



# **Rapport - Projet Programmation Orientée Objet**

Ricardo CARRIZALES  
Pedro ROHDE  
2EN

Novembre, 2018

# 1. Introduction

Dans ce travail, on avait pour but de développer un programme en C++ qui pourrait résoudre numériquement les équations différentielles des circuits électroniques. Dans ce rapport, on essaie de donner une vision générale du fonctionnement du programme, mais tous les détails sont aussi expliqués dans les commentaires du code (en annexe).

## 2. Classes

On a choisi d'avoir trois classes mères principaux: une pour les sources, une pour les EDOs et une pour les circuits. On a essayé de faire la classe des EDOs indépendamment de celle des circuits, pour qu'on pût aussi résoudre des EDOs qui ne sont pas liées à un circuit. On a représenté ces classes mère et leurs filles à l'aide des organigrammes.

### 2.1 Sources

On avait cinq types de source à coder: échelon, rectangulaire, créneau, triangulaire et sinusoïdal. Pour chacune de ces sources on a défini une classe. Les classes **echelon** et **rect** héritent de la classe **non\_periodique** de sources non-périodiques, et **creneau**, **triang** et **sinus** de la classe **periodique**, des sources périodiques.

Les classes **periodique** et **non\_periodique** sont filles de la classe mère général **source**. La différence entre elles est que toutes les sources périodiques ont besoin d'un paramètre **T**, leur période.

Un objet de ces classes est donc une source de tension. Chacun des types de source a besoin de certains attributs - parfois communs à plusieurs types - pour être réalisé:

- l'amplitude du signal, commune à tous les types de source, et donc déclaré en **source**;
- la décalage, commune à tous les types de source, déclaré en **source**;
- la période, exclusive des sources périodiques, déclaré en **periodique**;
- le rapport cyclique, uniquement pour le créneau, déclaré en **creneau**;
- la largeur du signal, uniquement pour la rectangulaire, déclaré en **rect**.

Les sources de tension fournissent une tension à un certain temps, donc, on les a tous attribué une méthode **Ve**, qui reçoit comme argument une valeur de temps et retourne la tension de la source dans ce moment-là. Cette méthode a des différents arguments pour chaque classe de source (d'après la liste juste avant).

Comme on ne peut pas définir une tension de sortie pour les sources des classes **periodique**, **non\_periodique** et **source**, leurs méthodes **Ve** sont purement virtuelles (pas définis), ce qui les rend abstraites. **Ve** sera donc surchargée par les classes filles, qui la définiront selon leurs spécifications.

On a choisi cette implémentation pour les sources au lieu d'une avec des vecteurs en pensant à une application réelle du code dans une machine avec une mémoire limitée et que l'on attend puisse être utilisé pendant une période de temps très large, voire infinie.

## 2.2 EDOs

Dans ce projet, on a deux types de circuits, du premier et second ordre. Ces circuits sont décrits par des équations différentielles du premier et second ordre, respectivement, avec des conditions initiales. On a donc créé deux classes **EDO1** et **EDO2**, filles d'une classe principal **EDO**. Les objets de ces classes sont des équations différentielles ordinaires.

La classe mère abstraite **EDO** a ses constructeurs et destructeur et aussi deux méthodes purement virtuelles: **euler** et **getU**, définis seulement pour les classes filles **EDO1** et **EDO2** (par surchargement). Ses attributs sont **t\_pre**, le temps précédent, et le pas **h**, qu'on utilise pour résoudre les EDOs. Ces attributs peuvent changer à chaque itération de la méthode de résolution, ce qui permet à l'utilisateur d'utiliser un pas variable. On les laisse protégés pour que les classes filles puissent y accéder.

Pour la classe des EDOs du premier ordre ( $u'(t) = f(u(t), t)$ ) **EDO1**, on a deux constructeurs, un avec un seul argument (la condition initiale de l'EDO  $u(0)$ ) et un sans arguments, le constructeur par défaut. Pour cette classe, la méthode **euler** a comme argument la valeur de la dérivée de la fonction dans un instant de temps donnée par l'autre argument (soient les deux arguments  $u'(t_i)$  et  $t_i$ ). La méthode retourne la valeur  $u(t_{i+1})$ , la solution numérique (par méthode d'Euler) de l'EDO au temps  $t_{i+1}$ .

La méthode **getU**, nous donne la dernière valeur calculée de  $u^{(i)}$ , la dérivée  $i$ -ème de  $u$ . Dans le cas des EDOs du premier ordre, on calcule seulement  $u^{(0)}$  (la fonction  $u$ ) et donc **getU** nous donne la dernière valeur de  $u$  pour quel que soit son argument. Cette valeur est l'attribut **U** de la classe **EDO1**.

Pour les EDOs du second ordre ( $u''(t) = f(u'(t), u(t), t)$ ), **EDO2** a aussi un destructeur et deux constructeurs, un sans arguments (défaut) et un dont les arguments sont les conditions initiales  $u'(0)$  et  $u(0)$ . Pour cette classe, la méthode **euler** reçoit  $u''(t_i)$  et  $t_i$ , et, comme avant, retourne  $u(t_{i+1})$ . Comme pour **EDO1**, ici **getU** nous donne  $u^{(i)}$ , où  $i$  est son argument. L'attribut **U[2]** de **EDO2**, est le vecteur  $U = [u(t_i), u'(t_i)]$ .

On a choisi cette implémentation de résolution d'EDO pour qu'elle pût être utilisée aussi dans d'autres applications, pas seulement pour la résolution des circuits. On considère que l'utilisateur aura accès, au fur et à mesure, aux valeurs des dérivées première ou seconde et qu'on lui donnera les valeurs de la fonction d'intérêt  $u$  à chaque itération.

## 2.3 Circuits

On avait quatre circuits à résoudre: deux du premier ordre (A et C) et deux du second ordre (B et D). On a défini donc une classe mère **general circuit**, ses deux filles **circuit\_1** et **circuit\_2**, soit circuits du premier et du second ordre, et, finalement, les classes **circuitA** et **circuitC**, filles de **circuit\_1**, et **circuitB** et **circuitD**, filles de **circuit\_2**. Les objets de ces classes sont des circuits électroniques.

La classe mère **circuit** a ses constructeurs et destructeur et aussi deux autres méthodes: une purement virtuelle, qui la rend abstraite, et, contrairement aux classes mère précédentes, une qui est bien défini. La méthode virtuelle **Vout** sera utilisé pour calculer la tension de sortie du circuit. Comme **circuit** est une classe générale, on ne peut pas définir la sortie de ses objets. Par contre, la méthode **getVg** sert à obtenir la tension des sources des circuits, ce qui est fait de la même façon pour tous les circuits. On la définit donc pour la classe mère et toutes ses classes filles iront simplement l'hériter.

Les sous-classes de **circuit** iront aussi hériter ses attributs protégés **Vg**, **Vs** et **Vsp** - la tension de la source, celle de sortie et sa dérivée - et aussi **src** et **eq\_diff**. **src** sera la source du circuit, un objet de la classe **source**. **eq\_diff** sera l'EDO du circuit, un objet de **EDO**. Comme on ne sait pas d'abord quelle sera la source (quelle sous-classe de **source** on aura besoin) et de quel ordre sera l'EDO (quelle sous-classe de **EDO**), on utilise du polymorphisme pour rendre le code plus générique et pour pouvoir traiter tous les cas. La source utilisé sera un argument du constructeur du circuit et l'ordre de l'EDO va dépendre du circuit choisi.

Les classes **circuit\_1** et **circuit\_2**, filles de **circuit**, sont aussi abstraites, puisque on ne peut pas définir les sorties de leurs objets (**Vout** est encore purement virtuelle). La différence entre les deux classes est l'attribut **Vs2p** de **circuit\_2**, qui représente la dérivée seconde du signal de sortie du circuit.

Dans **circuitA**, **circuitB**, **circuitC** et **circuitD** on fait le surchargement de la méthode **Vout**, qui va donner le signal de sortie du circuit et que l'on définit différemment pour chacune des quatre classes, selon la forme des circuits. Cette méthode prend un seul argument, un instant de temps  $t_i$ , et calcule la valeur du signal de sortie à cet instant. Ce calcul est fait à l'aide des méthodes de résolution d'EDO précédemment vus. À l'aide des attributs des classes, on peut utiliser les valeurs calculées d'une itération dans la prochaine.

