INTRODUCTION AU LANGAGE NETLOGO ET À LA SIMULATION À BASE D'AGENTS

Francisco Quesada Chaverri

Traduit de l'espagnol par Alain Albert

Février 2021

Table des matières

Préface du traducteur[1]

Cette version française du livre de Francisco Quesada Chaverri reflète tant mon intérêt pour les simulations multi-agents et le langage NetLogo que mon désir de mettre ce livre à la disposition des lecteurs de langue française.

En ce qui concerne mon intérêt pour le langage NetLogo il remonte à 2004, année au cours de laquelle j'ai eu l'occasion de suivre une formation dispensée par l'équipe d'Uri Wilensky (le concepteur de NetLogo) au Center for Connected Learning and Computer-Based Modeling de l'Université Northwestern aux États-Unis. Peu connu à cette époque, le langage NetLogo occupe aujourd'hui une place prépondérante au sein des langages de programmation multi-agents.

Pour ce qui est de mon désir de faire connaître ce livre aux lecteurs de langue française, il tient au fait que l'ouvrage se démarque des autres ouvrages du même genre sur deux points particuliers : sa démarche pédagogique et ses domaines d'application.

Tout d'abord, comme le souligne l'auteur, il s'agit d'un ouvrage qui, bien qu'exclusivement consacré au langage NetLogo, utilise ce langage pour initier les lecteurs aux principaux concepts communs à tous les langages de programmation[2]. Pour faciliter la compréhension et la maîtrise de ces concepts, l'auteur utilise des méthodes pédagogiques simples et efficaces. Ainsi les trois premiers chapitres contiennent des exemples simples de programmes qui illustrent les principaux concepts de programmation de base. Ces exemples de programmes facilitent la compréhension de la construction des modèles de simulations multi-agents exposés à partir du chapitre 4. Dans les autres chapitres du livre, la présentation alternée d'exemples de programmes et de modèles de simulation favorise l'apprentissage de NetLogo et permet au lecteur de bâtir ses propres modèles.

Au-delà de ces aspects pédagogiques, l'intérêt de l'ouvrage réside dans le type de modèles utilisés pour illustrer les possibilités d'application de NetLogo à des domaines différents de ceux que l'on trouve généralement dans les ouvrages consacrés à ce langage. Ainsi, plutôt que de reprendre les modèles traditionnels de simulations multi-agents en biologie, écologie, épidémiologie ou sciences sociales, l'auteur présente toute une gamme de modèles originaux qui touchent aussi bien des cas concrets de la vie quotidienne (organisation d'évènements, campagne de vaccination, trafic automobile en milieu urbain) que les domaines abstraits et théoriques de

[1] Le traducteur a été professeur d'économie et de gestion de projet (1975-2008) à l'Université du Québec en Outaouais (UQO) au Canada. Il est maintenant professeur honoraire (retraité) de cette institution.

[2] Tels les concepts de variable, expression conditionnelle, liste, récursivité, itération, etc.

l'arithmétique (nombres premiers), de la géométrie (calcul de superficies) et de la physique des particules.

De par leur originalité, ces types de modèles sont de nature à encourager le lecteur dans l'exploration des applications du langage NetLogo à ses propres champs d'expertise et d'intérêt. C'est à cet effort d'imagination créatrice qu'est convié le lecteur tout au long de son «parcours initiatique» au langage NetLogo.

Introduction

Qu'est-ce que NetLogo?

Le Manuel de l'utilisateur de NetLogo commence ainsi:
«*NetLogo est un environnement de programmation au sein duquel il est possible de simuler des phénomènes sociaux et naturels*». Un peu plus loin, les auteurs du manuel ajoutent que «*NetLogo est particulièrement adapté à la modélisation de systèmes complexes qui évoluent dans le temps. Les modélisateurs peuvent donner des instructions à des centaines, voire des milliers d'agents, qui fonctionnent indépendamment. Cela permet d'explorer le lien entre le comportement des individus au niveau micro et les configurations macro qui émergent des interactions entre lesdits agents.*»

À la lecture de cette brève description, on constate que NetLogo n'est pas simplement un langage de programmation. C'est en fait un environnement de calcul dont le noyau central est un langage de programmation mis au point pour faciliter la création et la simulation de modèles basés sur une multitude d'agents, une discipline connue sous le nom de modélisation et simulation multi-agents (Agent Based Modeling and Simulation en anglais)[3]. NetLogo a été créé en 1999 par Uri Wilensky et depuis lors, il est en développement continu, toujours sous sa direction, au Center for Connected Learning and Computer-Based Modeling (CCL) de l'Université Northwestern aux États-Unis. NetLogo est un produit entièrement gratuit, très bien documenté - bien que la documentation soit principalement en anglais – et qui bénéficie d'une communauté d'utilisateurs très active composée d'enseignants et de chercheurs de différents domaines de la science (physique, biologie, sciences sociales, etc.) et de l'ingénierie et d'amateurs de la science et de l'art de la programmation.

NetLogo est disponible pour les trois plates-formes: Windows, OS et Linux. L'un des avantages de NetLogo est le grand nombre de modèles disponibles, dont beaucoup sont inclus dans le logiciel lui-même ou sur le Web, ce qui est très utile pour apprendre le langage. L'environnement NetLogo comprend plusieurs modules ou extensions qui étendent ses possibilités

[3] Dans ce livre nous utiliserons l'abréviation française la plus courante «SMA» (pour simulation multi-agents). En anglais l'abréviation la plus courante est «ABM» (pour Agent-Based Modeling).

dans plusieurs directions. Il suffit de jeter un coup d'œil sur les différentes sections de la page d'accueil de Netlogo :
(https://ccl.northwestern.edu/netlogo)
pour apprécier l'immense variété d'articles, de livres, de modèles et de groupes d'intérêt qui touchent à cet environnement de programmation.

Quiconque aspire à s'aventurer dans le domaine de la modélisation et de la simulation multi-agents, soit pour construire ses propres modèles, soit pour comprendre le code de modèles construits par d'autres, doit commencer par apprendre les bases du langage de programmation dans lequel ces modèles sont écrits. Dans ce livre d'introduction au logiciel de SMA NetLogo, le langage occupe la première place. En particulier, on suppose que les lecteurs n'ont aucune connaissance en programmation. C'est pourquoi des explications générales sur les principaux concepts communs à *tous* les langages de programmation ont été inclus dans le livre. Parmi ces concepts figurent (entre autres) : les variables, les expressions conditionnelles, la récursion, l'itération, les listes, les chaînes, etc.

Le principal souhait de l'auteur est que les lecteurs qui auront achevé la lecture de ce livre aient acquis une connaissance suffisante du langage et des principales ressources de l'environnement NetLogo pour leur permettre de s'aventurer dans la création de leurs propres programmes et modèles. À partir de ce moment, il devrait être relativement facile aux lecteurs de développer et d'approfondir la connaissance de ce riche environnement, grâce aux nombreux exemples inclus dans le programme et aux ressources disponibles sur le site Web de NetLogo. Une version francaise du code des exemples et des modèles du présent livre est discuté en détail et peut être téléchargé à partir du site de l'auteur: http://www.franciscoquesada.com/.

Le livre est structuré comme suit: Les trois premiers chapitres présentent, au moyen d'exemples, un ensemble de primitives NetLogo et les concepts de base de la programmation, qui vous permettront de créer rapidement vos premiers programmes et modèles. Les chapitres 4 et 5 contiennent une collection de modèles, dont les codes utilisent les primitives et les concepts présentés dans les chapitres précédents, ainsi que des primitives et des concepts nouveaux nécessaires à la mise en œuvre de ces modèles. Le chapitre 6 présente quelques thèmes qui complètent l'exposition du langage et de l'interface NetLogo. Dans le chapitre 7 on introduit le concept de récursivité qui est ensuite illustré à l'aide de quelques exemples. Le huitième et dernier chapitre montre comment NetLogo peut faire l'objet d'applications géographiques simples[4] en combinant ce logiciel avec Google

[4] Les applications géographiques plus complexes de NetLogo nécessitent l'utilisation de l'extension NetLogo Gis (GIS est l'abbréviation de «Geographical

Earth. L'annexe A décrit brièvement certains aspects de NetLogo non inclus dans le livre. L'annexe B contient la liste des primitives (mots-clés) utilisées dans le livre.

NetLogo en français

Au moment de l'achèvement de la traduction de ce livre[5], il n'existe aucune version française de NetLogo : tant le langage utilisé pour écrire le code que l'interface qui affiche les noms des menus, des fenêtres et des boutons sont en anglais[6]. Il est impossible de prévoir si une version française de NetLogo sera disponible dans un proche avenir mais le niveau d'anglais requis pour pouvoir utiliser NetLogo est un anglais de base qui ne devrait pas inquiéter le lecteur[7]. Le fait qu'il n'existe pas de version française de NetLogo présente même des avantages: cela permet en effet de renforcer les liens avec la communauté internationale des utilisateurs de NetLogo dont les programmes sont en grande partie présentés en anglais. Ce qui est important pour la communauté francophone, c'est qu'il existe une documentation en français: manuels, tutoriels, livres et exemples commentés en français. Grâce à la version française de ce modeste tutoriel, l'auteur et le traducteur espèrent apporter une petite contribution dans ce sens.

Quelques observations sur la traduction en français

Bien que le jargon de la haute technologie s'inspire souvent de la langue anglaise, de nombreux efforts ont été faits pour éviter l'emploi abusif d'anglicismes dans les ouvrages sur l'informatique. Cependant tous les langages de programmation ont leurs spécificités et NetLogo n'échappe pas à cette règle. Ainsi, par exemple, dans le langage NetLogo la primitive «breed» est utilisée pour construire des catégories (ou classes) d'agents ayant des caractéristiques et/ou comportements spécifiques. La traduction littérale de breed est «race» mais comme, de nos jours, ce mot s'applique essentiellement aux animaux («a breed of dogs», une race de chiens) nous

---

Information Systems» - Systèmes d'information géographique), une extension qui est mentionnée dans l'Annexe A du présent ouvrage

[5] Février 2021

[6] La version espagnole (version originale) du présent livre utilise une interface en espagnol mais la langue utilisée pour codifier les modèles reste l'anglais.

[7] Ainsi, par exemple, une commande typique de NetLogo est du genre «show count turtles with [color = red], commande dont la traduction («afficher le nombre de tortues avec [couleur = rouge]») n'exige pas une connaissance approfondie de l'anglais

avons choisi de traduire «breed» par «famille» afin que ce terme puisse s'appliquer à de nombreuses catégories d'agents ou de liens entre agents[8]. Un autre exemple est celui de la catégorie de primitives appelées «reporter». Dans ce cas nous avons décidé d'utiliser le terme français «rapporteur» (et non «reporter») en écrivant, par exemple, que le rapporteur «mean» calcule la moyenne d'une série de nombres, moyenne qui est «rapportée» comme entrée («input» en anglais) pour utilisation dans une ou plusieurs partie du programme[9]. On pourrait multiplier ainsi les exemples mais nous arrêtons la liste ici et invitons le lecteur à consulter l'Annexe B pour plus d'explications.

Sources d'information supplémentaires.

Dans ce livre, nous ferons fréquemment référence à plusieurs sources d'informations d'une grande importance pour l'apprentissage NetLogo. La première, et sans doute la plus importante, est le Manuel de l'utilisateur «NetLogo 6.2.0 User Manual» [22], qui peut être téléchargé gratuitement sur le site Web Netlogo (https://ccl.northwestern.edu/netlogo/docs/) et dont il n'existe malheureusement pas de version française[10]. Nous recommandons aux lecteurs de télécharger la version pdf de ce document et de l'imprimer à titre de référence. Ce guide, que nous appellerons «le Manuel», contient non seulement la liste complète des primitives et de nombreux exemples importants, mais il comprend également une description de tous les aspects de NetLogo, y compris de ses extensions. L'autre source d'information à laquelle nous ferons fréquemment référence est l'excellent livre en anglais de Wilensky et Rand [21]. En plus d'enseigner comment programmer des modèles avec NetLogo, en commençant par les bases, ce livre est une source d'informations précieuses sur de nombreux autres aspects de NetLogo et sur la modélisation et la simulation multi-agents en général. Et bien sûr, nous ne pouvons pas manquer de mentionner comme autre source précieuse d'informations et

[8] Comme on le verra ultérieurement le mot «agent» a, dans NetLogo, une connotation différente de celle qui caractérise les autres langages de programmation multi-agents (ainsi, par exemple un lien entre deux agents est aussi un «agent» dans le langage NetLogo)

[9] Dans NetLogo un **rapporteur («reporter»)** calcule un résultat qu'il communique («retourne») à celui qui l'a appelé. Dans certains autres langages de programmation le terme utilisé est **fonction** (plutot que rapporteur).

[10] Il existe cependant une version française du Manuel de l'utilisateur NetLogo 4.0.4 mais celle-ci date quelque peu (2009). Cette version peut cependant être utile (à condition de tenir compte des changements intervenus entre la version 4.0.4 et la version actuelle (Version 6.2.0, Décembre 2020) et peut être consultée sur le site Web http://romainmejean.fr/manuel_netlogo/ (site consulté en Novembre 2020)

d'apprentissage les nombreux exemples disponibles dans la bibliothèque de modèles de NetLogo :
(https://ccl.northwestern.edu/netlogo/models/index.cgi).
Finalement, bien que d'un niveau légèrement plus élevé que celui de Wilensky et Rand, l'ouvrage de Railsback et Grimm [17] reste tout de même très accessible aux débutants et contient de nombreux exemples qui complètent ceux des ouvrages précédemment cités.

Les lecteurs francophones sont, quant à eux, invités à consulter les deux volumes de Banos, Lang et Marilleau sur la simulation spatiale à base d'agents avec NetLogo. Le volume 1 [3] est une introduction à NetLogo tandis que le volume 2 [4] porte sur des notions plus avancées de ce langage.

La langue de la version originale de ce livre (Quesada [16]) étant l'espagnol, les lecteurs qui lisent l'espagnol peuvent évidemment consulter cette version originale sur le site (déjà mentionné) de l'auteur. Il existe également une traduction en espagnol du dictionnaire des primitives de NetLogo. Cette version est, pour le moment, uniquement disponible en format pdf et peut être consultée ou téléchargée sur le site Web de Netlogo ou sur le site personnel de l'auteur. Outre le livre de Quesada, nous recommandons également l'excellent et très complet ouvrage des auteurs García Vazquez et Sancho Caparrini [8], qui existe dans les deux versions espagnole et anglaise.

Finalement, nous recommandons vivement aux personnes qui désirent apprendre à programmer avec NetLogo de télécharger et d'installer le programme NetLogo sur leur ordinateur et de télécharger ou de numériser les exemples du livre. Nous vous invitons également à expérimenter avec vos propres exemples de commandes ou de procédures. Netlogo peut être téléchargé sur le site officiel (page d'accueil):
https://ccl.northwestern.edu/netlogo/

Comment exécuter les exemples de ce livre et un problème concernant l'utilisation des guillemets[11].

La méthode consistant à copier le code des exemples ou des modèles du livre et à le coller (style «copier-coller») dans l'éditeur de code NetLogo n'est pas la manière la plus efficace d'importer les exemples. Il est préférable de copier

[11] L'écriture de certaines parties d'un code NetLogo nécessite l'utilisation de guillemets anglais (" ") et non celui de guillemets français (« »).

les exécutables (fichiers avec l'extension «.nlogo») et de les télécharger depuis NetLogo. Un dossier avec les modèles compressés (version française) est disponible sur le site de l'auteur. La méthode «copier le code du livre et le coller dans l'éditeur» soulève deux types de problèmes. Premièrement, la plupart des exemples et modèles présentés dans le livre requièrent la construction manuelle d'éléments (boutons, curseurs, graphiques) dans l'interface pour que le modèle s'exécute. Il n'est donc pas possible d'utiliser la méthode «copier-coller» pour copier ces éléments dans l'interface du programme. En deuxième lieu, l'opération qui consiste à copier le code publié dans le livre puis à le coller dans l'éditeur du programme NetLogo provoque parfois l'apparition d'un message d'erreur indiquant que NetLogo ne reconnaît pas le type de guillemets importés comme valide[12]. Bien que tout ait été fait dans le livre pour éviter ce probléme, la seule méthode fiable est, comme nous le suggérons plus haut, de copier les exécutables (version française) publiés sur le site de l'auteur[13].

[12] Cela n'arrive pas toujours et il est possible que cela ait à voir avec le type de police utilisé.

[13] En effet le programme ne sera pas exécuté si son code contient des guillemets non reconnus par NetLogo.

Chapitre 1: Environnement et langage I.

Quand on ouvre le logiciel NetLogo, la première chose qui apparaît sur l'écran de l'ordinateur est une fenêtre, qui constitue l'interface avec l'utilisateur. La figure ci-dessous illustre cette interface appelée «interface NetLogo».

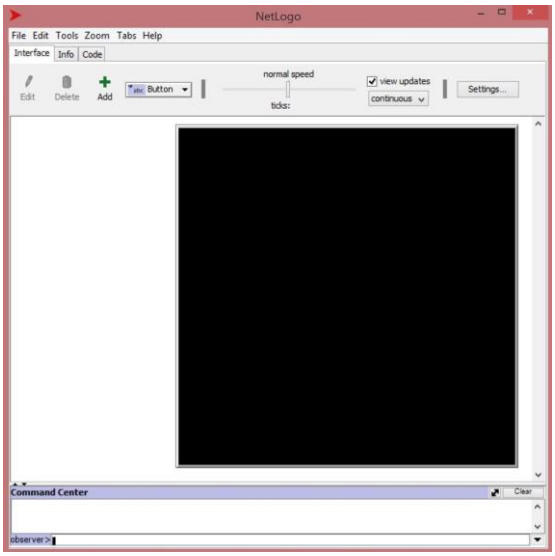


Figure 1.1 : L'interface NetLogo

Cette interface est composée d'un ensemble d'éléments, tels que des menus, des fenêtres et des boutons, destinés à faciliter la création des modèles. L'interface et le langage de programmation sous-jacent forment ce que l'on appelle un environnement de programmation. Les trois principaux éléments de l'environnement NetLogo sont :

• L'interface: c'est la fenêtre qui permet au programmeur de communiquer avec NetLogo pour construire les modèles et les exécuter.
• Le langage: il est constitué des mots et des constructions grammaticales avec lesquels les programmes NetLogo sont construits.
• Les agents: Ce sont les entités qui exécutent les actions du programme.

L'environnement NetLogo a été spécialement conçu pour faciliter la simulation de modèles multi-agents, ce qui explique la composition de l'interface NetLogo ainsi que les caractéristiques du langage. Il est à noter cependant que, malgré cette vocation spécifique de NetLogo, il est également possible de créer des programmes de nature plus diverse qui ne sont pas nécessairement à base d'agents. L'élément le plus remarquable de l'interface est le carré noir situé à droite, appelé le monde. C'est dans ce «décor» que les agents exécutent la majorité de leurs actions, comme le font des acteurs au cours d'une scène de théâtre. Les programmes construits avec NetLogo ne sont pas des programmes

autonomes («stand alone programs» en anglais), c'est-à-dire qu'ils ne s'exécutent pas si le logiciel NetLogo n'est pas présent dans l'ordinateur.

Les auteurs de NetLogo ont créé une version en ligne, qui permet aux programmes NetLogo d'être exécutés sans installer le logiciel. On peut accéder à cette version en ligne en se connectant à https://netlogoweb.org/.

Le langage

Le langage NetLogo est au cœur de l'environnement NetLogo et constitue sans aucun doute sa partie la plus importante. Un programme d'ordinateur n'est rien d'autre qu'une séquence logique de commandes, formulée dans un langage technique, qui permet à l'ordinateur d'exécuter une série de tâches proposées par le programmeur. Pour éviter d'avoir à écrire les commandes dans le langage de l'ordinateur - le «langage machine» composé d'une suite numérique binaire de zéros et de uns («bits») - des langages de programmation de haut niveau ont été créés. NetLogo est l'un de ces langages et, comme tout langage de programmation il a ses propres règles de syntaxe[14], qui doivent être strictement observées. Heureusement, la syntaxe de NetLogo est assez simple. Le langage Netlogo est du type dit «langage interprété», ce qui signifie que chaque commande d'un programme est prise séparément, traduite en langage machine et exécutée immédiatement par l'ordinateur.

Le programme qui exécute ce processus est appelé «l'interprète (ou l'interpréteur) de NetLogo» et nous le désignons indifféremment comme «l'interprète» ou «l'interpréteur». Le fonctionnement d'un langage interprété est différent de celui d'un langage dit «compilé». Dans un langage compilé, le texte constituant le programme, appelé «code source», est pris dans son intégralité et converti en un programme autonome appelé «code objet». Ce code objet étant écrit en langage machine, il peut être exécuté par n'importe quelle machine[15], sans la présence du langage de programmation dans lequel le code source a été écrit et compilé. Les langages interprétés offrent deux avantages très importants par rapport aux langages compilés: 1) il est possible de tester des commandes sans qu'elles fassent partie d'un programme. Cela permet de connaître l'effet de commandes isolées, ainsi que de vérifier l'exactitude de leur syntaxe et 2) il est possible, avec un langage interprété, d'écrire et de tester les programmes de manière incrémentale, c'est-à-dire d'observer les résultats du programme au fur et à mesure que l'on ajoute de

[14] Nous ne ferons pas ici la différence entre les concepts de «syntaxe» et de «grammaire» d'un langage

[15] Possédant le même système d'opération (par exemple Windows, OS ou Linux).

nouvelles commandes. Cette manière de procéder facilite la détection précoce des erreurs de programmation.

Cependant, les langages compilés ont aussi certains avantages par rapport aux langages interprétés : 1) leur vitesse d'exécution est plus rapide et 2) leur «portabilité» est plus grande car ils ne requièrent pas la présence du logiciel de compilation dans l'ordinateur qui exécute le code objet. Dans ce livre, nous essayons de maintenir un certain niveau de cohérence dans l'utilisation des mots «modèle», «programme», «procédure» et «code». Nous appellerons «modèle» informatique une représentation simplifiée d'un phénomène réel ou fictif en vue de réaliser un traitement (simulation par exemple) par ordinateur. Un «programme» informatique est une succession logique d'instructions (de commandes) exécutables par l'ordinateur. Avec NetLogo l'exécution de ces instructions permet, par exemple, de quantifier et de visualiser les résultats d'un modèle de simulation. Il est à noter cependant que tous les programmes informatiques n'ont pas pour objectif de modéliser des phénomènes réels ou fictifs. Le mot «procédure» est réservé, la plupart du temps, pour désigner les petites unités de code qui constituent un programme plus large. Enfin, nous utilisons le mot «code» - un mot souvent utilisé en programmation - pour désigner soit le texte intégral d'un programme ou d'une procédure, soit un extrait dudit texte.

La syntaxe du langage NetLogo est très similaire à celle du langage Logo, que l'on peut qualifier de proche parent. Les deux langues ont été inspirées par Lisp, l'un des plus anciens et plus puissantes langages de programmation existants. L'apprentissage du vocabulaire de base et des règles de syntaxe les plus importantes d'un langage de programmation s'acquiert par la pratique c'est-à-dire par la construction de programmes. L'interprète NetLogo a la capacité de détecter la plupart des erreurs d'orthographe ou de grammaire du langage. Pour simplifier, toutes ces erreurs seront appelées «erreurs de syntaxe» et la nature de ces erreurs est signalée dans des messages d'erreur.
Outre les erreurs de syntaxe, il existe deux autres types d'erreurs : les «erreurs d'exécution» («runtime errors» en anglais) et les «erreurs logiques» («logical errors» en anglais»). Les «erreurs d'exécution», se produisent pendant l'exécution et causent l'arrêt («crash») du programme en cours d'exécution. Ces erreurs peuvent être dues à des défaillances matérielles (mémoire insuffisante) mais aussi à des erreurs de programmation qui ne sont pas nécessairement des erreurs logiques comme par exemple diviser un nombre x par une variable aléatoire y qui peut prendre la valeur 0 ou bien vouloir utiliser les valeurs de données qui sont en dehors des limites de la plage autorisée pour ces données («data value out of range» en anglais). Le troisième type d'erreurs, les erreurs

logiques se produisent lorsque la conception logique du programme est incorrecte. Ces erreurs ne provoquent pas l'arrêt du programme, mais produisent des résultats différents de ceux prévus. Des trois types d'erreurs, les erreurs logiques sont les plus difficiles à détecter car les utilisateurs ne reçoivent aucune alerte en dehors de l'obtention de résultats inattendus ou, pire encore, reçoivent des résultats très similaires à ceux attendus et qui peuvent rester inaperçus pendant longtemps.

Primitives: le vocabulaire de base. Chaque langage de programmation a son propre vocabulaire de base, avec lequel les commandes ou instructions qui composent les programmes sont écrits. Dans ce livre, les mots «commande» et «instruction» ont la même signification. Les mots de base d'un langage de programmation sont désignés par le nom de «mots primitifs» ou, plus simplement, «primitives» du langage. NetLogo possède un grand nombre de primitives, qui permettent de représenter une grande variété de situations. Le dictionnaire des primitives de NetLogo, disponible sur le site NetLogo, contient une description du fonctionnement de chaque primitive du langage.

Les agents de NetLogo.

NetLogo dispose d'un ensemble d'agents préinstallés dans le système, prêts à être utilisés pour écrire les programmes. Les agents NetLogo peuvent être regroupés en quatre catégories:

- Des *agents mobiles* appelés tortues («turtles»)[16]. Ces agents peuvent se déplacer dans le monde (carré noir de l'interface déjà mentionné). La forme initiale des tortues est celle de petits triangles. Cette forme peut être modifiée à l'aide d'une commande qui permet de donner une forme spécifique (personnes, voitures, etc.) aux agents (voir le Guide d'édition des formes de NetLogo). La figure 1.2 illustre l'interface NetLogo avec cinq tortues de différente formes et couleurs aléatoirement éparpillées dans le monde.

[16] La coutume d'appeler ces agents mobiles des tortues provient du langage Logo, qui, comme nous l'avons déjà mentionné, est un parent proche de NetLogo.

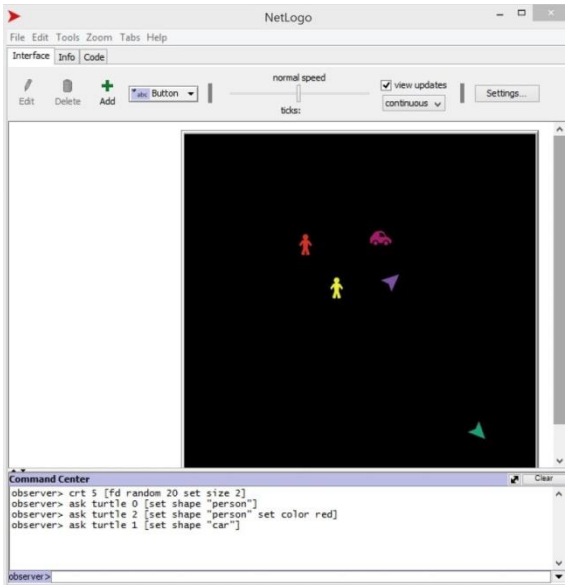


Figure 1.2 : Répartition aléatoire de 5 agents dans le monde

Il est possible de passer une commande aux tortues pour qu'elles laissent une trace au cours de leurs déplacements. Cette possibilité transforme les tortues en outils graphiques polyvalents.

• Des *agents statiques* appelés parcelles («patches»). Les parcelles forment une grille qui couvre le monde, comme si ce dernier était composé de petits carrés de céramique. La couleur noire attribuée par défaut à ces parcelles fait en sorte qu'elles ne peuvent être initialement distinguées les unes des autres, ce qui explique la couleur entièrement noire du monde au démarrage de NetLogo. Le nombre de parcelles, leur taille et leur couleur peuvent être modifiés. Lorsque le programme s'ouvre, le monde est représenté par une grille de 33 x 33 = 1089 parcelles de forme carrée. La figure 1.3 illustre un monde composé de parcelles de couleur différente générées aléatoirement.

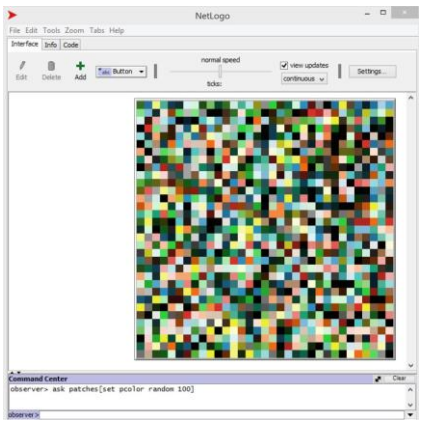


Figure 1.3 : Le monde coloré des parcelles du monde de NetLogo

• Des agents appelés *liens* («links» en anglais). Ces agents servent à «relier» deux ou plusieurs tortues entre elles. Lorsqu'un lien est établi, il devient visible sous la forme d'une ligne joignant les deux tortues. L'apparence de cette ligne peut être modifiée ou même cachée. Les liens peuvent être orientés ou non. Lorsqu'un lien est orienté, la tortue d'où le lien part est appelée «origine» ou «racine» du lien, tandis que la tortue où le lien se termine est appelée «destination», «but» ou «feuille» du lien. La figure 1.4 illustre un lien non orienté entre deux tortues.

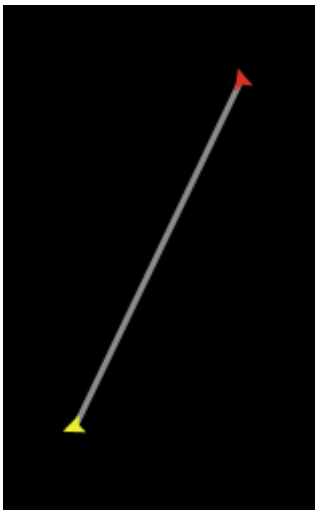


Figure 1.4 : Exemple de lien non orienté entre deux tortues

• Le quatrième agent est l'observateur («observer» en anglais). Cet agent, qui n'est pas visible, est l'agent situé au sommet de la hiérarchie. L'observateur peut donner des ordres aux autres agents. On communique avec lui par le biais de la «fenêtre de l'observateur» située tout en bas de l'interface.
Tous les agents sont programmables: l'essence de la programmation multi-agents réside précisément dans le fait que les agents sont des entités autonomes, capables de recevoir et d'exécuter des commandes individuellement.

Ensembles d'agents (ensemble-agents)

NetLogo permet de regrouper des agents dans des ensembles et de les traiter comme des entités auxquelles on peut donner des instructions. Ces ensembles sont appelés «ensemble-agents» («agentsets» en anglais). Il est possible de définir des ensemble-agents composés de tortues, de parcelles ou de liens, mais ils doivent toujours être du même type. On peut créer des ensemble-agents en imposant des restrictions à un ensemble-agent que l'on aurait

précédemment créé. Par exemple, après avoir créé un ensemble-agents composé de nombreuses tortues, on peut définir un nouvel ensemble-agents composé uniquement des tortues de couleur rouge (l'ensemble de tortues rouges est un sous-ensemble de l'ensemble initial). Comme nous le verrons dans certains des exemples utilisés dans ce livre, les ensemble-agents ne sont pas seulement une abstraction du langage ordinaire, mais sont aussi des entités possédant leur propre existence informatique dans l'environnement NetLogo, ce qui peut être très utile pour la construction de modèles.

Le monde

Nous énumérons et décrivons brièvement ci-après quelques-unes des caractéristiques les plus importantes du monde:

- À l'ouverture de NetLogo, le monde est composé d'un ensemble de petites parcelles carrées qui forment une grille.
- Chaque parcelle de cette grille est un agent programmable.
- Sur la grille, chaque parcelle est identifiée par une paire de coordonnées cartésiennes (x, y). L'origine des coordonnées (0, 0) est située au centre du monde, mais il est possible d'en changer d'emplacement.
- Les coordonnées d'une parcelle font référence à son point central et sont toujours des entiers (les tortues, par contre, peuvent occuper des positions dont les coordonnées ne sont pas nécessairement des nombres entiers).
- Les distances parcourues par les tortues sont mesurées en «pas» («steps» en anglais) et il existe une relation entre la longueur des pas d'une tortue et les dimensions des parcelles: un pas effectué par une tortue équivaut à la dimension du côté d'une parcelle, de sorte que, si une tortue est au centre d'une parcelle et avance, par exemple, d'un pas vers le haut, elle atteindra exactement le centre de la parcelle située juste au-dessus d'elle.
- Le nombre de parcelles qui composent le monde peut être modifié, de même que leur taille et leur forme.
- Dans sa configuration initiale, le monde se comporte comme si ses côtés opposés se rejoignaient: si une tortue quitte le monde du côté droit de la grille, nous la verrons réapparaître à la même hauteur du côté gauche et inversement. On peut constater également ce phénomène quand une tortue quitte la grille par le haut ou par le bas. Cette configuration topologique du monde peut, comme nous le verrons plus loin, être modifiée.

Les vues

L'aspect graphique du monde est constitué par ce que NetLogo appelle des «vues» («views» en anglais). Quand un programme est exécuté, l'apparence du monde varie, offrant des séquences de vues qui représentent l'évolution des phénomènes modélisés et donnent, à l'instar des images («frames») d'un film, la sensation de mouvement, Chacune de ces images constitue une vue, une image figée du monde à un moment donné. Cette image est celle d'un carré noir (ou d'un carré de la couleur que l'on a choisi de donner au monde) sur lequel apparaissent les parcelles, tortues et traces laissées par ces dernières. La vue n'inclut pas les autres éléments de l'interface qui sont hors du monde, tels les boutons, curseurs, menus graphiques ou fenêtres. Il est à noter que NetLogo permet d'opter soit pour une vue en 2 dimensions («2D view») soit pour une vue en 3 dimensions («3D view»)[17]. Les exemples traités dans ce livre ne concernent que les vues en 2D.

Où écrire le code NetLogo ?

Dans les langages de type interprété, comme NetLogo, il est possible de tester non seulement des programmes complets mais aussi des instructions isolées ou des séquences d'instructions qui n'ont pas la structure d'un programme complet. Dans NetLogo, ces deux situations sont gérées dans des zones différentes de l'interface. Les programmes NetLogo doivent être écrits dans la zone d'édition de programme (zone à laquelle on accède en cliquant sur l'onglet «Code»). Les séquences d'instructions qui ne constituent pas un programme complet doivent être écrites dans la «fenêtre de l'observateur» (fenêtre parfois appelée «console» dans d'autres langages).

La fenêtre de l'observateur

La fenêtre de l'observateur, zone allongée et étroite située tout au bas de l'interface et à la droite du mot «observer», est l'endroit où les instructions (commandes) isolées doivent être entrées. Une fois qu'une commande ou qu'une séquence de commandes a été écrite à cet endroit, il suffit d'appuyer sur la touche <Entrée> du clavier de l'ordinateur pour que la commande soit exécutée. Si la commande a été mal saisie, l'interpréteur NetLogo affichera un message d'erreur («Error») dans le terminal d'instructions, la fenêtre située immédiatement au-dessus de la fenêtre de l'observateur. Si vous avez déjà

[17] Voir la section «The 2D and 3D views» dans le guide de l'interface de Netlogo.

installé NetLogo sur votre ordinateur, nous vous invitons à entrer les premières commandes NetLogo dans cette fenêtre. Pour des raisons pédagogiques, les premiers exemples de commandes et de procédures donnés dans ce chapitre sont adressés aux agents.

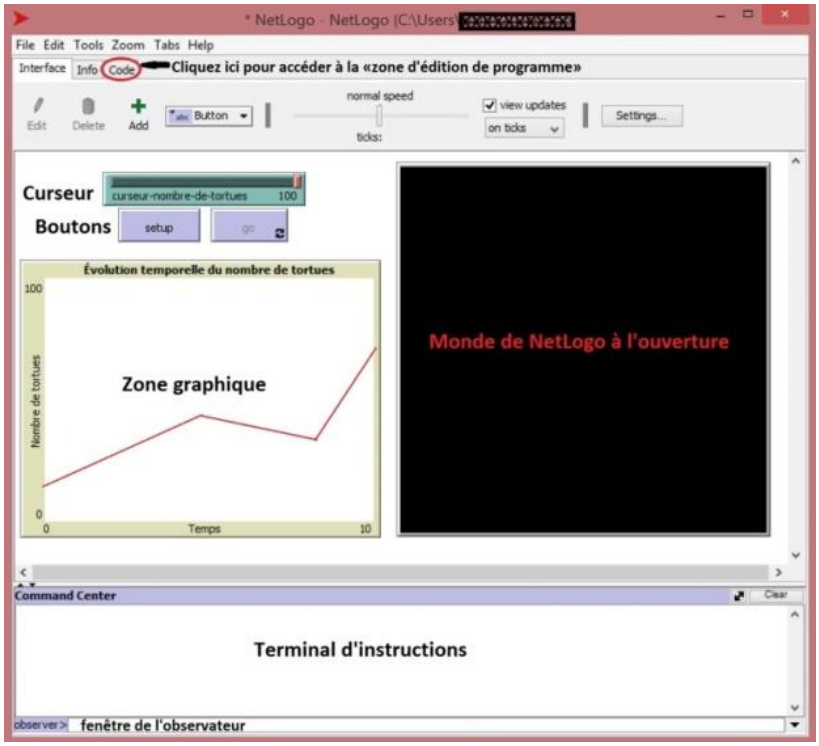


Figure 1.5 : Les composantes de l'interface NetLogo

Exemple 1 : Premières commandes dans la fenêtre de l'observateur

Primitives: create-turtles (créer-tortues), pendown (abaisse-stylet), forward (en avant, avancer), right (droite), ask (demander), turtles (tortues), + (signe d'addition), show (montrer), clear-all (effacer-tout).

Dans la fenêtre de l'observateur, tapez textuellement les commandes ci-dessous. Pour exécuter la commande, appuyez sur la touche <Entrée> de votre ordinateur après avoir écrit le texte de la commande.

1- Tapez:

create-turtles 2

Cette commande crée 2 tortues sur la parcelle (0 0) qui est le centre du monde et le lieu de naissance des tortues. Comme les tortues sont toutes créées au

même moment et sont sur la même parcelle, on a l'illusion qu'il n'en existe qu'une. Si vous commettez une erreur lors de l'écriture de la commande, un message d'erreur («ERROR») s'affiche dans le terminal d'instructions («Command Center») situé immédiatement au-dessus de la fenêtre de l'observateur. Si tel est le cas, lisez attentivement le message (son contenu dépend du type d'erreur qui a été faite) et exécutez à nouveau la commande après l'avoir réécrite correctement.

2. Tapez:

ask turtles [pendown forward 10]

La traduction littérale de cette commande est «demander tortues [abaisse stylet avance 10 pas]». Un résultat possible d'une telle commande est illustré sur la figure 1.6

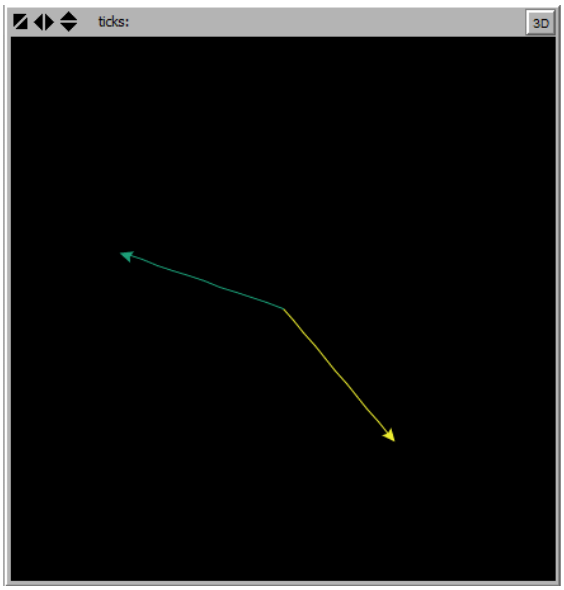


Figure 1.6 : Effet de l'utilisation de pendown

Lorsque les tortues abaissent leur stylet, elles laissent, en avançant, une trace de la même couleur que la leur. Les tortues doivent exécuter toutes les commandes inscrites à l'intérieur des crochets [ ] de l'expression «ask turtles». Les commandes sont séparées par des espaces, jamais par des virgules ou d'autres caractères. Les tortues naissent avec des couleurs et des orientations attribuées de manière aléatoire par le système NetLogo. On peut penser que c'est l'interprète qui assigne les valeurs de ces variables aux tortues.

3. Tapez:

ask turtles [show 3 + 3]
==> (turtle 0): 6

==> (turtle 1): 6

Cette commande demande aux tortues créées d'afficher le résultat de la somme 3 + 3. Nous allons utiliser le symbole « **==>** » pour indiquer les résultats affichés par NetLogo dans le terminal d'instructions («Command Center»), le lieu d'affichage par défaut du programme. Le résultat «6» est affiché deux fois car la commande s'adresse à toutes les tortues qui, pour l'instant, sont au nombre de 2.
Notez que lorsqu'une commande est écrite dans la fenêtre de l'observateur, une fois qu'elle est exécutée, le texte de la commande est répliqué (comme un écho ou une image miroir) à la droite du mot «observer» dans le terminal d'instructions. La primitive «show» montre le résultat de la somme, précédé du nom de l'agent qui a exécuté la commande. Il est possible de rediriger la sortie des résultats vers d'autres zones, par exemple vers une fenêtre de sortie ou vers un fichier externe, comme on le verra plus tard. Notez que les tortues sont numérotées au fur et à mesure de leur création, à partir du numéro 0. Un autre détail à prendre en compte est que dans NetLogo - comme dans le langage Logo - les espaces doivent être laissés de chaque côté des opérateurs arithmétiques binaires, comme les signes « + » ou « = » (l'omission de ces espaces génère un message d'erreur).

4. Tapez:

ask turtle 0 [right 90 forward 10]

Cette commande est seulement adressée à la tortue 0. Elle demande à cette dernière de pivoter de 90 degrés vers la droite sur son propre axe, puis d'avancer de 10 pas. Les commandes adressées à un agent ou à un groupe d'agents doivent être mises entre crochets, même s'il s'agit d'une commande unique ou d'un groupe d'agents constitué d'un seul membre, comme dans le présent exemple.

5. Tapez:

clear-all

La commande «clear-all» remet le monde à son état initial: elle élimine les tortues crées à l'aide des commandes antérieures ainsi que les traces laissées par ces tortues. La forme abrégée de cette commande est «ca».

6. Tapez:

show "Bonjour le monde!"
==> observer: "Bonjour le monde!"

Il est à noter que les commandes qui ne concernent pas les tortues, les parcelles ou les liens sont considérées comme étant adressées à l'observateur. Le texte à droite de la primitive «show» doit être placé entre guillemets anglais **" "**[18] . Si cette règle n'est pas respectée, l'interprète pensera que les mots dudit texte sont des primitives ou des noms de procédures NetLogo et essaiera de les évaluer comme tels. Un message d'erreur apparaitra alors. On rappelle que lorsque les tortues sont créées elles sont, sauf indication contraire, dotées d'une couleur et d'une orientation attribuées de manière aléatoire. L'orientation d'une tortue est la direction dans laquelle la tortue regarde et c'est la direction en ligne droite dans laquelle elle se déplacerait si on lui donnait l'ordre d'avancer (ou de reculer). L'orientation est déterminée au moyen d'un angle dont la valeur varie entre 0 et 360 degrés.

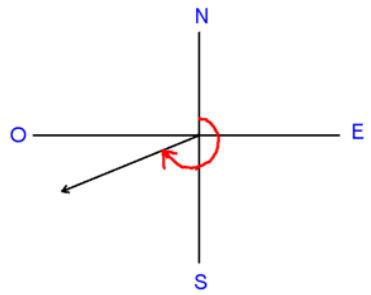


Figure 1.7 : Angle d'orientation des tortues

L'angle d'orientation des tortues est mesuré à partir du nord dans le sens des aiguilles d'une montre. La figure 1.7 montre un angle d'orientation situé dans le troisième quadrant et compris entre 180 ° et 270 °. Une orientation de valeur 0 correspond à la direction vers le haut de l'écran, appelée direction nord. Notez que cette convention diffère de celle couramment utilisée en mathématiques, où une orientation à 0 degré pointe vers la droite, dans la direction est. Certaines primitives ont des versions abrégées, telles que «forward», qui peut être abrégé en «fd» et «create-turtles» dont l'abréviation est «crt». Les commandes écrites dans la fenêtre de l'observateur sont stockées dans le centre de commande situé au-dessus de cette fenêtre et peuvent être réutilisées ultérieurement. Vous pouvez les retrouver en activant le curseur dans la fenêtre et en les recherchant à l'aide des touches de déplacement du

[18] Voir la note 7 de l'introduction pour une mise en garde concernant l'utilisation de la méthode «copier-coller» quand on utilise les guillemets anglais.

curseur. Il est aussi possible de supprimer toutes les commandes stockées en appuyant sur «clear» (à l'extrême droite du centre de commande)

L'éditeur de programmes

L'éditeur de programme (ou éditeur de code) est l'endroit où est écrit le code des programmes NetLogo. Une pratique courante en programmation est de structurer les programmes en les divisant en programmes plus petits appelés procédures. Dans l'éditeur de code, vous ne devez pas écrire des commandes ni des séquences de commandes isolées, c'est à dire qui ne font partie d'une procédure. L'éditeur de code s'ouvre en cliquant sur l'onglet portant le nom «Code» situé en haut de l'interface. Ce faisant, nous observons un changement dans l'apparence de l'interface: le monde est remplacé par une zone vide dans laquelle le code des procédures est écrit, comme indiqué sur la figure 1.8.

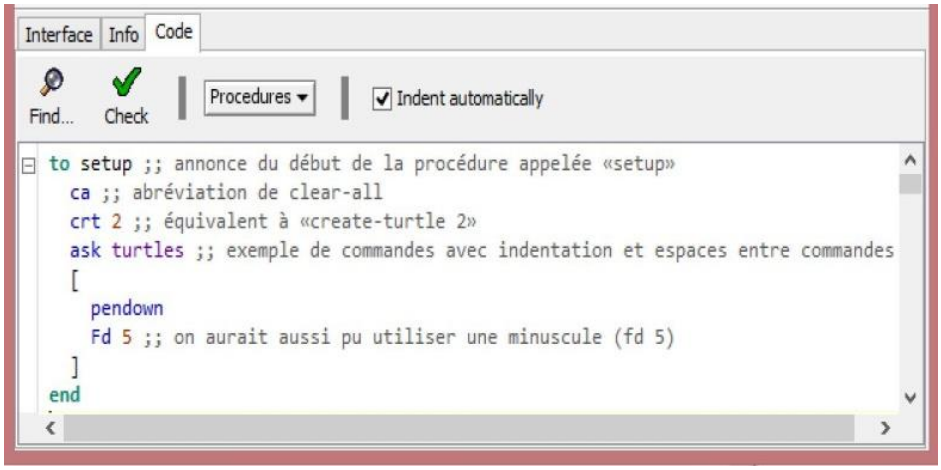


Figure 1.8 : Exemple de procédure d'initialisation du programme

Chaque procédure NetLogo est une séquence de commandes «empaquetées» sous un nom, à savoir le nom attribué à la procédure. Dans NetLogo, le format type de toute procédure est le suivant:

- La première ligne du code d'une procédure ne doit être occupée que par le nom de la procédure, précédé de la primitive «to» (que l'on peut traduire par «à» ou encore par «vers»).
- Dans les lignes qui suivent le nom de la procédure, on écrit le «corps» de la procédure, c'est-à-dire les commandes qui la composent. Ces commandes doivent être écrites l'une à la suite de l'autre et séparées seulement par des espaces blancs (aucun autre caractère de séparation tels que virgules, point-virgules, etc. n'est permis). Il est possible de distribuer les commandes sur autant de lignes que l'on veut car l'interprète est insensible aux espaces blancs et aux changements forcés

de ligne). L'interprète ne fait pas non plus de distinction entre majuscules et minuscules

Il est possible d'indenter librement le texte à l'aide de la touche de tabulation. Toutefois, afin de rendre la structure du programme plus compréhensible pour les lecteurs, il est conseillé de suivre certaines règles sur la façon d'indenter et de séparer le code en lignes. On trouvera des exemples de structures de programmes illustrant ces manières de faire dans la bibliothèque de modèles de NetLogo

• Toutes les procédures doivent se terminer par la primitive «end» (fin), écrite sur une ligne séparée.

Pour écrire une procédure, on commence par ouvrir l'éditeur de code, en cliquant sur l'onglet «Code». Le clignement du curseur dans la zone vide de cet onglet indique que l'interpréteur attend votre code. L'exemple qui suit (Exemple 2) est très simple: il consiste à nettoyer l'écran et à créer 10 tortues qui doivent avancent de 12 pas en laissant une trace. Nous appellerons la procédure «dix-petites-tortues» (le nom doit consister en un seul mot, ce qui explique la présence de tirets dans ce nom).

Exemple 2 : Ma première procédure

Primitives: to (à ou vers), end (fin), create-turtles (créer-tortues), pendown (stylet-abaissé) forward (en avant), ask (demander), turtles (tortues), clear-all (nettoyer-tout ou réinitialiser).

Ouvrez l'onglet «Code» et écrivez le code suivant dans l'espace vide de cet onglet :

```
to dix-petites-tortues
clear-all
create-turtles 10
ask turtles [pendown forward 12]
end
```

Explications et commentaires supplémentaires. Pour exécuter la procédure «dix-petites-tortues», vous devez revenir à l'Interface (cliquez sur l'onglet Interface), taper le nom de la procédure «dix-petites-tortues» dans la fenêtre de l'observateur et taper sur la touche «Entrée» de votre clavier.

Si vous avez suivi ces instructions l'interface devrait ressembler à la Figure 1.9

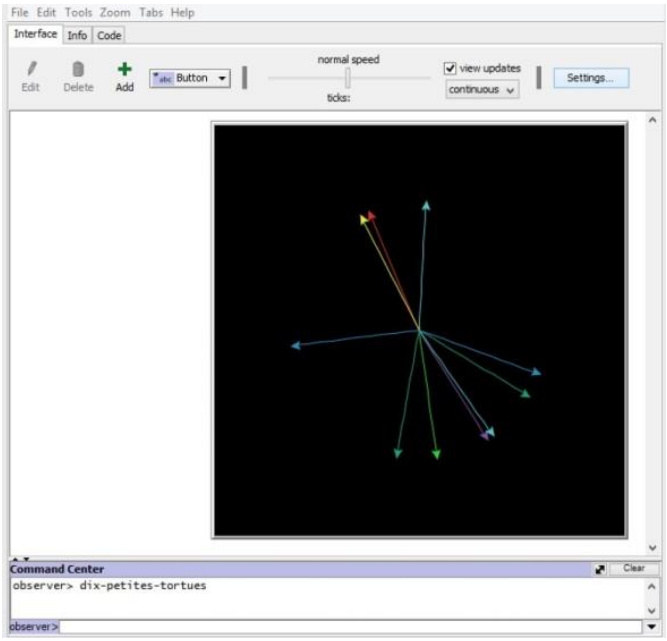


Figure 1.9 : Première procédure : dix-petites-tortues

La première commande «clear-all» vise à effacer les tortues (ainsi que leurs traces) crées par des procédures et commandes antérieures. L'omission de la commande «clear-all» entrainerait un mélange des traces laissées par les tortues déjà créées et des traces des dix nouvelles tortues qui s'ajouteraient au monde à chaque exécution de la procédure (vous pouvez vérifier vous-même ce résultat en supprimant la primitive «clear-all» de la procédure «dix-petites-tortues» et en tapant n fois de suite le nom de la procédure dans la fenêtre de l'observateur. Si vous faites une erreur en tapant le code vous verrez le nom de l'onglet «Code» affiché en rouge et un texte avec un arrière-plan jaune indiquant le type d'erreur trouvé par l'interprète. Lisez attentivement le texte de l'avertissement et essayez de trouver une solution au problème. Vous pouvez, si vous le désirez, revenir sur la fenêtre de l'Interface mais tant que l'erreur ne sera pas réparée la procédure ne sera pas exécutée. Et, tant qu'il reste des erreurs dans le code, l'onglet «Code» reste en lettres rouges.

Les tortues en tant qu'outils graphiques

L'idée de la tortue est due à Seymour Papert, l'un des créateurs du langage Logo. Papert a pensé à créer une entité virtuelle programmable capable, entre autres choses, de faire l'objet de représentations graphiques. La métaphore utilisée par Papert fut de représenter cette entité ou ce robot programmable par un animal - il a choisi une tortue - qui pouvait être dirigée au moyen de mouvements de base et qui pouvait laisser une trace en se déplaçant sur l'écran d'ordinateur [14]. Papert pensait que la manière la plus naturelle pour cette entité de se déplacer sur l'écran consistait à utiliser quatre mouvements de base, deux mouvements de déplacements en ligne droite et deux mouvements

de rotation, comme le font de nombreux animaux lorsqu'ils se déplacent dans l'espace. Les déplacements permettraient à la tortue d'avancer ou de reculer et ceux de rotation de changer de direction, en tournant sur un même axe, à droite ou à gauche.

Une primitive est affectée à chacun de ces quatre mouvements: pour se déplacer en ligne droite, les tortues utilisent les primitives «forward num» (en-avant num) et «back num» (en-arrière num) et, pour tourner, elles utilisent les primitives «right num» (droite num) et «left num» (gauche num). Pour les déplacements, l'argument «num» représente le nombre de pas que doit effectuer la tortue, tandis que pour les virages à droite ou à gauche, l'argument «num» exprime l'angle de rotation exprimé en degrés. Les tortues ont l'option de laisser une trace lorsqu'elles se déplacent, de se cacher ou de rester visibles, et même de changer d'aspect (voir le «Shapes Editor Guide» dans la section «Shapes» du Guide de l'usager [22]). Les tortues peuvent également effacer les traces qu'elles ont laissées ou estampiller leurs propres figures.

Cependant, il est important de noter qu'il existe quelques différences entre NetLogo et certaines variantes du langage Logo. En particulier, dans NetLogo, les tortues ne peuvent pas interagir avec leurs propres traces (sauf les effacer) car ces traces sont situées sur une couche différente de celle sur laquelle les tortues se déplacent. De ce fait, ces dernières ne peuvent détecter leurs propres traces. Autre conséquence : dans NetLogo, il n'existe pas de primitive permettant de colorer l'intérieur d'un contour fermé, tout comme le fait la primitive «fill» (colorer) dans certaines versions du langage Logo.

Dans le prochain exemple (exemple 3), une tortue est créée et chargée de dessiner un cercle. Comme les tortues ne peuvent se déplacer qu'en ligne droite, une tortue ne peut pas tracer de courbe au vrai sens du mot. Mais il est possible d'approximer une courbe par une ligne brisée composée de très courtes lignes droites.

Exemple 3: Une tortue dessine un cercle

Primitives: pd (abréviation de «pendown»), forward (en avant), right (à droite), set (assigne), color (couleur), yellow (jaune), repeat (répéter).

Nous allons construire un polygone avec un grand nombre de côtés de très petite taille, qui aura l'aspect d'un cercle. Nous ouvrons l'éditeur de code en cliquant sur l'onglet «Code» et écrivons le code suivant:

```
to cercle
clear-all
create-turtles 1
ask turtle 0 [pd set color yellow repeat 360 [forward 0.1
right 1 ] ]
end
```

Explications et commentaires supplémentaires. Les tortues sont numérotées au fur et à mesure de leur création, en partant de 0. Si au lieu de «ask turtle 0 [... commandes ...]», nous avions écrit «ask turtles [... commandes ...]», la commande aurait été adressée à toutes les tortues et l'effet aurait été le même parce que, à ce moment-là, la seule tortue qui peuple le monde est la tortue 0. La primitive «pd» est une abréviation de «pendown». La primitive «set» est utilisée pour attribuer une nouvelle valeur à une propriété d'un agent ou à une variable, telle que la couleur de la tortue.

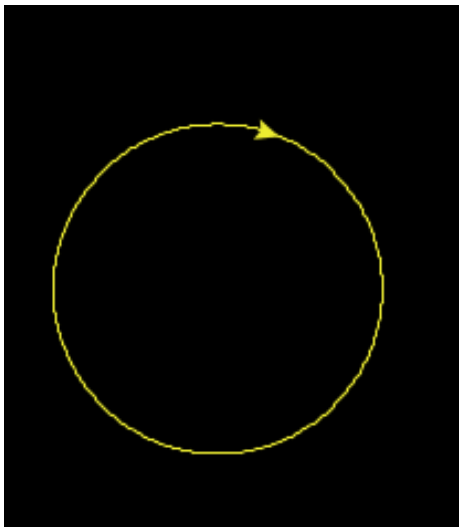


Figure 1.10 : Un cercle dessiné par une tortue

La commande «set color yellow» signifie «assigne la couleur jaune». Cette commande s'adresse à la tortue 0 car elle se trouve entre les crochets de «ask turtle 0 […]». La tortue prend la couleur jaune, couleur qui sera aussi la couleur de sa trace. Dans NetLogo, les couleurs ont un code numérique, mais les couleurs les plus courantes peuvent également être désignées par leur nom (pour plus d'informations sur les couleurs, voir l'option Couleurs dans le Guide de l'usager [22]).

La primitive «repeat num [..commandes ..]» permet de répéter les commandes entre crochets. Les commandes sont répétées autant de fois qu'indiqué par l'entrée «num». Dans notre exemple «repeat 360 [forward 0.1 right 1]», on demande à la tortue de répéter la séquence «forward 0.1 right 1», qui consiste

à avancer d'un dixième de pas et à tourner d'un degré vers la droite. Cela génère un polygone de 360 côtés de très petite taille (un dixième de pas), polygone qui a l'aspect d'un cercle, comme le montre la figure 1.10.

La procédure est lancée en revenant à la fenêtre de l'Interface et en tapant le mot «cercle» dans la fenêtre de l'Observateur. L'action d'écrire le nom d'une procédure s'appelle «appeler» ou «invoquer» la procédure. Les procédures peuvent également être invoquées à partir d'autres procédures.

Les parcelles

Les parcelles («patches») sont des agents statiques qui peuvent être programmés pour jouer des rôles très variés dans la création de modèles. Comme c'est le cas pour les tortues, chaque parcelle a ses propres caractéristiques telles que les coordonnées de sa position, sa couleur et sa taille. Ces caractéristiques peuvent être modifiées par les utilisateurs. Par défaut, le monde de NetLogo est constitué d'une grille de 1089 parcelles noires. Dans cette configuration, les coordonnées (16, 16) correspondent à la parcelle située dans le coin supérieur droit du monde. Dans NetLogo, les coordonnées d'une parcelle ne sont pas entre parenthèses et ne sont pas séparées par des virgules.

Ainsi, par exemple, si l'on veut faire référence à la parcelle de coordonnées (2, 4), il faut écrire «patch 2 4». Le langage NetLogo a également des primitives qui permettent de faire référence à des parcelles voisines d'une parcelle donnée. L'accès aux propriétés de ces voisines est un outil très puissant pour la création de modèles. Lorsque l'on fait varier le nombre de parcelles il est parfois nécessaire, pour que ces dernières restent visibles, de faire varier leur taille. Par exemple, un doublement du nombre de parcelles requiert une réduction de la taille des parcelles, afin que celles-ci restent dans le cadre du carré qui représente le monde.

Pour modifier le nombre ou la taille des parcelles, il faut ouvrir la fenêtre «Settings» de l'interface en cliquant sur le bouton qui porte ce nom. La taille de l'interface à l'écran peut être modifiée en faisant glisser les lignes horizontales ou verticales qui délimitent ses frontières, comme dans la plupart des applications Windows. La taille du monde peut également être augmentée ou réduite en cliquant sur l'onglet «Settings» et en choisissant les options appropriées. L'exemple qui suit (exemple 4) donne un exemple d'utilisation conjointe des parcelles et des tortues.

Exemple 4: Parcelles en diagonale

Primitives: patches (parcelles), with (avec), pxcor (coordonnée x d'une parcelle), pycor (coordonnée y d'une parcelle), pcolor (couleur d'une parcelle), = (signe égal), > (signe plus grand que), sprout (faire pousser, donner naissance).

```
to diagonale
clear-all
ask patches with [pxcor = pycor] [set pcolor
red]
ask patches with [pxcor = pycor + 5 ] [sprout
1]
end
```

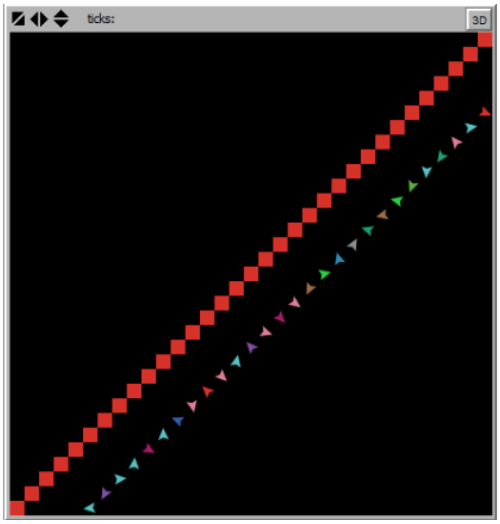


Figure 1.11 : Illustration de parcelles en diagonale

Explications et commentaires supplémentaires. Plusieurs primitives qui font référence aux parcelles portent des noms similaires à celles des tortues, mais elles s'en distinguent en commençant par la lettre «p». C'est le cas de pxcor, pycor ou pcolor, dont les équivalents pour les tortues sont xcor, ycor et color. La primitive «sprout num» (fait pousser num) n'appartient qu'aux parcelles et a pour effet de donner naissance à un nombre «num» (num = 1,2,3…n) de tortues sur les parcelles qui ont passé la commande.

Comme nous l'avons déjà mentionné, les tortues naissent avec des orientations et des couleurs attribuées au hasard. La première commande «ask patches with [pxcor = pycor] [set pcolor red]» pourrait être traduite par «demander aux parcelles dont les coordonnées x et y sont égales [coordonnée x = coordonnée

y] de se [colorier en rouge]». Dans cet exemple, il y a deux commandes qui utilisent la primitive «with» (avec). Cette primitive permet de former des commandes composées, qui élargissent les capacités d'expression du langage.

Avec cette primitive, il est possible de créer un ensemble-agents qui répond à une condition donnée : ensemble-agents with [condition].
La condition qui suit la primitive «with» doit être placée entre crochets.
Ainsi, la troisième ligne du code de l'exemple sur les parcelles en diagonale :

ask patches with [pxcor = pycor] [set pcolor red]

utilise la condition [pxcor = pycor] pour construire l'ensemble-agents des parcelles dont la coordonnée x est égale à leur coordonnée y (ce sont les parcelles situées sur la diagonale du monde). Il est ensuite demandé à ces parcelles de se colorier en rouge.

Les liens («links»)

Les liens sont des agents qui relient deux tortues. L'existence d'un lien nécessite l'existence des deux tortues qui lui servent d'extrêmité. NetLogo a deux types de liens: les *liens orientés* et les *liens non orientés*. Un lien orienté a une «origine» et une «destination», également appelées «racine» et «feuille», respectivement. Les liens peuvent être très utiles pour modéliser les relations entre les agents. Les réseaux ou graphes sont l'une des applications les plus importantes des liens.

Dans un graphe, les tortues peuvent représenter les nœuds (ou sommets) et les liens les arêtes. Dans NetLogo, on ne peut pas créer de liens entre les parcelles. Les liens entre les tortues doivent être du même type: orientés ou non orientés. Cependant, si l'on crée des familles de tortues - un concept que nous aborderons plus tard - il est possible que chaque famille ait son propre type de lien. Il n'est pas possible de créer un lien entre une tortue et elle-même (liens appelés boucles).

Cependant, une même tortue peut servir d'origine vers plusieurs autres tortues ou de destination en provenance d'autres tortues. NetLogo a de nombreuses primitives liées à ses agents, qu'il s'agisse de tortues, de parcelles ou de liens. Dans le cas des liens, l'expression «with» (avec) est caractéristique des primitives associées aux liens non orientés, tandis que les expressions «from» (de) et «to» (vers) sont utilisées pour les liens orientés. On peut, à titre d'exemple, citer les primitives «create-link-to» (créer-lien-vers), «create-link-from» (créer-lien-à-partir-de) ou «create-link-with» (créer-lien-avec). Chacune de ces trois primitives a une version plurielle, par exemple, «create-links-with»

(créer-liens-avec). Les liens, comme les tortues, peuvent être visibles (situation par défaut) ou non visibles.

Dans l'exemple qui suit (exemple 5), nous allons créer 6 tortues auxquelles il sera demandé d'avancer de 10 pas (pour les distinguer visuellement les unes des autres à l'écran) et à l'aide la primitive «create-link-to» ou «create-link-from», nous créerons des liens orientés entre certaines d'entre elles. Les tortues peuvent, par exemple, représenter un groupe de personnes dans un bureau, où l'on doit choisir la coordinatrice d'un projet. Un lien allant de la tortue A vers la tortue B pourrait alors signifier que la tortue A vote en faveur de la tortue B comme coordinatrice.

Exemple 5: Élection d'une coordonnatrice.

Primitives: create-link-to (créer-lien-vers), create-link-from (créer-lien-à-partir-de)

```
to election
clear-all
crt 6 [fd 10]
ask turtle 0 [create-link-to turtle 2]
ask turtle 0 [create-link-from turtle 5]
ask turtle 3 [create-link-to turtle 4]
ask turtle 1 [create-link-to turtle 4]
end
```

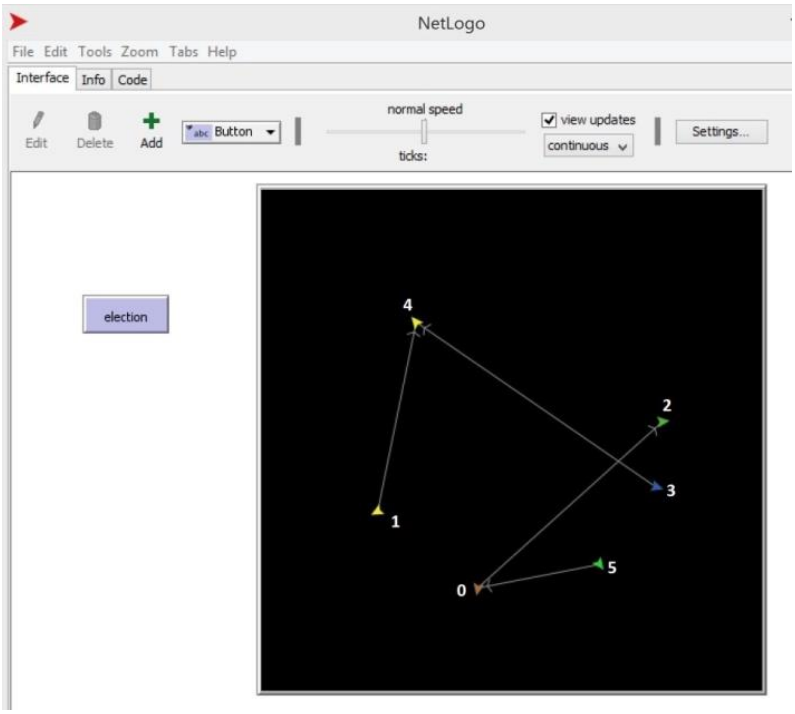


Figure 1.12: liens produits par la procédure «élection».

Explications et commentaires supplémentaires. Les suffixes «to» (vers) et «from» (à partir de) indiquent la direction d'un lien. L'instruction «ask turtle 0 [create-link-to turtle 2]» crée un lien orienté qui va de la tortue 0 vers la tortue 2, tandis que l'instruction «ask turtle 0 [create-link-from turtle 5]» crée un lien orienté à partir de la tortue 5 vers la tortue 0. Cet exemple montre comment une même tortue (la tortue 0) peut être à la fois une origine et une destination. La tortue 4 (qui est la seule à posséder deux liens qui pointent dans sa direction) remporte l'élection en tant que coordonnatrice. Si l'on avait voulu montrer un exemple de liens non orientés, on aurait utilisé la primitive «create-link-with». De tels liens sont utilisés pour modéliser des relations symétriques, telles les relations d'amitié.

Trois principes importants

Le principe de la concaténation.

Lorsqu'une procédure est appelée par une autre procédure, nous disons que les deux procédures sont *concaténées*. La procédure qui initie l'appel est le maillon supérieur de la chaîne et la procédure appelée est le maillon inférieur. Les gros programmes sont généralement constitués d'un ensemble de procédures plus petites qui sont concaténées pour former des réseaux. Un

programme peut même se concaténer avec lui-même, comme dans le schéma de *récursion*, qui sera abordé ultérieurement. La concaténation de procédures permet d'appliquer deux autres principes d'une grande importance en sciences et en génie et que nous appliquons fréquemment dans la vie quotidienne. Ces principes sont particulièrement importants dans le domaine de la programmation informatique.

