Ingénierie des systèmes orientés-objet

Laboratoire 3

Application graphique liée à l'évolution différentielle

Table des matières

Introduction

Ce dernier laboratoire consiste à réaliser une application graphique liée à l'algorithme d'évolution différentielle.

Objectifs spécifiques

Les objectifs pédagogiques de ce laboratoire sont énumérés en ordre décroissant d'importance:

- Prendre contact avec :
  - la bibliothèque professionnelle `Qt`;
  - la bibliothèque `delib`, spécifiquement créée pour ce projet, implémentant l'algorithme d'évolution différentielle.
- Mettre en pratique l'utilisation de l'encapsulation, de l'héritage et du polymorphisme à travers trois parties distinctes du projet :
  - les bibliothèques :
    - `Qt`
    - `delib`
  - l'application
- Reconnaître, apprécier, exploiter et concevoir des outils logiciels basés sur la puissance du polymorphisme.
- Développer deux solutions pertinentes à des problèmes d'optimisation intéressants en utilisant l'algorithme d'évolution différentielle.
- Être capable de réaliser un programme dont le schéma de conception est partiellement donné et concevoir les parties manquantes.
- Apprendre à développer une application utilisant une bibliothèque professionnelle et non professionnelle.
- Mettre en pratique les notions couvertes tout au long de la session :
  - les concepts du paradigme orienté objet
  - la pratique de la conception UML
  - le langage `C++`
  - l'application d'une norme de codage
  - les bonnes techniques de programmation.

Présentation générale du projet

Ce projet consiste à créer une application graphique exploitant l'algorithme d'évolution différentielle permettant la résolution de quatre problèmes différents. La bibliothèque `Qt` est utilisée pour créer l'application graphique. La bibliothèque `delib` est l'implémentation de l'algorithme d'évolution différentielle utilisée. Parmi les quatre problèmes à résoudre, deux sont déjà réalisés, un consiste à réaliser un problème imposé alors que le dernier consiste à résoudre un problème ouvert.

Ainsi, ce projet est divisé en quatre parties principales :

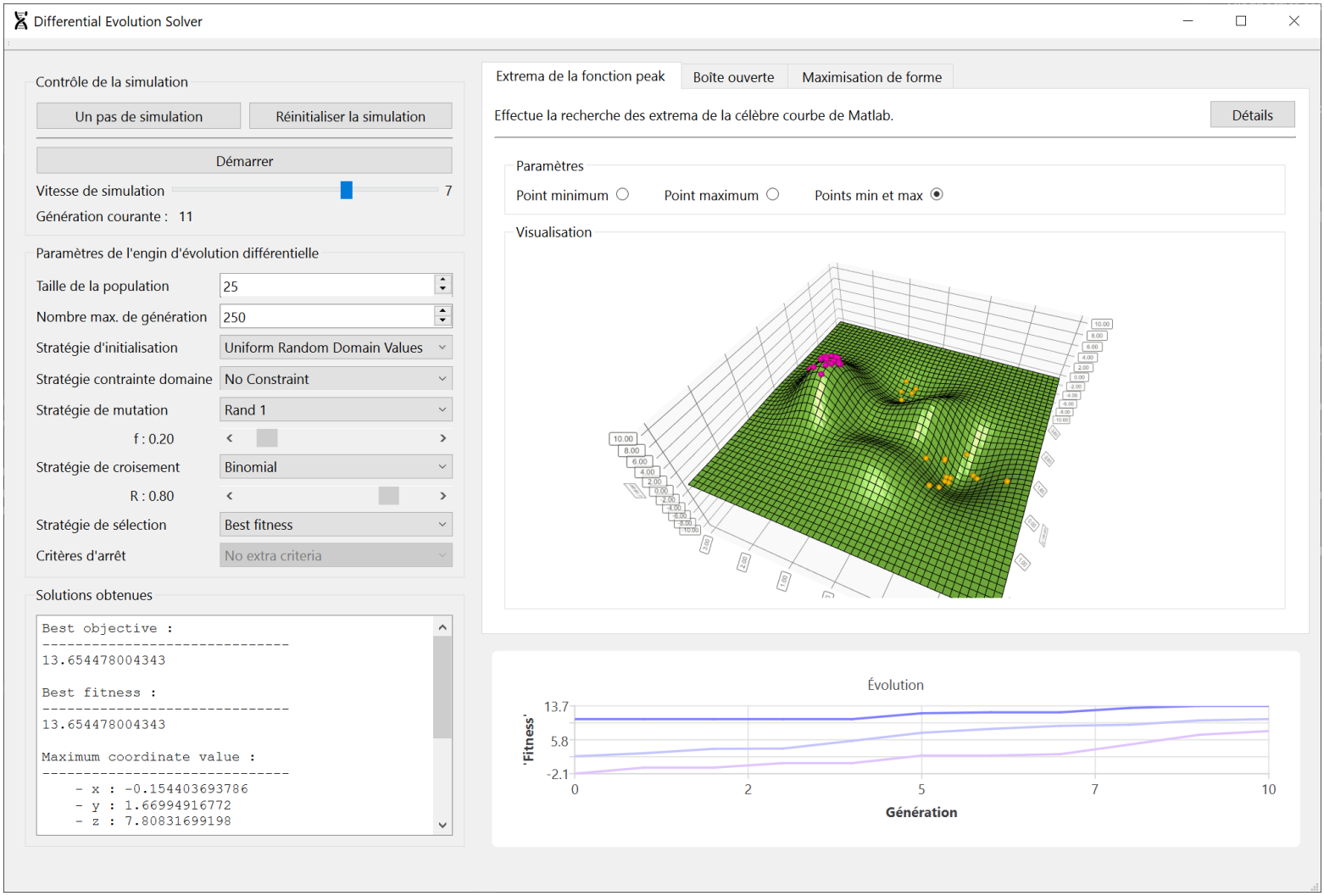
1. Assemblage de l'application principale incluant la mise en place de deux solutions existantes.
2. Création d'une nouvelle solution répondant à un problème imposé.
3. Création d'une nouvelle solution répondant à un problème ouvert.
4. Création d'une nouvelle stratégie de paramétrisation de l'algorithme.

Pour y arriver, 10 compétences techniques sont à développer :

- Plus spécifiquement liées à `Qt` et à l'infrastructure données :
  - Pratiquer les notions d'assemblage de *widgets* (structure composite de l'application).
  - Pratiquer la mise en application des signaux et des connecteurs.
  - Pratiquer la notion de dessin personnalisée.
  - Créer deux panneaux encapsulant la résolution d'un problème tout en offrant la possibilité de paramétrisation et de visualisation.
- Plus spécifiquement liées à `delib` :
  - Formuler une solution personnalisée à un problème donné (domaine, fonction objective et paramètres spécifiques de la problématique).
  - Configurer les différents paramètres génériques de l'algorithme.
  - Développer une stratégie servant de paramétrisation à l'algorithme.
- Plus spécifiquement liées à UML :
  - Comprendre et exploiter une conception UML existante.
  - Réaliser la conception de votre application à travers UML.
- Comprendre comment le polymorphisme permet le développement d'une telle application en simplifiant la plupart des étapes.

Application à développer

Vous devez créer l'application graphique suivante.



