If you think it's simple, then
you have misunderstood the problem.
Bjarne Stroustrup

CHAPITRE XVI

Calcul arrondi fiable et arithmétique d'intervalles

Objectif. Dans ce chapitre notre but est de comprendre les éléments du calcul arrondi fiable.
► L'arrondi permet de rendre certains calculs faisables (ou bien plus efficaces) sur ordinateur.
► En général le résultat ainsi calculé sera erroné ; il est donc nécessaire de majorer l'erreur commise.
Cette erreur provient de la méthode (approximation) et de l'arithmétique utilisée (erreurs d'arrondis). Dans cette problématique les arrondis dirigés peuvent donner des informations précieuses :
► L'arrondi vers $-\infty$ donne un résultat approché a qui fournit une minoration fiable.
► L'arrondi vers $+\infty$ donne un résultat approché b qui fournit une majoration fiable.
Les deux bornes ensemble fournissent un encadrement $[a,b]$ du résultat exact $r \in \mathbb{R}$. L'objectif d'une bonne implémentation est d'abord de garantir l'encadrement $a \leq r \leq b$, puis de minimiser l'écart $|b-a|$, et finalement d'économiser les ressources (temps et mémoire).

Le problème fondamental du calcul arrondi. Lors d'un calcul arrondi les valeurs approchées calculées peuvent s'éloigner de la valeur exacte cherchée. Comme à la fin nous ne disposons que de la valeur calculée (erronée) et non de la valeur cherchée (exacte), il nous faut un moyen de contrôler la marge d'erreur. Dans le pire des cas une « explosion d'erreur » peut rendre le calcul inutilisable, car la valeur calculée n'a plus aucun rapport avec la valeur cherchée. Comment savoir si un résultat est acceptable ou non ? Plusieurs stratégies sont imaginables pour assurer (ou au moins tester) la fiabilité des résultats calculés :

(1) Dans certains cas on peut éviter les arrondis en faisant un calcul *exact* dans un anneau effectif comme $\mathbb{Z}$ ou $\mathbb{Q}$ ou $\mathbb{Q}[\sqrt{2}]$ etc. On se ramène ainsi à une situation entièrement algébrique, où tout élément peut être représenté de manière exacte et toute opération peut être exécutée sans arrondi. Si le problème en question s'y prête et que le calcul n'est pas trop coûteux, c'est la solution la plus élégante et la plus sûre.

(2) En implémentant un calcul numérique, on est tenté de calculer avec le type `double`, disons, comme si c'était un calcul exact. Cette stratégie naïve est la plus répandue, mais aussi la plus périlleuse. Un calcul maladroit peut provoquer la perte totale de chiffres significatifs. Même un calcul habile ne fournit aucune indication quant à la marge d'erreur.

(3) Par mesure de sécurité on peut calculer avec beaucoup plus de chiffres que nécessaire, disons 40 décimales pour ne retenir que 20 chiffres significatifs à la fin du calcul. C'est un peu moins naïf, mais ne protège pas contre les catastrophes. Bien que les chances soient meilleures, on n'a toujours aucune garantie de correction

(4) On peut calculer avec ℓ chiffres, puis refaire le calcul avec 2ℓ chiffres de précision. On n'acceptera que les chiffres qui coïncident, en espérant que ce sont en quelque sorte des chiffres « stables », donc « corrects ». `(` Un programmeur craintif recalculera avec 3ℓ chiffres. `;-)` Cette recette marche souvent, mais on peut bien sûr tomber sur de méchants contre-exemples.

(5) Pour avoir des encadrements prouvables, il ne reste que le contrôle minutieux des arrondis. Typiquement on fait deux calculs séparés afin d'établir une minoration puis une majoration. Ceci demande une implémentation plus soigneuse, mais en contrepartie le résultat calculé est toujours honnête. Quand le calcul tourne mal, au moins on le verra ouvertement.

Suivant la structure du problème et l'importance du résultat on choisira une de ces stratégies, ou une des nombreuses variantes. Il est clair que l'on programmerait différemment les exercices de ce chapitre s'ils servaient à piloter une fusée. (Espérons, au moins.) Ce sont la méthode choisie et le soin apporté à l'implémentation qui détermineront la fiabilité du résultat.

Quels son les avantages d'un calcul fiable ? Soulignons que même dans un calcul arrondi fiable il y aura toujours des erreurs d'arrondi. C'est un phénomène caractéristique et inévitable : ce calcul fut inventé pour rendre les calculs plus efficaces, au prix des arrondis. Le but n'est donc pas de supprimer les arrondis, mais de contrôler la marge d'erreur dans le résultat final.

