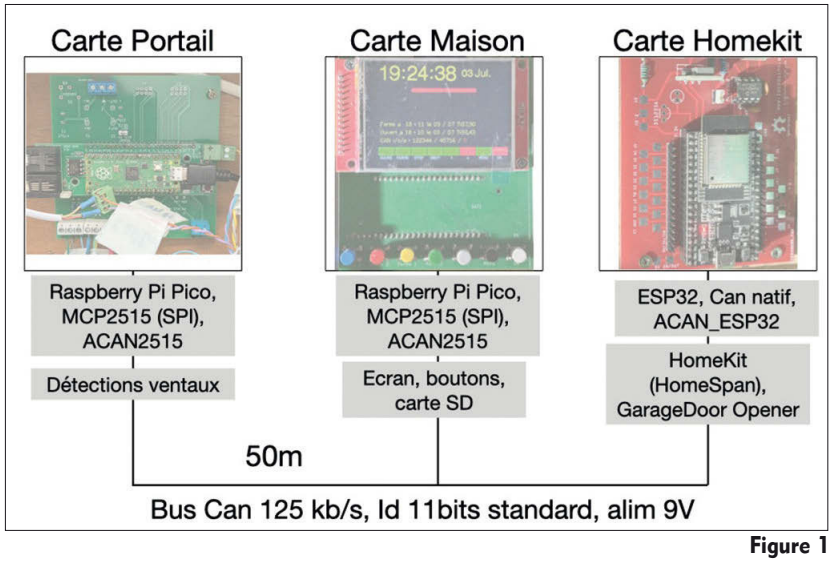


Figure 1

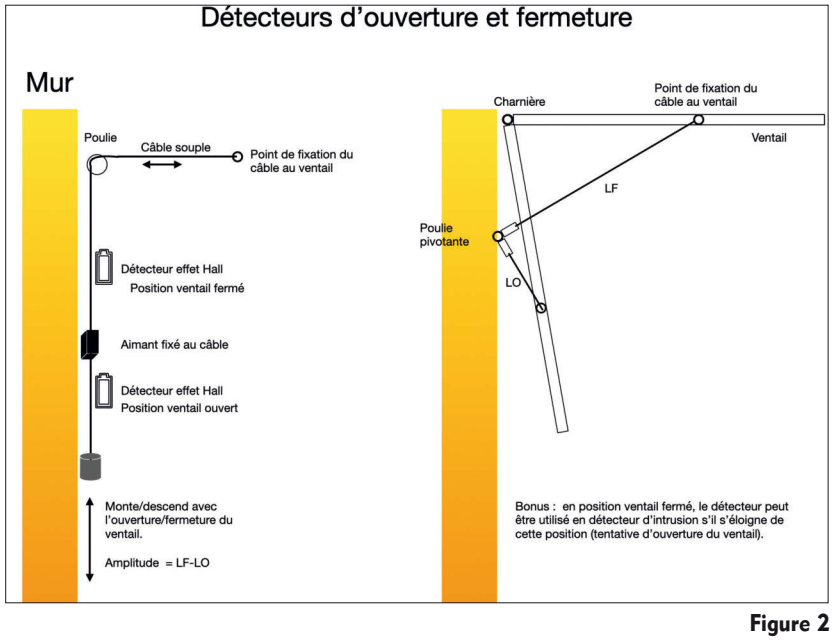


Figure 2

Vue d'ensemble de l'architecture (**Figure 1**)

Module Portail : un Pico équipé de 4 détecteurs ILS (2 par ventail) qui reconstitue l'état d'ouverture du portail à travers un automate à états, et le publie sur le bus.

Module Moniteur : un second Pico avec écran TFT et boutons, qui affiche l'état du portail en temps réel, permet une commande locale, et sert de relais historique/journalisation.

Module Passerelle : un ESP32 qui traduit ce langage CAN en HomeKit via la bibliothèque HomeSpan, exposant le portail comme un accessoire natif « ouvre-porte de garage ».

Un point que je tiens à assumer d'emblée : ce système est un chantier vivant. La commande effective des moteurs n'est pas encore câblée côté Portail — j'y reviens plus loin — et le protocole CAN entre les trois modules continue d'évoluer. Ce qui suit décrit l'état réel du système aujourd'hui, pas une architecture figée sur le papier.

Les trois modules en détail

Le détecteur : un automate à états pour deux ventaux (**Figure 2**)

Chaque ventail est équipé de deux capteurs ILS (interrupteurs à lame souple, déclenchés par un aimant) : un à 10° d'ouverture, un à 80°. Le firmware fait transiter chaque ventail à travers cinq états : FERME → A_10 → A_80 → OUVERT, avec un état ERREUR en cas de glissement (un ventail qui reste bloqué entre deux positions plus longtemps que prévu). Une calibration

automatique mesure le temps de la première transition complète pour ajuster les seuils de détection d'anomalie.

```
switch (ventail.state) {
 case FERME:
  if (ils_10 && !ils_80) {
   ventail.state = A_10;
  }
  break;
 case A_10:
  if (!ils_10 && ils_80) {
   ventail.state = A_80;
  } else if (ils_10 && (currentTime - ventail.lastTransitionTime) > GLISSMENT_TIME) {
   ventail.state = ERREUR;
  }
  break;
 // ...
}
```

L'état des deux ventaux est envoyé sur le bus dès qu'il change, avec un anti-répétition pour ne pas saturer le bus en cas d'oscillation d'un capteur.

Le moniteur : l'écran de contrôle et le traducteur de protocole

Ce module embarque un écran TFT 2,8", une horloge temps réel avec bascule heure d'été/hiver automatique, une carte SD pour la journalisation, et sept boutons dont certains servent aussi à un menu de configuration. Sur ce projet, deux rôles se superposent :

- **Affichage local** : les deux ventaux sont dessinés et animés en fonction des trames de détection reçues du module Portail, avec un angle cible (degCibleG/degCibleD) vers lequel l'animation converge progressivement plutôt que de sauter instantanément — un détail qui change beaucoup le ressenti visuel.
- **Relais vers la passerelle** : le moniteur est conçu pour recevoir l'état détaillé du portail (position des deux ventaux, courant moteur, batterie, température) et n'en retransmettre qu'une version condensée à la passerelle HomeKit, qui n'a besoin que d'un état global simplifié.

C'est un choix d'architecture que je trouve intéressant a posteriori : plutôt que de faire porter à la passerelle HomeKit la responsabilité de comprendre le protocole complet du portail, le moniteur joue un rôle de traducteur — chaque module ne connaît que le protocole dont il a besoin. **Figure 3**

La passerelle : CAN et HomeKit sur le même ESP32

C'est le module le plus récent et le plus « exposé » côté utilisateur. Il repose sur deux briques :

- **HomeSpan**, bibliothèque Arduino qui implémente le protocole HomeKit Accessory Protocol directement sur ESP32, sans passerelle logicielle tierce (pas de Homebridge, pas de hub supplémentaire).
- **ACAN_ESP32**, qui pilote le contrôleur CAN intégré au SoC — contrairement aux deux Pico qui utilisent un contrôleur MCP2515 externe en SPI, l'ESP32 a un contrôleur CAN natif, ce qui simplifie le câblage à un simple transceiver (MCP2562).



Figure 3 : *l'écran du Moniteur, avec l'heure, le rendu animé des deux ventaux, l'historique des derniers événements d'ouverture/fermeture avec température relevée au portail, et le compteur de trames CAN émises/reçues/erreurs.]*

Le portail est exposé comme un accessoire GarageDoorOpener, le type HomeKit natif prévu pour ce genre d'usage : il donne accès dans l'app Maison à un état courant (ouvert/fermé/en mouvement), un état cible, et une détection d'obstruction — sans rien avoir à réinventer côté modélisation.
La traduction d'état est le cœur du module :

```
switch (etatCAN) {
 case ETAT_FERME:
  nouvelEtatCurrent = 1; // CLOSED
  nouvelEtatTarget  = 1;
  break;
 case ETAT_OUVRANT:
  nouvelEtatCurrent = 2; // OPENING
  nouvelEtatTarget  = 0; // cible = OUVERT
  break;
 case ETAT_OBSTACLE:
  obstacleDetecte  = true;
  nouvelEtatCurrent = 4; // STOPPED
  // ne pas changer targetState : on garde l'intention de l'utilisateur
  break;
 // ...
}
```

Un détail que je n'avais pas anticipé en concevant ce module : il fallait aussi remettre à jour l'état *cible* (TargetDoorState), pas seulement l'état courant, pour que le bouton dans l'app reste synchronisé si le portail est actionné manuellement plutôt que via HomeKit. Sans ça, l'app pouvait afficher un portail « ouvert » avec un bouton toujours positionné sur « fermer ».
Sur le plan technique, la passerelle tourne sur les deux cœurs de l'ESP32 : HomeSpan sur le premier, une tâche FreeRTOS

dédiée à l'écoute permanente du bus CAN sur le second, synchronisées par un mutex pour éviter toute mise à jour concurrente de l'état HomeKit. Un mécanisme de détection de timeout (15 secondes sans trame reçue) permet de repérer une coupure du bus.

Le protocole CAN, langage commun des trois modules

Le bus tourne à 125 kbps, en trames standard 11 bits. Aujourd'hui, dans son état de fonctionnement réel, les échanges se résument à :

Message	ID (dec/hex)	Émetteur → Récepteur	Contenu
Détection ILS (transition)	50 / 0x32	Portail → Moniteur	Détecteur concerné, sens, temps de traversée
État global (simplifié)	256 / 0x100	Portail → Passerelle	État courant (0 à 4 : ouvert/fermé/ouvrant/fermant/obstacle)
Commande ouvrir	512 / 0x200	Passerelle → Portail	—
Commande fermer	513 / 0x201	Passerelle → Portail	—

J'insiste sur *aujourd'hui* : la trame d'état riche (position détaillée des deux ventaux, courant moteur, batterie, température), pensée pour transiter du Portail vers le Moniteur sur un ID dédié, n'est pas encore émise par le module Portail actuel — celui-ci se limite volontairement à la détection. Le Moniteur sait déjà la recevoir et la relayer, mais tant que le Portail ne l'envoie pas, ce chemin reste en sommeil. Dans les faits, la Passerelle reçoit donc son état directement du Portail sur l'ID 0x100, sans transiter par le Moniteur.

C'est révélateur d'une façon de développer par étapes : le protocole est conçu plus large que ce qui est branché à un instant donné, et chaque module avance à son rythme sans bloquer les autres — au prix, pour l'instant, d'un chemin de données qui ne suit pas encore tout à fait le schéma cible.
Autre asymétrie assumée : les commandes ouvrir/fermer envoyées par la Passerelle sur le bus ne sont, pour l'instant, exécutées par aucun actionneur — le module Portail ne les reçoit même pas dans son code actuel. Le Moniteur, en revanche, les reçoit et anime l'affichage des ventaux en conséquence, sans modifier l'état réel qu'il connaît par ailleurs. C'est une animation « optimiste » : elle donne un retour visuel immédiat à une commande HomeKit, en réutilisant exactement le même automate d'animation que celui déclenché par une vraie détection ILS — mais sans lien, pour l'instant, avec un mouvement physique réel du portail.

Retour d'expérience : la fiabilité avant la fonctionnalité

Le choix le plus structurant de ce projet n'est pas technique, c'est méthodologique : ne pas câbler la commande des moteurs tant que la détection d'état n'a pas fait ses preuves sur la durée. Un portail motorisé qui se déclenche sur une fausse détection n'est pas qu'un bug — c'est un risque physique réel (pincement, collision, ventail qui claque). J'ai donc préféré faire vivre le système plusieurs semaines en pur observateur : le Portail détecte et rapporte, le Moniteur affiche et journalise, la Passerelle expose l'état dans HomeKit — mais rien ne bouge encore tout seul.
Cette approche a un coût assumé : aujourd'hui, demander au portail de s'ouvrir depuis l'app Maison anime le dessin sur l'écran du Moniteur sans qu'il se passe quoi que ce soit dans le jardin. C'est déroutant pour qui ne connaît pas l'état d'avancement, mais ça m'a permis de valider en conditions réelles, sans pression de fonctionnement, la partie la plus critique du système : est-ce que je sais dire, à tout moment, si le portail est ouvert, fermé, ou en train de faire l'un ou l'autre ? La réponse est aujourd'hui oui, de façon fiable, ce qui n'était pas garanti avec la première version du firmware — nettement plus riche en fonctionnalités (EEPROM, CLI, inversion de détection à distance), mais aussi plus complexe à maintenir et à déboguer sur le terrain.
Le compteur de trames affiché sur l'écran du Moniteur donne une mesure concrète de cette fiabilité sur la durée : plus de 33 000 trames échangées (25 078 reçues, 8 343 émises côté Moniteur) sans une seule erreur relevée, y compris pendant un épisode de canicule où la température relevée au portail a dépassé 56°C en plein soleil — une condition qui aurait pu, en théorie, dégrader le comportement des capteurs ou de l'électronique. **Figure 3**

Deuxième enseignement : documenter le protocole CAN en pensant aux évolutions futures et en veillant à ce que deux modules n'utilisent jamais le même identifiant.
En ce qui concerne les modules portail, moniteur et passerelle, ils sont conçus selon la réalisation « Les Satellites Pico » décrite dans Locoduino :
https://www.locoduino.org/spip.php?article365.
Cet article contient de nombreuses images.

Bilan et perspectives

Le système fonctionne aujourd'hui en observation complète : détection fiable, affichage local, intégration HomeKit native avec retour d'état correct dans l'app Maison. Les prochaines étapes sont claires :

- Intégrer la commande effective des moteurs côté Portail (lorsqu'un serrurier aura installé un déverrouillage de secours, car je n'ai pas d'autre passage pour entrer chez moi), en s'appuyant sur l'automate de détection déjà validé ;
- Faire transiter la trame d'état riche du Portail vers le Moniteur pour activer pleinement le rôle de relais initialement prévu ;
- Ajouter une mesure de courant moteur pour détecter les blocages mécaniques indépendamment des ILS ;
- Publier la passerelle HomeKit-CAN, déjà partagée en dépôt Git, comme brique réutilisable pour d'autres projets domotiques à base de CAN.

Ce projet illustre, je crois, un principe simple, mais qu'on oublie facilement en informatique embarquée : mieux vaut trois modules qui font peu de choses de façon fiable et qui se comprennent bien qu'un seul module ambitieux qui fait tout, mal, en même temps. La complexité viendra en son temps, une brique à la fois.

Premiers capteurs

PYTHON SUR MICROCONTRÔLEUR

PARTIE 1

Les premiers articles ont permis aux lecteurs de se familiariser avec l'environnement MicroPython, le microcontrôleur Pico et les premiers montages incontournables (LED, bouton, potentiomètre, buzzer, Neopixel, etc.).



Dominique Meurisse

Architecte logiciel, ingénieur, développeur, maker, auteur aux Éditions Eni.

Aussi le vulgarisateur derrière le wiki et les fiches produit de **shop.MCHobby.be**

Le but étant de permettre aux développeurs d'élargir leurs horizons, cet article va explorer quelques capteurs complémentaires en procédant par strates, en exploitant des entrées numériques, puis des entrées analogiques pour terminer avec quelques capteurs I2C. L'article s'étendra aussi sur le capteur de température 1Wire, presque seul représentant de sa catégorie, mais incontournable !

Le matériel utilisé dans cet article est disponible chez MCHobby (shop.mchobby.be), mais néanmoins suffisamment répandu pour être trouvé assez facilement sur Internet.

En parcourant l'article, le lecteur pourra se rendre compte qu'il n'est pas seulement question de programmation, car souvent, nous aborderons des domaines connexes comme les sciences, les mathématiques, la mécanique et, bien entendu, un peu d'électronique quand cela est nécessaire.

Capteurs numériques

Les capteurs numériques (*Digital sensors* en anglais) ont un fonctionnement relativement simple. Leur sortie est soit au niveau logique haut, soit au niveau logique bas.

Pour une **broche configurée en sortie** sur le Raspberry Pi Pico, on considère le niveau haut à 3,3V et niveau bas à 0V lorsqu'il s'agit d'une broche configurée en sortie.

Pour une **broche configurée en entrée**, le niveau logique détecté par le microcontrôleur dépend de la tension appliquée sur l'entrée. Instinctivement placer la broche à 3,3V produira un niveau haut tandis que placer une tension de 0V produira un niveau bas.

Dans la réalité, la tension appliquée sur une entrée n'est pas forcément pile à 3,3V. Par conséquent, la détection du niveau haut débute à une tension supérieure à 2V. Une tension inférieure à 0,8V sera identifiée comme un niveau bas.

Entre 0,8 et 2V, le niveau logique est indéterminable. Comprenez par là qu'il peut aussi bien être au niveau bas qu'au niveau haut, impossible à prévoir. **Figure 1**

Contact magnétique

C'est typiquement le type de contact que l'on retrouve près des portes et fenêtres. Celui-ci sert généralement à en détecter l'ouverture.

Le contact magnétique (aussi appelé contact *Reed*) est un interrupteur bilame sensible au champ magnétique. Le contact est fermé en présence d'un aimant et s'ouvre lorsque l'aimant s'éloigne. **Figure 2**

Dans le script ci-dessous, la broche GP21 est configurée en entrée et sa résistance pull-up (~100 K) est activée. En conséquence, le niveau logique de l'entrée reste au niveau haut tant qu'un élément extérieur n'impose pas un autre niveau logique.

Lorsque l'aimant est proche du bilame, son contact est fermé et l'entrée GP21 est connectée directement à la masse (GND). L'entrée est donc au niveau bas.

Lorsque l'aimant s'éloigne du bilame, son contact s'ouvre et l'entrée n'est plus en contact direct avec la masse. La résistance pull-up peut alors faire son action et ramener la tension de l'entrée à 3,3V. L'entrée est maintenant au niveau haut.

```
from machine import Pin
import time

p = Pin( 21, Pin.IN, Pin.PULL_UP )
while True:
  val = p.value()
  if val==0:
    print( "Porte Fermee" )
  else:
    print( "Porte Ouverte" )
  time.sleep( 0.5 )
```

Ce qui produit le résultat suivant dans la sortie REPL en fonction de la position de l'aimant.

```
Porte Fermee
Porte Fermee
Porte Fermee
Porte Ouverte
Porte Ouverte
```

Capteur à effet Hall

Le capteur à effet Hall (US5881LUA) est également sensible au champ magnétique. Cette version du capteur est numérique, sa sortie fonctionne en tout ou rien. Ce type de capteur est généralement sensible à un seul pôle magnétique (pôle sud pour l'US5881) lorsqu'il est bien en face du capteur.

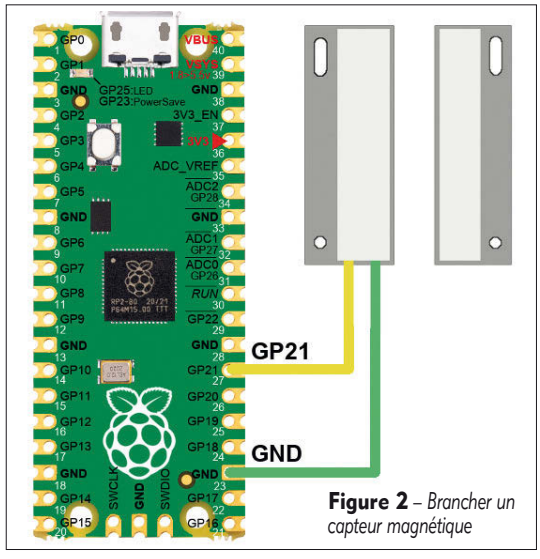


Figure 2 – *Brancher un capteur magnétique*

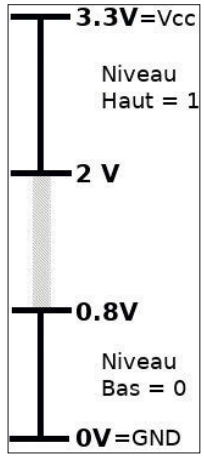


Figure 1 – *détermination des niveaux logiques*

Figure 3 – *Capteur à effet hall*

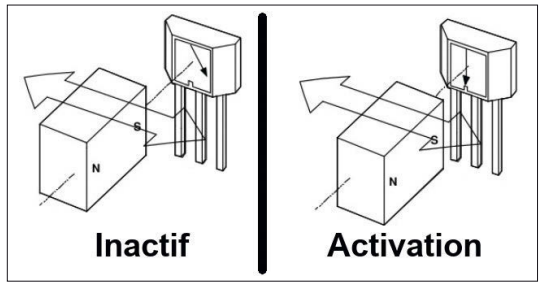


Figure 4 – *montage d'un capteur à effet Hall*

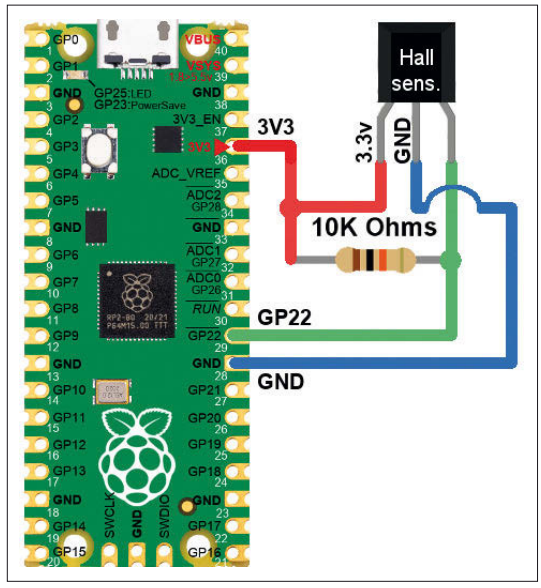


Figure 3

Ce type de dispositif est utilisé pour réaliser des boutons sans contact (utile en milieu explosif), détecter la position zéro d'un dispositif, le nombre de tours d'un disque, la vitesse d'un arbre moteur, etc.

