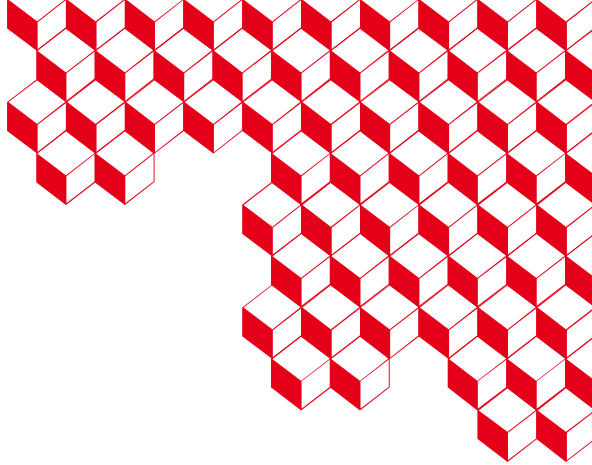
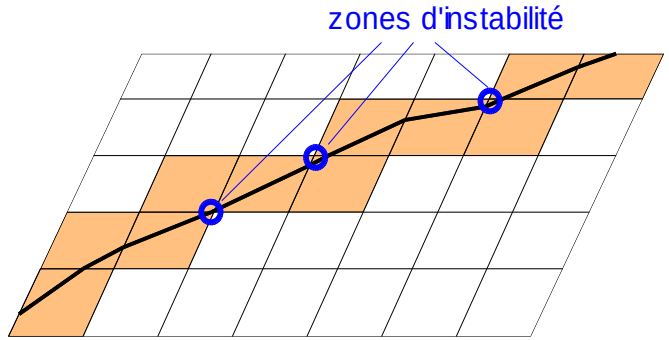
Gestion des branchements instables en analyse synchrone pour les applications industrielles

F. Védrine



Problématique

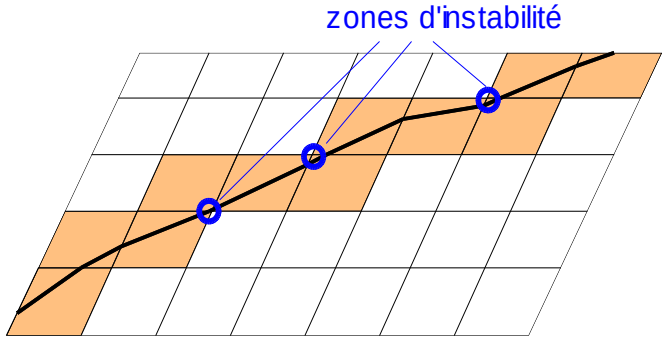
- Code de 7kloc en C++ - algorithmique d'optimisation avec des itérations (> 10^6) sur maillage
  - Maille suivante trouvée en fonction de la maille précédente
  - Interpolation pour trouver le point de sortie de la maille
  - Travail de maille en maille – les seuls calculs sont liés à l'interpolation



- Objectifs
  - Démontrer la stabilité du code
  - Démontrer que l'implémentation est correcte par rapport au papier et déterminer ses limites
  - Démontrer que les paramètres sont modifiables et quantifier l'impact sur le code

- Constat
  - La suite des mailles parcourues est instable
  - Il faut associer un coût à la suite des mailles et montrer que ce coût est continu

Analyse stochastique



- Cadna : précision sur le coût ~10^{-14}
- Verrou : précision sur le coût ~10^{-4}
- FLDLib : temps d'analyse prohibitif si appliqué sans annotations
  - if (x <= …) induit une contrainte qui se propage sur plusieurs milliers d'autres éléments d'un vecteur
  - explosion combinatoire du nombre de tests instables : > 6^{100000} chemins du code à explorer
- La différence provient de l'analyse asynchrone versus analyse synchrone à cause des tests instables
  - Discontinuité de l'ordre de 10^{-4} introduit par le changement de maille
- Comment ajouter des mécanismes passant à l'échelle pour les analyses synchrones ?