Principe de la subdivision: En quelques mots, ce principe consiste en la *subdivision appropriée d'une tâche complexe en sous-tâches de complexité et de taille réduites*. L'efficacité du principe réside dans le bon choix des sous-tâches et de leurs relations mutuelles. L'auteur d'un roman pourrait, par exemple, demander la révision de l'orthographe et de l'écriture de son roman en distribuant différents chapitres à différents relecteurs. Mais il serait absurde de décider de subdiviser la révision en deux parties dont l'une consisterait à relire tous les mots du roman commençant par une consonne tandis que l'autre porterait sur la relecture de ceux qui commencent par une voyelle. Le bon sens nous empêche de faire certaines subdivisions que nous pourrions qualifier d'absurdes, mais dans les projets très complexes, le choix des sous-tâches n'est pas toujours évident.

Principe de la réutilisation: si vous devez accrocher dix tableaux sur les murs de votre maison, vous devez utiliser au moins dix clous, mais vous n'avez pas besoin de dix marteaux, un seul suffit. Grâce au principe de réutilisation, il est possible d'utiliser de manière répétée des outils et des ressources déjà disponibles, chaque fois que cela est possible et utile. Un bon exemple de ce principe en informatique est fourni par les fichiers CSS sur un site Web, qui stockent les détails de style des pages Web. Si, par exemple, nous voulions changer le type de caractères ou la couleur des paragraphes des 100 pages d'un site, il suffirait de faire une petite modification du fichier CSS dans lequel le style est défini, au lieu de modifier les 100 pages l'une après l'autre.

Exemple 6: Concaténation, subdivision et réutilisation

Primitives: setxy (asignerxy), set (asigner), repeat (répéter), pd (abrev. de pendown, stylet-vers-le-bas), pu (abrev. de penup, stylet-vers-le-haut), fd (abrev. de forward, en-avant), rt (abrev. de right-turn, tourner-à-droite), ht abréviation de hide-turtle (cacher-tortue).

Dans cet exemple, une tortue dessine trois cercles entrelacés de couleurs différentes, tel qu'illustré sur la figure 1.13. La procédure est lancée en tapant «trois cercles» dans la fenêtre de l'observateur.

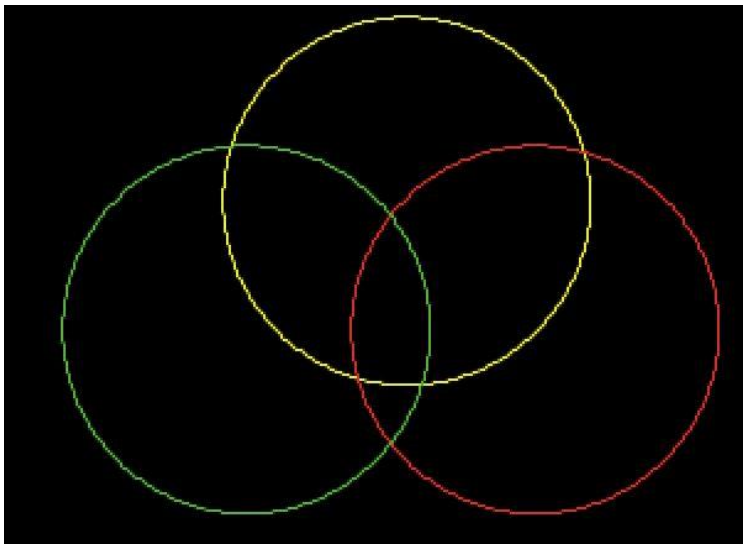


Figure 1.13: Trois cercles entrelacés

Voici le code :

```
to trois-cercles
clear-all
crt 1
ask turtle 0 [setxy 3 2 pd set color yellow
cercle
pu setxy 7 -2 set color red pd cercle
pu setxy -2 -2 set color green pd cercle ht
]
end

to cercle
repeat 360 [fd 0.1 rt 1]
end
```

Explications et commentaires supplémentaires. En dépit de sa simplicité, cette procédure illustre bien l'application des trois principes: subdivision, concaténation et réutilisation. La procédure «cercle» a été supprimée du corps de celle intitulée «trois-cercles» et est invoquée à partir de cette dernière (principes de concaténation et de subdivision). La procédure «cercle» est utilisée trois fois (principe de réutilisation). La primitive «setxy num1 num2»

déplace la tortue vers le point de coordonnées (num1, num2) et pd abaisse le stylet pour tracer le premier cercle (le cercle jaune). Comme on ne veut pas que la tortue laisse de trace quand elle se déplace au point de départ du cercle suivant (le cercle rouge dans l'exemple), il est nécessaire de d'abord passer la commande lever le stylet (pu), mais une fois que le nouveau point de départ est atteint, il faut abaisser à nouveau le stylet (pd) pour commencer à dessiner le nouveau cercle. Si l'on veut que la tortue qui a tracé les cercles (la tortue 0 dans l'exemple) ne soit pas visible à l'écran il suffit de passer la commande «hide-turtle» (abréviation ht) à la fin de la procédure.

Un autre avantage offert par la subdivision d'un programme en procédures exécutant des tâches spécifiques concerne l'introduction de modifications ou la réparation d'erreurs, c'est-à-dire la maintenance du programme. Si l'on veut, par exemple, diminuer ou agrandir la taille des cercles il suffit de modifier la procédure «cercle» (par exemple en remplaçant fd 0.1 par fd 0.05 pour diminuer la taille des trois cercles). Enfin, les trois principes ci-dessus offrent un autre avantage: le code résultant de leur application est plus clair et plus facile à comprendre.

Types de primitives NetLogo

Les primitives de NetLogo ont certaines caractéristiques importantes qui nous permettent de les regrouper en catégories. Nous citerons deux critères de classification des primitives, indépendants l'un de l'autre. La première classification est basée sur le fait que les primitives nécessitent ou non de données d'entrée. La deuxième fait une distinction entre les primitives d'action et les primitives informatives.

Primitives qui requièrent des données d'entrée. Prenons comme exemple la commande «forward num», qui ordonne à une tortue d'avancer d'un nombre «num» de pas. Dans le jargon informatique, on appelle «num» une «entrée» de la primitive «forward». Parfois, les mots «paramètre» ou «argument» sont également utilisés comme synonymes du mot «entrée». Dans NetLogo, des exemples de primitives nécessitant des entrées sont: «right», «create-turtles» et «set». Les commandes «setxy 10 3» ou «repeat 4 [forward 10 left 90]» montrent que ces deux primitives fonctionnent avec deux entrées. Dans le cas de «repeat» (répéter) la deuxième entrée n'est pas un nombre mais une liste de commandes. Il arrive souvent que les valeurs des entrées d'une primitive proviennent de primitives appartenant à d'autres primitives qui fournissent ou «rapportent» des valeurs. Ceci nous amène à parler de la deuxième catégorie de primitives.

Primitives d'action et primitives informatives («reporter»).

Une primitive est dite d'action si elle donne des ordres qui obligent les agents à «faire quelque chose». Quelques exemples de primitives d'action que l'on a rencontrées jusqu'à présent sont : «forward» (fd), «clear-all» (ca) et create-turtles (crt). Ces primitives génèrent, respectivement, les actions suivantes: avancer, réinitialiser le programme et créer de nouvelles tortues. Les primitives d'action sont également appelés «commandes». Les autres primitives – celles qui ne sont pas des primitives d'action - sont des primitives de type «reporter» (rapporteur). Ces primitives sont caractérisées par le fait qu'elles fournissent ou rapportent des données qu'elles mettent à la disposition des entités (autre primitive, procédure) qui en ont besoin. Un exemple de primitive de type «reporter» est «color». Au lieu de générer une action, cette primitive rapporte le numéro de couleur de la tortue à l'agent qui le demande. A titre d'exemple, examinons les deux commandes suivantes (dans la fenêtre de l'observateur):

```
crt 1
ask turtle 0 [show color]
==> (turtle 0): 75
```

La première commande crée une tortue (nous supposons que c'est la première tortue à être créée et son numéro d'identité est donc le numéro 0). Dans la deuxième commande, on demande à la tortue 0 de montrer sa couleur. La primitive «color» rapporte ici la valeur (75) c'est-à-dire qu'elle fournit la valeur 75 à la primitive «show», une primitive d'action qui nécessite une entrée pour fonctionner. «Show» prend la valeur rapportée par «color» et l'affiche sur le terminal d'instructions. Si, en testant plusieurs fois ces instructions, les lecteurs n'obtiennent pas la valeur 75, c'est tout à fait normal. Rappelez-vous que, par défaut, les couleurs des tortues sont attribuées de manière aléatoire.

Ce petit exemple illustre une construction très commune dans laquelle deux primitives sont combinées, l'une rapportant des données et l'autre les recevant comme entrées. Comme nous le verrons ultérieurement, on peut opérer de la même manière avec les procédures, l'une jouant le rôle de rapporteur alors que l'autre joue un rôle de récepteur.

Chapitre 2: Environnement et langage II.

Les boutons pour appeler des procédures

Si l'on ouvre quelques modèles de la bibliothèque de modèles du menu «File» (Fichier), on remarque que la grande majorité de ces modèles est caractérisée par la présence d'éléments (objets) tels que des boutons, des curseurs ou des graphiques dans la zone blanche à gauche du carré monde. Ces objets ont été construits pour remplir des fonctions spécifiques, par exemple exécuter une procédure, accélérer ou retarder un processus, fournir des informations sur l'évolution des variables d'un modèle, etc... Le type d'objet que l'on désire construire est choisi en cliquant sur l'onglet de l'icône intitulé «Button» (Bouton) qui est en fait un menu déroulant avec les différents objets qu'il est possible de construire (par défaut, l'objet «Button» est affiché en tête de liste du sélectionneur d'objets – voir Figure 2.1-).

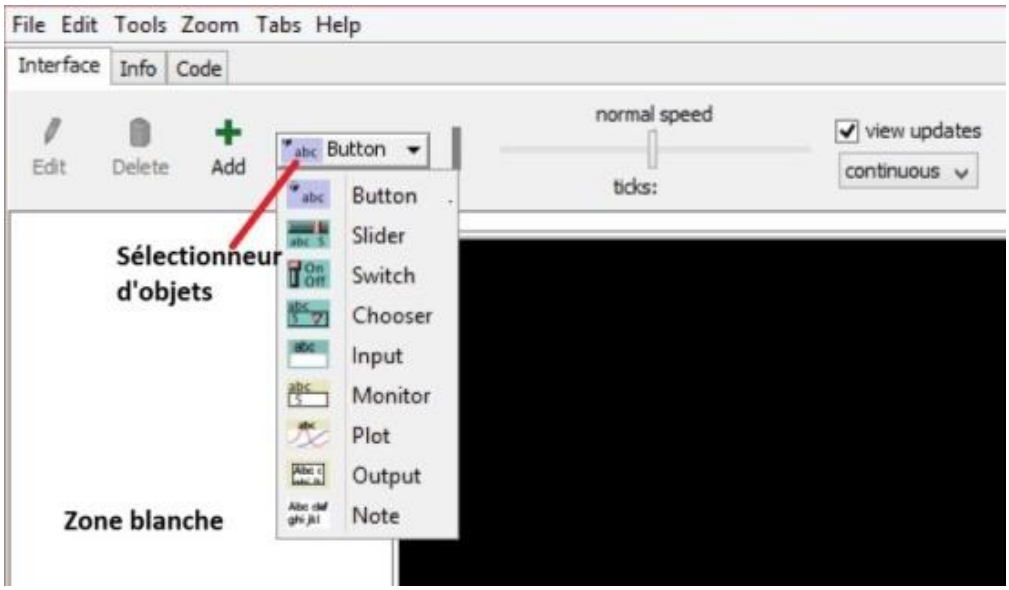


Figure 2.1 : Le sélectionneur d'objets

Les boutons «setup» et «go»

Une pratique courante dans la communauté NetLogo consiste à construire deux boutons appelés «setup» (configurer) et «go» (aller). Le bouton «setup» sert généralement à préparer le terrain avant de mettre en œuvre les actions d'un modèle. Il sert, par exemple, à supprimer les graphiques laissés par les procédures ou par les exécutions précédentes, attribuer les valeurs initiales aux variables, créer des agents, etc. Le bouton «go» lance les actions du modèle. La raison qui justifie la création de ces boutons est très simple: il est plus pratique d'exécuter une procédure en appuyant sur un bouton que de taper son nom dans la fenêtre de l'observateur. Outre les boutons «setup» et «go», on

construit souvent d'autres objets pour mettre en marche d'autres types de processus ou pour choisir différentes modalités d'exécution d'un modèle (voir Figure 2.2). L'idée n'est autre que de rendre la gestion et l'obtention des informations du modèle plus agiles.

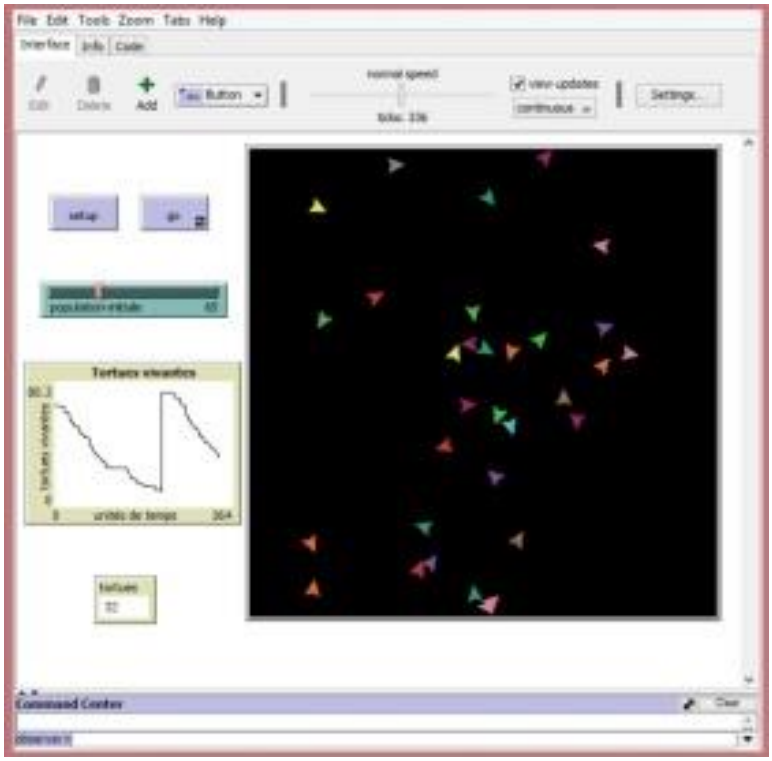


Figure 2.2 : Exemple d'objets d'un modèle créé avec NetLogo

Bien entendu, les commandes qui s'exécutent en cliquant sur un objet doivent se trouver quelque part. Ainsi, pour un bouton, il y a deux endroits où ces commandes (ou instructions) peuvent être écrites.

1. Dans l'éditeur de code, en écrivant une procédure dont le nom correspond à celui du bouton.
2. À l'intérieur de la fenêtre d'édition du bouton, fenêtre qui s'ouvre en cliquant sur le bouton droit de la souris. Les instructions doivent être écrites dans le cadre intitulé «Commands» de ladite fenêtre

À chaque bouton doit correspondre une procédure - généralement du même nom - dont le code se trouve dans l'éditeur de code. Cette procédure est activée en cliquant sur le bouton correspondant. Il est également possible d'écrire des instructions dans la fenêtre d'édition des boutons.

Comment construire un bouton
La construction d'un bouton se fait en trois étapes:

• Sélectionner. Si le mot «Bouton» n'apparait pas dans le sélectionneur d'objets utilisez le menu déroulant pour choisir l'option «Bouton».
• Construire. La construction est activée en cliquant sur l'icône avec le signe «+» (de couleur verte) au-dessus du mot «Add» (Ajouter) situé juste à la gauche du sélectionneur d'objets (voir Figures 2.1 et 2.2). Le pointeur prend alors la forme d'un signe «+». Placez le pointeur de la souris à l'endroit où vous voulez construire le bouton (habituellement, on choisit un endroit situé dans la zone blanche à gauche du monde) et cliquez sur cet endroit.
• Nommer La fenêtre d'édition du bouton que l'on construit s'ouvre immédiatement (voir Figure 2.3) et on écrit le nom de la procédure que le bouton doit invoquer (par exemple, «setup» ou «go») dans la case «Commands» à l'endroit où le curseur clignote. Pour finir, on clique sur l'option OK (encerclée sur la Figure 2.3) dans cette fenêtre. Le bouton a été construit et le nom de la procédure qui lui est associée s'affiche. Le nom du bouton reste en lettres rouges jusqu'à ce que l'on écrive la procédure associée au bouton dans l'éditeur de code ou les instructions dans la case «Commands». Il est possible de repositionner le bouton n'importe où dans la zone blanche en cliquant sur le bouton droit de la souris et en choisissant l'option «Sélectionner», ce qui permet de le faire glisser vers un nouvel emplacement.

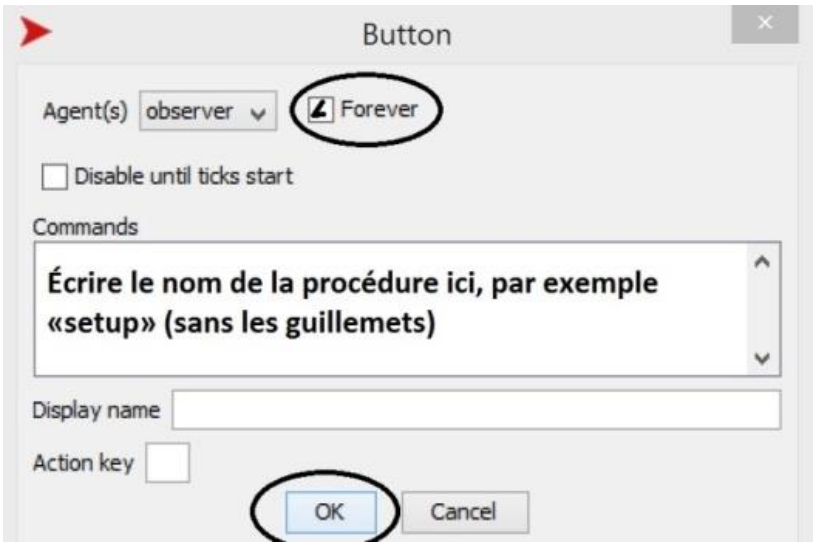


Figure 2.3 : La fenêtre d'édition d'un bouton

Chaque objet (bouton, curseur, interrupteur, etc.) possède sa propre fenêtre d'édition, qui s'ouvre par un clic droit sur l'objet en question.

Si l'on veut donner au bouton un nom différent de celui de la procédure qui lui est associée, il suffit d'écrire le nom désiré dans la case intitulée «Display name» de la fenêtre d'édition du bouton. Cependant, quand on clique sur le bouton, la

procédure qui est exécutée est celle dont le nom est écrit dans la case «Commands».

Répétition continue d'une procédure.

L'un des aspects le plus intéressant de l'étude d'un modèle est de suivre son évolution dans le temps. Cette évolution peut porter sur différents aspects du modèle tels que, par exemple, le nombre et le comportement des agents ou les valeurs des variables du modèle. Dans la plupart des cas, l'évolution temporelle est généralement modélisée par la répétition continue d'un ou de plusieurs processus, représentés par des procédures dont les paramètres varient généralement à chaque répétition. La question se pose alors de savoir s'il est possible de répéter la procédure un nombre illimité de fois. Dans NetLogo, comme dans d'autres langages de programmation, il existe des mécanismes qui permettent de programmer des processus itératifs, y compris des processus qui peuvent être répétés un nombre illimité de fois («Forever» dans le langage NetLogo). Ces mécanismes sont rendus possibles grâce à l'utilisation de primitives ou de schémas de programmation inclus dans le code d'un programme et auxquels nous ferons référence plus tard. NetLogo possède aussi un mécanisme simple qui permet l'itération continue d'une procédure sans avoir à la programmer dans le code. Ce mécanisme est activé à l'aide de la case «Forever» (sans arrêt, en continu), une case propre aux boutons et dont les caractéristiques sont présentées ci-dessous.

La case «Forever» Pour qu'une procédure soit répétée en continu (un nombre illimité de fois), il faut:

- Créer un bouton affecté à la procédure que l'on souhaite répéter.
- Ouvrir la fenêtre d'édition des boutons et cochez la case «Forever» (encerclée sur la Figure 2.3) qui, par défaut, est décochée.
- Cliquez sur le bouton OK pour fermer la fenêtre.

Une fois la case «Forever» cochée, chaque fois que la procédure est appelée en cliquant sur le bouton, cette dernière est répétée et ne s'arrête que dans l'un des trois cas suivants : 1) si une condition d'arrêt imposée au code est remplie, 2) si l'utilisateur force la fin des répétitions en appuyant de nouveau sur le bouton, ou 3) si l'utilisateur arrête les actions en ouvrant l'onglet «Tools» (Outils) de la barre de menus et en sélectionnant l'option «Halt» (Arrêter). Notez cependant que lorsqu'elle est appelée à partir de la fenêtre de l'observateur, la procédure ne s'exécute qu'une seule fois, même si la case «Forever» est cochée.

Lorsque la case «Forever» est cochée dans la fenêtre d'un bouton, la procédure qui lui est associée est répétée sans interruption et peut être interrompue en cliquant à nouveau sur le bouton.

Commentaires dans le code

Les commentaires et explications sur le code des exemples présentés jusqu'ici ont été ajoutés à la fin du code et, en raison de la nature pédagogique de ce livre, nous continuerons à utiliser cette pratique dans la plupart des exemples. Cependant, tous les langages informatiques offrent la possibilité d'insérer des commentaires explicatifs dans le code d'un modèle ou d'un programme. Ces commentaires s'adressent exclusivement aux humains et ont pour but de faciliter la compréhension du code par les personnes qui le lisent, y compris ceux qui l'ont écrit. La construction de gros programmes est généralement un travail d'équipe qui implique plusieurs groupes de personnes en charge des différents modules du programme. Les commentaires jouent un rôle très utile pour faciliter une connaissance partagée de l'ensemble du programme entre ces équipes. De plus, les commentaires peuvent également être très utiles pour les personnes qui ont écrit le code ou une partie de celui-ci, car très souvent, des programmes de maintenance ou d'amélioration doivent être apportés aux programmes écrits il y a longtemps.
Dans NetLogo, les commentaires commencent par une paire de points-virgules «;;» et se terminent à la fin de la ligne. Cette convention avertit l'interpréteur que le texte qui suit ces caractères et se termine à la fin de la ligne ne fait pas partie du code et doit être ignoré. Un commentaire peut être écrit sur la même ligne qu'une ligne de code s'il est écrit après le code et séparé de ce dernier par la paire de points-virgules. Si un commentaire nécessite plus d'une ligne, chaque nouvelle ligne doit commencer par les deux points-virgules. On peut constater, en examinant quelques exemples de modèles de la bibliothèque de NetLogo, la fréquence d'utilisation de l'incorporation de commentaires dans le code des programmes.

Exemple 7: Deux tortues attachées entre elles

Primitives: «setxy» (établirxy), «create-link-with» (créer-lien-avec), «tie» (attacher), «wait» (attendre).
Autres détails: On construit les boutons «setup» et «go», on contrôle la vitesse du processus à l'aide la primitive «wait», on incorpore des commentaires aux lignes de code.

Un lien crée une relation entre deux tortues, mais cette relation n'implique pas que, lorsqu'une tortue se déplace, l'autre doit également le faire. Par contre, une attache agit comme une sorte de corde qui lie les deux tortues de sorte que lorsque l'une bouge l'autre est tirée dans la même direction que la première.
Dans le présent exemple, on crée une attache entre deux tortues, ce qui nécessite la création préalable d'un lien entre ces tortues. Ensuite, on demande à l'une des tortues de dessiner un cercle. Comme le lien sur lequel est définie l'attache est non directionnel, les deux tortues peuvent indifféremment, être considérées comme tortue-origine ou comme tortue-destination.
Par conséquent, si la tortue à laquelle la commande est passée en premier tourne ou avance, l'autre doit procéder de la même manière, de sorte que la distance entre les deux tortues reste constante. Cependant, l'attache ne force pas la tortue qui est tirée à adopter la même orientation («heading») initiale que celle qui la tire.

```
to setup
clear-all ;; ceci est un exemple de commentaire qui
 ;; sera ignoré par l'interpréteur
crt 2
ask turtle 0 [set color red pd setxy 0 0] ;; «set color red» assigne la
couleur
 ;; rouge à  la tortue 0 et  «setxy» sert à assigner une position à cette
même tortue
ask turtle 1 [set color green pd setxy -2 -2]
end

to go
ask turtle 0 [ create-link-with turtle 1 [tie] ] ;; on crée un lien entre les
deux
 ;; tortues
ask turtle 1 [ repeat 360 [fd 0.1 rt 1 wait 0.1 ]]
end
```

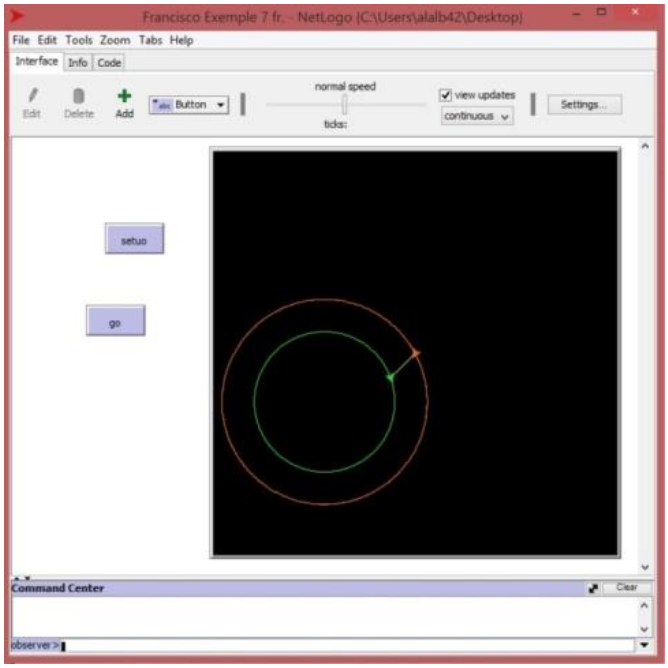


Figure 2.4: Deux tortues attachées dessinent un cercle

Explications et commentaires supplémentaires. Pour exécuter la procédure, vous devez d'abord cliquer sur le bouton «setup», puis sur le bouton «go». On demande à la tortue 1 (la tortue verte) de dessiner un cercle afin que son orientation soit toujours tangente au cercle. La tortue 0 (la tortue rouge), par contre, est «entraînée» par la tortue 1 et son orientation n'est pas nécessairement tangente au cercle qu'elle trace. Il est possible de définir des attaches asymétriques par le biais de liens orientés. Dans un tel cas, lorsque la tortue origine (racine) se déplace, la tortue cible doit la suivre, mais pas l'inverse. La primitive «setxy» attribue à la tortue les coordonnées indiquées comme entrées. Rappelez-vous que les coordonnées des parcelles ne sont pas écrites entre parenthèses, comme c'est la coutume en mathématiques. La commande «wait 0.1» (attendre 0.1) produit une attente d'environ un dixième de seconde. Cette commande a été introduite pour apprécier le mouvement des tortues, sinon tout se passerait trop vite (suggestion: supprimez la commande «wait 0.1» et observez ce qui se passe).

Exemple 8: La tortue parcourt un cercle sans arrêt

Primitives: «heading» (orientation), «wait» (attendre).
Autres détails: Une fois les boutons «go» et «setup» construits, il faut cocher la case «for ever» (sasn arrêt, en continu) du bouton «go» afin que la procédure se répète sans arrêt.

Dans cet exemple, une tortue dessine un cercle et le parcourt lentement mais sans relâche jusqu'à ce qu'elle soit stoppée par l'utilisateur. La primitive «heading» (orientation) est utilisée pour fixer l'orientation initiale de la tortue. La primitive «wait» est utilisée ici comme dans l'exemple précédent, pour réduire la vitesse de la tortue et ainsi pouvoir observer son mouvement. Le contrôle de la vitesse d'une procédure peut également être effectué à partir de l'interface, à l'aide du curseur (situé dans la partie supérieure du monde) dont la valeur par défaut est «normal speed» (vitesse normale). Pour exécuter le programme, cliquez sur le bouton «setup» puis sur le bouton «go». Pour l'arrêter, cliquez à nouveau sur le bouton «go». Le code est le suivant:

```
to setup
clear-all
crt 1 [set heading 0 pd]
end

to go
ask turtle 0 [fd 0.3 right
3]
wait 0.1
end
```

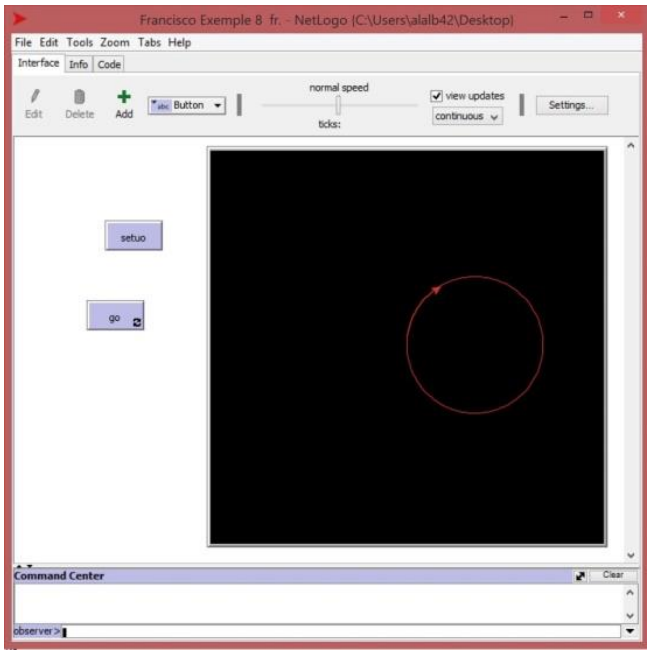


Figure 2.5 : Parcours circulaire d'une tortue

Le rôle du hasard dans la construction des modèles

Il existe de nombreux phénomènes de la vie réelle qui peuvent être modélisés efficacement à l'aide de mécanismes fondés sur le hasard. Pour générer des événements aléatoires, NetLogo a recours à la primitive «random num» (nombre aléatoire), qui fonctionne avec une valeur d'entrée numérique. Le fonctionnement de cette primitive est le suivant: «random n» rapporte un entier aléatoire compris entre 0 et n - 1. Par exemple, la commande «random 5» donner un nombre choisi aléatoirement parmi l'ensemble des 5 éléments {0, 1, 2, 3 4}. Avec un peu d'ingéniosité, vous pouvez faire en sorte que cette primitive choisisse des nombres aléatoires dans d'autres plages numériques, comme le montrent les exemples de commandes qui suivent (exemple 9).

Un autre mécanisme très utile pour générer des conditions aléatoires est fourni par la primitive «one-of» (une-parmi), qui a pour entrée une liste d'options. Cette primitive rapporte l'une des options de la liste en question en la choisissant au hasard. Dans NetLogo, une liste est construite en plaçant ses membres entre crochets et en les séparant par des espaces. Par exemple, [1 3 5 7] est une liste contenant quatre entiers. Sans l'avoir explicitement mentionné, nous avons déjà rencontré des listes, par exemple lorsque nous avons utilisé la primitive «repeat» laquelle fonctionne avec deux entrées, dont l'une est la liste des commandes. Nous aborderons les listes plus en détail plus loin.

Exemple 9: Commandes avec la primitive «random»

- **random 50**, rapporte un entier aléatoire compris entre 0 et 49.
- **random 50.3**, rapporte un entier aléatoire compris entre 0 et 49.
- **(random 50) + 50** rapporte un nombre entier aléatoire compris entre 50 et 99. Si, par exemple, random 50 génère le nombre 17, **(random 50) + 50** prend la valeur 17 + 50 = 67.
- **(random 100) / 100**, rapporte un nombre compris entre 0 et 1 choisi au hasard dans l'ensemble {0.00, 0.01, 0.02, 0.03, ... 0.98, 0.99}. Si, par exemple, random 100 génère le nombre 17, **(random 100) / 100** prend la valeur 17/100 = 0,17.
- **(random 1000) / 1000**, rapporte un nombre compris entre 0 et 1 choisi au hasard dans l'ensemble {0.000, 0.001, 0.002, 0.003, ... 0.998, 0.999}.

Exemple 10: Commandes avec la primitive «one-of»

- **Show one-of [1 2 3 4]**
  ==> observer: 2, un nombre, choisi au hasard, de la liste [1 2 3 4] est affiché.
- **show one-of {"A" "B" "C" "A" "A"}**
  ==> observer: A

En modifiant la fréquence des membres qui appartiennent à une liste on peut modifier la probabilité de sélectionner certains membres plutôt que d'autres. Dans la commande précédente, la lettre A est trois fois plus susceptible d'être sélectionnée au hasard que les lettres B ou C. En informatique, le processus de génération de nombre aléatoires est construit à partir d'un nombre de départ (une graine - «seed» en anglais-), dont les chiffres satisfont à des critères statistiques qui caractérisent le hasard (comme, par exemple, les chiffres de l'expansion décimale du nombre pi). Les utilisateurs peuvent modifier la valeur de la graine utilisée par NetLogo via la primitive «random-seed» (graine aléatoire). Le Manuel de l'utilisateur de NetLogo [16] fournit plus d'informations à ce sujet dans la section «Random numbers» (Nombres aléatoires).

Exemple 11: Une promenade aléatoire

Primitives: «random» (hasard), «wait» (attendre). Autres détails: La vitesse du processus est réglée avec la primitive «wait».

Dans cet exemple, une tortue effectue une promenade avec le stylet baissé («pendown» ou «pd»), de sorte qu'elle laisse une trace à mesure qu'elle avance. En répondant aux commandes d'avancer de deux pas sans arrêt («fd 2») et de se laisser guider par le hasard pour ce qui est de son orientation («random 360»), la tortue fait une promenade aléatoire.

Activités préparatoires: construire les boutons «setup» et «go» (cochez la case «forever» de «go») et écrivez le code suivant:

```
to setup
clear-all
crt 1 [pd set color yellow]
end
to go
```

ask turtle 0 [fd 2 set heading random 360]
wait 0.2
end

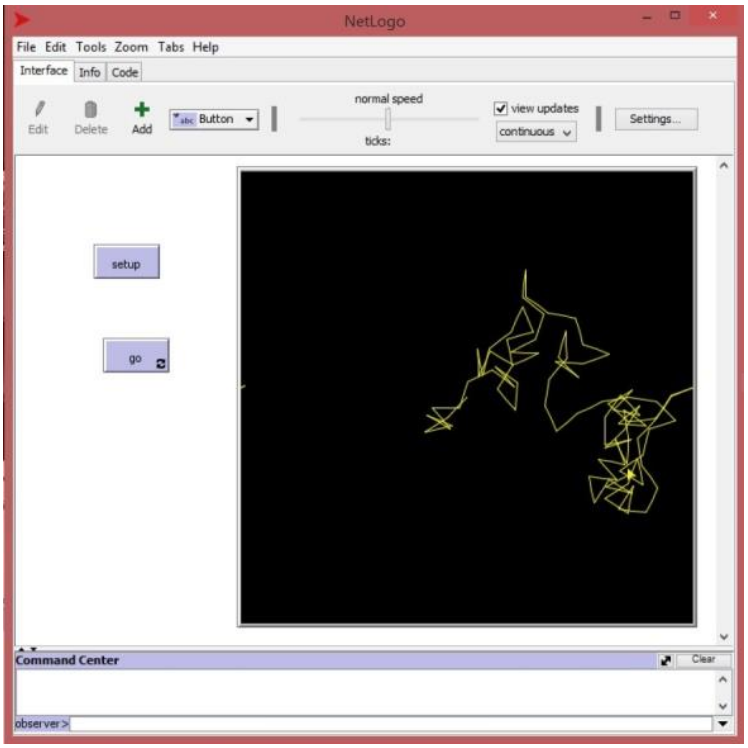


Figure 2.6 Trajectoire de la tortue après quelques rondes

Les variables dans NetLogo

Les variables constituent l'une des ressources les plus puissantes et les plus importantes des langages de programmation. Dans NetLogo, tout attribut des agents ou de l'environnement dont la valeur peut être modifiée est représenté par une variable. Il est utile d'imaginer une variable comme une boîte dans laquelle est stocké un objet dont la valeur peut être modifiée («variée») à tout moment. Les objets peuvent être de nature très différente: nombres, passages de texte, listes ou même ensemble d'agents. Une variable possède deux caractéristiques de base: son nom et sa valeur (également appelée son contenu). Pour en revenir à la comparaison d'une variable avec une boîte, le nom de la variable représenterait le nom de la boîte et la valeur serait son contenu. Il existe des variables créées par les utilisateurs et d'autres créées par le système. Nous appelons ces dernières «variables préinstallées» (dans le système). Cela suggère une première division des variables en deux grands groupes: Les variables préinstallées dans le système et les variables définies par les utilisateurs.

Première rencontre avec les variables : Variables préinstallées

Les agents de NetLogo possèdent plusieurs caractéristiques propres, qui peuvent être modifiées par les utilisateurs. Les tortues, par exemple, ont, entre autres, une couleur, une position (donnée par leurs coordonnées xcor et ycor), une orientation («heading») et un état de leur stylet («penup» ou «pendown»). Les parcelles «patches» ont, par exemple, des coordonnées de position (pxcor, pycor) et de couleur (pcolor). De leur côté, les liens ont également leurs propres caractéristiques, telles leurs extrémités (constituées par des tortues), leur épaisseur et la propriété d'être visible ou non. Toutes ces caractéristiques des agents sont représentées au moyen de variables. Parfois, la variation est produite par l'utilisateur lorsque, par exemple, celui-ci détermine la couleur d'une tortue et, parfois, le système NetLogo le fait automatiquement quand, par exemple, l'utilisateur, ayant demandé à une tortue d'avancer d'une certaine distance, le système ajuste automatiquement les valeurs de ses coordonnées de position. Dans l'exemple précédent, la couleur jaune a été attribuée à une tortue à l'aide de la commande «set color yellow». La valeur des variables prédéfinies est modifiée avec la primitive «set». Le format de la commande est: *set nom-de-la-variable nouvelle-valeur.*

Exemple 12: Commandes avec la primitive «set»

crt 2, on crée deux tortues dont les numéros d'identification sont 0 et 1.
ask turtle 1 [print heading]
==> 225, la tortue 1 montre son orientation.
ask turtle 1 [set heading 45 print heading]
==> 45, la tortue 1 assigne la nouvelle valeur 45 à son orientation et cette valeur apparaît dans le terminal d'instructions.
ask turtle 0 [print who]
==> 0, la tortue 0 fournit son numéro d'identification.
ask patch 3 6 [print pcolor]
==> 0, la parcelle 3 6 fournit sa couleur (0 = noir).
ask patch 3 6 [set pcolor red], la parcelle 3 6 change sa couleur à rouge.

La primitive «print» envoie la valeur de l'entrée au terminal d'instruction, mais contrairement à la primitive «show», la valeur n'est pas précédée du nom de l'agent qui exécute la commande. Certaines variables peuvent être consultées par les utilisateurs, mais elles ne peuvent pas être modifiées. Tel est le cas de la variable «who», qui reporte le numéro d'identification d'une tortue. Cette variable est créée par le système au moment de la naissance d'une tortue. Les numéros «who» sont attribués au fur et à mesure que les tortues sont créées, à partir de 0. Quand une tortue meurt, son numéro «who» n'est réaffecté à aucune des tortues qui restent en vie, c'est-à-dire qu'il n'y a pas de renumérotation de toutes les tortues vivantes. Le système pourrait cependant réaffecter le numéro de la tortue disparue à une tortue née plus tard.

Les structures topologiques du monde NetLogo

Le domaine des mathématiques qui étudie les propriétés fondamentales des ensembles, des figures et des espaces est appelé topologie. Contrairement à la géométrie, qui prend en compte la mesure de longueurs, d'aires et d'angles, la topologie ignore ces quantités et étudie des propriétés plus fondamentales, par exemple le nombre de trous d'une surface, la forme des connexions entre certaines parties, ou le nombre de voisins de chacune des parcelles lorsque la surface est quadrillée. Le monde NetLogo, qui consiste en une surface bidimensionnelle, peut adopter trois structures topologiques.

- Topologie du tore ou du beignet (beigne). C'est la topologie du monde NetLogo par défaut. Comme nous l'avons déjà vu, lorsqu'une tortue quitte le carré du monde à l'extrême droite, on la voit

réapparaître à l'extrême gauche et inversement. De même, si la tortue quitte le monde par le haut, on la voit émerger du bas et vice versa. Ce n'est pas une propriété géométrique du monde mais une propriété topologique. Une surface qui a cette propriété s'appelle un «tore» et a la forme d'un beignet (Figure 2.7).

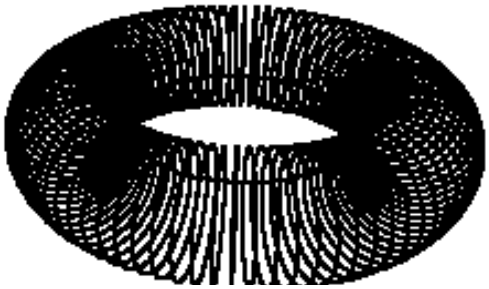


Figure 2.7 : Exemple de topologie de tore (ou de beignet)

Lorsque nous parlons de tore, nous faisons référence à l'objet bidimensionnel constitué par sa surface, et non au corps solide qui comprend l'intérieur de cette surface. L'écran d'ordinateur étant une surface plane semblable à une feuille de papier, le moyen le plus approprié de représenter la surface d'un tore à l'écran consiste à imaginer que l'on ouvre un beignet avec des cisailles et que l'on étend la surface qui résulte de cette opération sur l'écran. Cette opération est mieux comprise si le processus inverse est effectué, qui consiste à transformer un morceau de papier plat en tore en collant ses bords opposés. Imaginons que nous prenions une feuille de papier rectangulaire ou carrée et que nous joignions l'extrémité supérieure de la feuille à son extrémité inférieure. Le résultat serait une surface sous la forme d'un tube horizontal. Si ensuite - et en supposant que la feuille soit faite d'un matériau flexible – l'on courbait le tube jusqu'à ce que ses deux extrémités soient jointes, le résultat serait un beignet creux (un tore). Une belle animation de cette construction peut être consultée sur la page Web (page consultée de 2 novembre 2020):
http://mathematica.stackexchange.com/questions/42493/morphing-a-sheet-of-paper-into-a-torus.

Lorsque le monde NetLogo est configuré avec la topologie du tore (la topologie par défaut), il ne faut pas oublier que le carré que nous voyons à l'écran est un tore ouvert au moyen de deux découpes: une première section transversale, qui le transforme en tuyau et une

deuxième section longitudinale qui ouvre ledit tube et le convertit en un rectangle ou un carré, en fonction des dimensions du tube. Gardez à l'esprit que la surface d'un tore n'a ni bords ni coins, contrairement au carré qui le représente à l'écran. Cependant, il ne s'agit pas de bords réels, mais de bords apparents. Si nous quadrillons un tore, chaque parcelle de la grille a huit voisines, quatre sur les côtés et quatre autres aux coins, car la surface d'un tore n'a ni côtés ni coins. Lorsqu'une région carrée est quadrillée, comme une feuille de papier, nous observons que les parcelles des bords et des angles ont moins de voisines que les parcelles de l'intérieur.

• Topologie du cylindre ou du tube. Le monde acquiert la forme d'un cylindre ou d'un tube si deux des côtés opposés du carré représentant le monde sont autorisés à se rejoindre, tandis que les deux autres côtés ne le sont pas. Par exemple, si seuls les bords supérieur et inférieur sont joints, on obtient un tube ou un cylindre tronqué aux extrémités droite et gauche, lequel cylindre a une position horizontale. Dans ce cas, les bords gauche et droit du carré sont également les bords gauche et droit du monde et les parcelles des bords ont 5 voisines, tandis que les autres parcelles du monde ont 8 parcelles voisines. Dans la tolopogie du cylindre, il n'y a pas de parcelles de coin. Si seule l'union des bords droit et gauche est autorisée, le cylindre a une position verticale.

• Topologie de la boîte ou du carré. Dans cette structure topologique, le monde est comme une vraie feuille de papier (un quadrilatère bidimensionnel) limitée par ses quatre côtés. Dans ce monde, les bords et les coins sont réels. Lorsqu'une tortue atteint l'un des quatre bords, elle ne peut plus continuer à avancer car elle a atteint la frontière du monde. Dans cette structure topologique, les parcelles des bords ont 5 voisines, celles des coins seulement 3 voisines et les parcelles intérieures ont 8 voisines.

On passe d'une topologie à l'autre en cliquant sur le bouton «Settings» (Configuration) dans la barre des opérations et en cochant ou décochant les cases «World wraps horizontally» (Monde sans limite horizontale) ou «World wraps horizontally» (Monde sans limite verticale). Lors du démarrage de NetLogo, les deux cases sont marquées, ce qui donne au monde la topologie du beignet. La topologie du cylindre est obtenue en décochant l'une des deux cases et celle de la cage en décochant les deux cases. La figure ci-dessous (Fig 2.8) illustre la fenêtre de configuration du monde qui engendre une topologie de tore.

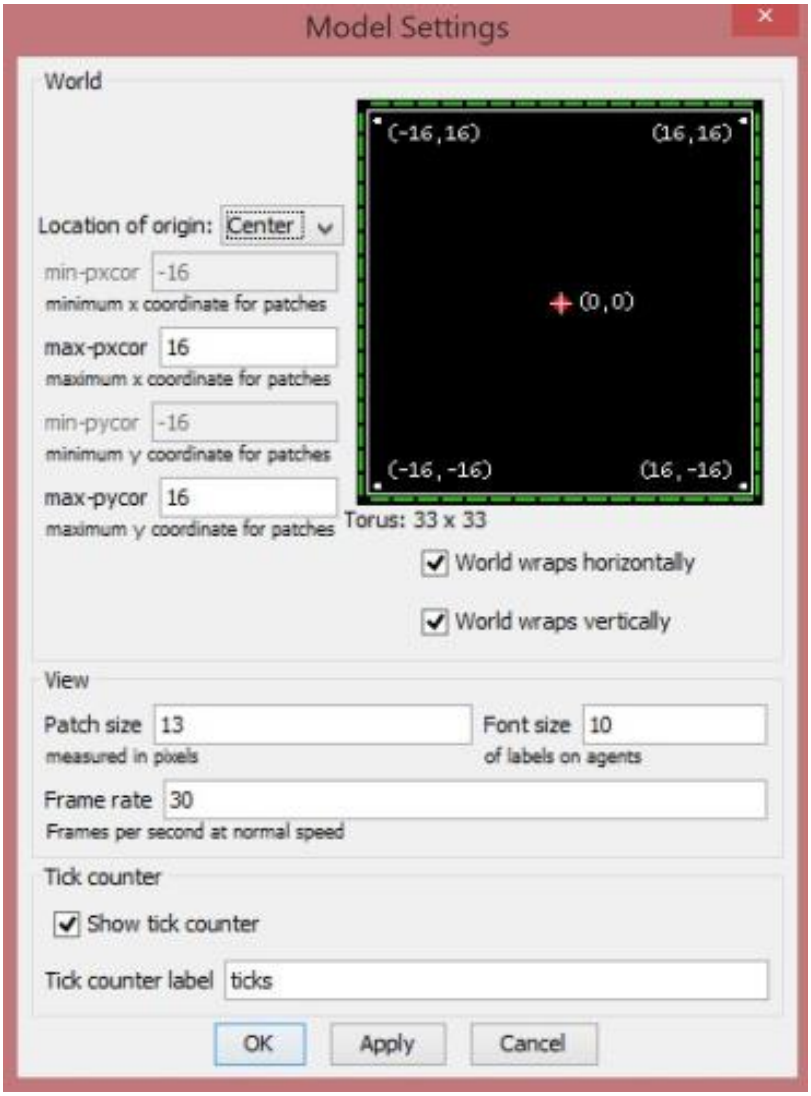


Figure 2.8 : Fenêtre de configuration du monde

Il faut noter que la topologie du monde peut affecter la mesure des distances et des angles. La distance entre deux agents - parcelles ou tortues - est toujours mesurée en fonction de la trajectoire la plus courte. Avec la topologie du beignet, par exemple, si une tortue est très proche du «bord apparent» à droite et une autre très proche du «bord apparent» à gauche, le chemin le plus court entre les deux sera celui qui sort par le «bord apparent» à droite et entre par le bord gauche. Les orientations des tortues peuvent également être affectées par la structure topologique du monde. Si une tortue est invitée à regarder (s'orienter) en direction d'une autre tortue ou d'une autre parcelle, elle le fera dans la direction du chemin le plus court. C'est au modélisateur que revient la décision de choisir le type de topologie appropriée au modèle qu'il veut développer.

Exemple 13: une mouche prise dans une cage

Primitives: random (hasard), lt (abrev. de «left» - gauche), wait (attendre).
Autres détails: la topologie du monde par défaut est remplacée par une topologie de cage (carré).Pour ce faire, on décoche les cases «world wraps horizontally» et «world wraps vertically» dans «settings»). On construit les boutons «go» et «setup» et on coche la case «forever» (en continu) du bouton «go».

Dans cet exemple une tortue adopte le comportement d'une mouche verte qui évolue à l'intérieur de son monde en forme de cage (boite carrée bidimensionnelle). Le vol de la mouche suit une trajectoire aléatoire et la mouche ne peut traverser aucune des frontières de son monde du fait que la topologie du monde est une topologie de cage.
Le code est le suivant:

```
to setup
to setup
clear-all
  crt 1 [set color green set shape "bug" set
size 3]
end

to go
ask turtle 0 [pen-down fd 5 lt random 360]
wait 0.1
end
```

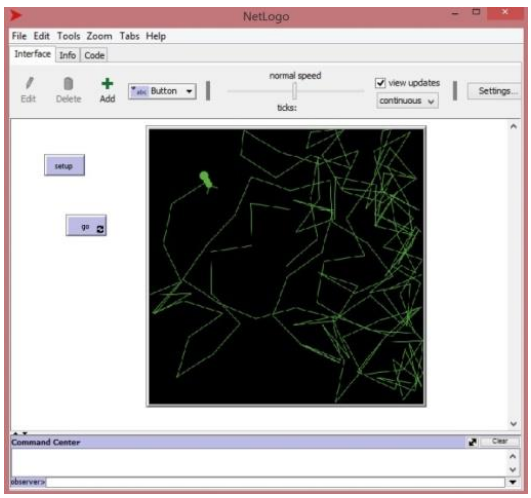


Figure 2.9 : Mouche à l'intérieur d'une cage

Commentaires et explications

On démarre le programme en appuyant d'abord sur le bouton «setup» et ensuite sur le bouton «go». Si l'on veut stopper le vol de la mouche, on appuie à nouveau sur le bouton «go». La primitive «wait num» a déjà été utilisée dans des exemples antérieurs dans le même but que dans cet exemple c'est-à-dire modérer la vitesse de la mouche.
Notez que lorsque la mouche (tortue) atteint le bord du monde avec une orientation perpendiculaire au bord et qu'on lui demande d'avancer, elle ne peut le faire car la topologie du monde est celle d'une boîte. NetLogo ne génère cependant pas de message d'erreur et la procédure passe à l'instruction suivante attribuant à la tortue une nouvelle orientation.

Le type de topologie du monde peut avoir une influence sur la valeur de la distance entre deux agents ou sur l'orientation d'un agent quand on lui demande de regarder en direction d'un autre agent.

Deuxième rencontre avec les variables : Variables créées par les utilisateurs

De nombreux modèles exigent que les utilisateurs puissent définir leurs propres variables pour stocker les valeurs des attributs dont le modèle a besoin. Par exemple, dans un modèle où la survie d'un ensemble d'agents représentant des moutons dépend de la quantité d'herbe trouvée lorsqu'ils errent sur une terre aride, il est utile d'attribuer à chaque mouton une variable dont la valeur est l'énergie vitale que celui-ci possède à chaque instant. À la naissance du mouton on attribue une valeur initiale à cette variable, valeur qui augmente chaque fois que le mouton mange un peu d'herbe et diminue lorsque le mouton erre sans pouvoir trouver d'herbe. Une fois que l'utilisateur a créé une variable, il peut lui attribuer des valeurs, consulter ou modifier ces valeurs et même supprimer la variable.

Le nom et la valeur initiale d'une variable sont attribués au moment de sa création. Par défaut, NetLogo attribue à chaque variable créée par l'utilisateur une valeur initiale de zéro, quel que soit le type de données qui caractérisent cette variable. Les noms attribués aux variables doivent nécessairement être composés d'un seul mot (les mots composés comptant pour un seul mot). Une bonne pratique en programmation consiste à attribuer à chaque variable un nom qui évoque l'utilisation prévue de la variable. Pour faciliter cet objectif, les noms composés peuvent être formés en collant plusieurs mots avec des points, des traits d'union et d'autres caractères. Si, par exemple, la variable a pour but de stocker des numéros d'immatriculation de voiture, son nom pourrait bien être

«ImmatriculationDeVoiture» ou encore «immatriculation-de-voiture». Si, par exemple, nous donnions à cette variable le nom «im», nous risquerions facilement d'oublier l'utilisation de la variable et les valeurs qui y sont stockées.

Variables globales et variables-agents.

Variables globales

Les variables qui peuvent être manipulées par n'importe quel agent à tout moment sont appelées «variables globales». On peut imaginer que les variables globales appartiennent à l'Observateur, lequel permet à tout agent de créer, consulter, modifier et même éliminer la variable. Les variables globales doivent être définies au début du code, avant les procédures en utilisant la primitive «globals», de la manière suivante:

globals [variable1 variable2 ....]

Lorsqu'il y a plusieurs variables globales, elles sont toutes incluses à l'intérieur des crochets de la primitive «globals» et séparées par des espaces. Une autre façon de créer des variables globales est par l'intermédiaire de la construction d'objets tels que curseurs, interrupteurs, etc. Dans ce cas il n'est pas nécessaire de définir ces variables au début du code.

Variables-agents

Les ensemble-agents peuvent aussi avoir leurs propres variables, que nous appellerons «variables-agents»[19]. Seuls les membres de l'ensemble-agents («agent-set» en anglais) qui est le propriétaire des variables ont le droit de manipuler, sans restriction, ces variables. Les variables de l'ensemble-agents sont également déclarées au début du code, avant les procédures qui font référence à cet ensemble. La forme générique de déclaration d'une variable-agents est «nom-own [variable1 variable2]» ou «nom» est le nom donné à l'ensemble-agents et «own» se traduit par «propre à». Par exemple, la déclaration des variables «âge», «taille» et «poids» appartenant à l'ensemble-agents «turtles» (tortues) s'exprime au moyen de l'expression:

turtles-own [âge taille poids]

[19] Ce nom n'est pas standardisé mais nous l'avons introduit par souci de clarté

Contrairement aux variables globales, dont la valeur est la même pour tous les agents, une variable appartenant à un ensemble-agents peut prendre des valeurs différentes, propres à chacun des agents de l'ensemble. Ainsi, pour reprendre l'exemple précédent de l'ensemble-agents composé de tortues, chaque tortue est caractérisée par son propre âge, sa propre taille et son propre poids.

Exemple 14: Création, consultation et modification de variables globales

Primitives : «globals», «set », «print».

Dans cet exemple, on crée d'abord deux variables globales appelées «épargne-Lydia» et «épargne-Pierre» dans l'éditeur de code. Ces deux variables globales représentent le solde des comptes d'épargne de Lydia et de Pierre. Avec seulement la définition des variables et sans écrire de procédure supplémentaire, il est possible de consulter ou de modifier leurs valeurs en écrivant les instructions appropriées dans la fenêtre de l'observateur. Dans l'éditeur de code, nous écrivons:

globals [épargne-Lydia épargne-Pierre]

Ensuite, dans la fenêtre de l'observateur, nous écrivons les commandes suivantes:

print Épargne-Lydia
==> 0, c'est la valeur initiale assignée par le système à la variable épargne-Lydia.
set épargne-Lydia 5000, ici on assigne une nouvelle valeur à épargne-Lydia.
print épargne-Lydia
==> 5000
print épargne-Pierre
==> 0
set épargne-Pierre 300
print épargne Pierre
==> 300
set épargne-Pierre épargne-Lydia, assigner à épargne-Pierre la valeur épargne-Lydia.
print épargne-Pierre
==> 5000

set épargne-Lydia épargne-Lydia + épargne-Pierre / 2
print épargne-Lydia
==> 7500

Exemple 15 : Différence entre variables globales et variables-agents

Primitives: globals, turtles-own (propre-aux-tortues), set (assigner).
Autres détails: Exécuter les deux procédures de cet exemple à partir de la fenêtre de l'observateur.

Les variables globales ont une valeur qui peut être consultée et modifiée par n'importe quel agent, que ce soit une tortue, une parcelle ou un lien. Si, par exemple, à un moment donné, une variable globale appelée «température» a la valeur 30, tout agent qui consulte la valeur de cette variable obtient la valeur 30. Si un autre agent changeait la valeur de la variable «température» à 25 alors tout agent qui consulterait la valeur de ladite variable obtiendrait la valeur 25. Les variables d'agent, générées avec des instructions du type «turtles-own [..] ou «patches-own» […], ont cet avantage que chaque agent peut attribuer une valeur différente à la variable, valeur qui n'est pas modifiée par les changements effectués par les autres agents. L'exemple suivant compare le comportement de la variable globale «température» à celui de la variable d'agent «âge».

```
globals [température]
turtles-own [âge]

to ma-température
clear-all
  crt 5 [fd 15]
  ask turtles [set température random 100]
  ask turtles [write température]
end

to mon-âge
  clear-all
  crt 5 [fd 10]
  ask turtles [set âge random 100]
  ask turtles [write âge]
end
```

Explications et commentaires supplémentaires.

Exécutez la procédure «ma-température» à partir de la fenêtre de l'observateur et observez comment la même valeur est obtenue pour tous les agents. En fait, l'instruction «ask turtles [set température random 100]» fait en sorte, qu'une après l'autre, les tortues fixent la valeur de la variable «température» au hasard. Chaque fois qu'une tortue définit la valeur, cette valeur change, mais elle s'applique à tous les agents. Par exemple, si la tortue 0 choisit aléatoirement la valeur 10, toutes les tortues ont une température égale à 10, si la tortue suivante choisit la valeur 62, toutes les tortues ont une température de 62 et ainsi de suite. Lorsque l'interpréteur passe à la commande suivante, «ask turtles [write température]» toutes les tortues possèdent la valeur choisie par la dernière tortue de la liste (nous ignorons cependant l'identité de cette tortue car l'interpréteur ordonne de façon aléatoire les membres de l'ensemble-agents chaque fois qu'il les invoque) .Par contre, lors de l'exécution de la procédure «mon-âge», on observe que dans ce cas les tortues ont des âges différents car à chaque fois qu'une tortue choisit la valeur de son âge, cette valeur lui est propre et n'est pas modifiée lorsque la tortue suivante dans la liste choisit la valeur de son âge.

Une variable globale stocke une valeur unique, tandis qu'une variable-agents stocke une valeur pour chaque agent. Cela permet d'affecter différentes valeurs aux agents.

Le type d'une variable.

En programmation, on appelle «type» de variable, le genre de données stockées dans cette variable. Les types les plus courants sont les suivants: entier, flottant (nombres décimaux), booléen (vrai ou faux), liste et chaîne. Dans NetLogo, on doit ajouter le type ensemble-agents («agent-set» en anglais). Dans de nombreux langages de programmation, en particulier ceux d'usage général, chaque fois que l'on crée une variable, le langage oblige le programmeur à indiquer le type de la variable

Le but de cette exigence est de permettre au système de réserver la quantité de mémoire appropriée au type auquel appartient la variable et d'être ainsi en mesure de mieux gérer les ressources de la machine (par exemple, le stockage d'un nombre décimal requiert davantage d'espace que celui d'un nombre entier). Comme nous l'avons vu, il n'est pas nécessaire, dans NetLogo, de déclarer le type de données censé être stocké dans les variables. Il existe cependant, comme nous le verrons ultérieurement, une exception à cette règle dans le cas où l'on utilise certaines fenêtres créées pour la saisie de données.

Expressions conditionnelles

Dans le langage ordinaire, nous trouvons souvent des expressions du type «**SI** cet évènement se produit **ALORS** ceci doit arriver (ou doit être fait)», expressions appelées expressions conditionnelles. Un langage de programmation qui n'aurait pas la capacité de gérer des expressions conditionnelles serait un langage d'utilisation très limité. Les expressions conditionnelles permettent à un langage de gérer des alternatives, où le flux du programme bifurque, prenant un chemin ou un autre selon qu'une condition C est remplie ou non. Dans NetLogo, les expressions conditionnelles sont construites à partie de deux primitives : la primitive «if» et la primitive «ifelse». Le mot anglais «if» correspond au mot français «si» utilisé dans une phrase telle que «si tu viens à la fête, alors je pourrai te présenter mon amie». Le mot «ifelse» n'est pas vraiment un mot anglais. C'est une fusion des mots «if» (si) et «else» («sinon», «dans l'autre cas»), qui dans certaines versions du langage Logo a été traduite en français par «sisinon». La syntaxe de la primitive «if» est la suivante:

if (condition) [commandes si la condition est remplie] commandes suivantes…

L'interpréteur examine si la condition est remplie, auquel cas la condition indique la valeur «true» (vrai). Si la condition n'est pas remplie, c'est la valeur «faux» qui est rapportée. Lorsque la condition est remplie, l'interpréteur exécute les commandes entre crochets, puis les autres commandes de la procédure, c'est-à-dire celles qui se trouvent après les crochets. Si la condition n'est pas remplie, l'interpréteur ignore les commandes entre crochets et exécute celles qui suivent. Par exemple, examinons la commande:

ask turtle 1 [if size < 2 [set color red] fd 4 lt 90

qui se traduit par «si la valeur de la variable «size» (taille) de la tortue 1 est inférieure à 2 [coloriez la tortue 1 en rouge], puis faites la avancer de 4 pas et faites la tourner de 90 degrés vers la gauche». Si la condition n'est pas remplie, l'interpréteur saute les crochets et passe directement aux commande «fd 4», «lt 90» et à celles qui suivent éventuellement. Une expression ou une variable qui rapporte les valeurs «true» ou «false» (vrai ou faux) est appelée expression ou variable «booléenne» (en l'honneur du mathématicien et philosophe anglais George Boole (1815-1864), qui a largement contribué à la logique et aux mathématiques). La primitive «ifelse» a un format très similaire à «if», à la différence qu'il y a deux crochets contenant des commandes, l'un pour le cas où la condition est remplie, l'autre pour le cas où elle n'est pas remplie. Étant donné le caractère binaire de la condition (elle est remplie ou elle ne l'est pas)[20],

[20] Principe *Tertium non datur* (príncipe du tiers-exclu) de la logique classique.

l'interpréteur exécutera toujours les commandes situées dans l'un des deux crochets, mais jamais les deux. La syntaxe de «ifelse» est la suivante:

ifelse (condition) [commandes si la condition est remplie] [commandes si la condition n'est pas remplie] commandes qui suivent l'expression conditionnelle ifelse ...

Ainsi, par exemple la commande :

ask turtles [ifelse size < 2 [set color blue][set color yellow] fd 3 rt 90]

se traduit par « si la valeur de la variable «size» des tortues est inférieure à 2 [coloriez les tortues en bleu], [sinon coloriez les en jaune] puis faites avancer toutes les tortues de 3 pas et faites les tourner de 90 degrés vers la droite.

Exemple 16: Créer des leurres pour échapper à une poursuivante

Primitives: if (si conditionnel), distance (distance), sprout (générer), stamp (estampiller), set (assigner), heading (orientation).
Autres détails: on construit une expression conditionnelle avec la primitive «if».

Bien que le but d'un programme soit d'illustrer un exemple d'utilisation d'une primitive ou d'un concept, il est toujours plus agréable de relier l'exemple à une histoire, même fictive. L'histoire de cet exemple est la suivante: deux tortues l'une de couleur rouge, l'autre blanche parcourent le monde de manière aléatoire. La tortue rouge pense que la blanche la poursuit et, pour l'induire en erreur, elle conçoit une stratégie consistant à imprimer des images inertes d'elle-même pour tromper la tortue blanche. Chaque fois que la distance qui sépare les deux tortues ne respecte pas un certain écart entre les deux tortues, la tortue blanche étampe une image d'elle-même (un leurre). Le bout de code correspondant à cette action conditionnelle s'écrit: «if écart < 5 [stamp]», ce qui signifie «si vous vous approchez de moins de 5 pas de moi, alors [je laisse une empreinte de mon image]». Les images créées avec «stamp» (estampiller) ont la même apparence que la tortue qui les génère, mais ce ne sont pas des agents mais seulement des images inertes. Dans un tel contexte, le monde se remplit d'empreintes immobiles tandis que les tortues continuent à déambuler. On peut, à tout moment, arrêter le programme en appuyant à nouveau sur le bouton «go» (voir figure 2.10 pour un résultat de cette simulation après quelques secondes).

Activités préparatoires : créez les boutons setup et go et cochez la case «Forever» de ce dernier.
Voici le code :

```
globals [écart]

to setup
clear-all
ask patch 10 0 [sprout 1] ;; de la parcelle 10 0 émerge une tortue
ask turtle 0 [set color white]
ask patch -10 0 [sprout 1 ] ;; de la parcelle -10 0 émerge une
tortue
ask turtle 1 [set color red]
end

to go
ask turtle 0 [set heading random 360 fd 2]
ask turtle 1
 [
  set heading random 360 fd 2
  set écart distance turtle 0
   if écart < 5 [stamp]
 ]
wait 0.1
end
```

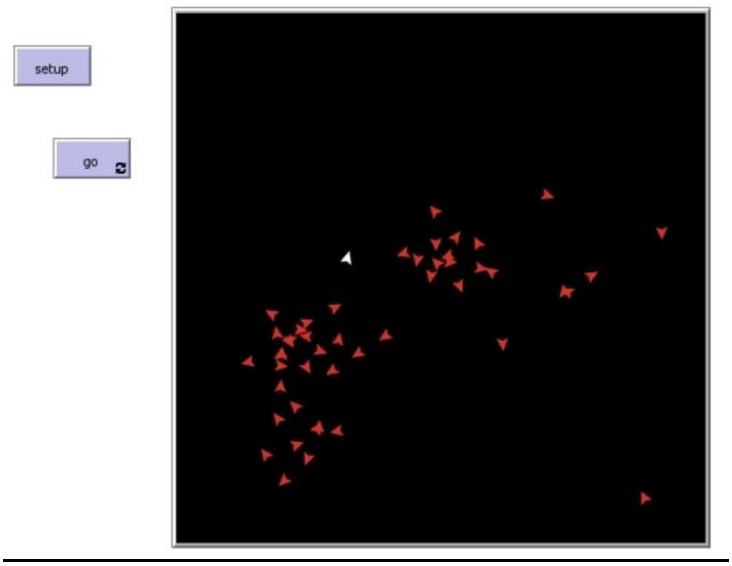


Figure 2.10 : Si conditionnels et création de leurres de même couleur

Commentaires et explications. La commande «set heading random 360» attribue aléatoirement à la tortue 0 une orientation mesurée par un angle entre 0 et 259 degrés. Notez que même si ce que l'on cherche à mesurer avant de planter un leurre est de mesurer la distance qui sépare les deux tortues, on ne peut pas utiliser le mot «distance» à la place du mot «écart» (voir procédure «go» ci-dessus). En effet, alors que le mot «écart» est une variable globale définie par l'utilisateur, le mot «distance» est une primitive (mot réservé) de NetLogo qui mesure la distance entre deux tortues. Cette primitive est d'ailleurs utilisée dans le code de la procédure go ci-dessus. Elle rapporte la distance qui existe entre l'agent qui place la commande et celui qui la reçoit comme entrée. Dans cet exemple, on demande à la tortue 1 «ask turtle 1 [... set écart distance turtle 0]» de fixer la valeur de la variable «écart» égale à la valeur reportée par la primitive «distance» c'est-à-dire égale à la distance qui la sépare de la tortue 0. Comme la commande se trouve entre les crochets de la tortue 1, c'est cette dernière qui étampe sa figure. C'est la tortue 1 qui fuit la tortue 0.