Un calcul fiable fournit un encadrement $[a,b]$ du résultat exact : cet encadrement est prouvable et met en évidence la marge d'erreur. Quoi qu'il arrive, nous obtiendrons toujours un résultat avec garantie de correction ! Dans le pire des cas l'intervalle de l'encadrement peut être trop grossier, mais même ce résultat décevant contient une information importante : il signale que le calcul a mal tourné, et qu'il faut traiter le résultat avec prudence. Si la précision atteinte est jugée insuffisante, il faudra refaire un calcul plus raffiné. En tout cas il sera malhonnête de prétendre une précision supérieure à l'intervalle établi par le calcul.

Si la précision est suffisante, on extrait une valeur approchée avec une précision garantie. Pour cette raison le calcul fiable est aussi appelé *calcul auto-certifiant* (terme publicitaire).

Comment s'y prendre ? On peut bien sûr faire appel à une bibliothèque toute faite, telle que la GMP (*GNU multiple precision library*) et ses extensions MPFR (*multiple-precision floating-point computations with correct rounding*) et MPFI (multiple-precision floating-point interval arithmetic). Vous êtes vivement invités à vous renseigner sur les sites respectifs `www.swox.com/gmp` et `www.mpfr.org` pour avoir une idée de ce qu'elles offrent. Ces informations sont également disponible en local, en tapant `info mpfr` et `info gmp C++` dans une ligne de commande.

De manière plus pédestre, nous nous proposons ici d'illustrer le principe par une implémentation « faite maison ». Comme bénéfice collatéral ceci nous donne l'occasion d'expliquer les nombres flottants et l'arithmétique arrondie en tout détail. On complète ainsi notre introduction aux nombres flottants esquissée au chapitre précédent.

Ayant implémentée *l'arithmétique arrondie fiable*, on peut pousser ce concept un peu plus loin. Afin de rendre les objets plus intuitifs et plus agréables à manipuler, nous implémentons *l'arithmétique d'intervalles* qui calcule majoration et minoration parallèlement. Nous arrivons ainsi à un calcul numérique, sous une forme naturelle et une écriture commode, qui fournit non seulement une valeur approchée mais en même temps la marge d'erreur sous forme d'encadrement.

Pour en savoir plus. Le calcul arrondi fiable et l'arithmétique d'intervalles ne datent pas d'hier, mais ils attirent de plus en plus d'attention ces dernières années, surtout dans les domaines sensibles qui exigent un très haut niveau de sécurité et une garantie de précision. L'arithmétique d'intervalles n'est pas le remède à tous les maux numériques, mais elle offre souvent une démarche sûre et simple.

Pour en savoir d'avantage, consultez le site `www.cs.utep.edu/interval-comp`, qui rassemble de nombreux liens utiles concernant l'arithmétique d'intervalles :

(1) Pour le coté vulgarisation scientifique, lire l'excellent survol de B. Hayes, *A lucid interval*, `www.cs.utep.edu/interval-comp/hayes.pdf`.

(2) Si cela vous chante, vous pouvez aussi regarder — puis analyser — un film destiné aux étudiants, `www-sop.inria.fr/coprin/logiciels/ALIAS/Movie/movie_undergraduate.mpg`.

(3) Pour une discussion provocatrice lire W. Kahan, *How futile are mindless assessments of roundoff in floating-point computation ?* `www.cs.berkeley.edu/~wkahan/Mindless.pdf`.

Sommaire

1. Deux types d'erreurs inévitables. 1.1. Erreurs de discrétisation. 1.2. Erreurs de calcul.

2. Calcul arrondi fiable. 2.1. Arrondis dirigés. 2.2. Arithmétique arrondie. 2.3. Une implémentation « faite maison ». 2.4. Comment arrondir ? 2.5. Comment multiplier ? 2.6. Comment diviser ? 2.7. Comment additionner ? 2.8. Newton-Héron revisité. 2.9. Instabilité numérique revisitée.

3. Arithmétique d'intervalles. 3.1. Arithmétique d'intervalles idéalisée. 3.2. Arithmétique d'intervalles arrondie. 3.3. Une implémentation « faite maison ». 3.4. Exemples d'utilisation.

4. Applications. 4.1. Retour sur les problèmes du chapitre XV. 4.2. La fonction zéta de Riemann. 4.3. Séries alternées : sin, cos, arctan, etc. 4.4. Calcul de π.