Présentation de l'interface utilisateur de l'application à développer

L'application est divisée en cinq sections mutuellement dépendantes :

1. Contrôle de la simulation : En haut à gauche, le panneau permettant de contrôler la résolution de la simulation.
2. Paramètres de l'engin d'évolution différentielle : À gauche au centre, le panneau permettant la configuration de l'engin de résolution. Ce panneau paramètre spécifiquement l'engin d'évolution sans considération pour le problème à résoudre.
3. Solutions obtenues : En bas à gauche, le panneau donnant des informations techniques sur l'évolution et la meilleure solution obtenue.
4. Problème à résoudre : En haut à droite, le panneau permettant de :
    a. Sélectionner le problème à résoudre.
    b. Configurer le problème à résoudre.
    c. Présenter une rétroaction pour :
        i. la présentation du problème,

ii. la résolution du problème pendant la phase de résolution,
iii. la meilleure solution obtenue.

5. Historique d'évolution : En bas à droite, un panneau présentant un diagramme de trois lignes montrant l'évolution des qualités obtenues (pire, moyenne et meilleure).

Widgets disponibles et assemblage

Vous avez à votre disposition plusieurs *widgets* déjà réalisés implémentant les parties importantes de cette application :

- `QDEControllerPanel`, représentant la zone de contrôle de l'algorithme
- `QDEEngineParametersPanel`, encapsulant les *widgets* permettant la paramétrisation de l'engin d'évolution différentielle
- `QDESolutionTabPanel`, un panneau facilitant la gestion de plusieurs solutions à résoudre dans la même application
    - Il est important de comprendre que ce widget possèdera lui-même plusieurs instances de la classe `QDESolutionPanel`. Cette dernière est l'interface commune de toutes les problématiques adressées par l'application.
- `QDEBestResult`, un *widget* présentant la meilleure solution obtenue lors de l'évolution
- `QDEHistoryChartPanel`, un *widget* présentant l'évolution des solutions au fil des générations

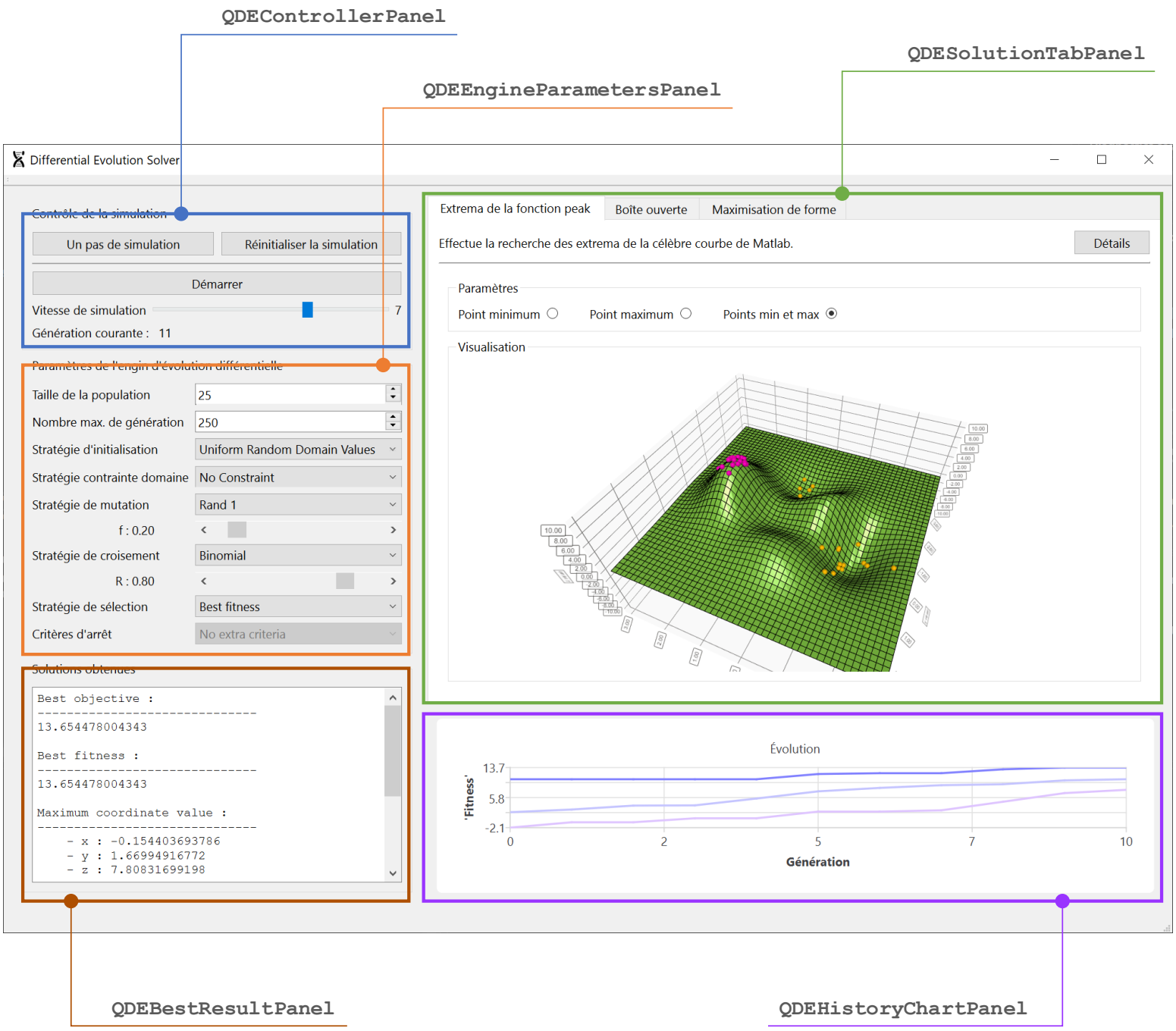
L'assemblage de cette application set réalisé en trois parties distinctes :

1. Effectuer les assemblages des *layouts* et *widgets* nécessaires au montage présenté.
2. Établir les connexions appropriées entre les différents composants.
3. Ajouter les panneaux des deux solutions mises à votre disposition.

Sachez que cette partie du laboratoire compte uniquement pour 25%.