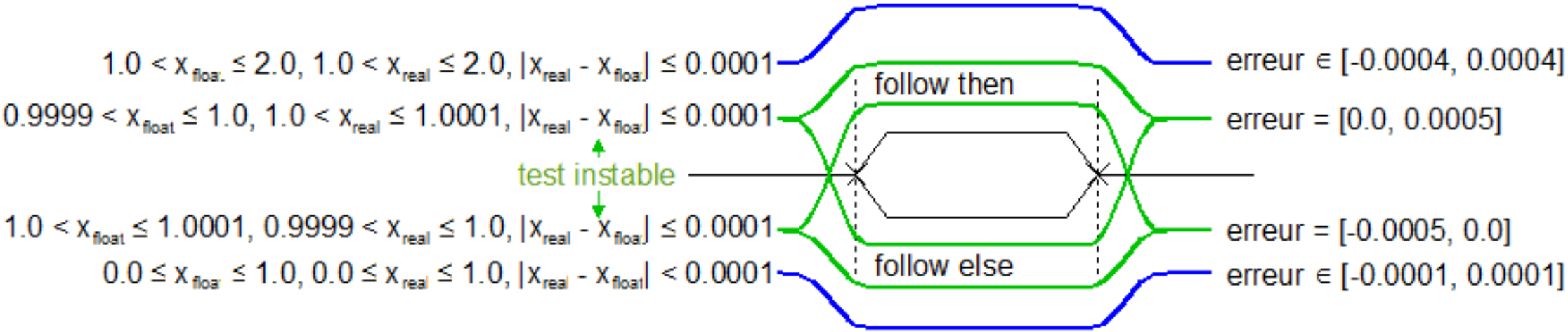
Principes des tests instables en analyse synchrone

- Définir des macros permettant d'exécuter un code plusieurs fois jusqu'à couvrir tous les branchements instables
- Utiliser ces macros de manière locale pour éviter une explosion combinatoire
- Introduire une erreur sur certains entiers et lier cette erreur à un identificateur de test instable
  - Modification statique des types et gestion dynamique – multiplication de méthodes en fonction du contexte d'appel
  - Le lien permet d'effectuer des raisonnements par cas en fonction du test instable rencontré
- Propager le raisonnement par cas à certains domaines flottants
  - pour gérer précisément les interactions entre entiers et flottants
- Et définir de nouveaux containers : la taille de std::vector peut être une constante conditionnée par un test instable !
- Spécificité analyses formelles :
  - raisonner avec $\exists$ fonction injective : indice vecteur $\rightarrow$ chemin dans un arbre binaire équilibré
  - pour remplacer les raisonnements par cas par des raisonnements de plus haut niveau.

Gestion basique des tests instables

```
double x = domaine réel [0.0, 2.0] + erreur [-0.0001, +0.0001];
if (x > 1) { then-branch; y = 4*x-3; }
else { else-branch; y = x; }
```

- 6 exécutions en arithmétique affine pour démontrer la stabilité du test



Multiplication des représentations

- Avant les concepts : utilisation de SFINAE : Substitution Failure is not an Error
- Gérer les interactions de la **`class`** `double_st` avec
  - les **`double`**, les **`float`**, les **`long double`** (constantes), les types entiers (variables et constantes)
  - les `double_st`,
  - les `float_st`, les `long_double_st`,

  - **New**: les flottants conditionnés par un branchement instable, les entiers conditionnés par un branchement instable, les flottants avec option, la « move semantics »

- Avec les concepts
  - Utilisation des « requires » pour cibler l'utilisation des bonnes méthodes
  - Définition de nouveaux « concepts » pour cibler l'utilisation des bonnes méthodes