Chapitre 3: Environnement et langage III

Troisième rencontre avec les variables : Variables locales

Dans cette troisième rencontre avec les variables, nous introduirons les *variables locales*. L'idée de base d'une variable locale est de limiter son existence à un certain contexte, à la fois dans l'espace et dans le temps. En dehors de ce contexte, la variable cesse d'exister, libérant ainsi l'espace mémoire qu'elle occupait. Les variables locales ne sont pas déclarées au début du code, comme les variables globales, et sont créées dans le code avec la primitive «let» (permettre), suivie du nom de la variable et de sa valeur. Par exemple, pour créer une variable locale appelée «énergie» avec une valeur initiale de 100, nous devons écrire l'instruction suivante: «let énergie 100» (*permettre* à la variable *énergie* de prendre la valeur *100*). Par le biais de cette commande, la variable «énergie» est créée et une valeur lui est affectée. Dans l'exemple précédent (exemple 16) «Créer des leurres pour échapper à une poursuivante», il aurait été plus économique (en terme d'espace mémoire) de faire de la variable «écart» une variable locale plutôt que globale. Pour cela, il suffit d'éliminer l'expression «globals [écart]» au début du code et de remplacer «set» par «let» dans la procédure «go»:

```
ask turtle 1 [set heading random 360 fd 2
let écart distance turtle 0 ;; «set» est remplacé par «let
if écart < 5 [stamp] ]
```

Avec cette modification, la variable «écart» devient une variable locale et n'existe qu'entre les crochets de l'instruction «ask turtle 1 [...]». Si, après l'exécution de la procédure, nous écrivons, dans la fenêtre de l'observateur, la commande «ask turtle 1 [show écart]», nous obtenons le message d'erreur «Nothing named ÉCART has been defined»(aucune variable nommée ÉCART n'a été définie), ce qui indique que la variable a déjà été éliminée. Par contre, si la variable «écart» était restée globale, ce message d'erreur ne serait pas apparu car ladite variable n'aurait pas cessé d'exister après l'exécution de la procédure.

Familles d'agents («breeds»)

NetLogo permet de créer certains ensembles-agents appelés «breeds» que nous traduirons en français par le mot «familles»[21] . Il est possible de créer des familles de tortues ou de liens, mais pas des familles de parcelles. Les familles doivent être constituées par des agents du même type. Par exemple, une famille de tortues appelées «loups» serait un sous-ensemble d'agents de l'ensemble-agents «turtles» (tortues) et répondrait aux commandes spécifiquement adressées aux loups. La création de familles facilite la manipulation informatique d'agents appartenant à des groupes dont les caractéristiques ou les comportements sont différents (les membres du groupe «chèvres» présentent des différences marquées avec ceux du groupe «loups»). Les familles doivent être déclarées au début du code, avant les procédures. Les familles de tortues sont déclarées à l'aide de la primitive «breed» et les familles de liens le sont avec les primitives «directed-link-breed» ou «undirected-link-breed» selon que le lien est de type orienté ou non. Dans la déclaration des familles, il est nécessaire d'inclure le nom des membres de la famille au pluriel ainsi que leur nom au singulier, entre crochets. Par exemple, si nous voulons créer une famille de tortues appelées «loups», elle doit être déclarée sous la forme:

```
breed [loups loup]
```

De même pour créer une famille de liens orientés on écrira :

```
directed-link-breed [chemins chemin]
```

On ne peut inclure qu'une seule famille à l'intérieur des crochets de la primitive «breed». Si l'on veut créer deux familles ou plus, chacune d'elles doit être déclarée séparément. Nous aurons l'occasion de voir comment utiliser les familles dans plusieurs exemples du livre. La raison pour laquelle l'inclusion des noms au singulier et au pluriel est requise est que NetLogo permet d'utiliser des expressions avec la version singulière ou plurielle du nom, selon les besoins. Par exemple, une fois que la famille «loups» est créée, vous pouvez utiliser des expressions telles que «ask loups [..commandes..]» ou bien «ask loup 3 [..commandes..]». Bien que NetLogo ne permette pas la création de familles de parcelles, il est

[21] La traduction française littérale du mot «breed» est «race». Comme le mot anglais «breed» s'applique aux animaux et non aux êtres humains nous avons opté pour le mot «famille» qui a une portée plus générale (nous aurions aussi pu opter pour le mot «groupe» de portée encore plus générale).

possible de créer des ensembles-agents de parcelles à l'aide de primitives auxiliaires telles que «with» ou «neighbors», ce qui compense cette lacune.

Variables propres à une famille. Une caractéristique très avantageuse des familles est le fait qu'elles peuvent avoir leurs propres variables. Ceci est vrai autant pour les tortues que pour les liens. Par exemple il est possible de créer des variables propres à la famille des chemins (exemple de lien orienté donné ci-dessus) avec l'expression «chemins-own [longueur]» ou la longueur est une variable propre aux chemins. De même, on peut créer des variables propres à la famille des loups avec l'expression «loups-own [... variables ...]».Cependant, il est nécessaire de préciser que les familles ainsi crées sont des sous-ensembles de l'ensemble plus large auxquelles elles appartiennent. Ainsi, par exemple, la famille des loups est un sous-ensemble d'agents de l'ensemble-agents «turtles» ce qui implique que les loups ne perdent pas la propriété d'appartenir à l'ensemble-agents «turtles». En ce qui concerne l'attribution des numéros «who», l'interpréteur attribue un numéro courant à toutes les tortues au fur et à mesure de leur création, quelle que soit la famille à laquelle elles appartiennent. Pour cette raison, les numéros des membres d'une famille ne sont pas nécessairement consécutifs car ces numéros dépend du moment où chacun des membres a été créé. L'exemple qui suit (exemple 17) aidera à clarifier les choses.

Exemple 17: Deux familles

Dans cet exemple, on crée une famille de loups ainsi qu'une famille de chèvres et l'on assigne deux variables à chacune des deux familles. La procédure «deux-familles» est invoquée à partir de la fenêtre de l'observateur et certaines commandes sont testées dans cette même fenêtre. Dans cet exemple il n'est pas nécessaire de créer un bouton «go» dans l'interface.

Voici le code du modèle:

```
breed [loups loup]
breed [chèvres chèvre]
loups-own [énergie vitesse]
chèvres-own [resistance poids]

to deux-familles
clear-all
```

crt 10
create-loups 10
create-chèvres 10
ask loups [set énergie random 100 set vitesse 40]
ask chèvres [set resistance 50 set poids (random 10) + 20]
;; le poids des chèvres varie entre entre 20 y 29 kilogs.
ask loup 12 [show énergie]
ask chèvre 20 [show resistance]
end

Invoquons la procédure «deux-familles» à partir de la fenêtre de l'observateur
deux-familles
==> (loup 12): 27 (le niveau d'énergie du loup 12 est égal à 27)
==> (chèvre 20): 50 (la résistance de la chèvre est égale à 50)

Passons maintenant des commandes isolées (c'est à dire non programmées dans la section «code» à partir de la fenêtre de l'observateur) :

ask turtle 1 [show énergie]
==> la famille TURTLES ne possède pas de variable ÉNERGIE
ask loup 8 [show énergie]
==> turtle 8 is not a LOUP (erreur : la tortue 8 n'est pas un loup, c'est une tortue).

ask chèvre 23 [show resistance]
==> (chèvre 23): 50
ask turtle 23 [show resistance]
==> (chèvre 23): 50

ask loup 15 [show poids]
==> **la famille LOUPS ne possède pas de variable nommée POIDS**
ask chèvres [forward 5] (les 10 chèvres avancent de 5 pas).
ask turtles [set color yellow] (les 30 tortues deviennent jaunes)

<u>Explications et commentaires supplémentaires</u>.

Parce que les tortues sont les premières à être créées elles reçoivent les numéros «who» de 0 à 9. Les loups reçoivent les numéros de 10 à 19 et les chèvres de 20 à 29. Les tortues 0 à 9 n'ont pas d'accès direct aux variables des loups ou des chèvres. Du fait que les membres d'une famille, par exemple, la famille «loups» sont des tortues «déguisées» en loups, il est possible de faire référence à chaque loup de deux manières différentes: soit en tant que loup soit en tant que tortue pour autant que, dans les deux cas, on utilise le même numéro «who». Ceci explique pourquoi l'instruction «ask turtle 23 [show resistance]» équivaut à l'instruction «ask chèvre 23 [show resistance]». Les agents «turtle 23» et «chèvre 23» sont le même agent. Par contre, l'instruction «ask turtle 23 [show vitesse]» produit le message d'erreur «la famille CHÈVRES ne possède pas de variable VITESSE» car l'observateur sait que la tortue 23 n'est pas un loup mais une chèvre. Une autre façon de considérer cette façon de voir est la suivante : les tortues numérotées de 0 à 9 sont des «tortues pures», celles numérotées de 10 à 19 sont des «tortues-loups» et celles numérotées de 20 à 29 sont des «tortues-chèvres». Mais du point de vue des groupes d'agents il n'y a que trois familles:

• tortues (30 membres): tortues numérotées de 0 à 29.
• loups (10 membres): les tortues numérotées de 10 à 19.
• chèvres (10 membres): tortues numérotées de 20 à 29.

La variable «ticks»

NetLogo contient une variable globale préinstallée dont le nom est «ticks». L'usage le plus fréquent de cette variable est de marquer le passage du temps en comptant le nombre de fois qu'une procédure se répète. La commande «tick» augmente d'une unité de temps la valeur de la variable «ticks». C'est pourquoi elle figure très fréquemment dans la procédure «go» quand l'on coche la case «Forever» (continuellement) et quand il est intéressant de prendre en considération le nombre de fois que cette procédure est répétée. La variable est activée par la commande «reset-ticks» avec une valeur initiale de 0. En ce qui concerne le traitement de la variable «ticks», les trois primitives suivantes sont disponibles:

• «reset-ticks» qui initialise la variable «ticks» avec la valeur 0.
• «ticks» qui rapporte la valeur actuelle de la variable «ticks».
• «tick» qui augmente d'une unité la valeur de la variable «ticks».

Curseurs et graphiques

En plus des boutons, les curseurs et les graphiques sont deux autres objets d'une grande utilité dans la construction de modèles. Pour créer l'un ou l'autre de ces objets, on procède de la même manière qu'avec les boutons: dans la fenêtre du sélectionneur d'objets on sélectionne «Slider» (curseur) ou «Plot» (graphique), selon le cas, et on place l'objet dans un endroit commode. Les curseurs et les graphiques ont leur propre fenêtre d'édition pour assigner les propriétés qui déterminent leur comportement et leur apparence. La fenêtre s'ouvre en posant le pointeur de la souris sur l'objet et en faisant un clic droit avec la souris.

Curseurs

Dans la construction des modèles, il est souvent nécessaire de définir les variables et paramètres d'entrée dont les valeurs influencent directement le comportement du modèle. Il est intéressant de pouvoir faire varier ces valeurs pour étudier la réponse du modèle à ces variations. La possibilité d'introduire ces changements des valeurs dans le modèle est facilitée par la construction de curseurs (qui sont l'un des objets du menu déroulant du sélectionneur d'objets). Ces objets ont une barre (horizontale) de défilement qui sert à modifier la valeur de la variable qu'ils représentent. Les variables ainsi définies sont de type global mais *ne doivent pas* être déclarées dans le code par le biais de la primitive «globals [...]». Prenons comme exemple un curseur qui définit la variable globale «temperature». Une fois créé ce curseur ressemble à la figure ci-dessous (Figure 3.1) :

Figure 3.1 : Exemple de curseur pour une variable nommée «température»

Dans la fenêtre du curseur, on doit indiquer: le nom de la variable globale, les valeurs minimum et maximum que prend la variable, la taille minimale des incrémentations de sa valeur initiale, les unités (facultatif) et la position verticale ou horizontale du curseur comme le montre la figure suivante (Figure 3.2) :

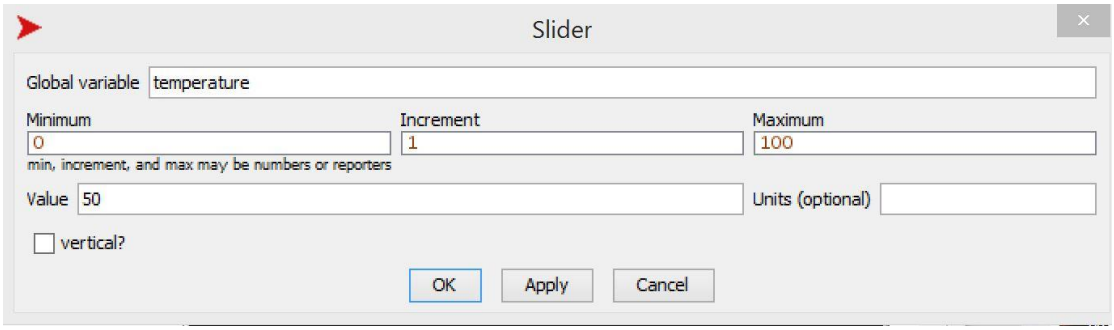


Figure 3.2 : Fenêtre du curseur pour une variable nommée «temperature»

Graphiques

Comme les boutons et les curseurs, les graphiques sont également l'une des options disponibles du menu déroulant du sélectionneur d'objets. L'exemple ci-après (exemple 18 : «Une population fluctuante») illustre la création d'un graphique qui montre l'évolution temporelle d'une population de tortues ainsi que les caractéristiques de la fenêtre du graphique.

Exemple 18: Une population fluctuante

Primitives : reset-ticks (réinitialiser les «ticks» - unités de temps-), tick (unité de temps), die (mourir), sprout (générer), count (compter).
Autres détails : on utilise des expressions conditionnelles, on construit un curseur et un graphique

Dans cet exemple, nous créons une population de tortues dont le nombre initial d'individus est représenté par la variable «population-initiale». La construction d'un curseur permet de modifier, à l'intérieur de certaines limites, la valeur de cette variable. Dans le présent exemple, les tortues empruntent des trajectoires aléatoires et meurent lorsqu'elles se rapprochent du bord droit du monde («if xcor> 14 [die]»), ce qui entraîne une diminution de la population au fil du temps. Lorsque la population compte moins de 10 individus, un nombre de tortues égal à la population initiale émerge d'une parcelle (patch 0 0): if count turtles < 10 [sprout population-initiale] et le cycle se répète. Les fluctuations dans le temps de la population de tortues sont représentées sur un graphique dont l'axe des ordonnées Y mesure le nombre de tortues vivantes et celui des abscisses X le temps.

Activités préparatoires : Construisez les boutons «setup» et «go» cochez la case «Forever» de ce dernier. Construisez également : 1) un curseur pour représenter la variable globale «population-initiale», 2) un graphique pour représenter l'évolution dans le temps du nombre de tortues vivantes et 3) un compteur qui indique le nombre de tortues vivantes à tout moment

Dans la fenêtre d'édition du curseur (Figure 3.3) indiquez le nom de la variable («population-initiale»), les nombres minimum (par exemple 5) et maximum (par exemple 200) de tortues qui composent la population initiale. Cette même fenêtre indique que l'incrément est de 10 et que la valeur de départ de la simulation est 50.

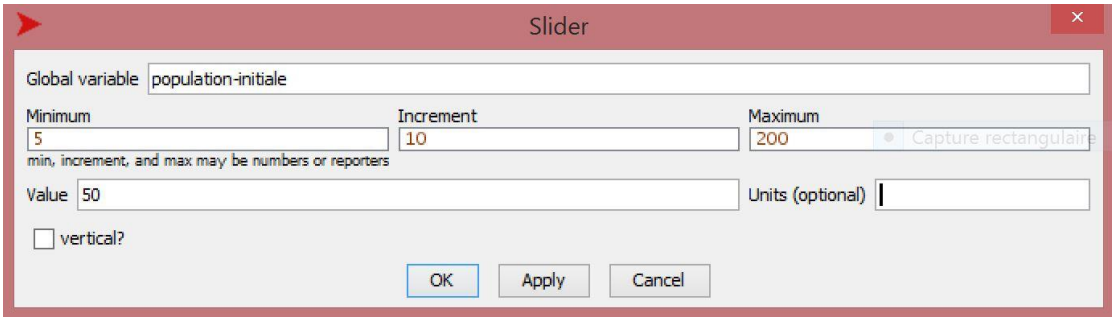


Figure 3.3 : Fenêtre d'édition du curseur nommé «population-initiale»

Ouvrez la fenêtre d'édition du graphique (Figure 3.4) et inscrivez les valeurs suivantes: 1. Name (Nom) : Tortues vivantes, 2. X axis label (titre de l'axe des X): unités de temps, 3. Y axis label (titre de l'axe des Y): tortues vivantes, 4. Cochez la case «Auto scale ?» (Echelle automatique?), 5. Pen update commands (commandes de mise à jour du stylet) : plot count turtles (tracer l'évolution du nombre de tortues).

Les valeurs des autres champs de la fenêtre ne sont pas modifiées. Lorsque la case «Auto scale ?» est cochée, Netlogo ignore les valeurs des champs X min, Y min, X max, Y max et ajuste automatiquement l'échelle du graphique afin que les valeurs des variables ne sortent pas des champs du graphique.

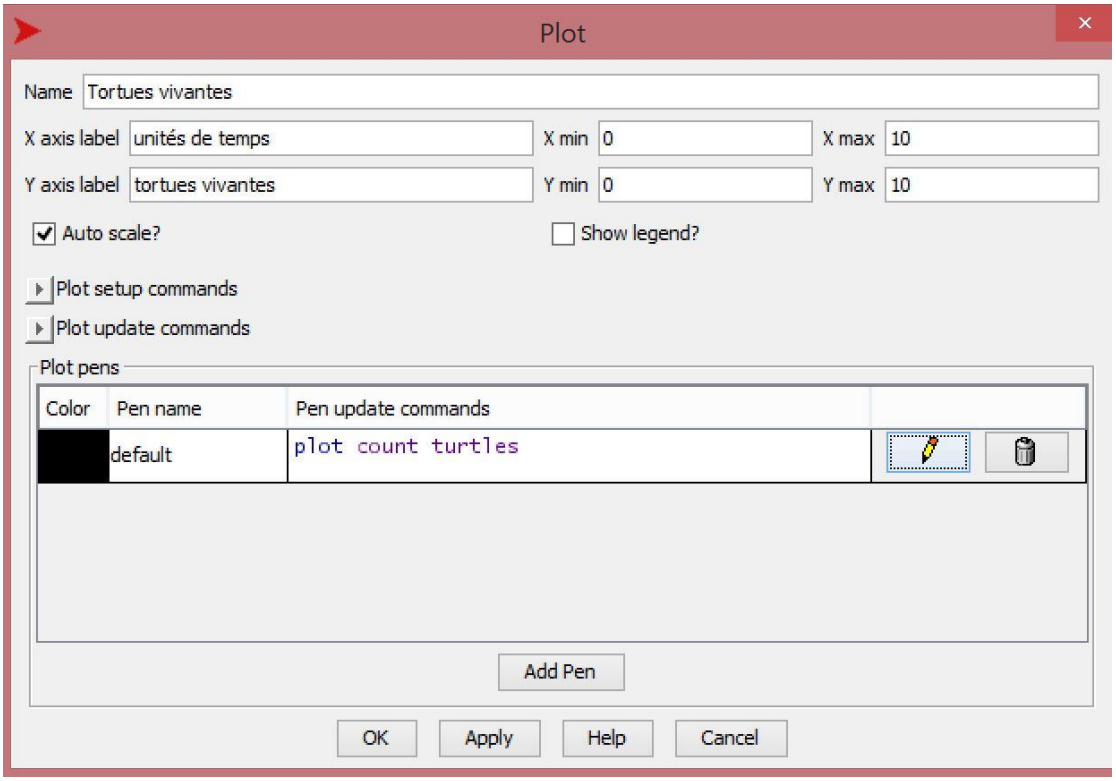


Figure 3.4 : Fenêtre d'édition du graphique

La construction d'un compteur («monitor») procède de la même manière que celle employée pour la construction des autres objets : on choisit l'objet à construire («monitor» dans ce cas) et, dans la fenêtre d'édition du compteur (Figure 3.5), on inscrit : 1) dans la case «Reporter», la commande désirée («count turtles» dans ce cas) et 2) le titre donné au compteur (tortues, par exemple). Les deux autres cases de cette fenêtre peuvent-être ignorées pour le moment.

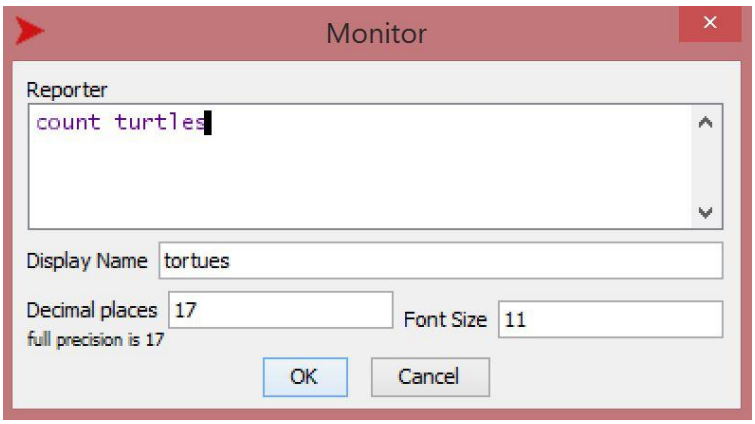


Figure 3.5 : Fenêtre d'édition du compteur

Le code est le suivant :

```
to setup
clear-all
crt population-initiale
;; on crée un certain nombre de tortues (compris entre 5 et 200)
 ;; avec le curseur intitulé  «population-initiale»
reset-ticks ;; réinitialise la variable ticks à la valeur 0
end

to go
ask turtles [fd 2 set heading random 360]
ask turtles [if xcor > 14 [die]] ;; les tortues qui s'approchent
  ;; de la bordure droite meurent
ask patch 0 0 [if count turtles < 10 [sprout population-initiale]]
  ;; si la population initiale est inférieure à 10, la population renaît
wait 0.04
tick ;; la valeur de tick augmente d'une unité de temps
end
```

Explications et commentaires supplémentaires. La primitive «count» rapporte le nombre de membres d'un ensemble-agents («agentset»). La primitive «sprout» génère un nombre de tortues égal à la valeur de la variable «population-initiale» sur la parcelle qui émet la commande (dans ce cas, la parcelle 0 0). L'orientation des tortues qui émergent de cette parcelle est aléatoire.

Procédures avec données d'entrée

Dans l'exemple précédent, nous avons vu comment assigner la valeur d'une variable à une procédure à l'aide d'un curseur dans l'interface. Nous allons maintenant voir dans l'exemple suivant (exemple 19) comment des valeurs peuvent être attribuées aux variables d'une procédure en inscrivant ces valeurs sous forme d'entrées, à droite du nom de la procédure. Les entrées de procédure se comportent comme des variables locales. Dans le code, les variables d'entrée d'une procédure - également appelée paramètres - sont déclarées à droite du nom (les lecteurs familiers avec le langage Logo se souviendront que, dans ce langage, les

paramètres d'entrée des procédures sont également déclarés à droite de la procédure, mais précédés du caractère « : » (deux points). Dans NetLogo, si la procédure comporte plusieurs entrées, elles sont toutes déclarées entre la même paire de crochets et séparées par des espaces. Lors de l'appel de la procédure, la valeur assignée à l'entrée est indiquée à droite du nom, mais, cette fois, sans inclure de crochets.

Exemple 19: Trois courtes procédures avec entrées

Primitives : setxy (assignerxy), type (écrire) Autres détails : on crée trois procédures avec des paramètres d'entrée.

On donne, ci-dessous, trois exemples de procédures comportant respectivement une, deux et trois entrées. Les procédures sont appelées à partir de la fenêtre de l'observateur.

1- Procédure avec une entrée.
La procédure suivante dessine un cercle dont la couleur dépend de l'entrée «coloration»:

Ouvrir l'onglet «Code» et y écrire le code suivant :

```
to cercle [coloration]
clear-all
crt 1 [setxy -8 8 pd set color
coloration]
ask turtle 0 [repeat 360 [fd 0.1
rt 1]]
end
```

Nous appelons la procédure dans la fenêtre de l'observateur (onglet «Interface») en donnant à l'entrée une valeur numérique, par exemple avec la couleur 15:
cercle 15, la variable *c*oloration prend la valeur 15 et la tortue dessine un cercle de couleur 15 (rouge).
cercle blue, la tortue dessine un cercle bleu. La primitive «blue» indique le nombre 105 (blue).

2- Procédure avec deux entrées.

Dans l'onglet «Code», écrire le code suivant :

```
to mariage [fiancé fiancée]
type fiancé type " et " type fiancée type ", je vous déclare mari et
femme"
end
```

Dans la fenêtre de l'observateur, nous appelons la procédure avec "Charles" et "Cécile" comme valeurs des entrées:

mariage " Charles " " Cécile "
==> Charles et Cécile, je vous déclare mari et femme

3- Procédure avec trois entrées.
Dans l'exemple ci-dessous on évalue un polynôme à trois variables x,y,z. Écrire le code suivant :

```
to polynôme [x y z]
show 5 * x * x + 2 * x * y - x * y * z
end
```

Dans la fenêtre de l'observateur, nous appelons la procédure avec x = 3, y = 2, z = 7 comme valeurs des entrées:

polynôme 3 2 7
==> observer: 15

Explications et commentaires supplémentaires: La primitive «setxy num1 num2» définit la position de la tortue avec les coordonnées (num1 num2). La primitive «type» est similaire à «show», à la différence qu'à la fin du texte, elle ne force pas le changement de ligne (retour de chariot). Cela permet au résultat de plusieurs expressions de «type» de rester sur la même ligne. Notez que l'espace est aussi un caractère et qu'il est pris en considération lorsqu'il est entre les guillemets de «type» ou de «show». Nous avons inclus un espace avant et après le mot «et».

Dans NetLogo, comme dans la plupart des langages informatiques, les conventions concernant la hiérarchie des opérations mathématiques sont respectées, ce qui évite l'utilisation excessive de parenthèses et améliore la compréhension des expressions. En raison de ces conventions, il n'a pas

été nécessaire d'écrire l'exemple de polynôme sous la forme (5 * x * x) + (2 * x * y) - (x * y * z). Si l'on veut déroger à l'utilisation de ces conventions il faut alors utiliser des parenthèses. Par exemple, l'expression (5 * x * x + 2) * ((x * y) - x) * y * z évaluée avec les mêmes entrées 3, 2 et 7 donnerait 1974 comme résultat.

Les paramètres d'entrée d'une procédure sont déclarés à la droite de son nom et sont mis entre crochets. Quand il y a plusieurs paramètres, ceux-ci sont séparés par des espaces.

Listes et chaînes («strings»)

Nous présentons ci-dessous deux notions très importantes du langage NetLogo: les listes et les chaînes («strings»). Les listes, ainsi que les chaînes, constituent les structures de données de base du langage NetLogo et des langages dérivés du langage Lisp[22]. Nous allons d'abord introduire le concept de «liste», ce qui nous permettra également d'introduire celui de «chaîne», car bon nombre de primitives utilisées pour le traitement des listes s'appliquent également aux chaînes. Une **liste** est un ensemble d'objets placés dans un certain ordre. Dans NetLogo, les membres ou les éléments d'une liste sont placés entre crochets […] ou sont écrits précédés de la primitive «list». Une **chaîne** est une séquence de caractères placés dans un certain ordre et placés entre guillemets, comme s'ils formaient un seul mot. Les espaces sont considérés comme un caractère et peuvent faire partie d'une chaîne.

Les chaînes de caractères «je-suis-une-chaine#2» et «je suis aussi une chaine» sont des exemples de chaînes de 20 et 24 caractères respectivement. Le fait que les membres d'une liste et ceux d'une chaîne soient ordonnés permet à chaque membre d'être caractérisé par la position qu'il occupe dans la liste ou la chaîne. La liste [1 2] et la liste [2 1] sont différentes, de même que les chaînes «a2» et «2a». La numérotation des membres d'une liste ou d'une chaîne commence à partir de zéro. La caractéristique la plus importante des listes est le fait que les membres d'une liste peuvent être, à leur tour, des listes. NetLogo possède un vaste répertoire de primitives permettant de manipuler les listes et les chaînes. Seules sont mentionnées ci-dessous les opérations les plus importantes qui peuvent être effectuées avec les listes:

[22] Le nom Lisp est la contraction de l'expression anglaise «list processing».

1) Créer une liste.
2) Indiquer vrai ou faux si un objet est une liste ou non.
3) Indiquer vrai ou faux si un objet est ou non membre d'une liste.
4) Identifier un membre (élément) d'une liste en fonction de la position qu'il occupe.
5) Ajouter des membres (éléments) à la première ou à la dernière place d'une liste.
6) Supprimer le premier ou le dernier membre d'une liste.
7) Compter le nombre de membres d'une liste.
8) Fusionner deux listes pour n'en faire qu'une.

Les exemples de commandes suivants serviront à illustrer l'utilisation de certaines de ces primitives.

Exemple 20 : Commandes avec des listes

Les listes sont construites avec la primitive «list» ou avec des crochets [ ].

show (list 1 2 "whisky"), montrer la liste composée des 3 membres 1, 2 et "whisky" (ce dernier membre est une chaîne)
==> observer: [1 2 "whisky"],
show [1 2 "whisky"], autre façon de construire la même liste
⇨ **observer: [1 2 "whiksy"]**

Les éléments donnés par des mots ou des morceaux de texte (chaînes) doivent être placés entre guillemets " "(guillemets anglais) afin que l'interpréteur ne tente pas de les évaluer comme s'ils étaient des expressions «vivantes» du langage, par exemple des noms de procédures, de primitives ou de variables.
Les listes d'agents ne peuvent être créées qu'avec la primitive «list», et non avec les crochets [ ]:

crt 3, créer 3 tortues
show (list turtle 0 turtle 1 turtle 2), montrer la liste composée des 3 tortues ainsi crées
==> observer: [(turtle 0) (turtle 1) (turtle2)]

La primitive «member?» (membre?) est utilisée pour savoir si un objet est membre d'une liste:

show member? "e" ["a""b""c""d"]

==> observer: false, indique faux parce que "e" n'appartient pas à la liste donnée.

Les primitives «fput» et «lput» placent l'élément sélectionné à la première (fput) ou à la dernière (lput) place de la liste. Les noms proviennent de la contraction des expressions anglaises «first-put» (placer en premier) et «last-put» (placer en dernier).

show fput 18 ["est mon numéro chanceux"]
==> observer : [18 "est mon numéro chanceux"]

La primitive «item» rapporte l'élément (membre) qui occupe la position indiquée :

show item 2 ["arbre" "bouteille" [6 7] 3]
==> observer : [6 7], la liste [6, 7] occupe la place numéro 2 de la liste ["arbre" "bouteille" [6 7] 3], car les éléments sont comptés à partir de zéro.

Pour éliminer un élément de la liste on utilise «remove-item» et on indique la position de l'élément que l'on veut éliminer

show remove-item 3 ["ce" "vin" "est" "vraiment" "imbuvable"]
==> observer: ["ce" "vin" "est" "imbuvable"]

Les primitives «but-last» (moins le dernier) et «but-first» (moins le premier) fournissent une nouvelle liste à laquelle ont été enlevés, respectivement, le dernier («but-last») ou le premier («but-first») élément de la liste initiale.

show but-last ["chat" 1 4 "chien"]
==> observer: ["chat" 1 4]
show but-first ["chat" 1 4 "chien"]
==> observer: [1 4 "chien"]

Les primitives «first« et «last» rapportent (montrent) respectivement le premier et le dernier membre d'une liste donnée.

show first [10 20 ["Aujourd'hui" "c'est" "lundi"]]
==> observer :10
show first last [10 20 ["Aujourd'hui" "c'est" "lundi"]]
⇨ **observer: "Aujourd'hui"**

“Aujourd’hui” est le premier membre de la liste [”Aujourd’hui” “c’est” “lundi”] et cette liste est le dernier membre de la liste [10 20 [”Aujourd’hui” “c’est” “lundi”]]
La primitive «length» rapporte le nombre d’éléments d’une liste.

show length [1 2 [3 4]]
==> observer : 3, parce que la liste a 3 membres. Il ne faut pas confondre «length» avec «count» (cette dernière primitive fournit le nombre de membres d’un ensemble-agents).

La primitive «sentence» (phrase) permet de fusionner deux listes.
show sentence [1 2][3 4]
==> observer : [1 2 3 4]

Ces exemples n'épuisent pas le répertoire des primitives utilisées pour manipuler des listes. En ce qui concerne les chaînes, la plupart des primitives qui s’appliquent aux listes sont valables pour les chaînes sans aucune modification. Pour plus de détails, il est recommandé de consulter le dictionnaire NetLogo qui donne des explications sur les listes des primitives de listes ainsi que sur celles de chaînes. L'utilisation des chaînes est présentée dans quelques exemples ci-dessous.

Exemple 21: Assignation de sièges dans un avion I

Dans cet exemple, nous supposons qu'un programme choisit au hasard l’emplacement du siège des personnes sur un vol. Cet emplacement est donné par un nombre indiquant la rangée dans laquelle se trouve le siège et une lettre indiquant la colonne. Les rangées sont extraites de la liste des nombres de 1 à 30 et les colonnes de la liste de lettres ["A" "B" "C" "D"]. Le programme «votre-siège» génère aléatoirement l’emplacement d'un siège et l'écrit dans le terminal d'instructions. L'exemple montre à nouveau l'utilisation de la primitive «let» pour définir une variable locale.

```
to votre-siège
let rangée one-of [1 2 3 4 5 6 7 8 9 10 1 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30]
let colonne one-of ["A" "B" "C" "D"]
type "Votre siège est le " type ligne type colonne
end
```

Exemple de résultat possible lors de l'exécution de la procédure dans la fenêtre de l’observateur (écrire votre-siège dans la fenêtre de l’observateur) :

votre-siège
==> Votre siège est le 4D

Explications et commentaires supplémentaires.

La primitive «one-of», (un-parmi ou une-parmi), que nous avons déjà utilisée, sélectionne de manière aléatoire un élément de la liste donnée. Si la procédure «votre-siège» est appelée à partir de la fenêtre de l'observateur, on obtient, comme résultat, la position d'un siège (siège 4D dans l'exemple ci-dessus). Les variables locales «rangée» et «colonne» définies avec «let» n'existent que lors de l'exécution de la procédure «votre-siège». Une fois cette procédure exécutée, ces variables disparaissent. Nous pouvons vérifier cette affirmation si nous essayons de consulter leur valeur dans la fenêtre de l'observateur en écrivant l'instruction «show rangée» ou «show colonne».

Procédures de type «reporter»

Tout comme il existe des primitives d'action (les commandes) et des primitives de type «reporter» (qui fournissent ou «rapportent» des résultats), il en va de même pour les procédures. Il est souvent nécessaire qu'une procédure fournisse des données à d'autres procédures. Les procédures de type «reporter» doivent commencer par la primitive «to-report» au lieu de la primitive «to» comme d'habitude. En outre, un tel type de procédure doit se terminer par l'expression «report donnée», où «donnée» est la valeur rapportée par la procédure. Par exemple, si l'on a la procédure :

to-report colonne-1
report one-of [”A” “B” “C” “D” “E” “F” “G”]
end

celle-ci peut être appelée à partir de la fenêtre de l'observateur à l'aide d'une primitive utilisant une entrée, telle que «type» ou «print»:

type colonne-1
==> “G”
Print colonne-1
==> “B”

Exemple 22 : assignation de sièges dans un avion II

Primitives: to-report, report (rapporter).
Autres détails: deux procédures de type «reporter» sont créés, on utilise des listes et le principe de réutilisation est appliqué.

Dans cet exemple, on crée deux procédures de type «reporter» pour générer les numéros de rangée et les lettres de colonne des sièges et également générer un code d'identification de billet du passager. Avec l'utilisation répétée (principe de réutilisation) des deux procédures «siège» et «billet», le siège et le code du billet du passager sont générés.

Voici le code:

```
to siège
type "Votre numéro de siège est " type rangée type colonne
end

to billet
type "Le code d'identification de votre billet est "
type colonne type colonne type rangée type rangée type colonne type
rangée
end

to-report colonne
report one-of ["A" "B" "C" "D" "E" "F" "G"]
end

to-report rangée
report one-of [1 2 3 4 5 6 7 8 9 10 1 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30]
end
```

Explications et commentaires supplémentaires. Nous avons créé deux procédures de type «reporter», l'une nommée «rangée» et l'autre «colonne». Ces deux procédures sélectionnent respectivement un nombre et une lettre et les rapportent aux procédures ou primitives qui les ont demandées. Dans l'exemple, ces procédures de type «reporter» sont appelées par la primitive «type» à partir des procédures «siège» et «billet». Si l'une des procédures «siège» ou «billet» est appelée à partir de la fenêtre de l'observateur, on obtient un numéro de un siège et un code d'identification du billet.

Par exemple:

siège
==>17F

billet
==>ED16C9

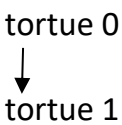
Comment arrêter une procédure avec «stop»

On ne veut pas que ce soit l'utilisateur qui doive toujours arrêter un modèle. Dans de nombreux cas, on peut souhaiter que le modèle s'arrête lorsque certaines conditions sont remplies. Par exemple il n'y a aucun intérêt à ce qu'un programme qui simule l'évolution démographique de deux espèces d'un modèle prédateur-proie continue à s'éxécuter advenant la disparition des deux espèces. C'est pourquoi il existe, dans NetLogo, une primitive «stop» qui permet d'arrêter, sous certaines conditions, un programme en cours. La primitive «stop» ne fonctionne pas comme le «bouton de panique» caractéristique de certaines applications qui, quel que soit l'état dans lequel elles se trouvent, stoppent immédiatement lorsque le bouton est activé.

La primitive «stop» peut complétement arrêter un modèle ou seulement certaines de ses actions en fonction du niveau d'où est donnée la commande. Pour comprendre le fonctionnement de cette primitive, nous devons considérer les différents niveaux auxquels se situe un agent au moment de l'exécution de la commande «stop». Si un agent est encapsulé dans les commandes d'un autre agent, nous dirons que le niveau de celui qui est encapsulé est inférieur au niveau de l'agent qui l'encapsule. Par exemple dans la commande:

ask turtle 0 [fd 5 ask turtle 1 [set color blue ask patch 3 3 [sprout 2]]]

il y a trois niveaux d'encapsulation impliquant trois agents: la tortue 0, la tortue 1 et la parcelle 3 3. Notez que la commande «ask patch 3 3 [sprout 2]» est encapsulée entre les crochets de la tortue 1 qui est elle-même encapsulée entre les crochets de la tortue 0. Cela signifie que l'agent parcelle 3 3 se situe un échelon plus bas que l'agent tortue 1 et deux échelons plus bas que l'agent tortue 0.



parcelle 3 3

La tortue 0 occupe le niveau le plus élevé et a le pouvoir d'arrêter ses propres actions, qui incluent celles de la tortue 1 et celle de la parcelle 3 3. Par contre, la tortue 1 pourrait arrêter ses propres actions et celles de la parcelle 3 3, mais elle n'a aucun pouvoir d'arrêter les actions de la tortue 0. Enfin, la parcelle 3 3 ne peut arrêter que les actions situées entre ses propres crochets. Nous illustrons les effets de l'encapsulation dans les deux exemples suivants (exemples 23 et 24).

Un agent a le pouvoir d'arrêter les actions de son propre niveau et celles des agents encapsulés à des niveaux inférieurs au sien, mais pas les actions des agents situés à des niveaux supérieurs.

Exemple 23: Commandes avec la primitive «stop»

Dans la fenêtre de l'observateur, écrivez les commandes suivantes:

crt 2 [set color red], d'abord on crée deux tortues rouges.

ask turtle 0 [ask turtle 1 [fd 10] set color white], la tortue 1 avance de 10 pas et la tortue 0 devient blanche. Notez que la tortue 0 se situe à un échelon plus élevé que la tortue 1

ask turtle 0 [ask turtle 1 [stop fd 10] set color white], la tortue 0 devient blanche, la tortue 1 n'avance pas car elle a mené ses propres actions avant la commande fd 10. Cependant elle n'a pas le pouvoir d'arrêter la commande «set color white» donnée par la tortue 0, laquelle est située à un niveau plus élevé que le sien.

ask turtle 0 [ask turtle 1 [fd 10] stop set color white], la tortue 1 avance de 10 pas et la tortue 0 ne devient pas blanche (reste rouge).

ask turtle 0 [stop ask turtle 1 [fd 10] set color white], la tortue 1 n'avance pas et la tortue 0 ne devient pas blanche. Ici la tortue 0 a mis fin à ses propres actions ainsi qu'à celle de la tortue 1 située un échelon au-dessous d'elle (la commande «stop» s'adresse aussi bien à la tortue 1 (ask turtle 1 [fd 10]) qu'à la tortue 0 elle-même («set color white»))

Les mêmes principes qui s'appliquent à l'encapsulation entre agents s'appliquent à l'encapsulation entre procédures. Une procédure qui appelle une autre procédure se situe à un niveau supérieur à celui de la procédure appelée. Une commande «stop» dans la procédure appelée n'arrête pas les actions dans la procédure appelante.

Exemple 24 : Une liste de mille nombres entiers

Primitives: ticks, reset-ticks, length (longueur d'une liste), stop (arrêter), lput (abréviation de «last-put», traduit par «placer-à-la-fin»).
Autres détails: on utilise une expression conditionnelle pour arrêter la procédure. Exemple de liste qui se remplit de nouveaux membres avec la primitive «lput».

Dans cet exemple, une liste appelée «nombres-entiers» est créée. Au début, cette liste est vide puis elle se remplit de nombres entiers (en commençant par 0) au cours de l'exécution du programme. Cet ajout de membres à la liste est dû au fait que, la case «Forever» de la procédure «go» étant cochée, la variable «ticks» augmente d'une unité à chaque passage par «go». Il revient à la primitive «lput» le rôle de placer chaque nouvelle valeur de la variable «ticks» à la fin de la liste existante à tout moment de l'exécution. C'est là le sens de la commande:

«set nombres-entiers lput ticks nombres-entiers»

La procédure s'arrête lorsque la liste «nombres-entiers» comprend 1000 membres.

Activités préparatoires: 1) Construire les boutons habituels «setup» et «go» (case «Forever» cochée pour ce dernier).

Le code est le suivant :

```
globals[nombres-entiers]

to setup
clear-all
set nombres-entiers [ ] ;; on crée une liste vide appelée
;; «nombres entiers»
reset-ticks ;; la variable ticks prend  la valeur initiale 0
end
```

```
to go
set nombres-entiers lput ticks nombres-entiers
;; la nouvelle valeur de la variable ticks est placée en fin de liste
if length nombres-entiers = 1000 [show nombres-entiers stop]
;; quand la liste atteint mille membres la liste est envoyée au terminal
 ;; d'instructions et la procédure s'arrête
tick ;; la variable ticks augmente d'une unité
end
```

Envoi des résultats (output) vers un fichier

La zone de sortie («output area») de NetLogo est par défaut le terminal d'instructions. L'exemple précédent montre à quel point ce terminal est inapproprié pour stocker de gros volumes d'informations. Heureusement, il existe d'autres options, telles que le transfert des informations vers une fenêtre de sortie ou un fichier texte. Cette dernière option fait l'objet de l'exemple 25.

Exemple 25: Envoi d'une liste de nombres dans un fichier

Dans cet exemple, les 1000 premiers nombres entiers positifs sont, comme dans l'exemple 24, construits et stockés dans une variable globale sous forme de liste. Mais au lieu d'être envoyés au terminal d'instructions, les résultats sont envoyés à un fichier texte par le biais d'une procédure appelée «envoyer-résultats».

Primitives: reset-ticks, tick, type (écrire), stop, lput (mettre-à-la-fin), file-print (imprimer-fichier), length (longueur), file-open (ouvrir-fichier), file-close (fermer-fichier).
Autres détails : On arrête la procédure à l'aide de la primitive «stop» et on envoie les résultats vers un fichier externe.

Activités préparatoires: 1) Construire les boutons habituels «setup» et «go» (cocher «Forever»). 2) Construire un troisième bouton appelé «envoyer-résultats» ou tout simplement écrire envoyer-résultats dans la fenêtre de l'observateur[23].

[23] Cette façon de faire fonctionne avec la présente version de NetLogo (Version 6.2.0 Décembre 2020). Il est possible qu'avec des versions antérieures il faille, avant de lancer la procédure, créer un fichier texte brut appelé «mille-nombres-entiers.txt» car

Voici le code:

```
globals[nombres-entiers]

to setup
clear-all
set nombres-entiers [ ]
;; on crée une liste appelée «nombres entiers» initialement
 ;; vide
reset-ticks ;; la variable ticks prend  la valeur initiale 0
end

to go
set nombres-entiers lput ticks nombres-entiers
;; la nouvelle valeur de la variable ticks est placée en fin de
;; liste
if length nombres-entiers = 1000 [envoyer-résultats stop]
;; quand la liste atteint mille membres
;;un fichier texte est créé et la liste est envoyée à ce fichier
;; texte
tick ;; la variable ticks augmente d'une unité
end

to envoyer-résultats
file-open «mille-nombres-entiers.txt» ;; voir note
;; de bas de page
file-print «Voici la liste des mille premiers nombres
entiers:»
file-print «\r\n»
file-print nombres-entiers
file-close
end
```

Explications et commentaires supplémentaires. Certains éditeurs de texte, par exemple Notepad de Windows, ne reconnaissent pas le retour de chariot («carriage return» en anglais) envoyé avec les primitives «print» et «show». Le but de l'instruction ««\ r \ n»« est de forcer le changement de ligne. Il est conseillé de vérifier si l'éditeur utilisé reconnaît

NetLogo ne le créera pas automatiquement. Ce fichier doit être placé dans le même répertoire que celui où est stocké le programme NetLogo de l'exemple.

le retour de chariot pour déterminer s'il est nécessaire d'inclure la commande mentionnée. N'oubliez pas que les primitives «type» et «write» n'incluent pas le retour de chariot.

Les primitives «show», «type», «print» et «write»

Pour envoyer du texte ou des valeurs numériques vers une zone de sortie, nous disposons des primitives «show», «print», «type» ou «write». Les primitives diffèrent par les aspects suivants: indiquer ou non l'agent qui a passé la commande, forcer ou non un changement de ligne (retour chariot) à la fin du texte, imprimer ou non les guillemets des chaînes, commencer ou non par un espace en blanc. Pour obtenir le résultat souhaité, il faut parfois utiliser des combinaisons de ces primitives. Les exemples suivants montrent quelques-unes des différences entre ces primitives:

type "bonjour monde"
==> bonjour monde

type "bonjour" type "monde"
==> bonjourmonde

type "bonjour" type " " type "monde"
==> bonjour monde

show "bonjour" show "monde"
==> observer: "bonjour "
==> observer: "monde "

write "bonjour" write "monde"
==> "bonjour" "monde"

print "bonjour" print "monde"
==> bonjour
==> monde

Ensemble-agents : deuxième rencontre

Les ensemble-agents sont toujours formés par des agents du même type: tortues, parcelles, liens ou familles de ces agents. Les ensemble-agents, contrairement aux listes, ne sont pas des ensembles ordonnés. Chaque fois qu'un ensemble-agents est appelé, par exemple, avec la primitive

«ask», l'ensemble est «mélangé», de sorte que ses agents sont tous dans un nouvel ordre aléatoire. Cela évite à certains de ses membres d'occuper des positions privilégiées. Les primitives qui permettent de manipuler les listes ne peuvent pas être utilisées pour les ensemble-agents. Par exemple, on utilise la primitive «count» pour compter le nombre de membres d'un ensemble-agents, tandis que pour les listes, on utilise la primitive «length».
Il est également possible de créer des ensemble-agents en imposant des restrictions à un ensemble-agents qui existe déjà. NetLogo a des primitives qui permettent la création de sous-ensemble-agents, telles que with, one-of, turtles-on, turtles-here, with-min, with-max. Très souvent, ces sous-ensemble-agents sont créés au moment de l'exécution du modèle puis ils disparaissent. On peut cependant leur donner une certaine permanence en les stockant dans des variables à l'aide de «let» ou de «set», selon le cas. Si, par exemple, nous avons créé une variable globale appelée «rouges», nous pouvons stocker l'ensemble-agents des tortues qui ont la couleur rouge à l'aide de l'expression ««set rouges turtles with [color = red]». Désormais, l'ensemble-agents appelé «rouges» peut recevoir et exécuter des commandes comme, par exemple, «ask rouges [...commandes...]».

Le parallélisme réel et le parallélisme simulé

Le concept de programmation à base d'agents repose fortement sur l'idée que les agents peuvent effectuer des tâches simultanément. Cela conduit inévitablement à la question suivante: la simultanéité des actions des agents Netlogo est-elle réelle ou simulée? La réponse à cette question est claire: dans les ordinateurs et les systèmes d'exploitation pour lesquels NetLogo et la plupart des programmes à base d'agents ont été développés, un réel parallélisme n'est pas possible. Les seules machines capables de fonctionner avec un réel parallélisme à grande échelle sont des superordinateurs installés dans des centres de recherche ou qui sont réservés à quelques institutions et qui nécessitent des systèmes d'exploitation et des logiciels spécialisés et coûteux. En dehors de ces quelques exemples, le monde de l'informatique continue de fonctionner selon le modèle séquentiel. Nous recommandons aux lecteurs intéressés par ce sujet l'excellent et très accessible livre de D. Hillis [8].

Ainsi, dans la plupart des cas, les actions qui semblent se produire simultanément se font en réalité de manière séquentielle, sous couvert d'un parallélisme simulé. Cependant, la rapidité des microprocesseurs

modernes permet à ce parallélisme simulé de ressembler à un vrai parallélisme. Ainsi, un moyen de mettre en œuvre des processus en parallèle consiste à interconnecter un ensemble d'ordinateurs en réseau afin que chacun effectue une partie du traitement. De plus, l'apparition récente d'ordinateurs dotés de deux, trois, sept cœurs et plus dans des microprocesseurs peut également apporter une aide à certains types de processus, sans toutefois résoudre le problème posé par l'exécution simultanée de tâches par des centaines ou des milliers d'agents en contexte d'interactions.

En dépit de tout ceci, les utilisateurs conservent toujours un certain niveau de contrôle sur l'efficacité du parallélisme simulé dans les modèles multiagents. Pour cela, il est utile de comprendre le fonctionnement de certaines instructions de NetLogo telle que «ask» qui est l'une des primitives les plus utilisées dans les modèles NetLogo. Lorsque cette primitive est adressée à un ensemble-agents, suivie d'une liste de commandes, telle que, dans «ask turtles [set color red set heading random 180 fd 1]», les choses se passent de la manière suivante pour exécuter les trois commandes entre crochets. Chaque fois que la primitive «ask» est invoquée, elle produit une liste aléatoire d'agents et chaque agent de la liste doit, à son tour, exécuter la séquence complète des trois commandes entre crochets. Cela signifie que tant qu'un agent n'a pas exécuté toutes les commandes de la séquence, l'agent suivant ne peut pas démarrer l'exécution. Toutefois, s'il fallait, pour les besoins de la simulation, modifier le parallélisme simulé, l'on pourrait, par exemple, scinder cette commande en trois commandes distinctes : «ask turtles [set color red]», «ask turtles [set heading random 180]» et «ask turtles [fd 1]». De cette façon, tous les agents adoptent d'abord, l'un après l'autre, la couleur rouge puis, toujours l'un après l'autre, fixent leur orientation pour finalement avancer d'un pas chacun à leur tour.

Une remarque pour terminer cette petite digression: dans un grand nombre de modèles mettant en scène des agents en interaction, il n'est pas nécessaire que les interactions se produisent simultanément au sens strict du terme. Dans de nombreux cas, l'écart entre les résultats obtenus au moyen d'un hypothétique parallélisme réel et ceux d'un parallèle simulé peut être négligeable voire ne produire aucune différence.

Chapitre 4: Modèles I

Introduction

Dans les chapitres précédents, nous avons présenté un ensemble de quelques primitives NetLogo ainsi qu'une série de concepts de base en matière de programmation. Nous avons également montré comment communiquer avec NetLogo via son interface. Bien qu'il reste encore beaucoup à apprendre sur le langage et l'environnement NetLogo, nous avons atteint un point où il est possible de commencer à construire des modèles simples. La plupart des exemples que nous avons conçus pour ce chapitre et pour le chapitre suivant appartiennent à ce que nous pourrions appeler des «modèles fantaisistes». Nous les appelons ainsi car ils modélisent des situations imaginaires, dont certaines pourraient cependant se produire en réalité d'une manière très similaire à celle décrite par le modèle. Nous pensons qu'il est plus agréable de relier un programme ou un modèle à une petite histoire que de montrer un code détaché de tout lien avec l'expérience.

Parfois, c'est un phénomène que l'on a observé, une expérience que l'on a vécue ou des nouvelles que l'on a lues dans les journaux qui sont à l'origine de la création d'un modèle. Comme nous le verrons, la description d'une petite histoire a également l'avantage de générer des idées sur les moyens d'élargir la portée du modèle, d'en imaginer des variantes ou de le relier à d'autres domaines du savoir. En ce sens, les exemples de ce chapitre commencent à montrer un aspect très précieux de l'activité de programmation, qu'il s'agisse de modèles ou de programmes en général. En laissant beaucoup d'espace à la créativité, la création de modèles est un terrain fertile pour l'activation des processus mentaux engendrés par la programmation du modèle. C'est ce qui donne une valeur toute particulière à la pratique de la programmation dans le domaine de l'éducation.

Certains lecteurs seront peut-être surpris de voir la richesse des idées qui peuvent émaner de certains programmes, malgré la petite taille de leur code. Comme on le verra, un programme n'est pas nécessairement plus complexe ou plus performant, parce qu'il utilise un plus grand nombre de primitives, ni en raison du nombre de lignes de code qu'il contient, de la même manière qu'une œuvre musicale n'est pas plus mélodieuse parce qu'elle utilise plus de notes de la gamme ou parce que le nombre de pages de sa partition est plus élevé. La musique de Mozart en est un exemple tout comme la vie elle-même avec son immense diversité et sa complexité et dont l'écriture ne nécessite qu'un petit alphabet de quatre lettres: A, T, C et G correspondant aux nucléotides Adénine, Thymine, Cytosine et Guanine.

Dans ce chapitre, nous continuerons avec la pratique qui consiste à intercaler, entre les modèles et les exemples, de nouveaux concepts et techniques de programmation. Contrairement aux modèles, les exemples sont de petites unités de code dont le but est de montrer le fonctionnement d'une primitive ou d'illustrer un concept de programmation.

Modèle 1: Tina et Magda visitent la ville

Primitives: remainder (reste), pu (abrev. de «pen up», «stylet-en-l'air»), one-of (un-parmi), xcor, ycor, set (assigner), sound:play-note (son:joue-note), beep (son bip), stop (arrête), or (disjonction «ou»), and (conjonction «et»).
Autres détails: Un curseur est construit, la simulation est arrêtée avec «stop» lorsqu'une condition est remplie, la topologie du carré est utilisée et l'extension «sound» est importée.

L'histoire Deux amies visitent une nouvelle ville à moto. Enthousiasmées à l'idée de connaître les nombreux lieux nouveaux et intéressants qu'offre la ville, elles se séparent et perdent contact. Comme les deux femmes ont décidé de ne pas prendre leurs téléphones cellulaires, elles n'ont d'autre moyen de se retrouver que de traverser les quartiers de la ville jusqu'à ce qu'elles réussissent à établir un contact visuel entre elles. Pour cela, il est nécessaire que les deux amies se trouvent à moins d'une certaine distance l'une de l'autre dans la même rue ou avenue, afin qu'aucun bâtiment n'obstrue leur champ de vision. La distance maximale qui doit les séparer pour qu'elles puissent se voir ne doit pas dépasser une certaine quantité (pieds ou mètres) dont la valeur est fixée par le curseur «max-visible» dans l'interface.
Pendant que les deux femmes conduisent en tentant de se revoir, elles décident de jouer avec les klaxons de leurs motocyclettes (cette activité est purement ludique et n'aide en rien les deux amies à se retrouver).
Afin de rendre la simulation plus réaliste, nous ferons en sorte d'empêcher les deux amies de revenir sur leurs pas pendant le trajet. C'est la raison pour laquelle elles ne sont pas autorisées à faire un tournant de 180 degrés au guidon de leur motocyclette. Les seuls virages autorisés sont: virage de 90 degrés à droite, virage de 90 degrés à gauche ou virage de 0 degré (ce dernier signifiant que l'on continue de rouler en ligne droite). La topologie du monde dans lequel évoluent Magda et Tina est celle du carré. La simulation s'arrête quand les deux femmes finissent par se rencontrer.

Activités préparatoires : Configurer le monde avec la topologie du carré. Construire les boutons «setup» et «go» (cocher «Forever» dans ce dernier). Construire un curseur pour la variable nommée «max-visible».

Problèmes à résoudre

1. Il est nécessaire de déterminer quand les deux amies se trouvent dans la même rue ou dans la même avenue et quand la distance qui les sépare est inférieure à la valeur définie par le curseur «max-visible».
Un moyen de résoudre ce problème est de définir quatre variables: Mxcor, Mycor, Txcor, Tycor, où la paire Mxcor, Mycor indique à tout moment la position de Magda et la paire Txcor, Tycor celles de Tina. Si nous appelons «rues» les lignes horizontales (coordonnée Y constante) et «avenues» les lignes verticales (coordonnée X constante), la condition pour que Tina et Magda puissent se voir est qu'elles se trouvent toutes les deux dans la même rue ou sur la même avenue. Cette condition est remplie si la coordonnée X de Tina est égale à la coordonnée X de Magda (même avenue) ou si la coordonnée Y de Tina est égale à la coordonnée Y de Magda (même rue). Nous pouvons exprimer cette condition dans le code à l'aide de l'expression:
Txcor = Mxcor or Tycor = Mycor
Dans laquelle la primitive «or» est équivalente à la disjonction inclusive «ou» en français. L'autre condition est que la distance entre les deux femmes («disTM») soit inférieure à la valeur définie par le curseur max-visible. Cette condition est représentée dans le code par l'expression «disTM < max-visible». Pour que Magda et Tina soient à nouveau réunies, les deux conditions précédemment mentionnées doivent être satisfaites *simultanément*. Cette exigence s'exprime en construisant la conjonction des deux conditions à l'aide de la primitive «and» («et» en français):

(disTM < max-visible) and (Txcor = Mxcor or Tycor = Mycor)

2. Magda et Tina doivent utiliser leurs klaxons de temps à autre, mais pas à l'unisson et les sons des klaxons doivent être différents. Pour cela, nous devons importer l'extension «sound» (son) dont la primitive «play-note» permet de produire le son des klaxons de moto. Pour contrôler le moment où chaque femme klaxonne, on a recours à la primitive «remainder num1 num2». Cette primitive donne le reste de la division de num1 par num2. Par exemple, «remainder 24 7» donne 3 (le reste de la division de 24 par 7 est 3) et «remainder 24 3» donne 0. Le code stipule que l'une des deux amies klaxonne chaque fois que la division de la variable ticks par 100 donne un reste égal à zéro:

«if remainder ticks 100 = 0 [sound:play-note...]».

Quant à l'autre amie, son code est tel qu'elle ne klaxonne que quand le reste de la division est égal à 20. Ainsi, dans les deux cas, chacune des deux femmes klaxonne toutes les 100 ticks, mais pas au même moment.

3. Il vous faut maintenant construire la topographie de la ville. Vous devez écrire une nouvelle procédure (appelée «blocs-urbains» dans l'exemple) qui représente cette topographie en traçant le réseau de rues et d'avenues de la ville. Comme ces rues et avenues sont représentées par, respectivement, des lignes horizontales et des lignes verticales elles délimitent un ensemble de petits blocs carrés dont les usages peuvent être multiples (maisons, commerces, parkings, terrains vides etc.). Nous verrons dans le Modèle 4 du présent chapitre comment cette topographie peut être utilisée pour illustrer un autre exemple de la vie quotidienne.

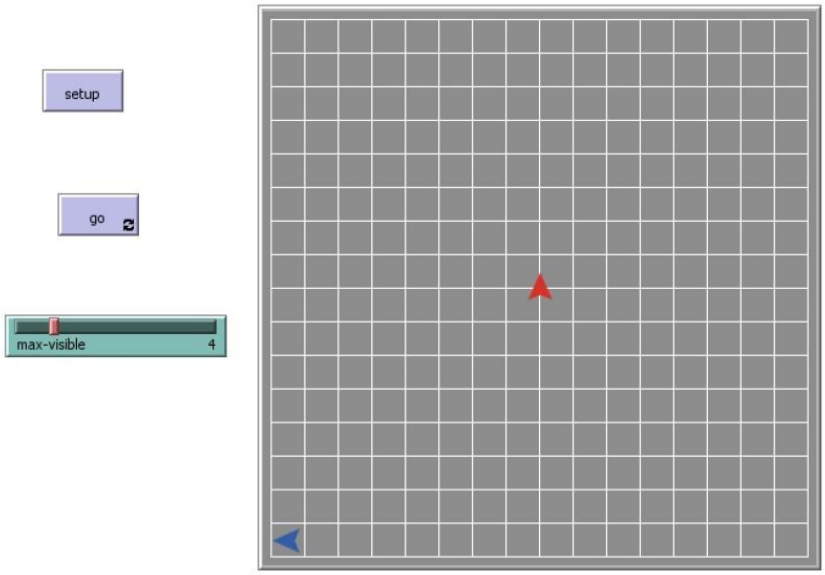


Figure 4.1 : Exemple de topographie urbaine

Voici le code complet :

```
extensions [sound]
globals[disTM Txcor Tycor Mxcor Mycor]


to setup
  clear-all
  crt 2 [set heading 0 set size 2]
 blocs-urbains ;; appelle la procédure «blocs-urbains»
  ask turtle 0 [pu set xcor -15  set ycor -15 set color blue]
  ask turtle 1 [pu set color red]
    reset-ticks
end

to go
  ask turtles [fd 1 right one-of [90 0 -90]]
```

```
  ask turtle 0 [set Txcor xcor set Tycor ycor
   if remainder ticks 100 = 0
     [sound:play-note "TRUMPET" 60 64 0.5]]
  ask turtle 1 [set Mxcor xcor set Mycor ycor
   if remainder ticks 100 = 20
     [sound:play-note "TRUMPET" 70 64 0.5]]
  ask turtle 0 [set disTM distance turtle 1]
  ;; commande «if»:condition d'arrêt lorsque Tina & Magda se rencontrent
  if (disTM < max-visible)  and (Txcor = Mxcor or Tycor = Mycor)
     [sound:play-note "TRUMPET" 80 64 0.5
      sound:play-note "TRUMPET" 90 64 0.5 stop]
     wait 0.04
tick
end

to blocs-urbains
  ask patches [set pcolor gray]
  ask turtle 0 [set color white pu set heading 0 set xcor -16
     set ycor -16 pd repeat 16 [fd 32 rt 90 fd 2 rt 90
     fd 32 lt 90 fd 2 lt 90 fd 32] bk 32 lt 90 repeat 16
     [fd 32 rt 90 fd 2 rt 90 fd 32 lt 90 fd 2 lt 90 fd 32]
     ]
end
```

Explications et commentaires supplémentaires. Les extensions sont déclarées au début du code, avant les procédures, avec la primitive «extensions». La primitive «play-note» doit être précédée de la primitive «sound:» qui est le nom de l'extension à laquelle elle appartient. Cette primitive a quatre entrées: nom de l'instrument, hauteur (pitch) de la note, volume (également appelé vitesse) et durée de la note en secondes. Dans l'ordre: *sound:play-note "TRUMPET" 80 64 0.5* signifie que l'instrument est la trompette, la note a une hauteur de 80, un volume de 64 et une durée de 0,5 seconde.

L'extension «sound» utilise les mêmes paramètres d'entrée que le protocole MIDI[24] bien connu, et la collection d'instruments correspond à la liste d'instruments General MIDI de ce protocole. Étant donné que la variable ticks augmente d'une unité à chaque fois que l'interprète passe par la procédure go (augmentation générée par la primitive «tick»), il est inévitable que tous les 100 ticks, chacun des 100 restes de la division «ticks / 100» se produise une fois. La valeur 100 a été choisie pour espacer les coups de klaxon tous les 100 ticks afin de ne pas trop «casser les oreilles» des utilisateurs.

[24] MIDI est l'acronyme de «Musical Instrument Digital Interface»

Exercice 4.1 Dans le modèle précédent, les deux amies klaxonnent de manière synchrone. Magda klaxonne toujours 20 fois après que Tina l'a fait. Écrivez un code qui ferait en sorte que les deux amies klaxonnent au hasard et de manière non coordonnée, ce qui donnerait une touche légèrement plus réaliste au modèle.
Le modèle que nous venons de décrire pourrait très bien s'adapter à plusieurs autres situations, que ce soit avec d'autres types d'agents ou dans d'autres contextes, par exemple, des personnes dans une forêt, des micro-organismes présents dans les tissus d'un organe, des insectes dans les mauvaises herbes, des molécules dans une solution ou dans une structure cristalline et même des personnes dans un grand magasin. En fait, l'exemple présenté dans cette section s'inspire d'une histoire racontée par un ami qui avait perdu sa femme dans un grand magasin à plusieurs étages (cet ami avait oublié de prendre son téléphone cellulaire ce jour-là).

Modèle 2 : Fonds de pension I

Primitives: if, write (écrire), turtles-own (propre-aux-tortues), setxy, random-xcor, random-ycor.
Autres détails: on construit des curseurs pour plusieurs variables du modèle et l'évolution du capital du fonds de pension et celle la population (travailleurs actifs + retraités) sont illustrées graphiquement.

L'histoire Les salariés d'une institution publique versent un pourcentage de leurs salaires à un fonds de pension tout au long de leur vie active, ce qui leur donne le droit de percevoir une pension jusqu'à leur décès. Les ressources du fonds proviennent de deux sources : d'une part, de la contribution des travailleurs et travailleuses sous forme d'un pourcentage des salaires versés par l'État ; d'autre part des revenus provenant des intérêts engendrés par les placements du capital accumulé du fonds sur le marché financier (obligations, actions ou compte d'épargne). Les salaires que les travailleurs perçoivent à titre de rémunération pour leur travail dans l'entreprise constituent des revenus indépendants du système qui gère les fonds et, étant donné qu'ils proviennent d'une source externe au système, ils n'affectent en rien le montant du capital du fonds.

Le modèle a pour objectif d'analyser la stabilité du fonds, c'est-à-dire les conditions dans lesquelles le fonds est actuariellement viable et celles dans lesquelles il ne l'est pas. On dit qu'un fonds est actuariellement viable s'il peut

continuer à verser des pensions de retraite aux travailleurs pour une durée indéterminée sans faire faillite. Dans la première version du modèle, on suppose que les personnes entrent sur le marché du travail à l'âge de 25 ans, travaillent tout au long de leur vie active et bénéficient du régime des retraités jusqu'à leur décès. Dans le modèle, l'unité de temps utilisée pour enregistrer les salaires et les déductions est l'année. La première version du modèle repose sur des hypothèses simplificatrices, mais elle contient le mécanisme essentiel du fonctionnement des fonds de pension. Dans la deuxième version, certaines des hypothèses simplificatrices seront éliminées et un modèle plus réaliste sera présenté. Nous décrivons d'abord la signification des variables du modèle définies par les curseurs de l'interface:

- Curseur «cycle-travail»: détermine le nombre d'années que les salariés doivent travailler avant de prendre leur retraite.
- Curseur «personnes-nouvelles»: détermine le nombre de nouveaux travailleurs qui entrent sur le marché du travail chaque année.
- Curseur «salaire» : définit le salaire annuel unique de tous les salariés. Ce salaire ne varie pas dans le temps (pas d'augmentations salariales).
- Curseur «taux-cotisation»: définit le pourcentage du salaire que les travailleurs actifs et retraités cotisent chaque année au fonds.
- Curseur «pourcentage-pension»: définit le pourcentage du salaire que le travailleur reçoit sous forme de pension.
- Curseur «âge-décès»: définit l'âge auquel les travailleurs décèdent (Ils décèdent tous après avoir atteint le même âge).
- Curseur «taux-rendement»: définit le taux annuel auquel le capital du fonds est placé pour générer des intérêts.

Dans ce modèle, les effets visuels qui représentent les agents importent peu et pourraient être supprimés. Cependant, les travailleurs actifs sont représentés par des points immobiles de couleur rouge tandis que les retraités par des points immobiles de couleur bleue. Cependant, on observe beaucoup d'activité à l'écran: les points rouges qui apparaissent sont les travailleurs qui entrent sur le marché du travail, les points bleus qui disparaissent sont les retraités qui meurent et les points rouges qui deviennent bleus sont les travailleurs qui partent à la retraite. Dans l'interface (Figure 4.2), deux graphiques retracent l'évolution du capital du fonds et celle du nombre de salariés en vie (actifs + retraités). Si la courbe du capital (axe des Y) décroit et finit par couper l'axe des X (axe du temps), cela signifie que le fonds a fait faillite et le programme s'arrête. Du fait que chaque année, un certain nombre N de salariés entrent sur le marché du travail à l'âge de 25 ans et que ces N salariés disparaissent en même temps à l'âge de leur décès, le graphique qui retrace l'évolution de la population atteint toujours, au bout d'un certain temps, un point où la population reste constante.