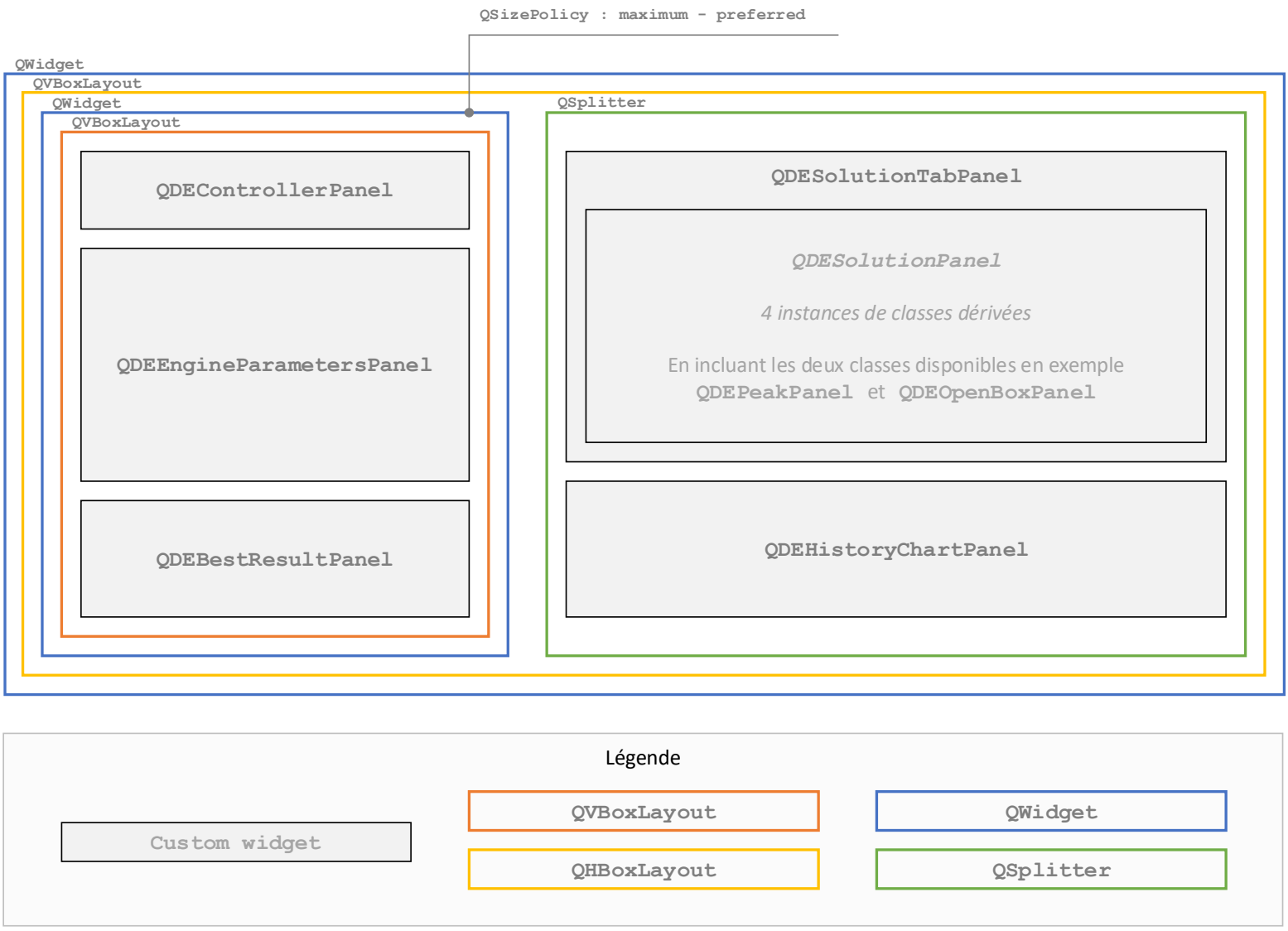
Partie 1 – Instanciation des widgets et assemblage

L'image suivante présente les *widgets* mis à votre disposition et leur disposition relative.



Disposition des *widgets* principaux mis à votre disposition

C'est à vous de réaliser l'assemblage nécessaire. Pour vous aider, le schéma suivant indique comment les *widgets* sont inclus les uns dans les autres.



Guide d'assemblage représentant les compositions nécessaires.

Lorsque vous aurez terminé cette partie, le `QDESolutionTabPanel` sera vide et aucune solution ne sera présente. La troisième partie vous permettra d'ajouter quelques problèmes à résoudre.

Note importante, plusieurs panneaux requièrent la passation d'un objet `QDEAdapter` à même leur constructeur à titre d'argument. Ainsi, ces panneaux sont liés, par agrégation, à l'objet de simulation. À titre d'exemple, voir la définition du constructeur suivant.

```
QDEControllerPanel(QDEAdapter const & dEAdapter, QWidget * parent = nullptr);
```

Ceci implique que votre application doit posséder par composition une instance de `QDEAdapter` (voir plus loin pour les détails). De plus, puisque chaque panneau requiert cet objet à même son constructeur, il est essentiel que l'objet `QDEAdapter` soit créé et initialisé au tout début de l'application, avant les *widgets* la constituant.

Partie 2 – Connexion des signaux aux connecteurs

Les différents *widgets* à assembler requièrent la connexion de 8 *signals* et *slots*.

	Émetteur		Récepteur	
	Objet*	**Signal**	**Objet***	**Slot**
1	QDEEngineParametersPanel	parameterChanged	QDEControllerPanel	resetSimulation
2	QDEEngineParametersPanel	parameterChanged	QDESolutionTabPanel	updateVisualization
3	QDESolutionTabPanel	solutionChanged	QDEEngineParametersPanel	setParametersFromSolution
4	QDESolutionTabPanel	solutionChanged	QDEControllerPanel	resetSimulation
5	QDEControllerPanel	evolutionStarted	QDEEngineParametersPanel	disable
6	QDEControllerPanel	evolutionStopped	QDEEngineParametersPanel	enable
7	QDEControllerPanel	evolutionStarted	QDESolutionTabPanel	disable
8	QDEControllerPanel	evolutionStopped	QDESolutionTabPanel	enable

* **IMPORTANT** : Les objets identifiés dans ce tableau représentent les instances créées à partir des classes identifiées. Puisque les noms peuvent changer selon les implémentations, vous devez utiliser les noms d'objets que vous avez définis.

Il est intéressant de réaliser qu'il existe un grand nombre de connexions supplémentaires dans l'application. Les *widgets* créés s'autoconnectent avec leurs constituants et l'objet `QDEAdapter` passé en argument pour ceux concernés.

Partie 3 – Ajouter les panneaux de résolution de problème

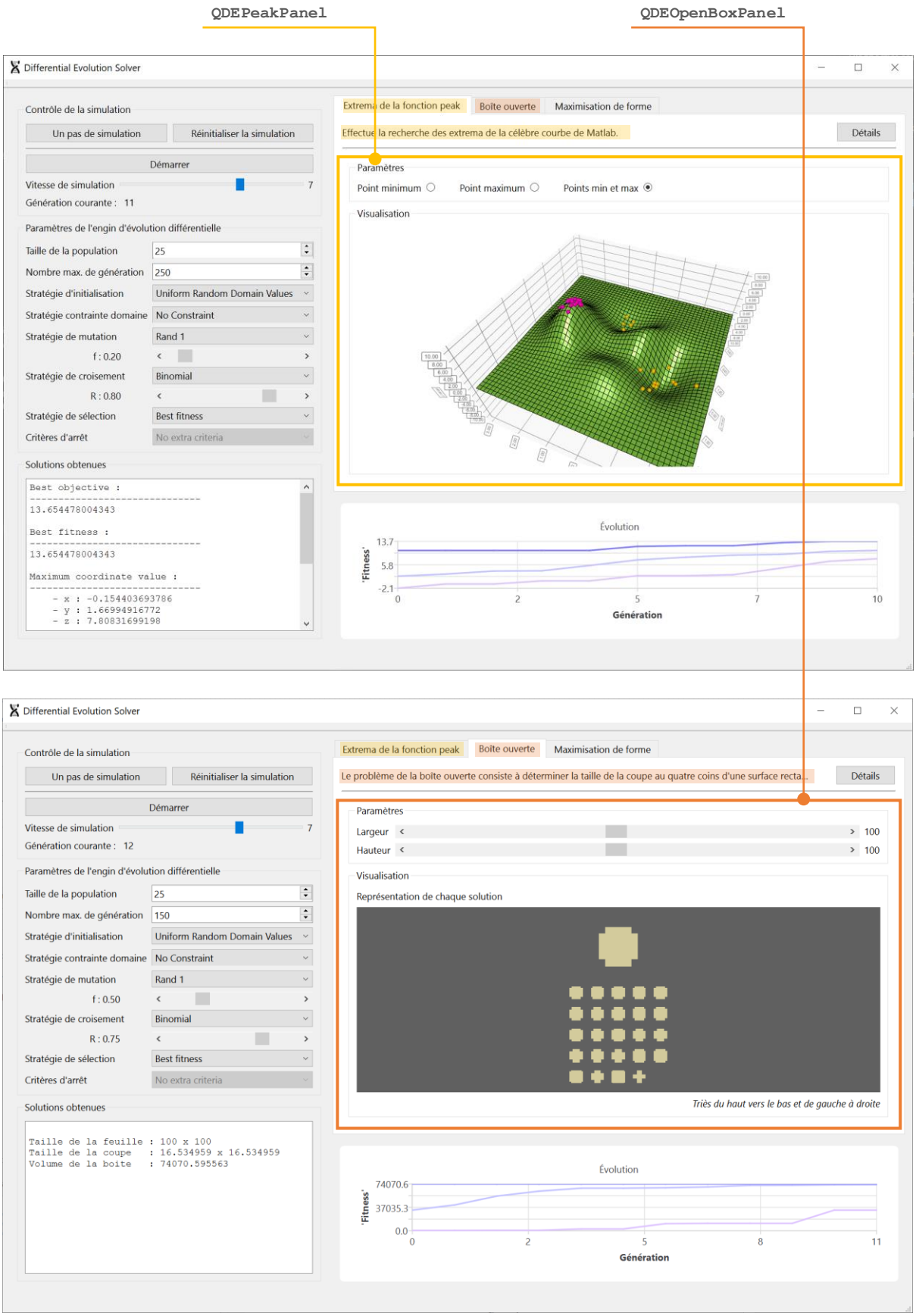
Comme discuté au point 1, le panneau `QDESolutionPanel` ne possède pas de problème à résoudre par défaut. C'est à vous de les ajouter explicitement. Il est possible de le faire à la fin du constructeur de l'application.