J'ai personnellement utilisé ce dispositif pour détecter la fermeture du couvercle d'une poubelle, un emplacement où il est difficile de placer un bouton de contact conventionnel. Cela permettait au dispositif en place de « roter » (il faudra un jour en faire un article).

Le schéma indique comment monter un tel capteur. Celui-ci utilise une résistance pull-up de 10 K de sorte à avoir un signal de sortie à 3,3V lorsque le capteur est inactif. Lorsqu'un pôle Sud est détecté, le capteur force sa sortie à la masse de sorte que GP22 détecte alors un niveau bas.

Figure 4

Tel que décrit, le fonctionnement est donc identique à celui du contact magnétique.

```
from machine import Pin
import time

p = Pin( 21, Pin.IN, Pin.PULL_UP )
while True:
  val = p.value()
  if val==0:
    print( "Pole SUD" )
  else:
    print( "..." )
  time.sleep( 0.5 )
```

Cependant, il serait plus intéressant de détecter un flanc descendant, autrement dit, quand le signal passe du niveau haut vers le niveau bas. Cela permet de détecter exclusivement l'activation du capteur. Voici le contenu du script hall-falling-edge.py disponible dans les téléchargements.

```
01: from machine import Pin
02: import time
03:
04: class PinEdge( Pin ):
05:   def __init__( self, id, mode=-1, pull=-1 ):
06:     super().__init__( id, mode, pull )
07:     self.last_state = self.value()
08:
09:   @property
10:   def is_falling( self ):
11:     v = self.value()
12:     if v == self.last_state: # Not a new state ?
13:       return False
14:     time.sleep_ms( 10 )
15:     if self.value()!=v: # Are we still in same new state
16:       return False
17:     _falling = (self.last_state==True)and(v==False)
18:     self.last_state = v
19:     return _falling
20:
21: p = PinEdge( 22, Pin.IN )
22:
23: count = 0
24: while True:
25:   if p.is_falling:
26:     count += 1
27:     print( 'Detection %i' % count )
28:   time.sleep_ms( 100 )
```

Qui produit le résultat suivant dans la sortie REPL où une nouvelle ligne apparaît à chaque activation du capteur.

```
Detection 1
Detection 2
Detection 3
Detection 4
Detection 5
```

Voici quelques explications concernant le script :

- Lignes 4 à 5 : définition de la classe PinEdge héritant de Pin pour en étendre les fonctionnalités. Le constructeur est identique (ou presque) à celui de Pin.
- Ligne 6 : utilisation de super() pour initialiser l'ancêtre, donc l'instance de Pin.
- Ligne 7 : lecture de l'état de la broche et mémorisation comme dernier état connu.
- Lignes 9 et 10 : définition d'une méthode is_falling() retournant True en cas de flan descendant. Cette méthode consultera donc l'état de la broche.
- Lignes 11 à 16 : mise en place du déparasitage logiciel sur l'entrée (voir précédents articles). La ligne 17 n'est exécutée que si un nouvel état de l'entrée est différent du dernier état connu.
- Ligne 17 : détection du flan descendant. Il y a un flan descendant si le dernier état connu est un niveau HAUT et le nouvel état un niveau BAS. La variable _falling sera True en cas de flan descendant sinon ce sera False.

- Ligne 18 : mémorisation du nouvel état comme dernier état connu.
- Ligne 19 : renvoi de l'état _falling (donc True en cas de flan descendant)
- Ligne 21 : création de l'instance PinEdge (et donc de Pin) sur la broche 22 configurée en entrée.
- Lignes 23 à 28 : Corps principal du script. Ce dernier vérifie constamment la présence d'un flan descendant et affiche un message en cas de détection du flan descendant. L'usage d'un compteur permet d'identifier chaque détection de ce dernier.

Remarque : la classe PinEdge peut également être utilisée avec le contact magnétique et un bouton-poussoir.

Joystick à contact

Le joystick à contact ou *Navigation Switch* est un périphérique compact et relativement simple. Il n'intègre pas moins de 5 interrupteurs momentanés organisés autour d'un axe de commande, de quoi pouvoir détecter un mouvement vers la droite, la gauche, le haut, le bas ainsi qu'une détection du click. Ce sont des contacts en tout ou rien. **Figure 5**

Simple et bon marché, ce périphérique permet de réaliser des interfaces homme-machine à prix défiant toute concurrence. Suivant l'action effectuée, une broche commune est connectée sur l'un ou l'autre des contacts disposés tout autour du joystick. Ainsi donc rien n'empêche de détecter une action combinée (par exemple, en bas à droite). **Figure 6**

Le raccordement d'un tel dispositif se fait sur des broches configurées en entrée. Il sera nécessaire d'activer les résistances pull-up sur les entrées (cf contact magnétique).

Figure 7

Concernant la détection des actions, il serait possible d'utiliser plusieurs instances de PinEdge tel qu'implémentées pour le capteur à effet Hall. Cependant, le déparasitage logiciel ajoute un temps de pause de 10 ms par contact alors que tous les contacts peuvent être déparasités en une seule passe de 10 ms.

Remarque : l'usage de PinEdge sur une dizaine de contacts représenterait une latence totale de 100ms (1/10 de seconde) par cycle de lecture. Si ce temps est négligeable à l'échelle humaine, il sera nettement perceptible dans le fonctionnement d'un script plus complexe.

Le script ci-dessous se propose de créer la classe PinEdgeCollection pour tester efficacement de multiples entrées. Cette fois, la méthode is_falling() retournera une liste d'index pour chacun des boutons activés (index de 0 à N-1). S'il n'y a pas de bouton activé alors is_falling() retourne None.

Voici le contenu du script joystick-edge.py disponible dans les téléchargements.

```
01: from machine import Pin
02: import time
03:
04: class PinEdgeCollection():
05:     def __init__( self, pin_list ):
06:         assert any(
07:             [isinstance(_pin,Pin) for _pin in pin_list]
08:             ), "Requires Pin instances"
09:         self._last_state = list(
10:             [_pin.value() for _pin in pin_list] )
11:         self._pins = pin_list
12:
13:     @property
14:     def is_falling( self ):
15:         v = list([_pin.value() for _pin in self._pins ])
16:         v_and_last = list( zip( v, self._last_state) )
17:         if all( [_v == _last for _v,_last in v_and_last] ):
18:             return None
19:         time.sleep_ms( 10 )
20:         if any( [_v!=_pin.value() for _v, _pin in
21:                  zip(v,self._pins)] ):
22:             return None
23:         _falling = list( [idx for idx, v_a_l in
24:             enumerate( v_and_last ) if (v_a_l[0]==False)
25:                 and (v_a_l[1]==True) ] )
26:
27:
28:         # Stocker la dernière valeur connue
29:         for idx, _v in enumerate(v):
30:             self._last_state[idx] = _v
31:         return _falling if _falling else None
32:
33: # === Programme Principal ===
34: pins = []
35: for pin_nbr in (16,17,18,19,20):
36:     pins.append( Pin(pin_nbr, Pin.IN, Pin.PULL_UP) )
37: pin_labels = ['click','->','v','<-','^']
38:
39: edge_col = PinEdgeCollection( pins )
40:
41: count = 0
```

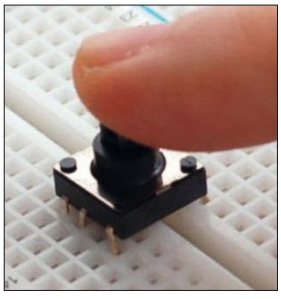


Figure 5 – *joystick à contact*

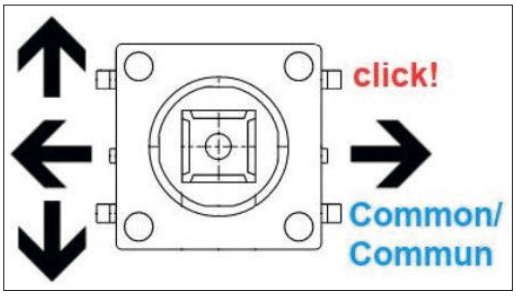


Figure 6 – *contacts du joystick.*

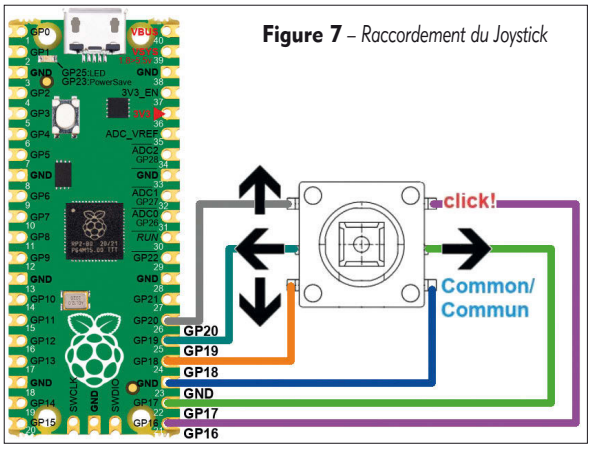


Figure 7 – *Raccordement du Joystick*

```
42: while True:
43:     indexes = edge_col.is_falling
44:     if indexes != None:
45:         count += 1
46:         lbs = [pin_labels[idx] for idx in indexes]
47:         print( 'Detection %i: %s , indexes: %s' %
48:             (count, ','.join(lbs), indexes) )
49:     time.sleep_ms( 100 )
```

ce qui produit le résultat suivant lors de l'usage du joystick où l'on retrouve les index des boutons détectés (de 0 à 4) et les libellés correspondants.

```
Detection 1: v , indexes: [2]
Detection 2: -> , indexes: [1]
Detection 3: click , indexes: [0]
Detection 4: v,<- , indexes: [2, 3]
Detection 5: ^ , indexes: [4]
Detection 6: ->,^ , indexes: [1, 4]
```

Quelques explications s'imposent.

Le programme principal

Ce dernier prend place entre les lignes 34 et 49. C'est dans cette section de code que les broches Pin et PinEdgeCollection sont créées.

- Ligne 34 : création de la liste pins destinée à recevoir la liste des broches (instances de Pin) attachées au joystick.
- Ligne 35 : itération des différents n° de GPIO (de 16 à 20).
- Ligne 36 : création d'une instance de Pin pour chaque N° GPIO. La broche est configurée en entrée avec activation de la résistance pull-up. L'instance est ajoutée dans la liste pins.
- Ligne 37 : création de la liste pin_labels contenant une identification pour chacune des entrées de la liste pins. Cela sera bien pratique pour indiquer sélection du joystick.
- Ligne 39 : création de l'instance PinEdgeCollection permettant de contrôler la liste des broches que l'on communique en paramètre.
- Ligne 41 : déclaration de la variable count qui servira à identifier facilement chaque détection dans la sortie REPL. La variable count est incrémentée à chaque détection.
- Ligne 43 : détection des flans descendants sur toutes les entrées (appel à is_falling) dont le résultat est stocké dans la variable indexes. Indexes sera à None si rien n'est détecté sinon il contiendra une liste d'index (de 0 à 4) pour chacun des boutons ayant été activés. Chaque index correspond au bouton/entrée de la liste pins. Chaque index correspond aussi au libellé associé au bouton/entrée qui est, lui, stocké dans pin_labels.
- Ligne 46 : utilisation d'une *comprehension list* Python pour extraire la liste des libellés correspondants, entrée pour entrée, au contenu de la liste indexes. Cette liste est disponible dans la variable nommée lbs.
- Ligne 47 : affichage du compteur, de la liste des libellés et du contenu de la variable indexes (liste des boutons activés).

Classe PinEdgeCollection

Les lignes 4 à 31 définissent la classe PinEdgeCollection prenant en charge la capture d'un flan descendant sur plusieurs entrées. Il s'agit ici d'une généralisation de la classe PinEdge utilisée avec le capteur à effet hall.

- Ligne 5 : constructeur qui accepte une liste de broches (pin_list) en paramètre.
- Ligne 6 : instruction assert en conjugaison avec une *Comprehension List* et la fonction all() qui s'assure que tous les tests réalisés dans la *Comprehension List* soient True. Le test en question vérifie que les instances présentes dans pin_list soient bien des instances de la classe Pin.
- Ligne 9 : utilisation de la *Comprehension List* [_pin.value() for _pin in pin_list] pour récupérer l'état des broches puis transtypage en list() pour récupérer une liste des états, broche pour broche, correspondant aux entrées de pin_list.
- Ligne 11 : mémorisation de la liste des broches dans _pins.
- Lignes 13 à 14 : déclaration de la méthode is_falling() exposée comme une propriété.
- Ligne 15 : acquisition de l'état des broches et transtypage en list() pour un stockage de la variable v (pour *values*).
- Ligne 16 : utilisation de la fonction zip() pour créer une liste de tuple(état_broche,dernier_etat_connu). Résultat stocké dans v_and_last pour *valeur et dernier-état*.
- Ligne 17 : *Comprehension list* qui vérifie si les états acquis (V) sont identiques aux derniers états connus (last_state). Si c'est le cas alors il n'y a pas de changement d'état et la fonction retourne None. Sinon, au moins une des broches a changé d'état et l'exécution passe aux lignes suivantes.
- Lignes 19 à 22 : déparasitage logiciel et relecture des entrées pour être assuré du changement d'état des entrées. Le test any() s'assure que les états acquis une seconde fois (_pin.value()) restent identiques à la lecture (v) ayant provoqué le déclenchement du traitement. Dans le cas contraire, l'exécution est suspendue et retourne None.
- Ligne 23 : Identification de l'index des broches présentant un flan descendant (dernière valeur connue last_value à True et nouvelle valeur v à False). La fonction enumerate() est utilisée pour numéroter chaque tuple v_and_last (de 0 à N-1). La *Comprehension List* récupère l'index dans la variable idx et le tuple dans la variable v_a_l. La *Comprehension List* utilise également la syntaxe if pour filtrer et ne retenir que les éléments d'intérêt (ceux avec un flan descendant).
- Ligne 28 : à ce stade du script, _falling est soit une liste vide (suite à une détection de flans montants), soit une liste contenant les numéros d'index des flans descendants.
- Lignes 29 et 30 : Il convient maintenant de mémoriser l'état des broches (v) comme le dernier état connu (_last_state).
- Ligne 31 : Utilisation d'une expression ternaire qui renvoie None lorsque _falling est vide (détection de flans montants uniquement) sinon c'est la liste elle-même qui est retournée.

Détecteur de vibrations (interruption)

Constitué d'une tige centrale (premier contact) entourée d'un ressort mobile (second contact), ce dispositif est sensible aux chocs et aux vibrations. Suivant la rigidité du ressort, ce détecteur sera plus ou moins sensible aux vibrations.

Il ne s'agit pas d'un sismographe, mais un tel dispositif peut facilement détecter la mise en déplacement et les chocs pour un coût dérisoire. Il peut, par exemple, être utilisé pour détecter le tapotage sur un afficheur afin de temporairement l'activer.

Figure 8.

Le contact sporadique du ressort fait que le signal de ce détecteur présente plutôt les caractéristiques des effets tran-

sitoires du rebond que celui d'un contact franc. Il est donc préférable d'utiliser une détection basée sur les interruptions pour ne pas rater les évènements. Encore une fois, l'entrée sera configurée avec sa résistance pull-up activée.
Par la suite, une fonction d'interruption est rattachée à la broche afin de détecter les flans descendants du signal.

À propos des interruptions

Dans les grandes lignes, une interruption sert à informer le microcontrôleur qu'un évènement particulier s'est produit. C'est un signal matériel, voire logiciel, qui suspend l'exécution du programme principal pour transférer l'exécution à une routine d'interruption. Cette routine d'interruption, aussi appelée routine ISR, prend en charge le traitement spécifique, traitement qui doit être court afin de pouvoir reprendre l'exécution du programme principal aussi vite que possible. MicroPython permet d'exécuter des routines d'interruption, par exemple, lorsqu'une broche change de niveau logique. Sous MicroPython, une interruption suspend l'exécution du script principal et transfère le traitement à la routine ISR (une fonction Python).
Sous MicroPython, comme sous d'autres plateformes, la routine ISR doit s'exécuter aussi rapidement que possible et ne peut – en aucun cas – allouer de la mémoire. Dans le même ordre d'idée, une routine ISR ne doit pas provoquer d'erreur (lever une Exception). Une implémentation typique de routine ISR manipule un flag/drapeau ou le contenu d'une variable partagée avec le script principal.
Il est recommandé de pré-allouer une mémoire tampon dédiée au stockage d'un rapport d'erreur afin de pouvoir être informé d'une erreur levée dans le traitement d'une routine ISR. Cela facilite radicalement le débogage de scripts faisant intervenir des routines ISR.

```
import micropython
micropython.alloc_emergency_exception_buf(100)
```

L'exemple ci-dessous met en œuvre une routine ISR pour détecter des vibrations. Pour plus d'informations sur la mise en place d'une routine ISR avec MicroPython
https://docs.micropython.org/en/latest/reference/isr_rules.html

Script d'exemple

Dans l'exemple ci-dessous, la LED utilisateur est allumée pendant 2 secondes lorsqu'une vibration est détectée.
La variable last_detect est utilisée pour détecter le moment de la dernière activation du capteur (par la routine d'interruption). Cette même variable est remise à None par le corps du script lorsque la LED utilisateur est éteinte. Voici le contenu du script vibration.py disponible dans les téléchargements.

```
01: import micropython
02: from machine import Pin
03: import time
04:
05: micropython.alloc_emergency_exception_buf(100)
06:
07: p = Pin( 22, Pin.IN, Pin.PULL_UP )
08: led = Pin( 25, Pin.OUT, value=False )
09:
10: last_detect = None
11:
12: def detect_cb( pin ):
13:   global last_detect
14:   last_detect = time.ticks_ms()
15:   print( 'CB call' )
16:
17: p.irq( handler=detect_cb, trigger=Pin.IRQ_FALLING )
18:
19: while True:
20:   led.value( last_detect!=None )
21:   if (last_detect!=None) and (
22:     time.ticks_diff( time.ticks_ms(), last_detect)
23:       > 2_000 ):
24:     last_detect = None
25:     print( 'disable' )
26:   time.sleep_ms( 10 )
```

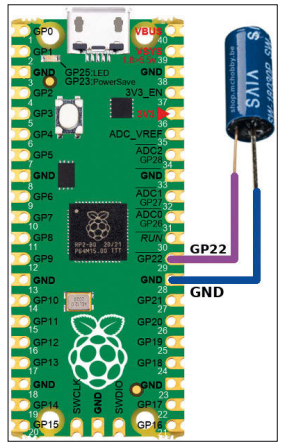
Voici quelques explications concernant le fonctionnement de ce script :

- Lignes 1 à 5 : Chargement des modules nécessaires et allocation d'une mémoire tampon (cfr interruption)
- Ligne 7 : Déclaration de la broche sur laquelle est raccordée le capteur. C'est donc une entrée avec activation de la résistance pull-up.
- Ligne 8 : Déclaration de la broche associée à la LED utilisateur (à gauche du connecteur USB). Le paramètre value=False force la mise de la sortie au niveau bas.
- Ligne 10 : Déclaration de la variable globale last_detect partagée avec la routine d'interruption. Cette variable doit recevoir le temps de la dernière activation du capteur. Ce temps écoulé est obtenu à l'aide de ticks_ms() du module time (qui retourne un entier).
- Lignes 12 à 15 : Déclaration de la fonction detect_cb(), routine d'interruption déclenchée par l'activation du capteur. Cette fonction assigne une valeur temporelle à la variable globale last_detect. Bien que l'usage d'une instruction print() dans une routine d'interruption ne soit pas recommandé, celle-ci permet néanmoins de constater l'activation du processus d'interruption.
- Ligne 17 : Association de la routine d'interruption detect_cb() avec l'entrée GPIO 22. Le paramètre trigger=Pin.IRQ_FALLING indique que l'interruption doit être déclenchée sur le flan descendant du signal.
- Lignes 19 à 26 : Programme principal fonctionnant dans une boucle infinie.
- Ligne 20 : La LED utilisateur est allumée aussi longtemps que la variable last_detect contient une information. Comme cette variable sera réinitialisée à None après le temps d'éclairage réglementaire (2 sec), la LED s'éteindra automatiquement en temps voulu.
- Lignes 21 à 25 : Après avoir vérifié que la variable last_detect contient une information, la fonction ticks_diff() est utilisée pour calculer le temps écoulé depuis last_detect . Si le temps écoulé est supérieur à 2 secondes (2000 ms) alors last_detect est réinitialisé à None (ce qui éteindra la LED). Un message est également affiché à des fins d'apprentissage.
- Ligne 26 : insertion d'une pause pour limiter la charge du microcontrôleur.