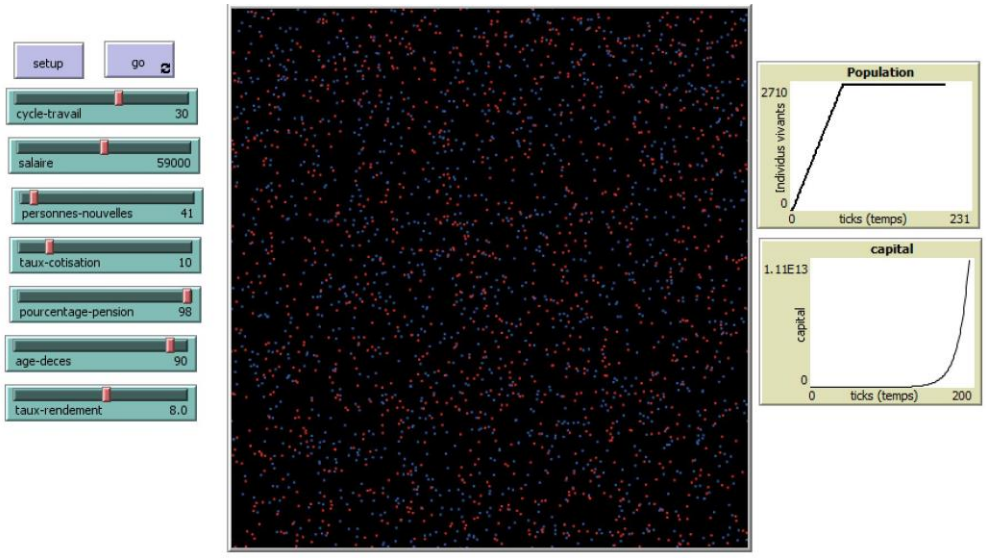


Figure 4.2 Vue de l'interface: le graphique en haut à droite montre l'évolution de la population qui, après un certain temps, se stabilise ; le graphique du dessous illustre la croissance du capital dans le temps.

Problèmes à résoudre Le principal problème est de faire en sorte que le capital du fonds soit actuariellement viable à tout moment. Les cotisations au fonds provenant des rémunérations des salariés sont calculées au moyen de la procédure «cotiser» et les dépenses - le paiement des pensions aux retraités - sont calculées par l'intermédiaire de la procédure «percevoir-pension». Pour que les salariés puissent avoir des âges différents, la variable «âge» ne doit pas être de type global mais du type «turtles-own». Dans la procédure «cotiser», chaque travailleur actif ajoute à la variable «capital» un pourcentage de son salaire:

set capital capital + salaire * (pourcentage-pension) / 100

Dans la procédure «percevoir-pension», chaque salarié soustrait de la variable «capital» un pourcentage de son salaire, pour tenir compte du fait que le retraité continue de cotiser à sa propre pension:

set capital capital - salaire * (pourcentage-pension - taux-cotisation) / 100

Les intérêts engendrés par le placement du capital dans une banque, dans des obligations ou dans des actions sont ajoutés au capital par le biais de la procédure «go»:

set capital capital + capital * taux-rendement / 100

Si le fonds ne fait pas faillite, il revient à l'utilisateur d'arrêter la simulation en cliquant sur le bouton «go».

Activités préparatoires. Construire les boutons «setup» et «go» (cocher «Forever») ainsi que les curseurs des sept variables mentionnées ci-dessus. La variable globale «capital» est déclarée au début du code et représente le capital dont le fonds dispose à tout moment. La valeur initiale du capital est zéro. Construire les graphiques qui retracent l'évolution de la population et celle du capital.

Dans la fenêtre d'édition du graphique pour la population (Figure 4.3), inscrivez:

Name (Nom) : population
X axis label (Étiquette de l'axe des x) : ticks (temps)
Y axis label (Étiquette de l'axe des Y): Individus vivants
Pen update commands (Instructions de mise à jour des tracés): plot count turtles

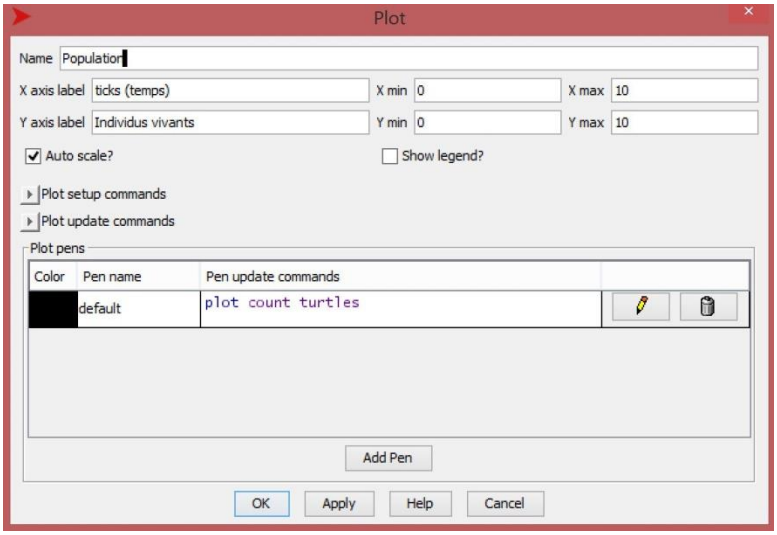


Figure 4.3 : Fenêtre d'édition du graphique pour la population

La fenêtre d'édition du graphique illustrant l'évolution du capital se construit de la même manière que le graphique précédent en changeant le nom du modèle («capital» au lieu de «population») et celui de l'étiquette de l'axe des Y («capital» au lieu de «individus vivants»)

Le code du modèle est le suivant:

globals[capital]
turtles-own[age]

to setup

```
clear-all
reset-ticks
ask turtles [set age 25]
end

to go
create-turtles personnes-nouvelles [ set age 25 set color red setxy random-xcor
random-ycor ]
ask turtles [ifelse age < 25 + cycle-travail [cotiser ]
[set color blue percevoir-pension]
if age > age-deces [die]
]
set capital capital + capital * taux-rendement / 100
if capital < 0 [show "Termine"
write "ticks " write ticks write " Capital: "
print capital stop]
tick
end

to cotiser
set capital capital + salaire * taux-cotisation / 100
set age age + 1
end

to percevoir-pension
set capital capital - salaire * (pourcentage-pension - taux-cotisation) / 100
set age age + 1
end
```

Explications et commentaires supplémentaires Chaque nouvelle mise en action de la procédure «go» représente le passage d'une année nouvelle dans le modèle, c'est-à-dire que de nouvelles personnes (tortues) sont créées qui entrent dans sur le marché du travail, d'autres passent du statut de travailleur actif à celui de retraité et d'autres meurent.

Le modèle débute lorsque les salariés ont 25 ans et commencent à cotiser au fonds (ticks = 0). Pour chaque salarié, la variable d'âge augmente d'une unité à chaque passage de «go» et les salariés qui entrent sur le marché du travail à des moments différents de passage par «go» ont des âges différents. Comme tous les salariés commencent à travailler à 25 ans, l'âge de la retraite est de 25 ans + cycle de travail (par exemple si l'on fixe, avec le curseur, la durée d'un cycle de travail – le nombre d'années de vie active – à 40 ans, l'âge de la retraite sera fixé à 65 ans). A chaque passage par «go» chaque tortue (personne) compare son âge à l'âge de la retraite. Si son âge est supérieur à 25 ans + cycle

de travail, la tortue est dirigée vers la procédure «percevoir-pension», sinon elle est dirigée vers la procédure «cotiser» pour continuer à travailler et à cotiser.

Stabilité du fonds. En faisant varier les différents paramètres du modèle, nous pouvons trouver des configurations proches des «points critiques», où de petites variations de certains des paramètres qui contribuent au renflouement du fond le rendent instable. On peut vérifier que la stabilité du fonds ne dépend pas de la valeur initiale du capital, ni du nombre N de personnes qui entrent annuellement sur le marché du travail tant que ce nombre reste constant. De même, cette stabilité ne dépend pas du fait que tous les salariés perçoivent le même salaire. Comme dans ce modèle, les tortues n'interagissent pas entre elles, le nombre de personnes qui entrent chaque année est sans importance: un fonds où entrent annuellement des centaines de nouvelles personnes a la même stabilité qu'un fonds où entre une seule personne par an. En ce qui concerne le montant du salaire, nous verrons dans la deuxième version du modèle que les écarts salariaux n'influencent pas la stabilité du fonds, tant que ces écarts restent à peu près stables: les salaires élevés paient des pensions élevées et les bas salaires paient des pensions faibles. Les valeurs des paramètres suivantes sont un exemple d'une configuration proche d'un point critique:

1. Durée du cycle de travail: 30 ans
2. Pourcentage du salaire cotisé («taux-cotisation») pour la pension: 9%.
3. Pourcentage du salaire perçu à titre de pension: 96% - cotisation de 9% = 87%
4. Âge au décès: 90 ans.
5. Taux de rendement du capital en pourcentage («taux-rendement»): 8%

Il est ainsi possible, en procédant à des analyses se sensibilité (c'est-à-dire en faisant varier à la hausse ou à la baisse, une à la fois, les valeurs précédentes) d'analyser l'impact de chacune des variables sur la viabilité financière du fonds. Par exemple, si l'on ramenait le taux de rendement du capital à 7% ou celui des cotisation des salariés à 8%, le fonds serait en faillite. Dans bien des cas, l'employeur (ou l'État) verse sa quote-part aux fonds de pension, ce qui permet d'obtenir des configurations de points critiques plus favorables aux salariés. Comme nous le verrons dans la deuxième version du modèle, la mortalité des retraités ne semble pas affecter négativement le fonds. Les retraités meurent plus fréquemment que les travailleurs actifs: un retraité qui décède avant l'âge de décès prévu (90 ans) est une personne à qui le fonds n'aura plus à payer de pension. Cela peut compenser l'effet de la mort des travailleurs actifs. Notre modèle de fonds de pension est totalement autonome, car il ne reçoit pas - comme beaucoup d'autres fonds dans la vie réelle - de contributions de l'État ou des employeurs. Des modèles similaires au fonds d'épargne actuel peuvent être traités mathématiquement au moyen d'équations en différences finies ou

d'équations différentielles. C'est l'approche utilisée dans la discipline appelée Dynamique de Systèmes [19], dans laquelle le programmeur n'a qu'à formuler les relations de base entre les variables et le logiciel se charge de traduire ces relations en équations différentielles et de trouver des solutions approximatives à l'aide de méthodes numériques (méthode d'Euler, méthode de Runge-Kutta, etc.).

NetLogo a une extension qui permet de créer des modèles selon la méthode de Dynamique de Systèmes, mais nous ne discuterons pas de ce sujet dans ce livre. Les lecteurs intéressés par cette extension peuvent la consulter la section extensions du Manuel de l'utilisateur. À cet égard, il est intéressant de souligner la simplicité du code du modèle à base d'agents de NetLogo et le fait qu'il ne fait pas de projections approximatives de l'évolution du fonds (comme dans la Dynamique de Systèmes), mais plutôt des projections exactes. C'est le fait que chaque agent a ses propres variables sur la base desquelles il peut effectuer ses propres calculs qui permet de programmer avec une telle simplicité et une telle efficacité des modèles multi-agents du genre de celui qui vient d'être présenté.

Modèle 3: Fonds de pension II

Primitives: Les mêmes que dans l'exemple précédent Autres détails: les salaires et l'âge au décès sont fixés au hasard. Un curseur est introduit pour faire varier les taux d'augmentation salariale. Les curseurs de salaire et d'âge au décès sont supprimés

Les trois modifications suivantes apportées au Modèle 2 («Fonds de pension I» rendent le nouveau modèle («Fonds de pension II») plus conforme à la réalité :

1. Le taux de mortalité des travailleurs actifs est différent de celui des retraités.
2. L'hypothèse d'un salaire unique est remplacée par celle d'une échelle salariale.
3. Les salaires et les pensions peuvent être indexés à l'inflation à l'aide du curseur taux annuel d'augmentation salariale.

Les taux différentiels de mortalité entre travailleurs actifs et retraités font que la population ne reste pas constante et connaît de très petites fluctuations autour d'une valeur constante. Cependant, ni la présence d'une échelle salariale, ni l'existence de taux différentiels de mortalité n'affectent la stabilité du fonds. En fait, la seule source de variation du fonds est l'inflation qui contribue à l'augmentation annuelle des salaires des travailleurs actifs et des retraités.

Ce fait oblige à renforcer dans une certaine mesure - qui dépend du pourcentage d'augmentation - les conditions pour que le fonds se maintienne. Selon le schéma observé dans les économies de plusieurs pays, une augmentation du taux d'inflation s'accompagne d'une augmentation des taux d'intérêt, ce qui permet au fonds de compenser l'augmentation du paiement des pensions en plaçant le capital à un taux d'intérêt plus élevé sur le marché financier. Pour créer une échelle salariale, (salaires exprimés en dollars, par exemple), on utilise la primitive «one-of» (un-parmi), qui sélectionne au hasard un élément dans une liste. Pour attribuer des probabilités différentes aux différents éléments de la liste, nous utilisons l'astuce qui consiste à répéter plusieurs fois le même élément. Par exemple, dans la commande «one-of [36000 60000 60000 60000 24000 24000 80000]», l'élément 60000 est plus susceptible d'être sélectionné que les autres; en fait, sa probabilité d'être sélectionné est de 3/7, tandis que pour les éléments 36000 et 80000 la probabilité est 1/7. Cette astuce est également utilisée pour attribuer différentes probabilités à l'âge au décès des personnes.

Ces modifications apportent quelques changements minimes dans le code et dans la fenêtre d'interface. Pour ce qui est de cette dernière, les curseurs «salaire» et «âge-décès» disparaissent dans le présent modèle car ces deux variables sont maintenant transformées en variables aléatoires par l'ajout des lignes de code:

```
set salaire one-of [12000  36000  36000  36000  60000 60000 60000 100000]
set âge-décès one-of [35 40 60 60 70 70 70 80 80 80 80 90
                      90 90 90 90 100]
```

On remarquera aussi que l'on a ajouté un nouveau curseur «taux-augmentation-salariale» qui ne figurait pas dans le modèle «Fonds de pension I» et ce pour tenir compte de l'impact de l'inflation sur les salaires et les montants versés au titre des pensions.

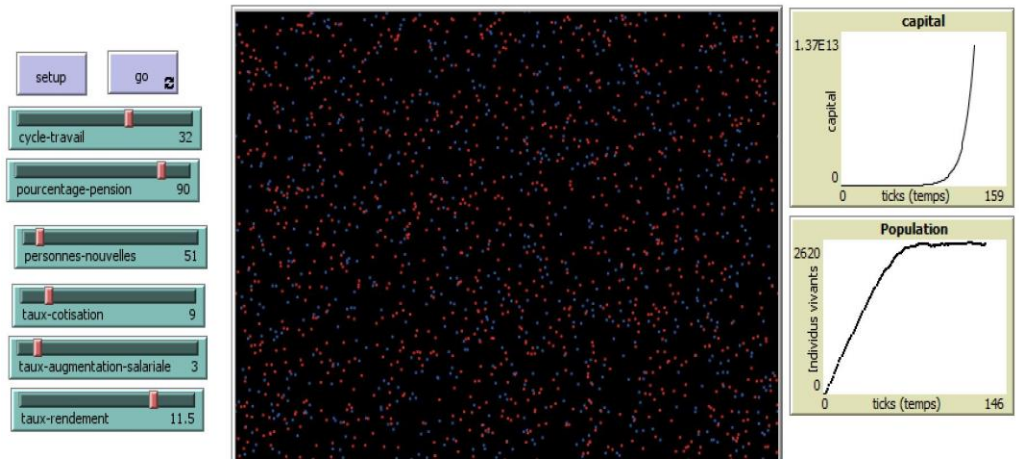


Figure 4.4 : Fenêtre d'interface du modèle 3

Ces modifications produisent très peu de changements dans le code.

```
globals[capital]
turtles-own[salaire années âge-décès]

to setup
clear-all
reset-ticks
end

to go
create-turtles personnes-nouvelles
  [ set années 25 set color red setxy random-xcor random-ycor
 set salaire one-of [12000 36000 36000 36000 60000 60000
                     60000 100000]
 set âge-décès one-of [35 40 60 60 70 70 70 80 80 80 80 90
                       90 90 90 90 100]
  ]
ask turtles [set salaire salaire + salaire * taux-augmentation-salariale / 100

   ifelse années < 25 + cycle-travail [cotiser ]
[set color blue percevoir-pension]
if années > âge-décès [die]
]
set capital capital + capital * taux-rendement / 100
if capital < 0 [show "Termine"
write "années " write ticks write " Capital: "
print capital stop]
tick
end

to cotiser
set capital capital + salaire * taux-cotisation / 100
set années années + 1
end

to percevoir-pension
set capital capital - salaire * (pourcentage-pension - taux-cotisation) / 100
set années années + 1
end
```

Explications et commentaires supplémentaires. Avec les nouveaux changements, un point critique est atteint dans les conditions suivantes:

cotisation de 9%, pourcentage du salaire perçu à titre de pension de 81% (90% moins 9% de cotisation), cycle de travail de 31 ans, mortalité aléatoire, taux de rendement du capital de 10%, augmentation annuelle de salaire de 3%.

Interactions entre agents I

Les deux exemples précédents sur le thème des fonds de pension mettent en œuvre de nombreux agents. Cependant, nous avons vu que ces agents n'interagissaient pas entre eux. En fait, dans la plupart des modèles multi-agents, les agents sont en interaction et ce de manières souvent très différentes. C'est pourquoi l'on s'attend à ce qu'un langage de programmation multi-agents offre une variété d'outils (primitives, mécanismes, etc.) facilitant la modélisation de divers types d'interactions. Ainsi, dans le modèle «Tina et Magda visitent la ville», il existe une interaction entre les deux agents par le biais de la comparaison des variables qui stockent les coordonnées des deux femmes et de la variable «distance», qui indique la distance entre les deux. La consultation ou la comparaison entre variables est l'un des mécanismes les plus fréquemment utilisés pour modéliser les interactions entre agents. Dans le modèle suivant, les tortues interagissent avec les parcelles en modifiant la valeur des variables de ces dernières.

Modèle 4: Comptage de visites pour une campagne de vaccination

Primitives: patches-own (propre-aux-parcelles), any? (certains?), not (non), max, min, max-one-of (max-un-parmi), with (avec), of (de), count (compter), opérateurs <,>,> = (supérieur ou égal), beep (bip sonore) and (conjonction et).
Autres détails: utilisation de formes conditionnelles, utilisation de «stop» pour arrêter la simulation, les tortues utilisent leur droit de modifier la valeur des variables des parcelles sur lesquelles elles se trouvent.

Dans l'exemple qui suit, nous nous inspirons de l'infrastructure urbaine du modèle «Tina et Magda visitent la ville» où deux amies parcourent les rues d'une ville de façon aléatoire. Dans le nouveau modèle les protagonistes ne sont pas Tina et Magda mais Pierre et Carmen, deux employés du Centre des Services de Santé (CSS) de la ville.

L'histoire. Les agents Pierre et Carmen, tous deux employés du CSS, doivent se rendre dans les maisons pour participer à une campagne de vaccination contre une épidémie. On se rappellera que la ville est divisée en «blocs urbains» (ou pâtés de maisons) insérés dans une topologie de carré. Chaque bloc est composé d'un certain nombre d'immeubles à appartements dans lesquels

résident les familles auxquelles Pierre et Carmen doivent rendre visite afin de vacciner les membres de la famille présents. Cependant, en raison de l'absence d'un certain nombre de résidents dont les horaires de travail ne coïncident pas avec les horaires de Pierre et Carmen, ces derniers ne peuvent souvent vacciner qu'une partie seulement des habitants des immeubles de chaque bloc où ils sont de passage. Après avoir vacciné les résidents présents dans les immeubles du bloc où ils viennent d'effectuer une visite, Pierre et Carmen se rendent dans un bloc voisin dans le but de poursuivre leurs opérations de vaccination. Et ils continuent ainsi de voyager de bloc en bloc sans se préoccuper de savoir si le prochain bloc qu'ils vont visiter a déjà été visité ou non. De ce fait, la trajectoire des visites de Carmen et de Pierre ressemble davantage à une trajectoire aléatoire qu'à une trajectoire planifiée. Cette stratégie de déplacement aléatoire d'un bloc à l'autre peut paraître étrange et on laisse le soin au lecteur d'imaginer un contexte dans lequel son adoption pourrait être justifiée (par exemple : il faut agir très rapidement, on n'a pas de données sur la densité de population de chacun des blocs et sur le nombre de personnes habituellement présentes dans les immeubles lors du passage des employés du CSS, etc.)

Une conséquence de cette façon de procéder est qu'à la fin de la campagne de vaccination, le nombre de visites reçues par chacun des blocs sera assez inégal: il y aura des blocs qui auront reçu beaucoup de visites, tandis que d'autres en auront reçu très peu, voire aucune. C'est pourquoi le CSS demande à chacun des deux agents de relever le nombre de visites qu'il a effectuées dans chaque bloc. La campagne se termine lorsque chaque bloc a été visité *au moins une fois* par l'un des agents. À la fin de la campagne, les agents communiquent les résultats des visites effectuées à la direction du CSS qui produit, à des fins statistiques, une carte en trois couleurs: les blocs ayant reçu entre 1 et 11 visites (somme des visites des deux agents) sont coloriés en jaune, ceux qui ont reçu entre 12 et 20 visites le sont en vert et ceux qui ont reçu plus de 20 visites le sont en rouge. La Figure 4.5 fournit un exemple de carte résultant d'une exécution de la simulation (la distribution des couleurs peut varier considérablement d'une exécution à l'autre).

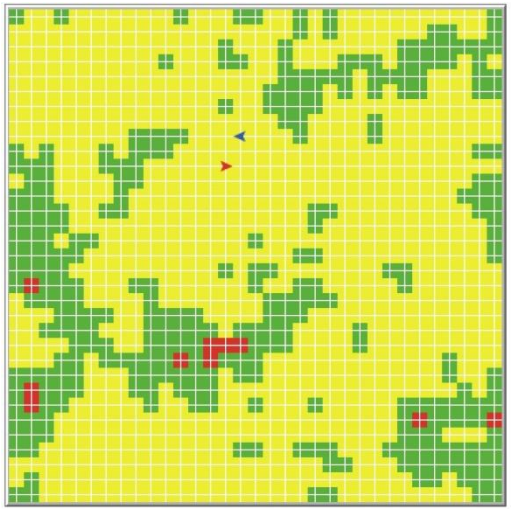


Figure 4.5 : Répartition spatiale du nombre de visites à la fin d'une simulation

Pendant que la simulation est en cours, on voit les agents en train de visiter les blocs de la ville et la couleur de ces derniers changer en fonction du nombre de visites reçues, jusqu'à ce que le processus s'arrête quand tous les blocs ont reçu *au moins* une visite. Une fois que la simulation s'arrête, il est possible de formuler des requêtes à partir de la fenêtre de l'observateur.

Activités préparatoires. Configurer le monde avec la topologie du carré. Construire les boutons «setup» et «go» (cochez «Forever» - en continu – dans le bouton «go»).

Plan général et problèmes à résoudre. Il faut tenir un registre du nombre de fois que Pierre et Carmen ont effectué une visite dans chacun des blocs de la ville. Chaque parcelle («patch») représente un bloc et ce sont les parcelles qui sont chargées de comptabiliser le nombre de fois qu'un agent est venu les visiter. Pour ce faire, on assigne deux variables à chaque bloc (parcelle) à l'aide de la primitive «patches-own», variables que l'on appelle «visitesP» et «visitesC». Ces deux variables servent à compter le nombre de visites respectives de chacune des parcelles par Pierre et Carmen. Dans le langage courant, la logique du modèle s'interprète ainsi:
si tu es Pierre (si ton numéro «who» = 0), alors augmente d'une unité la valeur de la variable visitesP de la parcelle sur laquelle tu te trouves ; si tu n'es pas Pierre (et donc si tu es Carmen), alors augmente d'une unité la valeur de la variable visitesC de cette parcelle.
Exprimé en ligne de code, ceci se traduit par:

```
Ask turtles [ifelse who = 0 [set visitesP  visitesP + 1] [set visitesC  visitesC + 1].
```

Notez que, dans cet ordre, il est demandé à chaque tortue d'augmenter d'une unité la valeur de la variable visitesP ou celle de visitesC, selon que son nombre

«who» égal à 0 ou non. Ces deux variables appartiennent aux parcelles («patches-own») et non aux tortues. Nous sommes ici en présence de la faculté qui est donnée aux tortues de traiter les variables de la parcelle sur laquelle elles se trouvent comme si c'était leurs propres variables.

Voici le code au complet:

```
patches-own[visitesP visitesC]
;; visitesP et visitesC sont des variables qui stockent le nombre de visites
;; de la tortue 0 (Pierre) et de la tortue 1 (Carmen) respectivement

to setup
  clear-all
  crt 2 [set heading 0 ]
blocs-urbains ;; on appelle la procédure qui construit les blocs urbains
ask turtle 0 [pu set xcor -15  set ycor -15 set color blue]
  ask turtle 1 [pu set color red]
  reset-ticks
end

to go
  ask turtles [ifelse who = 0 [set visitesP visitesP + 1 ]
    [set visitesC visitesC + 1 ]
    fd 1 rt one-of [90 -90 0]]
    ask patches [if visitesP + visitesC > 0
    and visitesP + visitesC < 12 [set pcolor yellow]
      if visitesP + visitesC >= 12 and visitesP + visitesC < 21
     [set pcolor green]
      if visitesP + visitesC >= 21 [set pcolor red]]
    if not any? patches with [pcolor = gray] [beep stop]
    wait 0.01 ;; pour ralentir le mouvement des tortues
    tick
end

to blocs-urbains ;; cette procédure construit les blocs urbains
  ask patches [set pcolor gray]
  ask turtle 0 [set color white pu set heading 0
    set xcor -16 set ycor -16 pd
    repeat 17 [fd 33 rt 90 fd 1
      rt 90 fd 33 lt 90 fd 1 lt 90 fd 33] bk 33 lt 90
    repeat 17 [fd 32 rt 90 fd 1
rt 90 fd 33 lt 90 fd 1 lt 90 fd 33] bk 33 lt 90
    repeat 17 [fd 32 rt 90 fd 1
```

```
    rt 90 fd 33 lt 90 fd 1 lt 90 fd 33]
     ]
end
```

Commentaires et explications supplémentaires. Les agents commencent leurs visites des parcelles en partant de différents points de la ville. Chaque parcelle se charge de compter les visites des deux agents à l'aide du mécanisme précédemment expliqué. Lorsque toutes les blocs ont reçu *au moins* une visite, c'est-à-dire lorsqu'il ne reste plus de parcelles grises, la procédure produit un bip et s'arrête (code : if not any? patches with [pcolor = grey] [beep stop]). L'expression «if not any ?» doit être comprise comme «si aucune» ou encore «s'il n'en reste plus aucune». Une fois la simulation exécutée, il est possible de recueillir plusieurs types d'informations sur les visites en consultant la fenêtre de l'observateur. Ainsi, par exemple, il est possible de connaître la moyenne, l'écart type et la variance des visites des blocs effectuées par chaque agent, à l'aide des primitives «mean», «standard-deviation» et «variance». Voici les résultats de quelques-unes de ces requêtes en utilisant les données obtenues à partir d'une exécution de la simulation:

print count patches
==> 1089, la totalité des parcelles du monde (par omission)
print (count patches with [pcolor = yellow]
==> 783, nombre de blocs jaunes.
print (count patches with [pcolor = yellow] / count patches) * 100
==> 71.9%, pourcentage de blocs jaunes.
print mean [visitesP] of patches with [pcolor = green]
==> 6.81, nombre moyen de visites effectuées par l'agent 0 (Pierre) dans les blocs verts
print standard-deviation [visitesP] of patches with [pcolor = green]
==> 3.05, est l'écart type des visites faites par Pierre aux blocs verts.
print max [visitesP + visitesC] of patches
==> 24, nombre maximum de visites reçues par un ou plusieurs blocs. L'information fournie indique qu'au moins un bloc a été visité 24 fois. Il est possible d'identifier ce bloc (ou l'un de ces blocs).
print max-one-of patches [visitesP + visitesC]
==> (patch -1 - 6), le bloc -1 -6 a reçu le nombre maximum de visites (24). Si plus d'un bloc a été visité 24 fois, le système en choisit un au hasard.
La commande «max-one-of agents [variable]» rapporte l'un des agents qui possède la valeur maximale de la variable entre crochets.
print count patches with [visitasP + visitasC = 24]
==> 1, un seul bloc seulement a reçu 24 visites.
print count patches with [visitesP + visitesC = 1]
==> 5, seulement 5 des 1089 blocs n'ont reçu qu'une seule visite.

print ticks
==> 5171, durée, en ticks, de la simulation ou bien encore nombre de visites effectuées par chaque agent.

Exercice 4.2
1- Si la simulation est exécutée plusieurs fois, pourquoi les premiers blocs verts ou rouges apparaissent-ils souvent sur les bords ou sur les coins?

2- Pourquoi la moyenne des visitesP est-elle égale à celle des visitesC, mais l'écart-type ne l'est pas?

3- Calculer le mode de visitesC. Cette valeur correspond au nombre de visites du bloc le plus visité par Carmen au cours du processus.

4- Les agents Carmen et Pierre sont traités sur un pied d'égalité dans le code. Toute différence dans la manière dont ils ont accompli leur tâche est due exclusivement au hasard et devrait disparaitre si l'on faisait de nombreuses simulations. Cependant, certaines différences peuvent être détectées au cours d'une seule simulation. En tant que responsable du CSS, serait-il possible de concevoir un (ou des) critère(s) permettant de déterminer lequel des deux agents a été plus efficace dans son travail, en regardant la carte et en analysant les informations fournies par l'observateur à la fin d'une seule simulation?

Interactions basées sur la proximité spatiale

Il existe un type d'interaction qui prend fréquemment place parmi les agents d'un modèle, où l'espace joue un rôle important dans l'interaction. Il peut s'agir d'un espace physique réel (bidimensionnel ou tridimensionnel) ou d'un espace abstrait représenté à l'aide des parcelles. Dans ce type d'interactions, la proximité spatiale entre les agents, ainsi que leur position ou orientation relative, peuvent être un facteur important de l'interaction. Dans un tel contexte, un agent agit (ou réagit) en fonction de la situation qui prévaut dans son «voisinage». Ce voisinage est défini par une topologie (cercle, cône) et une taille particulière ou par une relation spatiale qui permet aux agents d'identifier ce qui se passe en avant, à droite, à gauche ou dans une certaine direction et à une certaine distance d'eux. NetLogo a une variété de primitives qui permettent de modéliser ces interactions spatiales.

Tout cela peut être utile pour modéliser des phénomènes physiques tels que le comportement de particules dans des champs d'attraction-répulsion ou dans

des univers virtuels, où l'espace peut ne pas être homogène ou isotrope[25]. Le modèle présenté ci-dessous (Modèle 5 : Alfions et bétions) illustre un cas de ce type où les interactions présentent une asymétrie locale: dans cet exemple, un agent réagit différemment selon qu'il détecte un agent situé à sa gauche ou un agent situé à sa droite.

Dans les modèles qui suivent, nous utiliserons les primitives: «any?» (un(e)-quelconque?), «turtles-on» (tortues-sur), «patch-ahead» (parcelle-devant) et «patch-left-and-ahead» (parcelle-à-gauche-et-devant) pour modéliser les interactions. La primitive «turtles-on» rapporte l'ensemble-agents des tortues situés sur la parcelle indiquée comme entrée. Par exemple, «turtles-on patch 1 4» indique l'ensemble-agents des tortues qui se trouvent sur la parcelle de coordonnées (1, 4). La primitive «patch-ahead num» rapporte la parcelle qui se trouve «num» pas en avant de l'agent qui a passé la commande. Ces trois primitives peuvent être combinées dans des expressions telles que:

Ifelse any? turtles-on patch-ahead 2 [commandes si vrai] [commandes si faux]

qui se traduit en français par: s'il y a une tortue quelconque sur la parcelle, deux pas devant moi [commandes s'il y a des tortues] [commandes s'il n'y en a pas].

Avant d'exposer le modèle suivant, nous présentons une liste de primitives NetLogo pouvant être utilisées pour modéliser des interactions de proximité spatiale, extraites du dictionnaire des primitives:

distance (distance), downhill (en-descente), downhill4 (en-descente4), uphill (en-montée), uphill4 (en-montée4), face (regarder-vers), facexy (regarder-vers-xy), in-cone (intérieur-cone) move-to (se-diriger-vers), patch-ahead (parcelle-devant), patch-at (parcelle-à), patch-at-heading-and-distance (parcelle-vers-direction-et-distance), patch-here (parcelle-ici), patch-left-and-ahead (parcelle-à-gauche-et-devant), patch-right-and-ahead (parcelle-à-droite-et-devant), towards (vers), towardsxy, (vers-xy), neighbors (voisins), neighbors4 (voisins4).

Les primitives liées aux liens (qui sont aussi un puissant outil de modélisation des interactions entre agents) n'ont pas été incluses dans cette liste. Une description de ces primitives figure dans le dictionnaire de NetLogo (catégorie intitulée «Link-related») auquel nous invitons le lecteur à se rapporter.

[25] Un espace isotrope est un espace vide qui présente les mêmes propriétés dans toutes les directions.

Modèle 5 : Alfions et bétions

Primitives: patch-left-and-ahead (parcelle-à-gauche-et-devant), any? (certains?), patch-ahead (parcelle-devant), breeds-on (familles-sur).
Autres détails: on crée deux familles de tortues pour représenter des particules qui ont des comportements distincts.

L'histoire. Dans ce modèle, il y a deux types de particules élémentaires imaginaires appelées «alfions» et «betions», qui interagissent les unes avec les autres selon certaines règles fondées sur leur proximité spatiale. Les alfions répondent à des lois d'interaction asymétrique car elles ne réagissent que lorsqu'elles détectent des particules en avant d'elles et à leur gauche. Les bétions ont un comportement mixte car elles réagissent asymétriquement face aux alfions, mais de manière symétrique devant les particules de leur catégorie. La réaction de chaque particule dépendra de savoir si la particule détectée est de la même espèce ou non et si elle est devant et à sa gauche ou non.

Plan général et problèmes à résoudre. Les particules réagissant différemment, nous devons définir deux familles d'agents, une pour chaque type de particule. Les particules se distinguent visuellement par leur couleur: les alfions sont jaunes et les bétions rouges. Pour modéliser les interactions entre les particules, on utilise les primitives «turtles-on», «patch-ahead num» et «patch-left-and-ahead». Pour les interactions de type asymétrique décrites dans l'histoire, on utilise la primitive «patch-left-and-ahead» («parcelle-à-gauche-et-devant»), qui fonctionne avec deux entrées qui sont l'angle à gauche et la distance en avant mesurée en pas.

Dans l'exemple, cette primitive est utilisée avec les entrées «patch-left-and-ahead 90 1», laquelle rapporte la parcelle située à une distance d'un pas et à 90 degrés à gauche de l'agent qui émet la commande. On demande à chaque particule s'il y a des particules à un pas de distance et à leur gauche. Dans le cas des alfions, par exemple, la question se présente sous la forme:

```
ifelse any? alfions-on patch-left-and-ahead 90 1
```

dont la traduction en français est: s'il y a des alfions sur la parcelle située 90 degrés à gauche et à un pas de distance. La commande complète comprend alors deux conditions, une condition «if» encapsulée dans une condition «ifelse »

```
ask alfions [ifelse any? alfions-on patch-left-and-ahead 90 1 [left 90 fd 1]
[if any? bétions-on patch-left-and-ahead 90 1 [rt 90 fd 1] fd 1] ]
```

La commande pour les bétions a une structure similaire, avec toutefois quelques différences dans les actions que les particules sont invitées à exécuter.

Activités préparatoires. Construire les boutons «setup» et «go» (ce dernier activé en mode continu «Forever») ainsi que les curseurs pour les variables nombre d'alfions et nombre de bétions .

Le code est le suivant:

```
breed [alfions alfion]
breed [bétions bétion]

to setup
clear-all
create-alfions nombre-alfions [setxy random-xcor random-ycor set color yellow]
create-bétions nombre-bétions [setxy random-xcor random-ycor set color red]
end

to go
ask alfions [ifelse any? alfions-on  patch-left-and-ahead 90 1
[left 90 fd 1] [if any? bétions-on patch-left-and-ahead 90 1
[rt 90 fd 1] fd 1]
 ]
ask bétions [ifelse any? alfions-on  patch-left-and-ahead 90 1
[lt 90 fd 1] [if any? bétions-on patch-ahead 1 [bk 1] fd 1]
 ]
wait 0.1
end
```

Explications et commentaires supplémentaires. Pour faire en sorte que les commandes passées aux alfions soient ignorées par les bétions ou vice-versa, deux familles de particules appelées alfions et bétions ont été créées. Pour émettre des commandes qui ciblent des catégories particulières d'agents on peut également faire appel à certaines caractéristiques distinctives des agents (par exemple, la couleur), telles que «ask turtles with [color = red]». Notez qu'une fois les familles créées, les primitives faisant référence aux familles par leur nom s'appliquent automatiquement à ces familles. Ainsi, par exemple, «create alfions», «alfions-on» ou «ask alfion 0» sont des expressions reconnues par le code. Les variables globales «nombre-alfions» et «nombre-bétions» sont définies au moyen de deux curseurs dans l'interface. Etant donné que les alfions

sont créées avant les bétions, elles se voient attribuer les premiers numéros «who» de la liste.

Suggestions d'exploration. Notez la différence de comportement des deux types de particules. Observez le déroulement de la simulation et essayez d'expliquer le comportement des bétions, qui s'arrêtent et restent immobiles pendant des périodes de temps relativement longues. Le modèle se prête également très bien à l'observation des changements de comportement qui se produisent lorsque l'on fait varier l'une des entrées de la primitive «patch-left-and-ahead num num» . Il est également possible d'expérimenter en modifiant certaines des commandes des expressions conditionnelles entre crochets. Pour suivre la trajectoire typique des alfions, l'on peut demander à un alfion d'abaisser sa plume: ask one-of alfions [pd].

Modèle 6: Le noyau du diable

Primitives: patch-left-and-ahead, any? shape (forme), dot (point), distancexy.

L'histoire. Dans cette extension du modèle sur les particules élémentaires imaginaires, la nouveauté consiste en l'introduction d'une particule centrale, également appelée «noyau du diable» (très différente du boson de Higgs, appelée «la particule de Dieu»). Le noyau du diable semble imprévisible, attire et repousse parfois les autres particules et sa couleur est blanche.

Voici le code :

```
breed [alfions alfion]
breed [bétions betion]

to setup
clear-all
crt 1 [set color white] ;; c'est le noyau du diable
create-alfions nombre-alfions [setxy random-xcor
random-ycor set color yellow]
create-bétions nombre-bétions [setxy random-xcor
random-ycor set color red]
ask turtles [set shape "dot"]
ask turtle 1 [pu] ;; si l'on désire voir la trajectoire
;; d'un alfion, il suffit de remplacer pu par pd
end


to go
```

```
ask alfions [if distancexy 0 0 < 6 [rt 180 ]
ifelse any? alfions-on patch-left-and-ahead 90 1
[left 90 fd 1] [if any? bétions-on patch-left-and-ahead 90 1
[rt 90 fd 1] fd 1]
]
ask bétions [if distancexy 0 0 < 6 [rt 180 ]
ifelse any? alfions-on patch-left-and-ahead 1 90
[lt 90 fd 1] [if any? bétions-on patch-ahead 1 [fd 1] fd 1]
]
wait 0.1
end
```

Explications et commentaires supplémentaires Ce code diffère du précédent par l'ajout de quelques nouvelles commandes. Le noyau du diable étant situé sur la parcelle (0, 0), il est clair que la commande interdisant aux particules d'approcher du noyau est «if distancexy 0 0 < 6 [rt 180]». L'exécution du programme met en évidence un phénomène intéressant : certaines particules ne sont pas rejetées par le noyau et restent emprisonnées à proximité, dans un mouvement vibratoire continu. Au bout d'un moment, certaines de ces particules parviennent à s'échapper et à rejoindre l'ensemble des particules externes. Il arrive également que d'autres particules éloignées du noyau se rapprochent trop de lui et se retrouvent piégées. Un tel comportement semble aller à l'encontre des indications du code, mais une analyse attentive de la dynamique des particules, permet de comprendre pourquoi il en est ainsi.

Bien qu'il n'ait pas été conçu dans ce but, le présent modèle fournit un exemple intéressant de comportement émergent et une analogie inattendue avec le comportement des électrons des orbites externes des atomes. Ces électrons sont normalement piégés dans des orbites autour du noyau, mais ils peuvent éventuellement les quitter lorsqu'ils entrent en collision avec des électrons libres ou lorsqu'ils sont capturés par un atome voisin, lequel préfère capturer des électrons plutôt que de les laisser aller.

Le point important à souligner ici est le fait que l'exemple peut nous faire prendre conscience que de petits changements dans le comportement des agents peuvent produire des changements importants au niveau macro. Nous invitons les lecteurs et lectrices à expérimenter avec le modèle en faisant varier la distance de rejet du noyau (qui ne doit pas nécessairement être identique pour les deux particules) ou en ne permettant que le rejet des particules d'un type donné. Nous suggérons de remplacer l'angle «rt 180» (dans la procédure «go») par la valeur 90 et d'observer l'évolution du comportement des particules. En permettant de faire varier ses paramètres ou d'en modifier les commandes, le modèle se prête bien à la conduite d'études exploratoires.

Exercice 4.3 Expliquez pourquoi certaines particules restent emprisonnées à proximité du noyau du diable, animées d'un mouvement vibratoire continu et pourquoi, au bout d'un moment, certaines d'entre elles parviennent à s'échapper.

Modèle 7: Concert de rock à Whoolsock I

Primitives: reset-ticks, tick, ifelse, any? (certains?), turtles-on (tortues-sur), patch-ahead (parcelle-devant), random-xcor (aléatoire-xcor), random-ycor, watch (regarder).
Autres détails: la variable globale «fans» est définie par un curseur dans l'interface.

L'histoire Dans une ferme d'une grande ville appelée Whoolsock, un grand concert de rock du célèbre groupe des Stunning Rolls doit se dérouler. Comme l'accès à la zone se trouve du côté ouest et que la scène où le groupe va jouer se trouve du côté est, la foule se déplace d'ouest en est sur le terrain de la ferme. En raison du grand nombre de personnes qui marchent sur le terrain, les promeneurs changent de direction à plusieurs reprises pour éviter les collisions.

Quand la foule entend Jick Magger, le chanteur du groupe, réglant sa guitare, les cris et la joie sont indescriptibles. Le nombre de personnes participant au concert est fixé à l'aide du curseur «fans» de l'interface. Chaque fois que la procédure «go» est lancée, chaque tortue (personne) vérifie s'il y a une autre tortue sur la parcelle juste devant elle («patch-ahead»). Si la réponse est négative, la personne avance d'un pas en avant, mais si la réponse est affirmative la personne adopte, pour éviter la collision, une nouvelle orientation au hasard («random 180») et recommence d'essayer de faire un pas en avant en formulant la même question.

Plan général et problèmes à résoudre. La promenade des tortues doit être modélisée de manière à éviter les collisions entre tortues. Ces dernières se déplacent d'ouest en est, et, dans un tel contexte, leur trajectoire peut difficilement être en ligne droite. Pour éviter les collisions, une expression conditionnelle est utilisée par laquelle chaque tortue demande s'il y a d'autres tortues sur la parcelle situé un pas devant elle. Si la réponse est positive, la tortue choisit une nouvelle orientation au hasard et formule la même demande. La tortue avance d'un pas seulement lorsque la réponse est négative. Ce comportement est résumé par la commande:

```
ask turtles [ifelse any? turtles-on patch-ahead 1 [set heading random 180] [fd 1]]
```

qui se traduit par:

demander aux tortues [s'il y a des tortues dans la parcelle située 1 pas en avant [prender une orientation au hasard 180] [avancer 1 pas]]

Activités préparatoires. Construire les boutons «setup» et «go» ainsi qu'un curseur appelé «fans» qui détermine le nombre de personnes qui assistent au concert. Cocher la case «Forever» du bouton «go».

Le code est le suivant :

```
to setup
clear-all
crt fans [setxy random-xcor random-ycor set color yellow]
ask turtle 0 [set color red]
reset-ticks
end

to go
ask turtles [ifelse any? turtles-on patch-ahead 1
[set heading random 180]
[ fd 1]
]
wait 0.1
watch turtle 0;; trace un cercle autour de la tortue dont on veut suivre la
 ;; trajectoire
tick
end.
```

Explications et commentaires supplémentaires. Les positions initiales des tortues sont fixées au hasard. La raison pour laquelle la variable «tick» a été incluse dans la procédure go est que cela permet de compter le nombre de fois que l'interprète passe «go» afin de pouvoir mesurer, si nécessaire, le temps qui s'écoule. Cette information sera plus utile dans la version II du modèle (Modèle 8). La tortue 0 a été peinte en rouge, pour pouvoir suivre la trajectoire d'une tortue typique à l'aide de la primitive «watch» (regarde) qui dessine un cercle rouge autour de ladite tortue.

Exercice 4.4 Modifiez légèrement le code pour faire en sorte que les personnes se déplacent du nord au sud.

Modèle 8: Concert de rock à Whoolsock II

Primitives: Il n'y a pas de nouvelles primitives
Autres détails: Deux variables globales sont créées pour mesurer et comparer la longueur des trajectoires en ligne droite d'une tortue typique.

L'histoire. Dans l'exemple précédent on a fait remarquer combien il était difficile pour les promeneurs soucieux d'éviter des collisions de conserver une trajectoire en ligne droite. Ce qui ne signifie pas qu'ils sont sans cesse en train de zigzaguer: il leur arrive parfois de faire de nombreux pas en ligne droite avant de devoir dévier de leur chemin.
Dans cet exemple on va se poser la question: quelle est, au cours d'une simulation, la longueur maximale d'un trajet en ligne droite parcouru par une tortue typique ? Pour ce faire, nous allons étudier la longueur des trajets en ligne droite que les personnes parcourent lorsqu'elles se déplacent sur le terrain ou a lieu l'évènement. En particulier, nous suivons le déplacement d'une tortue représentative de l'ensemble, en supposant que toutes les tortues ont un comportement «statistiquement équivalent». Chaque fois qu'elle termine un trajet en ligne droite, la tortue représentative (tortue 0) doit enregistrer la longueur de ce trajet, mesurée en pas, dans une variable appelée « CeTrajet». Dans cette variable, on n'enregistre pas la longueur de tous les trajets en ligne droite, mais seulement celle du dernier trajet effectué par la tortue. Ainsi, chaque fois que la tortue commence un nouveau trajet, elle efface la longueur de celui stocké dans la variable et la remplace par la longueur du trajet qu'elle vient de terminer. En outre, nous demandons à chaque tortue d'enregistrer, dans une autre variable appelée « TrajetMax», la longueur du plus grand trajet rectiligne effectué à ce jour. La valeur de cette variable peut être imprimée dans le terminal d'instructions (à l'aide de la commande «type») ou faire l'objet d'un bouton de type «reporter» qui permet d'en suivre l'évolution. Cela nous permet de voir si les lignes droites s'allongent et à quel rythme elles le font. La simulation peut être arrêtée de deux manières : soit manuellement par l'utilisateur en appuyant sur le bouton «Go», soit en insérant la commande «if ticks > num [stop]» (ou num représente le nombre de ticks après lequel l'on désire arrêter la simulation) dans la première ligne de la procédure «go».

Plan général et problèmes à résoudre. Dans cette nouvelle version du modèle, la réponse à la question posée réside dans la manière dont la tortue 0 manipule les deux variables, CeTrajet et TrajetMax. Nous décrivons la solution montrant ce que la tortue 0 doit faire avec ces deux variables quand elle est en train de se déplacer sur le terrain. Tout d'abord, quand la tortue 0 effectue un nouveau trajet en ligne droite à partir de la parcelle ou elle se trouve actuellement, la

variable CeTrajet doit prendre une valeur égale à la longueur de ce nouveau trajet. La tortue 0 procède alors comme suit:

1. Si elle peut faire un nouveau pas en avant, alors elle doit augmenter d'une unité la valeur de la variable CeTrajet (ligne de code : fd 1 set CeTrajet CeTrajet + 1).
2. Si elle ne peut pas pas avancer car il y a des tortues sur la parcelle qui se trouve devant elle, elle doit, avant de changer de direction pour trouver un nouveau trajet en ligne droite, comparer la valeur de la variable CeTrajet à celle de TrajetMax. Si la valeur de CeTrajet est supérieure à celle de TrajetMax, la valeur de cette dernière variable est remplacée par celle de CeTrajet. En code:

if CeTrajet > TrajetMax [set TrajetMax CeTrajet]

3. Avant de commencer le nouveau trajet en ligne droite, il faut attribuer la valeur 0 à la variable CeTrajet et essayer une nouvelle direction dans laquelle il est possible d'avancer:

set CeTrajet 0 set heading random 180.

Les trois expressions précédentes sont incorporées dans le code et les lecteurs peuvent les identifier sans problème. La comparaison des variables indiquées au point 2 est effectuée à l'aide d'une procédure distincte appelée «comparer».

Activités préparatoires. Les boutons «setup» et «go» sont configurés (cochez «Forever» dans l'éditeur du bouton «go») et un curseur appelé « fans» est construit afin de choisir le nombre de participants au spectacle (ce nombre est une variable globale). La topologie du monde est celle d'un tore.

Le code est le suivant :

```
globals[TrajetMax CeTrajet]

to setup
clear-all
crt fans [setxy random-xcor random-ycor set color yellow]
ask turtle 0 [set color red pd]
reset-ticks
```

```
end

to go
ask turtle 0
[
ifelse any? turtles-on patch-ahead 1
[comparar set CeTrajet 0 set heading random 180]
[ fd 1 set CeTrajet CeTrajet + 1 ]
ask other turtles [ifelse any? turtles-on patch-ahead 1
[set heading random 180][fd 1]
]
]
wait 0.1
tick
end

to comparer
if CeTrajet > TrajetMax [set TrajetMax CeTrajet
show TrajetMax]
end

to résultat
type “Le trajet rectiligne le plus long à ce jour est de” type “ “ type TrajetMax
type “ pas”
;; taper «résultat» (sans les guillemets) dans le centre de commande
end
```

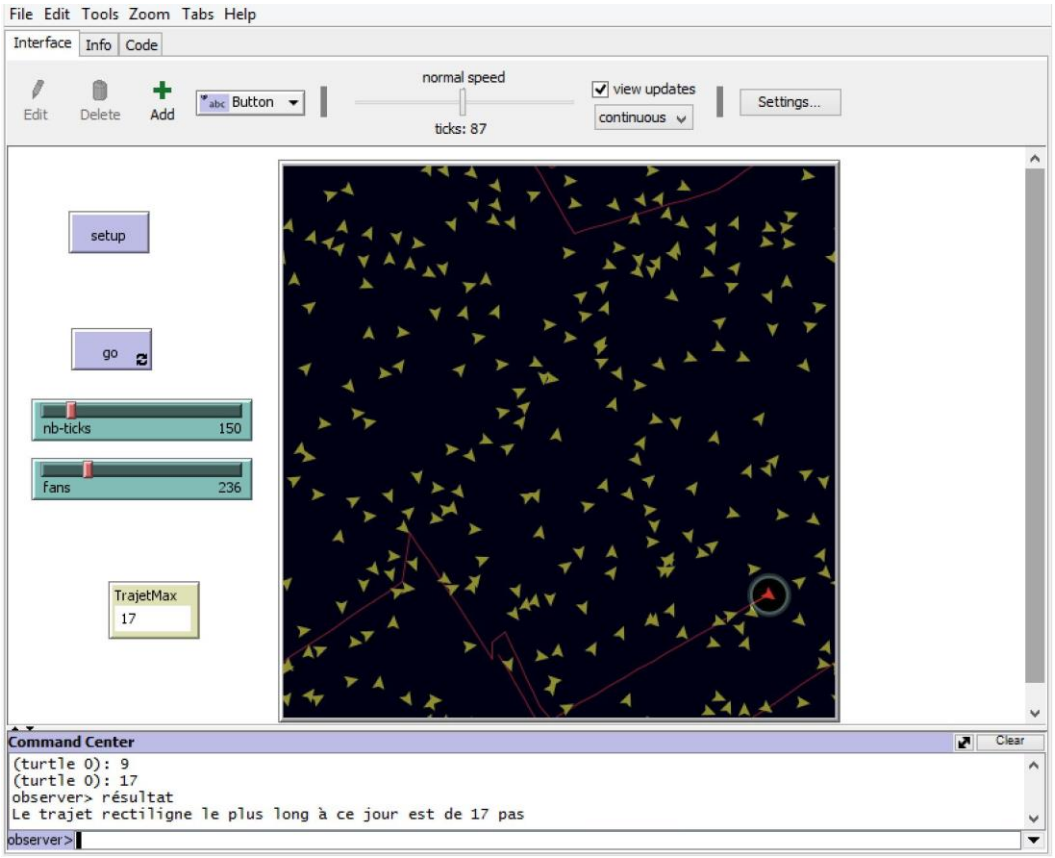


Figure 4.6 Trajectoire (en rouge) suivie par la tortue 0 après un certain nombre de passages par «go» (nombre de ticks).

Explications et commentaires supplémentaires. L'ordre dans lequel les tortues posent la question de savoir si d'autres tortues sont présentes sur la parcelle en avant d'elles poserait un problème si la topologie du monde était celle d'un carré. En effet, quand une tortue qui se déplace heurte la frontière du monde, la question « y a-t-il des tortues sur la parcelle en avant de moi ?» n'a pas de sens car il n'y a aucune parcelle en avant de la tortue. Pour éviter l'erreur engendrée par cette situation, le modèle doit correspondre à la topologie du tore.

Exercice 4.3. Modifiez le code afin que le modèle puisse être exécuté avec la topologie du carré.

Exercice 4.4 Expliquez le phénomène suivant: si le programme est exécuté plusieurs fois avec seulement quelques tortues, on remarque que certaines tortues changent « spontanément» de direction sans qu'il y ait de tortue en avant d'elles et avec laquelle elles peuvent entrer en collision. Ce comportement, apparemment étrange, est plus facile à observer si on utilise le modèle avec une seule tortue, qui, en tant que seule occupante du monde, n'a aucune raison de changer de direction, car elle ne peut entrer en collision avec

d’autres tortues. Cependant, si le modèle est exécuté plusieurs fois, on observe que des changements de direction que l’on pourrait qualifier de « spontanés» continuent de se produire. Pour être cohérent avec l'histoire initiale, on pourrait imaginer que les tortues changent de direction parce qu'elles ont décidé d'éviter une flaque d'eau ou un rocher sur le terrain. Nous invitons les lecteurs à trouver la raison informatique de ce comportement. Il s'agit d'un exemple de problèmes qui ne sont pas anticipés lors de l'écriture du code d'un modèle, et qui, lorsqu’ils sont découverts lors de l'exécution du modèle, doivent faire l’objet d’analyse approfondies.

Exercice 4.5. Si une liste était utilisée pour stocker les longueurs de chaque trajet rectiligne en créant une nouvelle variable appelée ListeDesTrajets, il ne serait pas nécessaire de définir la variable TrajetMax car la valeur du trajet rectiligne le plus long pourrait être obtenu à l’aide de la primitive « max» , qui indique la valeur maximale d'une liste. Le coût de calcul du recours à cette primitive est celui d'une augmentation de la mémoire requise pour l'exécution du modèle, car la variable ListeDesTrajets voit son nombre de valeurs accumulées croître au fur et à mesure de l'évolution du programme. Modifiez le code en appliquant ce mécanisme (utilisation de la primitive «max»).

Promenade I

La programmation met l’esprit en mouvement comme peu d’activités sont capables de le faire. Dès que nous avons terminé la construction d'un modèle ou pendant le processus de construction, nous sommes naturellement attirés par les aspects liés au modèle lui-même, ou même par des questions soulevées par le modèle dans d'autres domaines du savoir. Lorsque nous avons créé le modèle Concert à Whoolsock I, l’idée de départ était très simple: observer le comportement d’une population de tortues qui déambulaient dans une direction cardinale (d’ouest en est dans l’exemple) en évitant les collisions entre ses membres et modéliser ce comportement à l’aide de NetLogo. Mais dès que l’on voit le modèle fonctionner, des idées et des questions commencent immédiatement à surgir. À quoi pourrait ressembler une trajectoire typique des tortues? Pour le savoir il suffit de demander aux tortues de marcher avec le stylet abaissé. En voyant l'enchevêtrement de lignes produit par une telle commande, on est porté à penser qu'il vaudrait mieux se contenter d’observer la trajectoire d'une seule tortue. Mais laquelle d'entre elles? Pouvons-nous choisir n’importe laquelle? Toutes les trajectoires sont-elles «statistiquement équivalentes»? Qu'entend-on par trajectoires «statistiquement équivalentes»? Ces questions nous amènent à nous poser des questions relevant du domaine des statistiques. Nous exécutons alors le modèle plusieurs fois pour observer la trajectoire d’une seule tortue (la tortue 0) en faisant l’hypothèse qu’elle est représentative de l’ensemble des tortues en mouvement. Nous avons fait

quelques tests avec la couleur du trait laissé par cette tortue représentative et nous avons finalement opté pour le rouge, qui ressort bien sur le fond noir du monde (le blanc aurait pu aussi bien faire l'affaire).

A ce stade, la question que l'on peut se poser est de connaître la longueur du segment rectiligne le plus long de la trajectoire que dessine la tortue 0 au cours d'une simulation. Il semble évident que plus le temps d'exécution de la simulation est élevé (plus le nombre de «ticks» est grand), plus long est le tronçon rectiligne maximal observé.

Si nous stockons les longueurs d'un grand nombre de lignes droites, quelles sont alors la moyenne et l'écart type de ces longueurs? Comment cette moyenne varie-t-elle dans le temps ou par rapport au nombre de tortues? La moyenne ou l'écart-type finissent-ils par atteindre des valeurs stables? Il semble aller de soi que moins le nombre de tortues en mouvement est élevé, plus grande devrait être, en moyenne, la longueur des trajectoires rectilignes. Mais peut-on estimer quantitativement cette relation ?

Quiconque a déjà construit des modèles de ce type (que ce soit dans l'environnement NetLogo ou dans tout autre environnement) a certainement eu l'occasion de faire l'expérience de ce type de gymnastique mentale. Plus loin dans le livre, après avoir présenté plusieurs modèles, nous reviendrons sur ce sujet et sur le potentiel qu'il a en matière d'éducation. Nous ne pouvons pas manquer de souligner que la création de modèles présente également un potentiel énorme dans le domaine de l'expression artistique et du plaisir de la création, à l'instar d'activités telles que la peinture, la pratique d'un instrument, l'écriture ou la composition musicale.

Contrairement aux langages de programmation généraux, NetLogo dispose d'un ensemble d'outils préinstallés qui invitent les utilisateurs à créer des modèles, qu'il s'agisse de modèles simulant des aspects de la réalité ou de modèles représentant des mondes virtuels ou fantaisistes dont les agents sont les protagonistes principaux. Mais ce n'est pas un hasard: en observant la réalité de près, force est de constater que le monde qui nous entoure est constitué d'agrégats, de pluralités qui fourmillent d'exemples d'agents: les salariés d'un bureau, les électeurs du maire d'une ville, les molécules d'un vase de cristal, les particules élémentaires de la matière, les pages Web d'Internet, les prédateurs et leurs proies dans une forêt, les fourmis dans une fourmilière, les véhicules dans un réseau routier, les gènes dans l'ADN, les magasins d'un centre commercial, les marchandises d'un magasin, les comptes des clients d'une banque, les livres de votre bibliothèque personnelle, etc.

La programmation, et plus particulièrement la programmation de modèles, agit comme un aimant qui a le mérite d'attirer des idées dans les domaines les plus divers du savoir et de l'activité humaine.

Chapitre 5 : Modèles II

Introduction

Dans ce chapitre, nous présentons un deuxième ensemble de modèles un peu plus élaborés que ceux du chapitre précédent. Il reste encore à voir d'importantes primitives et d'autres aspects du langage qui nous fourniront les outils de base pour aborder la construction d'un plus large éventail de programmes et de modèles.

Dans les exemples du chapitre précédent, nous avons déjà remarqué que les obstacles les plus importants que l'on rencontre lors de la construction d'un modèle ne sont pas ceux liés à la connaissance et à la maîtrise du langage de programmation, mais ceux concernant les problèmes posés par le modèle lui-même. Les exemples qui suivent font appel à l'utilisation de nouvelles primitives et de nouvelles ressources d'interface, mais ils reposent sur aussi l'utilisation de certaines techniques de programmation basées sur les ressources du langage NetLogo précédemment introduites et grâce auxquelles il est possible de résoudre les problèmes présentés par les modèles.

Ces problèmes sont intrinsèques au modèle lui-même et la conception des stratégies de solution est en grande partie - bien que, comme nous ne le verrons, pas dans son intégralité – indépendante du langage dans lequel le modèle est programmé. Une fois que les plans d'attaque ont été formulés pour résoudre les problèmes soulevés par le modèle, ceux-ci doivent être traduits en code NetLogo. Parce que la variété des problèmes que peuvent présenter les modèles est énorme, il est pratiquement impossible de les aborder au moyen d'une liste de recettes ad hoc. L'ingéniosité est un outil que nous ne pouvons pas garder sous clé. Comme c'est le cas pour l'apprentissage de n'importe quel sujet, les styles individuels d'apprentissage et de résolution des problèmes se manifestent également dans la programmation.

Certains programmeurs commencent à programmer en écrivant les premiers «blocs» de code qui leur viennent à l'esprit. D'autres préfèrent faire une planification préalable générale, et bâtir sinon le modèle entier, du moins le noyau central ou certains de ses modules. Il est toujours judicieux de consacrer du temps à l'élaboration d'un plan général du modèle avant de se lancer impulsivement dans l'écriture du code. En tout état de cause, dans les programmes ou les modèles d'une certaine complexité, il est très difficile de faire un plan général détaillé et nombreuses sont les

situations qui exigent le recours aux traditionnelles méthodes d'essai et d'erreur. Le mathématicien hongrois George Pólya (1887-1985), dans son célèbre ouvrage intitulé "Comment résoudre les problèmes" [15], formule une série de recommandations utiles pour la résolution de problèmes mathématiques.

L'une de ces recommandations, que nous considérons comme très applicable à la programmation, est de commencer par traiter les versions simplifiées d'un problème complexe et difficile. Par exemple, avant de vérifier si le code qui commande à de nombreux agents de se comporter et d'interagir d'une certaine façon est correct, tester une version avec un nombre limité d'agents et utiliser uniquement les variables indispensables peut s'avérer très utile pour vérifier si les interactions se déroulent selon le mode attendu.

Exemple 26 : Différence entre les primitives «self» et «myself»

Nous présentons deux primitives de NetLogo similaires et très utiles dans la construction d'expressions: «self» et «myself». Dans le modèle suivant, nous n'utiliserons que «myself» mais nous en profiterons pour souligner la différence entre les deux. La primitive «myself» se refère à celle qui émet la commande (ask) tandis que la primitive «self» se refère à celle qui reçoit la commande.
Les exemples suivants aideront à clarifier la différence entre les deux primitives. Supposons qu'il n'existe que trois tortues (les tortues 0, 1 et 2) dans le monde et créons le petit programme suivant :

```
to setup
ca
  crt 3
  ask turtles [setxy random-pxcor random-pycor]
;; les trois tortues  occupent aléatoirement le centre d'une parcelle
end
```

Les tortues ont des questions à se poser et nous donnons ci-dessous le résultat de quelques exemples de questions et des commandes Netlogo correspondantes (appuyer sur le bouton setup de l'interface avant d'examiner chacun des cas suivants) :

- Cas 1 : La tortue 0 veut connaître la distance qui la sépare de la tortue 1

ask turtle 0 [ask turtle 1 [show distance myself]] , la tortue 0 demande à la tortue 1 «montre-moi la distance qui te sépare de moi-même, tortue 0)».

==> (turtle 1): 14.21

- Cas 2 : La tortue 0 veut connaître la distance qui sépare les tortues 1 et 2

  ask turtle 0 [ask turtle 1 [ask turtle 2 [show distance myself]]], la tortue 0 demande à la tortue 1 de demander à la tortue 2 la distance qui sépare les tortues 1 et 2 («myself» se réfère ici à la dernière demandeuse qui est dans ce cas la tortue 1). Bien entendu, on obtient le même résultat en inversant le rôle des tortues 1 et 2 dans cette commande.

  ==> (turtle 1): 21.26

- Cas 3 : La tortue 0 veut connaître la distance qui sépare la tortue 1 d'elle-même tortue 1 (question inutile en pratique mais utile pédagogiquement)

  ask turtle 0 [ask turtle 1 [show distance self]], la tortue 0 demande à la tortue 1 «montre-moi la distance qui te sépare de toi-même, tortue 1», laquelle est évidemment égale à 0.

  ==> (turtle 1): 0

Les résultats de ces trois exemples sont illustrés sur la figure 5.1

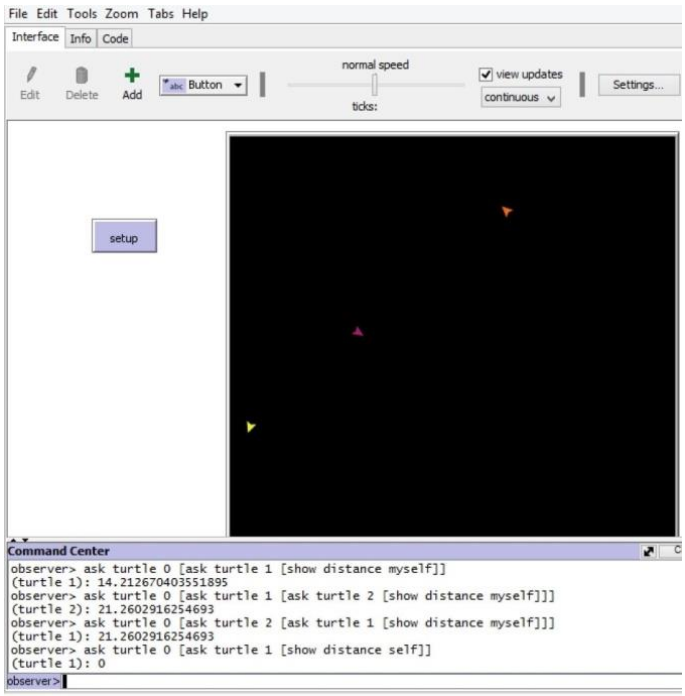


Figure 5.1: Illustration de l'utilisation des primitives «myself» et «self»

Modèle 9: Le voyageur de commerce

Primitives: face (faire face, regarder vers), [ ] (constructeur de listes), fput (mettre-devant), patch-here (parcelle-ici), with-min (avec-minimum), with-max (avec-maximum), myself (moi-même), clear-drawing (effacer-dessin), sprout-<famille> (engendrer-<famille>), in-radius (à-l'intérieur-d'un-rayon), int (nombre entier), breed (famille).
Autres détails: on crée deux familles et on donne un exemple de l'utilisation de la primitive «myself».

Le problème du voyageur de commerce («traveling salesman») a été formulé il y a plus de cent ans et constitue l'un des problèmes d'optimisation les plus étudiés. Le problème est le suivant: un voyageur de commerce qui doit visiter N villes désire calculer le plus court itinéraire qui ne passe qu'une seule fois par ville et qui se termine dans la ville où son voyage a commencé. Une solution serait de générer tous les itinéraires répondant à ces conditions, de calculer la distance parcourue pour chacun de ces itinéraires et de choisir celui dont la distance parcourue est la plus petite. Ceci est faisable pour des configurations dont le nombre de villes est très petit, puisque le nombre de routes possibles pour N villes est égal à «factoriel N», c'est à dire: 1 x 2 x 3 x 4 x 5 x 6 x .... x N, une quantité qui augmente très rapidement à mesure que la valeur de N augmente[26].

Il existe des méthodes heuristiques pour trouver une solution approximative au problème en utilisant certains types de stratégies. Dans ce modèle, nous présentons quatre types de stratégies utilisables par le voyageur de commerce (appelé ci-après le «voyageur»). Un bouton est attribué à chacune des quatre stratégies et chaque bouton représente donc un modèle différent.

Les stratégies sont les suivantes:

Proximité maximale. Le voyageur cherche à tout moment sur son l'itinéraire quelle est la ville la plus proche du point où il se trouve et se

[26] Pour donner une idée de la croissance de la factorielle d'un entier, la factorielle de 30 (notée 30 !) dépasse l'âge actuel de l'Univers exprimé en secondes. À titre d'exemple, pour 71 villes, le nombre de chemins candidats - inférieur au nombre de chemins possibles car un voyage de A vers B est identique à un voyage de B vers A - est supérieur au nombre d'atomes dans l'univers (estimé à environ $5x10^{80}$).

dirige vers cette ville. S'il existe plusieurs villes à la même distance minimale, le voyageur en choisit une au hasard.

Éloignement maximal Le voyageur cherche à tout moment sur son itinéraire quelle est la ville la plus éloignée du point où il se trouve et se dirige vers elle. De toute évidence, aucun voyageur n'utiliserait cette stratégie, à moins de vouloir se faire renvoyer par son patron. Cet itinéraire est cependant inclus pour servir de référence lors de la mesure de la distance totale parcourue.