Il est important de comprendre que ces panneaux spécialisés sont réalisés en héritant de la classe `QDESolutionPanel` (voir la section concernée pour plus de détails) et sont insérés dans le `QDESolutionTabPanel`. Pour chaque sous-panneau ajouté, un onglet au nom du problème est ajouté et le panneau inséré dans sa zone inférieure. Automatiquement, un sommaire est visible et le bouton `Détails` permet d'afficher une boîte de dialogue présentant la description du problème.

Vous avez, à titre d'exemple, deux panneaux mis à votre disposition :

- `QDEPeakPanel`, la solution au problème d'extrema de la courbe `peak` de `Matlab`.
- `QDEOpenBoxPanel`, la solution au problème de la boîte ouverte

C'est la méthode `QDESolutionTabPanel::addSolutionPanel` qui permet l'ajout des panneaux. N'oubliez pas que les panneaux à ajouter doivent être alloués dynamiquement et que la mémoire allouée devient la propriété et la responsabilité du `QDESolutionTabPanel`.



Présentation des panneaux de solution – Panneaux en encadré et textes en surbrillance

Widgets supplémentaires mis à votre disposition

Outre tous les *widgets* disponibles pour la mise en place de la solution (ceux terminant par `Panel`), vous avez à votre disposition deux *widgets* utilitaires supplémentaires :

- `QElidedLabel`, un `QLabel` ne contraignant pas sa largeur par celle du texte affiché. Plutôt, ce *widget* effectue une troncation automatique du texte et ajoute les points de suspension.
- `QImageViewer`, un `QLabel` spécialisé dans l'affichage d'une image. Cette dernière est toujours centrée et redimensionnée pour occuper entièrement l'espace disponible.

Ces *widgets* sont utilisés par ceux de l'application (ceux terminant par `Panel`).

Problèmes personnalisés à résoudre

Vous avez deux problèmes supplémentaires à développer :

1. Le problème d'optimisation d'une forme géométrique.
2. Un problème personnalisé que vous devez vous-même définir.

Cette section présente les parties suivantes :

1. Présentation de l'approche générale à adopter pour résoudre un problème.
2. Présentation du problème imposé.
3. Présentation des balises et contraintes du problème ouvert.

Création d'un panneau présentant un problème à résoudre

Pour réaliser un nouveau panneau et aborder un problème spécifique, deux étapes sont à réaliser :

- Avec `delib`, réaliser une classe adressant spécifiquement le problème d'optimisation.
- Avec le panneau `QDESolutionPanel`, réaliser une classe permettant d'avoir une interface graphique interactive et l'insérer dans l'application.

Il est vraiment important de réaliser que la bibliothèque `delib` est entièrement indépendante de `Qt`. Autrement dit, `delib` fonctionne sans `Qt` et peut être utilisé dans n'importe quel autre projet.

Utilisation de delib

La bibliothèque `delib` permet de créer simplement une classe qui définit tous les éléments du problème à résoudre. Contrairement à l'implémentation que vous avez réalisée au laboratoire précédent, le polymorphisme est utilisé pour permettre une flexibilité et modularité remarquable tout en gardant le formalisme désiré.

Vous trouverez toute la documentation de la bibliothèque sur le petit site web donné. Toutefois, La classe `de::SolutionStrategy` est la partie la plus importante à connaître pour utiliser la librairie. Plus spécifiquement, vous trouverez la documentation pertinente à ces endroits :

- Sur le site web de `delib` :
    - Sur la page d'accueil, deux exemples de code sont présents.
    - La documentation spécifique de `de::SolutionStrategy`.

- Dans les panneaux donnés solutionnant les deux problèmes déjà résous, des implémentations complètes à titre d'exemple et de référence :
    - `QDEPeakPanel`, présente comment utiliser une solution existante. La solution existe déjà dans la bibliothèque et il est possible de voir sa définition. Toutefois, son implémentation n'est pas accessible. Voir les classes :
        - `PeakFunctionMinSolution`
        - `PeakFunctionMaxSolution`
        - `PeakFunctionMinMaxSolution`
    - `QDEOpenBoxPanel`, présente une solution complète (déclaration et implémentation). Voir :
        - `QDEOpenBoxPanel::OpenBoxStrategy`

Malgré tout, la présentation qui suit est un guide intéressant sur la façon d'aborder un tel développement et les éléments importants à considérer :

- La classe `de::SolutionStrategy` est abstraite et doit être dérivée pour pouvoir être utilisée. Cette classe est utilisée polymorphiquement pour représenter le problème à résoudre.
- À même la liste d'initialisation membre de votre classe, l'*appel* explicite du constructeur parent **est requis** et doit fournir trois chaînes de caractères décrivant :
    - Le titre, une très courte identification
      c'est ce qui sera visible dans l'onglet
    - Le sommaire, une courte présentation présentant le problème.
      c'est ce qui sera affiché dans l'interface utilisateur dans la section sommaire
    - Une description, une description technique détaillée des constituants importants. On s'attend à voir :
        - La dimensionnalité du problème.
        - Une description du domaine pour chaque dimension.
        - Une présentation et justification de la fonction objective.

      c'est ce qui sera affiché lorsque l'usager appuiera sur le bouton Détails
- À même le corps du constructeur :
    - Il est **obligatoire** de définir le domaine.
        - Le domaine est disponible à titre de variable membre protégée.
        - La dimensionnalité et tous les intervalles doivent être définis.
    - Optionnellement, les paramètres suivants peuvent être définis :
        - `de::OptimizationStrategy`, maximisation par défaut, mais la minimisation est possible
        - `de::FitnessStrategy`, fonction identité par défaut
- Vous **devez** substituer (*override*)
    - La fonction `toString` :
        - Cette dernière permet de convertir une solution à une chaîne de caractères.
        - On s'attend à une présentation technique, mais élégante de la solution.
        - Ce texte sera visible et mis à jour dans le panneau `QDEBestSolutionPanel`.
    - La fonction `process` :
        - Cette fonction est la fonction objective, donc la plus importante!
        - Vous devez déterminer la valeur de la solution donnée et retourner un pointage de performance.
    - La fonction clone :
        - Cette fonction sert à copier l'instance polymorphique de la classe.