Utilisation des concepts de C++-20

Nouvelle interface : grosse et technique, mais elle passe sur les études

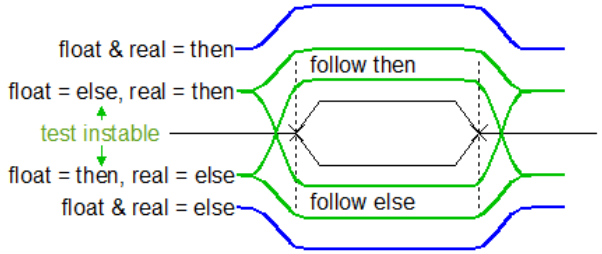
```
template <typename T1, typename T2>
concept same_as_without_references = std::same_as<std::remove_cvref_t<T1>, std::remove_cvref_t<T2>>;
template<typename T>
concept has_field_isBranch = requires {
    T::isBranch;
};
template <typename T>
concept enhanced_floating_point = not has_field_isBranch<T> and requires(T a) {
   { sqrt(a) } -> same_as_without_references<T>;
};

template <typename FloatingType> requires std::floating_point<Type> class floating_point_st;
template <typename FloatingType, typename T>
inline auto operator+(const floating_point_st<FloatingType>& first, T second)
      -> floating_point_st<decltype(FloatingType(0) + typename EFloatingPointType<T>::base_type(0))>
      requires std::floating_point<T>;
```

MACROS ⇒ diagnostic de test instable

définition d'un contexte & sauvegarde locale + **do** {

```
FLOAT_SPLIT_ALL(1,
        ScalarValue::MergeBranches::packer(PathIterator(path, 0), PathIterator(path, -1))
           >> cost >> ScalarValue::end(),
        ScalarValue::MergeBranches::packer(PathIterator(path, 0), PathIterator(path, -1))
           << ScalarValue::end())
```



…

```
FLOAT_MERGE_ALL(1, cost <<
        double::MergeBranches::packer(PathIterator(path, 0), PathIterator(path, -1))
           << double::end(),
        double::MergeBranches::packer(PathIterator(path, 0), PathIterator(path, -1))
           >> double::end())
```

Fusion des résultats & reconstitution + } **while** (!branche_a_explorer.empty())

Utilisation locale des macros

- Evident sur `y = (a < b) ? a : b;`

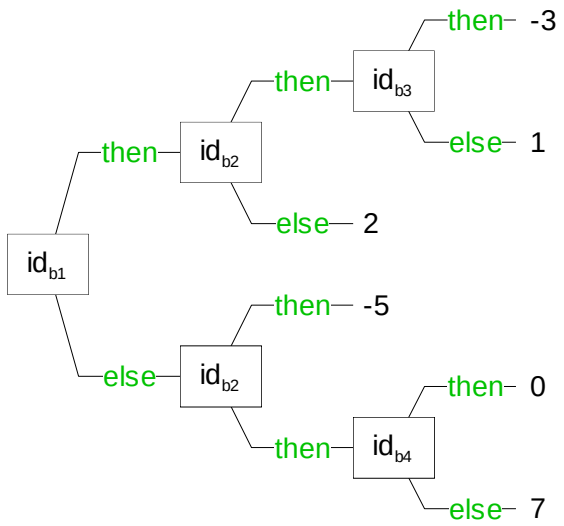
- `double compute_determinant(const Matrix& m, Matrix& lower, Matrix& upper)`
  - Choix de pivot = instable, se retrouve dans lower et upport
  - Le résultat est quant à lui continu