Proximité mixte À chaque moment de son parcours, le voyageur se pose la question de savoir s'il y a des villes dans un rayon de 10 km (10 pas). Si la réponse est positive, il choisit l'une de ces villes au hasard et se dirige vers elle, sinon il choisit l'une des villes restantes au hasard. La valeur 10 peut être remplacée par n'importe quelle autre valeur dans le code.

Parcours aléatoire A chaque moment de son voyage, le voyageur choisit la ville suivante au hasard.

À titre d'exemple, la Figure 5.2 illustre le trajet d'un voyageur qui visite 71 villes dans le cadre d'une stratégie de proximité maximale. L'origine et la fin du trajet (identiques) sont représentées par le carré blanc et les segments de droite jaunes constituent le trajet optimal (selon la stratégie de proximité maximale) qui relie les 71 villes (en rouge sur la Figure 5.2).

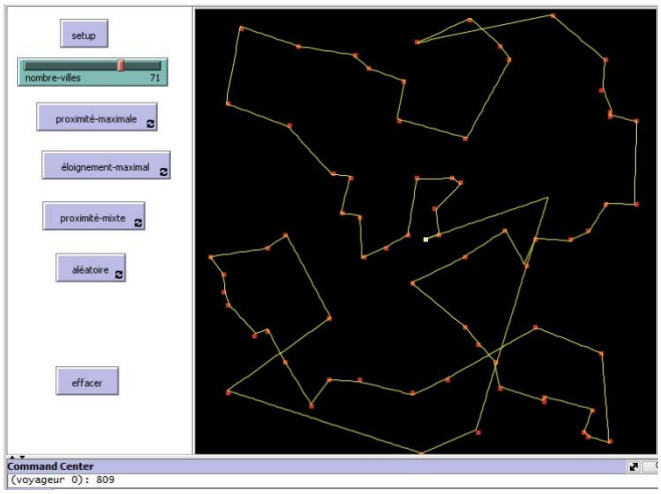


Figure 5.2 Trajet optimal selon la stratégie de proximité maximale

Comment exécuter le modèle. Le nombre de villes est défini avec le curseur «nombre-villes». Le bouton «setup» distribue de manière aléatoire les villes sur la surface du monde qui est configuré selon la topologie du carré. Le modèle peut être exécuté selon l'une des quatre modalités (stratégies) décrites ci-dessus en cliquant sur le bouton correspondant. Afin de pouvoir comparer la distance totale parcourue par le voyageur selon la stratégie choisie, mais sans changer le nombre ni l'agencement des villes, on a construit une procédure «effacer» qui possède son propre bouton et permet de supprimer le tracé de la dernière trajectoire tout en conservant la même distribution spatiale des villes (attention: après avoir cliqué sur «effacer» il faut, si l'on veut comparer les distances parcourues dans chacun des quatre cas, cliquer sur le bouton d'une autre modalité mais sans appuyer sur le bouton «setup» car cela générerait une nouvelle répartition des villes).

Activités préparatoires: Construire les boutons «setup» et «effacer». Construire un bouton différent pour chacune des quatre stratégies: proximité maximale, éloignement maximal, proximité mixte et parcours aléatoire. La case «Forever» de chacun de ces quatre boutons doit être cochée. Construire le curseur «nombre-villes». Configurer le monde selon la topologie du carré.

Plan général et problèmes à résoudre. Si l'on excepte la procédure «setup» qui est applicable à chacune des quatre stratégies, le code est séparé en quatre blocs distincts, un pour chaque stratégie choisie par le voyageur. Les codes de chacun des blocs diffèrent légèrement l'un de l'autre, les différences portant sur la partie qui définit la stratégie. Les tortues ont été utilisées pour représenter tant les villes que le voyageur. Le nombre de villes est défini à l'aide du curseur «nombre-villes» et la position de chaque ville est définie de manière aléatoire avec la commande «setxy random-xcor random-ycor», qui a déjà été utilisée dans certains exemples précédents. Afin de faire la distinction entre les tortues de type villes et celles de type voyageur, deux familles de tortues ont été créées: «villes» et «voyageurs» à l'aide des déclarations: «breed [villes ville]» et «breed [voyageurs voyageur]» La famille «voyageurs» ne comprend qu'un seul membre («create-voyageurs 1»). La ville où commence le trajet du voyageur est située sur une parcelle appelée «origine» («set origine patch-here»). Cette parcelle devient blanche lorsque le voyageur termine son trajet («ask origine [set pcolor white]»). Au cours du trajet, la prochaine ville visitée par le voyageur est la ville (tortue) appelée «destination» qui est créée avec la commande «set destination one-of villes with...». Le reste de la commande dépend de la stratégie suivie par le voyageur. Par

exemple, le reste de la commande dans le cas de la stratégie de proximité maximale est:

«set destination one-of villes with-min [distance myself]», ce qui signifie:
«Assigner à la variable «destination» l'une des villes dont la distance par rapport à moi-même est minimale."

La distance parcourue par l'agent est stockée dans la variable globale «distance parcourue». Chaque fois que le voyageur arrive dans une ville, la parcelle où se trouve la ville devient rouge et la tortue qui représente la ville est éliminée (avec la primitive «die»), mais la tortue voyageuse reste en vie. Pour savoir à quel moment le voyageur a visité toutes les villes et est retourné dans la ville d'origine, on compte le nombre de tortues-villes qui restent en vie. Dans le code du voyageur de commerce, l'utilisation de «myself» est incorporée dans la commande composée :

ask voyageur 0 [set destination one-of villes with-min [distance myself]

Nous savons que, comme un seul voyageur a été créé, son nombre «who» est égal à 0 et c'est pourquoi nous l'appelons «voyageur 0». Dans la commande précédente, «myself» (moi-même) désigne le voyageur 0, car c'est lui qui a passé la commande. Dans les blocs de code correspondant à chaque stratégie, la première chose que fait le voyageur est de vérifier s'il reste des villes à visiter et, si tel n'est pas le cas, de retourner à la ville de départ («origine»), de la peindre en blanc et d'indiquer la distance totale parcourue. La séquence de ces commandes est la suivante:

```
if count villes = 0 [ask voyageur 0 [face origine
set distance-parcourue distance-parcourue + distance origine
pd fd distance origine show int distance-parcourue
ask origine [set pcolor white]] stop]
```

Analysons les parties les plus importantes de ce bloc de commandes :

if count villes = 0 [ask voyageur 0 [face origine
(S'il ne reste plus de villes, demander au voyageur 0 de fixer son orientation en direction de l'origine).
set distance-parcourue distance-parcourue + distance origine
(ajoute la distance à l'origine à la distance parcourue).
pd fd distance origine show int distance-parcourue
(abaisser le stylet, avancer de la distance qui reste pour atteindre l'origine et afficher la distance parcourue sous forme d'un nombre entier – c'est-à-dire en éliminant les décimales-).

ask origine [set pcolor white]] stop]
(demander à l'origine [de blanchir] et arrêter la procédure).

Pour faire suite à ces observations, voici le code complet:

globals[origine destination distance-parcourue]
breed[voyageurs voyageur]
breed[villes ville]

to setup
clear-all
create-voyageurs 1 [set origine patch-here]
create-villes nombre-villes [setxy random-xcor random-ycor set color yellow]
end

to proximité-maximale
if count villes = 0 [ask voyageur 0 [face origine
set distance-parcourue distance-parcourue + distance origine pd fd distance origine
show int distance-parcourue
ask origine [set pcolor white]] stop]
ask voyageur 0 [set pcolor red]
ask voyageur 0 [set destination one-of villes with-min [distance myself] face destination set distance-parcourue distance-parcourue + distance destination pd fd distance destination]
ask destination [die]
wait 0.2
end

to éloignement-maximal
if count villes = 0 [ask voyageur 0 [face origine
set distance-parcourue distance-parcourue + distance origine pd fd distance origine
show int distance-parcourue
ask origine [set pcolor white]] stop]
ask voyageur 0 [set pcolor red]
ask voyageur 0 [set destination one-of villes with-max [distance myself] face destination set distance-parcourue distance-parcourue + distance destination pd fd distance destination]
ask destination [die]
wait 0.2
end

```
to proximité-mixte
if count villes = 0 [ask voyageur 0 [face origine
set distance-parcourue distance-parcourue + distance origine pd fd
distance origine
show int distance-parcourue ask origine [set pcolor white]] stop]
ask voyageur 0 [set pcolor red ifelse (count villes in-radius 10) > 0
[set destination one-of villes in-radius 10 face destination
set distance-parcourue distance-parcourue + distance destination pd fd
distance destination][set destination one-of villes  face destination
set distance-parcourue distance-parcourue + distance destination pd fd
distance destination] ]
ask destination [die]
wait 0.2
end

to parcours-aléatoire
if count villes = 0 [ask voyageur 0 [face origine
set distance-parcourue distance-parcourue + distance origine pd fd
distance origine
show int distance-parcourue
ask origine [set pcolor white]] stop]
ask voyageur 0 [set pcolor red ]
ask voyageur 0 [set destination one-of villes face destination
set distance-parcourue distance-parcourue + distance destination pd fd
distance destination]
ask destination [die]
wait 0.2
end

to effacer
clear-drawing
ask patches with [pcolor = red] [sprout-villes 1] set distance-parcourue 0
end
```

Explications et commentaires supplémentaires. La différence entre les procédures «éloignement maximal» et «proximité maximale» porte uniquement sur une primitive: «with-min» (proximité maximale) est remplacé par «with-max» (éloignement maximal). Dans la procédure «effacer», l'expression «sprout-villes 1» apparaît dans une commande adressée aux parcelles dont la couleur est rouge.
Notez que «villes» étant une famille, on peut utiliser ce nom au singulier ou au pluriel pour construire des commandes combinant le nom de la

famille avec d'autres primitives. Ici, on commande aux parcelles de couleur rouge de générer un agent de la famille «villes» (sprout-villes 1).

Si l'on veut vérifier que le voyageur respecte effectivement la modalité d'itinéraire indiquée, par exemple qu'il se dirige toujours vers la ville la plus proche dans le cadre de la stratégie de «proximité maximale», il suffit de décocher la case «Forever» dans le bouton correspondant de la stratégie pour pouvoir vérifier que le voyageur se déplace dans la ville suivante la plus proche chaque fois que l'on clique sur ce bouton.

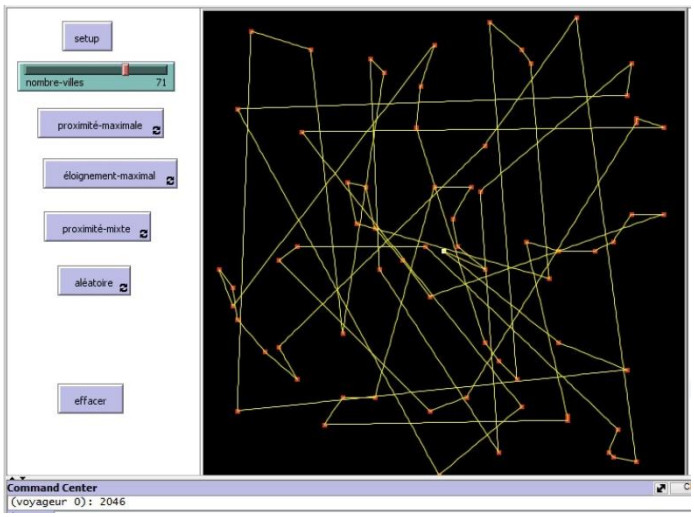


Figure 5. 3 Itinéraire selon la stratégie de proximité mixte

La Figure 5.3 illustre un itinéraire découlant de l'utilisation de la stratégie de proximité mixte. On peut constater que le voyageur a relié des villes très proches à divers points du parcours mais qu'il a également fait de grands bonds entre quelques paires de villes. Dans le cadre de cette dernière stratégie, la distance totale parcourue était de 2046 km. En conservant la même répartition spatiale des villes, les distances parcourues dans le cadre des autres stratégies étaient les suivantes: proximité maximale 809 km, éloignement maximal 5073 km et parcours aléatoire 3711 km. Une analyse exploratoire intéressante consiste à comparer les résultats en modifiant la topologie du monde. On observe une diminution des distances totales parcourues tant avec la topologie du tore qu'avec celle du cylindre bien que cette diminution soit plus prononcée dans le cas du tore que dans celui du cylindre (question pour le lecteur : expliquez pourquoi).

Exercice 5.1. Modifiez le code pour que les villes soient représentées par des parcelles et non par des tortues.

Association entre agents

L'association d'agents dans des groupes ayant certaines affinités ou effectuant des tâches spécifiques est courante dans de nombreux modèles. La réalité nous fournit d'innombrables exemples d'agents regroupés selon certaines règles. Les humains forment des groupes en fonction de critères tels que le sexe, l'affinité politique, la religion, le sport, la nationalité, la profession, les loisirs, etc. Les particules élémentaires se rassemblent pour former des atomes et ces derniers sont associés dans des groupes appelés molécules, les lettres de l'alphabet sont regroupées pour former des mots et les mots pour former des phrases.

Dans tous ces cas, la formation des groupes se fait suivant des critères ou règles que l'on pourrait appeler *règles d'association*, qui permettent ou excluent la possibilité de l'association de deux agents ou plus. Dans le monde réel, l'une des plus grandes difficultés auxquelles font face les scientifiques est de découvrir les règles d'association qui régissent le comportement des agents impliqués dans un phénomène (pourquoi certains types de cellules se sont-elles associées pour former un organe ? quand ont-elles cessé de le faire? Pourquoi tant de personnes ont-elles voté pour la présidence du candidat Kisétou Bonarien?). Dans l'exemple suivant, les tortues sont associées à des groupes appelés clubs. Le mécanisme d'association repose sur un échange d'informations entre les tortues et les parcelles.

Modèle 10 : La foire du livre

Primitives : length (longueur), count (compter), turtles-here (tortues-ici), die (mourir), lput (mettre-en-dernier), with (avec), sum (somme), of (de)

Figure 5.4 : Foire du livre

L’histoire Dans un parc urbain, se tient une grande foire du livre à laquelle participent de nombreuses maisons d'édition du monde entier. Les éditeurs sont regroupés sous les lettres N, E, L et W, correspondant à N = Amérique du Nord (États-Unis et Canada), L = Amérique latine, E = Europe et W = reste du monde. Dans le parc, de nombreux kiosques de toile ou «stands» sont installés, répartis sur une partie de la surface du parc où les livres sont vendus. Les kiosques sont de deux types: dans les kiosques de type NL, les livres sont vendus par les éditeurs N et L, dans ceux de type EW, les livres des éditeurs E et W sont vendus. À l'entrée de la foire, une machine distributrice remet une carte à chaque visiteur. Sur cette carte est imprimée l’une des quatre lettres N, E, L ou W générées aléatoirement par la machine. La carte permet à sa détentrice d’acheter, moyennant des rabais importants et ce tout au long de l’année, les livres des éditeurs qui correspondent à la lettre imprimée sur sa couverture. Mais pour obtenir ces rabais, la visiteuse doit d’abord se joindre au club de lecture de l’un des kiosques. S’associer au club de lecture d’un kiosque est très simple: en gros, il suffit d’arriver tôt au salon pour pouvoir trouver de la place dans le club de n’importe quel kiosque, car le nombre de places de chaque club est limité (un bon truc de marketing pour attirer la clientèle). Si, par exemple, vous avez une carte de type E, vous devez vous rendre à un kiosque de type EW. S'il reste encore des places à ce kiosque, vous êtes, sur présentation de votre carte, automatiquement inscrit au club de lecture du kiosque, sous la lettre E, ce qui vous permettra de bénéficier des réductions d'éditeurs du groupe E tout au long de l'année. Chaque kiosque enregistre les types de cartes de membre de chacun des clubs de lecture. Il est ainsi possible, de connaître à tout moment, la configuration de la distribution desdits clubs.

Plan général et problèmes à résoudre. Les problèmes suivants doivent être résolus:

1. Disséminer les kiosques dans le parc (le monde). Nous allons résoudre ce problème en assignant à chaque parcelle l'une des trois chaînes suivantes: "NL", "EW", "O". La chaîne "O" représentera les parcelles où il n'y a pas de kiosque. L’assignation se fera de manière aléatoire au moyen de l’instruction suivante:

ask patches [set lettre-du-kiosque one-of [“NL” “EW” “O” “O” “O” “O” “O” “O” “O” “O” “O”]

où "O" est inclus un plus grand nombre de fois, de sorte que la majorité des parcelles n'hébergent aucun kiosque. Les kiosques de type "NL" sont représentés en bleu et ceux de type "EW" en orange.

2. Concevoir un mécanisme permettant aux visiteurs de s'associer aux clubs de lecture.
Pour modéliser ce mécanisme, nous affectons des variables aux parcelles et aux tortues (les visiteurs) et effectuons des comparaisons entre ces variables. Les parcelles ont deux variables de type «patches-own»: la variable «kiosque» stocke certaines des chaînes "NL", "EW" ou "O" et la variable «club» stocke une liste avec les types de cartes de membre des visiteurs qui se sont inscrits à leur club de lecture. Par exemple, ["E" "W" "W" "E"] indique que dans le club du kiosque, deux personnes avec une carte de type E et deux avec une carte de type W se sont associées. Les visiteurs n'ont qu'une variable de type «turtles-own», qui garde en mémoire le type de carte que la machine distributrice leur a attribué: N, L, E ou W. Un curseur «num-max-membres» détermine le nombre maximum de membres que les clubs peuvent accepter et un curseur «population-initiale» détermine le nombre initial de visiteurs de la foire.

Le mécanisme d'association aux clubs. Les personnes (tortues) déambulent aléatoirement dans le parc. Quand une personne (tortue) arrive sur une parcelle, elle compare la valeur de la variable «kiosque» de la parcelle à la valeur de sa variable «carte». S'il existe une correspondance entre la valeur «carte» et l'une des deux lettres de la variable «kiosque», et si, de plus, une place est disponible (c'est-à-dire si «length club» ne dépasse pas «nombre-max-membres»), la personne est alors automatiquement inscrite au club de lecture du kiosque. Ceci se fait en ajoutant la lettre de la carte de la personne à la liste «club» de la parcelle. Par exemple, supposons que la carte d'Emilie soit "L", et qu'elle visite un kiosque (une parcelle) de type "NL" dont le nombre maximum de membres autorisés est de 6. Supposons de plus que la variable «club» du kiosque ait la valeur ["L" "L" "N"]. Cela signifie que «length club = 3» et qu'il reste encore trois places dans le club du kiosque. Ainsi, Emilie sera admise comme membre du club de ce kiosque et la variable «club» de la parcelle prendra la valeur ["L" "L""N" "L"].

Si une personne ne parvient pas à s'inscrire dans le club d'un kiosque, soit parce qu'il n'y a pas de places disponibles, soit parce que le type de carte de la personne ne correspond à aucune des lettres du kiosque, la personne doit alors poursuivre sa marche. Pour éviter que *les personnes ne s'inscrivent dans plus d'un club, elles sont éliminés au moment de l'inscription* (elles rentrent chez elles, heureuses d'avoir pu profiter des rabais). La procédure s'interrompt automatiquement quand plus personne (tortue) ne peut s'inscrire à un club. On peut aussi arrêter la simulation à tout moment en appuyant sur le bouton «go». Une fois le modèle arrêté, il est possible de poser des questions sur la répartition des personnes dans les clubs à partir de la fenêtre de l'observateur.

Activités préparatoires. Construire les boutons «setup» et «go» (cocher la case «Forever» de ce dernier). Construire également les curseurs «population-initiale» et «nombre-max-membres».

Voici le code:

```
patches-own[club kiosque]
turtles-own[carte]

to setup
  clear-all
  crt population-initiale [setxy random-xcor random-ycor
  set color green set carte one-of ["N" "E" "L" "W"]]
     ask patches [set kiosque one-of
     ["NL" "EW" "O" "O" "O" "O" "O" "O" "O" "O" "O"]]
  ask patches [set club [] if kiosque = "O"
     [set pcolor white]]
  reset-ticks
end

to go
 ask turtles [ fd 2 if (length club < nombre-max-membres) and kiosque =
"NL"
   and (carte = "N" or carte = "L") [ set club lput carte club
           die] ]
ask turtles [ fd 2 if (length club < nombre-max-membres) and kiosque =
"EW"
   and (carte = "E" or carte = "W") [ set club lput carte club
          die] ]
ask patches  [if kiosque = "NL"   [set pcolor blue]
    if kiosque = "EW" [set pcolor orange]]
   if count turtles = 0 [stop]
   wait 0.1
      tick
end

to montrer
ask patches with [length club > 0 ][show club]
end
```

Explications et commentaires supplémentaires. La procédure s'interrompt s'il n'y a plus de tortues qui cherchent à s'associer à des clubs. Cependant, il peut arriver que certaines tortues ne réussissent pas à trouver de club. Cela est dû au fait que, les tortues ayant une orientation initiale fixe, il est possible que certaines tortues marchent sans arrêt sans trouver un club auquel s'associer. Dans ce cas, vous devez arrêter la procédure manuellement en appuyant sur le bouton «go». La procédure «montrer» est facultative et doit être appelée à partir de la fenêtre de l'observateur. Cette procédure indique les parcelles où se trouvent des clubs ainsi que leur composition. Les parcelles ont été colorées selon des critères précis. Ainsi, initialement, lorsque la procédure d'initialisation («setup») est exécutée, les parcelles noires sont celles où il y a des kiosques de vente et les blanches celles où il n'y en a pas, tandis que toutes les tortues apparaissent en vert. Lorsque la procédure «go» est exécutée, les parcelles avec les kiosques de type NL deviennent bleues et celles avec les kiosques de type EW deviennent orange, comme le montre la figure 5.5.

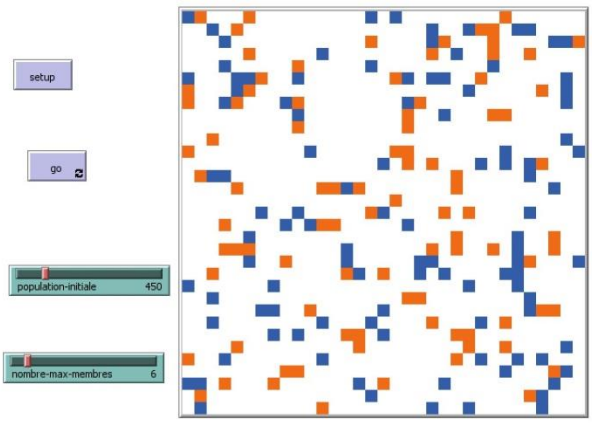


Figure 5.5 : État du monde à la fin de la simulation

Il est possible d'obtenir plus d'informations sur la répartition des tortues dans les clubs en faisant des requêtes dans la fenêtre observateur après l'exécution du modèle. Pour une simulation avec une population initiale de 1000 tortues et dont la valeur de la variable «nombre-max-membres» est fixée à 6, on obtient les résultats suivants après 2000 ticks (la simulation a été arrêtée manuellement car, à ce stade, un certain nombre de tortues n'avaient toujours pas trouvé de club auquel s'associer):

print count patches with [length club = 6]
==> 133, 133 kiosques ont réussi à remplir leurs clubs avec 6 membres
print count patches with [nombre-max-membres > 3]
==> 1089
print count patches with [club = ["N" "N" "L"]]

==> 5, notez que, comme il s'agit de listes, un club dont le statut aurait été ["N" "L" "N"] n'aurait pas été compté parmi ces 5 clubs.
print count patches with [nombre-max-membres > 2 and nombre-max-membres < 6]
==>159
print count patches with [length club = 4]
==>3

Choses à explorer. En faisant varier la valeur de la population présente à la foire (curseur «population-initiale») et le nombre maximum de membres admis dans les clubs (curseur «nombre-max-membres»), il est possible de tester la sensibilité du modèle aux changements des valeurs de ces deux variables. Ainsi, par exemple, si le nombre de participants est suffisamment élevé, tous les clubs seront pleins et une partie de la population ne pourra pas bénéficier de réductions de prix faute d'avoir pu trouver des clubs avec des places disponibles. Si le nombre de participants est faible, il y aura un mélange de kiosques avec des clubs pleins, des clubs à moitié pleins et même des clubs vides. Le modèle peut être adapté à plusieurs autres situations dans une population d'agents cherchant à s'associer pour former des groupes. Ce pourrait, par exemple, être le cas de personnes avec certains profils professionnels qui recherchent un emploi dans certains types d'institutions ou de sociétés, ou encore d' atomes qui se joignent à des molécules pour former de plus grosses molécules (un modèle de ce type sera présenté ultérieurement).

Exercice 5.2. Pour faciliter les consultations sur le degré d'intégration des clubs, ordonnez ses membres par ordre alphabétique. (Suggestion: utilisez la primitive
«sort-by»).

Modèle 11 : À la recherche d'un taxi en ville

Primitives: ifelse, count, with-min (avec-minimum), myself (moi-même), one-of, create-link-with (créer-lien-avec), tie (attacher), stop, distance.
Autres détails: emploi de la primitive «breed» (famille).

L'histoire. Au cours d'une journée de travail, dans une grande ville, de nombreuses personnes recherchent un taxi pour se rendre à leur travail ou en revenir, pour faire du tourisme ou pour d'autres activités. Les gens et les taxis sont dispersés dans toute la ville. Dans le modèle, nous supposerons que lorsqu'un client appelle un taxi à l'aide de son téléphone portable, un

système GPS envoie le taxi disponible le plus proche du client. Les taxis ne sont pas autorisés à prendre en charge un passager lorsqu'ils sont en service mais, après avoir terminé une course, ils peuvent répondre aux demandes d'autres clients. Le nombre de clients et de taxis est fixé à l'aide de curseurs dans l'interface. L'objectif du modèle est d'observer comment les services de taxi se répartissent entre les clients.

Plan général et problèmes à résoudre. Les personnes et les taxis sont représentés par des tortues appartenant à deux familles: les «taxis» et les «personnes» dont les membres sont répartis de manière aléatoire dans l'espace. Les destinations finales des clients sont des parcelles qui sont également choisies de façon aléatoire.

Chaque personne a deux variables qui lui sont propres («turtles-own»):

1) La variable «proches» a pour valeur l'ensemble-agents des taxis dont la distance par rapport à la personne qui appelle est la plus proche (minimum).

2) La variable «destination» a pour valeur la parcelle de destination sur laquelle le client doit arriver en taxi.

Les taxis, à leur tour, ont deux variables propres («taxis-own»):

1) La variable «état», qui peut prendre les valeurs 0 ou 1 selon que le taxi est disponible ou est en service.

2) La variable «courses», qui compte le nombre de courses réalisées par le taxi.

Chaque course de taxi est représentée par une ligne brisée avec une première section droite en jaune et une deuxième section droite en rouge. La section jaune représente l'itinéraire entre l'endroit où se trouve le taxi et celui où se trouve le client qui a passé l'appel téléphonique. La section rouge indique l'itinéraire entre le lieu de ramassage du passager et sa destination finale. Pour illustrer le fait que le taxi et le passager voyagent ensemble, un lien est créé entre eux au moment où le taxi arrive sur les lieux de ramassage du passager. Lorsque les deux arrivent à destination, la tortue qui représente le passager meurt. Cela empêche le passager de continuer à demander des services de taxi et d'aller continuellement d'un endroit à un autre. Une fois le modèle exécuté, il est intéressant de recueillir quelques informations statistiques.

Figure 5.6 : Exemple avec deux taxis et trois client(e)s

La figure 5.6 montre les services fournis par 2 taxis à 3 personnes. Les services commencent par la section jaune. La ligne brisée supérieure montre qu'un des taxis n'a effectué qu'une seule course (trajectoire jaune-rouge). La ligne brisée inférieure montre une trajectoire composée de 4 tronçons séquentiels jaune-rouge-jaune-rouge et nous indique que le taxi a effectué deux courses. La trajectoire commence par une section jaune (le taxi récupère le premier client) suivie d'une section rouge (le client est conduit à destination), elle-même suivie d'une deuxième section jaune (le taxi vient chercher le deuxième client) pour se terminer par un tronçon rouge (la deuxième cliente est arrivée à destination).

Une fois que l'on a créé les ensemble-agents de personnes et de taxis et défini les variables dans la procédure «setup», les actions les plus importantes sont appelées par la procédure «appeler-taxi» qui est elle-même appelée par la procédure «go»:

1) La personne appelle un taxi: *ask personnes [appeller-taxi]*
La procédure «appeller-taxi» déclenche les actions suivantes :

a. un taxi est sélectionné parmi l'ensemble-agents «proches»
set proches taxis with-min [distance myself]
b. on vérifie qu'il existe des taxis inoccupés et qu'il existe au moins un taxi proche disponible:
ifelse (count taxis with [état = 0]) > 0 [if count proches > 0…
c. On demande au taxi choisi de se colorer en jaune, de se tourner vers le passager, de se diriger vers lui et de définir son «état» comme «occupé» (état = 1) et d'ajouter 1 au nombre de courses effectuées :
[ask one-of proches [set color yellow face myself
pd fd distance myself set état 1 set courses courses + 1….

2) Ensuite, on demande au taxi qu'il établisse un lien avec la personne qui l'a appelé, qu'il attache ce lien entre lui-même et ladite personne, qu'il colore le taxi en rouge et, finalement, qu'il conduise le passager à destination:
….create-link-with myself [tie] set color red]]] aller-à-destination

La procedure aller-à-destination ne demande pas d'explications supplémentaires.

Activités préparatoires: Construire les boutons «setup» et «go» (cocher la case «Forever» de ce dernier) ainsi que les curseurs «nombre-personnes» et «nombre-taxis». Créer des familles de personnes et de taxis. Configurer le monde selon la topologie du carré.

Voici le code complet:

```
breed[taxis taxi]
breed[personnes personne]
personnes-own[destination proches]
taxis-own[ état courses]

to setup
clear-all
create-personnes nombre-personnes [setxy random-xcor random-ycor set color red set destination one-of patches ]
create-taxis nombre-taxis [setxy random-xcor random-ycor set color yellow]
end

to go
ask personnes [appeler-taxi]
end

to appeler-taxi
set proches taxis with-min [distance myself]
if (count taxis with [état = 0]) > 0 [if count proches > 0 [ask one-of proches
  [set color yellow face myself pd fd distance myself set état 1
    set courses courses + 1
   create-link-with myself [tie] set color red  ]]]
aller-à-destination
```

end

to aller-à-destination
wait 0.2 face destination pd fd distance destination die
end

Explications et commentaires supplémentaires.
On veut que lorsque la personne appelle un taxi, elle se voit attribuer celui qui se trouve le plus proche d'elle. S'il y a plus de deux taxis qui répondent à ce critère, l'interpreteur sélectionne l'un d'entre eux au hasard. Si l'on supprime la commande «die» qui élimine les personnes lorsqu'elles atteignent leur destination, on risque d'engendrer des phénomènes non conformes au script établi. Ainsi, par exemple, on pourrait observer des trajectoires avec des lignes rouges suivies d'autres lignes rouges, un phénomène qui pourrait signifier que certaines personnes parcourent de longues distances à pied vers d'autres destinations, alors qu'il y a encore des taxis disponibles.

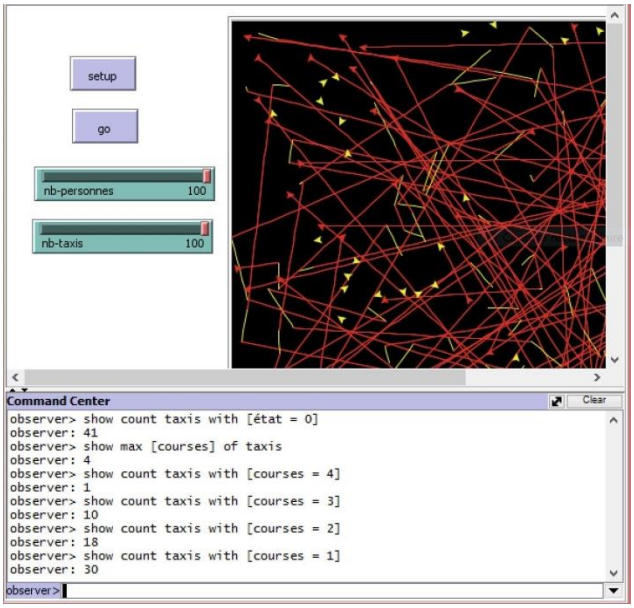


Figure 5.7 : Exemple de traces de trajectoires

La figure 5.7 illustre les courses effectuées par 100 taxis pour servir 100 personnes. Notez que certains taxis n'ont fait aucune course.

Voici quelques informations utiles pour analyser la simulation illustrée par la figure précédente.

Nombre de taxis n'ayant effectué aucun service:
show count taxis with [état = 0]

==>observer: 41
Nombre maximum de courses effectuées par un taxi
show max [courses] of taxis
==> observer: 4
Nombre de taxis qui ont effectué 4 courses :
show count taxis with [courses = 4]
==> observer: 1
Autres informations: 10 taxis ont effectué 3 courses, 18 en ont effectué et 30 en ont effectué 1.

Exercice 5.3: Adaptez le modèle de manière à ce que de nouvelles personnes qui ont besoin d'un service de taxi fassent périodiquement leur apparition au cours de la simulation.

Exercice 5.4: Modifiez le code de sorte que les taxis qui effectuent plus d'une course laissent un trait vert.

Modèle 12: À la recherche d'une cargaison de drogue dans la jungle

Primitives: remove (enlever), run (éxecuter), first (premier), length (longueur), but-first (sauf-le-premier), write (écrire), word (nom).
Autres détails: comment lire un à un les caractères d'une chaîne ou les membres d'une liste, comment utiliser la primitive "run".

L'histoire. L'agente de police Evans est une spécialiste des missions de pénétration dans la forêt tropicale. On l'envoie, avec deux chiens, à la recherche d'une cargaison de drogue enterrée dans un endroit de la jungle non protégé par les trafiquants de drogue. On demande à Evans de consigner dans un registre tous les mouvements de son parcours dans la jungle et de les envoyer par radio au poste de police central afin de permettre à un autre agent de la rencontrer une fois la cargaison trouvée. La marche de l'agente Evans (tortue 0) est basée sur quatre types de mouvements: avance d'un pas en avant [fd 1], tourne 90 degrés à droite [rt 90], tourne 90 degrés à gauche[lt 90] et tourne 180 degrés à droite [rt 180] (le virage à 180 degrés à gauche n'est pas inclus car son effet est le même que celui du virage à droite). Une fois qu'Evans aura trouvé l'endroit où est cachée la cargaison, l'agent Garcia, un membre du redoutable Commando Bleu, sera dépêché sur les lieux pour assister Evans et coordonner la saisie de la cargaison. García base sa trajectoire sur le registre envoyé par Evans, mais avant de partir à sa rencontre, le registre doit être débogué. Traverser la dense végétation d'une jungle où se dressent de nombreux autres obstacles et dangers n'est pas chose facile et Evans doit, à plusieurs reprises, avancer en procédant par

tâtonnements. C'est la raison pour laquelle le registre des mouvements envoyé par Evans contient des séquences de mouvements superflus, comme, par exemple, un virage à 90º vers la droite suivi d'un virage à 90º vers la gauche, mouvements superflus dont nous discuterons plus tard.

Comment exécuter le modèle**.**
1. Appuyez sur le bouton «setup», puis sur le bouton «go».
2. Arrêtez le modèle en appuyant à nouveau sur le bouton «go» lorsque l'agent Evans est censé avoir trouvé la cargaison (c'est l'utilisateur qui décide du moment de la découverte de la cargaison et qui met donc fin à la poursuite du trajet).
3. Cliquez sur le bouton «déboguer», dont l'effet est d'éliminer les séquences superflues du registre envoyé par l'agente Evans.
4. Appuyez sur le bouton «commando-bleu» pour que l'agent Garcia puisse se rendre dans la jungle et rencontrer l'agente Evans en suivant le chemin débogué.
5. Dans le terminal d'instruction on peut lire le nombre de mouvements inscrits dans le registre de l'agente Evans ainsi que celui de la trajectoire «déboguée» de l'agent Garcia.

Plan général et problèmes à résoudre. Le modèle présente deux problèmes principaux qui doivent être résolus:
1. Comment sauvegarder les informations de la trajectoire de l'agente Evans dans un message ?
2. Comment déboguer le message envoyé par l'agente Evans, en éliminant les séquences de mouvements superflus ?

Le premier point est résolu en enregistrant tous les mouvements de l'agent dans une variable appelée «registre». Ce registre pourrait être une liste, telle que: *set registre ["fd 1" "rt 90" "rt 90 "" fd 1 "" fd 1 "]*.... ou encore une chaîne, par exemple set registre *" fd 1 rt 90 rt 90 fd 1 fd 1 lt 90 "*. Pour éliminer les séquences superflues contenues dans le message envoyé par Evans, il faut commencer par construire une liste de ce type de séquences. Les séquences suivantes de mouvements effectuées par l'agente Evans sont considérées comme superflues:

Virages de même amplitude et en directions opposées à partir d'un même emplacement (ce qui ne change pas l'orientation de la tortue):
1. rt 90 lt 90
2. lt 90 rt 90
3. rt 180 rt 180

Avance d'un pas et retourne au même point en conservant la même orientation initiale :
4. fd 1 rt 180 fd 1 rt 180.
5. fd 1 rt 180 fd 1 rt 90 rt 90.
6. fd 1 rt 180 fd 1 lt 90 lt 90.
7. fd 1 rt 90 rt 90 fd 1 rt 180.
8. fd 1 lt 90 lt 90 fd 1 rt 180.
9. fd 1 rt 90 rt 90 fd 1 rt 90 rt 90.
10. fd 1 rt 90 rt 90 fd 1 lt 90 lt 90.
11. fd 1 lt 90 lt 90 fd 1 rt 90 rt 90.
12. fd 1 lt 90 lt 90 fd 1 lt 90 lt 90.

Ces 12 séquences de commandes laissent la tortue dans la même position et avec la même orientation qu'avant l'exécution de la séquence, de sorte qu'elles peuvent être éliminées. Il y a d'autres séquences superflues qui n'ont pas été incluses car leur présence est moins probable, comme une séquence de quatre tours de 90 degrés dans la même direction [lt 90 lt 90 lt 90 lt 90], ce qui équivaut à un tour à 360 degrés ou des séquences dans lesquelles la tortue avance de plusieurs pas et recule d'une même distance [fd 5 bk 5] pour se retrouver au point de départ et dans la même orientation initiale. En principe, le nombre de séquences superflues possibles est infini, mais plus les séquences sont improbables, plus elles sont longues. Dans ce modèle, nous ne considérerons que les 12 séquences superflues mentionnées ci-dessus. Si nous décidons de représenter le message envoyé par l'agent Evans au moyen d'une liste, le débogage consistera à éliminer de la liste les sous-listes qui correspondent à des séquences superflues. Mais il se trouve que NetLogo ne dispose pas de primitives capables de trouver («search») une sous-liste d'une liste donnée, sans devoir indiquer également l'emplacement où la sous-liste commence et où elle se termine. Mais il y a un problème: on ne sait pas dans quelle position de la liste on trouvera les sous-listes qui correspondent à des séquences superflues. Pour éviter de devoir créer des procédures qui remplissent cette fonction de recherche, il est plus facile de représenter la variable «registre» au moyen d'une chaîne, car il existe une primitive NetLogo appelée «remove» qui est disponible pour les chaînes. Cette primitive permet de supprimer une sous-chaîne sans avoir besoin d'indiquer la position que ladite sous-chaine occupe dans la chaîne qui la contient. Pour rendre le message débogué encore plus compact, nous représenterons les quatre mouvements que les agents sont autorisés à exécuter par des lettres, conformément à la convention suivante : les mouvements fd 1, rt 90, lt 90 et rt 180 seront représentés, respectivement, par les lettres f, r, l et h, où à chaque lettre est affectée une procédure avec les actions correspondantes. Ainsi, par

exemple, une chaîne telle que "ffrfl" représente la séquence de mouvements fd 1, fd 1, rt 90, fd 1 lt 90. Pour terminer, l'interprèteur transforme la chaîne en une succession de mouvements de tortues attribuant une procédure à chaque lettre. Par exemple, la lettre f est associée à la procédure:

```
to f
fd 1
end
```

Activités préparatoires Construire les boutons pour les procédures «setup», «go» (cochez la case «Forever»), «déboguer» et «commando-bleu». La topologie du monde est celle du carré.

Le code est le suivant:

```
globals [mouvements registre]

to setup
  clear-all
       crt 1 [set registre "" ;; ici on définit "registre"
  ;; identique à la chaîne vide""
  set heading 0 set color yellow setxy -8 -8 pd]
       ask turtle 0 [set mouvements ["f" "f" "f" "r" "l" "h"]]
    crt 1 [setxy -8 -8 set heading 0 pd set color blue]
       reset-ticks
end

to go
     ask turtle 0 [let pas one-of mouvements
    run pas
  wait 0.1  set registre word registre pas] ;; dans cette commande on
 ;; ajoute «pas» à la chaîne «registre»
tick
end

to f
  fd 1
end


to r
  rt 90
```

```
end

to l
  lt 90
end

to h
  rt 180
end

to déboguer
 type "mouvements de l'agente Evans: "   type " " write length registre
    set registre remove "frrfrr" registre
  set registre remove "fllfll" registre
  set registre remove "fllfrr" registre
  set registre remove "frrfll" registre
  set registre remove "frrfh" registre
  set registre remove "fllfh" registre
  set registre remove "fhfh" registre
  set registre remove "fhfll" registre
  set registre remove "fhfrr" registre
  set registre remove "rl" registre
  set registre remove "lr" registre
  set registre remove "hh" registre
  type "  " type "mouvements de l'agente García: " write length registre
  print ""
end

to commando-bleu
ask turtle 1 [  run first registre set registre but-first registre]
wait 0.1 if length registre = 0 [ask turtle 1 [set pcolor white]
   beep stop]
end
```

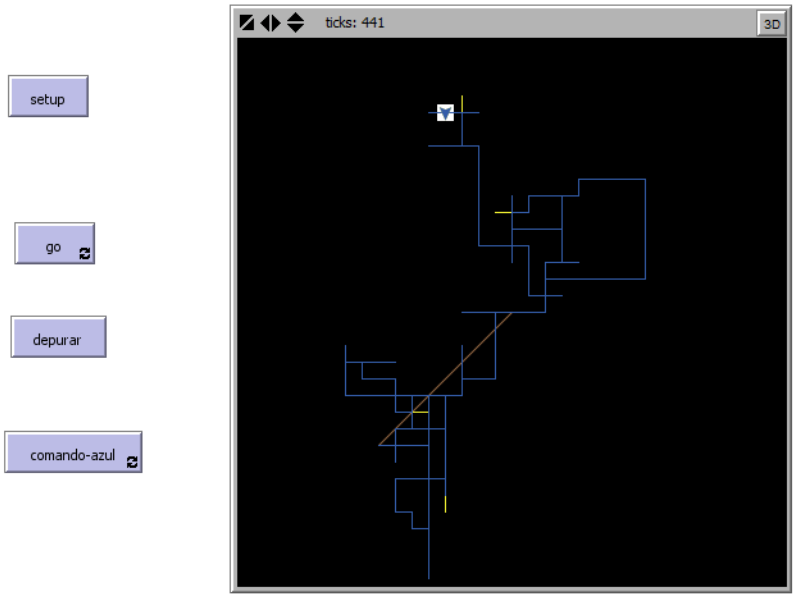


Figure 5.8 : Exemple de parcours des deux agents

La figure 5.8 illustre le parcours de l'agent Garcia, qui consiste en 334 étapes.
Le parcours initial de l'agente Evans consistait en 441 étapes. Le processus de débogage a réduit le parcours de 24%, éliminant les séquences superflues.

Explications et commentaires supplémentaires:
La commande *set registre* "" attribue à la variable «registre» la valeur de la chaîne vide "". La variable mouvements est définie comme étant la liste des mouvements possibles que les agents peuvent effectuer: ["f" "f" "r" "l" "h"] où chaque lettre est le nom d'une procédure qui correspond à un mouvement.
La répétition de "f" est intentionnelle pour donner plus de chances aux tortues d'avancer que de changer d'orientation. La variable «pas» est une variable temporaire, définie par la primitive «let», et à qui l'on attribue la valeur de l'une des procédures de la liste «mouvements»: «let pas one one-of mouvements». Pour exécuter la procédure hébergée dans la variable «pas», il est nécessaire d'utiliser la primitive «run». La commande «run pas» signifie littéralement «exécute la procédure contenue dans la variable pas. La primitive «word» permet de former une chaîne unique à partir de deux autres chaines. Ainsi l'expression «word registre pas» ajoute à la chaîne registre le contenu de la variable «pas» qui correspond au dernier mouvement effectué par l'agent. Au début et à la fin de la procédure «déboguer» on a inclus des commandes qui rapportent, dans le terminal d'instructions, la longueur de la chaîne «registre» («length registre») avant et après le débogage, afin de savoir combien de pas l'argent Garcia a

économisé pour rejoindre le lieu de découverte de la cargaison suite au débogage du registre par l'agente Evans.

Comme nous avons déjà eu l'occasion de le mentionner, la programmation de modèles est un véritable terreau pour la génération d'idées et l'établissement de liens avec d'autres domaines, ce qui facilite:

1) La formulation des questions liées au fonctionnement du modèle.
2) La découverte de liens entre le modèle et d'autres domaines de la connaissance.
3) La génération d'idées pour étendre le modèle.
4) Les explications sur les comportements non prévus dans le modèle.
5) Le débogage des erreurs de programmation et de planification (erreurs de conception et de syntaxe).

Utilisons le modèle précédent à titre d'exemple. Ce que l'on peut observer lors de l'exécution du modèle actuel, c'est que l'agent Garcia (tortue 1) laisse de petites traces du chemin - des espèces de pointes - en jaune, sans repeindre. Après avoir analysé la situation, nous pouvons constater qu'il s'agit de pointes d'une longueur d'un pas et qui ne sont que des petits morceaux superflus des types 4 à 12 de la liste décrite ci-dessus. Lorsqu'on cherche une explication à l'apparition de ces petites pointes jaunes, une autre question se pose, peut-être plus importante. Dans le processus de débogage, supposons que la chaîne "registre" contienne une sous-chaîne semblable à la chaine "frrfrrlf". Notez que cette sous-chaîne contient deux séquences superposées qui se chevauchent, qui sont "frrfrr" (séquence de type 9) et "rl" (séquence de type 1). Si l'interprèteur décide d'éliminer d'abord la séquence superflue "rl", on obtient "frrfrf" mais si la séquence superflue frrfrr est éliminée en premier, le résultat serait "lf". La question qui se pose maintenant est la suivante: l'élimination de «frrfrf» en premier produit-elle le même résultat que l'élimination de «lf» en premier? Une brève analyse de la situation révèle que ça n'est pas le cas. Bien que l'élimination de "frrfrr" ne modifie pas le chemin suivi par l'agent Garcia, l'élimination de "lr" entraine une telle modification. Cela signifie que, selon l'ordre dans lequel l'interprèteur supprime les séquences superflues, l'agent Garcia pourrait être en danger de s'écarter du chemin emprunté par Evans et de se perdre dans la jungle. Cependant, si on force la procédure de "débogage" à éliminer les longues séquences superflues des types 4 à 12, ce phénomène ne peut se produire. C'est la raison pour laquelle, dans le code de la procédure "debug", les séquences des types 4 à 12 sont les premières à apparaître.

Exercice 5.5. Expliquez la présence des pointes jaunes dans la trajectoire de l'agent Garcia.

Promenade II

On pourrait dire que dans le monde actuel de la programmation, il existe deux types de programmeurs. D'une part, il y a des programmeurs professionnels qui sont personnes ayant une solide expérience dans le domaine de l'informatique. La majorité des programmeurs professionnels ont été formés dans des universités ou des écoles d'informatique en tant qu'ingénieurs, scientifiques ou techniciens dans l'un des domaines de l'informatique et du génie logiciel. Au cours de leur formation, ils ont acquis la maîtrise d'un ou plusieurs grands langages de programmation à vocation générale tels que Pascal, C, C ++, Lisp ou Java, pour ne citer que quelques-uns. Outre leur connaissance des langages, ces diplômés en informatique ou en génie logiciel ont dû approfondir certains aspects de la technologie informatique (bases de données, compilateurs, systèmes d'exploitation, etc.). L'autre classe de programmeurs est constituée de ce que nous pourrions appeler des «programmeurs utilisateurs». Ce groupe de programmeurs, auxquels appartient l'auteur de ce livre, est composé de personnes qui utilisent un langage de programmation à usage spécifique. Nous apprenons à programmer dans ces langages pour développer des projets dans des domaines spécifiques tels que la gestion des affaires, les mathématiques, la statistique, la finance, l'animation ou la conception de pages Web, et bien d'autres encore. Des exemples de ces langages sont Logo, NegLogo, Mathematica, R, Excel, Blender, Scratch ou HTML, entre autres. Parmi les programmeurs utilisateurs, il semble pertinent d'inclure également les personnes qui apprennent à programmer dans l'un des langages appelés «langages de script». Ces langages permettent d'insérer du code dans différents types d'applications pour effectuer des tâches spécifiques. Javascript est un exemple de langage de script. Son code peut être inséré dans des pages Web pour être interprété par les navigateurs Internet. Ce livre s'adresse spécifiquement aux programmeurs utilisateurs intéressés par la programmation à base d'agents utilisant le langage NetLogo.

Il ne s'adresse pas aux programmeurs professionnels et c'est la raison pour laquelle des explications sur les aspects les plus fondamentaux de la programmation ont été incluses dans le livre. Les lectrices et lecteurs qui souhaitent approfondir le monde passionnant de la programmation ou d'autres sujets en informatique trouveront de nombreuses ressources sur le Web. Nous recommandons fortement, en particulier, l'excellent livre d'Abelson et Sussman [1], disponible gratuitement en anglais sur le Web. Ce livre est une source d'information très précieuse dans laquelle sont exposés les principes fondamentaux de la science et de l'ingénierie informatique. Il est à noter qu'il

existe une version française de cet ouvrage, version qui a été publiée par la maison d'édition InterEditions[27].

Modèle 13: La soupe primordiale de la vie

Primitives: with (avec), and (et), [ ] (constructeur de listes), first (premier), lput (mettre à la fin), sort-by (ordonner-selon) != (différent de).
Autres détails: le modèle montre comment utiliser et manipuler des listes dont les membres sont à leur tour des listes. Il montre également comment simplifier les expressions complexes à l'aide de procédures de type «rapporteur».

L'histoire. L'une des théories sur l'origine de la vie sur notre planète avance l'hypothèse selon laquelle, il y a plusieurs millions d'années de cela, alors que la Terre était dans un état très primitif, sa surface était recouverte d'un bouillon ou d'une soupe en grande partie constituée d'atomes qui forment les molécules de composés organiques. Les principaux éléments constitutifs de cette soupe étaient le carbone, l'oxygène, l'hydrogène et l'azote. On pense que de ce bouillon ont émergé les premiers composés organiques qui ont ensuite donné naissance à des organismes unicellulaires tels que les bactéries. La soupe de notre exemple ne comportera que trois de ces quatre éléments, à savoir C (carbone), O (Oxygène) et H (Hydrogène).
Les combinaisons ou molécules pouvant être formées dans notre modèle sont: H, H_2, H_2O (eau), C, CH, CH_2 (méthylène), CH_3 (méthyle), CH_4 (méthane), OH, O, O_2, O_3 (ozone), CO (monoxyde de carbone), CO_2 (dioxyde de carbone)[28]. Ainsi nous considérons au total, 14 molécules différentes, si l'on inclut les atomes des trois éléments originaux, H, C et O, en tant que molécules mono-atomiques. Le modèle a pour objectif de traduire, dans le langage NetLogo, une stratégie permettant aux atomes de la soupe originale de former des molécules, au moyen d'un processus aléatoire. Après avoir exécuté le modèle, nous pouvons obtenir une estimation des pourcentages de molécules de chaque type auxquelles ce processus aléatoire a donné naissance.

[27] Harold Abelson, Julie Sussman & Gerald Jay Sussman, Structure et interprétation des programmes informatiques (traduit de l'anglais par Michel Briand Isabelle Borne et André Pic), InterEditions, Paris 1992.

[28] Ce modèle n'inclut pas toutes les molécules qui peuvent être formées à partir des atomes H, C et O, ni les petites molécules de moins de 6 atomes

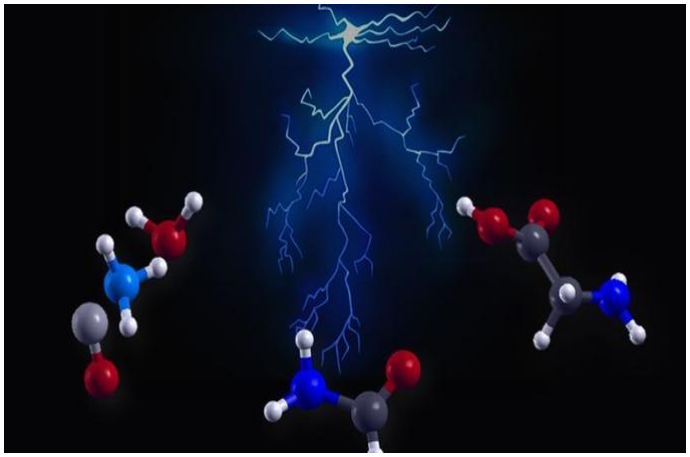


Figure 5.9 : Combinaison d'atomes et molécules

Plan général et problèmes à résoudre. L'idée centrale du plan est la suivante:

1. Les atomes sont répartis entre les tortues, en attribuant à chaque tortue une variable «atome» variable qui peut prendre l'une des trois valeurs "C", "H", "O" (carbone, hydrogène et oxygène), au moyen de la commande:

   ask turtles [set atome one-of ["H" "C" "O"]]

   Les atomes parcourent alors le monde de façon aléatoire en cherchant des molécules avec lesquelles se combiner.

2. Les parcelles hébergent des molécules représentées par des listes. Initialement, il n'y aura sur les parcelles que des molécules mono-atomiques de carbone, d'hydrogène ou d'oxygène respectivement représentées par les listes ["C"], ["H"] et ["O"]. Cependant à mesure que le modèle évolue, ces molécules acceptent de se combiner aux atomes ambulants «transportés» par les tortues pour former des molécules plus grosses, telles que la molécule d'eau ["H" "H" "O"] ou celle du monoxyde de carbone ["C" "O"]. Chaque parcelle enregistre la molécule qui occupe son territoire dans une variable appelée «molécule». La valeur initiale de cette variable est définie à l'aide de la commande suivante :

   ask patches [set molécule one-of [[”H”] [”C”] [”O”] [”V”] [”V”] [”V”] [”V”] [”V”] [”V”]]]

   où la valeur ["V"] représente les «parcelles vides», c'est-à-dire les parcelles où il n'y a pas de molécule mono-atomique H, C ou O.

Le grand nombre de parcelles de type «V» dans le monde est un choix délibéré du modélisateur.

3. Afin de réduire le nombre de listes représentant la même molécule, il convient de représenter les molécules au moyen de listes classées par ordre alphabétique, appelées «molécules ordonnées». Par exemple, la liste ["H" "O" "H"] ne représente pas la molécule d'eau car elle n'est pas classée par ordre alphabétique. La primitive «sort-by» (ordonner-selon) est chargée d'effectuer ce tri et de convertir la liste ["H" "O" "H"] en ["H" "H" "O"].

4. Les tortues (les atomes de "C", "O" et "H") parcourent le monde et chaque fois qu'un atome (une tortue) atteint une parcelle où se trouve une molécule, l'atome examine la possibilité de se combiner avec la molécule qui est sur cette parcelle pour former une plus grosse molécule[29]. Si cette combinaison est possible, l'atome doit obligatoirement se lier à la molécule qui occupe la parcelle pour former une nouvelle molécule. L'atome qui vient de se lier doit ensuite mourir («die»), de manière afin qu'il ne puisse pas se combiner à nouveau avec d'autres molécules (principe de conservation de la matière). En cas d'impossibilité de se combiner avec une molécule située sur une parcelle, l'atome poursuit son chemin à la recherche d'une meilleure occasion.

5. Lorsqu'un atome est combiné à une molécule située sur une parcelle, son nom est ajouté en dernier à la liste de la molécule. Par exemple, supposons une parcelle sur laquelle est située une molécule de type ["O"]. Si cette parcelle reçoit la visite d'un atome de type "H", du fait que, selon le modèle, l'hydrogène peut être combiné avec l'oxygène, la variable «molécule» de la parcelle passe de la valeur ["O"] à la nouvelle valeur ["O" "H"] qui, après le processus de classement par ordre alphabétique, prendra la valeur ["H" "O"].

Les règles ci-dessus décrivent le mécanisme de formation de nouvelles molécules dans la «soupe organique» du modèle. Ces règles stipulent que les molécules ne peuvent être formées que par le biais de l'interaction d'une tortue avec une parcelle, jamais par celui d'interactions entre deux tortues

[29] Les règles de combinaison entre atomes et molécules et leur traduction en langage NetLogo sont discutées plus loin dans le texte.

ou entre deux parcelles. Les règles 4 et 5 décrivent le mécanisme de base qui régit la formation de molécules et il faut maintenant le détailler.

Lorsqu'une tortue arrive sur une parcelle où se trouve déjà une autre molécule, elle doit se poser la question suivante:

Est-il possible que l'atome que je représente (que je «transporte») puisse être combiné à la molécule de la parcelle sur laquelle je viens d'arriver ?

Il faut traduire cette question en code NetLogo en utilisant une expression conditionnelle. Puisqu'il y a 3 types d'atomes et 14 types de molécules, 48 situations possibles peuvent se présenter (48 = 14 x 3). Certaines de ces situations permettent de former des molécules, d'autres pas. Lorsqu'elle arrive sur une parcelle, la tortue porteuse d'un atome doit avoir une réponse à la question ci-dessus et doit, pour ce faire identifier dans laquelle de ces 48 situations il se trouve. Cela impliquerait l'écriture de 48 expressions conditionnelles qu'il faudrait traduire en code NetLogo.
Heureusement, le nombre d'expressions conditionnelles peut être réduit à seulement 14 en introduisant ce que nous appellerons des «molécules extensionnées»[30]. Afin de savoir s'il peut se combiner ou non, l'atome passe en revue une liste de 14 conditions de base basées sur le concept de molécule extensionnée que nous expliquons ci-dessous. Si l'atome se trouve dans l'une des situations décrites par l'une des 14 expressions conditionnelles, il se combinera avec une autre molécule. Sinon il continuera sa marche aléatoire.

Simplement définie, *une molécule extensionnée est une manière de représenter la molécule sur la parcelle, molécule à laquelle on ajoute la liste d'atomes avec lesquels ladite molécule pourrait être combinée.*

Plus précisément, *une molécule extensionnée est constituée de deux listes : la première liste est formée par la molécule ordonnée qui se trouve sur la parcelle ; la seconde contient les atomes avec lesquels la molécule ordonnée pourrait être combinée (dans ce modèle)*.

Voici deux exemples pour clarifier le concept de molécule extensionnée

La molécule extensionnée du monoxyde de carbone CO est: [[" C " O "] [" O "]]

[30] Ce qualificatif a été préféré à «étendues» ou «élargies» car il évoque la notion de combinaison entre éléments comme dans l'exemple «Le produit X peut être extensionné avec un agrégat pour produire un béton à prise rapide».

La première liste représente la molécule de CO et la deuxième liste contient "O" qui est le seul atome avec lequel la molécule de CO est autorisée à se combiner (dans ce modèle).

La molécule extensionnée de l'eau H_2O est: [["H" "H" "O"] []]
La deuxième liste est vide car une molécule d'eau ne peut, dans ce modèle, être combinée avec un autre atome.

Il est important de préciser que les molécules extensionnées ne sont pas situées sur les parcelles. En effet, «molécule extensionnée» n'est pas, contrairement à «molécule», une variable propre aux parcelles, de type «patch-own» : c'est un simple mécanisme de requête qui fait partie du code et qui consiste en une liste d'expressions conditionnelles - une pour chaque type de molécule - que l'interpréteur parcourt à chaque fois qu'un atome arrive sur une parcelle, à la recherche d'une molécule avec qui se combiner.

Prenons l'exemple d'une tortue porteuse de l'atome "O" et qui arrive sur une parcelle où se trouve la molécule ["C"]. À cette dernière molécule correspond la molécule extensionnée [["C"] ["O" "H"]], ce qui signifie que la molécule ["C"] ne peut être extensionnée qu'en se combinant avec des atomes de type "O" ou "H".

Examinons maintenant le sens de certaines expressions figurant dans le code, en prenant toujours pour exemple un atome "O" qui atteint une parcelle où se trouve une molécule ["C"]:

1) "O" désigne la valeur de la variable «atome» de la tortue.
2) ["C"] est la valeur de la variable «molécule» de la parcelle
3) ["C"] est aussi la valeur de l'expression *first [["C"] ["O" "H"]]* car ["C"] est le premier membre de la liste [["C"] ["O" "H"]].
4) ["O" "H"] est la valeur de l'expression *last [["C"] ["O" "H"]]* car la liste ["O" "H"] est le dernier membre de la liste [["C"] ["O" "H"]], dont les éléments sont deux listes.
5) L'expression *member? atome last [["C"] ["H" "O"]]* a la valeur True (vrai) car cette expression est équivalente à la question : «atome» (dont la valeur est "O") est-il membre de la liste*["O" "H"]*? La réponse est évidemment oui.

La réponse à la question 5) étant affirmative, la condition est remplie et, par conséquent, l'interprèteur ordonne l'exécution des actions suivantes:
a) extensionner la liste ["C"] en plaçant l'atome "O" en dernier, b) mettre ladite liste extensionnée par ordre alphabétique et c) Demander à la tortue de disparaître («die»).

En langage NetLogo :

[set molécule ordonner lput atome molécule die].

La molécule ["C"] devient alors ["C" "O"]. Une fois ces explications fournies, nous pouvons alors comprendre la structure et la signification de n'importe laquelle des 14 expressions conditionnelles. La première d'entre elles correspond à la molécule de carbone ["C"]:

if molécule = first [["C"] ["H" "O"]] and member? atome last [["C"] ["H" "O"]]
[set molécule ordonner lput atome molécule die]

«Si la molécule sur cette parcelle est [" C "] et si l'atome que (toi tortue) tu portes est membre de la liste [" H" "O "], ajoute ton atome à la fin de la liste [" C "], ordonne (alphabétiquement) cette liste et disparais».

Voici le code du modèle:

turtles-own [atome]
patches-own [molécule]

to setup
  clear-all
  crt tortues [setxy random-xcor random-ycor]
ask turtles [set atome one-of ["H" "C" "O"]]
ask patches [set molécule one-of [["H"] ["C"] ["O"] ["V"] ["V"] ["V"] ["V"] ["V"] ["V"]]]
ask patches with [molécule != ["V"]] [set pcolor white]
   reset-ticks
end

to go
ask turtles [
     if molécule = first [["C"] ["H" "O"]] and
  member? atome last [["C"] ["H" "O"]]
     [set molécule ordonner lput atome molécule die]

     if molécule = first [["H"] ["C" "H" "O"]] and
  member? atome last [["H"] ["C" "H" "O"]]
     [set molécule ordonner lput atome molécule die]

```
   if molécule = first [["O"]   ["C" "H" "O"]]  and
member? atome last [["O"]   ["C" "H" "O"]]
   [set molécule ordonner lput atome molécule die]

    if molécule = first [["H" "H"]   ["O"]]  and
member? atome last [["H" "H"]   ["O"]]
   [set molécule ordonner lput atome molécule die]

    if molécule = first [["H" "O"]   ["H"]]  and
member? atome last [["H" "O"]   ["H"]]
   [set molécule ordonner lput atome molécule die]

    if molécule = first [["C" "H"]   ["H"]]  and
member? atome last [["C" "H"]   ["H"]]
   [set molécule ordonner lput atome molécule die]

    if molécule = first [["C" "H" "H"]  ["H"]]  and
member? atome last [["C" "H" "H"]  ["H"]]
   [set molécule ordonner lput atome molécule die]

    if molécule = first [["C" "H" "H" "H"]   ["H"]]  and
member? atome last [["C" "H" "H" "H"]   ["H"]]
   [set molécule ordonner lput atome molécule die]

    if molécule = first [["C" "H" "H" "H" "H"]   []]  and
member? atome last [["C" "H" "H" "H" "H"]   []]
   [set molécule ordonner lput atome molécule die]

   if molécule = first [["C" "O"]   ["O"]]  and
member? atome last [["C" "O"]   ["O"]]
   [set molécule ordonner lput atome molécule die]

  if molécule = first [["C" "O" "O"]   []]  and
member? atome last [["C" "O" "O"]   []]
   [set molécule ordonner lput atome molécule die]

  if molécule = first [["O" "O"]   ["O"]]  and
member? atome last [["O" "O"]   ["O"]]
   [set molécule ordonner lput atome molécule die]

  if molécule = first [["O" "O" "O"]   []]  and
member? atome last [["O" "O" "O"]   []]
   [set molécule ordonner lput atome molécule die]
```

```
    if molécule = first [["H" "O" "O"]   []]  and
  member? atome last [["H" "O" "O"]   []]
     [set molécule ordonner lput atome molécule die]

     set heading random 360 fd 1]
 if count turtles = 0 [stop]
end

to-report ordonner [molec]
report sort-by [ [x1 x2] -> first x1 < first x2]  molec
end

to montrer
ask patches with [first molécule != "V"] [show molécule]
end
```

Explications et commentaires supplémentaires.

L'idée de la molécule extensionnée est la base de la stratégie utilisée dans ce modèle pour résoudre le problème de la formation de nouvelles molécules. Nous ne prétendons pas que ce soit la seule stratégie possible ni la meilleure. C'est une idée qui a émergé après avoir analysé le problème et après avoir essayé d'autres pistes possibles. Les idées naissent dans nos cerveaux grâce à des mécanismes mentaux complexes et encore mal compris, dont l'une des ressources principales est le langage naturel, dans notre cas le français. Après avoir élaboré une stratégie de solution, le modélisateur essaie d'exprimer, dans le langage NetLogo ou dans le langage de programmation qu'il a choisi d'utiliser, le «prototype de solution trouvé» C'est, arrivé à cette deuxième étape, que le modélisateur se rendra compte si le langage choisi est adapté à la solution préconisée.

Résultats de quelques consultations.

Nous allons, pour terminer, exécuter le modèle avec une population initiale de 1000 tortues et un monde composé de 1089 parcelles (structure par défaut) et analyser les résultats d'un certain nombre de consultations faites à partir de la fenêtre de l'observateur. Pour savoir combien d'atomes de C, H ou O font initialement partie de la soupe, il faut ajouter aux 1000 atomes «portés» par les tortues ceux qui sont initialement situés sur les parcelles:

type count patches with [molécule != ["V"]]

==> 354, cela signifie qu'au total, le nombre d'atomes est 1000 + 354 = 1354

Voyons combien de molécules d'eau se sont formées après l'exécution du modèle:
type count patches with [molécule = ["H""H""O"]]
==> 95, ce qui correspond à 7 % (95/1354) d'eau.
Examinons la quantité de dioxyde de carbone:
type count patches with [molécule = ["C""O""O"]]
==> 112, ne vous inquiétez pas ! À ce stade, il est possible que le dioxyde de carbone soit même bénéfique pour le processus évolutif de la vie. Nous avons également trouvé 25 molécules de méthane CH_4, mais aucune de monoxyde de carbone CO. Si l'on modifiait la fréquence de chaque lettre H, C, O, V dans la liste de la procédure «setup», on observerait des changements dans la quantité et les types de molécules qui se forment.

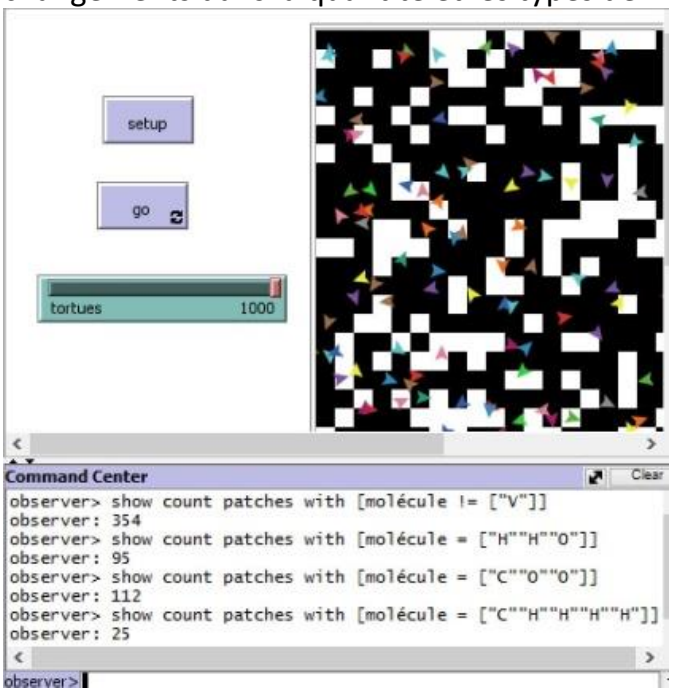


Figure 5.10 : Résultats de quelques consultations

Exercice 5.6: Ajoutez l'azote "N" au modèle et quelques nouvelles molécules, par exemple: dioxyde d'azote NO_2, ammoniaque NH_3, ammonium NH_4, acide nitrique HNO_3, acide nitreux HNO_2

Interaction entre agents II

Quand nous avons précédemment abordé le thème des interactions entre agents, nous avons seulement traité des interactions basées sur la proximité spatiale. Dans cette section, nous allons discuter des interactions basées sur la possibilité qu'ont les agents de consulter ou de modifier les valeurs des

variables des autres agents. À cet égard, établissons quelques principes qui peuvent servir de guide.