        - Pour cette fonction, vous devriez simplifier le développement en utilisant la macro définie :
          `DEFINE_OVERRIDE_CLONE_METHOD`
        - Voir son usage dans les exemples donnés.
- Optionnellement, vous pouvez substituer (*override*) :
    - La fonction `engineParameters` :
        - Cette fonction retourne par défaut les paramètres par défaut de la classe `de::EngineParameters`.
        - L'intention derrière cette fonction est de retourner une recommandation des paramètres pour accélérer et faciliter la résolution du problème.
        - Cette implémentation est techniquement optionnelle. Toutefois, passer à côté de son implémentation est vraiment contre-productif. Il est donc attendu que vous en fassiez une substitution (*override*).
    - La fonction `validateReadyness` :
        - Cette fonction sert à valider si votre solution est prête à être résolue.
        - Par défaut, la fonction retourne vraie.
        - Cette fonction peut être essentielle si votre solution requiert une certaine dépendance à d'autres constituants et que vous ne voulez pas faire planter l'application avant de lancer une simulation.
        - Généralement peu utilisée.
    - La fonction `prepare` :
        - Si votre solution doit préparer certains éléments (calculs, mémoire, ressources …) une fois avant la simulation, c'est cette fonction qui permet de le faire.
        - Par défaut, cette fonction ne fait rien.
        - Encore une fois, c'est une fonction utilitaire qui donne plus de flexibilité, mais est généralement peu utilisée.
    - Les deux fonctions surchargées `initializeRandomly` :
        - La première initialise une solution à la fois :
            - Cette fonction permet de définir une stratégie d'initialisation aléatoire spécifique au problème adressé.
            - Par défaut, initialise avec une valeur pseudo-aléatoire uniformément répartie selon le domaine.
            - Peut être très pertinent pour certain problème.
        - La deuxième initialise la population entière :
            - Permet une stratégie d'initialisation plus complexe en tenant compte de toute la diversité de la population.
            - Par défaut, appelle la fonction `initializeRandomly` individuelle pour chaque solution.
            - Moins fréquemment utilisé, mais tout de même pertinent.
- Un dernier point important est de réaliser que votre classe peut posséder toutes autres informations nécessaires (incluant ses accesseurs et mutateurs). Il est ainsi possible de paramétrer le problème via une classe plutôt que via une fonction comme il a été fait au laboratoire précédent.

Utilisation de QDESolutionPanel

Le panneau `QDESolutionPanel` offre une interface de programmation uniformisée afin de permettre au logiciel d'exécuter la résolution d'un problème.

Cette classe héritant de `QWidget` est abstraite et doit être dérivée pour présenter une problématique spécifique.

Sachez d'abord que vous trouverez de l'information :

- Sur le site web des classes de l'application (`deapp`) :
  - Vous trouvez une page détaillant la classe `QDESolutionPanel`.
- Dans les deux panneaux donnés en exemple :
  - Les panneaux `QDEPeakPanel` et `QDEOpenBoxPanel`.
  - Vous trouverez à la fois leur déclaration `*.h` et leur définition `*.cpp`.
  - Ces panneaux sont donnés à titre de référence et il est attendu que vous preniez le temps de comprendre leur structure.
  - Faites attention de ne pas vous étourdir avec tous les détails (comme l'affichage du graphique 3D qui n'est pas nécessairement sujet au cours). L'objectif est de comprendre la structure, les constituants et les liens qui les unis.

Lorsque le panneau est inséré par `QDESolutionTabPanel::addSolutionPanel`, plusieurs opérations sont faites automatiquement :

- Création d'un onglet avec le nom approprié.
- Insertion du panneau dans l'application.
- Mise à jour des textes : le sommaire et celui associé au bouton `Détails`.
- Connexion des *signaux* et des *slots*.

Évidemment, ce panneau a pour rôle de fournir une interface uniforme entre l'application et les problèmes de façon pouvoir les paramétrer avant l'évolution et les visualiser pendant l'évolution. Pour que votre panneau devienne un constituant de l'application, ce dernier doit correspondre à des règles strictes s'harmonisant avec le logiciel.

Cette classe est conceptuellement relativement simple. Trois concepts sont à connaître, et quoiqu'optionnel, deux aspects doivent être considérés.

Voici les trois concepts à connaître :

- Vous **devez** réaliser la substitution de la méthode `buildSolution` :
  - Cette méthode retourne tout simplement une instance de la classe que vous avez créée précédemment.
  - Autrement dit, le panneau ne gère pas le problème à proprement dit, mais offre un lien standardisé entre l'application et la bibliothèque `delib`.
  - Cette fonction doit allouer dynamiquement la nouvelle solution et la retourner. C'est l'application qui devient propriétaire et responsable de cette allocation.
- Vous **pouvez** réaliser la substitution de la méthode `updateVisualization` :
  - Cette méthode est appelée automatiquement lorsqu'un évènement justifie de mettre à jour la visualisation du panneau.
  - On se sert de cette fonction principalement pour faire la mise à jour de la vue pendant l'évolution.
- Il existe un signal permettant d'informer l'application principale qu'un changement a eu lieu : `parameterChanged`. Par défaut, ce signal n'est jamais émis. Toutefois, c'est votre responsabilité de l'émettre au moment opportun.

Pour bien utiliser un panneau, il ne faut pas perdre de vue que ce dernier est un widget et qu'il est destiné à servir d'interface utilisateur graphique dans l'application. Cette dernière est à forte

teneur éducative et tout est mis en place pour que le panneau offre deux services. Même si ces services sont formellement optionnels, on s'attend à ce qu'ils soient utilisés à bon escient.

- Paramétrisation du problème :
  - Votre *widget* doit afficher une interface graphique permettant de déterminer les paramètres spécifiques au problème.
  - Dans les exemples précédents, des sections sont créées pour :
    - `QDEPeakPanel` : Offrir le choix entre :
      - la valeur minimum
      - la valeur maximum
      - les 2 extrema globaux
      - les 6 extrema locaux.
    - `QDEOpenBoxPanel` : Définir la taille initiale de la feuille :
      - largeur
      - hauteur
    - Évidemment, ce sont des exemples simples et vos panneaux peuvent présenter une vaste gamme d'options.
  - Cet aspect implique deux considérations :
    - Vous devez créer et assembler ces *widgets* de paramétrisation dans le constructeur de cette classe.
    - Dès qu'un paramètre change, vous devez informer l'application d'un changement via l'émission du signal `parameterChanged`.
- Visualisation de l'évolution :
  - Votre *widget* doit afficher une rétroaction à l'usager pour mieux comprendre et suivre l'évolution de la solution.
  - Dans les exemples précédents, des sections sont créées pour :
    - `QDEPeakPanel` : on affiche la courbe `peak` de `Matlab` en 3d et les différentes solutions évoluent sur la surface.
    - `QDEOpenBoxSolution` : on affiche toutes les solutions (les feuilles découpées) simultanément pendant l'évolution.
  - Ces présentations sont des exemples de rétroaction possible.
    - Plusieurs autres types de rétroaction sont envisageables selon le problème adressé.
    - Principalement, voir l'évolution dans l'espace du problème et dans l'espace des solutions.
    - Parfois, selon la nature du problème, une représentation graphique n'est pas nécessairement possible ou pertinente, alors d'autres rétroactions restent intéressantes.
    - Dans tous les cas, une rétraction doit être produite.
- Dans les deux cas, il est attendu que vous avez des éléments de paramétrisation et une forme de rétroaction adaptée, pertinente et facile à comprendre.