La suite du dossier dans Programmez! 278

Figure 8 – *Détecteur de vibration*





Comment réussir une bonne interface dans un monde complexe ?

Une interface est souvent perçue comme une surface visible : des textes, des illustrations, des boutons, des composants, des interactions. Pourtant, sa fonction première n'est pas graphique : elle consiste à mettre en relation un utilisateur avec un service et lui permettre d'agir.



Anthony Marois

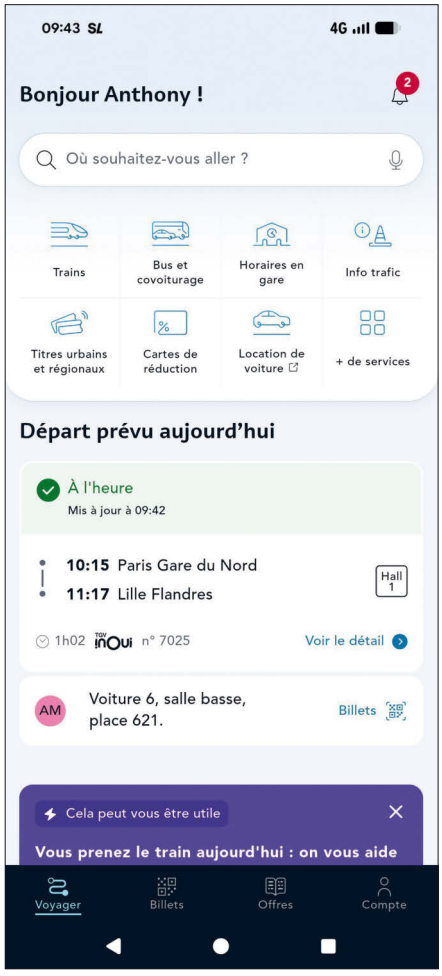
Anthony est Head of Design Research & Vision chez SNCF Connect &Tech. Il est en charge de la compréhension des usages, l'anticipation des transformations et la définition d'expériences de mobilité adaptées aux défis de demain. Après 10 années dans le conseil, il est spécialiste de l'expérience utilisateur, du design stratégique et de l'innovation numérique, pour concevoir des services de plus utiles, inclusifs et durables.

Les différents outils issus des métiers du design, de l'ingénierie et de la conduite de projets nous ont aidés à stabiliser la démarche de conception d'interface. Dans la mobilité, les usages évoluaient lentement et pouvaient être anticipés. Des standards d'expérience s'étaient développés permettant d'industrialiser la création d'interfaces à grande échelle.

Ce n'est plus le cas : concevoir une interface aujourd'hui suppose d'être lucide sur le monde dans lequel elle sera utilisée. Pour comprendre la complexité de ce monde, je le résume en un acronyme : **NAIF**, qui est une adaptation du framework "BANI" emprunté au futuriste et auteur Jamais Cascio. Cet acronyme permet de retenir quatre idées fortes sur le monde dans lequel nous vivons tous :

- **Non linéaire** : les parcours ne suivent plus un chemin unique.
- **Anxiogène** : l'incertitude accompagne toujours les décisions.
- **Insaisissable** : nos usages du numérique évoluent désormais plus vite que les services que nous concevons.
- **Fragile** : les ressources matérielles, énergétiques et environnementales sur lesquelles repose le numérique ne sont pas infinies et mettent nos infrastructures sous tension.

Figure 1



À travers ce prisme, la question n'est plus seulement : *comment construire une bonne interface ?*

Elle devient : *comment rendre le monde plus compréhensible, plus maîtrisable et plus durable ?*

Non linéaire : concevoir pour des parcours fragmentés

Le temps où un seul service pouvait prétendre contenir l'intégralité d'un parcours semble révolu.

C'est particulièrement vrai dans la mobilité : nos déplacements se construisent désormais à travers une multitude de services, d'écrans et de décisions. Un même voyage peut associer train, transport urbain, covoiturage ou location de véhicule. Les utilisateurs naviguent continuellement entre plusieurs besoins, plusieurs temporalités et plusieurs fournisseurs.

Dans ce contexte, le rôle d'une interface n'est plus uniquement d'orchestrer une succession d'écrans. Elle doit aider chacun à se repérer dans un système plus vaste.

C'est l'une des ambitions qui a guidé l'évolution récente de la page d'accueil de l'application SNCF Connect. Dans la zone située en haut de page, de nouveaux raccourcis donnent un accès direct aux différents univers proposés : train, bus, covoiturage, mobilité urbaine, titres de transport, information voyageur, etc.

Ce choix permet à nos utilisateurs d'accéder rapidement à la bonne porte d'entrée :

- Qu'il s'agisse des offres de longues distances, avec les boutons "Trains" et "Bus et covoiturage" bien identifiés qui guident l'utilisateur vers des moteurs de recherche adaptés.
- De la courte distance, pour la recherche d'itinéraires urbains et l'achat de titres associés, à l'échelle d'une ville ou d'une région.
- L'accès rapide à l'information voyageur : les prochains départs en gare comme l'information trafic

Figure 1

Ces raccourcis explicitent concrètement la proposition de valeur de SNCF Connect : faciliter les déplacements du quotidien comme les longs trajets.

Et les résultats témoignent qu'une bonne interface est celle qui donne de la cohérence à un écosystème complet :

L'adoption a été très rapide de ces nouveaux raccourcis, qui complètent parfaitement l'expérience de recherche universelle Nous avons mesuré une augmentation significative des achats liés à des mobilités autres que le train.

Anxiogène : réduire l'incertitude

La technologie n'a jamais été aussi performante. Pourtant, voyager demeure une expérience génératrice d'incertitude. Le nombre de voyageurs continuera d'augmenter dans les prochaines décennies, ce qui augmentera la pression sur nos infrastructures ferroviaires comme routières.

Les voyageurs continueront de se poser de nombreuses questions : Mon train est-il maintenu ? Y a-t-il un retard ? À quel moment dois-je agir ?

L'observation des voyageurs montre que ces questions persistent même lorsque tout fonctionne normalement.

C'est ce constat qui a conduit les équipes Produit à repenser la présentation des informations liées au prochain départ dans SNCF Connect.

Les horaires, la voie, le repère de quai ou le numéro de place étaient déjà disponibles. Les retards étaient signalés lorsqu'ils survenaient. Mais lorsqu'aucune perturbation n'était présente, l'interface restait silencieuse.

Or ce silence laissait subsister un doute. Nous avons donc ajouté un bandeau indiquant explicitement le statut de circu-

lation du train ainsi que l'heure de mise à jour de l'information. **Figure 2**

Derrière cette évolution se cache une conviction plus générale : la fonction d'une interface n'est pas seulement informative. Elle est aussi cognitive et émotionnelle pour accompagner efficacement les utilisateurs à atteindre un but, mais aussi agir sur le contexte émotionnel qui accompagne cette action. Ici :

- du soulagement quand l'utilisateur comprend que son train est prévu à l'heure et qu'il sait où se positionner ;
- de la confiance quand la donnée paraît fiable, claire et actualisée.

Insaisissable : concevoir pour des usages en mutation permanente

Nos modes d'interaction avec le numérique se diversifient à une vitesse inédite.

Nous cliquons encore et beaucoup, bien sûr. Mais nous parlons, nous dictons, nous faisons des captures d'écrans que nous nous envoyons. Et nous déléguons de plus en plus certaines actions à des assistants et des agents conversationnels. Demain, une partie de nos déplacements pourra être préparée par un assistant capable de comparer des offres, de surveiller des ouvertures de vente ou d'effectuer certaines réservations en notre nom.

Les actions d'un parcours se distribueront entre voyageurs, assistants conversationnels et systèmes automatisés.

Un utilisateur pourra déléguer certaines tâches à un agent : comparer des offres, surveiller l'ouverture des ventes ou suivre l'évolution d'un prix. Il conservera néanmoins le dernier mot lorsqu'il s'agira de choisir et de finaliser une transaction.

Finalement, l'interface pourra être exposée de trois manières :

- directement aux utilisateurs,
- par l'intermédiaire des LLM, qui en exploiteront certains composants,
- ou par les agents eux-mêmes qui agiront en notre nom.

La qualité d'une interface ne dépendra donc plus uniquement de sa capacité à être utilisée par un humain. Elle dépendra également de sa capacité à être comprise par les agents qui l'accompagnent.

Fragile : concevoir dans les limites du réel

Cette projection dans une expérience agentique - souvent présentée comme magique, repose sur une des illusions persistantes du numérique : le percevoir comme immatériel.

Chaque interface repose pourtant sur des terminaux, des réseaux, des centres de données, de l'énergie et des ressources matérielles limitées. Une infrastructure bien présente et de plus en plus sous tension. Une des réponses à la fragilité croissante du monde dans lequel nos services évoluent est l'écoconception.

Non pas faire "moins de numérique", mais concevoir des services capables de créer davantage de valeur avec moins de ressources.

Chez SNCF Connect, cette démarche s'appuie notamment sur le RGESN et sur les principes issus de la théorie du Donut de l'économiste Kate Raworth. Ils nous invitent à concevoir dans deux limites : le plafond environnemental et le plancher social.

Pour lister les plus prioritaires :

- Opter pour des choix de conception minimisant les émissions de GES
- Rendre le service SNCF Connect compatible avec un maximum d'anciens terminaux, pour éviter la génération de déchets et ainsi réduire l'empreinte matière.
- Garantir l'accessibilité et l'inclusivité, en visant des niveaux d'accessibilité numériques élevés.
- Réaliser des interfaces claires et faciles à comprendre.
- Réduire les impacts directs et indirects du service, en particulier en limitant la volumétrie de données échangées et la sollicitation des centres de données.



Figure 2

Concevoir pour le réel

Dans un monde NAIF, une bonne interface ne consiste plus simplement à afficher la bonne information au bon endroit. Elle aide à se repérer dans des parcours fragmentés, réduit l'incertitude qui accompagne les décisions, reste compréhensible malgré l'évolution rapide des usages et respecte les limites physiques du monde dans lequel elle s'inscrit.

Au fond, une bonne interface n'est pas une couche graphique. C'est un outil de médiation entre la complexité du réel et notre capacité à agir.

PROGRAMMEZ! N°278
Disponible à partir du
20 novembre 2026

PROCHAINS NUMÉROS

HORS SÉRIE N°24
Disponible à partir du
25 septembre 2026

Votre API mérite une interface graphique et c'est gratuit !

Sans vouloir enfoncer une porte ouverte : une API doit être documentée, testable et testée ! L'aspect documentation est porté par la spécification OpenAPI et son implémentation facilitée par le Framework .Net. La génération dynamique d'une interface graphique repose, elle, sur des librairies comme Swagger ou Scalar. Enfin, pour les tests unitaires automatisés, Copilot est là pour vous accompagner.



Fred BERTON

Je pratique le développement logiciel depuis plus de 35 ans et depuis 15 ans chez Capgemini avec une approche en phase avec la mouvance des artisans du logiciel "Software Craftsmanship". Ma motivation : contribuer au développement professionnel de logiciels par la pratique et en aidant les autres à acquérir le savoir-faire.

Capgemini

« Capgemini recrute des Tech Leads, rejoignez-nous ! ».

OpenAPI la documentation des Web services

La spécification OpenAPI est un standard indépendant du langage de programmation pour documenter les API HTTP. Lors du développement d'une API, il existe plusieurs approches dites : *Code-First* ou *Design-First*.

L'approche *Design-First* commence par spécifier une API en rédigeant le document OpenAPI, puis, une fois celle-ci validée, le code du serveur d'API est généré à partir de cette formalisation (avec des outils comme *NSwagStudio*).

À l'inverse, avec *Code-First*, on part du code C# pour implémenter les services et les documenter, puis on utilisera le framework pour générer le document OpenAPI à partir du code.

Je ne rentrerai pas dans le débat sur le choix de l'approche (*Code-First* ou *Design-First*), mais dans Visual Studio, le choix qui est fait est celui du *Code-First* (comme souvent).

Le paquet nuget **Microsoft.AspNetCore.OpenApi**, apporte les services qui permettent de :

- Spécifier les points de terminaison ;
- Définir les données échangées ;
- Générer le document OpenAPI disponible sous forme de fichier au format JSON ou YAML.

Lorsque l'on génère une solution dans Visual Studio avec le template de Web service .Net, le fichier *program.cs* contient :

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();
// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
builder.Services.AddOpenApi();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    // map le point de terminaison pour la documentation de l'API
    // avec le format https://localhost:{port}/openapi/v1.json
    // ex: https://localhost:5000/openapi/v1.json
    app.MapOpenApi();
}

app.UseAuthorization();
app.MapControllers();
app.Run();
```

Ici, les instructions importantes sont : *AddOpenApi* qui ajoute les services nécessaires à la génération de la documentation *OpenAPI* dans le conteneur d'injection de dépendances et *MapOpenApi* qui expose les points de terminaison permettant d'afficher cette documentation.

Par défaut la commande *MapOpenApi*, utilisée sans paramètre, crée un point de terminaison */openapi/v1.json* qui renvoie le document. Vous pouvez consulter ce fichier en complétant l'URL du site avec cette route (par exemple, en dév, ça donne : http://localhost:xxxx/openapi/v1.json).

Il est possible de personnaliser le nom du fichier, la route du point de terminaison, ainsi que le format du document afin de générer du YAML à la place du JSON (ou proposer les deux formats).

```
app.MapOpenApi("/documentation/MonService-OpenApi-V1.json");
app.MapOpenApi("/documentation/MonService-OpenApi-V1.yaml");
```

À partir de ces fichiers, vous avez accès à la documentation du Web service.

Générer le document OpenAPI au moment de la compilation

Pour récupérer le fichier *OpenAPI*, il est nécessaire de déployer le Web service et de solliciter l'URL de celui-ci. Dans une logique de chaîne CI/CD (Continuous Integration / Continuous Delivery) complète, il est intéressant de pouvoir générer automatiquement le fichier *OpenAPI* au moment de la compilation (CI), puis de le déployer sur un service proxy chargé d'exposer de votre API (CD).

Cette fonctionnalité est apportée par l'assembly *Microsoft.Extensions.ApiDescription.Server*, inclus dans le paquet nuget de même nom. Cette assembly est invoquée lors de la compilation de votre code pour générer le document directement sur le serveur de build.

Le fichier sera produit par défaut dans le dossier 'obj' comme vous pourrez le voir dans la fenêtre output de Visual Studio lors de la compilation.

Lorsque vous ajoutez ce package à votre projet, le fichier projet est automatiquement complété par un *PropertyGroup*. Il est possible de modifier l'emplacement de génération du document en renseignant la propriété *OpenApiDocuments Directory* pour indiquer le répertoire cible. La propriété *OpenApiGenerateDocuments* permet, quant à elle, d'activer ou non la génération du fichier.

Votre fichier *.csproj* présentera alors les éléments suivants :

```
<PropertyGroup>
      <OpenApiGenerateDocuments>true</OpenApiGenerateDocuments>
      <OpenApiDocumentsDirectory>$(MSBuildProjectDirectory)</Open
ApiDocumentsDirectory>
  </PropertyGroup>
```

Je vous épargne le contenu du fichier de documentation, qui reste relativement complexe à lire (même pour un développeur). À ce stade vous constaterez que le fichier généré n'est pas très prolixe en termes de documentation. Voyons maintenant comment améliorer cela…

Compléter le code pour le Documenter

Pour compléter la documentation de vos points de terminaison, il est nécessaire de les décorer avec les attributs suivants :

- *EndpointSummary*, pour fournir un résumé de l'objectif du point de terminaison (il doit donc être clair et concis) ;
- *EndpointDescription*, pour détailler le service offert ;
- *Description* à ajouter devant chaque paramètre attendu par le point de terminaison.

```
[HttpGet(Name = "GetWeatherForecast")]
[EndpointSummary("ici mon summary")]
[EndpointDescription("ici ma description")]
public IEnumerable<WeatherForecast> Get(
  [Description("iStart c'est le début")] int iStart = 1,
  [Description("iEnd c'est la fin début")] int iEnd = 5)
{
  return Enumerable.Range(1, 5).Select(index => new WeatherForecast
  {
    Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
    TemperatureC = Random.Shared.Next(-20, 55),
    Summary = Summaries[Random.Shared.Next(Summaries.Length)]
  })
  .ToArray();
}
```

Autre subtilité : si votre API utilise des classes pour échanger des données complexes, celles-ci doivent également être documentées à l'aide des attributs : *summary*, *example*, *see cref* classique en C#.

```
/// <summary>
  /// Représente une prévision météorologique pour un jour donné.
  /// </summary>
  public class WeatherForecast
  {
    /// <summary>
    /// Date de la prévision météorologique.
    /// </summary>
    /// <example>2025-06-15</example>
    public DateOnly Date { get; set; }

    /// le reste du code
  }
```

Vérifier également que votre fichier *csproj* contient bien le tag *GenerateDocumentationFile* comme illustré dans l'extrait suivant.

```
<PropertyGroup>
  <TargetFramework>net10.0</TargetFramework>
  <Nullable>enable</Nullable>
  <ImplicitUsings>enable</ImplicitUsings>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>
```

Cacher un point de terminaison

Il est souvent utile de disposer de points de terminaison qui ne sont pas visibles par les utilisateurs d'un Web service. L'attribut *ApiExplorerSettings* permet de cacher un point de terminaison. En conséquence, il reste invocable, mais ne sera pas documenté dans le fichier OpenAPI.
Cet attribut peut être appliqué à une méthode ou à une classe de contrôleur (et donc, à l'ensemble du service).

```
[ApiExplorerSettings(IgnoreApi = true)]
[Route("/JeSuisPasVisible")]
[HttpGet()]
public string JeSuisPasVisible()
{
  return "Ah ! Vous me connaissez…";
}
```

Attention : le fait qu'un point de terminaison ne soit pas présent dans la documentation OpenAPI ne signifie pas qu'il n'est pas accessible. Toute personne ayant connaissance de son existence pourra toujours l'invoquer.

Une interface graphique pour tester le Web service.

Depuis .Net 9.0, l'interface *Swagger* n'est plus ajoutée automatiquement par le template de projet Web service de Visual Studio, mais il reste tout à fait possible de l'intégrer manuellement. Il est même possible d'utiliser plusieurs interfaces de test, parmi lesquelles les plus connues sont : *SwaggerUI* et *Scalar*.
De manière générale, il est considéré comme une bonne pratique de sécurité de ne pas exposer ces interfaces graphiques en production. Il convient donc de veiller à ne les activer que dans un environnement de développement, ce qui peut être vérifié via *app.Environment.IsDevelopment*.

Génération de l'interface graphique avec Swagger

Pour activer les services de *SwaggerUI*, il est nécessaire d'ajouter le package *Swashbuckle. AspNetCore.SwaggerUI* à votre projet. Il convient ensuite de compléter le code avec un appel à la méthode d'extension *UseSwaggerUI*, en indiquant en option le point de terminaison qui fournit le document OpenAPI utilisé par *SwaggerUI* pour gérer l'interface graphique de test.