1) Un agent A peut consulter la valeur des variables d'un autre agent B, même s'il ne s'agit pas d'un agent du même type ou si A n'appartient pas à la famille de B, pourvu que l'identité de B soit spécifiée.

2) Un agent A peut modifier la valeur des variables spécifiques à un autre agent B en demandant à B d'effectuer lui-même la modification. Les tortues constituent une exception à cette règle: elles peuvent modifier les valeurs des variables de la parcelle sur lesquelles elles se trouvent comme s'il s'agissait de l'une de leurs propres variables. Des exemples de ce genre d'interactions ont été vus dans les modèles «Comptage de visites», «Foire du livre» et «Soupe primordiale de la vie».

L'ensemble de commandes suivant sur les interactions entre agents vous aidera à comprendre le problème. Pour illustrer ces exemples d'interactions, on construit un petit environnement dans lequel évoluent des agents de types différents avec leurs propres variables. Les requêtes sont effectuées à partir de la fenêtre de l'observateur.

Exemple 27: Les agents se posent des questions

Dans cet exemple, l'environnement est composé de : deux tortues, une famille de tortues appelées «loups», la totalité des parcelles du monde et un lien entre deux tortues. Chaque type d'agent a sa propre variable: les tortues la variable «âge», les loups la «vitesse», les parcelles la «hauteur» et les liens la variable «élasticité».

Définition de l'environnement :

```
turtles-own[âge]
patches-own[hauteur]
links-own[élasticité]
breed[loups loup]
loups-own[vitesse]

to setup
clear-all
crt 2 [set color red set âge random 50 fd 8]
create-loups 1 [set color white set vitesse 50 fd 5]
```

```
ask patches [set hauteur random 1000] ;; chaque parcelle a sa propre
hauteur
ask turtle 0 [create-link-with turtle 1] ;; on crée seulement un lien
ask link 0 1 [set élasticité 5]
end
```

Après avoir créé le bouton «setup» et exécuté les procédures précédentes qui constituent l'environnement, on écrit les commandes suivantes dans la fenêtre de l'observateur :

ask turtle 0 [show [âge] of turtle 1]
==> (turtle 0): 32, la tortue 0 a consulté l'âge de la tortue 1. La primitive «of» exige que la variable «âge» soit incluse entre crochets pour que la valeur de cette variable soit rapportée.

ask turtle 0 [show âge]
==> (turtle 0) : 25, lorsqu'un agent consulte la valeur d'une de ses propres variables, les crochets ne sont pas nécessaires
ask turtle 0 [show [âge] of turtle 0]

- ⇨ **(turtle 0): 25,** «of» nécessite l'emploi de crochets même si un agent consulte sa propre variable

ask turtle 0 [show [âge] of self], autre exemple ou l'emploi de «of» nécessite l'emploi de crochets.
==> (turtle 0): 25
ask turtle 0 [show [âge] of myself], un message d'erreur apparait car on ne peut utiliser «myself» dans cette situation.

Examinons quelques autres cas entre agents distincts:
ask turtle 0 [show [hauteur] of patch 8 8]
==> (turtle 0): 325, la tortue indique la valeur de la variable «hauteur» d'une parcelle spécifique.
ask turtle 1 [show hauteur]
==> (turtle 1): 598, surprise! La tortue 1 indique la hauteur de la parcelle sur laquelle elle se trouve sans qu'il soit nécessaire de spécifier l'identité de la parcelle, ni d'utiliser «of» ni de mettre «hauteur» entre crochets. Ceci est dû au fait que les tortues peuvent traiter les variables de la parcelle sur laquelle elles se trouvent comme si elles étaient leurs propres variables.

Ce privilège des tortues ne fonctionne pas à l'inverse, des parcelles aux tortues et la raison en est simple, la commande: «ask patch 0 0 [show âge]» produit une erreur car il pourrait y avoir plusieurs tortues sur la parcelle 0 0 et l'interpréteur ne sait pas à laquelle de celles-ci la question s'adresse. La forme correcte doit spécifier l'agent:

ask patch 0 0 [show [âge] of turtle 1], «of» requiert l'emploi de crochets pour la variable «âge»
==> (patch 0 0): 22
ask link 0 1 [show [âge] of turtle 1]
==> (link 0 1): 22
ask loup 2 [show [élasticité] of link 0 1]
==> (loup 2): 5, le loup 2 consulte la variable propre au lien 0 1.
ask link 0 1 [show [vitesse] of loup 2]
==> (loup 2): 50, le lien 0 1 consulte la variable propre au loup 2
ask link 0 1 [show élasticité]
==> (link 0 1): 5, ne requiert pas l'emploi de crochets car c'est un lien qui demande la valeur d'une de ses propres variables.
ask patch 8 9 [set hauteur [âge] of turtle 0], la parcelle 8 9 fixe sa hauteur à 25, qui est l'âge de la tortue 0.

Voyons maintenant des exemples dans lesquels un agent modifie la valeur d'une variable d'un autre agent. Commençons par demander son âge à la tortue 1 :

ask turtle 1 [show âge]
==> (turtle 1): 34
Maintenant la tortue 0 veut changer l'âge de la tortue 1 :
ask turtle 0 [ask turtle 1 [set âge 99]], vérifions que cela a fonctionné:
ask turtle 1 [show âge]
==> (turtle 1): 99

En plus de pouvoir consulter sans demander la permission, les tortues peuvent également, comme nous l'avons déjà mentionné, modifier les valeurs des variables de la parcelle sur laquelle elles se trouvent.

ask turtle 0 [show hauteur]
==> (turtle 0): 932
ask turtle 0 [show patch-here]
==> (turtle 0): (patch -8 2), la tortue 0 est sur la parcelle (-8, 2)
ask turtle 0 [set hauteur 19]
ask patch -8 2 [show hauteur]
==> (patch -8 2): 19

Promenade III

Les ordinateurs ont fait leur entrée dans les salles de classe à la fin des années 80. Au cours des années suivantes, nous avons assisté à un

développement spectaculaire de la capacité des ordinateurs personnels et à l'émergence de nombreuses applications logicielles, Ce développement, qui se poursuit encore aujourd'hui, a pénétré bien des domaines dont, en particulier le domaine des communications: aujourd'hui, un téléphone cellulaire est un petit ordinateur portable qui offre également une connectivité avec le reste du monde et un accès au Web, une ressource qui n'existait pas avant la fin des années 90. Ce développement impressionnant a inévitablement influencé la façon dont la technologie a modifié nos vies et a contribué à alimenter la discussion sur le rôle des technologies informatiques et des télécommunications dans les salles de classe. Un chapitre de cette discussion (qui ne date pas d'aujourd'hui) a été le débat sur l'opportunité d'introduire la science de l'art de la programmation en classe.

Ce débat remonte à la fin des années 80. C'est à cette époque que sont apparus les premiers «environnements de programmation», également appelés «environnements d'apprentissage». Ces environnements permettent aux étudiants de se concentrer sur des projets liés aux différentes matières de leur programme scolaire en utilisant un langage de programmation sous-jacent, qui devient un outil et non une fin en soi.
Mais même s'il n'est pas explicitement une fin en soi, le langage utilisé l'est implicitement parce que le développement de projets nécessite que les étudiants transmettent des instructions à la machine, soit sous forme de texte soit au moyen d'icônes et de la souris.

Dans les deux cas, cela met les étudiants en contact avec le raisonnement logique et le respect d'un certain nombre de règles formelles imposées par le langage sous-jacent. L'exemple le plus connu de ce type d'environnement et probablement le premier à apparaître est le langage Logo, créé en 1967 par Wally Fuerzeig, Seymour Papert et Cynthia Solomon (c'est à Papert que l'on doit l'introduction ultérieure du concept de tortue).

La phrase célèbre de Papert «apprendre au moyen de l'ordinateur et non sur l'ordinateur» résume très bien l'approche de cet environnement d'apprentissage. Par la suite, d'autres alternatives ont vu le jour, dont certaines, comme l'environnement Scratch mis au point par Resnick [18]. Ce type d'environnement ludique permet à des élèves, même d'âge préscolaire, de programmer la construction de petites histoires en manipulant des objets graphiques.

Même aujourd'hui, la discussion sur l'introduction de la programmation en salle de classe continue de soulever des questions telles que:

1. Quels environnements de programmation sont les plus appropriés?
2. Comment devraient-ils être introduits en classe?
3. À partir de quel âge faudrait-il le faire?
4. Combien d'heures par semaine est-il recommandé de consacrer à cette activité?
5. Faut-il laisser de la place à la programmation dans un calendrier scolaire serré même si cela implique qu'il faille sacrifier du temps consacré à d'autres matières?
6. Le moment d'apporter des modifications aux programmes scolaires pour faire de la place à la programmation est-il bien choisi ?
7. Le programme devrait-il s'adresser à tous les élèves ? Uniquement aux élèves intéressés ? Pendant les heures classes normales ou en dehors des heures de classe?

Les réponses aux questions précédentes ont suscité une variété d'options intéressantes adaptées aux différents niveaux du système d'éducation (du niveau primaire au niveau universitaire) et NetLogo est sans aucun doute l'une de ces options. Nous ne tenterons pas de répondre aux questions précédentes dans ce livre. Nous nous limiterons à formuler quelques observations sur des points moins controversés et nous énumérerons certains des bienfaits que procure l'activité de se livrer à la programmation.

La programmation renforce les domaines cognitifs énumérés ci-dessous

1. **Elle encourage l'expression correcte des idées et l'utilisation de la langue avec précision**. Le code d'un programme requiert un degré élevé de précision dans la formulation des unités sémantiques qui le composent, similaire à celui requis par le raisonnement scientifique, en particulier en mathématiques.
2. **Elle constitue un entrainement à la résolution de problèmes**. Chaque modèle présente des problèmes à résoudre, pour lesquels il est nécessaire de concevoir et de coordonner des stratégies de résolution. La variété des problèmes qui se posent est pratiquement inépuisable, comme l'est la variété des modèles qui peuvent être conçus. La résolution des problèmes passe par les quatre étapes suivantes:

    a. Identification du problème
    b. Conception et formulation d'une stratégie de solution en langage courant (parlé ou écrit).
    c. Traduction de cette stratégie de solution en langage codé propre au langage de programmation utilisé.

d. Vérification de la solution proposée.

3. **Elle constitue une formation à l'analyse des erreurs**. La détection des raisons pour lesquelles des sections du code ne produisent pas les effets attendus est un exercice incessant d'analyse qui nécessite de la patience et de la persévérance. La détection et la correction des erreurs exigent une grande concentration qui entretient l'habitude de patience nécessaire à la résolution de problèmes.
4. **Elle favorise la concentration et l'attention focalisée**. Les problèmes posés par la création d'un modèle ou d'un programme nécessitent des moments de concentration et d'attention focalisée pendant lesquels le programmeur doit isoler le point focal de son attention des détails extérieurs au sujet. La programmation peut être un exercice bénéfique pour les personnes souffrant de difficultés de concentration telles que le «déficit d'attention».
5. **Elle fournit l'occasion de s'habituer et de se perfectionner au travail en équipe**. En effet un programme a toutes les caractéristiques d'un projet et, en tant que tel, il se prête à de nombreuses façons d'assigner différentes tâches à des personnes qui travaillent au sein d'une équipe de projet.
6. **Elle offre un terrain fertile à l'exercice de la créativité**. Confronté à un mot dont la signification est aussi large qu'insaisissable que celle attachée au mot «créativité», la meilleure réponse que l'on puisse donner à la question, «est-ce que la programmation développe la créativité?» est de regarder les modèles créés à l'aide de NetLogo (par exemple, dans la bibliothèque de modèles du menu Fichiers ou parmi les nombreux modèles disponibles sur le Web). Bien qu'il soit très difficile d'évaluer la contribution d'une activité au «développement» de la créativité, on peut certainement affirmer qu'au même titre que les mathématiques, la peinture ou la musique, la programmation offre un très large champ pour «exercer» la créativité.
7. **Avantages supplémentaires de la modélisation à base d'agents**. S'il est vrai que les avantages mentionnés ci-dessus sont applicables à n'importe quel langage de programmation, qu'il s'agisse d'un langage à vocation générale ou d'un langage à usage spécifique, la programmation à base d'agents offrent des avantages supplémentaires par rapport aux autres langages de programmation.

    a. Dans de nombreux modèles dans lesquels une tâche doit être effectuée par un ensemble d'agents, le problème n'est pas

résolu en concevant simplement une stratégie pour un agent particulier et en la répliquant pour chaque agent appartenant à l'ensemble des agents. La conception de mécanismes de coopération et de coordination entre agents est souvent nécessaire.

b. Certains de ces modèles offrent une excellente occasion de susciter des discussions en classe, discussions qui impliquent des agents représentés par des personnes et des stratégies organisationnelles formulées un peu à la manière de «pièces de théâtre» qui rappellent aux étudiants des expériences qu'ils vivent dans leur quotidien.

c. L'un des aspects les plus remarquables de la modélisation à base d'agents est qu'elle offre la possibilité d'explorer le thème des «comportements émergents» [18 et 21], thème qui étudie la relation entre les interactions au niveau individuel (au niveau des agents ou niveau «micro») et la configuration globale ou «macro» du système résultant de ces interactions.

d. Source infinie de plaisir et de loisirs créatifs. Ce point n'est pas très souvent abordé lorsque nous parlons des avantages possibles de la programmation. En ce sens, l'auteur considère que la programmation est aussi proche du domaine des arts que de la science: c'est une source inépuisable de joie où la création apparaît comme la fusion de la libre imagination et de la rigueur requise par la formulation d'une stratégie de solution et la transcription ultérieure de cette stratégie en code.

Modèle 14: distance moyenne entre les tortues dans une région délimitée

Primitives: creat-links-to (créer-liens-vers), hide-link (cacher-liens), link-lenght (longueur-de-line), sum (additionner).
Autres caractéristiques: on suggère d'augmenter la vitesse de traitement en faisant glisser la poignée du curseur de vitesse de l'interface de son mode par défaut «normal speed» vers son extrémité droite («faster»).

Dans ce modèle, nous utiliserons NetLogo pour résoudre le problème suivant: si, dans une région délimitée, c'est-à-dire comprise entre des frontières, on place un ensemble de tortues occupant des positions aléatoires, on veut savoir comment varie la distance moyenne entre les

tortues quand on augmente le nombre de tortues[31]. Le modèle se prête bien à une discussion de groupe, par exemple dans un atelier NetLogo. Il est clair que la distance moyenne entre les tortues devrait dépendre de la configuration, c'est-à-dire de la position des tortues dans le monde. Très vite dans la discussion, on observera que, si les tortues sont proches les unes des autres, les distances moyennes entre paires de tortues peuvent être réduites au minimum, voire à zéro, si elles se trouvent toutes au même endroit. D'autre part, on ne sait pas quelle pourrait être la valeur maximale que peut prendre la moyenne. La première chose qui sera observée est que la distance moyenne augmente avec la taille de la région. En revanche, le comportement de la moyenne semble être régi par deux tendances opposées: à mesure que le nombre de tortues augmente, si deux groupes de tortues sont séparés l'un de l'autre, la distance entre les membres de groupes différents fait augmenter la valeur de la moyenne, tandis que la distance entre les membres du même groupe la fait diminuer. On découvre bientôt que, sauf dans de très petits groupes, le principe selon lequel «il n'est pas possible d'être à égale distance du monde entier» s'applique. Une façon d'étudier le problème consiste à placer les tortues au hasard sur le terrain. Une autre façon, que nous présentons comme un jeu dans le chapitre suivant, consiste à placer les tortues manuellement, avec la souris, en essayant d'obtenir la valeur la plus élevée pour la moyenne. La distance moyenne est calculée en additionnant les valeurs des distances entre toutes les paires de tortues et en les divisant par le nombre de paires. Par exemple, s'il n'y avait que trois tortues, numérotées 0, 1, 2 et que nous appelons D_{ij} la distance entre les tortue i et j, la distance moyenne D serait donnée par l'expression:

$D = (D_{01} + D_{02} + D_{12}) / 3.$

Activités préparatoires. Créer un curseur appelé «population», qui détermine le nombre de tortues dans le monde. Configurer les boutons «setup» et «go» (cette fois, **ne pas** cocher pas la case «Forever»). Définir le monde selon la topologie du carré.

La procédure est exécutée en appuyant sur les boutons «setup» et «go». Afin d'accélérer la vitesse d'exécution de la procédure, il est recommandé de faire glisser vers l'extrême droite («faster») la poignée du curseur qui contrôle la vitesse des «ticks» dans l'interface.

[31] La pandémie de COVID-19 qui sévit au moment ou sont écrites ces lignes (Novembre 2020) et les règles de distanciation sociale et physique recommandées pour minimiser les risques de propagation du virus illustrent le caractère pratique des simulations multiagents.

Plan général et problèmes à résoudre.

Le problème à résoudre ici est de savoir comment programmer le calcul de la distance moyenne, problème qui se réduit en réalité à la manière de calculer les distances entre toutes les paires de tortues possibles. En procédant à ce calcul il faut éviter les répétitions, c'est-à-dire que si la distance D_{12} de la tortue 1 à la tortue 2 est calculée, la distance D_{21} de la tortue 2 à la tortue 1 ne doit pas être incluse, car évidemment c'est la même distance.

Une solution basée sur l'utilisation de liens.
Avant de présenter le code de cette stratégie, certaines remarques sont nécessaires.

1. Une fois qu'un ensemble de tortues a été créé (à l'aide du curseur de la variable «population»), la commande «create-links-to other turtles» permet de créer des liens entre chaque paire de tortues. Mais cette commande, au lieu de créer un lien par paire, crée deux liens. En effet, l'interpréteur prend chaque tortue une par une et crée un lien avec chacune des autres tortues. Par exemple, lorsque c'est au tour de la tortue 0, un lien est créé avec la tortue 1, mais lorsque c'est au tour de la tortue 1, un lien sera aussi créé avec la tortue 0, ce qui produira la duplication mentionnée. Cette duplication est prise en compte lors du comptage des liens à l'aide de la commande «count links».
2. Avec la commande «set total lput link-length total», on construit une liste (initialement vide) appelée «total». Cette liste contient les longueurs («link-length») de tous les liens. La longueur de chaque nouveau lien créé est placée en dernière position dans la liste. Comme chaque lien est dupliqué, chaque longueur est également dupliquée dans la liste «total».
3. Pour calculer la moyenne, nous devons d'abord ajouter les longueurs de tous les liens. Ces longueurs sont stockées dans la liste «total». La primitive «sum» permet d'ajouter toutes les valeurs numériques contenues dans une liste. Cela nous évite de traiter la liste en utilisant certains des mécanismes d'itération, tels que la récursion (présentée dans le chapitre 7).
4. Nous pourrions calculer la moyenne des longueurs des liens en invoquant la primitive «mean» (moyenne) ou le faire en utilisant la formule :
   Moyenne des longueurs = somme des longueurs des liens / nombre de liens
   Si, dans la formule précédente, il y avait deux liens pour chaque paire de tortues, le numérateur et le dénominateur seraient dupliqués,

mais cela n'affecterait pas la valeur de la moyenne car la duplication du numérateur annule celle du dénominateur. Cependant, il est important d'éliminer la duplication de liens pour une raison plus importante, à savoir éviter de dupliquer le temps de traitement de la simulation.

5. La duplication des liens peut être éliminée en ajoutant une condition: un lien ne sera établi entre la tortue i et la tortue j que si leurs numéros d'identification obéissent à la relation (i > j). Si cette condition est remplie, un seul lien sera créé pour chaque paire de tortues. Le code pour exprimer cette condition est:

   ask turtles [create-links-to other turtles with [who > [who] of myself]]

   dont la traduction française est: «demander aux tortues de créer des liens avec les autres tortues dont le numéro d'identification est supérieur à mon propre numéro d'identification».

   Pour augmenter la vitesse de la simulation, il est recommandé de déplacer le curseur «ticks» de l'interface, de sa position intermédiaire («normal speed» -vitesse normale-) vers son extrémité droite («faster» - vitesse accélérée-).

Le code du modèle est le suivant :

```
globals[total]

to setup
  clear-all
  crt population [setxy random-xcor random-ycor]
  set total []
end

to go
  ask turtles [create-links-to other turtles with [who > [who] of myself]]
   ask links [
    set total lput link-length total]
  calculer-moyenne
  show length total
end

to calculer-moyenne
  type "La moyenne est: " print sum total / count links
```

end

Explications et commentaires supplémentaires.

En mathématiques combinatoires, on sait que le nombre de paires pouvant être formées à partir d'un ensemble de N individus est égal à $(N^2 - N) / 2$. Cela montre que le nombre de paires de tortues (et donc le nombre de liens) augmente approximativement en raison directe du carré du nombre de tortues[32].En conséquence, l'interpréteur doit effectuer un grand nombre de calculs lorsque N augmente. Par exemple, pour un millier de tortues, le nombre de liens serait proche du demi-million. Nous aurions pu utiliser la primitive «mean» pour calculer la moyenne des distances entre les paires, mais nous ne l'avons pas fait, de façon délibérée, pour revoir l'utilisation des primitives «sum» et «count» et pour étendre la portée de la discussion.

Avec la topologie du carré et avec une population de 1000 tortues aléatoirement positionnées, on observe que la distance moyenne est proche de la valeur 17 lorsque le carré du monde a une longueur de 32 unités, dont la moitié (ou «rayon») est 16. Si on construit le monde selon une topologie toroïdale, la valeur de la moyenne décroît sensiblement jusqu'à une valeur proche de 11. Cela est parfaitement logique car il y aura maintenant des paires de tortues dont la distance est inférieure dans cette topologie par rapport à celle du carré.

Un autre point intéressant à mentionner est le suivant. Nous pourrions répartir les tortues au hasard à l'intérieur d'un disque en utilisant un système de coordonnées polaires[33]. Sachant que les tortues naissent avec des orientations attribuées au hasard, on pourrait demander à chacune d'entre elles d'avancer d'un certain nombre de pas au hasard, mais sans dépasser la distance entre le centre du monde et l'un de ses côtés (par défaut, cette valeur est de 16 pour un monde dont la topologie est celle du carré). Nous pourrions obtenir une distribution radiale en remplaçant la commande *crt population [setxy random-xcor random-ycor]* par la commande *crt [fd random 17*][34].

[32] C'est un résultat facilement démontrable par la méthode d'induction mathématique

[33] On rappelle qu'en mathématique les coordonnées polaires sont un système de coordonnées dans lequel chaque point du plan est représenté par un angle et une distance.

[34] Petit rappel : random 17 indique une valeur aléatoire comprise entre 0 et 16.

Peut-être pourrions-nous nous attendre à une moyenne de 16, en pensant que l'unité de plus que nous avons obtenue au début est due aux coins du carré qui constitue le monde. Avec 1000 tortues dans une région circulaire de rayon 16, on obtient des résultats illustrés par la figure suivante (Figure 5.11):

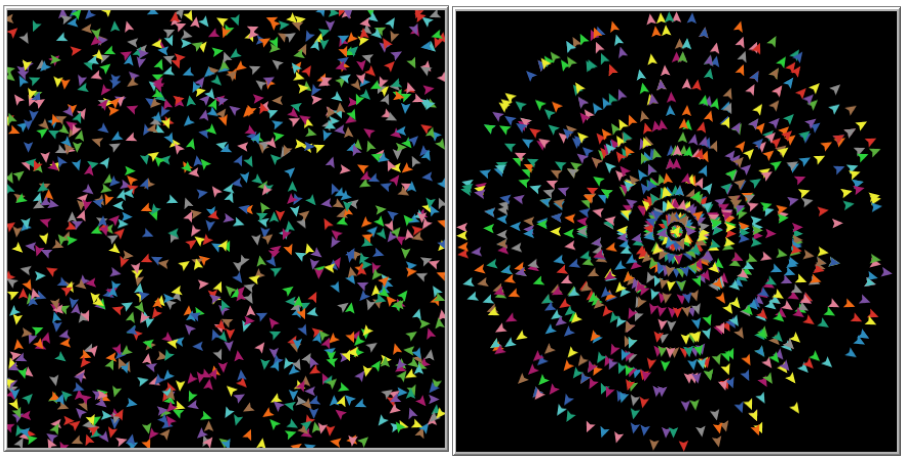


Figure 5.11 : A gauche mille tortues avec la distribution cartésienne initiale et à droite avec la distribution radiale

Alors que la valeur de la moyenne de la distribution aléatoire «cartésienne» à gauche semble tendre vers 17, la valeur de la moyenne de la distribution radiale semble être très proche de 12, une valeur nettement inférieure à celle du premier cas. Est-il possible de trouver une explication à cette différence? Nous pouvons poursuivre l'expérience en distribuant les tortues à l'intérieur d'un disque de rayon 16 au moyen d'un mécanisme aléatoire cartésien plutôt que d'un mécanisme aléatoire radial. Cela pourrait être réalisé en éliminant les tortues dont la distance au centre est supérieure à 16. Il suffirait d'éliminer les tortues situées dans les coins avec l'instruction:

crt population [setxy random-xcor random-ycor if distance patch 0 0 > 16 [die]]

Le résultat de cette modification du code est illustré sur la figure 5.12

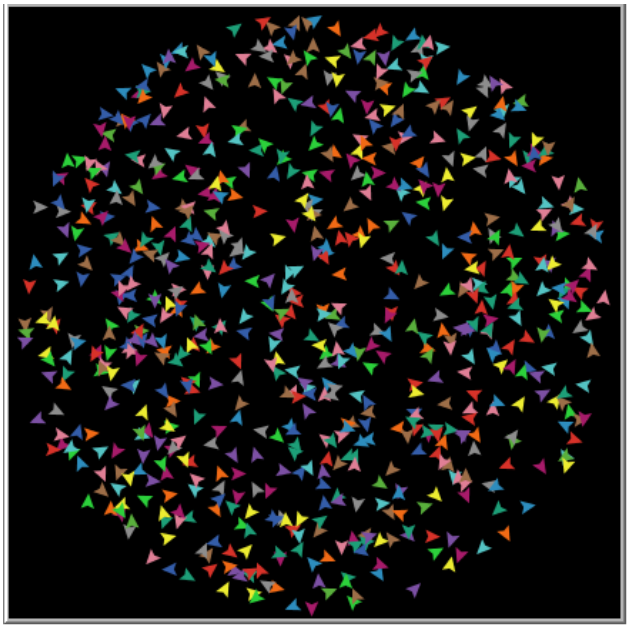


Figure 5.12 : Population aléatoire de mille tortues (moins celles qui sont retirées des coins) dans un cercle de rayon égal à la moitié du côté du carré du monde.

Exemples de projets individuels ou de groupe

L'un des avantages de travailler avec la modélisation à base d'agents dans le domaine de l'éducation est la facilité avec laquelle il est possible de trouver des phénomènes naturels ou du comportement humain (y compris les phénomènes de la vie quotidienne) qui se prêtent à ce type de modélisation notamment en utilisant le langage NetLogo. Nous en avons déjà exposé l'une des raisons: l'ubiquité de phénomènes impliquant des agrégats d'entités possédant un certain degré d'autonomie. Nous présentons ci-dessous trois exemples de ce que pourraient être des projets de travaux individuels ou de groupe en classe. Nous nous limitons à la présentation du thème des projets et nous soulignons quelques-uns des avantages qu'ils offrent lorsqu'ils sont développés dans le cadre de projets de groupe.

Projet n ° 1: Une tornade détruit un entrepôt.

Un édifice dans lequel étaient entreposées des matières à recycler est détruit par une tornade. L'entrepôt contenait des morceaux de pièces laminées de trois types de matériaux: aluminium, plastique et carton. Suite à la tornade, ces morceaux ont été dispersés dans un rayon très étendu autour de l'entrepôt. Heureusement, la zone entourant l'entrepôt est un vaste terrain agricole inhabité, ce qui va faciliter la récupération des matériaux. La société Les Recycleurs Modernes (LRM), propriétaire de l'entrepôt, est une société dont les dirigeants ont une grande conscience

environnementale et qui savent de plus que les matériaux n'ont pas perdu de leur valeur commerciale après leur dissémination sur le terrain avoisinant. Les dirigeants de LRM décident de faire appel à NetSin, une société locale spécialisée dans le nettoyage après sinistre pour prendre en charge la collecte des morceaux de matériaux éparpillés ainsi que le nettoyage du terrain. NetSin envoie un petit groupe d'employés pour effectuer la collecte des matériaux à la main (le nettoyage subséquent du terrain se fera à l'aide de moyens mécaniques et ne nous concerne pas dans cet exercice de modélisation). Modélisez, à l'aide de Netlogo, le mécanisme de collecte des matériaux en supposant que les morceaux ont été disséminées sur une zone tellement vaste qu'il est impossible pour une personne seule de la couvrir à l'œil nu (ce qui nécessite une coordination entre les ramasseurs de déchets).

Projet n ° 2: Distribution de marchandises dans la cave du grand magasin en ligne.

Dans l'un des centres d'entreposage de la société Monazan, qui vend toutes sortes de biens et services sur Internet, les articles envoyés par les fournisseurs arrivent de façon continue et les articles commandés par les clients sortent au même rythme. Le milliardaire J.F. Bisous, propriétaire de la société, souhaite comparer deux types de stratégies de stockage des objets dans ses entrepôts géants. Dans le premier type d'entrepôts, la stratégie de traitement des objets qui arrivent consiste à les classer par catégories, en leur attribuant des rangées et des étagères regroupant des articles du même type. Par exemple, lorsqu'un appareil photo numérique arrive, on le place dans la rangée des articles électroniques, sous-secteur audio et vidéo et sur des étagères pour les appareils photographiques numériques dans des casiers propres aux différentes marques d'appareils. Dans le deuxième type d'entrepôts, un article qui entre dans l'entrepôt est placé sur la première étagère inoccupée trouvée par l'employé qui le ramasse. Cette stratégie fait que l'on trouve des articles très différents sur une même étagère. Par exemple, sur l'étagère 151H, on peut trouver un matelas gonflable, un DVD de la septième symphonie de Mahler, un flacon de pilules Dormilak contre l'insomnie, c'est-à-dire toutes sortes d'articles qui ont très peu ou pas de relations entre eux (cependant, un employé de Monazan fanatique de musique electro-funk a déclaré que pour lui les trois articles sont bien classés car ils poursuivent le même but). Votre travail est de modéliser les deux stratégies à l'aide de NetLogo et de concevoir une procédure qui permet de comparer l'efficacité de chacune des deux stratégies.

Projet n ° 3: Rotation des stocks dans un entrepôt.

La tenue d'inventaire est l'un des problèmes majeurs auxquels font face les commerçants qui offrent plusieurs gammes de produits pour satisfaire la demande de leur clientèle. Ce problème touche autant les petites et moyennes entreprises commerciales (pharmacies, quincailleries, vendeurs de pièces d'automobiles, etc.) que les grandes surfaces commerciales dont les directeurs de vente veulent tous éviter à leurs employés de livrer leur traditionnel refrain à la cliente «Je suis désolé, madame, nous n'avons pas cet article pour le moment, mais nous l'attendons dans quelques jours».
La manière dont le contrôle de l'inventaire doit être organisé, le choix de variables qui doivent être attribuées à chaque article ou à chaque catégorie d'articles, etc. constituent, pour les étudiants, un terrain de discussion très riche pour les initier à la planification, au suivi et au contrôle des activités. Par exemple, dans une pharmacie ou dans un magasin d'alimentation, le contrôle des dates d'expiration des articles est un facteur qui doit être pris en compte dans le modèle. C'est un exemple où la pratique consistant à commencer par mettre en œuvre un modèle de petite taille (quelques articles comportant peu de variables chacun) peut être très utile avant de passer à un modèle de plus grande taille.
Le modèle peut être développé de façon itérative et incrémentale, augmentant ainsi son réalisme au fur et à mesure que l'on en crée une nouvelle version. Voici, par exemple, certaines des variables que chaque article devrait avoir: prix d'achat, marge bénéficiaire, date d'entrée et date d'expiration. Mais, dépendant des secteurs et du contexte, d'autres variables pourraient être tout aussi pertinentes (délais de livraison des fournisseurs, saison de l'année, capacité physique de stockage, niveau désiré du stock de sécurité, etc.). Votre travail est de concevoir et de modéliser à l'aide de NetLogo et de façon incrémentale (en commençant par un petit nombre de produits et un petit nombre de variables) un système de gestion des stocks adapté à une petite entreprise commerciale de votre choix.

Chapitre 6: Thèmes supplémentaires

Promenade IV

Le maitre de cérémonies du théâtre de l'opéra s'adressant à la Comtesse:
«Très chère Comtesse, si cela ne vous dérange pas, je vous demanderais respectueusement de bien vouloir vous déplacer de dix sièges en avant et de bien vouloir prendre place au fauteuil numéro 10 réservé à votre illustre personne».

Ou plus succinctement dans la langue NetLogo:

ask comtesse 1 [forward 10 prendre-place 10][35]

La précision et la brièveté sont deux objectifs primordiaux des langages de programmation et sont aussi leurs plus grandes vertus. De ce point de vue, les langages de programmation présentent une grande similitude avec le langage mathématique. Les deux disciplines apprécient la précision et détestent l'ambiguïté. Tant en mathématiques qu'en programmation, les concepts et les idées doivent avoir une signification précise et unique, non sujette à plus d'une interprétation[36]. C'est pour cela que la programmation peut être incluse parmi les sciences dites exactes. Cette similitude dans le langage des deux disciplines peut parfois amener à croire, à tort, que pour programmer, il est nécessaire de posséder une grande connaissance des mathématiques.

Il faut reconnaitre qu'il peut arriver qu'un modèle ou un programme particulier nécessite des connaissances mathématiques spécifiques, ou encore des connaissances dans des domaines spécifiques tels la chimie ou le droit. Il faut également reconnaître qu'il existe des sciences hautement mathématisées telles que la physique, les statistiques, la météorologie ou l'économie. Lorsqu'il s'agit de concevoir des modèles et d'écrire des programmes informatiques dans ces domaines, il est normal que les spécialistes du domaine fournissent aux programmeurs des connaissances spécifiques et que les deux groupes, (programmeurs et spécialistes) travaillent en étroite collaboration.

Il existe d'innombrables exemples de programmes, modèles ou applications développés dans divers domaines, dont les auteurs sont des personnes ne possédant pas plus de connaissances en mathématiques que celles acquises au secondaire. Si l'on examine la bibliothèque de modèles NetLogo, on peut être surpris de découvrir que plusieurs exemples utilisent des connaissances de base en mathématiques. Pour citer un exemple spécifique, le modèle appelé «Héros et Couards» (Heroes and Cowards)[37], fait appel à de simples notions de géométrie analytique afin de calculer les coordonnées du point médian entre

[35] Note : «prendre-place» n'est pas une primitive. Il faudrait écrire une procédure pour que la Comtesse puisse savoir exactement ou s'asseoir. Le lecteur est invité à écrire une procédure qui assignerait une parcelle particulière (un siège) à une tortue donnée (la Comtesse).

[36] Cependant, certains philosophes des mathématiques, tels qu'Imre Lakatos, ont mis en doute le fait que toutes les affirmations de la mathématique informelle ont une signification unique et précise.

[37] Ce modèle est présenté et commenté dans le chapitre 2 de l'ouvrage de Wilensky et Rand [21], pages 68 et suivantes.

deux points donnés d'une droite. Il s'agit là d'un problème simple et si vos souvenirs sont lointains, vous pouvez toujours consulter un livre de base de géométrie, aller sur Internet ou faire appel à des personnes de votre entourage (famille, amis, collègues de travail, etc.).

D'autre part, il faut admettre que de nombreuses et importantes contributions aux «sciences computationnelles» et à l'informatique ont été réalisées par des mathématiciens ou des personnes proches de cette discipline ayant une solide formation mathématique: physique, génie électrique et bien sûr sciences computationnelles elles-mêmes. Prenons quelques exemples: John von Neumann, l'un des plus grands mathématiciens du XXe siècle, est à l'origine de la conception de base des ordinateurs modernes au prestigieux Institute for Advanced Studies de Princeton, alors que certains de ses collègues désapprouvaient la «pollution» de l'environnement intellectuel de l'Institut où, seules les idées – et non les «artefacts» - devaient avoir leur place. Parmi les mathématiciens qui ont inventé les langages de programmation, on peut aussi citer John McCarthy (Lisp), John Backus (Fortran), Dennis Ritchie (C), Guido van Rossum (Python). C'est à Donald Knuth que l'on doit, entre autres, l'œuvre monumentale *The Art of Computer Programming* et le langage TeX et au physicien Stephen Wolfram, le puissant langage du traitement numérique et symbolique Mathematica. En informatique et en ingénierie, il faut citer Alain Colmerauer et Allan Kay à qui l'on doit, respectivement, les langages Prolog et Smalltalk. Parmi les langages plus spécifiquement destinés au milieu éducatif, on peut citer, entre autres, Logo (Seymour Papert), NetLogo (Uri Wilensky), Scratch (Mitch Resnick) et Alice (Randy Pausch).

Interaction de l'utilisateur avec le modèle à l'aide de la souris.

Dans NetLogo, il est possible d'utiliser la souris pour interagir avec un modèle ou un programme. Quatre primitives permettent cette interaction:

mouse-down? Cette primitive rapporte la valeur «vrai» quand le pointeur de la souris se trouve dans le carré du monde et le bouton gauche de la souris est enfoncé. Sinon, c'est la valeur «faux» qui est rapportée.
mouse-inside? Rapporte «vrai» quand le pointeur de la souris se trouve dans le carré du monde et «faux» dans le cas contraire.
mouse-xcor. Rapporte la coordonnée «x» du pointeur de la souris, lorsque celui-ci se trouve dans le carré du monde.
mouse-ycor. Rapporte la coordonnée «y» du pointeur de la souris, lorsque celui-ci se trouve dans le carré du monde.

Exemple 28: Utilisation de la souris pour mettre fin à des actions

Primitives: create-link-with, one-of, other (autre), mouse-down? (souris-vers-le bas?), mouse-inside? (souris-à -l'intérieur?), thickness (épaisseur), let, sprout.
Autres détails: l'utilisateur peut interagir avec le modèle à l'aide de la souris.

Dans cet exemple, on crée des tortues sur environ un cinquième des parcelles ainsi que deux procédures indépendantes appelées «go» et «go2». Dans la procédure «go», des liens sont créés entre les tortues toutes les secondes et la création des liens est interrompue en maintenant un court moment son doigt appuyé sur le bouton gauche de la souris dont le pointeur (la flèche) fait ainsi partie du monde. Le processus de création des liens suit son cours si l'on appuie à nouveau sur le bouton «go» et la procédure s'arrête si l'on appuie une nouvelle fois sur ce bouton. Dans la procédure «go2», il suffit de cliquer sur le bouton de ladite procédure et de déplacer le pointeur de la souris sur le monde (le carré noir de l'interface) pour que des liens soient créés toutes les demi-secondes. Pour arrêter le processus de création des liens il suffit de déplacer le pointeur à l'extérieur du monde. Cependant, cette dernière action n'arrête pas la procédure (on peut vérifier cela en replaçant le pointeur sur le monde : le processus de création des liens suit son cours). Pour arrêter la procédure, il faut cliquer à nouveau sur le bouton «go2». La primitive «thickness» (épaisseur) permet de définir l'épaisseur des liens.

Activités préparatoires : Construire les boutons «setup», «go» et «go2».
Cochez la case «Forever» des deux dernières procédures.

Le code est les suivant :

```
to setup
clear-all
ask patches [let état one-of [0 1 2 3 4]
if état = 1 [sprout 1] ]
;; sur approximativement la cinquième partie des parcelles
;; la variable état est égale à 1.
end

to go
ask one-of turtles [create-link-with one-of other turtles]
ask links [set thickness 0.5]
wait 1
if mouse-down? [stop]
end

to go2
```

```
if mouse-inside? [ask one-of turtles [create-link-with one-of other
turtles
ask links [set thickness 0.2] ] ]
wait 0.5
end
```

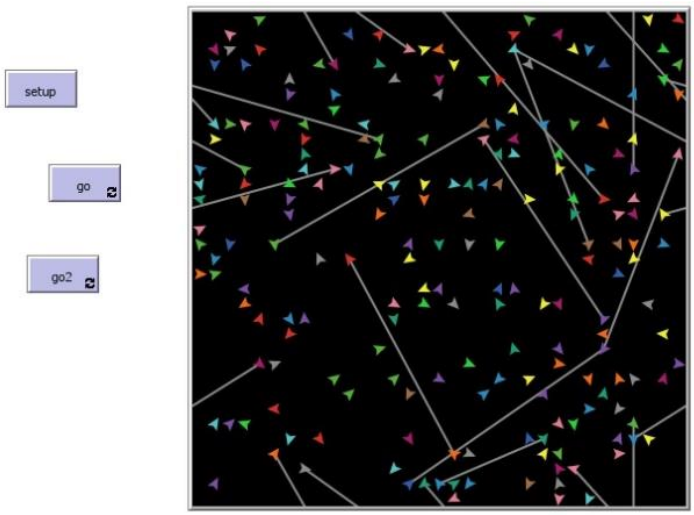


Figure 6.1 : Liens créés au hasard entre une population de tortues occupant environ un cinquième des parcelles

Explications et commentaires supplémentaires. La présence de la primitive «other» dans l'expression: «create-link-with other turtles» a pour but d'empêcher la tortue d'essayer de créer un lien avec elle-même, ce qui provoquerait une erreur. Pour peupler le monde de sorte qu'environ un cinquième des parcelles engendre («sprout») une tortue, une variable locale «état» est créée et une valeur choisie dans la liste des 5 valeurs [0 1 2 3 4], est attribuée à chaque parcelle. Seules les parcelles auxquelles la valeur 1 a été attribuée engendrent des tortues et celles-ci constituent environ un cinquième du total.

Dans de nombreuses situations, les effets obtenus par l'utilisation de la souris ne sont qu'une option et des effets identiques pourraient être obtenus à l'aide de boutons dans l'interface. C'est le cas des deux procédures précédentes «go» et «go2». Dans le prochain l'exemple, les actions de la souris ne peuvent pas être menées à bien à l'aide de boutons et l'utilisation de la souris s'avère donc indispensable.

Modèle 15: Le retour redouté de la tortue tueuse

Primitives: any? one-of, other, turtles-on, neighbors (8-voisins), neighbors4 (4-voisins), mouse-down?
Autres détails: l'utilisateur peut interagir avec le modèle à l'aide de la souris.

Un groupe de 50 tortues occupe des positions statiques dans le monde et l'une d'entre elles, la tortue tueuse, suit une trajectoire telle que les tortues qui se trouvent sur les parcelles voisines de sa trajectoire sont éliminées. L'utilisateur peut créer de nouvelles tortues à tout moment et à l'endroit qu'il veut, en appuyant sur le bouton de la souris dans le carré du monde. La procédure s'interrompt lorsque la tortue tueuse en a terminé avec toutes les autres tortues.

Activités préparatoires : Construire les boutons «setup», «go4» et «go8». Cochez les cases «Forever» des boutons «go4» et «go8».

```
to setup
clear-all
crt 50 [setxy random-xcor random-ycor]
end

to go4
ask turtle 0 [fd 1 set heading random 30 ]
ask turtle 0 [if any? other turtles-on neighbors4 [ask one-of turtles-on
neighbors4 [die]]]
wait 0.1
if count turtles = 1 [stop]
naissance
end

to go8
ask turtle 0 [fd 1 set heading random 30 ]
ask turtle 0 [if any? other turtles-on neighbors [ask one-of turtles-on
neighbors [die]]]
wait 0.1
if count turtles = 1 [stop]
naissance
end

to naissance
if mouse-down? [ask patch mouse-xcor mouse-ycor [sprout 1]]
end
```

Explications et commentaires supplémentaires. C'est le premier exemple dans lequel nous utilisons les primitives «neighbors» et «neighbors4», qui

rapportent certaines parcelles voisines de l'agent qui donne l'ordre. Dans la topologie du tore, la primitive «neighbors» rapporte un voisinage constitué des huit parcelles adjacentes à une parcelle donnée (diagonales comprises) tandis que la primitive «neighbors4» ne rapporte que le voisinage constitué des quatre parcelles adjacentes (horizontalement et verticalement) à une parcelle donnée[38]. Les boutons «go8» et «go4» correspondent, respectivement, à l'utilisation des primitives «neighbors» et «neighbors4». La commande «ask one-of turtles-on neighbors» se traduit par «demander à une tortue dans l'une des huit parcelles voisines».

La commande «set heading random 30» fait avancer la tortue tueuse en zigzags suivant une trajectoire approximativement rectiligne, avec des oscillations d'un maximum de 30 degrés. Cette trajectoire en zigzags de la tortue tueuse fait qu'il est très difficile aux autres tortues statiques de lui échapper longtemps (comme l'on peut s'en douter les tortues statiques vivent plus longtemps si le voisinage est du type von Neuman – 4 cellules adjacentes - plutôt que du type Moore – 8 cellules adjacentes -).

Modèle 16: Qui obtiendra la plus grande moyenne?

(Jeu basé sur le modèle 14 de la distance moyenne entre les tortues)

Primitives: to-report, report (rapporter), create-links-to, hide-links (masquer-liens), link-length (longueur-lien), mouse-down ? (souris-vers-le-bas ?)
Autres détails: l'utilisateur crée des tortues à l'aide de la souris.

Le présent modèle consiste en un jeu basé sur le modèle 14, jeu qui consistait à calculer la moyenne des distances entre un ensemble de tortues placées au hasard dans le monde.

Activités préparatoires: construire les boutons setup, go, joueur-1 et joueur-2, sans cocher la case «Forever» de ces quatre boutons. Dans la fenêtre d'édition,

[38] La primitive «neighbors» définit ce que l'on appelle le *voisinage de Moore* d'une parcelle donnée tandis que la primitive «neighbors4» définit le voisinage dit de *von Neuman* de ladite parcelle.

construire un curseur appelé «population» et un autre appelé «temps» avec les valeurs indiquées ci-après.

Les règles du jeu sont les suivantes: à partir d'une configuration de tortues placées aléatoirement dans le monde (par le biais de la procédure «setup»), chaque joueur a le droit d'ajouter une tortue dans le monde. Le joueur qui obtient une moyenne supérieure à celle de son adversaire gagne.

1. À l'aide du curseur «population», les deux joueurs s'accordent sur le nombre de tortues dans le monde et à l'aide du curseur «temps», ils s'accordent sur le temps (temps en secondes dont ils disposent pour jouer chacun à leur tour). Les valeurs par défaut de ces curseurs sont fixées, respectivement, à 10 tortues et à 2 secondes.
2. Pour lancer le jeu on actionne le bouton «setup». NetLogo place alors les tortues de manière aléatoire et affiche la valeur de la distance moyenne qui les sépare. Dans la version actuelle du jeu, la valeur moyenne de la configuration initiale peut être consultée en appuyant sur le bouton «go», mais il s'agit d'une consultation facultative qui n'a aucun impact sur le déroulement du jeu. Cette valeur n'est qu'une valeur de référence et on peut jouer le jeu sans appuyer sur le bouton «go».
3. Le premier joueur appuie sur le bouton «joueur-1» pour placer une tortue additionnelle dans le monde. Pour ce faire, il faut placer le pointeur de la souris à l'endroit choisi et cliquer le bouton gauche de la souris. Mais attention: ce faisant, il faut maintenir le bouton enfoncé jusqu'à ce que les liens de la nouvelle tortue avec les autres dans le monde apparaissent. Une fois ces liens visibles, le bouton de la souris peut être relâché. Les liens restent visibles pendant le nombre de secondes indiqué par le curseur «temps», puis disparaissent. La nouvelle valeur de la distance moyenne obtenue par le joueur 1 est alors affichée dans le terminal d'instructions.
4. Le joueur suivant appuie sur le bouton «joueur-2» et procède de la même manière que le premier joueur.
5. Si un joueur ne parvient pas à jouer dans le délai indiqué par le curseur «temps», il perd la partie.
6. Le joueur qui obtient la valeur moyenne la plus élevée gagne (en d'autres termes le joueur gagnant est celui des deux joueurs qui parvient à être le plus éloigné des autres tortues peuplant le monde).
7. Etant donné que le premier joueur a l'avantage de jouer avec moins de tortues que le deuxième, le jeu doit être répété en échangeant les rôles de premier et second joueurs.

Le code de ce jeu est le suivant:

globals [total]

```
to setup
  clear-all
  crt population_tortues [setxy random-xcor random-ycor]
  set total []
  ask turtles [create-links-to other turtles with [who > [who] of myself]]
  wait 1 ask links [hide-link] ;; Les liens sont cachés pour la clarté de la figure.
end

to go
ask links [set total lput link-length total] ;; chaque lien ajoute sa longueur à la
liste
 ;; «total»
 type "La moyenne est: " print moyenne
end

to-report moyenne
  report mean total ;; ici, nous laissons le calcul de la moyenne à la primitive
 ;; «mean»
end

to joueur-1
  wait temps;; attendre un temps déterminé
  if mouse-down? [ask patch mouse-xcor mouse-ycor [sprout 1 ]]
  set total []
  ask turtles [create-links-to other turtles with [who > [who] of myself]]
  ask links [set total lput link-length total]
  wait 2 ask links [hide-link] ;; permet de voir les nouveaux liens pendant 2
secondes
   type  "La moyenne du joueur 1 est: "  type moyenne print " "
end

to joueur-2
wait temps;; attendre un temps déterminé
  if mouse-down? [ask patch mouse-xcor mouse-ycor [sprout 1]]
  set total []
 ask turtles [create-links-to other turtles with [who > [who] of myself]]
  ask links [set total lput link-length total]
  wait 2 ask links [hide-link] ;; permet de voir les nouveaux liens pendant 2
secondes
  type  "La moyenne du joueur 2 est: "  type moyenne print " "
end
```

Explications et commentaires supplémentaires.
Dans les procédures joueur-1 et joueur-2, il est nécessaire de vider la liste «total» pour pouvoir recalculer la moyenne de la longueur des liens (les distances entre paires de tortues) avec la tortue qui a été ajoutée, ce qui est fait avec la commande «set total []».

Dans les pages précédentes, nous avons vu quelques méthodes qui permettent aux utilisateurs de NetLogo de saisir des informations, ainsi que les zones vers lesquelles les sorties (output) peuvent être dirigées. Mais, comme nous le verrons dans les deux prochaines sections, d'autres options peuvent être employées pour l'entrée et la sortie des données.

Options pour la saisie de données

Dans NetLogo, la saisie de données peut s'effectuer de quatre manières différentes: 1. Par une procédure faisant appel aux paramètres de saisie, 2. Par une fenêtre de saisie de données, 3. Par la primitive «user-input» et 4. En lisant les données d'un fichier.

1. Procédures avec paramètres. Ce mode a été décrit au chapitre 3, section «Procédures avec données d'entrée».
2. Objet «Fenêtre de saisie». Cette fenêtre est l'une des options offertes par l'icône «sélectionneur d'objets» de la barre d'outils de l'interface sous le nom «Input». Au moment où un objet de type «Input» est créé, deux fenêtres s'affichent : une fenêtre d'édition et une fenêtre plus petite pour la saisie de données (voir Figure 6.2).

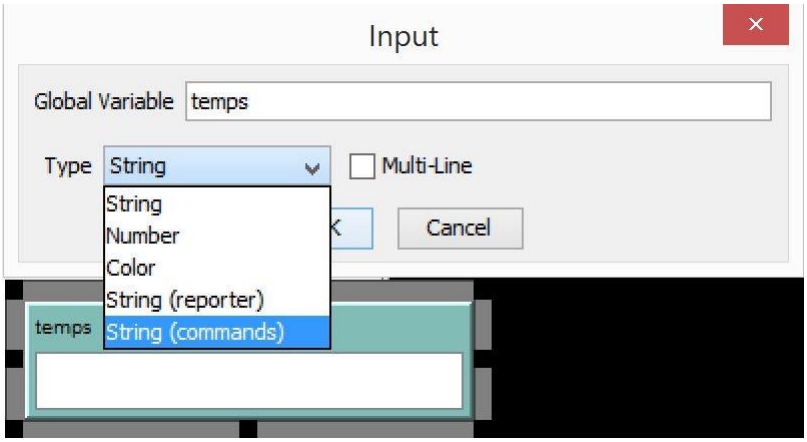


Figure 6.2 : Fenêtre de saisie

Dans la fenêtre d'édition, il faut choisir un nom pour la fenêtre de saisie de données (par exemple «temps» - qui, comme on le voit devient, de facto, une variable globale-), ainsi que le type de données à y stocker. Les options sont :

a. String (Texte)

b. Number (Nombre)

c. Color (Couleur)

d. String (rapporteur)

e. String (commandes)

La valeur par défaut est «String» (Texte). On ferme la fenêtre d'édition en cliquant sur le bouton OK. Pour rouvrir la fenêtre d'édition, il faut cliquer sur le bouton droit de la souris sur la bande supérieure (bande verte) de la fenêtre de saisie.

Exemple 29: Saisie de données à l'aide de l'option «Input» du sélectionneur d'objets

Détails: On construit un objet (fenêtre) du type «Input» à l'aide du menu déroulant du sélectionneur d'objets et les données sont saisies dans cette fenêtre.

Activités préparatoires: Dans le sélectionneur d'objets, sélectionnez l'option «Input» comme fenêtre de saisie, puis insérez cette fenêtre dans l'interface. Une fenêtre d'édition s'ouvre automatiquement et dans cette fenêtre, à la droite de «Global variable» (variable globale), écrivez «Nombre_de_pas» et choisissez l'option «Number» (Nombre) dans le champ intitulé «Type» (Figure 6.3). Pour finir, cliquez sur le bouton «OK».

Figure 6.3 : Fenêtre d'édition

Une petite fenêtre verte apparait alors dans l'interface. Entrez un nombre (par exemple 10) dans cette fenêtre (Figure 6.4).

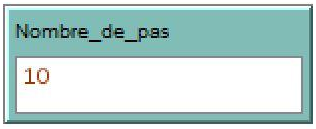


Figure 6.4 : Fenêtre de saisie d'information (nombre de pas)

Écrivons alors une petite procédure dans la zone d'édition de code, procédure dans laquelle la variable numérique «Nombre_de_pas» définie dans la fenêtre verte de saisie est utilisée:

```
to marcher
clear-all
crt 5
ask turtles [fd
Nombre_de_pas]
end
```

Quand la procédure «marcher» est appelée à partir de la fenêtre de l'observateur, cinq tortues sont créées, lesquelles avancent de 10 pas (Figure 6.5).

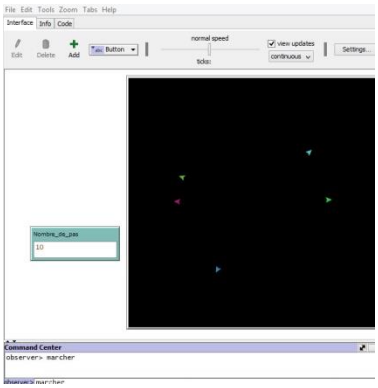


Figure 6.5 : État de l'interface après l'appel de la procédure «marcher»

3. Fenêtre «user-input» (fenêtre de saisie par l'utilisateur). Aussitôt que la primitive «user-input» est invoquée dans le centre de commande, une fenêtre apparaît dans l'interface, fenêtre dans laquelle l'utilisateur peut entrer des données. La fenêtre doit avoir un nom et un contenu. Les données entrées dans la fenêtre (le contenu) se comportent comme une variable locale et sont du type chaîne (string). Cependant, il est possible de saisir d'autres types de variables si on leur applique préalablement la primitive «read-from-string», laquelle change le type de la variable selon les souhaits l'utilisateur. L'invocation de la primitive dans le centre de commande se fait en écrivant *show user-input "Nom"* à la droite du mot observer>, où "Nom" est le nom de la fenêtre. Quand la fenêtre apparaît, on entre les données désirées et l'action est validée en appuyant sur le bouton OK. Ainsi, dans la

figure ci-dessous (Figure 6.6), le nom de la fenêtre est «Avertissement» et les données (entrées par l'utilisateur) sont «Parquet glissant»

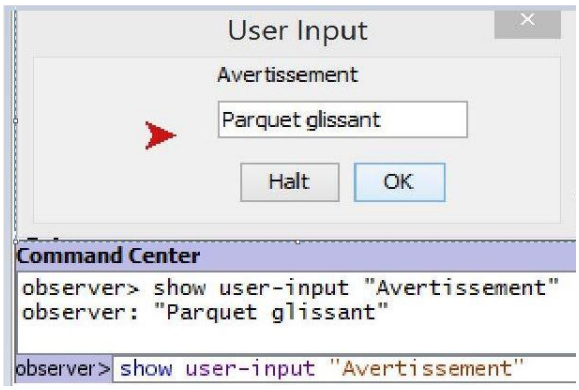


Figure 6.6 : Fenêtre de saisie par l'utilisateur

4. Lecture des données d'un fichier texte. Pour ce faire, l'on doit utiliser les primitives «file-read», «file-read-line», «file-read-characters n», «file-at-end?». La chaîne est lue dans le fichier comme si nous l'avions écrite dans le centre de commande. Il est nécessaire d'ouvrir le fichier au préalable à l'aide de la commande «file-open».

Saisie de données avec la primitive «user-input»

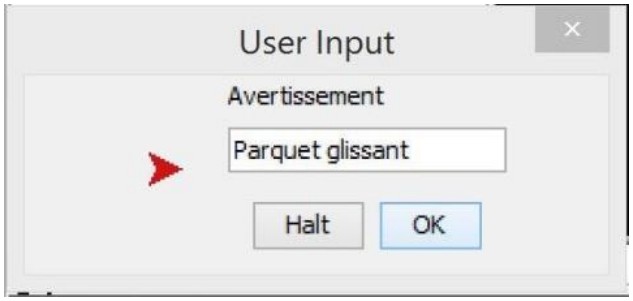
Primitives: user-input, read-from-string Autres détails: les données sont saisies dans la fenêtre qui émerge de la commande utilisant la primitive «user-input».

Les deux procédures «faire-attention» (exemple 30 a.) et «marcher» (exemple 30 b.) illustrent l'utilisation de la primitive «user-input» avec des variables de deux types différents: texte et nombre.

Exemple 30.a : procédure «faire attention»

```
to faire-attention
clear-all
show user-input“Avertissement”
end
```

Effet: le texte saisi dans la fenêtre «Avertissement» est envoyé au terminal d'instructions. Dans cet exemple, le texte saisi est «Parquet glissant»



Exemple 30.b: procedure «marcher»

to marcher
clear-all
crt 5
ask turtles [fd read-from-string user-input
“Nombre_de_pas”]
end

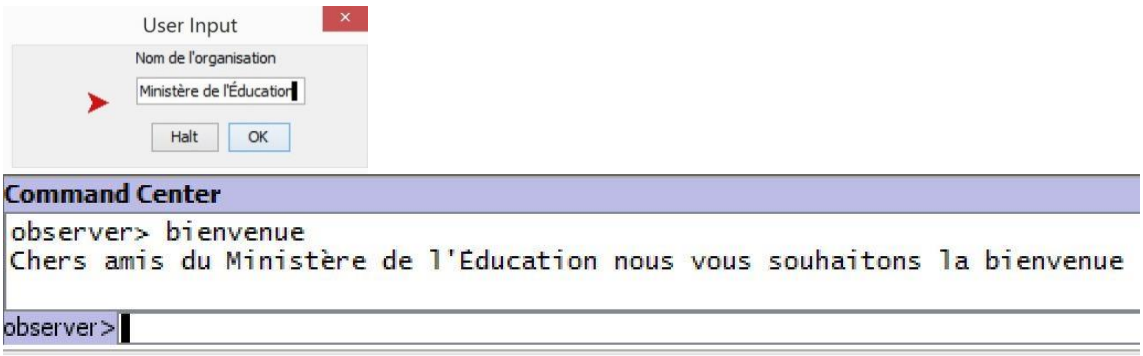
Effet: La première tortue avance du nombre de pas entré dans la fenêtre contextuelle «Nombre_de_pas», qui reste ouverte pour recevoir le nombre de pas de la deuxième tortue et ainsi de suite jusqu'à la dernière tortue. Si l’on souhaite interrompre ce processus, il suffit d’appuyer sur le bouton «Halt». La primitive «read-from-string» convertit la chaîne initiale qui a été entrée en un type numérique.

Exemple 30.c : procédure «bienvenue»

Pour conserver le contenu de la fenêtre «user-input» pour une quelconque action ultérieure, on peut utiliser une variable locale ou une variable globale. Dans l'exemple ci-dessous nous allons l'enregistrer comme variable locale appelée «institution»:

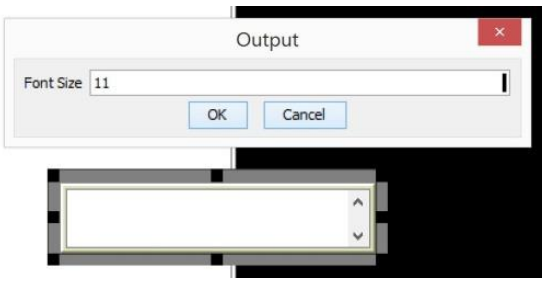
```
to bienvenue
clear-all
let institution user-input "Nom de l'organisation"
type "Chers amis du "  type institution  type " nous vous souhaitons la bienvenue"
end
```

Ici, le nom de la fenêtre est «Nom de l'organisation» et son contenu est stocké dans la variable locale appelée «institution». Si, par exemple, dans la fenêtre «user-input», on saisit le texte «Ministère de l'éducation», le terminal d'instruction affiche le résultat suivant: «Chers amis du Ministère de l'éducation, nous vous souhaitons la bienvenue.» Notez que l'espace blanc après le mot «institution» est nécessaire pour séparer les mots.



Diriger la sortie vers d'autres zones

En ce qui concerne la sortie des données, deux options ont, jusqu'à présent, été présentées: 1) le terminal d'instruction et 2) un fichier texte brut. Nous allons ajouter une troisième option qui consiste à diriger la sortie vers un objet de type «Fenêtre de sortie». Pour lancer cette option, la première chose à faire est de placer la fenêtre dans l'interface, en suivant la même méthode utilisée pour placer des boutons: on sélectionne d'abord l'option «Output» dans le menu déroulant du sélectionneur d'objets puis on place l'objet «Output» dans l'interface. Une fois cet objet placé, une seconde fenêtre apparaît dans l'interface, demandant à l'utilisateur de saisir la taille de police de son choix pour le texte de sortie (la valeur par défaut est 11 points).



La fenêtre de sortie peut être agrandie ou déplacée en utilisant les petits carrés noir situés sur les 4 côtés de la fenêtre. Pour diriger les outputs vers la fenêtre de sortie on peut employer l'une quelconque des primitives de sortie «output-show», «output-write», «output-type» ou «output-print».

Exemple 31: Envoyer les résultats à un objet du type «fenêtre de sortie»

Primitives: output-write, one-of, let, while (tant que).
Autres détails: On utilise la primitive «while» pour exécuter des itérations et on construit une fenêtre de sortie des résultats (choisir «output» dans le menu déroulant du sélectionneur d'objets intitulé «Button»).

Des mots aléatoires sont générés à partir de la phrase ["Aujourd'hui" "c'est" "mon" "jour" "de" "chance"] et lorsque l'exemple est exécuté en appelant la procédure «setup» à partir de la fenêtre de l'observateur, les mots choisis aléatoirement apparaissent dans la fenêtre de sortie.

Activités préparatoires: Construire une fenêtre de sortie de résultats.

Voici le code:

```
to setup
  clear-all
  reset-ticks
écrire
end

to écrire
let message ["Aujourd'hui" "c'est" "mon" "jour" "de" "chance"]
while [random 6 != 3]   [output-write one-of message tick show
ticks]
end
```

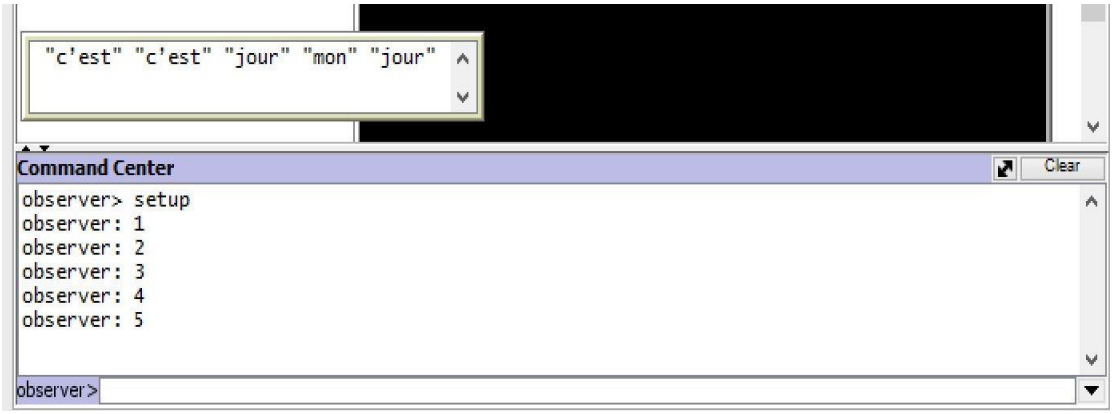


Fig 6.7 : Fenêtre de sortie des résultats

Explications et commentaires supplémentaires. Dans cet exemple, dans la procédure «écrire», nous introduisons la primitive «while» (tant que). Cette primitive permet de répéter des blocs de commandes. Son format est *while [condition] [bloc de commandes].* Tant que la condition est remplie, le bloc de commandes est exécuté. Lorsque la condition cesse d'être respectée, le bloc de commandes est ignoré et l'interpréteur continue à exécuter les commandes qui suivent (dans cet exemple l'exécution du programme s'arrête car il n'y a pas de commande qui suit). Dans l'exemple ci-dessus, la condition «while» consiste à prendre au hasard un entier dans l'ensemble (0, 1, 2, 3, 4, 5) et à vérifier que cet entier est différent de 3. Lorsque c'est le cas, l'un des mots de la liste «message» est choisi au hasard («one-of» c'est-à-dire «un-parmi») et envoyé à la fenêtre de sortie. Lorsque le nombre entier est 3, la procédure se termine. La variable «ticks» a été incluse pour compter le nombre de fois que la routine «while» est exécutée. L'instruction «tick» accroit d'une unité la valeur de la variable ticks et *show ticks* envoie cette valeur à la fenêtre de l'observateur.

Expressions «ask» encapsulées

Dans la programmation multiagents, il est courant de trouver des situations dans lesquelles un agent demande à un autre agent de faire quelque chose, voire des situations dans lesquelles un agent demande à un autre agent de demander à un autre agent de faire quelque chose. Dans un programme, ces situations sont généralement représentées par le biais de commandes composées de plusieurs niveaux de primitives «ask» encapsulées. La plupart du temps, l'identité (le numéro «who») des agents qui participent aux commandes n'est pas connue, car ces commandes sont adressées à des ensemble-agents composés de plus d'un agent. L'exemple qui suit (exemple 32) illustre le cas de deux commandes «ask» encapsulées dans une procédure qui construit un graphe aléatoire avec 5 nœuds.

Un graphe ou un réseau consiste en un ensemble de points appelés «nœuds», dont certains se connectent les uns aux autres au moyen de lignes appelées «arêtes» (cas des graphes non orientés) ou «arcs» (cas des graphes orientés). Nous pouvons facilement représenter les nœuds dans NetLogo en utilisant des tortues en tant que nœuds et des liens en tant qu'arêtes ou arcs. Grâce à la technologie informatique les réseaux sont devenus, ces dernières années, un sujet central d'étude et de recherche en sciences économiques et sociales [11], ainsi qu'en ingénierie, en mathématiques pures [7] et en mathématiques appliquées. NetLogo est particulièrement adapté à la création de modèles basés sur le réseau et il existe une extension dédiée à ce sujet (*NetLogo Nw extension*)[39].