QDEAdapter

Lors de la réalisation d'un projet informatique, il est très fréquent d'avoir à utiliser des bibliothèques et des outils génériques qui ne sont pas destinés à être utilisés les uns avec les autres. C'est exactement la situation ici avec `delib` et `Qt`. En effet, étant indépendant de `Qt`, `delib` n'utilisent pas intrinsèquement les utilités qu'offre `Qt`. Attention à ne pas confondre, avec ces utilités viennent des avantages, mais aussi des contraintes, limitations et désavantages.

Par exemple, la classe `de::DifferentialEvolution` n'utilise pas les signaux, slots et autres mécanismes de `Qt`. Pourtant, il serait vraiment pertinent et utile que cette application puisse les utiliser s'ils existaient.

Il existe une approche orientée objet permettant de faire ce passage : un adaptateur. Ce dernier permet de créer une interface de programmation différente pour une classe ou un ensemble de classes existantes. Ainsi, l'adaptateur crée le pont entre deux mondes. À cet effet, vous avez la classe `QDEAdapter` qui possède à l'interne un objet `de::DifferentialEvolution` et offre une interface adaptée pour cette application.

`QDEAdapter` implémente les accès nécessaires à son objet d'évolution différentielle privée et tous les mécanismes de signaux et de slots qu'offre `Qt`.

Ainsi, pour votre projet, **vous devez créer un objet de la classe** `QDEAdapter` à même votre application principale. C'est cet objet qui devient le moteur de l'évolution différentielle.

Problème personnalisé 1 – Optimisation géométrique

Vous devez concevoir et développer un panneau adressant un problème d'optimisation géométrique. Ce dernier est pertinent, car il présente des avantages académiques intéressants :

1. Il s'explique simplement et il est facile à comprendre.
2. Il n'existe pas de solution triviale.
3. Il est facile de trouver des critères de paramétrisation.
4. Il est facile de créer une rétroaction visuelle intéressante.

Énoncé

L'énoncé est le suivant : on vous demande de trouver la transformation affine permettant de disposer la plus grande forme géométrique sur une surface parsemée d'obstacles.

Plus spécifiquement, vous devez résoudre le problème suivant :

- vous disposez d'un canevas à deux dimensions (surface rectangulaire) :
  - le canevas est défini par sa largeur et sa hauteur
  - la taille du canevas est fixé au début du problème et ne change pas pendant l'évolution
- sur le canevas se trouvent n_p point à deux dimensions correspondant à des obstacles :
  - $n_p \geq 0$
  - chaque point est disposé sur le canevas au début du problème et ne change pas pendant l'évolution
  - la disposition est déterminée aléatoirement
- vous disposez d'une forme géométrique quelconque à deux dimensions :
  - la forme est définie par un polygone de n_s sommets :
    - $n_s \geq 3$
    - le polygone peut être convexe ou concave, mais ne doit pas se croiser lui-même
  - il est possible d'effectuer ces trois types de transformation sur le polygone :
    - translation;
    - rotation;
    - homothétie (mise à l'échelle ou « *scaling* »);
  - vous ne pouvez pas déformer la forme d'aucune façon;
  - la forme :
    - peut :
      - toucher au contour du canevas
      - toucher aux obstacles;
    - ne peut pas :
      - dépasser la zone rectangulaire du canevas
      - avoir un obstacle à l'intérieur;
- on cherche la transformation qui maximise la surface de la forme à l'intérieur du canevas sans entrer en contact avec les obstacles.

Finalement, il est attendu que la forme soit disposée de façon telle à maximiser sa taille sans enfreindre les règles énoncées.

Contraintes de développement

Vous devez respecter ces contraintes de développement:

- Conception `UML` :
    - On vous demande de produire un diagramme de classes `UML` de la solution et du panneau **avant** leur réalisation.
    - Pendant le développement, le diagramme doit être maintenu à jour.
    - Ce diagramme de classe est à remettre dans votre rapport.
- Approche polymorphique :
    - Outre le polymorphisme que vous exploitez tout au long du projet, vous devez créer votre propre approche polymorphique pour la création des polygones.
    - Cette partie de la conception doit être mise en évidence dans votre diagramme de classes.

Interface utilisateur

Pour chaque résolution de problème, trois paramètres fondamentaux doivent être déterminés et rester immuables tout au long de la résolution du problème :

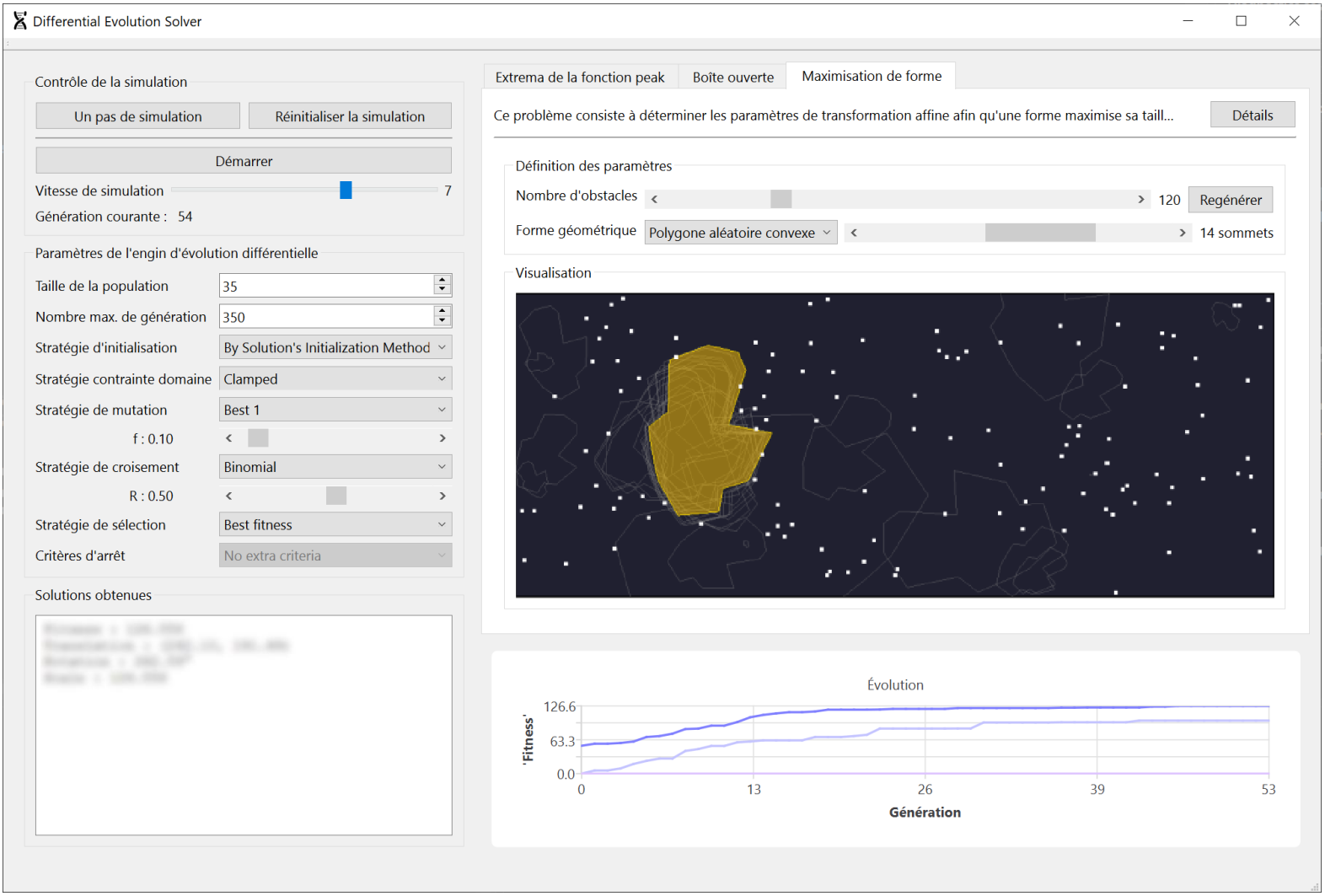
- la taille du canevas (la paramétrisation par programmation est suffisante);
- le nombre d'obstacles et la disposition de ces derniers;
- la forme géométrique à utiliser (au moins 3 formes géométriques doivent être offertes).