À cause de la forme du circuit D, on a besoin de calculer la dérivée de son signal d'entrée ( $V_e(t)$ ), et donc la classe **circuitD** a trois attributs de plus: **t\_pre** et **Vg\_pre**, le temps de l'itération précédente et la tension de la source dans ce moment ( $t_{i-1}$  et  $V_e(t_{i-1})$ ), et **Vgp**, la valeur de la dérivée première de cette tension, qu'on calcule avec les deux autres attributs et qui sera utilisé pour déterminer le signal de sortie du circuit.

### 3. Programme principal (main)

On avait d'abord fait une implémentation où les classes de circuit n'avaient pas un attribut des classes **source** et **EDO**. Dans ce cas, le programme principal était responsable d'appeler les méthodes de **source** pour obtenir le signal d'entrée, puis passer ce signal à une méthode de **circuit** qui nous donnerait la dérivée première ou seconde du signal de sortie, et cette valeur serait finalement passé à une méthode de **EDO** pour qu'on pût obtenir le signal de sortie.

Dans notre code final, au lieu de faire tous ces trois pas, on a besoin seulement de déclarer d'abord le circuit avec la source qu'on veut et après appeler la méthode **Vout** itérativement pour obtenir le signal de sortie, ce qui rend le code du programme principal beaucoup plus simple, sans changer sa fonctionnalité. C'est la notion d'encapsulation: le circuit est une boîte noire, dont les calculs internes restent cachés pour que l'utilisateur puisse l'utiliser simplement.

Si on pense à un utilisateur qui aura accès aux classes **source**, **EDO** et **circuit** pour construire son propre programme, il pourra utiliser la méthode de calcul du signal de sortie sans avoir nécessairement besoin de savoir comment ce calcul est fait.

Le programme principal a une fonction **menu**, qui demande à l'utilisateur de définir les paramètres de simulation (choix de circuit, source, pas, temps final et fichier de sortie). Cette fonction est appelée dans la première ligne de **main**, et après on initialise le fichier de sortie.

Finalement, on commence la simulation, au temps initial  $t_0 = 0ns$ . On appelle la méthode **Vout** de notre circuit pour avoir le premier point du signal de sortie. À titre informatif, on appelle aussi **getVg** pour savoir la tension de la source à ce moment-là. On sauvegarde ces valeurs dans le fichier de sortie. On incrémente le temps de  $\Delta t$  (le pas) et on sauvegarde les nouvelles valeurs. On répète cela jusqu'à ce qu'on atteigne la valeur finale du temps. Après, il suffit de bien fermer le fichier de sortie.

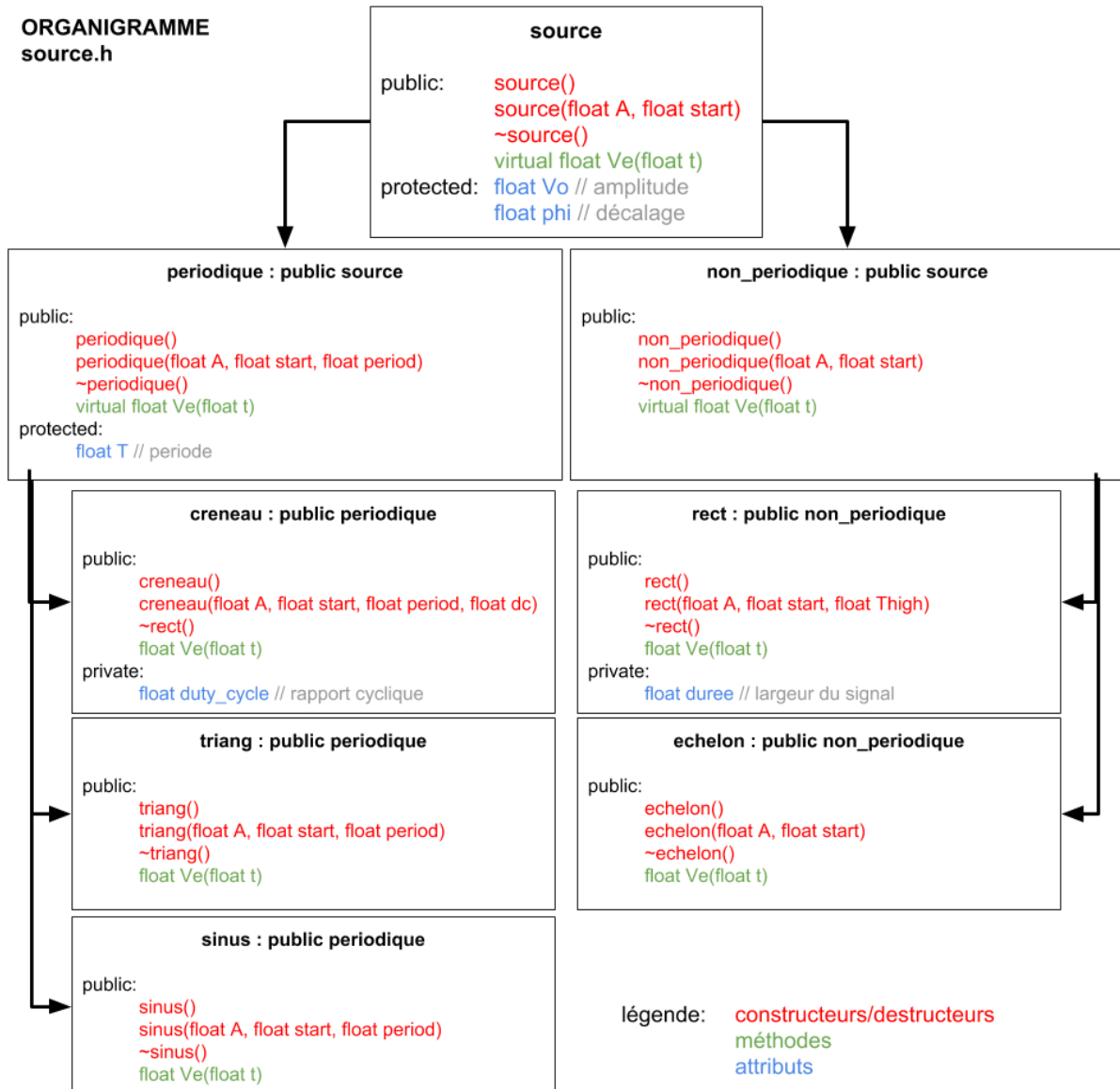
Après la fin de la simulation, les valeurs de chaque instant de temps simulé, et ses respectives valeurs de tension d'entrée et de sortie seront dans le fichier de sortie.

## 4. Conclusion

On a fait dans ce projet un programme de résolution de circuits électroniques, mais pas seulement cela. On a aussi développé un programme qui résout des équations différentielles quelconques, qui ne sont pas nécessairement liées à un circuit. Les résultats obtenus sont bien fidèles à ceux que l'on attendait, pour les circuits et aussi pour la résolution d'autres EDOs (voir annexe). On a aussi codé ce programme de manière qu'il fût le plus simple possible à utiliser par l'utilisateur.

Dans les prochaines pages, les annexes avec les organigrammes, le code complet commenté, quelques simulations et des captures d'écran de l'interface du programme.