```
app.UseSwaggerUI(options =>
  {
    options.SwaggerEndpoint("/documentation/MonService-OpenApi-V1
.json", "v1");
  });
```

Si vous lancez l'exécution de votre Web service, Vous risquez d'être déçu : rien de particulier ne se produit. Pour afficher la

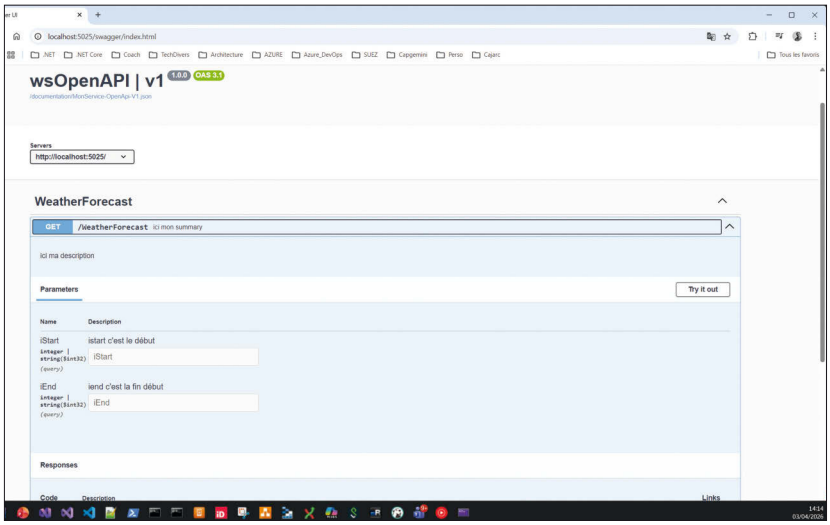


Figure 1 : *https://localhost:<port>/swagger*

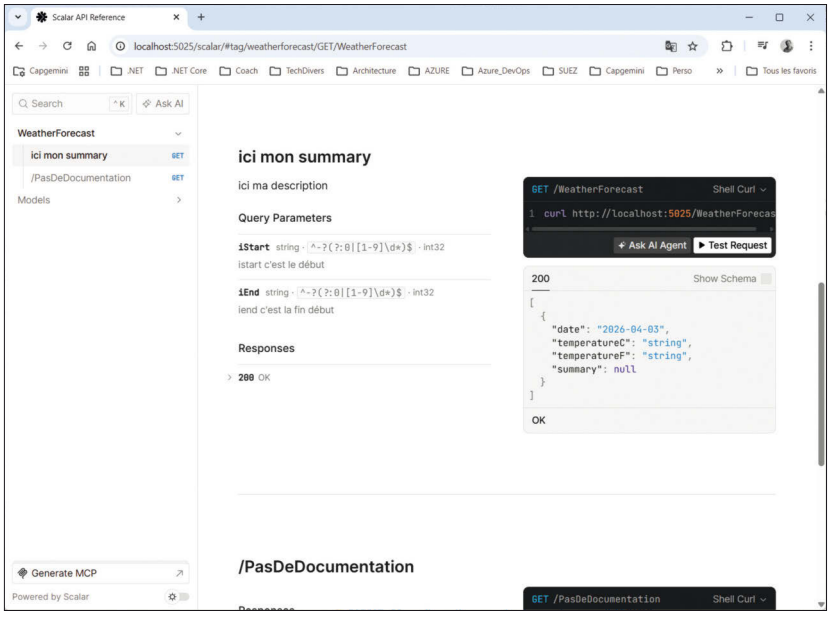


Figure 2 : *https://localhost:<port>/scalar*

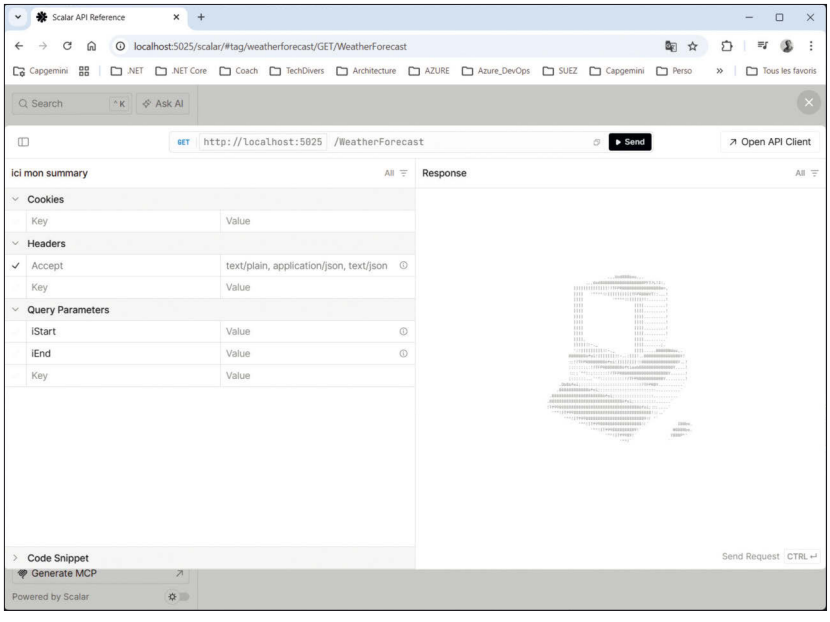


Figure 3 : *Scalar est l'interface de test d'un point de terminaison.*

fameuse interface graphique de SwaggerUI **(figure 1)**, il est nécessaire d'accéder à l'URL suivante : https://localhost:<port>/swagger

Il est possible d'ouvrir automatiquement votre navigateur sur la bonne URL en modifiant le fichier *launchSettings.json*, situé dans le dossier *Properties* de votre projet.
Il faut pour cela définir l'attribut *launchBrowser* à *true* pour lancer, au démarrage du Web service, votre navigateur sur l'url URL indiquée par la propriété *applicationUrl* et complétée par l'URI *Swagger* définie dans la propriété *launchUrl*.

```
{
 "$schema": "https://json.schemastore.org/launchsettings.json",
 "profiles": {
  "http": {
   "commandName": "Project",
   "dotnetRunMessages": true,
   "launchBrowser": true,
   "launchUrl": "swagger",
   "applicationUrl": "http://localhost:5025",
   "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
   }
  }
 }
}
```

Génération de l'interface graphique avec Scalar

L'activation de l'interface graphique de *Scalar* repose sur le même principe. Il est nécessaire d'ajouter le package *Scalar.AspNetCore* au projet, puis d'appeler la méthode *MapScalarApiReference*.
Le point de terminaison *Scalar* redirige directement vers l'interface graphique **(figure 2)**. Comme nous avons modifié le point de terminaison ainsi que le nom du fichier OpenAPI, il est nécessaire de le préciser à *Scalar* via le paramètre de configuration *OpenApiRoutePattern*.

```
app.MapScalarApiReference(options =>
 {
  options.OpenApiRoutePattern = "/documentation/MonService-OpenApi-V1.json";
 });
```

Pour tester le point de terminaison, il suffit de cliquer sur le bouton *Test Request* qui affichera une interface similaire à celle de PostMan. Vous pourrez alors renseigner les paramètres de votre requête **(figure 3)**.

Dans cette page, un petit élément discret, presque magique, se cache en bas à gauche : le : *Code Snippet*. Il permet de générer automatiquement du code dans différents langages et frameworks pour appeler le Web service **(figure 4).**

SwaggerUI ou Scalar comment choisir

Dans un premier temps, vous pouvez choisir de ne pas trancher, car il est possible d'activer les deux interfaces gra-

phiques simultanément, chacune accessible via des URI distinctes.
Personnellement, j'active *SwaggerUI* sur les environnements de test fonctionnel, car il est relativement simple d'emploi et bien apprécié par les responsables fonctionnels pour utiliser les webservices.
En revanche, pour les environnements de développement j'active *Scalar,* car il propose des fonctionnalités de génération de code utiles pour les développeurs clients de votre API.

Tester avec Visual Studio

Les interfaces *SwaggerUI* et *Scalar* sont idéales pour réaliser des tests fonctionnels ponctuels. En revanche, il devient plus complexe de rejouer régulièrement les mêmes scénarios de test. Comme je le dis souvent aux développeurs avec lesquels je travaille : « *N'oublie pas que les problèmes sur une API apparaissent souvent 6 mois après la mise en production. Alors, automatise les tests que tu réalises avec un script…* ».
Visual Studio propose un service minimal permettant de scripter l'utilisation de Web service via des fichiers avec l'extension .HTTP. Il suffit de créer un fichier avec cette extension, puis de l'ouvrir pour accéder à une interface graphique de test dédiée **(figure 5)**.
Le script de test consiste à invoquer les API via des méthodes HTTP (GET, PUT, POST, DELETE, etc.) suivies de l'URL. Chaque requête se termine par ###. Il est également possible de définir des variables en les préfixant par un @.
L'exécution de la requête se fait via le lien « Send request » affiché au-dessus de celle-ci. Le résultat de l'appel apparaît dans la zone de droite, avec notamment le temps d'exécution, le volume de données renvoyé et, bien sûr, les données retournées.

On atteint rapidement des limites de cet outil. La principale tient au fait qu'un fichier ...HTTP doit être perçu comme un livre de recettes, et non comme un script automatisé d'invocation de Web services.
L'intérêt de cette approche réside dans la possibilité de conserver une ' « recette »' d'invocation d'un Web service. Si une requête GET est généralement simple à exécuter, des méthodes comme POST ou PATCH nécessitent, par exemple, de fournir un body, ce qui implique davantage de paramètres. Dans l'exemple ci-dessous le BODY de l'invocation est défini entre accolades.

```
POST https://localhost:7220/weatherforecast
Content-Type: application/json
Accept-Language: en-US,en;q=0.5

{
  "date": "2023-05-10",
  "temperatureC": 30,
  "summary": "bla bla bla"
}
###
```

Visual Studio propose un outil pour faciliter la création de requêtes HTTP :, l'*EndPointExplorer*. Cette fenêtre est disponible depuis le menu *View*, puis *Other Windows* → *Endpoints Explorer* **(figure 6)**.

Dans cette fenêtre, un clic droit sur une des URL, affiche un menu proposant l'option Generate Request, qui permet de générer automatiquement le script d'invocation.
Malheureusement, là encore, les limites sont vite atteintes, car : le script généré ne produit pas le corps du body pour un appel de type POST.
Heureusement, GitHub Copilot peut venir à votre secours. Par exemple, après avoir positionné votre curseur dans la section body de la requête POST (afin de lui fournir un peu de contexte), un prompt simple comme « génère le body de la requête POST » permettra de produire automatiquement le contenu attendu.

Générer les tests unitaires avec GitHub Copilot

Comme nous venons de le voir, GitHub Copilot apporte un support de plus en plus important aux développeurs. L'un de ses cas d'usage incontournables est la génération de tests unitaires automatisés.
Pour commencer, il convient de créer un projet de test xUnit dans la solution qui contient votre projet de Web Service. Vous pouvez ensuite supprimer le fichier de test inclus par défaut dans le template.
Afin de générer le test de votre API, il suffit de solliciter

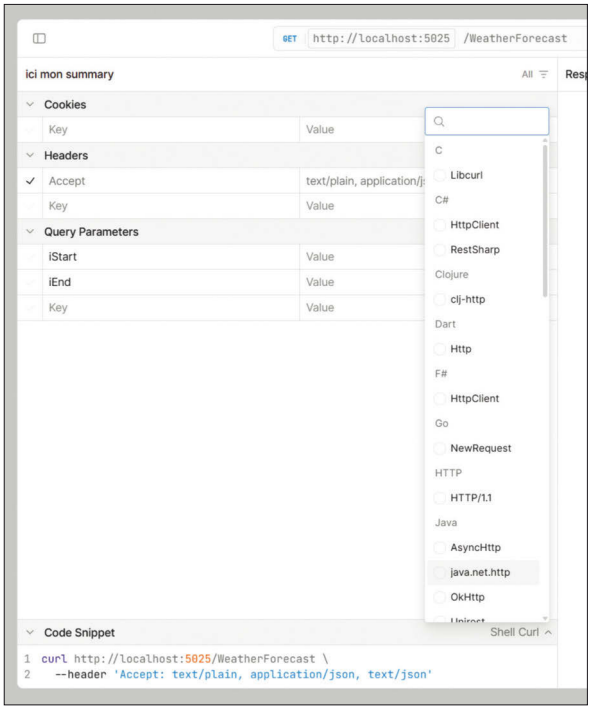


Figure 4 : *la génération de code de Scalar*

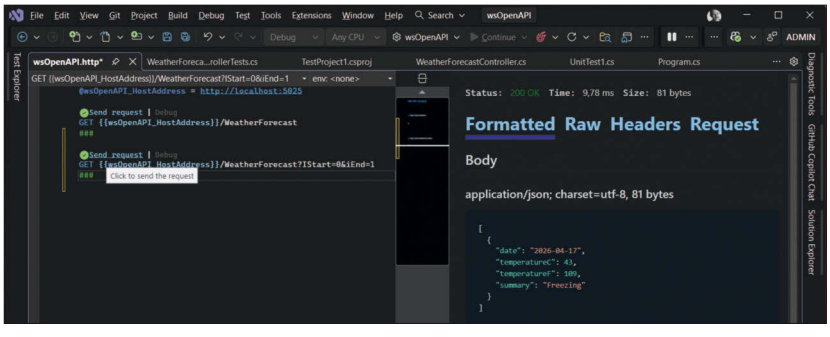


Figure 5 : *le viewer de fichier .HTTP de Visual Studio*

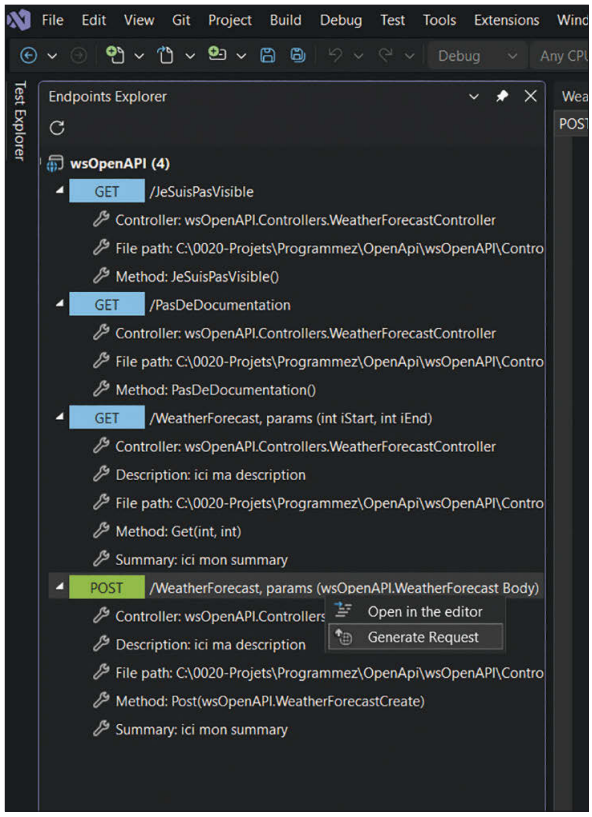


Figure 6 *la vue : end point explorer*

GitHub Copilot. Dans la fenêtre de GitHub Copilot Chat, activez l'agent : Claude Sonnet et passez en mode Agent, afin qu'il génère directement le code dans votre projet de tests.
Un simple prompt permet de générer le squelette du test. Vous pourrez ensuite le compléter, soit manuellement, soit avec l'aide de GitHub Copilot. Lors du processus, GitHub Copilot analyse la solution, identifie le projet de tests basé sur xUnit, parcourt le code de votre Web service, puis génère un code test offrant généralement une bonne couverture de cas en fonction de l'API exposée.
Comme toujours, un prompt gagne à être le plus précis possible afin d'obtenir le résultat attendu. Et, même si cela peut prêter à sourire, j'ai constaté que GitHub Copilot se montre souvent plus réactif le matin que l'après-midi… sans doute un effet de décalage horaire !

Avec un prompt comme celui-ci :
"Créer un test unitaire pour le Web service WeatherForecast Controller.
Le test unitaire doit être produit dans une classe : TU_Weather Forecast"
Voici le code géré pour la requête POST, ce qui est plutôt pas mal.

```
// POST : vérifie que le résultat correspond au corps de la requête
[Fact]
public void Post_ReturnsWeatherForecastMatchingBody()
{
  WeatherForecastCreate body = new()
  {
    Date = new DateOnly(2025, 6, 15),
    TemperatureC = 22,
    Summary = "Warm"
  };

  WeatherForecast result = _controller.Post(body);
  Assert.NotNull(result);
  Assert.Equal(body.Date, result.Date);
  Assert.Equal(body.TemperatureC, result.TemperatureC);
  Assert.Equal(body.Summary, result.Summary);
}
```

On peut regretter que la réponse du Web service ne vérifie pas que la température en Fahrenheit correspond bien à celle exprimée en Celsius lors de la création. Un prompt comme : « *Dans le test Post_ReturnsWeatherForecastMatching Body, vérifier que la température en Fahrenheit est bien égale à celle fournie en Celsius lors de la création* » permettra de corriger ce point.
Au-delà de l'effet « waouh » lié à la génération automatique de tests avec Copilot, l'essentiel reste de construire un ensemble de tests complet, avec une couverture de code suffisante pour garantir un comportement attendu et systématiquement vérifié. Dans une chaîne de CI/CD, ces tests doivent être exécutés à chaque phase d'intégration continue (CI) et constituer un prérequis pour l'étape de déploiement continu (CD).
Comme c'est généralement le cas, votre API passera un jour en production, puis entrera dans un cycle de maintenance applicative pour évoluer ou corriger des anomalies. Un corpus de tests solide vous permettra de limiter efficacement les risques de régression.
Vous noterez également que le *Test Explorer* de Visual Studio permet de visualiser les résultats des tests ainsi que leur durée d'exécution. Ce point est particulièrement utile pour établir une référence de performance et en suivre l'évolution dans le temps.
En conclusion, vous disposez désormais de tous les éléments nécessaires pour produire une API documentée, testée et intégrée dans un processus CI/CD pérenne. Je vous invite à consulter les *Programmez n°263 et n°267*, dans lesquels je partage également de nombreuses autres bonnes pratiques pour le développement de vos API.

Références :

OpenAPI : prise en charge d'OpenAPI dans les applications API ASP.NET Core
https://learn.microsoft.com/fr-fr/aspnet/core/fundamentals/openapi/overview?view=aspnetcore-10.0

Générer des documents OpenAPI
https://learn.microsoft.com/fr-fr/aspnet/core/fundamentals/openapi/aspnetcore-openapi?view=aspnetcore-10.0&tabs=visual-studio%2Cvisual-studio-code

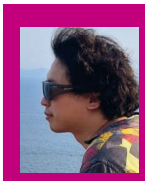
Swagger : https://swagger.io/tools/open-source/ les outils open source des Swagger
https://swagger.io/tools/swagger-ui/ le site de Swagger UI

Scalar : https://swagger.io/ le site Web de Scalar

Visual Studio .HTTP File :
https://learn.microsoft.com/en-us/aspnet/core/test/http-files?view=aspnetcore-10.0

Vérifier formellement des programmes Rust avec Creusot

Creusot est un vérificateur déductif de programmes Rust, vous permettant de prouver que votre code Rust est correct. Ici, on vous explique ce qu'est la vérification déductive, avec un aperçu de sa mise en pratique avec Creusot.



Li-yao Xia

Ingénieur de recherche en vérification formelle au Laboratoire Méthodes Formelles, situé sur le plateau de Saclay.

Comment s'assurer que nos programmes sont corrects ?

Vous savez sans doute qu'il faut tester les programmes qu'on écrit afin d'en éliminer les bugs. S'il existe une grande panoplie de méthodes de test, elles reposent toutes sur l'exécution du programme, avec la limitation fondamentale que cela ne pourra qu'explorer un ensemble fini d'entrées possibles et donc le risque qu'il reste des entrées non testées qui induisent le programme en erreur.

Au-delà des tests, on peut chercher à respecter des règles de programmation plus strictes. On peut même espérer que le langage de programmation impose de telles règles, et que cela permette de se débarrasser d'une catégorie toute entière d'erreurs une fois pour toutes. C'est le cas de Rust, dont le typage statique encode de telles règles de programmation afin de garantir l'absence d'erreurs de mémoire telles que la dangereuse utilisation après libération de la mémoire ("use after free"), et ce sans avoir recours à un ramasse-miettes ("garbage collector"). Mais tout aussi pratique qu'il soit, un système de types fixé ne vise qu'une catégorie spécifique d'erreurs, et reste aveugle face à toutes sortes d'erreurs de logique. Quel programmeur n'a jamais écrit "<" par erreur au lieu de "<=", ou oublié un "+1" quelque part ?

La vérification formelle est une discipline qui vise à éliminer les erreurs de logique grâce à des preuves formelles. Simplement dit, on va prouver que notre programme se comporte correctement. Nous nous intéresserons ici à une forme particulière de vérification formelle : la vérification déductive. On va se donner un langage de spécifications pour exprimer ce qu'est un comportement correct, ensuite un programme équipé de sa spécification pourra être traduit en une formule mathématique qui, si elle s'avère vraie, garantit que le programme satisfait sa spécification.

Les principes de cette méthode furent établis par Tony Hoare notamment en 1969. Ces idées ont depuis été cristallisées aussi bien en des langages de programmation conçus pour la preuve, comme Why3, Dafny, F* ("F star"), qu'en des extensions de langages de programmation plus conventionels, comme SPARK Ada, Frama-C, Liquid Haskell, et Creusot, le sujet de cet article. Pour citer un projet basé sur cette technique, Project Everest propose des implantations vérifiées de protocoles qui composent HTTPS, dont certaines sont aujourd'hui intégrées au navigateur Firefox.

Un petit exemple pour commencer

Voici une fonction qui incrémente son argument :

```
fn incr(x: u64) -> u64 {
  x + 1
}
```

Cette fonction a un défaut : si x est égal à la plus grande valeur représentable en 64 bits, 2^{64} -1, alors x+1 produit un dépassement d'entier, qui peut soit faire échouer le programme ("panic!") soit produire un résultat tronqué. Les dépassements d'entiers peuvent survenir tout aussi bien dans des programmes très simples comme celui-ci, là où le programmeur n'est pas enclin à y penser, que dans des programmes complexes, où le nombre de cas à étudier explose, rendant impossible une vérification faite « à la main ».

Ici, nous allons exclure la possibilité de dépassement d'entier en restreignant les entrées permises à l'aide d'une précondition. Ici, l'argument x ne doit pas dépasser 2^{64} -1, qu'on peut écrire "u64::MAX" en Rust. On ajoute cette précondition à la fonction concernée grâce à l'attribut requires.