La forme peut subir une ou plusieurs transformations affines (translation, rotation et homothétie), mais ne peut être modifiée autrement. Par exemple, si la forme ressemble à un « L », elle le restera jusqu'à la fin de la résolution. Toutes les proportions du « L » seront gardées : longueur relative entre la barre verticale et la barre horizontale ainsi que l'épaisseur relative du trait.

De plus, on s'attend à une rétroaction graphique où sont affichés :

- le canevas
- la liste d'obstacles
- toutes les formes issue de la population
- la meilleure forme mise en évidence sur le dessin.

La capture suivante donne un exemple possible de cette interface utilisateur.



Présentation de l'interface graphique proposée

Indices pour la réalisation de ce projet

Les classes géométriques sont particulièrement intéressantes et pertinentes pour résoudre le problème. Voir :

- `QPointF`
- `QSize`
- `QRectF`
  - `contains` (voir les surcharges)
- `QPolygonF`
  - `boundingRect`
  - `containsPoint`
- `QTransform`
  - `translate`
  - `rotate`
  - `scale`
  - `map` (voir les surcharges)

Pour l'affichage, les outils de dessins de Qt sont nombreux, les classes et méthodes suivantes sont à connaître :

- `QImage`
- `QPen`
- `QBrush`
- `QPainter`
    - `setPen`
    - `setBrush`
    - `draw...` (les très nombreuses méthodes de dessin)
        - `drawRect`
        - `drawPolygon`
    - `rotate`, `translate`, `scale`, `save` et `restore`

Évaluation

Sachez que cette partie du laboratoire compte pour 35% et que les points sont attribués selon :

- la qualité et l'effort de la conception :
    - qualité de la solution orientée objet;
    - effort particulier lié au développement du polymorphisme;
    - qualité et pertinence du diagramme de classes `UML`;
- la pertinence de la définition de la solution (domaine, fonction objective, paramètres proposés)
- la qualité de l'assemblage des *widgets*
- la paramétrisation du problème
- la qualité de la rétroaction
- la qualité générale de la réalisation.

Problème personnalisé 2 – Sujet ouvert

Vous devez produire la solution pour un problème de votre choix en considérant les contraintes suivantes :

- Plus votre problème possède de dimensions **pertinentes**, plus vous avez de points.
- Le problème doit offrir au moins trois éléments de paramétrisation influençant la simulation.
- Vous devez offrir une visualisation pertinente de l'évolution. Vous n'êtes pas tenu d'offrir un élément graphique si cette dernière ne s'applique pas à votre problème.
- Votre solution doit converger (imparfaitement est acceptable)
- Vous avez carte blanche sur le sujet abordé pourvu que ce dernier soit de bon goût et absent d'éléments de moral douteuse.
- Votre sujet **doit** être validé par l'auxiliaire d'enseignement au plus tard à la deuxième période du laboratoire.

Contraintes de développement

Vous devez respecter ces contraintes de développement:

- Conception `UML` :
  - On vous demande de produire un diagramme de classes `UML` de la solution et du panneau **avant** leur réalisation.
  - Pendant le développement, le diagramme doit être maintenu à jour.
  - Ce diagramme de classe est à remettre dans votre rapport.
- Approche polymorphique :
  - Il est attendu qu'une approche polymorphique soit utilisée pour votre dernière problématique.
  - Cette partie de la conception doit être mise en évidence dans votre diagramme de classes.

Évaluation

Sachez que cette partie du laboratoire compte pour 40% et que les points sont attribués selon :

- la pertinence :
  - du sujet;
  - des options paramétrables;
  - de la rétroaction;
- le niveau de difficulté lié au projet :
  - la dimensionnalité du problème;
  - plusieurs autres aspects sont possibles dont la fonction objective;
- la qualité et l'effort de la conception :
  - qualité de la solution orientée objet;
  - effort particulier lié au développement du polymorphisme;
  - qualité et pertinence du diagramme de classes `UML`;
- la qualité de la réalisation.

Ressources

Ce projet est relativement exigeant dans le sens où une tonne d'informations est à assimiler rapidement. Pour vous aider à bien naviguer à travers ce large volume de données, vous devez garder en tête les trois sources d'informations suivantes :

- La documentation en ligne de la bibliothèque `Qt`.
  - Cette dernière est exhaustive et de très bonne qualité.
  - Toutefois, faites attention de trouver le bon équilibre entre lire la documentation et poser des questions à l'auxiliaire d'enseignement.
  - La documentation est trop vaste pour pouvoir être lue entièrement.
  - Dans ce document, on précise aussi quelques aspects à connaître spécifiquement.
- La documentation de la bibliothèque `delib`.
  - Comme dans les laboratoires précédents, vous avez à votre disposition un petit site web présentant une documentation fonctionnelle de la librairie.
  - Le site possède une page d'accueil vous permettant une prise en main rapide de la bibliothèque. Il est essentiel d'en faire une lecture pour mieux comprendre son usage.
  - N'oubliez pas que `delib` est très semblable à l'algorithme que vous avez implémenté lors du laboratoire précédent. Les différences sont un peu dans ses capacités, mais surtout dans son architecture.
  - La classe la plus importante à connaître pour un développeur qui désire résoudre un problème est : `SolutionStrategy`
  - La documentation se trouve dans : `./delib/doc/html/index.html`
  - Rappelez-vous que vous avez aussi accès au `*.h` de chaque classe. Voir le dossier : `./delib/doc/include/`
- La documentation des *widgets* mis à votre disposition pour créer l'application.
  - Encore une fois, un petit site web est mis à votre disposition pour présenter sommairement les différents *widgets* servant à l'application.
  - La classe `QDESolutionPanel` est certainement la plus importante à connaître en détail puisqu'elle définit l'interface de programmation de tous les problèmes à résoudre.
  - La documentation se trouve dans : `./deapp/doc/html/index.html`
  - Finalement, le projet est assemblé de façon telle que tous les fichiers sont déjà disponibles. Vous avez donc accès aux déclarations `*.h,` mais aussi aux définitions `*.cpp`.
  - Tous les fichiers se trouvent dans le dossier du projet et son préassemblés dans la solution de `Microsoft Visual Studio`.
- Évidemment, les ressources générales du `C++` sont aussi à consulter.
- N'oubliez pas les techniques de bonne pratique pour minimiser la perte de temps.
- Finalement :
  - La réalisation de ce laboratoire repose sur l'accumulation de beaucoup de connaissances.
  - Toutefois, il est monté de façon à provoquer l'opportunité de consolider les sujets présentés pendant la session.
  - Il est donc important de savoir quand et comment poser des questions; que ce soit avec l'enseignant, l'auxiliaire d'enseignement, sur les sites spécialisés, en ligne ou avec `ChatGPT`.