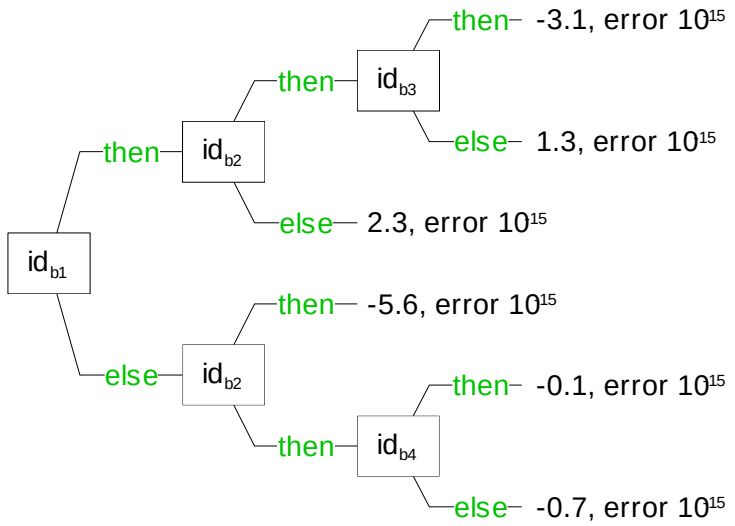
- ne pas synchroniser : précis, mais enchaînement des tests instables ⇒ explosion
- synchroniser : précis pour le résultat, lower et upper avec une très grosse erreur impossible à réduire
  - La multiplication des coefficients de la diagonale de upper ⇒ grosse erreur par rapport au résultat

- **Solution** : synchroniser, mais conserver la conditionnalité au test instable dans lower et upper
  - upper[1][1] = (choix p1 == 1) ? 4.9 : -5.3 et upper[3][3] = (choix p1 == 1) ? 5.3 : -4.9

Entiers et flottants conditionnels

- utilisation de std::shared_ptr pour partager des bouts d'arbre (copy on write)
- Ordre total sur les identificateurs





- Diminution très forte attendue de la complexité (temps) d'analyse: 10 versus 6^{100000}
- Un même test instable intervient sur plusieurs variables qui intéragissent dans la suite de l'algo pour obtenir la continuité de certains résultats

Adaptation des containers

- `std::vector<int>` devient `TIntegerVectorBranch<int>`
  - champs `size` de type `TIntegerBranch<size_t>`
  - champs `content` de type `std::vector< TIntegerBranch<int> >`
  - méthode combinant les arbres conditionnels de la taille avec l'arbre conditionnel des éléments
    - `push_back` intervient sur différentes cellules de `std::vector`, ainsi que `pop_back`

- `std::vector<double>` devient `TFloatingPointVectorBranch<double>`
  - champs `size` de type `TIntegerBranch<size_t>`
  - champs `content` de type `std::vector< TFloatingBranch<double> >`

- Modèle différent à trouver pour chaque container et ses itérateurs

Entiers et flottants conditionnels

- Utilisé sur des scénarios fins d'analyse

- Adapté pour le traitement des tables d'interpolation sur des scénarios larges

```
// x ∈ [0.0, 3.99], erreur ∈ [-10^-3, 10^-3]
int i = (int) x;
// cas 1: x ∈ [0.0, 1.0[ et i = 0 TODO = test instable
// cas 2: x ∈ [1.0, 2.0[ et i = 1
// cas 3: x ∈ [2.0, 3.0[ et i = 2
// cas 4: x ∈ [3.0, 3.99] et i = 3
```

- Raisonnements par cas avec reconstitution des erreurs avec des annotations spéciales

Analyse de robustesse avec des méthodes formelles

- Généralisation sur l'utilisation de scénarios larges + analyse de bibliothèques sans information sur les arguments

- Comprendre ce que fait l'algorithmique

- Remplacer un container complet avec une formule contenant
  - de nouveaux domaines avec de la logique d'ordre supérieur pour gérer les pointeurs dans les std::map

    $\exists$ *f, fonction injective: [0, N-1]* $\rightarrow$ *IN. f(index) = chemin binaire dans l'arbre rouge-noir.*

    $\forall$ *index* $\in$ [0, N-1]. distances_set[f(index)] = $\sum_{1 \leq i \leq index}$ *distance(arr[index−1], arr[index])*
    - en lien avec l'origine de leur construction
  - des méthodes qui permettent de manipuler ces nouveaux domaines

- Stratégie : laisser à l'utilisateur expert le soin de décrire et de déboguer ses propres domaines avec un outillage adapté