1. Deux types d'erreurs inévitables

1.1. Erreurs de discrétisation. Afin de modéliser le calcul numérique dans $\mathbb{R}$ nous sommes obligés de nous restreindre à un sous-ensemble $R \subset \mathbb{R}$ de nombres exactement représentables sur machine. À titre d'exemple, R peut être constitué des nombres flottants de longueur ℓ. Pour simplifier nous autorisons des exposants $e \in \mathbb{Z}$. (Nous sous-entendons que l'exposant e ne sera pas trop grand, mais nous n'imposons pas de limites a priori.) Ainsi l'ensemble des nombres machine est donné par

$$R = \{0\} \cup \left\{ m \cdot 2^e \;\middle|\; m, e \in \mathbb{Z} \text{ avec } -2^\ell < m < 2^\ell \right\}. \tag{1}$$

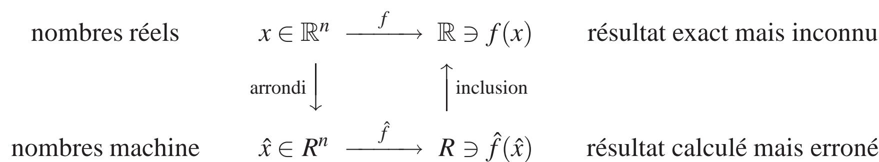
Rappelons que dans l'écriture $m \cdot 2^e$ on appelle m la *mantisse* et *e l'exposant*. En fixant la longueur de la mantisse à ℓ bits on restreint la précision des valeurs ainsi représentables, mais en contrepartie on diminue aussi le coût des calculs (en temps et en mémoire).

Contrairement aux nombres réels $\mathbb{R}$ qui modélisent des phénomènes continus, le modèle informatique des nombres flottants R est forcément discrétisé. Néanmoins on est obligé d'approcher tout nombre réel $x \in \mathbb{R}$ par un nombre flottant $\hat{x} \in R$. Pour presque tout nombre réel x nous avons $x \neq \hat{x}$, et l'application $x \mapsto \hat{x}$ introduit donc une *erreur d'arrondi* $|\hat{x} - x|$, aussi appelé *erreur de discrétisation*.

La différence $|\hat{x} - x|$ est *l'erreur absolue*. Il est souvent plus informatif de considérer *l'erreur relative* $\varepsilon = |\frac{\hat{x}-x}{x}|$. En arrondissant au plus proche on obtient $\varepsilon \leq 2^{-\ell}$, autrement dit la différence entre la valeur réelle $x \in \mathbb{R}$ et son approximation discrète $\hat{x} \in R$ s'exprime comme $\hat{x} = x(1+\delta)$ avec un facteur de perturbation δ vérifiant $|\delta| \leq 2^{-\ell}$. (Voir le chap. XV, §4.)

☞ *La longueur ℓ de la mantisse détermine la granularité de la discrétisation.* ✍

1.2. Erreurs de calcul. Nous résumons la différence entre un calcul exact dans $\mathbb{R}$ et un calcul approché dans $R \subset \mathbb{R}$ par le diagramme suivant. L'approximation $\mathbb{R}^n \to R^n$ envoie $(x_1, \ldots, x_n)$ sur $(\hat{x}_1, \ldots, \hat{x}_n)$ où chaque coefficient $x_k \in \mathbb{R}$ est représenté par une valeur approchée $\hat{x}_k \in R$. De même, $\hat{f} \colon R^n \to R$ est une fonction calculable sur machine qui ne fournit qu'une approximation de la fonction réelle $f \colon \mathbb{R}^n \to \mathbb{R}$.



☞ *En général ce diagramme ne commute pas, c'est-à-dire $f(x) \neq \hat{f}(\hat{x})$.*
Seule la connaissance de la marge d'erreur donne autorité au résultat calculé $\hat{f}(\hat{x})$. ✍

Plus explicitement, l'écart entre le résultat exact et la valeur approchée calculée vaut

$$f(x) - \hat{f}(\hat{x}) = \big[f(x) - f(\hat{x})\big] + \big[f(\hat{x}) - \hat{f}(\hat{x})\big].$$

On voit apparaître la somme de deux erreurs :

(1) La première erreur est due à la discrétisation de x et dépend des nombres machine utilisés : un nombre réel n'est stocké qu'avec un nombre limité de chiffres significatifs (disons ℓ bits).

(2) La seconde erreur est due à la méthode utilisée pour approcher f. On a tout intérêt à utiliser une approximation $\hat{f}$ qui soit aussi proche de f que possible, mais il restera souvent un écart.

Soulignons à nouveau qu'il est indispensable de majorer l'erreur totale. Sinon il est inutile, voire nuisible, de se lancer dans le calcul. Pour certains calculs très simples il est relativement facile de majorer l'erreur. Pour des calculs réalistes ceci devient de plus en plus dur : une majoration fine est souvent inextricable, et des majorations grossières ne donnent souvent pas de résultat satisfaisant. Pour cette raison nous développons dans la suite les éléments d'un calcul arrondi fiable, aussi appelé *calcul auto-certifiant* :

☞ *On ne peut espérer $\hat{f}(\hat{x}) = f(x)$, mais on peut garantir $\hat{f}(\hat{x}) \leq f(x)$ ou $f(x) \leq \hat{f}(\hat{x})$, respectivement, et ainsi encadrer la valeur exacte par deux valeurs approchées.* ✍

2. Calcul arrondi fiable

Ce qui gâche les calculs avec le type `double` n'est souvent pas la précision insuffisante : la longueur de la mantisse est de 53 bits, soit 16 décimales environ, ce qui est suffisant pour beaucoup d'applications. Le problème est surtout la difficulté de contrôler les erreurs d'arrondi. Ce paragraphe essaie d'apporter quelques réponses à ce problème et de développer les éléments d'un calcul arrondi fiable.