## ORGANIGRAMME source.h

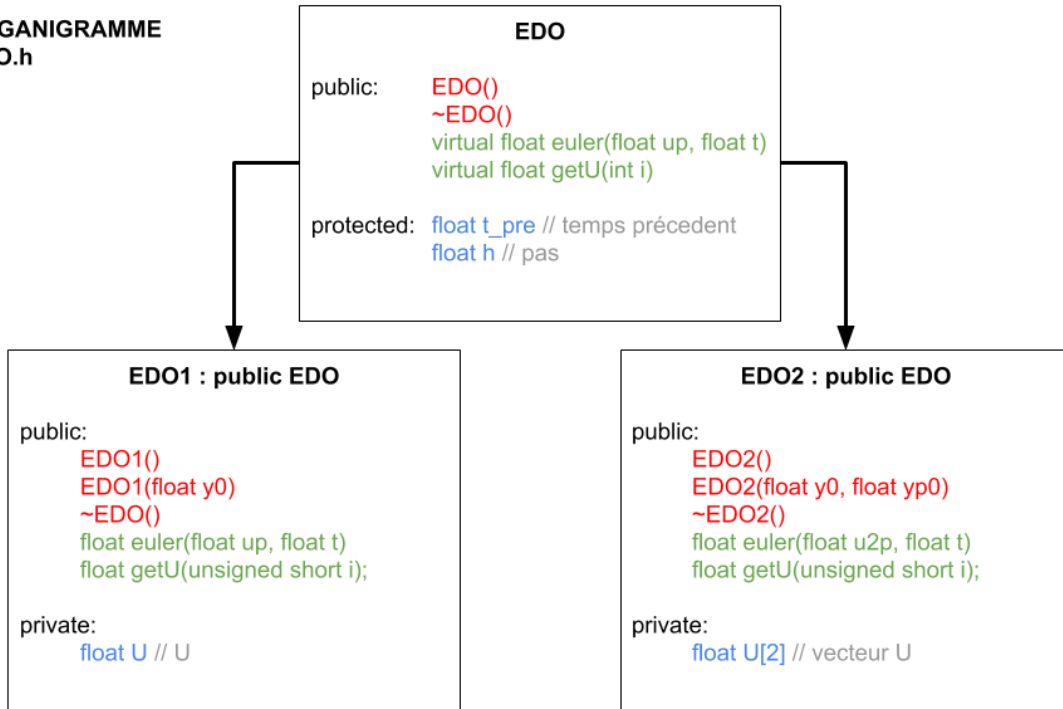


Pour les sources, on a la classe mère **source**, ses deux filles **periodique** et **non\_periodique** et leurs filles. Comme toutes les types de sources ont une amplitude et un décalage, ces attributs sont définis déjà dans la classe mère. Seulement les sources périodiques ont besoin d'une période, donc on déclare cet attribut seulement dans **periodique** et ses filles l'héritent. La source créneau est la seule à avoir un attribut pour le rapport cyclique, bien comme la source rectangulaire est la seule à avoir besoin d'un attribut pour la durée du signal.

Comme règle général, on a défini tous les attributs comme privés, pour les protéger d'une mauvaise utilisation de l'utilisateur. Pour que les classes filles peuvent encore hériter des attributs de ses mères, on déclare ces attributs comme protégés dans le classes de niveau plus haut. Toutes les méthodes, constructeurs et destructeurs sont publiques pour permettre leur utilisation dans le programme principal.

La méthode **Ve** est déclaré comme purement virtuelle dans **source** pour que ses filles puissent la surcharger.

# ORGANIGRAMME EDO.h



légende: constructeurs/destructeurs  
méthodes  
attributs

Pour les équations différentielles, on a la classe mère **EDO** et ses deux filles **EDO1**, pour les EDOs du premier ordre, et **EDO2**, pour les EDOs du seconde ordre. La classe mère a deux attributs: temps précédent et pas. Ces deux attributs sont déclarés déjà ici parce que toutes les filles en auront besoin. **EDO1** a un attribut **U** pour la solution de l'EDO, mais dans **EDO2** cet attribut est un vecteur de taille 2. On laisse tous les attributs de la classe mère comme protégés et ceux des filles comme privés, comme pour **source**.

Les méthodes **euler** et **getU** ne peuvent pas être définis en source, et donc elles sont déclarées comme purement virtuelles pour que **EDO1** et **EDO2** les surchargent. Toutes les méthodes, constructeurs et destructeurs sont publiques pour permettre leur utilisation dans le programme principal.



# ORGANIGRAMME circuit.h



Pour les circuits, on a la classe mère **circuit**, ses deux filles **circuit\_1** (premier ordre) et **circuit\_2** (second ordre) et leurs filles. Comme chaque circuit de premier ou second ordre peut avoir des composants différents, on ne déclare pas des attributs pour des composants ni en **circuit\_1**, ni en **circuit\_2**, ni en **circuit**. Par contre, tous les circuits ont en commun ces attributs que l'on déclare déjà dans **circuit**: la tension de la source, le signal de sortie, sa dérivée, la source et l'EDO qui les décrit. Les circuits du second ordre utilisent aussi la dérivée seconde du signal de sortie, attribut déclarée en **circuit\_2**.

Après, les circuits A, B, C, D héritent les attributs de **circuit** et de **circuit\_1** (A et C) ou **circuit\_2** (B et D). Pour chacun de ces quatre classes de circuit, on déclare aussi les attributs qui représentent ses composants. Pour le cas spécial du **circuitD**, on a aussi d'autres attributs qui nous aident à calculer le signal de sortie de ce circuit (à partir de son EDO).

Comme on ne peut pas définir le signal de sortie pour **circuit**, **circuit\_1** et **circuit\_2**, la méthode **Vout** est purement virtuelle dans ces classes et sera surchargée par leurs filles. Par contre, la méthode **getVg** est la même pour tous les circuits, et on la déclare seulement pour la classe mère, et les autres classes l'héritent.

On déclare tous les attributs des classes mères (**circuit**, **circuit\_1** et **circuit\_2**) comme protégés pour que leurs filles puissent y accéder. Pour les sous-classes, les attributs sont tous privés, comme avant. Toutes les méthodes sont publiques pour qu'elles puissent être utilisées par l'utilisateur.

# Annexe: main.cpp

```
#include "circuit.h"
#include <cstring>

void menu(); // déclaration de la fonction menu

// déclarations de variables qui seront utilisées dans ce fichier:
static char source_type, circuit_type;
static float t, deltat = 0.1*NANO, tf = 500*NANO;
static float y, Ve, u0, up0, R1, R2, C1, L1, A, phi, duty, T, f;
static char nom_fichier[20] = "out.txt";

static source *gen; // on définira la source après
static circuit *circ; // on définira le circuit après

main(){

    menu(); // menu pour définir les paramètres de simulation

    // quelques exemples de déclaration de sources:
    ///// SOURCES /////
    //gen = new echelon(1,1*NANO);
    //gen = new echelon();
    //gen = new creneau(1,0,100*NANO,0.5);
    //gen = new creneau();
    //gen = new rect(1,50*NANO,100*NANO);
    //gen = new rect();
    //gen = new sinus(1,0,100*NANO);
    //gen = new sinus();

    // quelques exemples de déclaration de circuits:
    ///// CIRCUITS /////
    //circ = new circuitA(gen,0,50,1*NANO);
    //circ = new circuitA(gen);
    //circ = new circuitB(gen,0,0,120,0.1*NANO,1*MICRO);
    //circ = new circuitB(gen);
    //circ = new circuitC(gen,0,36,1*NANO,180);
    //circ = new circuitC(gen);
    //circ = new circuitD(gen,0,0,1.2,0.1*MICRO,10*NANO);
    //circ = new circuitD(gen);

    // on initialise le fichier de sortie:
    std::ofstream outfile;
    outfile.open(nom_fichier);
    outfile << "circuit " << circuit_type << "\nt\tVe\tVs" << std::endl;

    // simulation de t = 0 à tf (temps final),
    // avec un pas deltat
    for(t = 0; t < tf; t+=deltat) {
        // ici, on utilise du polymorphisme
        // les méthodes Vout et getVg dépendent du type de circuit
        // la méthode correspondante de ce type de circuit va être automatiquement appelée
        y = (*circ).Vout(t); // on calcule la tension de sortie du circuit
                           // à chaque instant de temps
        Ve = (*circ).getVg(); // on prend aussi la tension de la source
        // on les écrit dans le fichier de sortie:
        outfile << t << "\t" << Ve << "\t" << y << std::endl;

    }
    outfile.close(); // fermeture du fichier de sortie
    return 0; // fin du programme
}
```

```

void menu(){
    // ce menu va permettre à l'utilisateur de choisir les paramètres de la simulation
    // il peut prendre les valeurs standard (ceux des exemples) ou choisir d'autres
    // ses choix vont être sauvegardés dans les variables déclarées au début de ce fichier
    // notamment, l'utilisateur choisira une source et un circuit
    char c;
    std::cout << "\n\nChoisissez un circuit:\n";
    std::cout << "A, B, C ou D\n";
    std::cin >> circuit_type;
    std::cout << "\n\nChoisissez une source:\n";
    std::cout << "rectangulaire(R)\ncreneau (C)\nechelon(E)\n";
    std::cout << "triangulaire(T)\nsinusoidal(S)\n\nVotre choix: ";
    std::cin >> source_type;
    std::cout << "\nUtiliser source comme dans les exemples? (o/n) ";
    std::cin >> c;
    switch(source_type){
        case 'r': case 'R':
            if(c == 'n' || c == 'N'){
                std::cout << "Amplitude: "; std::cin >> A;
                std::cout << "phi: "; std::cin >> phi;
                std::cout << "Duree: "; std::cin >> duty;
                gen = new rect(A,phi,duty);} // allocation dynamique
            else gen = new rect(); // allocation dynamique, constructeur défaut
            break;
        case 'c': case 'C':
            if(c == 'n' || c == 'N'){
                std::cout << "Amplitude: "; std::cin >> A;
                std::cout << "phi: "; std::cin >> phi;
                std::cout << "Period: "; std::cin >> T;
                std::cout << "Duty cycle: "; std::cin >> duty;
                gen = new creneau(A,phi,T,duty);} // allocation dynamique
            else gen = new creneau(); // allocation dynamique, constructeur défaut
            break;
        case 'e': case 'E':
            if(c == 'n' || c == 'N'){
                std::cout << "Amplitude: "; std::cin >> A;
                std::cout << "phi: "; std::cin >> phi;
                gen = new echelon(A,phi);} // allocation dynamique
            else gen = new echelon(); // allocation dynamique, constructeur défaut
            break;
        case 't': case 'T':
            if(c == 'n' || c == 'N'){
                std::cout << "Amplitude: "; std::cin >> A;
                std::cout << "phi: "; std::cin >> phi;
                std::cout << "Period: "; std::cin >> T;
                gen = new triang(A,phi,T);} // allocation dynamique
            else gen = new triang(); // allocation dynamique, constructeur défaut
            break;
        case 's': case 'S':
            if(c == 'n' || c == 'N'){
                std::cout << "Amplitude: "; std::cin >> A;
                std::cout << "phi: "; std::cin >> phi;
                std::cout << "Frequence: "; std::cin >> f; T = 1.0/f;
                gen = new sinus(A,phi,T);} // allocation dynamique
            else gen = new sinus(); // allocation dynamique, constructeur défaut
            break;
        default: break;};

    std::cout << "\nUtiliser circuit comme dans les exemples? (o/n) "; std::cin >> c;
    switch(circuit_type){
        case 'a': case 'A':
            if(c == 'n' || c == 'N'){
                std::cout << "condition initiale u(0) = "; std::cin >> u0;
                std::cout << "R1 = "; std::cin >> R1;
                std::cout << "C1 = "; std::cin >> C1;
                circ = new circuitA(gen,u0,R1,C1);} // allocation dynamique
            else circ = new circuitA(gen); // allocation dynamique
            break;
        case 'b': case 'B':
            if(c == 'n' || c == 'N'){