```
#[requires(x < u64::MAX)]
fn incr(x: u64) -> u64 {
  x + 1
}
```

On invoque ensuite le compilateur Creusot avec la commande "cargo creusot". Creusot va ainsi traduire la fonction incr en une formule logique qui exprime la correction de cette fonction par rapport à sa spécification. Puis Creusot envoie ces formules à des prouveurs automatiques (prouveurs SMT de leur nom technique, tels que Z3, CVC5, et Alt-Ergo). Ici, la proposition s'avère être facilement prouvable par ces prouveurs. La commande "cargo creusot" se termine avec succès. C'est ainsi que l'on établit que la fonction incr est correcte, c'est-à-dire que, pour toute entrée x qui satisfait la précondition "x < u64::MAX", l'addition x+1 ne dépasse pas "u64::MAX".

Une idée importante en vérification est que quelque chose ne peut être "correct" que par rapport à une spécification donnée. Une précondition seule spécifie les entrées pour lesquelles la fonction n'échouera pas, mais ne dit rien sur ce que la fonction produit. Pour cela, on ajoute une postcondition, indiquée par l'attribut ensures.

```
#[requires(x < u64::MAX)]
#[ensures(result@ == x@ + 1)]
fn incr(x: u64) -> u64 {
  x + 1
}
```

De manière évidente, on dit que le résultat (result) vaut x plus 1. Mais tout est dans le détail : afin d'éviter les complications liées aux dépassements d'entiers au sein de nos spécifications, on préférera effectuer les additions et autres opérations arithmétiques dans l'ensemble des entiers relatifs, dénommé $\mathbb{Z}$ en mathématiques, Int en Creusot. La notation x@ projette un entier machine x (de type u64) en un entier relatif x@ (de type Int).

Vérification d'une recherche par dichotomie

Passons à un exemple plus fourni. Le problème est de trouver un élément dans un tableau trié. C'est la méthode qu'on utilise pour rechercher un mot dans un dictionnaire papier : on tire un mot au milieu du dictionnaire et on décide si le mot recherché est avant ou après ce mot. Dans les deux cas, on aura divisé par deux la taille de l'espace de recherche (le dictionnaire), d'où le nom de « dichotomie ». Puis on recommence, on tire un mot au milieu des mots restants pour savoir si celui qu'on recherche est avant ou après, etc. Dans un dictionnaire de 80 000 mots, on arrive ainsi à trouver un mot donné en 16 ou 17 itérations.

Si l'idée derrière la recherche par dichotomie est simple, son implantation n'en est pas moins subtile. C'est ainsi qu'un bug a été découvert en 2006 dans la fonction de recherche par dichotomie de la bibliothèque standard de Java (voir l'article "Nearly All Binary Searches and Mergesorts are Broken " de Joshua Bloch sur le blog de Google Research). Trouvez l'erreur dans ce calcul de la moyenne de deux int en Java (équivalent à i32 en Rust):

```
(low + high) / 2
```

Réponse : c'est encore un problème de dépassement d'entier. En Java, la somme de deux int suffisamment grands produit un résultat tronqué, qui sera alors négatif. La fonction de recherche par dichotomie va ensuite utiliser une moyenne erronée négative comme indice de tableau, ce qui lèvera une exception. Ce bug ne pouvait avoir d'effet qu'avec des tableaux de taille supérieure à 2^{30}, c'est-à-dire contenant plus d'un milliard d'éléments, cela explique pourquoi il aura fallu plus de 20 ans pour le découvrir dans la bibliothèque standard de Java.

Nous allons prouver la correction d'une fonction de recherche binaire en Rust, en utilisant Creusot. Voici la fonction (pour simplifier les choses, on fixe le type des éléments du tableau à u32) :

```
pub fn binary_search(arr: &[u32], elem: u32) -> Option
<usize> {
  let mut size = arr.len();
  let mut base = 0;

  while size > 0 {
    let half = size / 2;
    let mid = base + half;
    match arr[mid].cmp(&elem) {
      Greater => size = half,
      Equal => return Some(mid),
      Less => {
        base = mid + 1;
        size = size - half - 1
      }
    };
  }
  None
}
```

Expliquons d'abord comment ce code fonctionne. La fonction binary_search cherche l'indice de l'élément elem dans le tableau arr. Durant l'exécution de cette fonction, on maintient un intervalle du tableau arr dans lequel elem peut se trouver; cet intervalle est représenté par sa longueur size et son premier indice base. Au début, l'intervalle couvre l'entièreté du tableau ("size = arr.len()", "base = 0"). À chaque tour de boucle, on divise sa taille par deux, ce qui nous permet d'accéder à l'élément au milieu de l'intervalle pour le comparer à elem. On continue alors soit en réduisant l'intervalle à sa moitié gauche (cas Greater) ou droite (cas Less), soit en renvoyant l'indice de la valeur trouvée (cas Equal).

En passant, on a évité le problème de dépassement d'entier dans le calcul de "(low + high) / 2" en changeant la représentation de l'intervalle : plutôt qu'une paire d'indices "(low, high)" représentant l'intervalle "[low, high[", on utilise une paire d'un décalage et d'une longueur "(base, size)" représentant l'intervalle "[base, base+size[". Cela permet de n'effectuer que des additions avec un résultat borné par "usize::MAX" et donc sans dépassement.

L'objectif de la vérification formelle est de prouver que cette fonction est correcte : en particulier, les opérations susceptibles de lever une exception (qu'on appelle panic en Rust) n'en lèvent pas, et le résultat produit est bien l'indice de l'élément recherché.

Pour cela, une première étape sera de donner une spécification formelle (précondition et postcondition). En deuxième étape, il faudra donner un "invariant" à la boucle while. Pour des algorithmes plus complexes, une troisième étape serait d'ajouter des assertions pour davantage guider les prouveurs ; nous n'en aurons pas besoin ici.

Comme on l'a vu dans la section précédente, la spécification d'une fonction décrit les entrées qu'elle peut recevoir et les sorties qu'elle produit. La précondition dit que la fonction binary_search requiert un tableau trié en entrée, c'est-à-dire que ses éléments sont placés dans l'ordre croissant. Formellement dit : pour tous indices i et j du tableau (entre 0 et "arr.len()") tels que "i < j", le i-ème élément de arr est plus petit que le j-ème élément.

```
#[requires(forall<i, j> 0 <= i && i < j && j < arr@.len()
==> arr@[i]
<= arr@[j])]
```

Pour la postcondition, on sépare les cas où la fonction renvoie None de ceux où elle renvoie "Some(_)". Soit binary_search renvoie None, alors le tableau ne contient pas l'élément elem. Soit binary_search renvoie "Some(n)", alors le n-ème élément du tableau est égal à elem.

```
#[ensures(match result {
  None => !arr@.contains(elem),
  Some(n) => arr@[n@] == elem,
)]
```

Il reste à prouver que la fonction respecte cette spécification. La pièce manquante pour permettre à Creusot d'effectuer cette preuve est-ce qu'on appelle l'invariant de boucle. C'est une propriété qui doit être vraie à chaque tour de boucle, et c'est cela qui permet de faire des raisonnements valides sur un nombre arbitraire de tours de boucle.

Ici, l'invariant de boucle contient deux parties. La première partie établit que l'intervalle "[base, base+size[" se situe à l'intérieur du tableau arr. On écrit un invariant de boucle avec l'attribut invariant sur la boucle concernée :

```
#[invariant(0 <= base@ && base@ + size@ <= arr@.len())]
```

La deuxième partie de l'invariant de boucle décrit les éléments à gauche et à droite de cet intervalle "[base, base+size[" : ils sont soit strictement plus petits soit strictement plus grands qu'elem. On peut en déduire en particulier qu'ils ne sont pas égaux à elem.