- Sachez qu'un usage intelligent de `ChatGPT` dans un contexte d'apprentissage est considéré une bonne pratique. Néanmoins, soyez sensible au fait qu'une pratique abusive et mal ciblée garantit systématiquement l'échec des objectifs pédagogiques du cours. Cette réalité sera d'autant plus visible lors de l'examen final où vous n'y avez pas droit et où, le double seuil devient un mur difficile à surmonter.

Qt

Installation

Tel qu'indiqué dans les notes de cours, assurez-vous d'avoir :

- installé Visual Studio adéquatement et effectuée sa mise à jour
- installé `Qt` convenablement avec les modules `Charts` et `DataVisualization`
- installé l'extension de `Qt` dans Visual Studio
- fait un test de fonctionnement de l'ensemble.

Utilisation dans ce projet

La bibliothèque `Qt` est principalement utilisée dans ce projet afin de construire l'IUG (Interface Utilisateur Graphique ou, en anglais, *GUI – Graphical User Interface*).

Avec `Qt`, la réalisation d'une IUG est principalement basée sur la classe `QWidget`. Cette classe, héritant de `QObject` et `QPaintDevice`, utilise le polymorphisme afin d'offrir une interface de développement modulaire, flexible et performante.

Tous les contrôles graphiques (communément nommé *widget* en informatique) héritent de `QWidget` : `QPushButton`, `QRadioButton`, `QLineEdit`, `QScrollBar`, `QLabel`, `QTabWidget`, etc. Aussi, chaque *widget* possède une liste d'objets appelés enfants qui sont logiquement disposés à l'écran en position relative au parent. Cette structure en arbre permet un assemblage de *widgets*.

Il est important de comprendre qu'un objet `QWidget` possède (par composition) un objet `QLayout`. À son tour, un objet `QLayout` possède lui-même, entre autres, *m* `QWidget` et *n* `QLayout`.

Assurez-vous de bien comprendre cette notion présentée en classe pour faciliter la réalisation de ce laboratoire.

Solution Visual Studio mise à votre disposition

Vous avez à votre disposition une solution Visual Studio prémontée avec la librairie `delib` et les fichiers nécessaires au projet.

Avant de débuter le projet, assurez-vous que cette solution compile et s'exécute. À son démarrage, vous devriez voir une fenêtre vide.

Contraintes

Voici les contraintes du projet :

- Contraintes techniques :
  - Vous devez respecter la norme de codage imposée.
  - Il est attendu que votre code soit documenté adéquatement.
  - Aucun *warning* ne devrait apparaître lors de la compilation et de l'édition de liens (*linker*).
  - Votre code ne doit pas utiliser :
    - l'instruction `goto`;
    - des variables globales;
    - des littéraux sans variables symboliques (sauf lorsque trivial).
  - Il est obligatoire de réaliser le projet avec le langage de programmation `C++`, les bibliothèques `Qt` et `delib` ainsi qu'utiliser l'environnement de développement `Microsoft Visual Studio`. Pour les deux cas, l'utilisation des versions les plus récentes est attendue.
- Contraintes liées au travail et à la remise :
  - Vous avez jusqu'à la fin de la session pour réaliser ce laboratoire.
  - Ce travail est à faire en équipe de deux et la pratique de la programmation en binôme est obligatoire.
  - Sans faire de plagiat, l'entre aide constructive est encouragée.
  - Assurez-vous de bien comprendre les éléments que vous remettez. Si l'auxiliaire d'enseignement vous pose une question sur votre travail et que vous n'êtes pas capable de répondre ou d'expliquer ce dernier, vous aurez systématiquement une note de zéro pour plagiat. Une telle situation suivra le protocole de discipline de l'ÉTS prévu à cet effet.

Rapport

On vous demande de créer un rapport répondant à ces questions. Votre rapport vaut 5% du projet.

Ce rapport a pour objectif de présenter votre travail. Cet exercice est pertinent autant pour vous afin de mettre en pratique votre capacité à résumer un travail technique que pour le correcteur qui pourra mieux comprendre votre cheminement de pensée.

Pour toutes vos réponses, on vous demande d'utiliser un vocabulaire technique, d'être concis et précis.

1. Mettre, au début du fichier : 0.5 point
   a. le numéro et le nom du cours
   b. le titre du projet
   c. les noms des membres de l'équipe (avec matricule)
   d. la date
2. Résolution du problème d'optimisation géométrique
   (soyez technique, concis et précis) : 1 point
   a. [0.5 points] Présentez et justifiez:
      i. du domaine
      ii. de la fonction objective
   b. [0.5 point] Produisez un diagramme de classe présentant votre solution.

3. Résolution du problème à sujet ouvert (soyez technique, concis et précis) : 3.5 points
    a. [1 point] Faites une présentation technique du problème à résoudre. On désire comprendre l'objectif du problème, les intrants et les extrants.
    b. [1 point] Présentez et justifiez:
        i. du domaine
        ii. de la fonction objective
    c. [1.5 point] Produisez un diagramme des cas d'utilisation et un diagramme de classes `UML` présentant votre solution.
4. Tout commentaire constructif sur le projet est apprécié : commentaires, coquilles et suggestions.

Vous devez produire ce rapport dans deux fichiers :

- `rapport.txt` présentant les éléments textuels demandés
- `uml.mdj` présentant les diagrammes `UML` demandés (format du logiciel `StarUML`)

Ces documents sont **obligatoires** pour que votre travail soit corrigé.

Quiz de laboratoire

Un quiz de laboratoire permettra d'évaluer la qualité de votre compréhension au travail que vous aurez effectué.

Ce quiz aura lieu à la dernière période de laboratoire de la session. La présence en classe est obligatoire pour faire le quiz. Attention, votre projet pourra être remis après pendant la semaine d'examen, toutefois, le quiz couvre l'entièreté du laboratoire. Donc, assurez-vous de l'avoir avancé suffisamment pour bien vous préparer.

Ce quiz vaut pour 25% du projet!

Tous les éléments couverts dans ce laboratoire sont sujet dans le quiz.

Pénalités

Les pénalités suivantes s'appliquent selon votre situation :

- Retard 10% par jours de retard
- Remise inadéquate problèmes mineurs 5.0%, problèmes majeurs 10.0%
- Présence de « warning » lors de la compilation et de l'édition de lien (ceux venant de votre code)
  2.5% par « warning » de catégorie différente, jusqu'à 15%
- Qualité du français. jusqu'à 10%

Remise

La remise se fait sur Moodle :

- pendant la semaine des examens finaux, la date définitive vous sera donnée par l'enseignant;
- une seule remise est attendue par équipe;
- Le document ZIP contenant votre travail :

- nom du document ZIP : `GPA434_Lab3_NomPrenomEtudiant1_NomPrenomEtudiant2_.zip`
- le document ZIP doit contenir votre solution **fonctionnelle**
- **assurez-vous de faire le nettoyage** de votre solution dans Visual Studio avant la remise.

Notes complémentaires

Pour les étudiants intéressés à pousser des concepts de développement logiciel, ce projet exploite directement ou indirectement divers patrons de conception. Certains sont bien ciblés et très utilisés alors que d'autres font quelques apparitions épisodiques.

- composite : à travers `Qt` pour ses assemblages de *widgets*.
- observateur : à travers les signaux et connecteurs de `Qt`.
- adaptateur : avec la classe `QDEAdapter`.
- stratégie : avec les nombreuses classes dont le nom termine par `Strategy`.
- prototype : avec la fonction `clone` dans les différentes stratégies.
- fabrique : avec la fonction `QDESolutionPanel::buildSolution`.
- Itérateur : dans la bibliothèque `delib` et selon votre façon de parcourir les données.
- ...