Exemple 32 : Un graphe aléatoire I

Dans l'exemple ci-dessous un graphe aléatoire est construit au moyen d'une commande utilisant l'encapsulation. Le graphe résultant de l'exécution du code est illustré par la figure 6.8.

```
to graphe-1
clear-all
crt 5 [setxy random-xcor random-ycor]
ask turtles [set size 2]
ask turtle 4 [set color white]
ask turtle 3 [set color blue]
ask turtle 2 [set color red]
ask turtle 1 [set color yellow]
ask turtle 0 [set color green]
ask turtles [ask one-of other turtles [create-link-to
myself]]
end
```

[39] Les informations concernant cette extension peuvent être consultées sur le site NetLogo à : https://ccl.northwestern.edu/netlogo/docs/nw.html

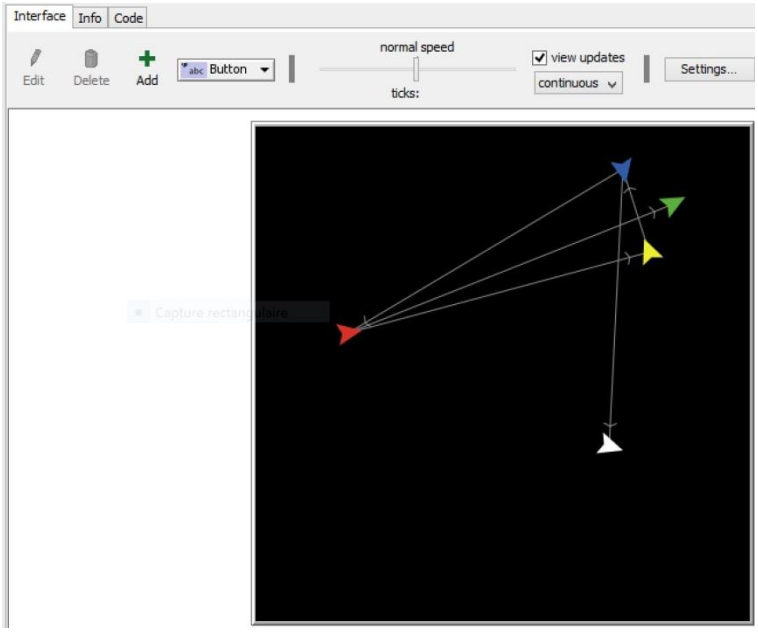


Figure 6.8 : Graphe aléatoire

L'encapsulation est courante dans le langage ordinaire. Si l'identité des personnes impliquées dans une phrase encapsulée est connue, il y a moins de risques que la phrase soit source de confusion. Par exemple: «Dis à Patrick de dire à Christèle de demander à Georges s'il va inviter Marielle à la fête». Une commande NetLogo qui contient trois niveaux «ask» emboités et dans laquelle l'identité des agents à qui il est demandé d'agir, ressemble à ceci:

ask turtle 0 [ask turtle 1 [ask turtle 2 [show distance turtle 0] create-link-with turtle 0]]

qui peut se traduire ainsi :

la tortue 0 demande à la tortue 1

a. de demander à la tortue 2 d'indiquer la distance qui la («la» étant la tortue 2 elle-même) sépare de la tortue 0 et
b. de créer un lien entre elle-même (la tortue 1) et la tortue 0

Le code correspondant à l'exemple ci-dessus est le
suivant :

to graphe-2
clear-all
crt 5 [setxy random-xcor random-ycor]
ask turtles [set size 2]
ask turtle 4 [set color white]
ask turtle 3 [set color blue]

```
ask turtle 2 [set color red]
ask turtle 1 [set color yellow]
ask turtle 0 [set color green]
ask turtle 0 [ask turtle 1 [ask turtle 2 [show distance turtle 0] create-link-with turtle 0]]
end
```

La figure ci-dessous (Figure 6.9) permet de constater que la tortue 2 (rouge) a bien indiqué la distance (11.0387) qui la sépare de la tortue 0 (verte) et que la tortue 1 (jaune) a bien créé un lien entre elle-même et la tortue 0 (verte)

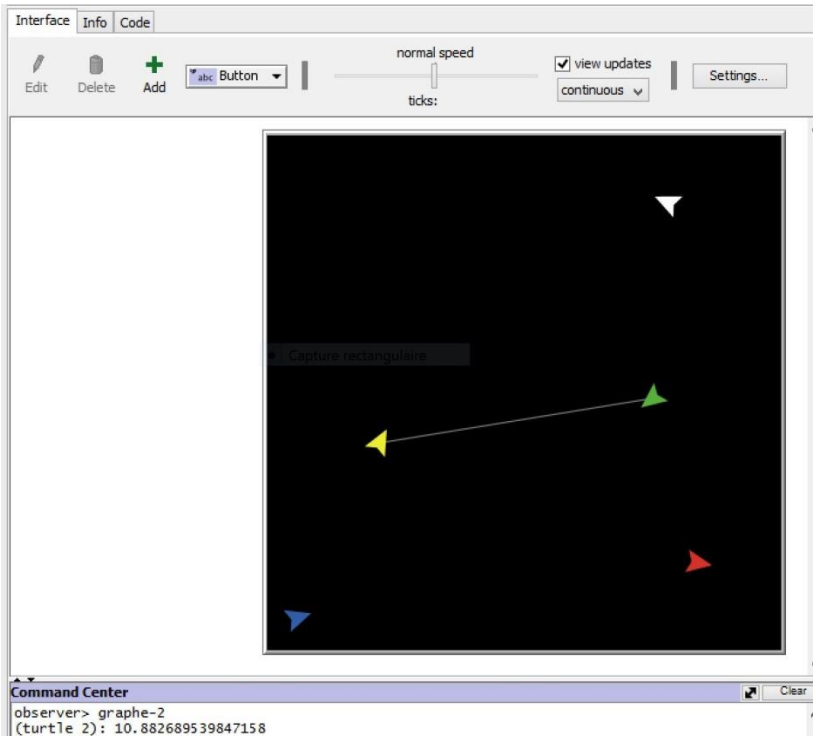


Figure 6.9 : Création de liens au moyen de l'encapsulation

La procédure ci-dessous ajoute un lien (Figure 6.10) entre la tortue 0 (verte) et la tortue 2 (rouge) car la tortue 0 a ajouté à sa requête initiale à la tortue 1 (*en italique* dans le code ci-dessous) une autre requête, à savoir créer un lien avec la tortue 2 (en **caractères gras soulignés** dans le code ci-dessous) :

```
to graphe-3
clear-all
crt 5 [setxy random-xcor random-ycor]
ask turtles [set size 2]
ask turtle 4 [set color white]
ask turtle 3 [set color blue]
```

ask turtle 2 [set color red]
ask turtle 1 [set color yellow]
ask turtle 0 [set color green]
ask turtle 0 [**ask turtle 1** *[ask turtle 2 [show distance turtle 0] create-link-with turtle 0]* **create-link-with turtle 2**]
end

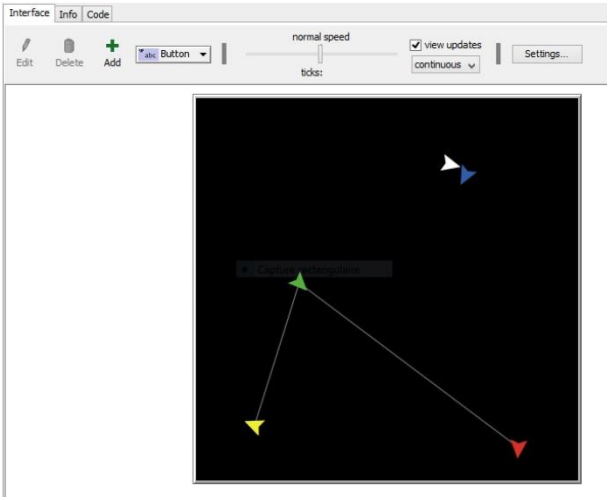


Figure 6.10 : Ajout d'un lien supplémentaire au moyen de l'encapsulation

Exercice : modifier la procédure précédente de manière à créer un lien entre la tortue 1 (jaune) et la tortue 2 (rouge).

L'exemple qui suit (exemple 33) illustre une encapsulation à quatre niveaux, dans laquelle des liens sont établis entre deux paires de tortues dont l'identité (leur numéro d'identification) est connue. Des couleurs distinctes sont attribuées aux tortues afin de mieux vérifier leur identité.

Exemple 33: Encapsulations avec des agents dont l'identité est connue

to liaisons1
clear-all
crt 4 [fd 10]
ask turtle 0 [set color white ask turtle 1
[set color yellow ask turtle 2
[set color red ask turtle 3 [set color blue
create-link-with turtle 2]]
create-link-with turtle 0]]
end

La primitive «myself» peut être utilisée pour faire référence à un agent dont l'identité est connue, comme dans l'exemple ci-dessous dont le code est équivalent à celui de l'exemple précédent (Figure 6.11).

to liaisons2
clear-all
crt 4 [fd 10]
ask turtle 0 [set color white ask turtle 1 [set color yellow ask turtle 2
[set color red ask turtle 3 [set color blue create-link-with myself]]
create-link-with myself]]
end

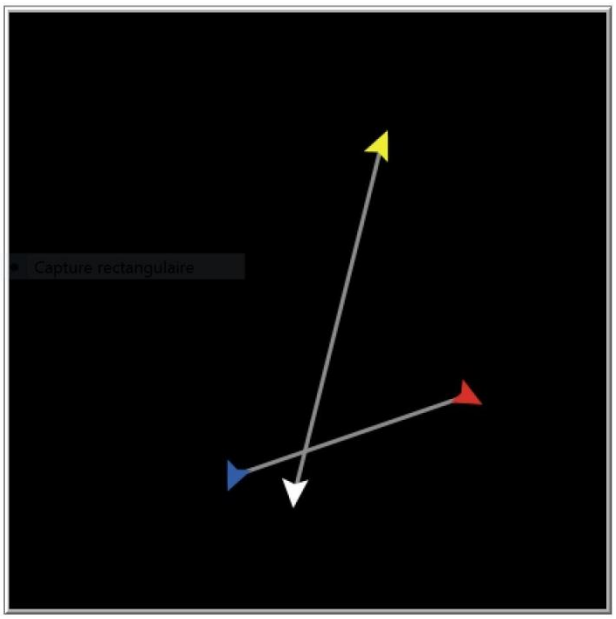


Figure 6.11 : Résultats identiques des procédures liaison1 et liaison2

Dans les deux versions (liaisons1 et liaisons2) de l'exemple 33, des liens sont construits entre les tortues 0 (blanche) et 1 (jaune) et entre les tortues 2 (rouge) et 3 (bleue). L'ordre dans lequel apparaissent les crochets régit les relations entre les agents, de la même manière que le font les parenthèses le dans les formules mathématiques.

Il existe cependant des situations où l'on doit donner des ordres sans connaître l'identité des personnes, par exemple lorsque le propriétaire d'une entreprise dit à son employé: «*Dites à chaque client qui arrive au magasin d'appeler l'un de ses proches afin que ce dernier demande à l'un de ses voisins de former une équipe avec lui, pour que les deux puissent participer au concours que nous organisons dans le magasin*».

Dans cet ordre «encapsulé», on ignore qui est le client ou qui est le proche parent et encore moins qui est le voisin du parent. L'efficacité de cet ordre réside précisément dans l'anonymat de ces personnes, car c'est ce qui permet de l'appliquer à de nombreuses personnes dont on ignore l'identité. Leurs identités sont connues au fur et à mesure de l'exécution de l'ordre, c'est à-dire au fur et à mesure que les clients se présentent au magasin. À la fin de la journée, les identités seront connues et un registre pourrait même être préparé: la cliente Marthe a appelé son cousin Ernest qui a demandé à sa voisine Raymonde de faire équipe avec lui. Puis est arrivé le client Fernand qui a appelé sa sœur Ruth, etc. Dans la construction du modèle, il est important de pouvoir créer des commandes avec une encapsulation dans laquelle les agents sont anonymes. NetLogo peut prendre en charge plusieurs niveaux d'encapsulation (en principe un nombre illimité). L'exemple 34 (section suivante) contient une commande avec trois niveaux d'encapsulation, commande dans laquelle les agents sont anonymes. Le résultat est, de nouveau, un graphe aléatoire.

Exemple 34: Graphe aléatoire avec encapsulation et agents anonymes

```
to graphe
clear-all
crt 5 [set size 2 fd 7]
ask turtles [ask other turtles [ask one-of other turtles [create-link-to
myself ]]]
end
```

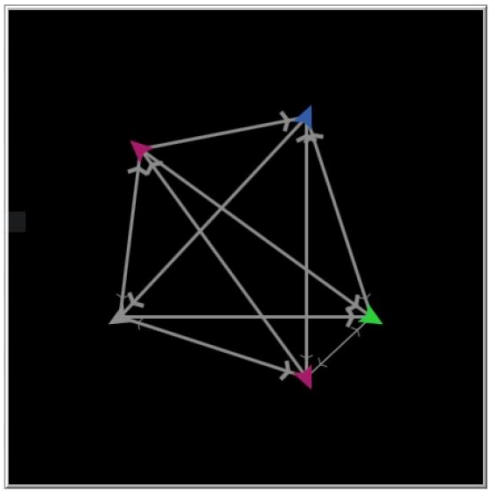


Figure 6.12 : Graphe aléatoire avec encapsulation et agents anonymes

La figure précédente (Figure 6.12) montre le résultat de: «demander aux tortues de demander à chacune des autres tortues de demander à l'une des autres tortues de créer un lien avec elles».

S'il est vrai que de nombreuses parenthèses dans les langages de programmation ne trompent pas l'interpréteur, il est plus difficile pour nous, êtres humains, de manipuler et d'interpréter les expressions comportant de nombreuses parenthèses encapsulées ainsi que des agents anonymes. Dans NetLogo, comme dans d'autres langages de programmation, on peut contrôler les relations entre agents dans les expressions de ce type en utilisant des variables locales qui jouent un rôle semblable à celui des «jokers» de certains jeux de cartes[40] pour attribuer des noms temporaires à certains agents anonymes, comme dans l'exemple qui suit (Exemple 35).

Exemple 35: Un graphe aléatoire utilisant des variables génériques

```
to setup
clear-all
crt 20 [setxy random-xcor random-ycor]
go
end

to go
ask turtles [let Mikey one-of other
turtles
ask Mikey [create-link-to myself]]
end
```

Le nom Mikey[41]est utilisé pour chaque tortue, puis abandonné pour être utilisé par la tortue suivante dans la commande «ask turtles». Chacune des 20 tortues de l'ensemble-agents («agentset») se nomme Mikey à un moment donné, mais il n'y a jamais deux tortues qui utilisent ce nom en même temps. Une variable générique peut également être utilisée comme alias d'un ensemble-agents. Si nous appelions «amies» l'ensemble-agents des «autres tortues», il serait parfaitement licite d'écrire:

```
ask turtles [let other turtles amies ask amies [create-link-to myself]].
```

[40] Dans certaines parties des cartes, le «joker» est une carte qui peut prendre la valeur que le joueur qui la possède lui donne.
[41] Mikey est le surnom de Michelangelo, la plus célèbre des tortues Ninja, caractérisée par le port d'un bandana orange et son goût marqué pour la pizza.

Primitives opérant sur des listes par le biais d'autres primitives

NetLogo a plusieurs primitives qui effectuent des opérations sur les listes. Nous en présenterons quatre ici: «map», «foreach», «filter» et «reduce». Dans cette première présentation, la discussion portera sur l'interaction entre ces primitives et les listes utilisant d'autres primitives qui, dans certains cas, sont des acteurs et, dans d'autres, des rapporteurs. Dans les deux sections qui suivent nous aborderons le cas où les actions ou le rapport sont le produit de procédures de type anonyme. A titre d'explication préliminaire, notez que les primitives: map (mapper), foreach (pour chaque), réduire (réduire), filter (filtrer), n-values (n-valeurs) et sort-by (réordonner-selon) fonctionnent avec deux arguments, l'un étant une liste et l'autre une procédure qui s'applique à chaque élément de la liste ou à des listes données. Dans cette section, la procédure sera fournie par une primitive.

map.
Format: map reporter listes.
C'est une primitive du type «rapporteur» qui peut fonctionner avec une ou plusieurs listes.

show map abs [2 -4 -5 1]
==> observer: [2 4 5 1]
«map» a été utilisé pour mapper la primitive «abs» (valeur absolue) sur une liste de nombres.

show map is-number? [1 "banana" 3]
==> observer: true false true, dans ce cas on a mappé le rapporteur «is-number?» sur une liste. Evidemment "banane" n'est pas un chiffre.

Si on l'utilise avec plusieurs listes, ces dernières doivent avoir la même taille :
show (map * [1 2 3][4 5 6])
==> observer: [4 10 18], Comme l'opération * fonctionne avec deux entrées, deux listes sont nécessaires pour pouvoir prendre un élément de chaque liste à la fois: 1 * 4 = 4, 2 * 5 = 10, 3 * 6 = 18. Le nombre de listes doit correspondre au nombre d'entrées (arguments) de la primitive ou de la procédure rapporteur.

foreach
Format: foreach liste commande.
Appliquer la commande à chaque ítem de la liste

foreach [1 2 3] write

==>1 2 3, la tâche est donnée par la primitive «write», laquelle s'applique à chacun des membres de la liste [1 2 3].

filter
Format: filter condition liste.
Filtre une liste rapportant uniquement les membres qui remplissent la condition donnée.

show filter is-number? [1 "2" 3]
==> observer: [1 3], la commande affiche la liste dont les membres remplissent la condition d'être des numéros

reduce
Format: reduce opérateur de rapportage liste.
Cette primitive réduit les éléments d'une liste en fonction de l'opération de rapportage fournie en entrée.

show reduce + [1 2 3 4]
==> observer: 10

L'opérateur indiqué devant la liste représente une opération connue (une addition dans cet exemple) et, à chaque étape de la procédure, la liste à laquelle l'opérateur est appliqué est séparée en deux éléments: le premier membre de la liste (appelé tête de liste) et le reste de la liste (appelée la queue de la liste), selon le schéma suivant :

Opérateur [a b c] = a opération (opérateur [b c]) = a (opération (b opération (opérateur [c])))

Explication détaillé de la procédure utilisée dans l'exemple

show reduce + [1 2 3 4]
==> observer: 10

L'opération indiquée par le symbole **+** devant la liste (dans ce cas la somme) s'applique au premier élément de la liste et au résultat de l'application du même symbole + à la liste restante.

Il convient de noter que le signe **+** (intentionnellement en gras) devant les crochets représente l'opération «réduire ou convertir en somme ordinaire»[42]. Nous représentons cette dernière au moyen du signe + (caractère ordinaire). Pour comprendre le processus, il faut tenir compte du fait que la réduction **+**

[42] Notez que, bien qu'au bout du compte, l'opérateur + résulte en une somme ordinaire cette opération ne représente pas une somme ordinaire entre des nombres puisque la somme ordinaire ne peut pas être appliquée à deux objets, dont l'un est une liste.

appliquée à un nombre doit avoir une signification précise, dans ce cas cela signifie ajouter 0 au nombre, par exemple : **+** [5] = 5 + 0 = 5.

Revenons à notre exemple :

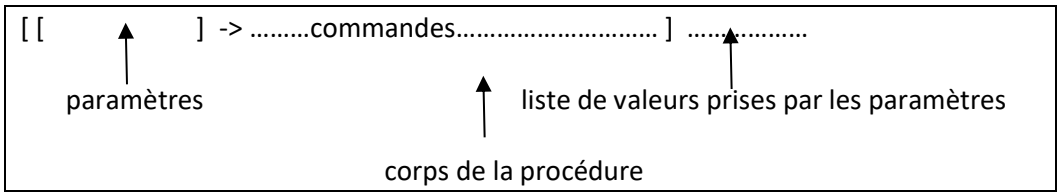
+ [1 2 3 4] = 1 + (**+** [2 3 4]) = 1 + (2 + (**+** [3 4]) = 1 + (2 + (3 **+** [4])) = 1 + (2 + (3 + (4 + 0))) =
1 + (2 + (3 + 4)) = 1 + (2 + 7) = 1 + 9 = 10.

Lorsque combinées aux procédures dites anonymes, les primitives précédentes sont de puissants moyens d'expression.

Procédures anonymes

À l'instar de plusieurs autres langages de programmation, NetLogo supporte des procédures dites anonymes. Une procédure anonyme, comme son nom l'indique, est une procédure qui admet des paramètres et possède un corps, comme toute autre procédure, mais qui n'a pas de nom ou qui n'a pas été définie dans l'éditeur de code. Les procédures anonymes existent dans d'autres langages de programmation tels que Lisp et Scheme et sont connues sous le nom d'*expressions lambda*, en l'honneur du logicien américain Alonzo Church, inventeur du calcul Lambda. La première question qui se pose est la suivante: pourquoi voudrions-nous d'une procédure présentant ces caractéristiques?

Les procédures anonymes sont généralement utilisées dans des modèles ou des programmes d'un certain degré de complexité, où certaines tâches ne sont effectuées qu'une seule fois et pour lesquelles cela ne vaut pas la peine de les stocker sous un nom et ceci afin d'économiser de l'espace mémoire[43]. Une procédure anonyme se compose de trois parties. La première partie est une liste qui contient les paramètres nécessaires à la procédure, la deuxième est l'ensemble des commandes à appliquer aux paramètres tandis que la troisième est la liste des valeurs de paramètres auxquelles la procédure doit être appliquée. Voici le format:



[43] Ces procédures sont également appelées «expressions lambda» car elles sont inspirées du calcul Lambda dont il a été question dans le texte. Certaines procédures des langages du type dit fonctionnel, tels que Lisp et Scheme, sont des applications concrètes du calcul Lambda.

Après la liste des paramètres, on doit écrire le symbole «->» (signe «moins» suivi du signe «plus grand que»).

Comme premier exemple, nous choisirons la primitive «run» (exécuter). C'est la primitive qui offre la plus grande polyvalence pour être combinée avec des procédures anonymes.

(run [[x] -> write x] " J'aime ça ")
==> "J'aime ça"

La chaîne «J'aime ça» est la valeur substituée dans la variable ou le paramètre «x» dans la procédure anonyme [[x] -> write x], laquelle signifie «je suis une procédure qui fait ce qui suit: tout ce qui est inscrit à la place de x, je promets de l'écrire lors de l'exécution de la procédure» (d'où la nécessité d'inclure run).

Prenons un exemple avec deux paramètres:

(run [[nombre-tortues nombre-pas] -> crt nombre-tortues ask turtles [fd nombre-pas]] 10 5)

La primitive «run» demande l'éxécution de la procédure anonyme:
[[nombre-tortues nombre-pas] -> crt nombre-tortues ask turtles [fd nombre-pas]]
appliquée aux valeurs 10 et 5. Cela entraîne la création de 10 tortues à qui l'on demande d'avancer de 5 pas, mais cette procédure attend que la primitive «run» l'exécute.

La primitive «run» est une primitive «active» (qui donne des ordres). Quand on veut plutôt rapporter un résultat on utilise la primitive «runresult».

show (runresult [[a b] -> a + b] 10 5)
==> observer: 15

Ici, il ne serait pas approprié d'utiliser «run». La commande:

show (run [a b] -> a + b] 10 5)

engendre une erreur car «run» n'est pas une primitive de type «reporter» (rapporteur).

Primitives opérant sur des listes par le biais de procédures anonymes.

Les primitives qui agissent sur des listes peuvent également utiliser des procédures anonymes comme entrées. Voyons quelques exemples.

show map [[x] -> 3 * x] [6 10]
==> observer: [18 30]

Ici, on mappe la procédure anonyme [[x] -> 3 * x] sur la liste [6 10] .
La procédure ordonne de tripler chaque x de la liste [6 10].

show (map [[a b c] -> a + b = c] [1 2 3] [2 4 6] [3 5 9])
==> observer: [True False True]

Ici, nous avons trois listes et les éléments sont mappés trois par trois, un de chaque liste à la fois. La primitive «map» produit True lorsqu'elle satisfait à l'égalité a + b = c, comme c'est le cas pour les premier et troisième éléments de chaque liste. En effet 1 + 2 = 3 et 3 + 6 = 9, alors que 2 + 4 n'est pas égal à 5.

show filter [i -> i < 3] [1 3 2]
==> observer: [1 2], traduction: filtrer les éléments i de la liste [1 3 2] tels que i < 3.

show (filter [[s] -> first s > 3] [[20 3 4][5 1 6] [2 11 15]])
observer: [[20 3 4] [5 1 6]], Affiche les listes s, dont le premier élément est supérieur à 3.

foreach [1 4 9 16] [[x] -> write sqrt (x)]
==> 1 2 3 4, pour chaque nombre de la liste, écrivez sa racine carrée.

foreach ["une" "grande" "échevine""élue"][[x]-> write but-last x]
==> "un" "grand" "échevin""élu"

On demande d'écrire chaque mot de la liste en supprimant sa dernière lettre.

Un exemple avec deux listes extrait du dictionnaire NetLogo (on suppose que trois tortues ont préalablement été créées) :

(foreach list (turtle 1) (turtle 2) [3 4]
[[la-tortue nombre-pas] -> ask la-tortue [fd nombre-pas]])

La tortue 1 avance de 3 pas et la tortue 2 avance de 4 pas.

La figure ci-dessous (Figure 6.13) récapitule les exemples de primitives opérant sur des listes via des procédures anonymes présentés dans cette section.

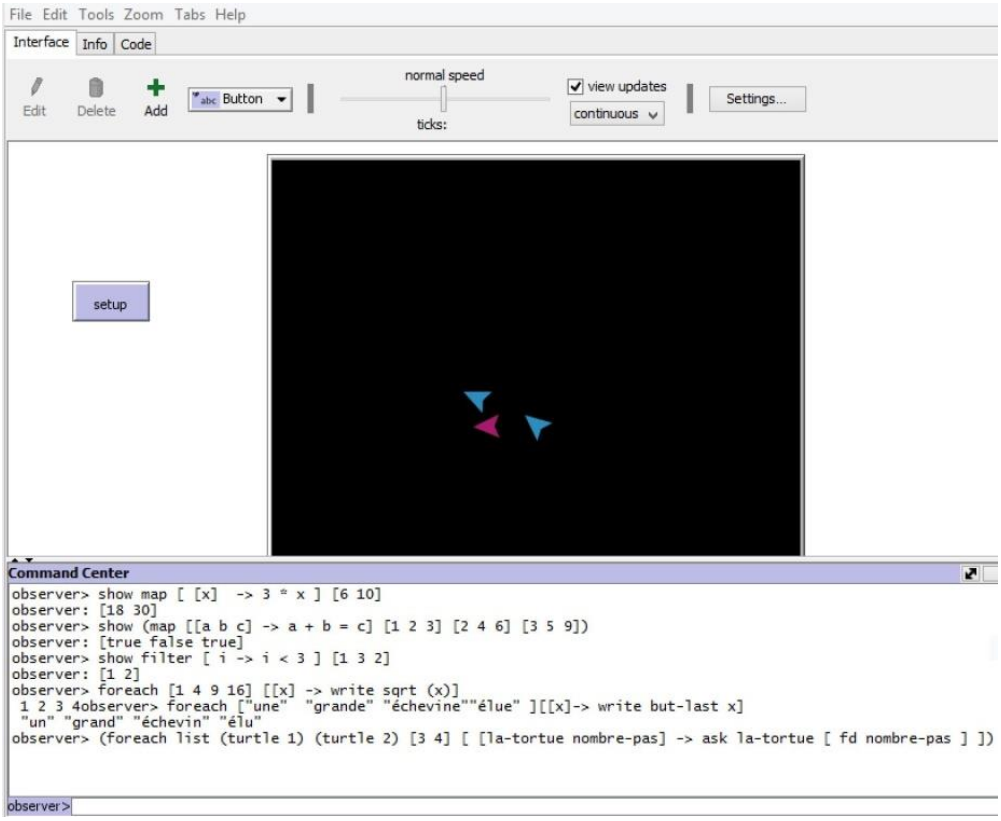


Figure 6.13 : Résultats récapitulatifs

Notez que toutes les primitives fonctionnant avec des listes n'utilisent pas la même syntaxe. Par exemple, dans la primitive «foreach», la liste à laquelle s'applique la procédure anonyme n'est pas écrite après la procédure mais avant. Il est à noter que la syntaxe de ces primitives a varié. On ne peut plus utiliser le caractère «?» (ou à défaut n'importe quel autre caractère) comme caractère générique, à la manière des exemples utilisés dans le dictionnaire Netlogo. Ces exemples ne fonctionnent plus. Pour plus d'informations sur les procédures anonymes, nous vous recommandons de consulter la section «Anonymous procedures » (Procédures anonymes) dans le Guide de programmation de NetLogo.

L'exportation et l'importation de données dans NetLogo

Comme nous l'avons vu, l'environnement NetLogo est composé de plusieurs éléments ou composantes. Parfois, il est souhaitable d'exporter une composante du modèle pour l'intégrer dans une autre application ou d'importer certaines des composantes précédemment exportées d'un modèle. Il est également possible d'importer certains formats de fichiers graphiques

pour les intégrer dans un modèle NetLogo. Les actions d'exportation ou d'importation sont activées à partir du menu «File» (Fichier), en utilisant les options «Export» (Exporter) ou «Import» (Importer). (Figure 6.14). Pour chaque option, une fenêtre s'ouvre affichant les éléments éligibles possibles. Plusieurs composantes peuvent être exportées en cliquant sur le bouton droit de la souris sur leurs zones ou fenêtres respectives ou à partir du code via les commandes.

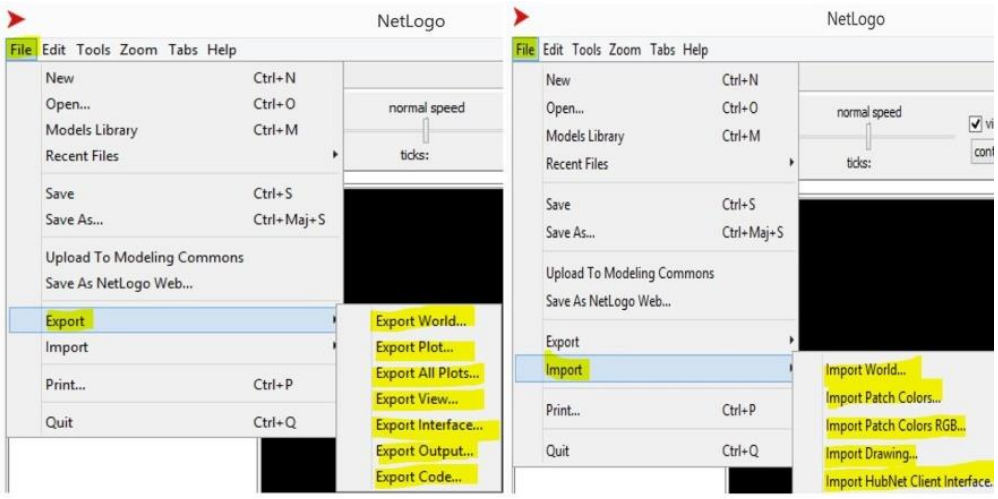


Figure 6.14 : Interfaces d'exportation et importation de données

Composantes exportables

Dans le menu «File», NetLogo offre les sept options d'exportation suivantes :

1. Exporter le monde Cette option enregistre dans un fichier de données les valeurs des variables, l'état des tortues et des parcelles, la couche de dessin et les graphiques, y compris certains paramètres définissant la configuration de l'environnement. L'exportation peut également être effectuée à partir d'un code indiquant le nom du fichier vers lequel l'on désire exporter, avec la commande *export-world «nom-de-fichier».*

2. Exporter un graphique Enregistre le graphique sélectionné dans un fichier PNG. Une fenêtre contextuelle permet de sélectionner le graphique que l'on souhaite exporter. L'exportation peut être effectuée en cliquant avec le bouton droit de la souris sur le graphique. L'exportation peut également être effectuée à partir d'un code indiquant le nom du fichier vers lequel l'on veut exporter, avec la commande *export-plot «nom-de-fichier»*.

3. Exporter tous les graphiques. Enregistre tous les graphiques dans un fichier. L'exportation peut également être effectuée à partir d'un code

indiquant le nom du fichier vers lequel l'on désire exporter, avec la commande *export-all-plots «nom-de-fichier»*.

4. Exporter la vue Enregistre l'état dans lequel se trouve la vue dans un fichier graphique PNG, c'est-à-dire exporte «une photo» de la place où se trouvent les parcelles et où habitent les tortues. L'exportation peut être effectuée en cliquant avec le bouton droit de la souris sur une partie de la vue. L'exportation peut également être effectuée à partir d'un code indiquant le nom du fichier vers lequel l'on souhaite exporter, avec la commande *export-view «nom-de-fichier».*

5. Exporter l'interface Enregistre l'état actuel de l'interface dans un fichier au format graphique PNG. La figure inclut non seulement la vue du monde, mais également des éléments supplémentaires tels que des boutons, des curseurs et des graphiques, mais n'inclut pas les barres de menus, la fenêtre de l'observateur ou le terminal d'instructions. L'exportation peut être effectuée en cliquant avec la souris sur une zone vide de l'interface. L'exportation peut également être effectuée à partir d'un code indiquant le nom du fichier vers lequel l'on veut exporter, avec la commande *export-interface «nombre-de-fichier».*

6. Exporter les outputs (résultats) Enregistre le contenu de la zone vers laquelle la sortie a été dirigée, que ce soit le terminal d'instructions ou une fenêtre de sortie. L'exportation peut également être effectuée à partir d'un code indiquant le nom du fichier vers lequel l'on souhaite exporter, avec la commande *export-output «nom-de-fichier».*

   Différence entre l'envoi de la sortie dans un fichier et l'exportation d'une composante d'un modèle.
   Lorsque l'output est envoyé dans un fichier à l'aide des primitives file-show, file-type, etc., le fichier doit avoir été créé et ouvert auparavant, sinon la commande ne sera pas exécutée. Ces opérations (création et ouverture préalables d'un fichier), ne sont plus nécessaires à partir de la version 6 de NetLogo. Lorsque des éléments d'un modèle sont exportés à l'aide de l'une des trois options disponibles (option «Export» du menu Fichier, bouton droit de la souris sur l'élément à exporter ou code avec des commandes de type export-élément «nom-de-fichier»), l'effet est équivalent à une action du type «enregistrer sous» (save as), qui entraine l'ouverture d'une fenêtre de manière à pouvoir donner un nom au fichier nouvellement créé qui contiendra les informations à exporter.

7. Exporter le code : Enregistre le code du modèle dans un fichier HTML en préservant les couleurs.

Composantes importables

Dans le menu Fichier, l'option «Import» (Importer) propose les cinq options suivantes:

1. Importer le monde («import-world»). Télécharge un fichier contenant toutes les données enregistrées lors d'une exportation existante du monde. L'action peut être effectuée à partir d'un code à l'aide de la commande *import-world «nom-de-fichier».*

2. Importer une image («import-drawing»). Télécharge une image sur la couche de dessin du monde en l'ajustant à sa taille. On a déjà expliqué que, bien que les tortues puissent créer et même effacer les traces qu'elles produisent sur cette couche, elles ne peuvent toutefois pas détecter de telles traces, ce qui empêche toute interaction entre les tortues et leurs traces. L'action peut être effectuée à partir d'un code en utilisant la commande *import-drawing «nom-de-fichier».*

   Pour les trois options restantes: 3. Importer les couleurs de parcelle («import-pcolors»), 4. Importer les couleurs de parcelle au format RGB («import-pcolors-rgb») et 5. Importer une interface client Hubnet , nous renvoyons les lecteurs au Manuel de l'utilisateur de NetLogo [16].

Chapitre 7: Itération et récursivité.

La répétition d'un processus

En informatique, comme dans la vie quotidienne, il est courant de devoir répéter plusieurs fois une opération ou un processus. Lorsque nous demandons à un traitement de texte de rechercher toutes les occurrences d'un mot donné, c'est par l'application répétée du même processus de recherche que le processeur trouve toutes les occurrences du mot recherché. Dans ce chapitre, nous examinerons les ressources dont dispose le langage NetLogo pour qu'un processus se répète. Pour ce faire NetLogo offre cinq façons de procéder:

1. Au moyen de la primitive «repeat»
2. Au moyen de la primitive «loop»
3. Au moyen de la primitive «while»
4. Au moyen du bouton «Forever»
5. Au moyen d'un schéma de récursivité

Cette liste d'options pourrait être réduite sans que le langage NetLogo ne perde de sa puissance d'expression. Il est courant que les langues offrent une certaine redondance, afin de satisfaire les goûts et le style de programmation des utilisateurs.

Il existe des primitives qui agissent sur les éléments d'une liste ou d'une chaîne, telles que «foreach», «map», «run» et «reduce». Cependant, comme nous le verrons plus tard, ces primitives n'ont pas pour effet de répéter les instructions contenues dans la liste, mais plutôt de faire en sorte que les membres de la liste ne soient traités qu'une seule fois d'une certaine manière.

Modèle 17: Une règle simple pour calculer l'aire d'un terrain polygonal[44]

Un polygone est une figure géométrique plane composée de points appelés sommets joints par des lignes droites appelés arêtes placées de sorte qu'elles forment un tracé fermé. Un polygone est dit simple s'il ne contient pas d'arêtes qui s'entrecroisent. Étant donné un polygone simple, la question suivante se pose: est-t-il possible de calculer sa superficie intérieure (son aire) si l'on ne connaît que les coordonnées rectangulaires de ses sommets?

[44] Cette règle (ou algorithme) de calcul est connue en anglais sous les noms de «shoelace algorithm» (algorithme du lacet de chaussure) ou encore «surveyor's rule» (règle de l'arpenteur).

En fait, c'est possible et la règle que nous présentons dans les lignes qui suivent est un exemple de phénomène intéressant et bien connu en mathématiques. Cette règle décrit la relation entre une intégrale curviligne le long d'une courbe fermée et l'intégrale double à l'intérieur de la région délimitée par ladite courbe (théorème de Green en calcul à variables multiples).

Cette règle exige seulement que le polygone soit simple, c'est-à-dire que ses lignes ne s'entrecroisent pas, mais elle n'exige pas que le polygone soit régulier ou même convexe[45]. Les sommets du polygone peuvent être situés dans n'importe lequel des quatre quadrants du plan cartésien. L'utilisateur doit faire attention à la direction dans laquelle il parcourt le périmètre. La valeur de la superficie est positive si l'on parcourt ses sommets dans le sens inverse des aiguilles d'une montre. Si l'on effectue le parcours dans l'autre sens, la valeur est négative (on prend alors sa valeur absolue pour exprimer cette superficie).

Voici la règle. Supposons que nous ayons un polygone simple dont les n sommets sont a,b,c…n. Les coordonnées rectangulaires de ces points sont notées a (x_1, y_1), b (x_2, y_2), c (x_3, y_3),…. n (x_n, y_n). Nous parcourons le périmètre du polygone en partant de n'importe lequel des n points et nous les identifions au fur et à mesure que nous les trouvons sur l'itinéraire. Une fois identifiés, nous plaçons les noms et coordonnées des points dans des colonnes distinctes en **répétant** le premier point (le point de départ a dans ce cas) en dernier (dernière ligne du Tableau 17.1).

Points	Abscisses X	Ordonnées Y
a	x_1	y_1
b	x_2	y_2
c	x_3	y_3
d	x_4	y_4
….	….	….
….	….	….
n	x_n	y_n
a	x_1	y_1

Tableau 17.1 : Tableau de calcul de l'aire d'un polygone

[45] Une figure est dite convexe si, étant donné deux points situés à l'intérieur de la figure, tous les points du segment de droite qui joint ces deux points sont également à l'intérieur de la figure.

Pour calculer l'aire A du polygone, il suffit de faire la somme des produits dans le sens nord-ouest sud-est (flèches ↘), de soustraire de cette somme la somme des produits dans le sens nord-est sud-ouest (flèches ↙) et de diviser le résultat P de cette soustraction par 2 soit :

$$P = x_1y_2 + x_2y_3 + x_3y_4 + \ldots + x_ny_1 \; - (y_1x_2 + y_2x_3 + y_3x_4 + \ldots + y_nx_1)$$

L'aire A du polygone est alors égale à P/2, la moitié de la somme ci-dessus

Par exemple, soit le polygone formé par les quatre points: a (-2, 3), b (2, 1), c (6, 6), d(1,3). La première chose à vérifier est que les lignes qui forment les arêtes du polygone ne s'entrecroisent pas. Pour cela, il suffit de représenter graphiquement le polygone dans un plan cartésien (Figure 17.1) en suivant un ordre identique (a,b,c,d) à celui dans lequel les points apparaissent dans la liste.

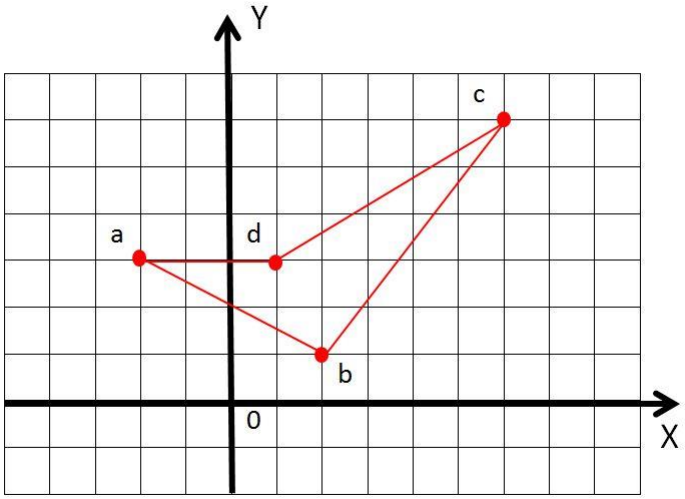


Figure 17.1 : Exemple de polygone simple (non convexe)

Il n'y a effectivement pas de lignes qui s'entrecroisent si l'ordre abcd (ou si l'ordre adcb est suivi). A partir de la figure, nous pouvons voir que les points sont les sommets d'un quadrilatère non convexe. Si nous parcourons les côtés du polygone dans l'ordre abcd (sens inverse des aiguilles d'une montre) et calculons son aire selon la méthode précédemment décrite, nous obtenons une valeur positive.

L'exemple suivant illustre les calculs requis pour estimer l'aire du polygone de la Figure 17.1.

Points	Abscisses X	Ordonnées Y
a	-2	3
b	2	1
c	6	6
d	1	3
a	-2	3

Tableau 17.2a

Points	Abscisses X	Ordonnées Y
A	-2	3
B	2	1
C	6	6
D	1	3
A	-2	3

Tableau 17.2b

Dans le tableau 17.2a on multiplie l'abscisse d'un sommet par l'ordonnée du suivant (en répétant le point de départ a à la fin du tableau) et on fait la somme des produits obtenus (-2)(1)+(2)(6)+(6)(3)+(1)(3) = 31. Dans le tableau 17.2b, on multiplie l'ordonnée d'un sommet par l'abscisse du suivant (toujours en répétant le point de départ a) et on fait également la somme des produits obtenus (3)(2)+(1)(6)+(6)(1)+(3)(-2) = 12. L'aire du polygone abcd est alors (31-12)/2=9,5

Pour programmer la règle, il est plus facile d'implémenter cet algorithme en transformant l'expression $x_1y_2 + x_2y_3 + x_3y_4 + ... + x_ny_1 - (y_1x_2 + y_2x_3 + y_3x_4 + ... + y_nx_1)$ en soustractions impliquant des paires de point consécutives, à partir de les deux premiers: $(x_1y_2 - y_1x_2) + (x_2y_3 - y_2x_3) + (x_3y_4 - y_3x_4) + ... + (x_ny_1 - y_nx_1)$. Cela permet de traiter l'expression de manière itérative en prenant les deux premiers points de la liste, puis en éliminant le premier et en répétant l'opération. Nous baserons le mécanisme d'itération sur la primitive «while» (il pourrait également être implémenté en utilisant le schéma de récursivité décrit plus bas).

Activités préparatoires: Construire les boutons «setup» et «aire» de l'installation. Ne cochez pas la case «Forever» de ces deux boutons.

Comment exécuter le programme.
1. Cliquez sur le bouton «setup».
2. Cliquez sur le bouton «aire» pour saisir les points du polygone. Ces points doivent être saisis sous forme de **liste de listes**. Par exemple, dans l'exemple précédent, avec les points a (-2, 3), b (2, 1), c (6, 6), d(1,3). il faut saisir ces données de la manière suivante, sans oublier que le premier point doit être répété: [[-2 3] [2 1] [6 6] [1 3] [-2 3]]

Voici le code :

```
globals[points total]

to setup
  clear-all
end

to aire
  clear-all
  set points read-from-string user-input "mes-points"
  while [length points > 1]
  [let x1 item 0 item 0 points
   let y1 item 1 item 0 points
   let x2 item 0 item 1 points
   let y2 item 1 item 1 points
   set total total + (x1 * y2 - x2 * y1)
   set points but-first points]
  show total / 2
  end
```

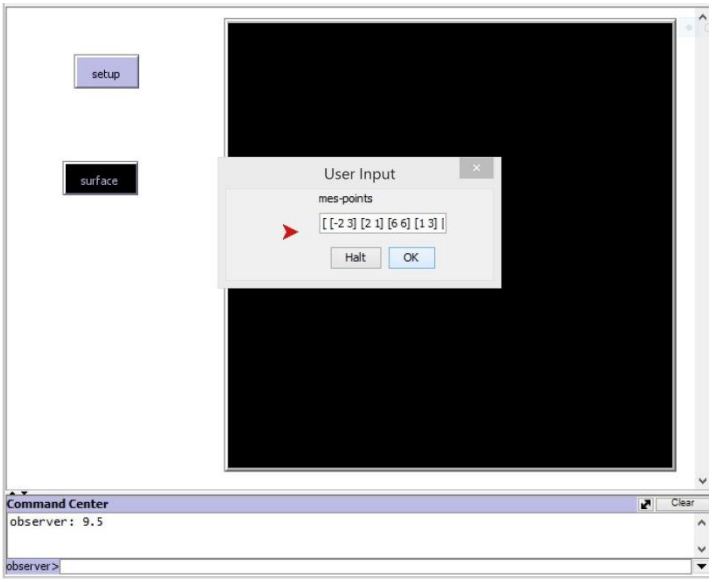


Figure 17.2 : Résultat de l'éxécution du programme «aire»

Explications et commentaires supplémentaires.

L'instruction composée «read-from-string user-input "mes-points"» a pour fonction d'ouvrir une fenêtre qui permet à l'utilisateur de saisir les points du polygone. Ces points doivent être saisis sous forme de liste de listes. L'expression que l'utilisateur entre dans la fenêtre «user-input» est toujours

interprétée comme une chaîne. La primitive «read-from-string» fait en sorte qu'elle soit interprétée comme si elle était écrite dans la fenêtre de l'observateur, dans ce cas comme une liste. L'expression "mes-points" est le nom ou l'identifiant qui a été donné à la fenêtre «user-input» et n'est pas le nom d'une variable

Le schéma de récursivité

En informatique, une procédure récursive est une procédure qui s'appelle elle-même quelque part à l'intérieur de son propre code. Les procédures récursives constituent l'un des mécanismes par lesquels on peut effectuer la répétition ou l'itération d'une procédure ou d'un ensemble d'Instructions. Ces mécanismes constituent une ressource puissante pour résoudre certains types de problèmes. Dans ce livre, dont l'approche est éminemment pratique, nous allons utiliser le concept de récursivité dans le sens syntaxique: si une procédure s'appelle directement ou indirectement, nous dirons que la procédure est récursive. Comme premier exemple, considérons la petite procédure suivante:

```
to iterer
crt 1
ask turtles [fd 1]
iterer
end
```

Cette procédure commence par la création d'une tortue, puis ensuite on demande à toutes les tortues - au début il n'y en a qu'une - d'avancer d'un pas, après quoi la procédure s'appelle elle-même, de sorte que l'interpréteur l'exécute une seconde fois. Cela crée une autre tortue (elles sont deux maintenant) à qui l'interpréteur ordonne d'avancer d'un pas et la procédure s'invoque elle-même pour la troisième fois, créant une troisième tortue et ainsi de suite. La procédure continue à s'invoquer elle-même, en créant plus de tortues qui avancent d'un pas et en principe, s'il n'y avait pas de limite physique, la procédure continuerait à se répéter indéfiniment.

La procédure décrite dans cet exemple n'effectue aucune tâche intéressante, mais illustre le fait que NetLogo est un langage qui prend en charge l'existence de procédures qui s'appellent elles-mêmes. En pratique, il est courant qu'une procédure récursive ne s'exécute pas de manière identique à chaque fois, car elle peut contenir des variables qui changent à chaque appel récursif. Il est également courant qu'une procédure récursive inclue un mécanisme permettant à la procédure de s'arrêter lorsqu'une certaine condition est remplie. L'exemple ci-dessous montre une procédure qui affiche des nombres entiers dans le terminal d'instructions à partir d'un entier fourni en entrée. La procédure s'arrête lorsque le nombre entier 101 est atteint.

```
to montrer-nombres-entiers
[nombre]
if nombre > 100 [stop] show
nombre
wait 0.2
montrer-nombres-entiers nombre
+ 1
end
```

La procédure est appelée, à partir du terminal d'instructions, en attribuant une valeur numérique au paramètre «nombre», par exemple 10 comme dans la commande «*montrer-nombres-entiers 10*». Après avoir donné cette instruction à l'observateur dans le terminal d'instructions les nombres entiers apparaissent alors l'un après l'autre:

```
observer: 10
observer: 11
observer: 12
……..
observer: 99
observer: 100
```

La condition d'arrêt «if nombre > 100 [stop]» met fin à la procédure lorsque la valeur 101 est atteinte.

Dans le chapitre 2, on a vu un exemple dans lequel une tortue dessine un cercle et le parcourt sans arrêt car on avait coché la case «Forever» de la procédure «go». Le même effet pourrait être obtenu en décochant cette case et en plaçant un appel à la procédure «go» dans l'avant dernière ligne de ladite procédure:

```
to go
ask turtle 0 [fd 0.3 right
3]
wait 0.1
go
end
```

Une procédure récursive peut être arrêtée au milieu de répétitions avec l'option «Halt» du menu «Tools».

Traitement récursif d'une liste ou d'une chaîne

Un mécanisme fréquemment utilisé en programmation est le traitement récursif d'une liste ou d'une chaîne: la procédure commence par la liste ou la chaîne et dans chaque appel récursif, c'est-à-dire que chaque fois que la procédure est appelée, une action est effectuée sur la liste ou la chaîne Les actions les plus courantes consistent à supprimer ou à ajouter un nouveau membre à la liste ou à la chaîne. Dans le cas des listes, nous avons vu que la primitive «map» permet également d'effectuer certains types d'opérations sur les listes. Dans ce qui suit, trois manières d'afficher une annonce à l'aide d'un panneau publicitaire sont présentées. Dans les trois cas, le mécanisme de traitement du contenu du panneau est récursif.

Exemple 36: impression d'un panneau publicitaire I

Dans le présent exemple, le texte d'un panneau publicitaire est entré en tant que paramètre d'une procédure. Lorsque la procédure est exécutée, le texte est imprimé dans le terminal d'instruction. Le panneau est représenté par une liste de mots (chaque mot du panneau doit être placé entre guillemets). Le texte peut également être représenté sous forme de chaîne, tel qu'expliqué plus bas, ce qui évite d'avoir à utiliser des guillemets pour chaque mot. La liste est traitée de manière récursive de la manière suivante:

1. Le premier membre de la liste «panneau» est imprimé: *type first* panneau
2. Le premier membre déjà imprimé est supprimé: *set panneau but-first panneau*
3. Le premier membre de la liste, qui occupe maintenant la deuxième place dans la liste (deuxième mot du panneau) est réimprimé et ainsi de suite.
4. L'instruction conditionnelle «if panneau = [ ] [stop]» permet d'arrêter la procédure dès lors que la liste devient vide.

```
to traiter [panneau]
if panneau = [ ] [stop] ;; condition d'arrêt
type first panneau
type " " ;; on imprime un espace blanc pour séparer les mots
wait 0.3
set panneau but-first panneau
traiter panneau
end
```

Explications et commentaires supplémentaires.

Pour exécuter la procédure, tapez, par exemple, les instructions suivantes dans la fenêtre de l'observateur:
traiter ["Chers" "clients ""bonjour :""Nous""avons""déménagé""150""mètres" "au" "sud" "de" "notre" "ancien" "local."]
La phrase précédente est alors imprimée dans le terminal d'instruction avec un délai de 0,3 seconde entre chaque mot.

Les mots sont imprimés sans guillemets (la primitive «type» ne les inclut pas). La procédure s'arrête lorsque la liste est vide. Il est possible d'éviter de mettre chaque mot du panneau entre guillemets en entrant le panneau sous forme de chaîne plutôt que de liste: "Cher client, nous avons déménagé à 150 mètres au sud de notre ancien local", mais il serait alors nécessaire d'adapter la condition d'arrêt pour tenir compte du moment où la chaine devient vide.

if panneau = " " **[stop]**, serait la nouvelle condition d'arrêt

Une autre différence à noter lorsqu'on utilise une chaîne plutôt qu'une liste pour construire la procédure est qu'avec une chaîne le panneau s'imprime caractère par caractère au lieu de mot par mot, car les éléments d'une chaîne sont ses caractères. L'inclusion d'un paramètre dont la fonction est de compter le nombre de répétitions de la procédure récursive est souvent utilisée pour construire la condition d'arrêt.

Exemple 37: Un message aléatoire

Les caractères d'une chaîne sont imprimés de manière aléatoire. Le paramètre appelé «compteur» conserve la trace des appels récursifs et la procédure s'arrête lorsque la valeur de ce paramètre est égale à 50. L'exemple est lancé en appelant la procédure «setup» à partir de la fenêtre de l'observateur.

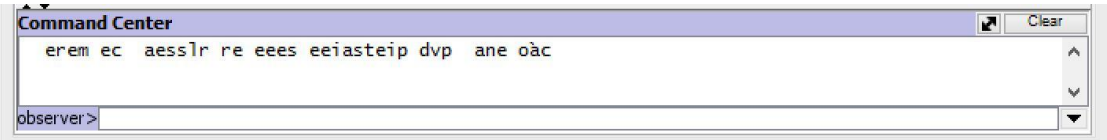
```
globals [message]

to setup
clear-all
set message "La meilleure chose qui puisse arriver à une personne est de travailler pour faire ce qu'elle aime faire"
traiter message 1
end

to traiter [texte compteur]
if compteur = 50 [stop] ;; condition d'arrêt
type item random 84 message
wait 0.2
```

traiter texte compteur + 1 ;; appel récursif
end

Explications et commentaires supplémentaires: La procédure «traiter» est appelée à partir de la procédure «setup» en donnant à «texte» la valeur de la variable «message» et au «compteur» la valeur 1. Le message à partir duquel les caractères sont extraits comporte 84 caractères (les blancs sont des caractères). Le texte obtenu est une chaîne aléatoire de 50 caractères sans signification. Par exemple, au cours d'une exécution de la procédure «setup», nous avons obtenu: «erem ec aesslr re eees eeiasteip dvp ane oàc».



Dans l'exemple suivant, nous allons traiter un tableau d'affichage de manière récursive afin que son contenu apparaisse sur les parcelles du monde. Pour cela, nous utiliserons la primitive «plabel» (étiquette-de-parcelle).

Modèle 18 : Une annonce publicitaire se déplace sur les parcelles

Primitives: user-input, length, item, pxcor, pycor, max-pxcor, reset-ticks, ticks, plabel (étiquette-de-parcelle), plabel-color (couleur-d'étiquette-de-parcelle), stop.
Autres détails: On utilise une fenêtre pour entrer le texte que l'utilisateur veut voir apparaitre sur les parcelles du monde.

Dans cet exemple, nous allons construire un panneau d'affichage dont le contenu (un texte) se déroule de droite à gauche sur les parcelles du monde. Le texte du contenu est saisi par les utilisateurs via la primitive «user-input» qui, comme nous avons déjà eu l'occasion de le voir, ouvre automatiquement une fenêtre permettant de saisir le texte.

Activités préparatoires: Construire les boutons «setup» et «go» (en cochant la case «Forever» de ce dernier) ; sélectionner une taille plus grande pour les caractères de l'étiquette (par exemple 16) à l'aide du le bouton «Configuration» de l'interface (choisir une taille de caractère - «Font settings» - de 20 par exemple), de sorte que le contenu du panneau soit plus visible.

Plan général et problèmes à résoudre. Il existe de nombreuses façons de modéliser un texte animé (mobile) sur un panneau. Le texte du panneau sera saisi par les utilisateurs dans la fenêtre qui s'ouvre automatiquement en utilisant la primitive «user-input» et il sera stocké dans la variable globale «panneau». Le modèle a été conçu de manière à ce que 1) les parcelles du centre du monde (patches with [pycor = 0]) montrent un caractère du texte qui apparaitra sur le panneau et que 2) chaque fois que la procédure «go» est répétée, chaque parcelle lira les caractères suivants du texte.

Cela produira un effet de mouvement. Pour déterminer le caractère que doit lire une parcelle, deux éléments doivent être pris en compte: 1) le temps écoulé depuis le début du défilement du texte sur l'écran, temps déterminé par la variable «ticks» et 2) la position horizontale de la parcelle, donnée par pxcor. A chaque instant, le caractère que la parcelle doit lire (plabel) est donné par l'expression: «set plabel item (pxcor + ticks) panneau» qui peut se traduire par «définir l'étiquette de la parcelle comme l'élément (l'item) occupant la position «pxcor + ticks» de la chaîne «panneau». Il faut s'assurer de ne pas invoquer (stop) les parcelles qui quittent le monde par la gauche «(pxcor + ticks) <0» ou par la droite «(ticks + pxcor> length panneau - 1)».

Le code est le suivant :

```
globals [panneau]

to setup
clear-all
set panneau user-input "Contenu de l'annonce"
reset-ticks
end

to go
if ticks > max-pxcor + length panneau [stop]
ask patches with [pycor = 0]
[ifelse (pxcor + ticks) < 0  or (ticks + pxcor > length panneau -
1)
[set plabel "" stop]
[set plabel-color red set plabel item (pxcor + ticks )
panneau]]
wait 0.3
tick
go
end
```

Explications et commentaires supplémentaires: La primitive «max-pxcor» rapporte la coordonnée «x» des parcelles qui occupent les positions les plus à droite, c'est-à-dire celles dont la valeur de la coordonnée «x» est maximale.

Récursion terminale («tail-recursion»)

L'histoire d'Anne. Anne habite dans une communauté où il y a beaucoup d'enfants et son travail (sa procédure) consiste à lire des histoires, un service qu'Anne rend à la communauté grâce à un soutien municipal. Chaque fois qu'un garçon ou une fille l'appelle au téléphone, elle doit se rendre chez lui ou elle et lire l'histoire qui lui est demandée. Un jour, Anne reçoit un appel de Carmen qui lui demande de raconter l'histoire de Blanche-Neige. Anne commence à lire l'histoire à Carmen, mais à peine arrivée à la page 6, elle reçoit un appel de Jacques un jeune garçon qui veut qu'on lui lise l'histoire de Simbad le marin. Anne interrompt la lecture de Blanche-Neige, après avoir promis à Carmen de revenir plus tard pour finir de lire Blanche-Neige

Avant de quitter, Anne écrit l'adresse de Jacques et l'histoire qu'il veut qu'elle lui lise, mais elle doit également écrire dans son calepin l'adresse de Carmen (7 Avenue des Ormes), le nom de l'histoire qu'elle lisait (Blanche-Neige) et le numéro de la page (page 6) où elle était arrivée quand elle a dû interrompre la lecture. Une fois arrivée chez Jacques, Anne commence à lui lire l'histoire de Simbad le marin, mais, alors qu'elle arrive à la page 11, Anne reçoit un appel de Thérèse, qui veut qu'on lui lise l'histoire du Petit Poucet. Anne ajoute à son cahier l'adresse de Thérèse et l'histoire que cette dernière souhaite écouter, mais elle doit également ajouter l'adresse de Jacques (18 rue de la Patience), l'histoire qu'elle lui racontait (Simbad le marin) et le numéro de la page (page 11) où elle était arrivée lorsque Thérèse a appelé.

Si le nombre d'enfants intéressés par la lecture est élevé et les appels se multiplient, il va arriver un moment où Anne n'aura plus de place pour noter toutes ces informations dans son calepin. Quelque chose de similaire se produit lors d'une procédure récursive. À chaque nouvel appel de la procédure, il faut enregistrer l'état dans lequel les appels récursifs précédents se trouvaient. Anne pourrait cependant adopter une stratégie différente qui lui permettrait de travailler seulement avec une petite feuille de papier, un crayon et une gomme à effacer. Ce qu'Anne devrait faire, c'est tout simplement refuser de répondre aux nouvelles demandes des enfants avant de terminer un récit.

Ainsi, lorsqu'elle répond à l'appel de Raymond, le prochain enfant, c'est parce qu'elle a fini de lire une histoire. Tout ce qu'elle doit écrire dans son calepin,

c'est l'adresse de Raymond et l'histoire qu'il demande. Cela peut se faire sur une seule feuille de papier. Chaque fois qu'une nouvelle demande arrive, Anne supprime les données de la demande précédente et note les nouvelles données. Anne peut ainsi faire son travail et répondre à toutes les demandes en utilisant uniquement une feuille de papier, un crayon et une gomme. Cette forme d'organisation d'une procédure récursive est communément appelée «récursion terminale» car l'appel récursif est placé comme dernière instruction de la procédure récursive.

Le reste de ce chapitre est consacré à des thèmes de mathématiques. Leur compréhension nécessite des connaissances mathématiques qui ne vont pas au-delà de celles acquises au cours des premières années de l'école secondaire (lycées et collèges). Cependant, si les lectrices/lecteurs souhaitent ignorer les parties concernant ces aspects mathématiques elles/ils peuvent passer directement au chapitre suivant qui présente quelques façons d'appliquer NetLogo dans le domaine de la géographie.

La récursivité appliquée à des exemples numériques

La notion de récursivité est particulièrement utile dans certaines procédures effectuant des calculs mathématiques. En fait, cette notion puise ses origines dans la notion mathématique de fonction récursive, et l'un des premiers à l'utiliser fut le grand logicien et mathématicien du XXe siècle, Kurt Gödel, dans la démonstration de son célèbre théorème d'incomplétude [7]. A titre d'exemple, nous allons utiliser la récursivité pour calculer la fonction factorielle d'un entier positif. La factorielle d'un entier positif est définie comme le produit de tous les entiers positifs inférieurs ou égaux au nombre donné et supérieurs à zéro. Par exemple: factorielle 3 = 1 x 2 x 3 = 6, factorielle 4 = 1 x 2 x 3 x 4 = 24. Il est à noter que la factorielle de 1 est égale à 1 et que la factorielle de 0 est, par convention, aussi égale à 1[46].Nous allons, dans ce qui suit, montrer deux manières différentes de calculer de manière récursive la factorielle d'un entier positif.

Exemple 38: Premier schéma

Dans le premier schéma, la procédure «factorielle [nb]», dont l'entrée «nb» est l'entier dont on veut calculer la factorielle, lance la procédure récursive «fact», qui possède trois paramètres d'entrée: [nombre prod compteur].
Le paramètre «nombre» de cette procédure stocke le nombre dont on veut calculer la factorielle, «prod» stocke les produits que calcule la factorielle et

[46] En mathématiques cette opération est notée avec un point d'exclamation. Par exemple factorielle 3 s'écrit 3!

«compteur» est le facteur suivant par lequel est multiplié «prod» dans chaque appel récursif.

Voici le code:

```
to factorielle [nb]
fact nb 1 1
end


to fact [nombre prod compteur]
if compteur = nombre [type prod stop] ;; condition d'arrêt
let produit prod * (compteur + 1)
fact nombre produit compteur + 1
end
```

Explications et commentaires supplémentaires. Pour comprendre ce que fait l'interpréteur, nous allons suivre l'évolution de la procédure factorielle à l'aide d'un exemple. Nous allons calculer la factorielle de 3.
Lorsque nous écrivons «factorielle 3», cette procédure appelle la procédure «fact» avec les valeurs nombre = 3, prod = 1 et compteur = 1. Voyons comment évolue la procédure «fact 3 1 1».

1. Premier appel récursif: fact 3 1 1.
   La condition «if compteur = nombre […]» rapporte faux car la valeur de compteur est 1 et celle de nombre est 3. On passe donc à l'instruction suivante «let produit prod * (compteur + 1)» qui devient «let produit 1 * (1 + 1)», c'est-à-dire que le produit prend la valeur 2. L'instruction suivante «fact nombre produit nombre compteur + 1» est un appel à «fact» avec les nouvelles valeurs: fact 3 2 2:
2. Deuxième appel récursif: fact 3 2 2
   La condition «if compteur = nombre […]» rapporte à nouveau faux car compteur = 2 et nombre = 3). On passe alors à «let produit prod * (compteur + 1)» qui devient «let produit 2 * (2 + 1)», c'est-à-dire que le produit prend la valeur 6. L'instruction suivante «fact produit number compteur + 1» est un appel à «fact» avec les nouvelles valeurs: fact 3 6 3
3. Troisième appel récursif: fact 3 6 3
   La condition «if compteur = nombre […]» rapporte maintenant vrai car «compteur» et «nombre» ont la même valeur 3.
   De ce fait, étant donné que prod = 6, l'instruction entre […] «type prod stop» équivaut à «type 6 stop» et la condition d'arrêt est vérifiée.

Le nombre 6 s'affiche alors dans le terminal d'instructions et la procédure «fact» arrête de faire des appels récursifs (appels s'adressant à elle-même). Une fois ces appels récursifs terminés, l'interpréteur revient à la procédure qui les a déclenchés, à savoir l'instruction «fact 3 1 1» de la procédure factorielle 3. Comme l'instruction qui suit «fact 311» est l'instruction «end», l'ensemble du processus prend fin.

Exemple 39: Deuxième schéma

Prenons le même exemple de la factorielle d'un entier positif, mais en utilisant un schéma récursif différent. Dans ce schéma, nous économisons plusieurs choses: tout d'abord, il n'est pas nécessaire de recourir à une procédure de lancement invoquant la procédure récursive «fact» ; de plus, on pourrait omettre les paramètres additionnels «prod» (qui sert à stocker le produit) et «counter» (qui stocke le facteur suivant). Mais, comme nous le verrons, ce schéma ne présente pas que des avantages, car il reproduit le comportement d'Anne lorsqu'elle prend des appels sans avoir terminé une histoire.

Voici le code:

```
to-report factoriel [nombre]
ifelse nombre < 1 [report 1]
[report nombre *  factoriel (nombre - 1)]
end
```

Examinons le déroulement de cette procédure dans le cas du calcul de 4! (factorielle 4)
Premier appel récursif : factoriel 4
La condition 4 < 1 (nombre < 1) est fausse, ce qui entraine l'exécution du second crochet :
report nombre * factoriel (nombre - 1), c'est à dire
report 4 * factoriel 3, qui se transforme en
report 4 * (3 * factoriel 2), qui se transforme en
report 4 * (3 * (2 * factoriel 1)) qui se transforme en
report 4 * (3 * (2 * factoriel 0)) qui, étant donné que factoriel 0 rapporte 1, se transforme en
report 4 * (3 * (2 * 1))
report 4 * (3 * 2)
report 4 * 6
report 24

L'efficacité d'un code aussi court est surprenante. Mais la chose la plus importante à noter est que chaque appel récursif, à l'exception du dernier, est interrompu pour procéder au prochain appel, puis au prochain.., et ainsi de

suite, jusqu'à ce que l'on atteigne l'appel factoriel 0. Ce processus, dans lequel il reste des opérations en attente au cours de chaque appel récursif nécessite une dépense de mémoire croissante car à chaque appel il est nécessaire de conserver l'état dans lequel se trouvaient les appels précédents.
Si le nombre factoriel que l'on veut calculer est très élevé, la mémoire finit par devenir insuffisante et la procédure s'arrête faute d'espace. Avec le premier schéma, cela ne se produit pas car les informations nécessaires pour calculer le prochain appel récursif de la procédure sont stockées dans les paramètres «prod» et «compteur» et sont renouvelées chaque fois que la procédure est appelée, de sorte que l'espace mémoire utilisé reste constant[47]. Ce second schéma est semblable à la première méthode utilisée par Anne, la jeune femme qui raconte des histoires aux enfants, quand elle est interrompue par un appel téléphonique au beau milieu d'une histoire, alors que le premier schéma correspond à la méthode finalement adoptée par Anne, à qui il suffit de disposer seulement d'une feuille de papier, d'un crayon et d'une gomme pour gérer ses activités.

Les nombres premiers

Les trois exemples de cette section (Modèles 18,19 et 29) sont consacrés à un sujet de mathématiques, à savoir, générer des listes de nombres premiers [5] et dans les deux cas c'est le premier schéma de récursivité qui est utilisé. Afin de maintenir l'uniformité avec les dénominations des sections du livre, nous avons décidé d'appeler ces exemples de «modèles», bien qu'il soit peut-être plus approprié de les appeler «programmes». Les connaissances mathématiques nécessaires à la compréhension de ces modèles sont minimes et n'excèdent pas la connaissance de la division par des nombres entiers apprises au primaire[48]. Le principal défi ne réside donc pas tant dans les mathématiques que dans la conception de la stratégie de calcul et dans sa traduction en code NetLogo. On vérifie si un nombre est un nombre entier en essayant de trouver des diviseurs plus petits que ledit nombre.

Dans le premier modèle, une liste de nombres premiers est construite et dans le second, des «nombres premiers jumeaux» sont extraits de cette liste. Sur Internet vous trouverez un grand nombre de documents en pdf sur la théorie élémentaire des nombres et sur les nombres premiers (voir par exemple [5]).