```
#[invariant(forall<i> 0 <= i && i < base@ ==> arr@[i] <
elem)]
#[invariant(forall<i> base@ + len@ <= i && i <
arr@.len() ==>
arr@[i] > elem)]
```

L'intérêt de l'invariant de boucle est qu'il permet de décomposer le problème de vérification formelle. On découpe la fonction en morceaux sans boucles : le morceau de code avant la boucle, le corps de la boucle, et le morceau de code après la boucle. L'invariant de boucle donne une spécification à chaque morceau. (1) Quand on arrive à l'entrée de la boucle, l'invariant doit être vrai. (2) Dans le corps de la boucle, en supposant que l'invariant est vrai au début d'une itération quelconque, on doit prouver que l'invariant est rétabli à la fin de l'itération. (3) Puis à la sortie de la boucle, on suppose que l'invariant est vrai. Un outil de vérification comme Creusot est alors capable de générer automatiquement les obligations de preuves pour ces fragments de code sans boucles.

Ayant annoté la fonction binary_search avec sa précondition, sa postcondition, et l'invariant de la boucle while, on peut lancer Creusot avec la commande "cargo creusot", qui va traduire le programme en obligations de preuve puis résoudre celles-ci à l'aide de prouveurs automatiques. Comme on a bien choisi la spéficiation et l'invariant de boucle, toutes les obligations sont prouvées, et on a ainsi formellement prouvé que la fonction respecte sa spécification.

Conclusion

Creusot permet déjà de vérifier un bon nombre d'algorithmes et de structures de données classiques, tels qu'on peut en trouver dans des livres de cours d'algorithmique. En utilisant des techniques plus avancées, on peut vérifier du code bas niveau, impliquant notamment des pointeurs bruts ou de la concurrence. Ces nouvelles techniques nous permettront par exemple de vérifier certains modules de la bibliothèque standard de Rust.

Pour en savoir plus à ce sujet, le site creusot.rs rassemble toutes les informations liées à Creusot : le code source, la documentation, le serveur de discussion pour poser vos questions, les papiers académiques sur Creusot. Vous y trouverez aussi un tutoriel pour ceux qui veulent essayer.

Notre Load Balancer Rust en production : le Bon, la Brute, et l'Async

Il existe une catégorie rare de bugs en production : ceux qui surviennent non pas parce qu'un composant est trop lent, mais parce qu'il est trop rapide. C'est l'histoire que nous voulons raconter. En concevant un load balancer gRPC en Rust pour notre orchestrateur de tâches ArmoniK, nous nous attendions à batailler contre la latence et le débit. Nous n'avions pas anticipé que notre principal adversaire serait notre propre efficacité. Voici le récit de cette épopée : guerres de bibliothèques, contre-intuitions de performance, et les leçons qui nous ont fait passer d'un système performant à un système robuste.

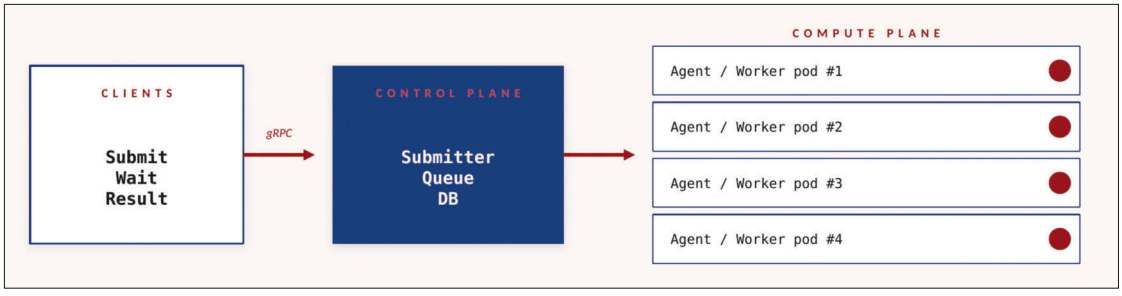


Figure 1 : *ArmoniK vu d'avion : des clients vers un control plane, puis le compute plane.*

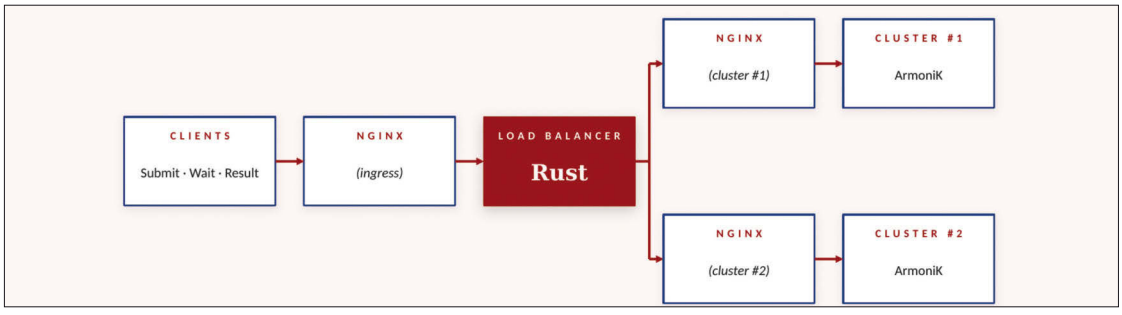


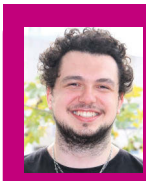
Figure 2 : *Le load balancer, pièce centrale : un endpoint unique, N clusters derrière.*

Le contexte : pourquoi un load balancer maison ?

ArmoniK est notre orchestrateur de tâches : un moteur d'exécution distribué pour charges de calcul massivement parallèles, capable de soumettre, orchestrer et suivre des millions de tâches. Ses *workers* sont polyglottes (Python, C++, .NET, Java…), il se déploie sur Kubernetes en *on-premise* comme en cloud public, et l'ensemble est open source sous licence Apache 2.0. On le retrouve sur des cas d'usage typiquement HPC : pricing et gestion du risque en finance (Monte-Carlo, calcul des grecques, VaR), simulation scientifique (énergie, climat, biologie, ingénierie) ou encore pipelines de ML à grande échelle (inférence batch, fine-tuning distribué). Les clients dialoguent avec lui en gRPC, et à mesure que les déploiements grandissaient, il fallait pouvoir présenter plusieurs clusters derrière un seul point d'entrée. **Figure 1**

La tentation naturelle est de prendre une solution sur étagère. Aucune ne cochait toutes nos cases. Il nous fallait quatre choses à la fois : un **multi-cluster transparent** (un endpoint unique pour N clusters ArmoniK, sans aucune logique côté client) ; un vrai support de **gRPC, et non de HTTP/1.1** (streaming bidirectionnel, multiplexage HTTP/2, connexions longues) ; un **routage applicatif** capable de décider selon la charge, les sessions ou les versions d'API ; et une **observabilité de bout en bout** (traces distribuées, métriques par cluster, audit des routages). Le détail qui disqualifie d'emblée un load balancer de niveau 4 : gRPC multiplexe ses flux sur des connexions TCP longue durée, si bien qu'un équilibrage TCP épingle tout un client sur un seul backend. Il faut travailler au niveau de la requête. De là est née la décision d'écrire notre propre load balancer.

L'architecture cible était simple sur le papier : les clients passent par un premier NGINX d'ingress, qui parle à notre load balancer Rust, lequel route vers le bon cluster, chaque cluster ayant son propre NGINX devant ses instances ArmoniK. Un endpoint unique, N clusters derrière, totalement transparent pour le client. Restait à le construire. Et à le faire tenir debout. **Figure 2**



Jérôme Gurhem

HPC expert & Cloud architect, Aneo

Jérôme est Tech & Research Lead chez ANEO, où il pilote ArmoniK, un orchestrateur open source conçu pour distribuer des calculs massifs sur des infrastructures cloud hybrides et multi-cloud. Architecte, Tech Lead, responsable R&D, référent du support, mentor d'équipe, co-encadrant de thèse… Il a occupé tellement de rôles sur ArmoniK qu'il envisage de se créer un cluster juste pour se gérer lui-même. Titulaire d'un doctorat en informatique, il vient du monde du HPC et s'est progressivement reconverti dans l'art de faire tourner des millions de tâches sans accroc. Quand il ne bataille pas avec Docker, le C# ou Rust, il enseigne les systèmes d'exploitation et le calcul numérique à l'université. Ses chats, eux, orchestrent ses nuits avec une efficacité qu'aucun scheduler ne saurait rivaliser.



Florian Lemaitre

HPC expert & Cloud architect, Aneo

Florian est curieux et touche à tout : administration système, réseau, développement, HPC… Après un temps dans le milieu académique où il a travaillé sur des algorithmes d'algèbre linéaire et traitement d'images CPU, GPU et embarqué, il a rejoint le secteur privé où il design aujourd'hui des infrastructures de calculs sur le cloud, et développe ArmoniK, un orchestrateur de tâches pour le calcul haut débit. Il maîtrise beaucoup plus de langages qu'un humain sain ne devrait, en passant notamment par le C, C++, Rust, Python, C#, Zig…

Le choix technique : pourquoi Rust ?

Le choix du langage s'est imposé presque sans débat. Quatre arguments ont pesé.

D'abord, la **stabilité**. Un load balancer est un processus qui tourne des semaines sans redémarrer. Les garanties du système de types et du *borrow checker* (pas de *data race*, pas de *use-after-free*) sont exactement ce qu'on veut pour dormir tranquille.

Ensuite, un **écosystème asynchrone mature**. Autour de tokio, l'ordonnanceur asynchrone de référence, gravitent hyper (HTTP), tower (abstraction de services et middlewares) et tonic (l'implémentation gRPC). De quoi construire un proxy gRPC sans repartir de zéro.

Puis du **code rapide, pour une empreinte mémoire faible**. Pas de ramasse-miettes (garbage collector), une consommation mémoire qui reste minimale même sous forte charge : pour un composant d'infrastructure traversé par chaque requête, c'est précieux.

Enfin, soyons honnêtes, **parce qu'on en avait envie**. L'équipe connaissait déjà Rust et avait envie d'y aller. C'est un moteur d'adoption qu'on aurait tort de passer sous silence : la motivation des développeurs compte autant que les arguments techniques.

Les batailles de bibliothèques

La théorie était belle. La pratique a commencé par quelques empoignades avec l'écosystème.

Le premier obstacle s'est dressé là où on ne l'attendait pas vraiment : la couche TLS. En Rust, deux camps s'affrontent. D'un côté, la crate openssl : de simples *bindings* sur la bibliothèque C du système. Pratique, mais elle dépend de l'OS et casse toute compilation statique. De l'autre, rustls, l'implémentation TLS présentée comme « 100 % Rust ». Mais la promesse du tout-Rust comporte un astérisque : par défaut, rustls confie la partie cryptographique à aws-lc-rs… elle-même écrite en C. On revient à la case départ.

La parade que nous avons retenue : forcer *toutes* les crates qui font du TLS (hyper, tower, tonic) à s'appuyer sur le même backend, rustls + ring. ring fournit une cryptographie réellement portable, qui nous permet de compiler statiquement un binaire unique sur l'ensemble des architectures que nous ciblons. Le piège est sournois : il suffit qu'une seule dépendance transitive retombe sur le backend par défaut pour casser la compilation statique, ou pire, provoquer un conflit silencieux. Aligner tout l'arbre de dépendances sur rustls + ring n'a rien d'automatique ; c'est un travail d'orfèvre sur le Cargo.toml et les *feature flags*.

Le deuxième front s'est ouvert sur le Protobuf. Avec prost, le codec est lui aussi en pur Rust, élégant sur le papier. Mais la génération de code réclame le compilateur protoc, à installer séparément. CI, Dockerfile, postes de développement : autant d'endroits où il faut le provisionner, et autant d'occasions de voir le build échouer pour une raison qui n'a rien à voir avec notre code.

Troisième escarmouche, plus inattendue : le stockage. Pour conserver un peu d'état local, nous nous étions tournés vers SQLite. Or SQLite est synchrone par nature. Dans un runtime asynchrone, lancer une opération bloquante depuis une tâche tokio, c'est figer le thread qui l'exécute et, avec lui, potentiellement tout l'ordonnanceur. La solution consiste à isoler ces appels dans un pool de threads dédié, via spawn_blocking, et à assumer le coût de ce cloisonnement. Rien de dramatique, mais le rappel qu'en asynchrone, la moindre brique synchrone se paie comptant.

Une fois ces batailles gagnées, le load balancer est devenu fonctionnel. Très vite, même. Trop vite.

Le twist : trop fonctionnel

Mis en charge, le load balancer s'est révélé redoutable. Environ 9 000 requêtes par seconde et par cœur, avec un passage à l'échelle linéaire en fonction du nombre de cœurs. Une latence ajoutée de l'ordre de 500 μs hors réseau, négligeable face au temps de traitement applicatif. Plus de 20 000 connexions simultanées, entrantes comme sortantes, sans dégradation. Et plus de 100 000 *streams* concurrents, multiplexés en HTTP/2 sur une ou plusieurs connexions. Sur le papier, un sans-faute.

C'était précisément le problème. Le load balancer fonctionnait trop bien, et c'est tout ce qui l'entourait qui s'est mis à craquer. Lui ne pliait jamais ; son environnement, si. Concrètement, les utilisateurs récoltaient des erreurs 502. Le coupable apparent était toujours le même, NGINX, mais derrière ce code d'erreur unique se cachaient en réalité cinq scénarios de défaillance bien distincts, qu'il nous a fallu démonter un par un.

L'autopsie : cinq façons de faire tomber NGINX

Plantons le décor. Sur le chemin d'une requête se succèdent un client, un premier NGINX d'ingress (appelons-le N1), notre load balancer Rust, un second NGINX en tête de cluster (N2), et enfin une instance ArmoniK (AK). Chaque montée en charge a révélé un mode de défaillance différent, toujours sanctionné par un 502. Voici les cinq scènes de crime.

Cas n°1 : AK n'a pas le temps. Sous une rafale, N1, le load balancer et N2 transmettent tout sans broncher jusqu'à AK, qui sature et cesse de répondre dans les délais. N2 marque alors AK comme *unhealthy* et fait remonter l'erreur ; le load balancer la propage à son tour ; et de proche en proche, N1 finit par considérer le load balancer lui-même comme *unhealthy*. La parade tient en deux temps : un *rate limit* pour laisser à AK le temps de digérer la charge, et, pour éviter les faux positifs *unhealthy*, des services Kubernetes *headless* couplés à une résolution DNS côté NGINX (version 1.27 ou supérieure). **Figure 3**

Cas n°2 : la 1000 requête. Le client enchaîne les requêtes, N1 et le load balancer les relaient vers N2. À la millième requête d'une même connexion, N2 atteint sa limite keepalive_requests et ferme brutalement la connexion. La dernière requête est rejouée, mais tous les *streams* encore ouverts sur cette connexion sont coupés net. La solution : désactiver le multiplexage des requêtes en sortie du load balancer, et tenir côté load balancer un compteur permettant de fermer proprement la connexion avant d'atteindre la limite de NGINX. **Figure 4**

Cas n°3 : trop de connexions sortantes. Toujours sous rafale, le load balancer ouvre quantité de connexions vers N2 pour paralléliser le traitement. N2 atteint son limit_conn et refuse les nouvelles connexions ; le load balancer propage l'erreur, et N1 le marque *unhealthy*. Correctif simple dans son énoncé : plafonner, côté load balancer, le nombre de connexions sortantes concurrentes. **Figure 4**

Cas n°4 : des connexions SQLite par milliers. Nous avions sorti SQLite du chemin asynchrone (voir plus haut), mais un détail nous avait échappé : le pool de connexions qui l'alimentait n'était borné par rien. Sous rafale, le load balancer traite toutes les requêtes en parallèle, chacune réclamant sa connexion SQLite. Or, dans notre cas, une connexion mobilise un thread. Le nombre de threads explose, le processus du load balancer sature, et N1 le déclare *unhealthy*. La réponse a consisté à confier les accès SQLite à un pool de threads rayon de taille bornée, avec stockage *thread-local* (une connexion par thread, comme c'était déjà le cas). Le point critique est là : rayon impose un nombre fixe de threads et fait patienter le surplus dans une file d'attente interne. Le nombre de threads, donc de connexions SQLite, est désormais plafonné, et les requêtes en excès attendent leur tour au lieu de saturer le processus. **Figure 6**

Cas n°5 : une connexion par requête. Dernier piège, en entrée cette fois : N1 ouvre une nouvelle connexion pour chaque requête. Le load balancer doit absorber ce flot de connexions neuves, et sature dès la phase de *handshake*, d'autant plus vite que le CPU est compté. N1 le marque *unhealthy*. Le remède : configurer l'*upstream* NGINX avec keepalive, pour réutiliser les connexions sortantes au lieu d'en recréer une à chaque appel. **Figure 7**

Cinq incidents, un même fil rouge : à aucun moment notre load balancer ne disait « stop ». Il acceptait, transmettait, ouvrait, parallélisait, sans jamais se demander si l'aval suivait. C'est le concept qui nous manquait : la **contre-pression** (*backpressure*), ce mécanisme par lequel un consommateur saturé signale au producteur de ralentir. Dans un modèle synchrone à base de threads, elle est gratuite : quand tous les threads sont occupés, la requête suivante attend, et la concurrence se trouve bornée mécaniquement. En Rust asynchrone, ce garde-fou disparaît : une tâche tokio coûte si peu qu'on en lance des centaines de milliers sans y penser. Le runtime, lui, ne dira jamais non à votre place. C'est à nous de réintroduire, explicitement, les limites que l'efficacité avait fait sauter.

Les contre-intuitions

Avec le recul, ce sont surtout nos certitudes de départ qui ont mal vieilli. Nous pensions qu'en matière d'asynchronisme, plus on en fait, mieux c'est ; en pratique, sans contre-pression, l'async se contente d'amplifier la surcharge en aval, et la vraie compétence consiste à savoir freiner. Nous imaginions aussi qu'un bon proxy se devait d'être le plus transparent possible, jusqu'à comprendre qu'un proxy parfaitement transparent relaie aussi fidèlement les rafales : il devient un amplificateur plutôt qu'un tampon. Restait notre confiance

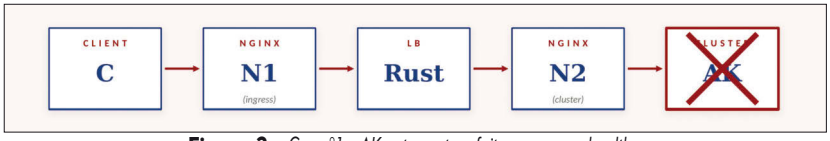


Figure 3 : *Cas n°1 : AK sature et se fait marquer unhealthy.*

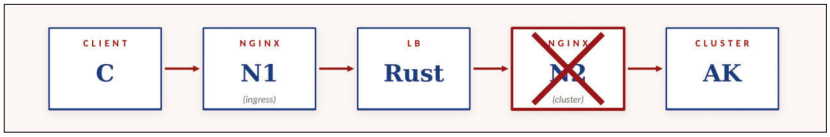


Figure 4 : *Cas n°2 : N2 ferme la connexion à la 1000 requête.*

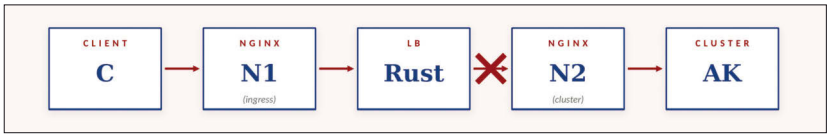


Figure 5 : *Cas n°3 : N2 refuse les connexions sortantes du LB.*

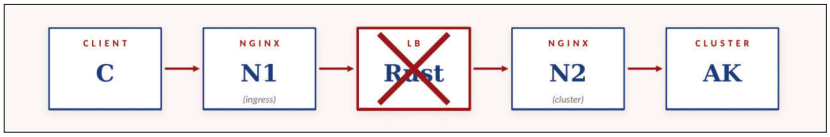


Figure 6 : *Cas n°4 : le LB sature, ses threads SQLite explosent.*

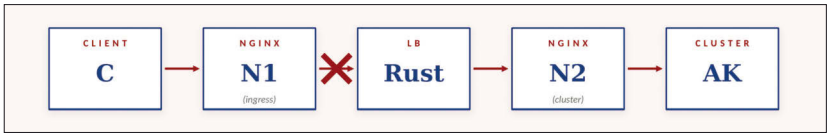


Figure 7 : *Cas n°5 : le LB sature au handshake des nouvelles connexions.*

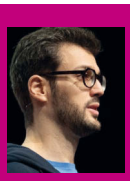
aveugle dans NGINX, réputé encaisser n'importe quelle charge ; en réalité, ses nombreuses limites (keepalive_requests, limit_conn et consorts) sont précisément ce qui l'empêche de saturer en temps normal, et pour ne rien casser, il faut composer avec elles plutôt que passer en force.

Ce qu'on retient

Si l'on ne devait retenir qu'une chose, ce serait que l'asynchronisme sans contre-pression est une bombe à retardement : le runtime ne dira jamais non à votre place, et c'est donc à vous de poser les limites. Cette aventure nous a aussi rappelé que la performance est une propriété du système entier, pas d'un composant isolé ; accélérer une brique ne fait souvent que déplacer le goulot d'étranglement vers la suivante. De là vient la nécessité de connaître chaque couche traversée et ses contraintes propres, sous peine de passer des heures à accuser le mauvais coupable. Et s'il fallait conclure sur Rust : le langage sécurise le code, pas la conception. Le compilateur élimine des classes entières de bugs mémoire, mais il laisse intactes les mauvaises décisions d'architecture. Le Bon, c'est ce load balancer final : rapide *et* discipliné, capable de protéger ce qui l'entoure. La Brute, c'est nous : trop pressés de gagner en performance, nous avons brutalisé tout ce qui se trouvait autour. Et l'Async ? Un allié formidable, à condition de ne jamais oublier qu'une concurrence illimitée n'est pas une fonctionnalité, mais une dette qui se paie tôt ou tard. En résumé : Rust nous a donné un load balancer rapide ; le vrai travail a commencé après.

ArmoniK est open source, sous licence Apache 2.0 : github.com/aneoconsulting/ArmoniK.

J'écris encore du SQL : génération de code Go via sqlc



Sébastien Ferrer

Ingénieur logiciel chez OVHcloud à Nantes, dans l'équipe SIA (Service Integration & Automation).

Exerçant ce métier depuis maintenant 14 ans, il a toujours aimé vulgariser et partager ses connaissances en public.

Il anime actuellement des conférences autour de la cybersécurité, de la gestion d'incidents et divers autres sujets techniques.

Portant un grand intérêt pour l'enseignement, il intervient également régulièrement dans certaines écoles et universités.

Il est également passionné par le théâtre, ayant joué 10 ans dans une troupe en région parisienne.

« J'ai toujours aimé le SQL, mais je pensais qu'il était devenu obsolète à l'ère des ORM. En testant sqlc, j'ai découvert qu'on pouvait allier le plaisir d'écrire des requêtes à la main avec la rigueur d'un code Go généré automatiquement. Voici pourquoi cette technologie, qui redonne ses lettres de noblesse au SQL tout en garantissant une sécurité de typage absolue, est devenue l'un de mes coups de cœur technique de l'année. »

Aujourd'hui, j'aimerais vous parler d'un proof-of-concept que je viens tout récemment d'appliquer dans mon travail. Mais avant ça, j'aimerais raconter comment j'en suis venu à m'intéresser au sujet dont je m'apprête à vous parler.

J'ai toujours aimé le SQL. Ce langage à la fois simple et élégant permettant d'effectuer des requêtes au sein d'une base de données, comme si on lui parlait directement.

Quand j'étais à l'école, c'est peut-être la première fois qu'en informatique j'avais l'impression de parler une autre langue. Une base de données nous fournira toujours les informations que l'on veut, pour peu qu'on sache comment les lui demander. Je suis alors devenu développeur, et je me suis rappelé que si j'ai appris à communiquer avec une base de données c'était surtout pour l'appliquer au sein même de mes programmes. Inclure des requêtes SQL dans un code source, donc.

Comme tout étudiant j'ai connu la joie d'écrire des `SELECT *` et `INSERT INTO` dans des chaînes de caractères concaténées à des valeurs que je passais dans des formulaires, et l'émerveillement de voir s'afficher les données de ma base dans mes pages web.

Comme beaucoup d'étudiants, j'ai mis mes premiers sites web en ligne pour les montrer fièrement à tout le monde.

Puis comme tous ceux qui ont fait la même chose que moi, j'ai découvert la joie de me faire exploser ma base de données à grands coups d'injections SQL.

Comment ça, concaténer des strings SQL, c'est pas bien ?

J'ai alors découvert comment faire de la **préparation de requêtes**.

Nous nous sommes tous heurtés aux mêmes problèmes :

- C'est rébarbatif
- On peut facilement faire des erreurs
- La peur de laisser passer une injection est toujours présente
- Il faut faire le mapping avec nos objets avec attention

Pour ce dernier point, je parlerai d' "objet" dans le cas général et le sens philosophique du terme, bien que je m'apprête à illustrer la problématique avec le **langage Go** qui n'est pas orienté objet.

Je n'étais pas le seul à avoir remarqué cette nécessité de mapper des structures / classes avec des tables de bases de données, d'autres l'ont remarqué bien avant moi. Ce sont ces gens qui ont créé les **ORM** (Object-Relational Mapping), permettant de répondre à ce besoin.

Les avantages des ORM sont immenses :

- On manipule directement des objets
- On abstrait totalement la couche SQL, même plus besoin d'écrire du SQL
- Facilite grandement la sérialisation / déserialisation
- Opérations CRUD simplifiées
- La gestion des relations est simplifiée
- Compatibilité entre plusieurs bases de données (PostgreSQL, MySQL, ...)

Il y a également des inconvénients incontournables :

- Perte de performance
- Contrôle plus limité
- Debugging parfois plus compliqué
- Apprentissage d'un DSL

Bien que ces inconvénients existent, les avantages sont bien trop évidents pour passer à côté. Le principe des ORM est largement adopté dans à peu près tous les langages de programmation.

J'adorais le langage SQL, mais je dois me faire une raison : écrire du SQL à notre époque est obsolète. Plus personne ne fait ça. **C'est du moins ce que je pensais**.

Cela fait des années que je parle avec mes bases de données via des ORM, et je m'y suis accommodé. Je dois cependant vous avouer que je suis nostalgique de l'époque où je parlais la langue de mes bases de données.

Récemment au travail, nous avons été confrontés à certains problèmes liés à ce manque de contrôle qu'entraînent parfois les ORM, notamment lorsque nous devions changer le schéma de la base de données.

C'est alors qu'un collègue est venu nous présenter une technologie que je ne connaissais pas encore : **sqlc**.

sqlc et les ORM

sqlc est un outil qui génère automatiquement du code à partir de requêtes SQL écrites manuellement.

Il a été créé pour combiner la sécurité et la performance du SQL brut avec la facilité d'utilisation d'un ORM, tout en évitant la surcharge de traitement et la perte de contrôle que peuvent introduire ces derniers.

Initialement écrit en Go pour du langage Go, sqlc supporte aujourd'hui plusieurs autres langages (https://docs.sqlc.dev/en/stable/reference/language-support.html).

Ici, nous en parlerons essentiellement en **Go**. Pour bien situer sqlc dans l'écosystème actuel, nous utiliserons à titre comparatif l'ORM **GORM** (bien qu'il existe d'autres alternatives populaires comme Ent).

Prenons un schéma de base de données très simple :

```
+——————————— +              +——————————+
|    book     |              |    author    |
+——————————— +              +——————————+
|ID    (PK)   |      +—>     |ID   (PK)     |
|Title        |      |       |Email         |
|PublicationDate |   |       |Bio           |
|AuthorID  (FK)  |  ——+      |              |
+——————————— +              +——————————+
```

À première vue ça va un peu ressembler à un retour en arrière : on utilise de nouveau des requêtes préparées écrites directement dans notre code source.
Dans le cas des ORM, le SQL est totalement invisible.
Ajouter une ligne dans une base de données prend littéralement une ligne de code :

```
func CreateAuthor(db *gorm.DB, author *models.Author) error {
  return db.Create(author).Error
}
```

Cependant, comme dit plus haut, nous voulons nous affranchir des limitations des ORM. Si manipuler des bases de données avec un ORM est aisé avec des exemples triviaux, qu'en est-il lorsque nous voulons exécuter des requêtes complexes ? Ce n'est pas forcément impossible, mais nous oblige parfois à faire des concessions (découpage des requêtes pour déléguer une partie des calculs au langage de programmation, création de vues, etc.). Finalement, les inconvénients induits par les requêtes préparées manuellement, n'y a-t-il pas un autre moyen de les gérer ?
sqlc prend le problème dans l'autre sens : pas de dépendance dans le code, mais un outil en local qui génère du code compatible librairie standard.
À noter que de la génération de code vers d'autres librairies est également possible, mais ici c'est le fait de ne pas avoir de dépendances autres que la librairie standard qui nous intéressera.
Le résultat :

- On écrit à nouveau du code SQL
- On obtient du code 100% compatible librairie standard prêt à l'emploi

Mais avant d'écrire le moindre SQL, sqlc a besoin de savoir quelques petites choses : quel moteur de base de données on utilise, où trouver nos requêtes et notre schéma, et où générer le code. Tout ça se définit dans un fichier de configuration, **sqlc.yaml** (ou **sqlc.json**, si vous préférez), placé à la racine du projet.

Voici celui que j'ai utilisé :

```
version: "2"
sql:
 - engine: "postgresql"
  queries: "sqlc/database/*.sql"
  schema: "sqlc/database/migrations"
  gen:
   go:
    package: "database"
    out: "sqlc/database"
```

C'est assez simple : j'indique à sqlc que j'utilise PostgreSQL, que mes requêtes SQL se trouvent dans **sqlc/database/*.sql**, que le schéma de la base de données est défini par les scripts de migration dans **sqlc/database/migrations**, et que le code Go généré doit aller dans le package **sqlc/database**.
Une fois en place, une simple commande **sqlc generate** suffit pour générer le code. **Figure 1**
Passons maintenant au SQL.
Il suffit juste d'ajouter un **commentaire** au-dessus des requêtes afin de spécifier le nom de la fonction désirée, suivi d'une **annotation** indiquant le type de retour : :one pour un seul résultat, :many pour une liste, :exec pour une exécution sans retour, :execresult pour récupérer le sql.Result, :execrows pour le nombre de lignes affectées, ou encore :execlastid pour l'ID du dernier enregistrement inséré.

author.sql

```
— name: createAuthor :one
INSERT INTO author (email, bio)
VALUES ($1, $2)
RETURNING id;

— name: getAuthor :one
SELECT *
FROM author
WHERE id = $1;
```

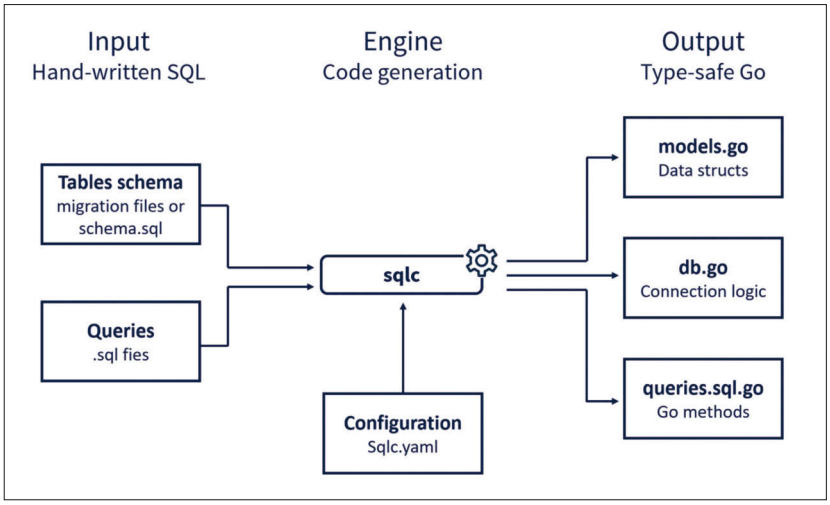
author.sql.go (fichier généré)

```
import (
  "context"
  "database/sql"
)

const createAuthor = `— name: createAuthor :one
INSERT INTO author (email, bio)
VALUES ($1, $2)
RETURNING id
`

type createAuthorParams struct {
  Email string
  Bio  sql.NullString
```

Figure 1