```

```

        std::cout << "condition initiale u(0) = "; std::cin >> u0;
        std::cout << "condition initiale u'(0) = "; std::cin >> up0;
        std::cout << "R1 = "; std::cin >> R1;
        std::cout << "L1 = "; std::cin >> L1;
        std::cout << "C1 = "; std::cin >> C1;
        circ = new circuitB(gen,u0,up0,R1,C1,L1);} // allocation dynamique
    else circ = new circuitB(gen); // allocation dynamique
    break;
case 'c': case 'C':
    if(c == 'n' || c == 'N'){
        std::cout << "condition initiale u(0) = "; std::cin >> u0;
        std::cout << "R1 = "; std::cin >> R1;
        std::cout << "R2 = "; std::cin >> R2;
        std::cout << "C1 = "; std::cin >> C1;
        circ = new circuitC(gen,u0,R1,C1,R2);} // allocation dynamique
    else circ = new circuitC(gen); // allocation dynamique
    break;
case 'd': case 'D':
    if(c == 'n' || c == 'N'){
        std::cout << "condition initiale u(0) = "; std::cin >> u0;
        std::cout << "condition initiale u'(0) = "; std::cin >> up0;
        std::cout << "R1 = "; std::cin >> R1;
        std::cout << "L1 = "; std::cin >> L1;
        std::cout << "C1 = "; std::cin >> C1;
        circ = new circuitD(gen,u0,up0,R1,C1,L1);} // allocation dynamique
    else circ = new circuitD(gen); // allocation dynamique
    break;
default: break;};

std::cout << "Parametres de simulation:\n";
std::cout << "Pas (delta t) standard? (o/n) "; std::cin >> c;
    if(c == 'n' || c == 'N'){
        std::cout << "delta t = "; std::cin >> deltat;
    }
    std::cout << "Temps final standard? (o/n) "; std::cin >> c;
    if(c == 'n' || c == 'N'){
        std::cout << "tf = "; std::cin >> tf;
    }
    std::cout << "Fichier de sortie standard? (o/n) "; std::cin >> c;
    if(c == 'n' || c == 'N'){
        std::cout << "nom du fichier: "; std::cin >> nom_fichier;}}

```

# Annexe: source.h

```
#include <iostream>
#include <math.h>
#include <fstream>
// quelques constantes utiles:
#define MILLI 0.001
#define MICRO 0.000001
#define NANO 0.000000001
#define PICO 0.00000000001

// classes des sources:

// classe mère générale
class source{

public:
    source(); // constructeur défaut
    source(float A, float start); // constructeur
    ~source(); // destructeur
    // méthode:
    virtual float Ve(float t) = 0; // pour calculer la tension de la source
    // méthode est purement virtuelle => classe abstraite
    // cette méthode sera surchargée par les filles
protected: // attributs privés qui peuvent être accédés par les filles
    // toutes les filles en auront besoin
    float Vo, phi; // l'amplitude et la décalage
};

// classe des sources périodiques:
class periodique : public source{ // classe fille qui hérite de source
public:
    periodique(); // constructeur défaut
    periodique(float A, float start, float period); // constructeur
    // arguments: amplitude, décalage et période
    ~periodique(); // destructeur
    // méthode:
    virtual float Ve(float t) = 0; // pour calculer la tension de la source
    // méthode est purement virtuelle => classe abstraite
    // cette méthode sera surchargée par les filles
protected: // attribut privé qui peut être accédé par les filles
    // toutes les filles en auront besoin
    float T; // période
};

// classe des sources non-périodiques:
class non_periodique : public source{ // classe fille qui hérite de source
public:
    non_periodique(); // constructeur défaut
    non_periodique(float A, float start); // constructeur
    // arguments: amplitude, décalage
    ~non_periodique(); // destructeur
    // méthode:
    virtual float Ve(float t) = 0; // pour calculer la tension de la source
    // méthode est purement virtuelle => classe abstraite
    // cette méthode sera surchargée par les filles
};

// classe des sources rectangulaires:
class rect : public non_periodique{ // classe fille qui hérite de non_periodique
public:
    rect(); // constructeur défaut
    rect(float A, float start, float Thigh); // constructeur
    // arguments: amplitude, décalage, durée du signal
    ~rect(); // destructeur
    // méthode surchargée:
```

```

        float Ve(float t); // pour calculer la tension de la source
private: // attribut privé
    float duree; // durée du signal
};

// classe des sources créneaux:
class creneau : public periodique{ // classe fille qui hérite de periodique
public:
    creneau(); // constructeur défaut
    creneau(float A, float start, float period, float dc); // constructeur
    // arguments: amplitude, décalage, période, rapport cyclique
    ~creneau(); // destructeur
    // méthode surchargée:
    float Ve(float t); // pour calculer la tension de la source
private: // attribut privé
    float duty_cycle; // rapport cyclique du signal
};

// classe des sources échelon:
class echelon : public non_periodique{ // classe fille qui hérite de non_periodique
public:
    echelon(); // constructeur défaut
    echelon(float A, float start); // constructeur
    // arguments: amplitude, décalage
    ~echelon(); // destructeur
    // méthode surchargée:
    float Ve(float t); // pour calculer la tension de la source
};

// classes des sources triangulaires:
class triang : public periodique{ // classe fille qui hérite de periodique
public:
    triang(); // constructeur défaut
    triang(float A, float start, float period); // constructeur
    // arguments: amplitude, décalage, période
    ~triang(); // destructeur
    // méthode surchargée:
    float Ve(float t); // pour calculer la tension de la source
};

// classes des sources sinusoïdales:
class sinus : public periodique{ // classe fille qui hérite de periodique
public:
    sinus(); // constructeur défaut
    sinus(float A, float start, float period); // constructeur
    // arguments: amplitude, décalage, période
    ~sinus(); // destructeur
    // méthode surchargée:
    float Ve(float t); // pour calculer la tension de la source
};

```

# Annexe: EDO.h

```
// classes des EDOs

// la classe mère générale:
class EDO{
public:
    EDO(); // constructeur
    ~EDO(); // destructeur
    // méthodes:
    virtual float euler(float,float) = 0;
    virtual float getU(unsigned short) = 0;
    // méthodes sont purement virtuelles => classe abstraite
    // ces méthodes seront surchargées par les filles
protected: // attributs privés qui peuvent être accédés par les filles
    float t_pre,h; // le temps précédent et le pas
};

class ED01 : public EDO{ // classe fille qui hérite de EDO
public:
    ED01(); // constructeur défaut
    ED01(float y0); // constructeur avec un argument: condition initiale y(0)
    ~ED01(); // destructeur
    float euler(float up, float t); // méthode pour implémenter la méthode d'Euler
        /// arguments: valeur de y'(ti) et ti
    float getU(unsigned short i); // méthode pour obtenir la valeur de U
private: // attribut privé
    float U; // U
};

class ED02 : public EDO{ // classe fille qui hérite de EDO
public:
    ED02(); // constructeur défaut
    ED02(float y0, float yp0); // constructeur avec 2 arguments:
        /// conditions initiales y'(0) et y(0)
    ~ED02(); // destructeur
    float euler(float, float); // méthode pour implémenter la méthode d'Euler
        /// arguments: valeur de y'(ti) et ti
    float getU(unsigned short); // méthode pour obtenir la valeur de U
private:
    float U[2]; // vecteur U = [U, U']
};
```