[47] Mais il varie bien sûr en fonction de la taille du nombre dont on veut calculer la factorielle.
[48] Les lecteurs et lectrices désireux de sauter ces trois exemples peuvent le faire sans que cela n'affecte la compréhension des autres exemples traités dans ce livre.

Rappelons tout d'abord qu'un nombre entier N est considéré comme premier s'il a exactement deux diviseurs, à savoir l'unité et ce nombre lui-même. Les douze premiers nombres premiers sont les suivants: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 et 37. Les entiers non premiers sont appelés nombres composés. Un nombre composé est un nombre différent de zéro qui a 3 diviseurs ou plus (Il a au minimum un autre diviseur en plus de 1 et de lui-même). Le nombre 1 n'est considéré ni premier ni composé car il n'a qu'un seul diviseur. D'autre part, deux nombres premiers séparés uniquement par un entier, sont appelés nombres premiers jumeaux. Dans la liste précédente de 12 nombres premiers, nous pouvons trouver 5 paires de nombres premiers jumeaux, à savoir: 3 et 5, 5 et 7, 11 et 13, 17 et 19, 29 et 31[49]. On sait depuis plus de deux mille ans qu'il y a un nombre infini de nombres premiers[50]. La première démonstration connue est due à Euclide, environ 300 ans avant notre ère. Par contre, on ignore jusqu'à présent s'il existe un nombre infini de paires de nombres premiers jumeaux. Ce problème, simple à formuler mais compliqué à résoudre constitue sans aucun doute l'un de ces problèmes qui feraient la gloire de ceux qui parviendraient à le résoudre[51].

Une liste de nombres premiers

Le moyen le plus direct de déterminer si un nombre N est un nombre premier est de rechercher des diviseurs compris entre des nombres inférieurs à N et supérieurs à 1. Nous allons montrer trois méthodes pour mettre en œuvre cette recherche, chacune étant plus efficace que la précédente. Les trois méthodes décrites ci-dessus utilisent toutes le schéma de récursion terminale, mais seuls les codes des deux dernières méthodes seront présentés et discutés ci-après.

Première méthode: Pour déterminer si un entier N est un nombre premier, on cherche à voir s'il est divisible par 1 et par chacun des entiers qui lui sont inférieurs. Si aucun diviseur autre que 1 et N n'est possible on peut en conclure que l'entier N est premier. Le plus grand nombre de divisions à effectuer se produit précisément lorsque l'entier étudié est un nombre premier. En effet, dans ce cas N - 2 divisions doivent être effectuées (à l'exclusion des divisions par 1 et par l'entier N lui-même). Par exemple, si l'on veut savoir si 101 est un

[49] La paire [3,5] est séparée par le nombre entier 4, la paire [5,7] par 6, etc.

[50] Selon Wikipedia, le plus grand nombre premier connu à ce jour (avril 2021) est le nombre: $2^{82\,589\,933} - 1$, un nombre qui dépasse vingt-quatre millions de décimales (exactement 24 862 048 millions).

[51] En octobre 2019 la plus grande paire connue est $[2996863034895 \cdot 2^{1290000} - 1, 2996863034895 \cdot 2^{1290000} + 1]$. Découverts en 2016, les deux nombres premiers qui composent cette paire ont chacun une longueur de 388,342 chiffres.

nombre premier, l'utilisation de cette première méthode ne permet d'avoir une réponse qu'après 99 divisions. Cette méthode fonctionne infailliblement, mais elle est inefficace et, comme nous le verrons plus loin, elle peut être sensiblement améliorée en réduisant le nombre de divisions à effectuer pour tester la primalité d'un nombre entier.

Deuxième méthode: Cette méthode est basée sur le fait que chaque fois que nous trouvons un diviseur d'un entier, nous en trouvons en fait deux: tant le diviseur que le quotient (ou résultat de la division) sont des diviseurs du nombre. À l'école, on apprend que si on divise 24 par 6, le résultat est 4 et le reste, 0. Cela nous apprend également à prouver que la division est correcte en multipliant 6 x 4 et en vérifiant que le résultat est bien 24. Cette multiplication nous montre que 6 est un diviseur de 24 mais montre également que 4 est un diviseur de 24 (bien que l'on ne nous ait jamais dit cela à l'école primaire). Nous voyons alors que les diviseurs d'un entier N viennent par paires que nous pouvons appeler des paires de diviseurs *complémentaires*. Cette observation nous permet de rechercher les diviseurs d'un entier en concentrant la recherche sur un seul des membres de chaque paire. Pour ce faire, nous allons diviser les diviseurs d'un entier en deux parties de sorte que si l'un des deux diviseurs se trouve dans une partie, son partenaire ou diviseur complémentaire est dans l'autre.

Par exemple, si nous supprimons 1 et 24, les deux parties de la liste des diviseurs de 24 sont:

	Première partie			Deuxième partie		
Diviseurs	2	3	4	6	8	12

En effet, chaque diviseur de la première partie a une paire complémentaire dans la deuxième partie: 2 a 12 comme diviseur complémentaire, 3 a 8 et 4 a 6. Voici comment on peut tirer parti de cette présentation:

1) Pour savoir si un nombre a des diviseurs, il suffit d'effectuer la recherche dans l'une des deux parties car, si on ne trouve aucun diviseur dans la partie inspectée, on peut être sûr qu'il n'y en a pas non plus dans l'autre partie, où devraient se trouver les diviseurs complémentaires (il n'y a pas de diviseurs «célibataires», sans couple, à une exception près dont il sera question plus loin).
2) Il est plus rapide de faire la recherche dans la première partie car celle-ci contient moins de chiffres.

Si nous reprenons l'exemple du nombre 24, dans la première partie, nous rechercherions des diviseurs compris entre 2 et 4 (trois nombres), tandis que dans la deuxième partie, nous devrions les rechercher dans l'intervalle [5,23] (19 nombres). Un cas particulier se produit lorsque le nombre entier est un carré parfait. Dans ce cas, sa racine carrée est un diviseur situé exactement à la frontière entre les deux parties. Par exemple, cela se produit avec le nombre 36, dont les deux parties contenant les paires de diviseurs sont:

	Première partie				Deuxième partie			
Diviseurs	2	3	4	**6**	**6**	9	12	18

C'est le seul cas d'un diviseur dont le partenaire est identique (les deux membres de la paire ont la même valeur) et qui est caractérisé par le fait que la racine carrée du nombre marque la limite entre les deux parties. Au vu de ces remarques, nous pouvons réduire considérablement le nombre de tests à effectuer pour déterminer si un entier est premier: *nous recherchons des diviseurs dans la première partie, qui va de 2 à l'entier le plus proche de la racine carrée du nombre et ne dépasse pas la valeur de cette racine. Si aucun diviseur n'apparaît dans cette partie, nous saurons que le nombre est un nombre premier car il ne peut y avoir aucun diviseur complémentaire dans la deuxième partie*. Illustrons cette méthode en l'appliquant au nombre 101 dont on suppose que l'on ignore qu'il est un nombre premier.

Puisque la racine carrée de 101 est environ 10,049, l'entier le plus proche de 10,049 ne dépassant pas ce nombre est 10 (le premier nombre de la deuxième partie est 11). La première partie des diviseurs de 101 est donc constituée des nombres compris entre 2 et 10. Nous allons donc commencer à chercher des diviseurs de 101 parmi ces nombres (2, 3, 4, 5, 6, 7, 8, 9 et 10). Avec cette nouvelle stratégie, au lieu de devoir faire 99 divisions, nous n'aurons qu'à en faire 9, une différence non négligeable. À mesure que la taille de N augmente, la différence entre le nombre de divisions à examiner entre les deux méthodes augmente considérablement. Pour un nombre de l'ordre du million, tel que le nombre premier 1,000,003 (un million trois), il faudrait, avec la première méthode, faire 1,000,001 (un million et une) divisions, tandis qu'avec la seconde méthode, il ne faudrait en faire que mille puisque la racine carrée de 1,000,003 est environ 1000. Cela fait une différence de 999,001 divisions.

Troisième méthode: Avec cette méthode, nous allons encore réduire le nombre de divisions. Pour ce faire, nous allons rechercher dans la première partie (qui contient, on s'en souvient, les diviseurs de N compris entre 2 et le nombre entier juste inférieur a la racine carrée de N) les diviseurs possibles de N qui sont des nombres premiers. La justification de cette dernière exigence est très simple: si nous savons déjà que N n'est pas divisible par un nombre premier, pourquoi devrions-nous rechercher des diviseurs de N parmi les multiples dudit nombre premier, même s'ils se trouvent dans la première partie? Par exemple, si nous savons que N n'est pas divisible par 2, il ne pourra évidemment pas être divisible par 4, par 6 ou par tout autre multiple de 2. C'est pourquoi nous **n'utiliserons pas**, pour faire le test de primalité de N, les nombres de la première partie **qui ne sont pas** des nombres premiers.

Pour établir, par exemple, que 101 est un nombre premier, il suffirait de vérifier que ce nombre n'est divisible par aucun des nombres premiers de la première tranche qui sont 2, 3, 5 et 7, ce qui réduit encore le nombre de divisions de 9 à 4. Pour le nombre 1,000,003, le nombre de divisions est réduit de 1000 avec la seconde méthode à 168 avec cette nouvelle méthode, car il n'y a que 168 nombres premiers qui ne dépassent pas la racine carrée de 1,000,003. Mais ne risque-t-on pas à ce point de tomber dans le piège de la circularité, car pour générer des nombre premiers, on doit préalablement avoir une liste de nombre premiers pour tester les diviseurs ? Un tel piège est cependant évitable: la liste des nombres premiers sera construite au cours du processus lui-même et nous n'utiliserons jamais de nombres premiers plus grands que le nombre dont nous examinons la possible «primalité».
Nous allons, dans ce qui suit, exposer les codes des deuxième et troisième méthodes. Vous pourrez analyser les légères différences qui existent dans ces deux codes. Les tests de divisibilité entre deux nombres entiers sont effectués avec la primitive «remainder» que nous avons déjà utilisée auparavant, primitive qui rapporte le reste de la division entre deux nombres entiers. Un nombre est divisible par un autre si le reste de la division est égal à zéro, c'est-à-dire si «remainder N d» rapporte la valeur 0.

Modèle 19: génération de nombres-premiers selon la deuxième méthode

Cet algorithme divise les entiers inférieurs à la racine carrée du nombre N dont la primalité est examinée. Le programme est lancé en appelant la procédure «lister-nombres-premiers» à partir de la fenêtre de l'observateur.

Voici le code :
globals [nombres-premiers]

```
to lister-nombres-premiers
clear-all
set nombres-premiers [2]
generer 2 nombres-premiers
;; on commence en essayant le nombre 2 comme possible diviseur
end

to generer [nombre liste]
diviser nombre 2
generer nombre + 1 nombres-premiers
end

to generer [nombre liste]
diviser nombre 2
generer nombre + 1 nombres-premiers
end

to diviser [dividende diviseur]
if (remainder dividende diviseur) = 0 [stop]
if diviseur > sqrt dividende
[set nombres-premiers se nombres-premiers dividende  stop]
diviser dividende diviseur + 1
end
```

Explications et commentaires supplémentaires. La procédure doit être arrêtée manuellement à l'aide de l'option «Halt» du menu « Tools» (Outils). Si on n'arrête pas la procédure, les nombres premiers continueront à être générés (ils ne sont pas envoyés au terminal d'instructions mais stockés dans la variable «nombres-premiers»). Une fois la procédure arrêtée, la liste générée des nombres premiers et la taille de la liste peuvent être consultées dans le terminal d'instruction avec, respectivement, les commandes «show nombres-premiers» ou «show length nombres-premiers».

Modèle 20: génération de nombres premiers selon la troisième méthode

Cet algorithme teste les divisions par les nombres premiers plus petits ou au plus égaux à la racine carrée du nombre N dont la primalité est examinée. Les nombres premiers à tester comme diviseurs possibles sont extraits de la liste des nombres premiers générés par le programme, liste qui est stockée dans la variable «nombres- premiers». Le programme est lancé en appelant la procédure «lister-nombres- premiers» à partir de la fenêtre de l'observateur.
Le code est le suivant :

```
globals[nombres-premiers]

to lister-nombres-premiers
clear-all
set nombres-premiers [2]
générer 2 nombres-premiers
end

to générer [dividende liste]
let i 0
diviser dividende i
générer dividende + 1 nombres-premiers
end

to diviser [dividende i]
let i-ème-nombre-premier item i nombres-premiers
if (remainder dividende i-ème-nombre-premier) = 0 [stop]
if i-ème-nombre-premier > sqrt dividende [set nombres-premiers se nombres-premiers dividende  stop]
diviser dividende i + 1
end
```

On peut consulter la liste des nombres premiers générés dans le terminal d'instructions tel qu'indiqué dans le modèle précédent. Cependant, il est plus pratique de lire des listes volumineuses en envoyant les résultats vers un fichier externe. Les deux méthodes peuvent utiliser la procédure ci-dessous, appelée «résultats», pour afficher la liste des nombres premiers générés dans le fichier «nombres-premiers.txt»[52].

```
to résultats
show last nombres-premiers
file-open "nombres-premiers.txt"
file-type "Ceci est une liste de quelques nombres premiers."
file-type "\r\n"
file-type "La liste contient " file-type length nombres-premiers
file-type " elements"
file-type "\r\n"
file-write nombres-premiers
file-close
end
```

[52] On rappelle qu'il peut être nécessaire de créer un fichier texte vierge intitulé «nombres-premiers.txt» si NetLogo ne le crée pas automatiquement.

Explications et commentaires supplémentaires. La différence fondamentale de la troisième méthode par rapport à la seconde réside dans le fait que les diviseurs testés par le programme proviennent de la liste «nombres-premiers» générée par le programme lui-même.

Le prochain nombre premier à tester en tant que diviseur possible du nombre appelé «dividende» s'appelle «i-ème-nombre-premier» et sa valeur prend celle du i-ème élément de la liste «nombres-premiers» dans l'instruction: «let i-ème-nombre-premier item i nombres-premiers». La procédure «résultats» peut être invoquée une fois que la procédure «lister-nombres-premiers» a été exécutée par l'une ou l'autre des deux méthodes. Dans la collection de modèles du livre, les deux méthodes se trouvent sous les noms de «nombres-premiers2» et «nombres-premiers3». Notez qu'il existe deux procédures récursives, dont les noms sont «générer» et «diviser». La procédure «générer» appelle «diviser», qui, une fois sa routine récursive terminée, retourne à «générer», de sorte que cette procédure s'invoque elle-même avec les nouvelles valeurs de ses variables. Voici un extrait de la dernière partie de la liste des 182239 nombres premiers générés par la méthode 3 en 10 secondes:

2482889 2482903 2482913 2482933 2482937 2482943 2482967 2482973 2482981 2482993
2482999 2483017 2483027 2483059 2483077 2483093 2483099 2483113 2483119 2483137
2483141 2483147 2483161 2483171 2483179 2483219 2483233 2483291 2483381 2483417
2483431 2483447 2483461 2483483 2483519 2483521 2483543 2483549 2483561 2483567
2483599 2483603 2483617 2483641 2483653 2483659 2483669 2483671 2483687 2483693
2483707 2483711 2483713 2483729 2483743 2483749 2483753 2483777 2483797 2483827
2483837 2483861 2483867 2483869 2483881 2483911 2483917 2483939 2483953 2484011
2484017 2484019 2484037 2484049 2484059 2484089 2484109 2484113 2484127 2484133
2484151 2484179 2484191 2484197 2484199 2484203 2484233 2484241 2484259 2484271
2484289 2484311 2484319 2484323 2484331 2484353 2484359 2484379 2484473 2484491
2484509 2484523 2484527 2484539 2484563 2484569 2484571 2484589 2484593 2484617
2484623 2484631 2484653 2484673 2484679 2484683 2484689 2484697 2484707 2484721
2484731 2484733 2484739 2484751 2484803 2484827 2484857 2484863 2484871 2484893
2484899 2484901 2484917 2484919 2484931 2484959 2484961 2484971 2484973 2485001
2485003 2485027 2485033 2485037 2485061 2485069 2485073 2485121 2485123 2485129
2485159 2485169 2485183 2485187 2485193 2485207 2485211 2485243 2485277 2485279
2485283 2485303 2485319 2485339 2485367 2485381 2485391 2485393 2485397 2485421
2485429 2485453 2485477 2485481 2485489 2485507 2485513 2485537 2485547 2485559
2485573 2485579 2485607 2485627 2485631 2485643 2485649 2485657 2485663 2485667
2485669 2485687 2485727 2485733 2485739 2485759 2485801 2485807 2485831 2485849
2485867 2485897 2485907 2485937 2485939 2485949 2485991 2485997 2485999 2486009
2486027 2486039 2486041 2486059 2486069 2486089 2486101 2486123 2486137 2486147
2486149 2486153 2486167 2486189 2486191 2486203 2486219 2486221 2486243 2486251
2486269 2486273 2486287 2486291 2486333 2486371 2486381 2486383 2486387 2486423
2486443 2486459 2486467 2486483 2486501 2486509 2486513 2486521 2486531 2486551
2486557 2486563 2486567 2486579 2486581 2486591 2486593 2486611 2486623 2486639
2486651 2486669 2486677 2486681 2486689 2486699 2486713 2486717 2486747 2486753
2486761 2486767 2486801 2486831 2486833 2486843 2486857 2486863 2486867 2486873
2486951 2486963 2486969 2486971 2486987 2486993 2487047 2487061 2487071 2487073

2487091 2487097 2487113 2487137 2487139 2487143 2487167 2487203 2487211 2487227
2487229 2487259 2487269 2487281 2487293 2487299 2487307 2487313 2487319 2487341
2487349 2487367 2487383 2487391 2487413 2487431 2487439 2487467 2487481 2487493
2487497 2487517 2487521 2487523 2487557 2487571 2487581 2487587 2487599 2487601
2487619 2487623 2487629 2487637

Exercice: Modifiez la procédure précédente «lister-nombres-premiers», dans l'une quelconque de ses deux versions, afin qu'elle lance la recherche de nombres premiers à partir d'un nombre initial donné en entrée. Par exemple, si la procédure est appelée sous la forme «lister-nombres-premiers 1000», la liste des nombres premiers est générée à partir du nombre 1000.

Modèle 21 : Liste de nombres premiers jumeaux

Dans ce modèle, une liste contenant des paires de nombres premiers jumeaux est générée. Tel que mentionné précédemment, on dit que deux nombres premiers sont des jumeaux si la différence entre les deux est égale à 2, c'est-à-dire si, entre eux deux, il n'y a qu'un seul entier. C'est le cas des paires de nombres entiers 5 et 7, 11 et 13 ou 29 et 31. Nous allons construire une procédure qui prend comme entrée une liste de nombres premiers et extrait de cette liste uniquement les paires de nombres premiers jumeaux. La liste des nombres premiers en entrée est générée par la procédure du modèle précédent, procédure qui génère des nombres premiers. Tout ce que la nouvelle procédure «lister-premiers-jumeaux» doit faire est de comparer chaque entier de la liste d'entrée (nombres premier ou pas) avec l'entier qui le précède et de vérifier si la différence entre les deux est égale à deux. Si tel est le cas, les deux entiers sont inclus dans la nouvelle liste de nombres premiers jumeaux. Nous présentons le code de cette procédure précédé du code de la procédure précédente du générateur de nombres premiers, en ajoutant la variable «premiers-jumeaux» à la liste des variables globales. Cette variable est utilisée uniquement dans la procédure «lister-nombres-premiers-jumeaux».

Voici le code:

```
globals [nombres-premiers nombres-premiers-jumeaux]

to lister-nombres-premiers
clear-all
set nombres-premiers [2]
générer 2 nombres-premiers
end

to générer [dividende liste]
let i 0
```

```
diviser dividende i
générer dividende + 1 nombres-premiers
end

to diviser [dividende i]
let i-ème-nombre-premier item i nombres-premiers
if (remainder dividende i-ème-nombre-premier) = 0 [stop]
if i-ème-nombre-premier > sqrt dividende [set nombres-premiers se nombres-premiers dividende  stop]
diviser dividende i + 1
end

;; Ici commence la nouvelle procédure qui sélectionne les paires de nombres
 ;; premiers jumeaux générés par la procédure précédente.

to  lister-nombres-premiers-jumeaux
set nombres-premiers-jumeaux []
traiter-nombres-premiers-jumeaux nombres-premiers 1
end

to traiter-nombres-premiers-jumeaux  [liste i]
if liste = [] [type "La liste de nombres premiers est vide" stop]
if (item (i + 1) nombres-premiers - item i nombres-premiers = 2)
[set nombres-premiers-jumeaux lput (item i nombres-premiers) nombres-premiers-jumeaux
set nombres-premiers-jumeaux lput (item (i + 1) nombres-premiers) nombres-premiers-jumeaux]
if (item (i + 1) nombres-premiers) = last nombres-premiers [stop]
traiter-nombres-premiers-jumeaux  nombres-premiers i + 1
end
```

Les procédures qui suivent doivent être appelées à partir de la fenêtre de l'observateur et elles envoient la liste des nombres premiers ou des nombres premiers jumeaux vers un fichier externe appelé «nombres-premiers.txt».

```
to voir-nombres-premiers
file-open "nombres-premiers.txt"
file-type  "Ceci est une liste des premiers nombres premiers."
file-type "\r\n"
file-type "La  liste contient "
file-type length nombres-premiers
file-type " nombres-premiers"
file-type "\r\n"
file-write nombres-premiers
```

```
file-type "\r\n"   ;; C'est une bonne idée de fermer avec un retour de charriot
;; au cas où l'on voudrait ajouter quelque chose au fichier plus tard
file-close
end

to voir-nombres-premiers-jumeaux
file-open "nombres-premiers.txt"
file-type "Ceci est une liste de nombres premiers jumeaux "
file-type "\r\n"
file-type "La liste contient "
file-type length nombres-premiers-jumeaux
file-type " nombres-premiers-jumeaux"
file-type "\r\n"
file-write nombres-premiers-jumeaux
file-type "\r\n"
file-close
end
```

Explications et commentaires supplémentaires. La procédure «lister-nombres-premiers-jumeaux» doit être exécutée après la procédure «lister-nombres-premiers». Si ce n'est pas fait, le programme affichera un message indiquant que la liste des nombres premiers est vide. Dans la liste de 182 239 nombres premiers générée précédemment en 10 secondes, 35 750 nombres-premiers-jumeaux (17 875 paires) ont été trouvés. Voici les dernières paires de nombres premiers jumeaux de la liste:

2468099 2468101 2468129 2468131 2468447 2468449 2468951 2468953 2468969 2468971
2469281 2469283 2469317 2469319 2469407 2469409 2469431 2469433 2469557 2469559
2469581 2469583 2469869 2469871 2470001 2470003 2470121 2470123 2470199 2470201
2470241 2470243 2470331 2470333 2470337 2470339 2470691 2470693 2470889 2470891
2471057 2471059 2471087 2471089 2471321 2471323 2471471 2471473 2471531 2471533
2472179 2472181 2472539 2472541 2472557 2472559 2472851 2472853 2472929 2472931
2472959 2472961 2473127 2473129 2473181 2473183 2473421 2473423 2473451 2473453
2473607 2473609 2473631 2473633 2474051 2474053 2474117 2474119 2474207 2474209
2474711 2474713 2474861 2474863 2475089 2475091 2475287 2475289 2475437 2475439
2475797 2475799 2475857 2475859 2475959 2475961 2476037 2476039 2476079 2476081
2476391 2476393 2476421 2476423 2476751 2476753 2477129 2477131 2477159 2477161
2477171 2477173 2477309 2477311 2477327 2477329 2477411 2477413 2477609 2477611
2477639 2477641 2478239 2478241 2478269 2478271 2478347 2478349 2478521 2478523
2478527 2478529 2478587 2478589 2479487 2479489 2479661 2479663 2479667 2479669
2479691 2479693 2479847 2479849 2479901 2479903 2480081 2480083 2480207 2480209
2480501 2480503 2480717 2480719 2480909 2480911 2481137 2481139 2481179 2481181
2481317 2481319 2481497 2481499 2481839 2481841 2481887 2481889 2481977 2481979
2482349 2482351 2482619 2482621 2482769 2482771 2483519 2483521 2483669 2483671
2483711 2483713 2483867 2483869 2484017 2484019 2484197 2484199 2484569 2484571
2484731 2484733 2484899 2484901 2484917 2484919 2484959 2484961 2484971 2484973
2485001 2485003 2485121 2485123 2485277 2485279 2485391 2485393 2485667 2485669

2485937 2485939 2485997 2485999 2486039 2486041 2486147 2486149 2486189 2486191
2486219 2486221 2486381 2486383 2486579 2486581 2486591 2486593 2486831 2486833
2486969 2486971 2487071 2487073 2487137 2487139 2487227 2487229

Chapitre 8: Exploration géographique.

“Doctor Livingstone I presume?”
(Question du journaliste et explorateur Henry Morton Stanley à l'explorateur David Livingstone.)

is-turtle? (demanda la parcelle à la tortue qui occupait sa surface).
==> True

Promenade V

«Docteur Livingstone, je présume?» Ces mots furent les premiers que prononça Stanley en rencontrant Livingstone dans le petit village d'Ujiji sur les rives du lac Tanganyika après une recherche intense à travers le continent africain. Livingstone était le seul homme blanc à des milliers de kilomètres à la ronde, ce que Stanley savait bien, c'est pourquoi cette phrase est aujourd’hui gravée dans toutes les mémoires. Livingstone mourut en 1873, peu de temps après sa rencontre avec Stanley, terrassé par la fatigue et le paludisme, après plusieurs années d’explorations sur le continent africain à la recherche des sources du Nil blanc (l'une des deux grandes branches qui composent le Nil). La grande admiration et l’amitié intense que Stanley eut pour Livingstone fut ce qui le motiva à poursuivre et à finir l'entreprise initiée par ce dernier. Cependant, au lieu de trouver les sources du Nil Blanc, Stanley découvrit l'origine du deuxième plus grand fleuve du monde, surpassé uniquement par le fleuve Amazone: le fleuve Congo [7].
Les modèles présentés dans ce chapitre suggèrent quelques idées sur la façon d'appliquer NetLogo dans les domaines de la géographie et de l'étude de l'utilisation des terres, en tirant parti des photographies satellites offertes par Google Earth. La raison principale de la construction de ces modèles est de stimuler les idées sur l'utilisation de NetLogo en profitant de la possibilité d'importer des cartes et des figures et de les placer comme arrière-plan du monde. Il est merveilleux de penser qu'aujourd'hui n'importe qui peut ouvrir l'application gratuite de Google Earth et avec la souris d'ordinateur, sans quitter le confort de son bureau et sans avoir à affronter les dangers du paludisme et des virus[53], «voler» pour atteindre les endroits les plus exotiques et les plus reculés de ce monde comme si nous étions Superman ou Superwoman.

[53] Ce chapitre est écrit en Novembre 2020 alors que sévit la pandémie COVID-19 qui secoue la planète entière

Nous pourrions, par exemple, partir de l'embouchure d'un des grands fleuves et suivre son cours à contre-courant jusqu'à atteindre sa source. Nous pourrions partir de l'embouchure du Nil en Égypte et prendre à Khartoum, la capitale du Soudan, la branche connue sous le nom de Nil Bleu[54] (branche à gauche lorsque nous volons à contre-courant) pour remonter à ses origines situées dans le lac Tana dans les montagnes d'Éthiopie. L'intérieur du continent africain fut un mystère fascinant pour les Européens jusqu'au début du XXe siècle. L'invention et le développement de l'aviation ont permis de faire des pas de géant dans la connaissance de notre propre monde mais elles ont, en même temps, éliminé une grande partie du mystère et de l'aventure engendrés par l'exploration terrestre de grandes zones inconnues de notre planète[55] avec comme seule aide des porteurs et des bêtes de somme.

NetLogo interagit avec le paysage

Dans cet exemple, nous allons montrer comment NetLogo peut être utilisé pour interagir avec une figure importée, par exemple, une photographie satellite prise avec la version gratuite de Google Earth. La photographie est utilisée comme arrière-plan du monde NetLogo.

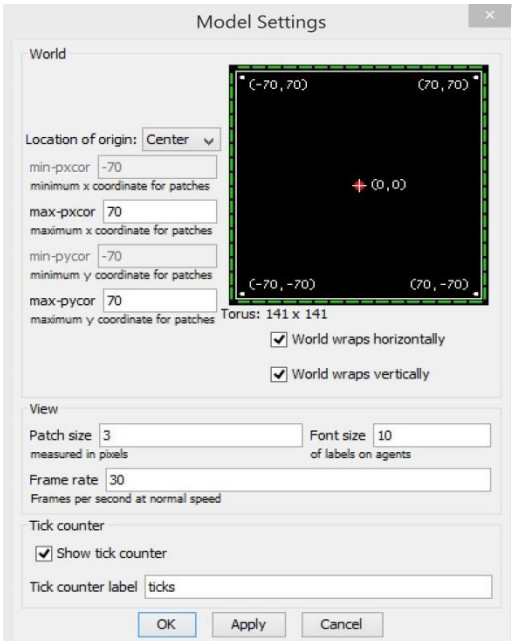
Nous allons commencer par construire un premier modèle (modèle 22.a) qui permet de représenter certains types d'interactions avec un paysage importé de Google Earth. Ensuite, nous modifierons ce modèle en incorporant de nouvelles fonctionnalités (modèle 22.b).

Le premier modèle consiste à montrer comment, au moyen de boutons de l'interface et de très simples et courtes procédures (comme on le fait avec les programmes de la famille Lisp), il est possible pour l'utilisateur de conduire la tortue à travers un paysage emprunté à Google Earth, de manière à laisser l'utilisateur guider le chemin de la tortue. Dans ce modèle on mesure la distance (nombre de pas) parcourue par la tortue et on enregistre cette distance dans une variable. Le tableau suivant contient les primitives utilisées dans les modèles 22a et 22b.

[54] À Khartoum, capitale du Soudan, deux grands cours d'eau appelés le Nil Blanc et le Nil Bleu se rejoignent pour former le grand Nil.
[55] Les chutes de l'ange (Salto Angel), dans la jungle vénézuélienne ont, pour le «monde civilisé», été découvertes par le pilote nord-américain Jimmy Angel dont le nom de famille leur a donné le nom qu'elles portent actuellement.

Primitives: Import-drawing (importer-dessin), import-pcolors-rgb (importer-couleurs-parcelles-rgb) mouse-down? (souris-en-bas?), mouse-xcor, mouse-ycor. Autres détails: La recherche d'image a été faite en utilisant l'application gratuite Google Earth et, pour capturer une région de l'image choisie, nous avons utilisé l'application gratuite de capture d'écran Cropper

Activités préparatoires. Configurez le monde avec des petites parcelles: cliquez sur le bouton «Settings» de l'interface, entrez les valeurs indiquées sur la figure 8.1a (si elles n'y sont pas déjà par défaut) et appuyez sur le bouton OK:



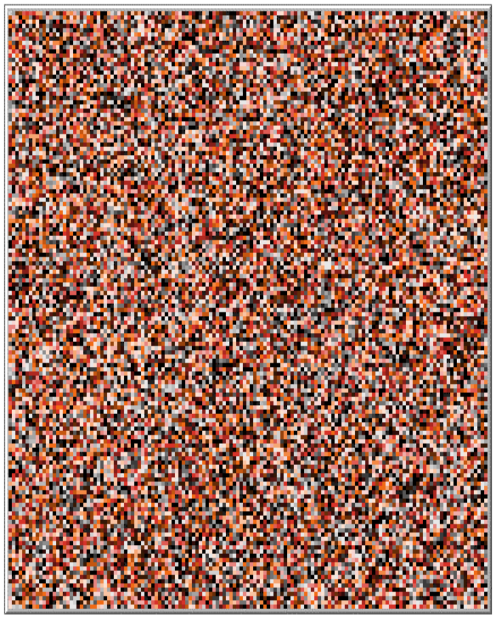


Figure 8.1a

Figure 8.1b

Après avoir configuré le monde de la manière indiquée dans la fenêtre «Model Settings» (Figure 8.1a), nous pouvons voir le nombre et la taille des parcelles qui le composent (figure 8.1b) en les peignant au moyen de couleurs aléatoires en utilisant, par exemple, l'instruction «ask patches [set pcolor random 30]». Le monde est composé de 141x141 = 19881 parcelles[56] (pour les fins poursuivies dans les modèles 22a et 22b, il est utile d'avoir un monde «à grain fin»).

[56] Chaque côté du monde contient 141 = 70 + 70 +1 parcelles ou bien mesure 141 pas de tortue (1 représente la parcelle située au centre d'un côté).

Modèle 22a: Promenade à travers le paysage

Activités préparatoires: Construire les boutons pour les procédures setup, place-souris, avance, droite, gauche, carte1, voir-parcelles, nettoyer et distance. La case «Forever» de chacun de ces boutons ne doit **pas** être cochée.

1. Bouton «setup»: prépare l'environnement.
2. Bouton «carte1»: importe le paysage et le place comme arrière-plan du monde.
3. Bouton «place-souris»: donne un petit temps d'attente à l'utilisateur pour lui permettre de choisir le point de départ de la promenade de la tortue sur la carte.
4. Bouton «avance»: fait avancer la tortue d'un pas en avant.
5. Bouton «droite»: fait tourner la tortue de 10 degrés vers la droite.
6. Bouton «gauche»: fait tourner la tortue de 10 degrés vers la gauche.
7. Bouton «voir-parcelles»: affiche les petites parcelles qui composent le monde (comme sur la Figure 8.1b).
8. Bouton «nettoyer»: nettoie la carte qui a été placée comme arrière-plan du monde ainsi que les tracés.
9. Bouton «distance» : mesure (en pas) le chemin parcouru par la tortue.

L'idée du modèle est de guider la tortue à travers le paysage en appuyant à plusieurs reprises sur les boutons <<avance>>, <<droite>> et <<gauche>>. La tortue laissera une trace rouge (ou une couleur qui contraste avec le paysage) au fur et à mesure de sa progression. On suppose qu'avec une progression de 2 pas et des virages de 10 degrés, il est possible de guider la tortue avec un degré de précision acceptable. La diminution de ces valeurs permettrait une plus grande précision tandis que leur augmentation permettrait à la tortue d'avancer plus rapidement mais ce au détriment d'une moindre précision.

Figure 8.2

La figure 8.2 illustre la trajectoire (chemin en rouge) de la tortue le long d'une section de route dans la ville de Namwala (Zambie) située sur les rives de la rivière Kafue, le principal affluent du Zambèze, où se trouvent les spectaculaires chutes Victoria. L'objectif de la caméra est situé à 1004 m au-dessus du sol. En appuyant sur le bouton «distance parcourue» on vérifie que la distance parcourue par la tortue est, dans le cas illustré par la Figure 8.2, de 25 pas. Le code du modèle 22a est fourni ci-après.

```
globals [parcours]

to setup
clear-all
crt 1 [set color yellow set size 3]
end

to nettoyer
clear-all
end

to placer-souris
 wait 2 if mouse-down? [ask turtle 0 [pu setxy mouse-xcor mouse-ycor
   set heading 0 ]]
  ;; amenez rapidement la souris au point désiré,
;; maintenez le bouton de la souris enfoncé pendant 2 secondes
;; (jusqu'à ce que le bouton «placer-souris» ne soit plus noir).
end
```

```
to avance
ask turtle 0 [pd set color red fd 2 set parcours parcours + 1]
end

to gauche
ask turtle 0 [pd  lt 10]
end

to droite
ask turtle 0 [pd  rt 10]
end

to carte1
import-drawing "Namwala.bmp"
end

to voir-parcelles
clear-all
ask patches [set pcolor random 30]
end

to distance-parcourue
type parcours type " "
end
```

Explications et commentaires supplémentaires. L'image a été capturée avec Google Earth et une partie de celle-ci avec l'application gratuite Cropper au format BMP. Les autres formats pris en charge par NetLogo sont JPG, GIF et PNG. L'instruction «if mouse-down? [ask turtle 0 [pu setxy mouse-xcor mouse-ycor set heading 0]]» demande si le bouton de la souris est enfoncé et si tel est le cas (true) demande à la tortue de lever le stylet et (à la fin des 2 secondes d'attente de la commande précédente) de se diriger vers le point dont les coordonnées x, y sont celles où se trouve la souris. Cette instruction donne également la valeur 0 à l'orientation (pour que l'on sache dans quelle direction la tortue commence à marcher). Si l'image importée ne se trouve pas dans le même répertoire que le code du modèle, il est nécessaire de fournir le chemin complet de l'endroit où se trouve l'image.
En fonction de l'ordre dans lequel les boutons de procédure sont enfoncés, il peut parfois arriver qu'une fenêtre d'erreur s'affiche: *«ASK expected input to be an agent or agentset but got NOBODY instead»* indiquant que la primitive «ask» attend un agent et ne l'a pas trouvé. Cela est dû au fait que

l’on a invoqué une procédure qui envoie des instructions à une tortue avant la création de ladite tortue. Étant donné que la procédure «setup» crée une tortue, il est recommandé de commencer la promenade avec cette procédure. Nous aurions pu inclure la commande «crt 1» dans la procédure «carte1» mais nous ne l'avons pas fait pour ne pas créer plus de tortues que nécessaire. Les utilisateurs peuvent, s’ils le désirent, expérimenter en introduisant des variantes de cet aspect.

Calcul des distances réelles en mètres ou en kilomètres.

Pour calculer la distance parcourue par la tortue en mètres ou en kilomètres, il est nécessaire de faire un petit exercice d'étalonnage. Les distances parcourues par les tortues sont mesurées en pas dont la longueur est constante et indépendante de la carte qui sert de fond au monde. Les distances des images Google Earth, en revanche, dépendent de la hauteur de l’objectif de la caméra et peuvent être mesurées avec l'outil «règle» disponible dans Google Earth, un outil qui s'adapte automatiquement à la hauteur de la caméra. Pour cette raison, il est essentiel d'enregistrer la hauteur à laquelle Google Earth a pris l'image (indiquée dans le coin inférieur droit de l'image).

Pour convertir les pas des tortues en distances réelles sur la carte Google Earth, il est nécessaire d'établir un facteur de conversion entre les deux mesures. Il faut commencer par préparer la carte Google Earth avec laquelle nous désirons travailler. Nous proposons la méthode de mesure suivante.

1. Choisissez la carte de travail sur Google Earth (bien que cela ne soit pas nécessaire il est recommandé d'enregistrer la hauteur de l’objectif de la caméra).
2. Utilisez la règle Google Earth pour mesurer une distance sur la carte (plus la distance est grande, plus l'erreur est petite).
3. Capturez l'image incluant le segment de droite de la règle et enregistrez cette image dans le dossier ou se trouve la procédure «carte» (on suppose que l’on a nommé l’image «ma-carte» et que cette dernière a été enregistrée sous le format bmp).
4. Ouvrez la procédure «carte» et chargez l’image en écrivant la commande «import-drawing "ma-carte"» dans la fenêtre de l’observateur.
5. Faites en sorte que la tortue parcourt le même tracé que celui marqué par la règle de Google Earth en respectant d’aussi près que possible ce tracé.
6. Appuyez sur le bouton «distance» et enregistrez la longueur du trajet de la tortue en pas.
7. Calculez le facteur de conversion en utilisant une simple règle de trois.

8. Maintenant que nous disposons du facteur de conversion nous pouvons effacer l'écran à l'aide de la procédure «nettoyer», recharger «ma-carte» et effectuer de nouveaux trajets avec la tortue.

Modèle 22b: Calcul des aires sur le paysage (Estelí)

Ce modèle est une extension du modèle précédent. Notre but est de marquer avec la souris des points de repère du paysage qui, lorsqu'ils sont joints par des lignes droites, forment un simple polygone (pas nécessairement régulier ou convexe[57]). Une fois ces points de repère posés, il est possible de calculer l'aire de cette région fermée constituée par le polygone à l'aide de la règle de calcul de l'aire d'un terrain polygonal (Modèle 17). Les points de repère sont stockés dans une liste. Une petite procédure de correction qui permet de supprimer le dernier point de repère de la liste a été construite au cas où nous considérerions que sa position n'était pas la bonne. La procédure qui place les points de repère est distincte de la procédure qui trace le contour du polygone.

Activités préparatoires. Construisez des boutons pour les nouvelles procédures: marquer, supprimer, polygone et zone. La case «Forever» ne doit en aucun cas être cochée.

Figure 8.3 Vue de l'interface du modèle de l'exemple 22b.

9. Bouton «marquer»: inclut dans la liste des «points de repère» les coordonnées du point marqué avec la souris.

[57] Une région du plan est dite convexe si, étant donné deux points quelconques à l'intérieur de cette région, le segment de droite qui les joint est totalement contenu dans la région. Une étoile est l'exemple-type d'une figure non convexe.

10. Bouton «supprimer»: supprime les coordonnées du dernier point de la liste des «points de repère».
11. Bouton «polygone» : dessine le contour du polygone dans l'ordre dans lequel les points ont été marqués. Il est alors possible de vérifier si le polygone est formé de côtés non entrecroisés.
12. Bouton «aire»: calcule l'aire de la région délimitée par les côtés du polygone. L'unité utilisée pour effectuer ce calcul correspond à l'aire d'un carré d'un côté égal à un pas de tortue, c'est-à-dire à l'aire d'une parcelle. Pour convertir l'aire de la région polygonale en unités réelles (mètres carrés ou hectares), il faut utiliser le facteur de conversion des distances élevé au carré.

Comment dessiner un polygone et calculer son aire dans le modèle 22b.

1. Chargez une figure avec la procédure «carte2»[58].
2. Pour marquer les points du polygone, appuyez sur le bouton «marquer» et amenez le pointeur de la souris à l'endroit où vous voulez placer le premier point, faites un clic gauche et maintenez le bouton jusqu'à ce que le bouton «marquer» revienne à sa couleur normale (vous disposez de 3 secondes pour faire cette opération). Faites de même pour les autres points du polygone.
3. Une fois que vous avez placé tous les points (vous ne devez pas marquer le premier point une deuxième fois), appuyez sur le bouton «polygone-points-de repère 0» pour voir le polygone que vous avez construit.
4. Si vous souhaitez connaître la valeur de l'aire de ce polygone, appuyez sur le bouton «aire».
Le code du modèle 22b avec les nouvelles procédures mises en évidence en caractères gras est fourni ci-après.

```
globals [parcours points-de-repère aire-totale]

to setup
  clear-all
  crt 1 [set color yellow set size 3]
  set points-de-repère []
  end

to nettoyer
clear-all
end
```

[58] Cette carte représente un paysage situé dans le Département d'Estelí, l'un des 15 départements du Nicaragua.

```
to placer-souris
 wait 2 if mouse-down? [ask turtle 0 [pu setxy mouse-xcor mouse-ycor
   set heading 0 ]]
  ;; amenez rapidement la souris au point désiré,
;; maintenez le bouton de la souris enfoncé pendant 2 secondes
;; (jusqu'à ce que le bouton «placer-souris» ne soit plus noir).
end

to avance
ask turtle 0 [pd set color red fd 2 set parcours parcours + 1]
end

to gauche
  ask turtle 0 [pd  lt 10]
end

to droite
  ask turtle 0 [pd  rt 10]
end


to carte2
import-pcolors-rgb "Esteli.bmp"
 end

to voir-parcelles
  clear-all
ask patches [set pcolor random 30]
end

to distance-parcourue
type parcours type " "
end

to marquer
  wait 3 if mouse-inside? [if mouse-down?
   [ask turtle 0 [pu setxy mouse-xcor mouse-ycor stamp]
    set points-de-repère lput (list mouse-xcor mouse-ycor) points-de-repère
]]
  ask turtle 0 [ pu setxy item 0 item 0  points-de-repère item 1 item 0
points-de-repère ]
end
```

```
to effacer-points-de-repère
  if length points-de-repère > 0 [set points-de-repère butlast points-de-
repère]
end

to polygone [liste m]
  ask turtle 0 [set color yellow]
   ifelse m = length liste [ask turtle 0 [pd setxy item 0 item 0 liste item 1
item 0 liste stop]]
  [ask turtle 0 [pd setxy item 0 item m liste item 1 item m liste] polygone
liste m + 1]
end

to aire
;; le premier point de repère est répété à la dernière place:
  set points-de-repère  lput first points-de-repère points-de-repère
  while [length points-de-repère > 1]
  [let x1 item 0 item 0 points-de-repère
   let y1 item 1 item 0 points-de-repère
   let x2 item 0 item 1 points-de-repère
   let y2 item 1 item 1 points-de-repère
   set aire-totale aire-totale + (x1 * y2 - x2 * y1)
   set points-de-repère but-first  points-de-repère]
  show aire-totale / 2
end
```

Explications et commentaires supplémentaires.

Le bouton «polygone» correspond à la procédure «polygone [liste m]» qui fonctionne à l'aide de deux entrées. Il est nécessaire d'inclure dans le nom du bouton les valeurs des deux entrées, qui sont: la liste des points de repère et la longueur initiale de ladite liste.

L'un des objectifs du modèle 22 est d'élargir le champ des possibilités offertes par NetLogo et de motiver les lecteurs à construire des modèles qui prennent avantage de la collaboration entre NetLogo et des outils utiles (et gratuits!) tels que Google Earth, Google Maps ou des outils de capture d'écran tels que Cropper, ou tous autres outils disponibles sur Internet. Nous invitons les lecteurs à modifier et à améliorer les modèles 22a et 22b. Nous terminerons ce chapitre en donnant un exemple de possibilités liées à la capture d'image.

Jusqu'à présent, nous n'avons pas exploité le fait que le monde NetLogo est constitué de très petites parcelles. Dans les deux exemples précédents, nous avons travaillé avec un monde composé de 141x141 = 19881 parcelles, chacune d'entre elles étant un agent capable de stocker ses propres variables et procédures, mais cette propriété a été peu utilisée. Les cartes importées dans les exemples 22a, à l'aide de la primitive «import-drawings» sont des figures inertes qui sont placées sur une couche, au-dessus des parcelles. Cependant, il est possible de demander aux parcelles elles-mêmes d'importer une figure, copiant (approximativement) la couleur de la partie de carte qui occupe la même position.
Pour cela, la primitive «import-pcolors-rgb» est utilisée. C'est ce que fait la procédure «carte2», laquelle contient la commande «import-pcolors-rgb "Esteli.bmp"»
La figure 8.4 illustre le résultat de cette opération.

Figure 8.4 : Image obtenue à l'aide de «import-pcolors-rgb»

L'importation de la figure à l'aide de cette deuxième procédure entraine une résolution légèrement inférieure à celle de la primitive «import-drawing». La raison en est que les unités de couleur ou «pixels» de l'image sont de la taille des parcelles et chaque parcelle ne peut stocker plus d'une couleur. Cependant, il est possible de placer des tortues dans le monde pour qu'elles fassent tout ce qu'elles savent faire, comme marquer des points de repère et dessiner des polygones. Ce qui est intéressant dans cette nouvelle approche, ce sont les possibilités que nous offre le fait d'avoir une carte «vivante» en ce sens qu'elle est constituée d'agents capables de stocker des informations et d'effectuer des opérations comme changer de couleur. Nous

invitons le lecteur à explorer le thème des couleurs dans NetLogo, en consultant le Manuel NetLogo et le Dictionnaire de Primitives.
Les couleurs dans NetLogo peuvent être gérées de deux façons:

1. D'une façon simple, en décrivant certaines couleurs par leur nom et un nombre qui leur est associé. NetLogo reconnaît ainsi 16 noms de couleurs différents, chacune étant représentée par un nombre spécifique (noir = 0, gris = 5, rouge = 15, etc.). Cela ne signifie cependant pas que NetLogo ne reconnaît que 16 couleurs. Pour obtenir les différentes nuances de couleurs désirées il faut alors avoir recours à la deuxième façon de gérer les couleurs : l'approche technique.

2. Dans l'approche technique, les couleurs sont définies par trois nombres. Il existe deux systèmes techniques pris en charge par NetLogo, le système RGB (initiales anglaises de Red-Green-Blue, Rouge-Vert-Bleu) qui est le système par défaut et le système HSB (initiales anglaises de Hue-Saturation-Brightness, Teinte-Saturation-Luminosité en français). Si, lorsqu'une parcelle est invitée à indiquer sa couleur, cette dernière figure dans la liste des 16 couleurs auxquelles est attribué un nombre, la parcelle rapportera le nombre correspondant. Dans le cas contraire elle rapportera la liste des trois nombres du système RGB.

Par exemple, avec la figure précédente chargée avec la primitive «import-pcolors-rgb», après la commande «ask turtle 0 [type pcolor]» (qui demande la couleur de la parcelle sur laquelle se trouve la tortue 0) la réponse obtenue est [107 123 124]. Ensuite, nous pourrions, par exemple, formuler la commande «ask patches with [pcolor = [107 123 124]] [set pcolor yellow]» qui colorierait en jaune toutes les parcelles avec la couleur RGB susmentionnée ([107 123 124]). Il convient cependant de noter que dans l'ensemble des régions d'un paysage, il existe de nombreuses nuances différentes de vert ou de toute autre couleur, ce qui signifie que très peu de parcelles auront exactement les mêmes trois nombres RGB.

Annexe A: Aspects de NetLogo non abordés dans le livre

Comme indiqué dans l'introduction, le but de ce livre n'est que de fournir une introduction à l'environnement de programmation NetLogo en mettant l'accent sur son langage. Les lecteurs doivent savoir qu'il existe plusieurs aspects de NetLogo qui ne sont pas abordés dans ce livre. Ainsi, parmi les aspects non couverts, il y a un bon nombre de primitives ainsi que certains sujets liés à l'interface et aux diverses extensions du langage. Les lectrices et lecteurs intéréssés à approfondir leurs connaissances et leur maîtrise du langage NetLogo sont invités à consulter le Manuel de l'utilisateur de NetLogo [22].

En ce qui concerne la modélisation et la simulation multi-agents, le thème n'a été abordé qu'en relation avec les exemples et modèles présentés dans le livre et n'a pas été développé en tant que thème en soi. Pour plus d'informations sur ce thème, nous recommandons le livre de Wilensky et Rand [21], qui traite le sujet de manière très large, tant du point de vue général des environnements pour modélisation et la simulation multi-agents, que du point de vue particulier de NetLogo. Les lecteurs et lectrices peuvent également consulter l'article de Manzo [12]. Vous trouverez des informations sur tous les autres aspects mentionnés ci-dessous dans le Manuel déjà cité. Certaines des fonctionnalités NetLogo supplémentaires non mentionnées dans le Manuel sont disponibles dans le menu Outils, tandis que d'autres doivent être chargées en tant qu'extensions. Ainsi, dans le menu Outils, il est possible d'accéder à:

HubNet

Selon le Manuel de l'utilisateur, «HubNet est une technologie qui utilise NetLogo pour exécuter des simulations participatives en classe. Dans une simulation participative, le comportement d'un système est déterminé par l'ensemble de la classe, chaque étudiant contrôlant une partie du système à l'aide d'un périphérique individuel, tel qu'un ordinateur connecté en réseau».

BehaviorSpace

Selon le Manuel de l'utilisateur «BehaviorSpace est un outil logiciel intégré à NetLogo, qui permet d'expérimenter avec les modèles. BehaviorSpace exécute un modèle plusieurs fois, en modifiant systématiquement les paramètres du modèle et en enregistrant les résultats de chaque exécution. Ce processus est parfois appelé «balayage de paramètre». Il permet d'explorer «l'espace» des comportements possibles du modèle et de déterminer les combinaisons de paramètres à l'origine des résultats intéressants du modèle. Si votre ordinateur possède un processeur à cœurs multiples, par défaut l'exécution du modèle se fera en parallèle, une par cœur.

Dans le menu «Tools» (Outils), se trouvent également:

- Un **éditeur de formes** qui permet de changer l'apparence des tortues.
- Un **autre éditeur de formes** qui permet de modifier l'apparence des liens.
- Une **vitrine des couleurs** disponibles.
- Des **moniteurs** qui fournissent des informations sur l'état:
  - des variables globales
  - des tortues
  - des parcelles
  - des liens

NetLogo 3D

Permet de construire et de visualiser des modèles en trois dimensions. Il est chargé à partir du menu Outils. Il peut également être chargé directement à partir de sa propre icône indépendante.

Dynamique des systèmes

Permet de construire des modèles basés sur la technique de modélisation mathématique connue sous le nom de Dynamique des systèmes [19]. On peut y accéder à partir du menu Outils.

Lien avec Mathematica

Permet de charger et d'exécuter des modèles NetLogo à partir de ce puissant logiciel. Ce lien s'installe à partir de Mathematica[59]

Les extensions

Les extensions sont des ensembles d'outils qui ajoutent des fonctionnalités au programme NetLogo. Les extensions sont chargées avec la primitive «extensions []», dans laquelle on inclut les extensions désirées entre les crochets et séparées par des espaces. Par exemple, pour charger les extensions «sound» et «csv», on écrit: «extensions [sound csv]». La commande de chargement des extensions doit être écrite au début, avant les procédures.

[59] On peut trouver des précisions sur l'installation de ce lien sur Internet (par exemple : https://github.com/NetLogo/Mathematica-Link/blob/master/NetLogo-Mathematica%20Tutorial.pdf lien consulté le 3 Novembre 2019)

La liste ci-dessous décrit brièvement les extensions qui, selon l'auteur, sont les plus importantes. Pour plus d'informations sur les extensions, les lecteurs et lectrices sont invités à consulter le Manuel de l'utilisateur [22].

Array («Tableau»[60])

Permet de construire et de manipuler des tableaux de données, tels que des matrices numériques

Arduino

Permet de se connecter avec une carte électronique Arduino, pour apprendre et créer des projets en électronique.

Bitmap

Permet de manipuler et d'importer des images dans les parcelles et les dessins. Contient des primitives non incluses dans l'ensemble de base des primitives NetLogo.

CSV

Permet de manipuler de gros fichiers de type CSV (valeurs séparées par des virgules).

GIS

Selon le Manuel de l'utilisateur: «Cette extension ajoute la prise en charge de SIG (Système d'Information Géographique) par NetLogo. Elle permet d'incorporer des données vectorielles SIG (points, lignes et polygones) et des données raster (grille ou ensemble de cellules) SIG dans votre modèle».

Matrix

Ajoute les matrices comme structure de données utilisables dans un modèle.

Network

Ajoute des outils pour construire et manipuler des réseaux (graphes).

Palette

Augmente la capacité de NetLogo pour travailler avec les couleurs.

R

Contient des primitives pour travailler avec les applications statistiques de R dans NetLogo [6].

Sound

Permet de travailler avec des sons MIDI et des sons préenregistrés.

---

[60] En informatique le mot «Tableau» est une traduction approximative du mot anglais «Array» (originaire du vieux français «Aroi» qui signifie arrangement, disposition).

Table

Offre la possibilité de travailler avec des tableaux.

Vid

Permet de connecter une source vidéo pour l'intégrer à des projets ou à des modèles.

Annexe B : Liste des primitives présentées dans le livre

L'annexe présente une liste des primitives utilisées dans les exemples, modèles et explications du livre. Chaque primitive est accompagnée d'une brève traduction (sauf lorsque la traduction est évidente) et d'une courte description de ce que fait la primitive. Dans de nombreux cas, les descriptions n'offrent pas une explication complète de la fonction remplie par la primitive et l'on recommande de consulter le dictionnaire de primitives inclus dans le Manuel de l'utilisateur. Les primitives peuvent être utilisées avec la première lettre en majuscule ou en minuscule, car NetLogo est insensible à cet aspect.

Liste des primitives de A à Z

A

Abs. Valeur absolue d'un nombre.
And. Conjonction «et» (connective logique).
Any? Certain(e)s ? (rapporte «vrai» si l'ensemble-agents n'est pas vide, «faux» autrement).
Ask. Demander (à un agent ou à un ensemble d'agents d'exécuter certaines commandes).

B

Beep. Bip (produit un son court).
Blue. Bleu (rapporte la couleur bleue, # 105).
Breed. Familles (utilisé pour créer des groupes d'agents appelés «familles» - breeds-).
Butfirst. Moins-le-premier. Rapporte une liste sans son premier membre.
Butlast. Moins-le-dernier. Rapporte une liste sans son dernier membre.

C

Ca. Abréviation de «clear-all».
Clear-all. Effacer-tout (supprime les tortues et efface les traces et les variables).
Clear-drawing. Effacer-dessin (efface les traces laissées par les tortues).
Color. Couleur (rapporte la couleur de la tortue).
Count. Compte (rapporte le nombre d'agents composant un ensemble-agents).
Create. Créer (utilisé pour créer des agents).

Create-link-from, create-links-from. Créer un lien à partir de, créer des liens à partir de.
Create-link-to, **create-links-to**. Créer un lien vers, créer des liens vers.
Create-link-with, **create-links-with**. Créer un lien avec, créer des liens avec.

D

Die. Mourir (utilisé pour éliminer les tortues).
Distance. Distance (rapporte la distance entre deux agents).

E

End. Fin (rapporte la fin d'une procédure).
Export-interface, exporter-interface (enregistre l'interface dans un fichier au format PNG).
Export-plot, exporter-graphique (enregistre le graphique dans un fichier).
Export-plots, exporter-graphiques (enregistre tous les graphiques dans un seul fichier).
Export-view, exporter-vue (enregistre la vue dans un fichier au format PNG).
Export-world, exporter-monde (enregistre l'état du monde dans un fichier).
Extensions (utilisé pour charger une extension du langage NetLogo)

F

Face. S'orienter en direction d'un agent.
File-close Fermer-fichier (ferme un fichier précédemment ouvert).
File-open. Ouvrir-fichier (ouvre un fichier précédemment créé).
File-print. Imprimer-fichier (envoie la sortie vers un fichier précédemment ouvert). Les primitives **file-type**, **file-show** et **file-write** fonctionnent de la même manière, avec des différences mineures.
First. Premier (rapporte le premier membre d'une liste ou d'une chaîne).
Filter. Filtre les membres d'une liste selon qu'ils remplissent une condition.
Foreach. (pour-chacun). Applique une opération à chaque membre d'une liste.
Forward. En avant (utilisé pour avancer).
Fput. Abréviation de «first-put» (utilisé pour placer un membre d'une liste en premier dans cette liste).

G

Globals. Globales (utilisé pour déclarer les variables globales).
Gray. Gris (rapporte la couleur gris, # 5).
Green. Vert (rapporte la couleur vert, # 55).

H

Heading. Orientation (rapporte la direction dans laquelle la tortue regarde).
Hide-link. Cacher-lien

I

If. Si (utilisé pour créer des expressions conditionnelles avec une seule alternative).
Ifelse. Si-sinon (utilisé pour créer des expressions conditionnelles avec deux alternatives)
Import-drawing. importer-dessin (télécharge un dessin dans le calque de dessin du monde).
Import-pcolors, importez une image sur le monde. (voir le manuel [16]).
Import-pcolors-rgb, les cellules acquièrent les couleurs d'une image (voir Manuel [16]).
Import-world, import-world (charge un fichier avec les données du monde).
Int. Abréviation de «nombre entier» (rapporte la partie entière d'un nombre).
In-radius. Dans-un-rayon-de (rapporte les agents inclus dans un cercle de rayon donné).
Is-number? Est-ce-un-nombre? (rapporte «true» -vrai ou «false» - faux- selon que l'entrée est un nombre ou non).
Item. Élément (rapporte un membre d'une liste ou d'une chaîne).

L

Last. Last (rapporte le dernier membre d'une liste ou d'une chaîne).
Length. Longueur (rapporte le nombre de membres d'une liste ou d'une chaîne).
Left. Gauche (utilisé pour faire tourner la tortue vers la gauche).
Let. Attribue une valeur aux variables locales.
Links. Rapporte l'ensemble-agents des liens.
Link-length. Rapporte la longueur du lien.
List. Liste (utilisé pour créer des listes).
Lput. Abréviation de «last-put» (utilisé pour placer un membre de la liste en dernier dans une liste).
Lt. Abréviation de «gauche».

M

Map. Applique une opération aux membres d'une ou plusieurs listes.
Max. Indique le maximum d'une liste de valeurs.
Max-one-of. Rapporte l'agent qui possède la valeur maximale d'une quantité.

Max-pxcor. Rapporte la valeur maximale de la coordonnée x des parcelles du monde.
Max-pycor. Rapporte la valeur maximale de la coordonnée y des parcelles du monde.
Mean. Moyenne (rapporte la moyenne d'un ensemble de valeurs).
Myself. La tortue, le patch ou le lien qui m'a demandé de faire ce que je suis en train de faire.

P

Penup. Stylet-vers-le-haut (utilisé pour désactiver le stylet de la tortue).
Pendown. Stylet-vers-le-bas (utilisé pour activer le stylet de la tortue).
Plabel. Etiquette-de-parcelle (rapporte l'étiquette d'une parcelle donnée).
Play-note. Joue-note (joue la note indiquée dans l'extension «son»).
Print. Imprime (écrit le résultat dans le terminal de commande ou une autre sortie).
Pd. Abréviation de «pendown».
Pu. Abréviation de «penup».
Pxcor. Rapporte la coordonnée x du centre de la parcelle sur laquelle se trouve la tortue.
Pycor. Rapporte la coordonnée y du centre de la parcelle sur laquelle se trouve la tortue.

R

Random. Aléatoire (génère et rapporte des nombres aléatoires).
Random-seed. Graine-aléatoire (génère une nouvelle graine aléatoire).
Random-xcor. Rapporte la coordonnée x au hasard.
Random-ycor. Rapporte la coordonnée y au hasard.
Red. Rouge (indique la couleur rouge, # 15).
Read-from-string. Lire-la-chaîne (rapporte la chaîne en question comme si elle avait été écrite dans la fenêtre de l'observateur, c'est-à-dire avec le type de données qui correspond à ce qui a été entré dans la fenêtre).
Reduce. Applique un processus de réduction aux membres d'une liste.
Remainder. Reste (indique le reste d'une division entre nombres entiers).
Remove. Supprime (supprime un membre d'une liste ou d'une chaîne).
Repeat. Répéter (utilisé pour répéter un ensemble de commandes).
Report. Rapporte (utilisé en conjonction avec «to-report» pour rapporter un résultat).
Reset-ticks. Réinitialise la variable ticks à la valeur 0.
Right. Droite (utilisé pour tourner à droite).
Run. Exécute (exécute un bloc d'instructions donné).

S

Self. Moi-même (utilisé dans la construction des commandes composées).
Sentence. Phrase (fusionne deux listes ou chaînes en une seule liste).
Se. Abréviation de la primitive «sentence».
Set. Assigner (utilisé pour attribuer des valeurs aux variables).
Setxy. Assignerxy (utilisé pour attribuer des coordonnées de position à une tortue).
Show. Montrer (écrit le résultat dans le terminal de commande ou tout autre sortie).
Sort-by. Trier ou trier selon (rapporte une liste ordonnée selon un critère donné).
Sound. Son (Active l'extension des sons).
Sprout. Faire pousser, donner naissance à (utilisé pour faire germer des tortues dans une parcelle).
Sqrt. Indique la racine carrée d'un nombre.
Stamp. Estampille (la tortue laisse une empreinte d'elle même).
Standard-deviation. Rapporte l'écart type d'un ensemble de valeurs.
Stop. Arrêter (arrête un bloc d'instructions ou une procédure).
Sum. Rapporte la somme d'une liste de nombres.

T

Thickness. Épaisseur (rapporte l'épaisseur d'un lien).
Tick. Augmente la valeur de la variable ticks.
Ticks. Rapporte la valeur de la variable «ticks» préinstallée dans le système.
Tie. Attache, relie (crée un lien entre deux tortues).
To. Préposition utilisée pour nommer une procédure.
To-report (utilisé pour créer des procédures qui fournissent une information).
Turtle, turtles. Tortue, tortues.
Turtles-à (rapporte les tortues sur la parcelle située à une distance dx, dy de la parcelle actuelle).
Turtles-here. Tortues-ici (rapporte les tortues situées sur cette parcelle).
Turtles-on. Tortues-sur (rapporte les tortues situées sur une parcelle ou sur un ensemble de parcelles).
Turtles-own. Propre-aux tortues (utilisé pour déclarer les variables propres aux tortues).
Type. Ecrire (écrit le résultat dans le terminal de commande ou une autre sortie).

U

User-input. Entrée-utilisateur (rapporte l'entrée saisie par l'utilisateur).

X

Xcor. Rapporte la coordonnée x de la tortue

Y

Ycor. Rapporte la coordonnée y de la tortue.
Yellow. Jaune (rapporte la couleur jaune, # 45).

W

Wait. Attendre (retarde les actions pendant le nombre de secondes spécifiées).
While. Tandis que (un bloc de commandes est répété pendant qu'une condition est remplie).
White. Blanc (rapporte la couleur blanc, # 9.9).
With. Préposition «avec» (utilisé pour construire des commandes composées).
With-max. Avec-le-maximum (utilisé pour rapporter les agents dont l'une des caractéristiques a une valeur maximale).
With-min. Avec le minimum (utilisé pour rapporter les agents dont l'une des caractéristiques a une valeur minimale).
Who. Qui (rapporte le numéro qui identifie une tortue spécifique).
Word. Mot (rapporte la liste ou la chaîne formée par l'union de deux listes ou chaînes).
Write. Ecrire (écrire un résultat dans le terminal d'instructions ou dans une autre sortie).

Les signes (mathématiques ou autres)

+. Signe d'addition.
-. Signe de soustraction.
*. Signe de multiplication.
/. Signe de division.
". Guillemet anglais (utilisé pour définir des chaînes).
=. Signe «égal à».
¡=. Signe «différent de».
<. Signe «inférieur à».
<=. Signer «inférieur ou égal à».
>. Signe «supérieur à».

> =. Signe «supérieur ou égal à».
[]. Crochets (utilisés pour créer des listes).
(). Parenthèses pour regrouper des expressions.
\ r \ n. Force un retour de chariot (changement de ligne forcé).

Références

1. Abelson H., Sussman G. J., Sussman J., (1985), Structure and Interpretation of Computer Programs, Cambridge, Mass., The MIT Press. https://mitpress.mit.edu/sites/default/files/6515.pdf
2. Bodin, A., Arsicaud, L.,Bernard, N., Recher, F. (2017), Scratch au college, Exo.7 (http://exo7.emath.fr/cours/livre-scratch.pdf)
3. Banos, A., Lang, C. et Marilleau, N. (2015), Simulation spatiale à base d'agents avec NetLogo 1: Introduction et bases, ISTE Éditions.
4. Banos, A., Lang, C. et Marilleau, N. (2017), Simulation spatiale à base d'agents avec NetLogo 2: Notions avancées, ISTE Éditions.
5. Dealhaye, J.P., (2013) Merveilleux nombres premiers, Belin, Paris.
6. De Vries A., Meys J., (2012), R for dummies, John Wiley & Sons, Ltd., England. (http://sgpwe.izt.uam.mx/files/users/uami/gma/R_for_dummies.pdf)
7. Distel R. Graph theory, 3rd ed., (2006) Springer. Il existe une version électronique plus ancienne (2000) de cet ouvrage sur le site: http://www.esi2.us.es/~mbilbao/pdffiles/DiestelGT.pdf
8. García Vázquez J. C., Sancho Caparrini F. (sans date, versions espganole et anglaise), NetLogo: una herramienta de modelado/NetLogo: A modeling Tool. (http://www.cs.us.es/~fsancho/?e=128).
9. Hamilton, A. (1988), Logic for Mathematicians, 2ème ed. Cambridge University Press.
10. Hillis, D. W. (1985), The connection machine, The MIT Press, Boston.
11. Jackson M. O. (2008), Social and economic networks, Princeton University Press.
12. Manzo, G. (2014), La simulation multi-agents : principes et applications aux phénomènes sociaux, Revue française de sociologie, n° 55-4.
13. Marji, M. (2014), Le grand livre de Scratch, Eyrolles, Paris.
14. Papert S. (1980), Jaillissement de l'esprit. Ordinateurs et apprentissage, Flammarion, Paris.
15. Polya G., (1965), Comment poser et résoudre un problème, 2e éd., Eyrolles, Paris.
16. Quesada, F. (2019), Introducción al lenguaje NetLogo y la Programación Basada en Agentes (http://franciscoquesada.com/index.php/netlogo/)
17. Railsback, S.F. et Grimm, V. (2019), Agent-based and Individual-based Modeling: A Practical Introduction, 2nd ed., Princeton University Press.

18. Resnick, M. (1994), Learning about life. Artificial Life, vol. 1, no. 1-2.
19. Salini, P (2017), Introduction à la dynamique des systèmes, L'Harmattan, Paris.
20. Varenne, F. (2010), Les simulations computationnelles dans les sciences sociales. Nouvelles perspectives en sciences sociales, 5 (2), 17–49. https://www.erudit.org/fr/revues/npss/2010-v5-n2-npss3882/044073ar.pdf
21. Wilensky U., Rand W. (2015), An introduction to Agent Based Modeling: modeling natural, social and engineered complex systems with NetLogo. MIT Press, Boston.
22. Wilensky U. (2020), NetLogo User Manual, version 6.2.0: https://ccl.northwestern.edu/netlogo/docs/