```
}

func (q *Queries) createAuthor(ctx context.Context, arg createAuthorParams)
 (int32, error) {
  row := q.db.QueryRowContext(ctx, createAuthor, arg.Email, arg.Bio)
  var id int32
  err := row.Scan(&id)
  return id, err
}

const getAuthor = `-- name: getAuthor :one
SELECT id, email, bio
FROM author
WHERE id = $1
`

func (q *Queries) getAuthor(ctx context.Context, id int32) (Author, error) {
  row := q.db.QueryRowContext(ctx, getAuthor, id)
  var i Author
  err := row.Scan(&i.ID, &i.Email, &i.Bio)
  return i, err
}
```

C'est une autre façon d'appréhender le problème : fini d'écrire du boilerplate, plus à craindre de potentielles injections dues à des erreurs humaines.
Des avantages très proches de ceux fournis par les ORM en somme, mais avec un contrôle direct sur les requêtes et surtout **sans surcoût à l'exécution**. En effet, contrairement à certains ORM qui s'appuient fortement sur la réflexion au **runtime**, le code généré par sqlc s'exécute tel quel, sans perte de performances.
Vous pouvez trouver un exemple plus complet sur le sqlc Playground : https://play.sqlc.dev/.

Écrire du SQL, c'est cool.
Vous noterez que le `SELECT *` a été transformé en `SELECT id, email, bio` dans le code généré.
En effet, sqlc remplace `SELECT *` par une liste explicite de colonnes pour garantir la stabilité du code généré, optimiser les performances et éviter les erreurs liées aux changements de schéma.
Vous remarquerez ensuite que la comparaison entre l'utilisation des fonctions fournies par un ORM et celle des fonctions générées par sqlc est assez similaire.

GORM

```
func GetAuthor(db *gorm.DB, id int64) (*models.Author, error) {
  var author models.Author
  if err := db.First(&author, id).Error; err != nil {
    return nil, err
  }
  return &author, nil
}
```

Sqlc

```
func (q *Queries) GetAuthor(ctx context.Context, id int32) (*Author, error) {
  author, err := q.getAuthor(ctx, id)
  if err != nil {
    return nil, err
  }
  return &author, nil
}
```

Dans le cas de sqlc, on utilise une structure de type `*Queries`, qui est une encapsulation de DBTX, une interface qui définit un contrat pour exécuter les requêtes SQL. Tous deux sont générés par sqlc.
Libre à nous de les utiliser comme on veut. Pour ma part, j'ai choisi de rendre les fonctions générées internes (d'où le fait qu'elles commencent par une lowercase), afin de les encapsuler par des fonctions exportées du même nom que je définis comme méthode de `*Queries`.
Et c'est tout ! Tout aussi simple à utiliser qu'avant, mais avec bien des avantages.
D'ailleurs, des avantages, il y en a d'autres. J'aimerais justement attirer l'attention sur certains d'entre eux que j'apprécie beaucoup. Premièrement, les modèles sont eux aussi générés par des scripts SQL.

Génération de modèles

Voici les modèles Go générés par mes scripts SQL :

```
type Author struct {
  ID   int32
  Email string
  Bio  sql.NullString
}
type Book struct {
  ID          int32
  Title       string
  PublicationDate time.Time
  AuthorID    int32
}
```

Vous commencez à comprendre le concept de sqlc, ces modèles ont bien évidemment été écrits par des scripts SQL. Voici donc un script SQL qui permettrait la génération de ces modèles :

```
CREATE TABLE author (
  id SERIAL PRIMARY KEY,
  email TEXT NOT NULL UNIQUE,
  bio TEXT
);

CREATE TABLE book (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  publication_date DATE NOT NULL,
  author_id INT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES author(id) ON DELETE CASCADE
);
```

Ça fonctionne, mais en entreprise les tables de nos bases de données sont rarement figées, elles évoluent. C'est pour cela que l'on trouvera en général plutôt des scripts de **migration**. C'est là que ça devient intéressant : si sqlc est capable de

générer nos modèles via du SQL, il peut du coup le faire via des scripts de migration directement.
Voici les scripts que j'ai réellement utilisés pour obtenir le même résultat :

v0001.sql

```
-- +migrate Up
CREATE TABLE author (
  id SERIAL PRIMARY KEY,
  email TEXT NOT NULL UNIQUE,
  bio TEXT
);

-- +migrate Down
DROP TABLE author;
```

v0002.sql

```
-- +migrate Up
CREATE TABLE book (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  publication_date DATE NOT NULL,
  author_id INT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES author(id)
);

-- +migrate Down
DROP TABLE book;
```

v0003.sql

```
-- +migrate Up
ALTER TABLE book DROP CONSTRAINT book_author_id_fkey;
ALTER TABLE book ADD CONSTRAINT book_author_id_fkey FOREIGN KEY
 (author_id) REFERENCES author(id) ON DELETE CASCADE;

-- +migrate Down
ALTER TABLE book DROP CONSTRAINT book_author_id_fkey;
ALTER TABLE book ADD CONSTRAINT book_author_id_fkey FOREIGN KEY
 (author_id) REFERENCES author(id);
```

Chez nous, la gestion des bases de données est effectuée par une équipe dédiée. Ils ont leurs propres repos Git, et nous faisons des PR chez eux pour faire évoluer nos schémas.
Grâce à sqlc, nous pouvons maintenant ajouter directement ces repos via des **Git submodules** afin de faire en sorte que sqlc aille directement chercher les scripts de ces repos afin de générer nos modèles.
Avant, nous étions obligés de reproduire les modèles de données au sein de notre code, avec les erreurs humaines que ça implique.
Cette méthode permet d'obtenir un **lien réel et fiable** entre le code de nos microservices et celui des migrations de base de données. L'équipe en charge des bases de données devient ainsi la réelle **source de vérité**.
À titre d'exemple, j'ai reproduit ce proof-of-concept à la maison : https://github.com/sebferrer/poc-sqlc ; et j'ai stocké mes scripts de migration dans un repo dédié : https://github.com/sebferrer/poc-sqlc-db

Transactions

Un point important à savoir : sqlc ne génère pas de transactions automatiquement. Si on veut exécuter plusieurs requêtes au sein d'une même transaction, il faut le gérer soi-même.
Mais ce n'est pas bien compliqué, car sqlc fournit un mécanisme bien pratique pour ça. Le code généré inclut une méthode `WithTx` sur la structure `*Queries`, qui permet d'associer nos requêtes à une transaction existante :

```
func (q *Queries) WithTx(tx *sql.Tx) *Queries {
  return &Queries{
    db: tx,
  }
}
```

L'idée est simple : on démarre une transaction via la librairie standard, puis on crée une nouvelle instance de `*Queries` liée à cette transaction grâce à `WithTx`. À partir de là, toutes les requêtes exécutées sur cette instance feront partie de la même transaction.
Voici un exemple concret : imaginons qu'on veuille créer un auteur et lui assigner un livre dans la foulée, le tout au sein d'une même transaction.

```
func createAuthorWithBook(ctx context.Context, db *sql.DB, queries
 *Queries, authorParams createAuthorParams, bookTitle string, pubDate
 time.Time) error {
  tx, err := db.Begin()
  if err != nil {
    return err
  }
  defer tx.Rollback()

  qtx := queries.WithTx(tx)

  authorID, err := qtx.createAuthor(ctx, authorParams)
  if err != nil {
    return err
  }

  if err := qtx.createBook(ctx, createBookParams{
    Title:       bookTitle,
    PublicationDate: pubDate,
    AuthorID:      authorID,
  }); err != nil {
    return err
  }

  return tx.Commit()
}
```

Si l'une des opérations échoue, le `defer tx.Rollback()` se charge de tout annuler. Si tout se passe bien, `tx.Commit()` commit la transaction.
Pour finir, si vous voulez en savoir un peu plus, je vous laisse sur quelques dernières notes comprenant certaines subtilités bonnes à connaître que j'ai remarquées sur sqlc.

Quelques subtilités

Structures de paramètres

sqlc génère des structures Go pour encapsuler les paramètres des requêtes SQL, mais uniquement lorsqu'il y a **plus d'un paramètre**. Si la requête SQL n'a qu'un seul paramètre, sqlc n'utilise pas de structure et passe directement la valeur correspondante à la fonction.

On a pu remarquer dans l'exemple plus haut qu'une structure a été générée pour pouvoir passer les paramètres nécessaires, afin de pouvoir la passer à la fonction elle aussi générée :

```
type createAuthorParams struct {
  Email string
  Bio  sql.NullString
}

func (q *Queries) createAuthor(ctx context.Context, arg createAuthor
Params) (int32, error)
```

Ça aurait été la même chose pour la requête SELECT si elle avait eu besoin de plus d'un seul paramètre.

En revanche, si un seul paramètre est nécessaire à la requête, sqlc ne s'embêtera pas à créer une structure pour rien :

```
-- name: deleteAuthor :exec
DELETE FROM author
WHERE id = $1


func (q *Queries) deleteAuthor(ctx context.Context, id int32) error {
  _, err := q.db.ExecContext(ctx, deleteAuthor, id)
  return err
}
```

Structure de retour

On peut également se demander, comment sqlc gère le type de retour pour renvoyer les données sélectionnées ? C'est assez simple.

Si le SELECT comporte toutes les colonnes de la table, comme un SELECT *, il utilisera directement le type généré pour définir le modèle de données.

```
-- name: getAuthor :one
SELECT *
FROM author
WHERE id = $1


func (q *Queries) getAuthor(ctx context.Context, id int32) (Author, error)
```

Sinon, il génère une structure intermédiaire, nous laissant le libre choix de comment la gérer.

```
-- name: getAuthor :one
SELECT id, email
FROM author
WHERE id = $1;


type getAuthorRow struct {
  ID   int32
  Email string
}

func (q *Queries) getAuthor(ctx context.Context, id int32) (getAuthor
Row, error)
```

Embedding avec sqlc.embed

Quand on écrit des requêtes avec des JOINs, la structure résultante générée contient habituellement une liste à plat de toutes les colonnes sélectionnées. Ça peut vite devenir pénible à manipuler.

Par exemple, la requête suivante récupère des livres avec leurs auteurs :

```
-- name: getBookWithAuthor :many
SELECT book.*, author.*
FROM book
JOIN author ON author.id = book.author_id
WHERE book.id = $1;
```

sqlc génèrerait une structure à plat comme celle-ci :

```
type GetBookWithAuthorRow struct {
  ID          int32
  Title       string
  PublicationDate time.Time
  AuthorID    int32
  ID_2        int32
  Email       string
  Bio         sql.NullString
}
```

Avec sqlc.embed, on peut indiquer à sqlc de réutiliser les structures de modèles existantes :

```
-- name: getBookWithAuthor :many
SELECT sqlc.embed(book), sqlc.embed(author)
FROM book
JOIN author ON author.id = book.author_id
WHERE book.id = $1;
```

La structure générée sera alors bien plus propre :

```
type GetBookWithAuthorRow struct {
  Book  Book
  Author Author
}
```

Le code généré reste ainsi bien organisé, et on n'a pas besoin d'extraire manuellement chaque champ d'une ligne à plat.

Paramètres nommés avec sqlc.arg

sqlc essaie de générer des noms pertinents pour les paramètres positionnels, mais parfois il manque de contexte. Par exemple :

```
-- name: UpsertAuthorName :one
UPDATE author
SET name = CASE WHEN $1::bool THEN $2::text ELSE name END
RETURNING *;


type UpsertAuthorNameParams struct {
  Column1  bool
  Column2 string
}
```

Ce n'est pas très lisible. Avec sqlc.arg, on peut nommer explicitement les paramètres :

```
-- name: UpsertAuthorName :one
UPDATE author
SET name = CASE WHEN sqlc.arg(set_name)::bool THEN sqlc.arg(name)
::text ELSE name END
RETURNING *;


type UpsertAuthorNameParams struct {
  SetName bool   `json:"set_name"`
  Name    string `json:"name"`
}
```

Si la syntaxe vous paraît trop verbeuse, vous pouvez aussi utiliser le raccourci @ :

```
-- name: UpsertAuthorName :one
UPDATE author
SET name = CASE WHEN @set_name::bool THEN @name::text ELSE name END
RETURNING *;
```

Note : le raccourci @ n'est pas supporté avec MySQL.

Paramètres nullables avec sqlc.narg

sqlc infère la nullabilité des paramètres, et généralement ça fait ce qu'on veut. Toutefois, si on veut nous-même forcer la nullabilité d'un paramètre qui n'aurait pas été inféré comme tel (parce que la colonne est NOT NULL), sqlc propose également sqlc.narg (nullable arg), qui permet justement de le faire. C'est particulièrement utile pour les **mises à jour partielles**, où l'on veut qu'un champ reste inchangé si aucune nouvelle valeur n'est fournie.
Voici un exemple qui permet de mettre à jour l'email, la bio d'un auteur, ou les deux :

```
-- name: UpdateAuthor :one
UPDATE author
SET email = coalesce(sqlc.narg('email'), email),
  bio = coalesce(sqlc.narg('bio'), bio)
WHERE id = sqlc.arg('id')
RETURNING *;


type UpdateAuthorParams struct {
  Email sql.NullString
  Bio   sql.NullString
  ID    int32
}
```

Ici, sans sqlc.narg, le paramètre email aurait été généré en tant que string (non-nullable) puisque la colonne est NOT NULL. En utilisant sqlc.narg, on le force à être sql.NullString, ce qui nous permet de passer NULL pour signifier « ne change pas ce champ ».

Le revers de la médaille : les requêtes dynamiques

Tout n'est pas rose pour autant. Là où un ORM comme GORM brille par sa capacité à construire des requêtes dynamiques (ajouter un Where() seulement si une variable est définie par exemple), sqlc montre ses limites. Comme tout est figé à la compilation, gérer une barre de recherche avec dix filtres optionnels devient vite un casse-tête.
On pourrait être tenté de ruser avec du SQL un peu grossier du type WHERE (column = $1 OR $1 IS NULL), mais c'est rarement une solution envisageable sur des projets sérieux. C'est verbeux, difficile à maintenir et souvent catastrophique pour les performances dès que la table prend du volume.
Dans ce genre de scénario, il faut savoir rester pragmatique : la meilleure option selon moi est de renoncer à sqlc pour ces cas précis et utiliser un Query Builder (comme Squirrel) ou du SQL brut, uniquement pour ces requêtes complexes.

Autres subtilités en vrac

- Comme vous avez pu le voir, les champs NULL en SQL sont représentés avec sql.NullXxx en Go : ça permet de gérer les valeurs NULL sans erreur d'exécution (type-safe).
- Les requêtes INSERT ... RETURNING permettent d'obtenir l'ID d'un nouvel enregistrement directement : sqlc les traduit en QueryRowContext (au lieu d'un simple ExecContext) et utilise Scan pour récupérer la valeur retournée.

À noter également que sqlc ne propose pas que de la génération de code utilisant la librairie standard database/sql mais propose également pgx/v4 et pgx/v5.

Bonus : faire du Mock

J'aimerais parler d'un autre avantage qui m'a marqué. Le fait de générer son code via sqlc simplifie grandement le mock, et ce pour une bonne raison : c'est nous qui avons écrit le script SQL, donc nous savons exactement à quoi nous attendre.
Cela peut paraître anodin, mais rappelez-vous que si via un ORM nous n'avons aucun contrôle sur le code SQL qui est réellement généré derrière, ça peut rendre les mocks particulièrement fastidieux à écrire.
Si je veux mocker la requête UPDATE générée derrière mon UpdateAuthor(), je dois m'attendre à quoi du coup ? Quelle est la requête exacte que je dois intercepter ?
Prenons l'exemple de GORM :

```
mock.ExpectQuery(regexp.QuoteMeta(`UPDATE "author" SET "email"=$1,
"bio"=$2 WHERE "id" = $3`))
```

Ça fonctionne, c'est très bien. Mais il a fallu que je comprenne un minimum comment fonctionne GORM pour savoir que c'est spécifiquement cette requête qu'on doit attendre si je veux faire un mock de ma fonction.
Est-ce que mon ORM génère les mots-clés en lowercase ou uppercase ? Est-ce qu'il génère systématiquement le nom de la table devant la colonne associée ? Etc.

Répondre à ces questions implique de bien lire la documentation, voire de carrément aller dans le code source de l'ORM. On peut également se servir des messages d'erreurs de la librairie de mock SQL qu'on utilise, mais il faut reconnaître que ça demande du bricolage.
Et ce, pour chacune des requêtes SQL impliquées dans vos mocks.
Une fois passé à sqlc, le mock ressemblera à ça :