# Annexe: circuit.h

```
#include <iostream>
#include "source.h"
#include "EDO.h"

// classes des circuits:

// classe mère générale
class circuit{
public: circuit(); // constructeur défaut
      circuit(source *gen); // constructeur
      // argument: source du circuit
      ~circuit(); // destructeur
      // méthodes:
      float getVg(); // pour calculer la tension de la source
      virtual float Vout(float t) = 0; // pour calculer la tension de sortie
      // Vout: purement virtuelle => classe circuit abstraite
      // cette méthode sera surchargée par les filles
protected: // attributs privés qui peuvent être accédés par les filles
      // toutes les filles en auront besoin
      float Vg,Vs,Vsp; // tension de la source, de sortie et dérivée de la sortie
      source *src; // source du circuit
      EDO *eq_diff; // EDO du circuit
};

// classe des circuits d'ordre 1
class circuit_1 : public circuit{ // classe fille qui hérite de circuit
public: circuit_1(); // constructeur défaut
      circuit_1(source *gen); // constructeur
      // argument: source du circuit
      circuit_1(source *gen, float Vout0); // constructeur
      // arguments: source du circuit et Vs(0)
      ~circuit_1(); // destructeur
      // méthode:
      virtual float Vout(float t) = 0; // pour calculer la tension de sortie
      // méthode purement virtuelle => classe circuit_1 abstraite
      // cette méthode sera surchargée par les filles
};

// classe des circuits d'ordre 2
class circuit_2 : public circuit{ // classe fille qui hérite de circuit
public: circuit_2(); // constructeur défaut
      circuit_2(source *gen); // constructeur
      // argument: source du circuit
      circuit_2(source *gen, float Vout0, float Voutp0); // constructeur
      // arguments: source du circuit, Vs(0) et Vs'(0)
      ~circuit_2(); // destructeur
      // méthode:
      virtual float Vout(float t) = 0; // pour calculer la tension de sortie
      // méthode purement virtuelle => classe circuit_2 abstraite
      // cette méthode sera surchargée par les filles
protected: // attribut privé qui peut être accédé par les filles
      // toutes les filles en auront besoin
      float Vs2p; // Vs''(ti) - dérivée seconde du signal de sortie
};

// classe des circuits du type A (1er ordre)
class circuitA : public circuit_1{ // classe fille qui hérite de circuit_1
public: circuitA(); // constructeur défaut
      circuitA(source *gen); // constructeur
      // argument: source du circuit
      circuitA(source *gen, float Vout0, float Res1, float Cap1); // constructeur
      // arguments: source du circuit, Vs(0), R1 et C1
      ~circuitA(); // destructeur
      // méthode surchargée:
```



```

        float Vout(float t); // pour calculer la tension de sortie
private: // attributs privés
    float R1,C1; // R1 et C1
};

// classe des circuits du type B (2e ordre)
class circuitB : public circuit_2{ // classe fille qui hérite de circuit_2
public: circuitB(); // constructeur défaut
    circuitB(source *gen); // constructeur
    // argument: source du circuit
    circuitB(source *gen, float Vout0, float Voutp0, float Res1, float Cap1, float Ind1); //
constructeur
    // arguments: source du circuit, Vs(0), Vs'(0), R1, C1 et L1
    ~circuitB(); // destructeur
    // méthode surchargée:
    float Vout(float t); // pour calculer la tension de sortie
private: // attributs privés
    float L1,R1,C1; // L1, R1 et C1
};

// classe des circuits du type C (1er ordre)
class circuitC : public circuit_1{ // classe fille qui hérite de circuit_1
public: circuitC(); // constructeur défaut
    circuitC(source *gen); // constructeur
    // argument: source du circuit
    circuitC(source *gen, float Vout0, float Res1, float Cap1, float Res2); // constructeur
    // arguments: source du circuit, Vs(0), R1, C1 et R2
    ~circuitC(); // destructeur
    // méthode surchargée:
    float Vout(float t); // pour calculer la tension de sortie
private: // attributs privés
    float R1,C1,R2; // R1, C1 et R2
};

// classe des circuits du type D (2e ordre)
class circuitD : public circuit_2{ // classe fille qui hérite de circuit_2
public: circuitD(); // constructeur défaut
    circuitD(source *gen); // constructeur
    // argument: source du circuit
    circuitD(source *gen, float Vout0, float Voutp0, float Res1, float Cap1, float Ind1); //
constructeur
    // arguments: source du circuit, Vs(0), Vs'(0), R1, C1 et L1
    ~circuitD(); // destructeur
    // méthode surchargée:
    float Vout(float t); // pour calculer la tension de sortie
private: // attributs privés
    float L1,R1,C1; // L1, R1, C1
    float Vgp, Vg_pre, t_pre; // Ve'(t_i) - dérivée du signal d'entrée
    // Vg(t_i-1) - valeur précédente de Vg
    // t_i-1 - temps précédent
};

```

# Annexe: source.cpp

```
#define PI 3.14159265358979323846
#include "source.h"

//// source //////////////////////////////////////

// constructeur défaut
source::source(){
    // on définit les attributs avec des valeurs standard
    Vo = 1;
    phi = 0;}

// constructeur
source::source(float A, float start){
    // on définit les attributs avec les paramètres précisés
    Vo = A;
    phi = start;}

// destructeur
source::~source(){
}

//// periodique //////////////////////////////////////

// constructeur défaut
periodique::periodique(){
    // on définit les attributs avec des valeurs standard
    Vo = 1;
    T = 1;
    phi = 0;}

// constructeur
periodique::periodique(float A, float start, float period){
    // on définit les attributs avec les paramètres précisés
    Vo = A;
    T = period;
    phi = start;}

// destructeur
periodique::~periodique(){
}

//// non_periodique //////////////////////////////////////

// constructeur défaut
non_periodique::non_periodique(){
    // on définit les attributs avec des valeurs standard
    Vo = 1;
    phi = 0;}

// constructeur
non_periodique::non_periodique(float A, float start){
    // on définit les attributs avec les paramètres précisés
    Vo = A;
    phi = start;}

// destructeur
non_periodique::~non_periodique(){
}

//// rect //////////////////////////////////////

// constructeur défaut
rect::rect(){
```

```

    // on définit les attributs avec des valeurs standard
    Vo = 1;
    duree = 100*NANO;
    phi = 50*NANO;}

// constructeur
rect::rect(float A, float start, float Thigh){
    // on définit les attributs avec les paramètres précisés
    Vo = A;
    duree = Thigh;
    phi = start;}

// destructeur
rect::~rect(){
}

// pour calculer la tension de la source
float rect::Ve(float t){
    // Ve = A si t entre phi et durée
    if (t < 0) return 0;
    else if (t >= phi && t < phi+duree) return Vo;
    else return 0;
}

///// creneau //////////////////////////////////////

// constructeur défaut
creneau::creneau(){
    // on définit les attributs avec des valeurs standard
    Vo = 1;
    T = 100*NANO;
    phi = 0;
    duty_cycle = 0.5;}

// constructeur
creneau::creneau(float A, float start, float period, float dc){
    // on définit les attributs avec les paramètres précisés
    Vo = A;
    T = period;
    phi = start;
    duty_cycle = dc;}

// destructeur
creneau::~creneau(){
}

// pour calculer la tension de la source
float creneau::Ve(float t){
    if(t < 0)
        return 0;
    while(phi >= T) phi -= T; // normalisation de phi (phi -> [0,T[)
    t += (T-phi);
    while(t >= T) t -= T; // normalisation de t (t -> [0,T[)
    if(t < duty_cycle*T) return Vo; // Ve = A si t E [0, duty_cycle[)
    else return 0;
}

///// echelon //////////////////////////////////////

// constructeur défaut
echelon::echelon(){
    // on définit les attributs avec des valeurs standard
    Vo = 1;
    phi = 1*NANO;}

// constructeur
echelon::echelon(float A, float start){
    // on définit les attributs avec les paramètres précisés
    Vo = A;

```

```

        phi = start;}}

// destructeur
echelon::~echelon(){
}

// pour calculer la tension de la source
float echelon::Ve(float t){
    if (t >= phi) return Vo; // Ve = toujours A après phi secondes
    else return 0;
}

///// triang //////////////////////////////////////

// constructeur défaut
triang::triang(){
    // on définit les attributs avec des valeurs standard
    Vo = 1;
    T = 100*NANO;
    phi = 0;}

// constructeur
triang::triang(float A, float start, float period){
    // on définit les attributs avec les paramètres précisés
    Vo = A;
    T = period;
    phi = start;}

// destructeur
triang::~triang(){
}

// pour calculer la tension de la source
float triang::Ve(float t){
    if(t < 0) return 0;
    t -= phi; // normalisation de t
    while(t >= T) t -= T; // normalisation de t
    // droites du triangle:
    if (t <= 0) return (-(2*Vo/T)*(t));
    else if (t <= T/2) return ((2*Vo/T)*(t));
    else return (Vo - (2*Vo/T)*(t-T/2));
}

///// sinus //////////////////////////////////////

// constructeur défaut
sinus::sinus(){
    // on définit les attributs avec des valeurs standard
    Vo = 1;
    T = 100*NANO;
    phi = 0;}

// constructeur
sinus::sinus(float A, float start, float period){
    // on définit les attributs avec les paramètres précisés
    Vo = A;
    T = period;
    phi = start;}

// destructeur
sinus::~sinus(){
}

// pour calculer la tension de la source
float sinus::Ve(float t){
    if (t < 0) return 0;
    else return Vo*sin((2*PI/T)*t - phi); // sinus défini en <math.h>
}

```

## Annexe: EDO.cpp

```
#include "EDO.h"

//// EDO //////////////////////////////////////

// constructeur défaut
EDO::EDO(){
    // on définit les attributs
    t_pre = 0; h = 0;
}

// destructeur
EDO::~EDO(){
}

//// ED01 //////////////////////////////////////

// constructeur défaut
EDO1::EDO1(){
    // on définit les attributs avec des valeurs standard
    U = 0; // condition initiale  $y(0) = 0$ 
    t_pre = 0; h = 0;
}

// destructeur
EDO1::~EDO1(){
}

// constructeur
EDO1::EDO1(float y0){
    // on définit l'attribut avec le paramètre
    U = y0; // condition initiale  $y(0)$ 
    // les autres attributs restent avec ces valeurs standard
    t_pre = 0; h = 0;
}

// méthode d'Euler de résolution d'EDOs
float EDO1::euler(float up, float t) {
    h = t-t_pre; // on calcule le pas
    // le pas peut changer d'une itération à l'autre
    U = U + h*up; // on calcule U (Euler)
    t_pre = t; // on sauvegarde le temps pour la prochaine itération
    return U;
}

// méthode pour obtenir la valeur de U
float EDO1::getU(unsigned short i) {
    // le paramètre i n'importe pas puisque U n'est pas un vecteur
    return U;
}

//// ED02 //////////////////////////////////////

// constructeur défaut
EDO2::EDO2(){
    // on définit les attributs avec des valeurs standard
    U[0] = 0; // condition initiale  $y(0) = 0$ 
    U[1] = 0; // condition initiale  $y'(0) = 0$ 
    t_pre = 0; h = 0;
}

// destructeur
EDO2::~EDO2(){
}
```

```

// constructeur
ED02::ED02(float y0, float yp0){
    // on définit les attributs avec les paramètres
    U[0] = y0; // condition initiale y(0)
    U[1] = yp0; // condition initiale y'(0)
    // les autres attributs restent avec ces valeurs standard
    t_pre = 0; h = 0;
}

// méthode d'Euler de résolution d'ED0s
float ED02::euler(float u2p, float t) {
    h = t-t_pre; // on calcule le pas
    // le pas peut changer d'une itération à l'autre
    // on calcule le vecteur U (d'après Euler):
    U[0] = U[0] + h*U[1];
    U[1] = U[1] + h*u2p;
    t_pre = t; // on sauvegarde le temps pour la prochaine itération
    return U[0];
}

// méthode pour obtenir la valeur de U[i]
float ED02::getU(unsigned short i) {
    if (i <= 1) return U[i];
    else return 0;
}

```

# Annexe: circuit.cpp

```
#include "circuit.h"

//// circuit //////////////////////////////////////

// constructeur défaut
circuit::circuit() {
    // on définit les attributs avec des valeurs standard
    src = new rect(1,0,0.5);
    eq_diff = new ED01();
}

// constructeur
circuit::circuit(source *gen) {
    // on définit les attributs avec les paramètres précisés
    src = gen;
    eq_diff = new ED01();
}

// destructeur
circuit::~circuit(){
}

// pour calculer la tension de la source
float circuit::getVg(){
    return Vg;
};

//// circuit_1 //////////////////////////////////////

// constructeur défaut
circuit_1::circuit_1() {
    // on définit les attributs avec des valeurs standard
    src = new echelon(1,0);
    eq_diff = new ED01(0);
}

// constructeur avec source seulement
circuit_1::circuit_1(source *gen) {
    // on définit l'attribut src avec le paramètre
    src = gen;
    // et eq_diff avec un valeur standard
    eq_diff = new ED01(0);
}

// constructeur avec source et condition initiale
circuit_1::circuit_1(source *gen, float Vout0){
    // on définit les attributs avec les paramètres précisés
    src = gen;
    eq_diff = new ED01(Vout0);
}

// destructeur
circuit_1::~circuit_1(){
}

//// circuit_2 //////////////////////////////////////

// constructeur défaut
circuit_2::circuit_2() {
    // on définit les attributs avec des valeurs standard
    src = new echelon(1,0);
    eq_diff = new ED02(0,0);
}
```

```

// constructeur avec source seulement
circuit_2::circuit_2(source *gen) {
    // on définit l'attribut src avec le paramètre
    src = gen;
    // et eq_diff avec des valeurs standard
    eq_diff = new ED02(0,0);
}

// constructeur avec source et conditions initiales
circuit_2::circuit_2(source *gen, float Vout0, float Voutp0){
    // on définit les attributs avec les paramètres précisés
    src = gen;
    eq_diff = new ED02(Vout0,Voutp0);
}

// destructeur
circuit_2::~circuit_2(){
}

//// circuitA //////////////////////////////////////

// constructeur défaut
circuitA::circuitA(){
    // on définit les attributs avec des valeurs standard
    R1 = 50;
    C1 = 1*NANO;
    src = new echelon(1,0);
    eq_diff = new ED01(0);
    Vs = 0;
}

// constructeur avec source seulement
circuitA::circuitA(source *gen){
    // on définit l'attribut src avec le paramètre
    src = gen;
    // et les autres avec des valeurs standard
    R1 = 50;
    C1 = 1*NANO;
    eq_diff = new ED01(0);
    Vs = 0;
}

// constructeur avec tous les attributs
circuitA::circuitA(source *gen, float Vout0, float Res1, float Cap1){
    // on définit les attributs avec les paramètres précisés
    R1 = Res1;
    C1 = Cap1;
    src = gen;
    eq_diff = new ED01(Vout0);
    Vs = Vout0;
}

// destructeur
circuitA::~circuitA(){
}

// pour calculer la tension de sortie
float circuitA::Vout(float t){
    Vg = (*src).Ve(t); // tension de la source
    Vsp = (Vg - Vs)/(R1*C1); // l'equation differentielle
    Vs = (*eq_diff).euler(Vsp,t); // on la résout
    return Vs;
}

//// circuitB //////////////////////////////////////

// constructeur défaut
circuitB::circuitB(){
    // on définit les attributs avec des valeurs standard