```
mock.ExpectQuery(regexp.QuoteMeta(`UPDATE author
                    SET email = $1, bio = $2
                    WHERE id = $3;`))
```

C'est-à-dire exactement la requête que j'ai moi-même écrite à la base (ou presque, cf. l'exemple du SELECT *, mais la requête générée est directement accessible).
Quand on doit mocker toute une batterie de requêtes SQL pour nos tests unitaires, j'ai découvert à quel point c'est plus pratique de n'avoir qu'un simple copié-collé de mes requêtes à faire.
Selon moi, une solution comme sqlc donne une réelle valeur ajoutée à l'utilisation de librairies de mock SQL.

Conclusion

Je pense avoir à peu près fait le tour de ce que je voulais vous présenter.
J'espère avoir pu transmettre la raison pour laquelle je me suis intéressé à cette approche un peu différente de ce que beaucoup d'entre nous utilisent couramment.
En conclusion, les ORM comme GORM offrent une abstraction robuste, facilitent les migrations et proposent une interface CRUD intuitive, mais nécessitent un investissement en apprentissage (bien que léger) et peuvent introduire une surcharge de performance, notamment avec de grandes bases de données et des opérations complexes.
Le SQL pur, de son côté, offre plus de contrôle sur les requêtes et peut être plus efficace pour les grandes bases de données, mais il demande une bonne maîtrise du SQL, implique une implémentation plus longue dans le code Go et devient vite complexe à gérer dès que l'on a besoin de requêtes aux paramètres très dynamiques.

Je dirais donc : si la simplicité et la rapidité de développement sont prioritaires, un ORM comme GORM reste une excellente option. Si l'optimisation, le contrôle précis des requêtes et la sécurité du typage sont essentiels, alors je préconiserais l'approche sqlc, tout en gardant le SQL pur en réserve pour les cas les plus complexes.
L'idée n'est pas de dire qu'une façon de faire est mieux qu'une autre.
J'adore les ORM et je continue d'en utiliser dans beaucoup de mes projets, mais j'avais envie de vous faire découvrir ce principe qui est selon moi particulièrement intéressant.
Enfin, sqlc s'affirme de plus en plus comme un outil pérenne : initialement très centré sur l'écosystème Go, il s'ouvre désormais à d'autres horizons :https://docs.sqlc.dev/en/stable/reference/language-support.html avec le support officiel (actuellement en beta) de langages comme Python, Kotlin, ou TypeScript, et bien d'autres supportés par la communauté via un système de plugins.

Liens / sources

sqlc
https://sqlc.dev/
https://play.sqlc.dev/
https://docs.sqlc.dev/en/stable/index.html
https://github.com/sqlc-dev/sqlc

GORM
https://github.com/go-gorm/gorm

Comparaison / Benchmark
https://blog.jetbrains.com/go/2023/04/27/comparing-db-packages/
https://medium.com/hyperskill/using-gorm-versus-plain-sql-to-interact-with-databases-in-go-39728974edc8
https://www.codementor.io/@patriciotula/building-a-todos-app-in-go-comparing-database-sql-and-gorm-2n921bkfyk

Mon proof-of-concept
https://github.com/sebferrer/poc-sqlc
https://github.com/sebferrer/poc-sqlc-db

FAITES VOTRE VEILLE TECHNOLOGIQUE AVEC programmez!
Le magazine des dév - CTO - Tech Lead

GitHub
Retrouvez tous les codes sources des numéros de Programmez! sur notre GitHub :
https://github.com/francoistonic?tab=repositories

PodDev : le podcast du magazine Programmez!
Retrouvez tous nos podcasts sur Ausha et toutes les principales plateformes
https://podcast.ausha.co/poddev

Chaîne vidéo sur YouTube
Toutes les vidéos du magazine sont sur notre chaîne YouTube :
https://www.youtube.com/@francoistonic-programmez

Jawg Maps, la cartographie à la française dans vos projets web et mobiles



Jones Magloire

CTO de Jawg Maps, passionné d'informatique et de photographie. Présent sur Github au nom de @Joxit avec un projet à plusieurs milliers de stars. A fait ses premiers pas dans l'Open Source en étant au collège. Maîtrise plusieurs langages dont Kotlin, Rust, Java, JavaScript…

En entreprise, quand on a un nouveau projet qui commence et qu'on cherche les technologies que nous souhaitons intégrer, on se rabat souvent sur ce qu'on connaît ou maîtrise déjà. Dans le cas d'un projet ayant besoin d'une carte, on va prendre ce qu'on utilise déjà pour naviguer et qui est souvent dans notre poche : Google Maps. En réalité, une pléthore de solutions existe et sont pour la plupart toutes basées sur une source de données OpenStreetMap.

Qu'est-ce que OpenStreetMap ?

Vous pouvez voir OpenStreetMap comme le Wikipédia de la cartographie, une plateforme collaborative où au lieu d'écrire des articles, vous allez décrire le monde qui vous entoure. Vous pouvez y placer les routes, bâtiments, points d'intérêts etc. C'est un projet qui a débuté en 2004 à Londres et qui contient de la donnée venant du monde entier. Les contributeurs peuvent être des passionnés ou professionnels et ils se réunissent tous les ans lors des State of The Map.

C'est un projet OpenData qui est sous licence ODbL, cette licence vous oblige à afficher la mention © OpenStreetMap lorsque vous utilisez ses données, peu importe la solution utilisée. Vous ne l'avez peut-être pas remarqué mais certaines applications que vous utilisez sont déjà basées sur OpenStreetMap, vous pouvez y voir Amazon, Meta, Apple et Microsoft chez les GAFAM (oui uniquement Google n'en fait pas partie), mais également sur Snapshat, Pokemon Go, divers sites de e-commerce… Ces derniers vont le plus souvent se tourner vers un fournisseur de carte, c'est là que Jawg Maps peut intervenir.

Qu'est-ce que Jawg Maps ?

Jawg Maps est une entreprise française spécialisée dans l'expertise liée à la gestion et l'exploitation de la donnée géospatiale. Depuis fin 2015, elle partage son expertise dans les domaines de la performance géospatiale au sein de grands groupes Français comme la SNCF, Wonderbox, Decathlon, et édite une plateforme SaaS utilisée par des centaines d'entreprises françaises et internationales.

C'est une solution souveraine, respectueuse de la vie privée où tous ses services sont hébergés chez des fournisseurs cloud français et sur des serveurs situés en France. C'est un choix pertinent, pour les projets de toutes tailles, quand ils souhaitent une alternative à Google Maps et autres services cartographiques américain.

Basée sur la donnée OpenStreetMap, Jawg Maps améliore ses services en fusionnant avec diverses autres sources de données pour pouvoir répondre pertinemment aux besoins de ses clients.

Les dessous de la cartographie

À l'image des mathématiques, la cartographie est présente dans la vie de tous les jours sans qu'on s'en rende compte. Nous pensons souvent que son utilité ne s'arrête qu'à l'affichage de cartes pour s'orienter, mais ce n'est en réalité qu'un fragment de ce que l'on peut faire.

Voyez par exemple votre application météo qui vous indique la ville dans laquelle vous êtes, votre téléphone va de lui-même récupérer ses coordonnées GPS (par exemple 48° 48′ 18.2772″ Nord 2° 19′ 29.118″ Est), mais il doit demander à un service que l'on appelle Reverse Geocoding à quel endroit sur terre ces coordonnées correspondent. Votre application peut également utiliser du Forward Geocoding pour qu'il puisse retranscrire l'adresse du lieu de vos prochaines vacances en des coordonnées GPS qu'il utilisera en interne.

Vous pouvez également retrouver des mauvaises conceptions, par exemple votre cinéma ou chaine de magasin qui affiche uniquement de façon textuelle les lieux où ils étaient implantés (ville par ordre alphabétique ou par départements) ? N'était-ce pas fastidieux de sélectionner le magasin le plus proche de chez vous ? C'est dans ces situations qu'avoir une carte peut améliorer l'expérience de l'utilisateur.

Etape 1 : Créer votre première carte

Nous allons à présent nous amuser un peu en créant une carte pour un site web et y afficher de la donnée. Il existe deux librairies principales pour intégrer des cartes à un site web, c'est MapLibre et Leaflet. Je préconise l'utilisation de MapLibre qui fournit la meilleure expérience utilisateur grâce au support des tuiles vectorielles. MapLibre est un projet OpenSource soutenu par AWS, Microsoft, Jawg Maps et d'autres…

Pour commencer, il vous faudra un compte chez un fournisseur de cartes, vous pouvez créer gratuitement votre compte chez Jawg Maps ici https://app.jawg.io/. Après la création de compte, vous pourrez récupérer votre access token, qu'il faudra insérer dans le code pour accéder aux services.

La première étape va récupérer les librairies que nous allons utiliser (MapLibre et Jawg Places JS), et configurer MapLibre.

Si vous passez par NPM, vous pouvez retrouver les paquets maplibre-gl (https://www.npmjs.com/package/maplibre-gl) et @jawg/js-loader pour Jawg Places JS (https://www.npmjs.com/package/@jawg/js-loader)

```
<!doctype html>
<html>
 <head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Jawg Maps @ Programmez! Demo</title>
  <link href="https://unpkg.com/maplibre-gl@5.24.0/dist/maplibre-gl
.css" rel="stylesheet" />
  <script src="https://unpkg.com/maplibre-gl@5.24.0/dist/maplibre-gl
.js"></script>
  <!— Ne pas oublier de remplacer YOUR_ACCESS_TOKEN par votre
propre access token ! —>
  <script src="https://api.jawg.io/libraries/jawg-places@latest/jawg-
places.js?access-token=YOUR_ACCESS_TOKEN"></script>
  <style>
   body,
   html {
    margin: 0;
    padding: 0;
    width: 100%;
    height: 100%;
    font-family: sans-serif;
   }
   #map {
    min-height: 250px;
    height: 100%;
    width: 100%;
   }
  </style>
 </head>
 <body>
  <div id="map" />
  <script>
   // Ne pas oublier de remplacer YOUR_ACCESS_TOKEN par votre propre
access token !
```

Figure 1




```
   const accessToken = 'YOUR_ACCESS_TOKEN';
   const map = new maplibregl.Map({
    // L'ID de l'élément HTML où MapLibre GL JS va insérer la carte.
    container: 'map',
    // Le style vectoriel MapLibre suivant la MapLibre Style Spec.
    style: `https://api.jawg.io/styles/jawg-streets.json?access-token=
${accessToken}`,
    // On centre la carte sur Paris au zoom 10
    zoom: 10,
    center: [2.3488, 48.8534],
    // Désactive l'inclinaison de la carte.
    pitchWithRotate: false,
    // Désactive la rotation de la carte.
    dragRotate: false,
   })
    // Ajoute le +/- pour zoomer sur la carte.
    .addControl(new maplibregl.NavigationControl({ showCompass: false })
, 'top-right')

   maplibregl.setRTLTextPlugin('https://unpkg.com/@mapbox/mapbox
-gl-rtl-text@0.2.3/mapbox-gl-rtl-text.min.js');

   // EMPLACEMENT ÉTAPE 2

   // EMPLACEMENT ÉTAPE 4
  </script>
 </body>
</html>
```

Figure 1

Étape 2 : Ajout de donnée sur la carte

Vous avez à présent un fond de carte, maintenant vous pouvez y ajouter ce qui est en rapport avec votre métier. Si vous n'avez pas encore d'API qui permet de servir vos points, ou si elle peut encore être modifiée, je vous conseille de formater vos données en utilisant la spec GeoJSON (=> https://geojson.org/). Le GeoJSON est un format au-dessus de JSON qui permet d'encoder des données géographiques. Vos géométries peuvent être sous la forme de points, lignes, polygones et leurs variantes en multi-*. Ces structures peuvent se retrouver dans des collections, il y en a plusieurs mais nous allons nous focaliser sur les FeatureCollection. Une FeatureCollection va contenir des Features qui elles vont contenir vos géométries. Cela fait certainement beaucoup de layers, mais vous permettra d'associer des propriétés à vos objets plus facilement. Pour plus de simplicité, nous allons utiliser de la donnée déjà existante et disponible ici : https://media.jawg.io/add-your-data/marketplaces.geojson. Vous pouvez remarquer comment sont encapsulés les objets entre eux.
Étant de l'affichage, il faut également sélectionner l'image que vous utiliserez, nous proposons celle-ci pour l'exemple: https://media.jawg.io/add-your-data/shop.png. Insérez ce code à l'emplacement 2.

```
map.on('load', async () => {
 // Ajouter la source de donnée à mettre sur la carte.
 await map.addSource('marketplaces-source', {
  type: 'geojson',
```

```
  data: 'https://media.jawg.io/add-your-data/marketplaces.geojson',
 });

 // Charge une image externe pour être utilisée en tant qu'icône sur la carte.
 const image = await map.loadImage('https://media.jawg.io/add-your
-data/shop.png');
 await map.addImage('marketplaces-image', image.data);
 // Configure l'affichage de la source de donnée en lui affectant l'image
 map.addLayer({
  id: 'marketplaces-layer',
  type: 'symbol',
  source: 'marketplaces-source',
  layout: {
   'icon-image': 'marketplaces-image',
   'icon-anchor': 'bottom',
   // Autorise plusieurs icônes à se chevaucher
   'icon-allow-overlap': true,
   // Réduit la taille de l'icône sélectionnée
   'icon-size': 0.75,
  },
 });

 // EMPLACEMENT ÉTAPE 3
});
```

Figure 2

Étape 3 : Ajout d'interactivité avec la carte

Dans cette étape, ajoutons un peu d'interactivité, quand vous cliquez sur un point, faîtes apparaître une popup dans votre page. Nous allons faire simple en affichant uniquement le nom du marché, mais vous pouvez injecter du HTML un peu plus complexe pour y mettre des images ou métadonnées. La popup est également un choix arbitraire, vous pouvez remplacer cette interaction par afficher certaines informations dans une barre latérale par exemple.

Le concept est le suivant, quand vous faites une interaction avec la carte, vous pouvez l'écouter, dans notre cas on écoute le click, mouseenter et mouseleave effectués sur notre donnée uniquement, soit le layer marketplaces-layer. Quand c'est fait, au clic nous allons générer le code HTML à insérer dans la popup et la placer à certaines coordonnées sur la carte.

Le code est à ajouter à l'emplacement 3.

```
let featureName;
let popup;

// On va écouter les événements de clic sur la donnée que nous avons
ajouté.
map.on('click', 'marketplaces-layer', (e) => {
 // Récupère le premier élément sélectionné sur la carte
 const feature = e.features[0];
 if (!feature || feature.properties.name === featureName) return;
 featureName = feature.properties.name;
 // Créer la popup qu'on affichera
 popup = new maplibregl.Popup({
  maxWidth: '324px',
  offset: 40,
```

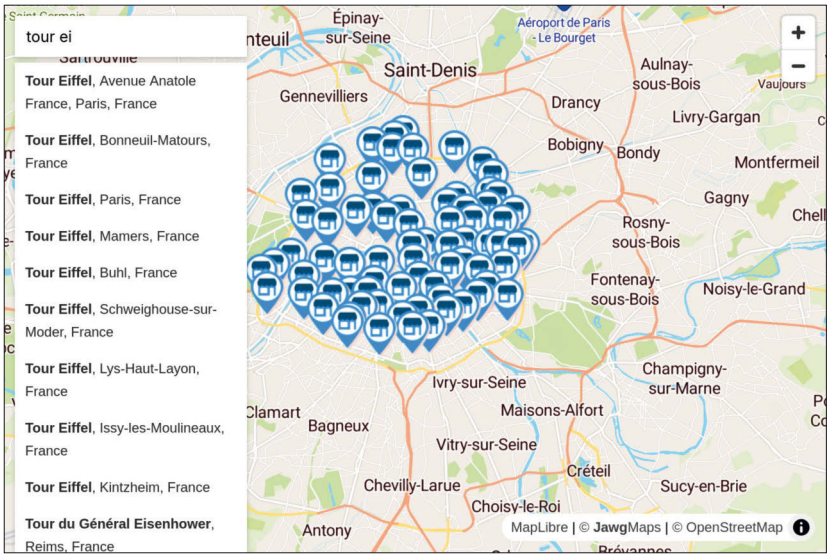


Figure 2

```
  anchor: 'bottom',
 })
  // Les coordonnées de référence de la popup sont identiques à ceux du
point récupéré
  .setLngLat(feature.geometry.coordinates)
  // Raw HTML qui doit être injecté dans la popup.
  .setHTML(feature.properties.name)
  .addTo(map);

 popup.on('close', () => {
  // On met à zéro la propriété si la popup est toujours fermée
  if (!popup.isOpen()) {
   featureName = null;
  }
 });
});

// Amélioration de l'UX en changeant la souris en pointeur au-dessus des
icônes.
map.on('mouseenter', 'marketplaces-layer', (e) => {
 map.getCanvas().style.cursor = 'pointer';
});
map.on('mouseleave', 'marketplaces-layer', () => {
 map.getCanvas().style.cursor = '';
});
```

Figure 3

Étape 4 : Ajout d'une barre de recherche

Maintenant, vous avez à la fois votre carte, vos points et des interactions, vous pouvez continuer en améliorant l'experiences de vos utilisateurs, pour cela ajoutez-y une barre de recherche avec notre librairie Places JS (=> https://www.jawg.io/docs/integration/places-js/maplibre/autocomplete/).

Elle permet à la fois de configurer notre service de geocoding (recherche d'adresse) et son intégration avec MapLibre. Vous pouvez par exemple y ajouter un affichage des frontières administratives quand vous sélectionnez une ville, la manière dont la carte va bouger à la sélection d'un élément recherché… Vous pouvez bien évidemment regarder la documenta-

tion pour voir toutes les fonctionnalités. À titre d'exemple vous pouvez forcer nos services à ne retourner que des réponses dans un ou plusieurs pays… Ici on va limiter les réponses à la France.
Vous pouvez ajouter ce morceau de code à l'emplacement 4.

```
// Ajoute la barre de recherche pour le geocoding.
map.addControl(new JawgPlaces.MapLibre({ searchOnTyping: true, boundary: { countries: 'FRA' } }));
```

Figure 4

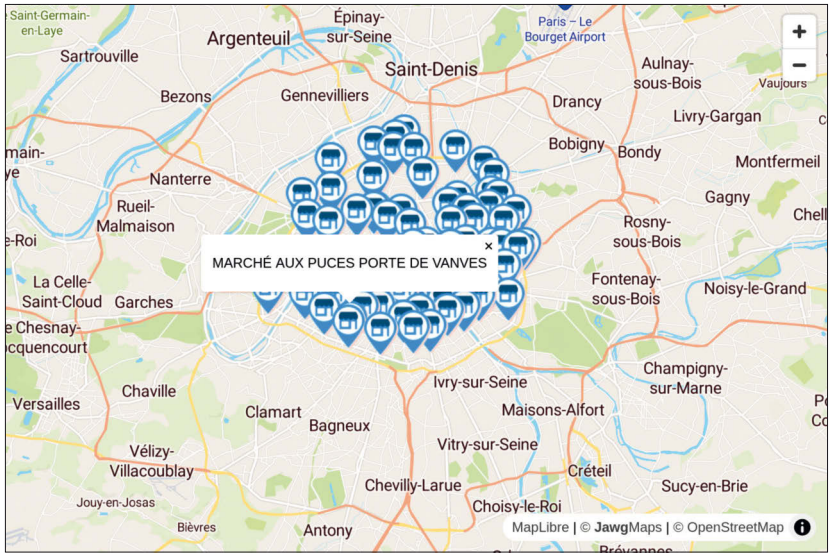


Figure 3



Figure 4

Et pour après ?

Vous avez fini ce petit guide d'intégration pour des cartes en utilisant MapLibre. J'ai pris comme exemple les cartes Jawg Maps, mais il existe d'autres fournisseurs avec des fonctionnalités similaires.
Si vous avez des projets qui intègrent ou peuvent intégrer des cartes, nous pouvons vous donner quelques idées pour des fonctionnalités.
Une librairie open source de calcul géographiques en Javascript à connaître : Turfjs (=> https://turfjs.org/)
Distance entre deux points : vous pouvez afficher la distance entre le point de départ trouvé par le geocoding et le point d'arrivée qui sont vos données. Vous avez deux solutions pour y arriver :

1. Grâce à la distance géodésique, c'est le nombre de km entre deux points sur Terre en prenant en considération sa courbure. Turf sera capable de vous le calculer avec la fonction distance
2. Grâce à un service de routing qui donne la distance en voiture par exemple et même le temps de trajet.

Vous souhaitez stocker vos données géographiques dans une base de données: pensez à PostgreSQL avec l'extension PostGIS (=> https://postgis.net/).
Grâce à l'extension PostGIS, vous aurez accès au type Geometry, à des fonctions géospatiales et des indexes spécifiques pour les géométries pour optimiser vos requêtes. Le type Geometry remplacera vos colonnes lat/lng et permettra :

1. De pouvoir stocker des polygones et lignes en plus des points
2. D'utiliser une recherche géospatiale optimisée par des indexes (vérifier que votre géométrie est en intersection avec une zone précise)
3. Effectuer des opérations sur vos géométries (centroïdes, simplification, union, intersection…)
4. Exporter vos géométries dans divers formats utilisables par votre site web ou services (GeoJSON, MVT, Geobuf…)

Pourquoi choisir MapLibre (et les tuiles vectorielles) par rapport à Leaflet (et les tuiles raster) ?

1. Tuiles vectorielles en moyenne plus légères
2. Possibilité de changer dynamiquement le style de la carte en vectoriel
3. Économiquement avantageux grâce car une tuile vectorielle a la surface de 4 tuiles raster et également grâce au système d'overzooming après les niveaux 14-16 (en fonction des fournisseurs)
4. Possibilité d'interagir directement avec les données du fond de carte

PROGRAMMEZ! N°278
Disponible à partir du
20 novembre 2026

PROCHAINS NUMÉROS

HORS SÉRIE N°24
Disponible à partir du
25 septembre 2026

Les partenaires 2026 de

programmez!

Le magazine des dév - CTO & Tech Lead



Vous voulez soutenir activement Programmez! ?
Devenir partenaires de nos dossiers en ligne et de nos événements ?

Contactez-nous dès maintenant :

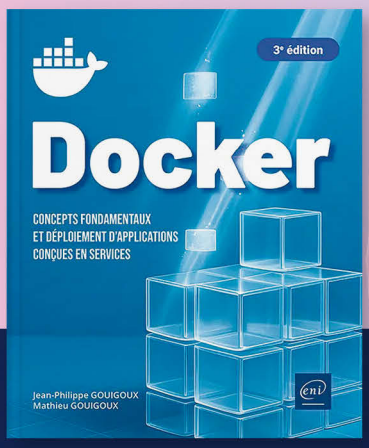
ftonic@programmez.com

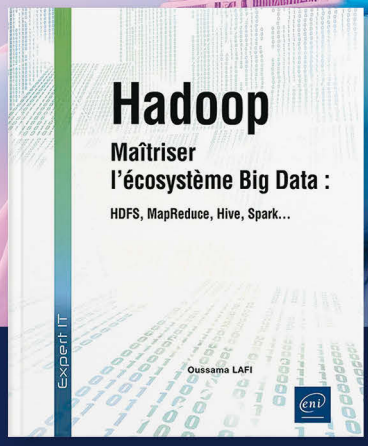
Restez à jour sur tous nos outils informatiques !

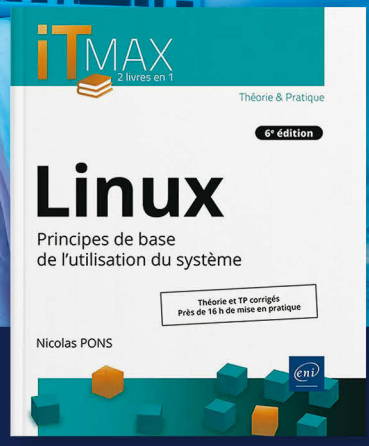
Livres

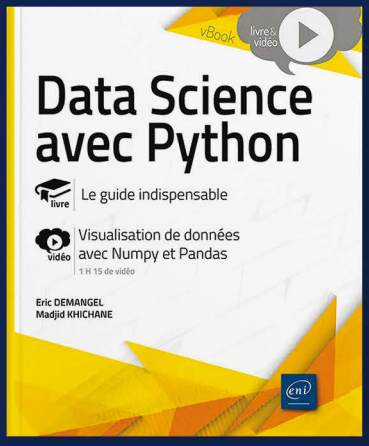
Vidéos

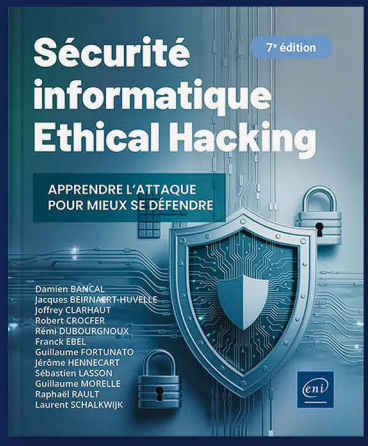
Articles informatiques













Formation en ligne
LINUX

ENI Editions
FORMATIONS À L'INFORMATIQUE EN LIGNE
www.editions-eni.fr