```



```

        R1 = 120;
        C1 = 0.1*NANO;
        L1 = 1*MICRO;
        src = new echelon(1,0);
        eq_diff = new EDO2(0,0);
        Vs = 0;
        Vsp = 0;
    }

    // constructeur avec source seulement
    circuitB::circuitB(source *gen){
        // on définit l'attribut src avec le paramètre
        src = gen;
        // et les autres avec des valeurs standard
        R1 = 120;
        C1 = 0.1*NANO;
        L1 = 1*MICRO;
        eq_diff = new EDO2(0,0);
        Vs = 0;
        Vsp = 0;
    }

    // constructeur avec tous les attributs
    circuitB::circuitB(source *gen, float Vout0, float Voutp0, float Res1, float Cap1, float Ind1){
        // on définit les attributs avec les paramètres précisés
        R1 = Res1;
        C1 = Cap1;
        L1 = Ind1;
        src = gen;
        eq_diff = new EDO2(Vout0,Voutp0);
        Vs = Vout0;
        Vsp = Voutp0;
    }

    // destructeur
    circuitB::~circuitB(){
    }

    // pour calculer la tension de sortie
    float circuitB::Vout(float t){
        Vg = (*src).Ve(t); // tension de la source
        Vs2p = -(R1/L1)*Vsp + (Vg - Vs)/(L1*C1); // EDO
        // Vsp = Vs' de l'itération précédente
        // Vs = Vs de l'itération précédente
        Vs = (*eq_diff).euler(Vs2p,t); // on résoud l'EDO
        // et sauvegarde Vs (pour la prochaine itération)
        Vsp = (*eq_diff).getU(1); // on sauvegarde Vs'
        // (pour la prochaine itération)
        return Vs;
    }

    //// circuitC //////////////////////////////////////

    // constructeur défaut
    circuitC::circuitC(){
        // on définit les attributs avec des valeurs standard
        R1 = 36;
        C1 = 1*NANO;
        R2 = 180;
        src = new echelon(1,0);
        eq_diff = new EDO1(0);
        Vs = 0;
    }

    // constructeur avec source seulement
    circuitC::circuitC(source *gen){
        // on définit l'attribut src avec le paramètre
        src = gen;
        // et les autres avec des valeurs standard
        R1 = 36;

```

```

        C1 = 1*NANO;
        R2 = 180;
        eq_diff = new EDO1(0);
        Vs = 0;
    }

    // constructeur avec tous les attributs
    circuitC::circuitC(source *gen, float Vout0, float Res1, float Cap1, float Res2){
        // on définit les attributs avec les paramètres précisés
        R1 = Res1;
        C1 = Cap1;
        R2 = Res2;
        src = gen;
        eq_diff = new EDO1(Vout0);
        Vs = Vout0;
    }

    // destructeur
    circuitC::~circuitC(){
    }

    // pour calculer la tension de sortie
    float circuitC::Vout(float t){
        Vg = (*src).Ve(t); // on prend la tension de la source
        // on calcule Vs' (d'après l'EDO):
        if(Vg > 0.6)
            Vsp = -(1/C1)*(1/R1+1/R2)*Vs+(Vg - 0.6)/(R1*C1);
        else
            Vsp = -Vs/(R2*C1);
        // Vs = Vs de l'itération précédente
        Vs = (*eq_diff).euler(Vsp,t); // on résoud l'EDO
        // et sauvegarde Vs (pour la prochaine itération)
        return Vs;
    }

    //// circuitC //////////////////////////////////////

    // constructeur défaut
    circuitD::circuitD(){
        // on définit les attributs avec des valeurs standard
        R1 = 1.2;
        C1 = 0.1*MICRO;
        L1 = 10*NANO;
        src = new echelon(1,0);
        eq_diff = new EDO2(0,0);
        Vs = 0;
        Vsp = 0;
        Vg_pre = 0;
        t_pre = 0;
        Vgp = 0;
    }

    // constructeur avec source seulement
    circuitD::circuitD(source *gen){
        // on définit l'attribut src avec le paramètre
        src = gen;
        // et les autres avec des valeurs standard
        R1 = 1.2;
        C1 = 0.1*MICRO;
        L1 = 10*NANO;
        eq_diff = new EDO2(0,0);
        Vs = 0;
        Vsp = 0;
        Vg_pre = 0;
        t_pre = 0;
        Vgp = (*src).Ve(0);
    }

    // constructeur avec tous les attributs
    circuitD::circuitD(source *gen, float Vout0, float Voutp0, float Res1, float Cap1, float Ind1){

```

```

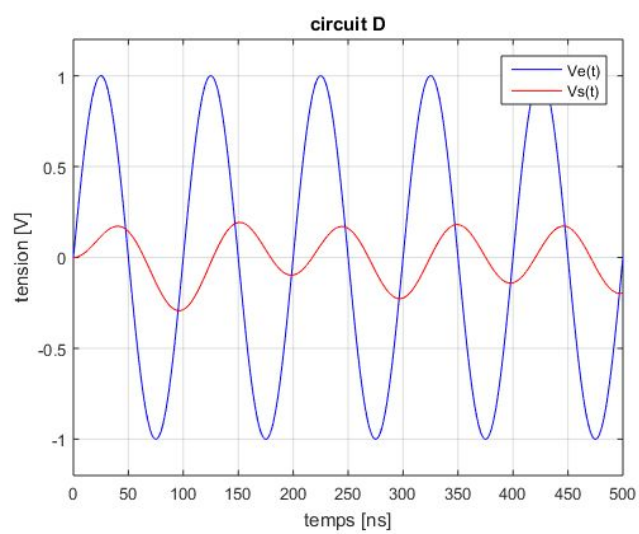
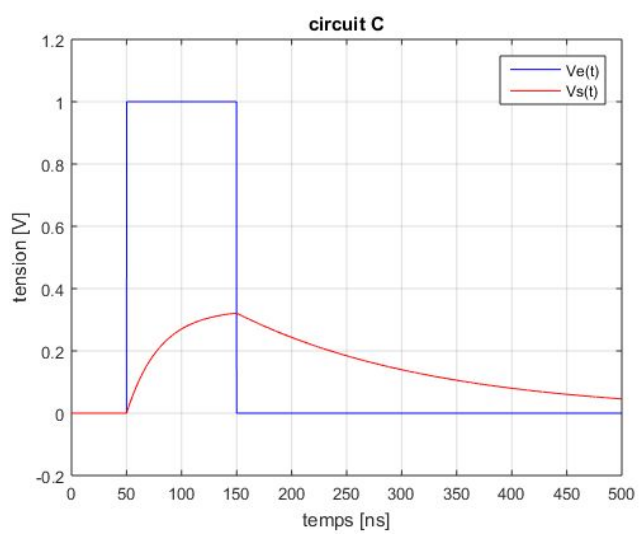
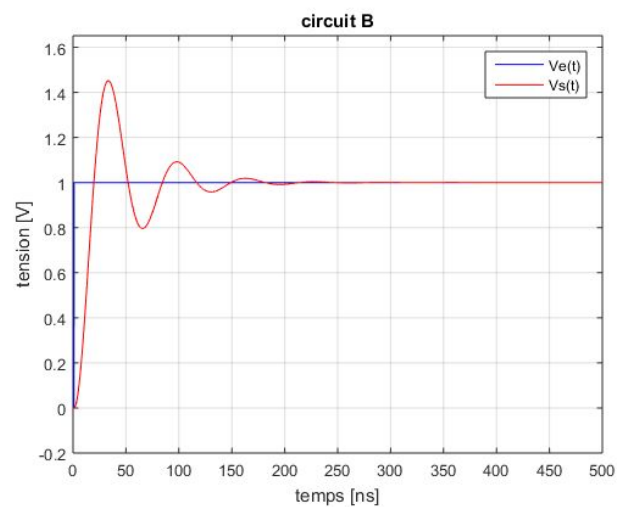
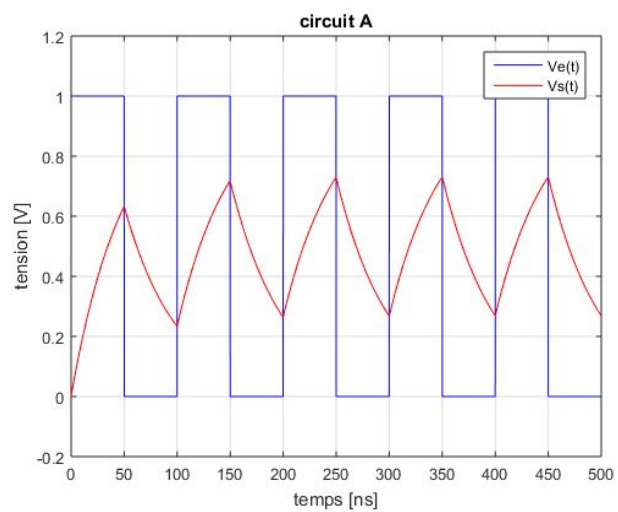
// on définit les attributs avec les paramètres précisés
R1 = Res1;
C1 = Cap1;
L1 = Ind1;
src = gen;
eq_diff = new EDO2(Vout0,Voutp0);
Vs = Vout0;
Vsp = Voutp0;
Vg_pre = 0;
t_pre = 0;
Vgp = (*src).Ve(0);
}

// destructeur
circuitD::~circuitD(){
}

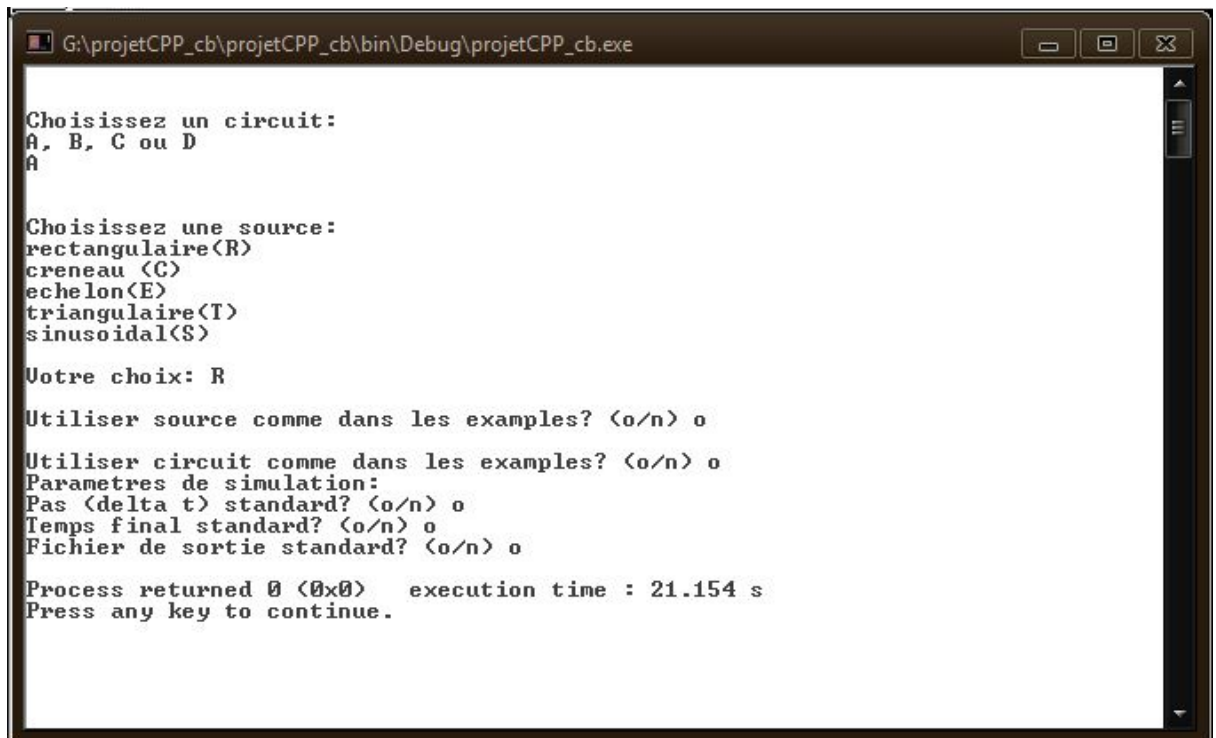
// pour calculer la tension de sortie
float circuitD::Vout(float t){
    if(t){ // si t = 0, les valeurs définies dans le constructeur restent les mêmes
        Vg = (*src).Ve(t); // on prend la tension de la source
        if (t-t_pre) Vgp = (Vg-Vg_pre)/(t-t_pre); // on calcule sa dérivée (Vg')
        // si temps = temps précédent, Vg' est le même
        Vs2p = (Vgp-Vsp)/(R1*C1) - Vs/(L1*C1); // Vs'' (d'après l'EDO)
        // Vsp = Vs' de l'itération précédente
        // Vs = Vs de l'itération précédente
        Vs = (*eq_diff).euler(Vs2p,t); // on résout l'EDO et sauvegarde Vs
        Vsp = (*eq_diff).getU(1); // on sauvegarde la valeur de Vs' pour la prochaine itération
        Vg_pre = Vg; // on sauvegarde Ve pour la prochaine itération (calcul de Vg')
        t_pre = t; // on sauvegarde le temps pour la prochaine itération (calcul de Vg')
    }
    return Vs;
}

```

## Annexe: exemples de simulation de circuits



## Annexe: captures d'écran du fonctionnement du programme



```
G:\projetCPP_cb\projetCPP_cb\bin\Debug\projetCPP_cb.exe

Choisissez un circuit:
A, B, C ou D
A

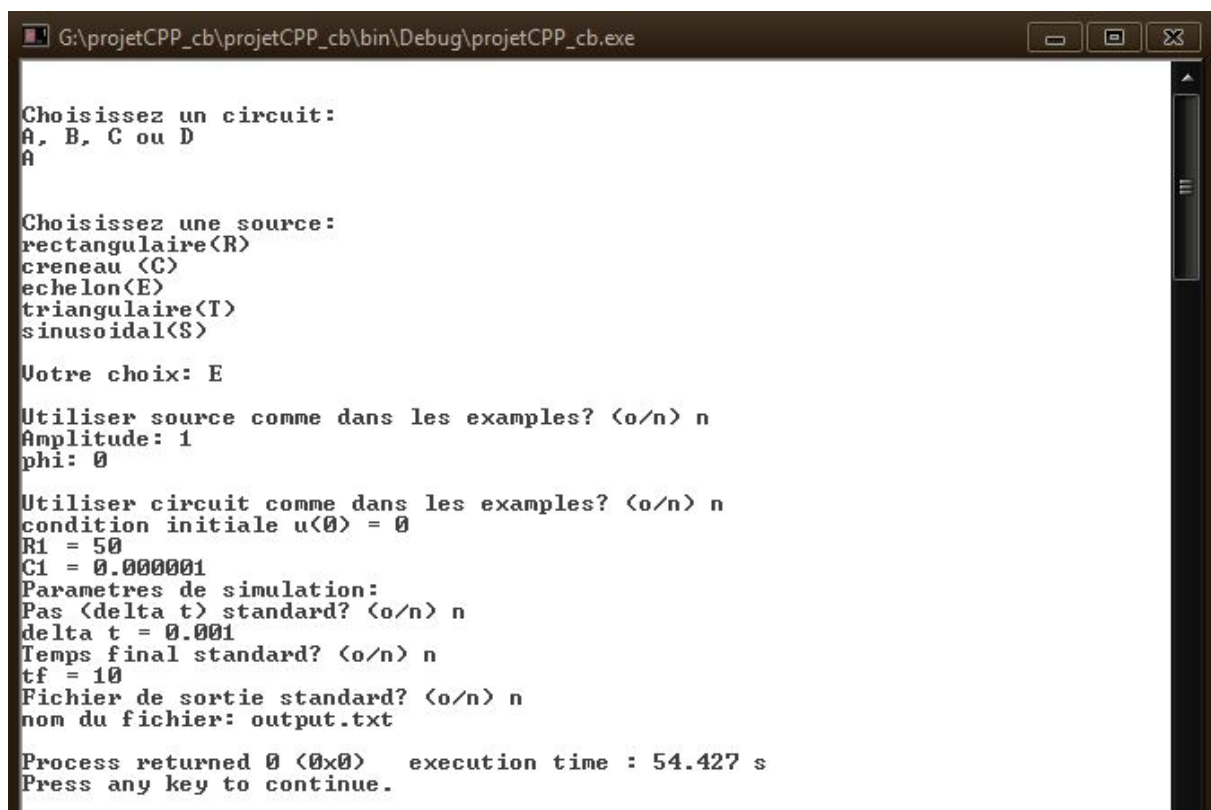
Choisissez une source:
rectangulaire(R)
creneau (C)
echelon(E)
triangulaire(T)
sinusoidal(S)

Votre choix: R

Utiliser source comme dans les exemples? <o/n> o

Utiliser circuit comme dans les exemples? <o/n> o
Parametres de simulation:
Pas (delta t) standard? <o/n> o
Temps final standard? <o/n> o
Fichier de sortie standard? <o/n> o

Process returned 0 (0x0)   execution time : 21.154 s
Press any key to continue.
```



```
G:\projetCPP_cb\projetCPP_cb\bin\Debug\projetCPP_cb.exe

Choisissez un circuit:
A, B, C ou D
A

Choisissez une source:
rectangulaire(R)
creneau (C)
echelon(E)
triangulaire(T)
sinusoidal(S)

Votre choix: E

Utiliser source comme dans les exemples? <o/n> n
Amplitude: 1
phi: 0

Utiliser circuit comme dans les exemples? <o/n> n
condition initiale u(0) = 0
R1 = 50
C1 = 0.000001
Parametres de simulation:
Pas (delta t) standard? <o/n> n
delta t = 0.001
Temps final standard? <o/n> n
tf = 10
Fichier de sortie standard? <o/n> n
nom du fichier: output.txt

Process returned 0 (0x0)   execution time : 54.427 s
Press any key to continue.
```