2.1. Arrondis dirigés. Supposons que durant un long calcul numérique chaque résultat intermédiaire est arrondi vers le nombre machine le plus proche. A priori c'est une bonne idée car on minimise localement l'erreur de chaque opération élémentaire. Mais ce n'est vrai que *localement*. L'écart accumulé peut être assez grand, sans que l'utilisateur soit averti. (Revoir les pièges discutés au chapitre XV.) Ce qu'il faut assurer est une relation fiable entre le résultat calculé (mais erroné) et le résultat exact (mais inconnu).

☞ *À première vue l'arrondi vers le plus proche semble toujours la meilleure option. C'est le mode le plus courant, mais il offre le moins de contrôle sur le résultat.* ✍

Pour les nombres réels $\mathbb{R}$, la relation fiable à garantir sera l'ordre : tout nombre réel x peut être encadré par deux nombres machine $\underline{x}, \overline{x} \in R$ de sorte que $\underline{x} \le x \le \overline{x}$. Si jamais $x \in R$, on pourra tomber sur $\underline{x} = x = \overline{x}$, mais cela restera une exception rare. En général on a une inégalité stricte $\underline{x} < x < \overline{x}$ et il faut décider avec laquelle des deux approximations on continuera le calcul.

☞ *En choisissant judicieusement les modes d'arrondi on peut garantir un encadrement fiable du résultat final.* ✍

Définition 2.1 (modes d'arrondi)**.** Nous supposons que les nombres machine $R \subset \mathbb{R}$ forment un sous-ensemble fermé dans la topologie de $\mathbb{R}$. Nous exigeons aussi que R contienne au moins $0, \pm 1, \pm 2$, qu'il soit symétrique ($R = -R$) et qu'il « recouvre » tout $\mathbb{R}$ dans le sens que $\sup R = +\infty$ et $\inf R = -\infty$. Toutes ces exigences sont satisfaites pour notre ensemble (1).

Étant donné un nombre réel $x \in \mathbb{R}$, on ne peut en général pas le représenter de manière exacte par un nombre machine. On prendra donc une valeur approchée $\hat{x} \in R$, le meilleur choix étant un de ses deux « voisins » dans R. Au moins cinq modes d'arrondi $\mathbb{R} \to R$ s'offrent à nous :

$\lfloor x \rfloor_R := \sup\{\underline{x} \in R \mid \underline{x} \le x\}$	arrondi vers $-\infty$, vers le bas	(RNDD = *round down*)
$\lceil x \rceil_R := \inf\{\overline{x} \in R \mid x \le \overline{x}\}$	arrondi vers $+\infty$, vers le haut	(RNDU = *round up*)
$[x]_R := \begin{cases} \lfloor x \rfloor_R & \text{si } x \ge 0 \\ \lceil x \rceil_R & \text{si } x \le 0 \end{cases}$	arrondi vers zéro	(RNDZ = *round to zero*)
$]x[_R := \begin{cases} \lceil x \rceil_R & \text{si } x \ge 0 \\ \lfloor x \rfloor_R & \text{si } x \le 0 \end{cases}$	arrondi vers $\pm\infty$	(RNDI = *round to infinity*)
$\lfloor x \rceil_R := \begin{cases} \lfloor x \rfloor_R & \text{si } x < m \\ \lceil x \rceil_R & \text{si } x > m \end{cases}$	arrondi vers le plus proche	(RNDN = *round to nearest*)

Pour l'arrondi vers le plus proche on pose $m = \frac{1}{2}(\lfloor x \rfloor_R + \lceil x \rceil_R)$. En cas d'égalité $x = m$ il faut fixer une règle d'arbitrage. Pour les nombres flottants (1) on convient de choisir celui dont la mantisse est paire, i.e. finit par un zéro dans son développement binaire (« arrondi pair »).

Exercice/M 2.2. Vérifier que pour $R = \mathbb{Z}$ on obtient les définitions usuelles de $\lfloor x \rfloor$, $\lceil x \rceil$, $[x]$, $\lfloor x \rceil$.

(1) Plus généralement, si $R \subset \mathbb{R}$ est un sous-ensemble *fermé* dans la topologie de $\mathbb{R}$, montrer que $\lfloor x \rfloor_R$ et $\lceil x \rceil_R$ sont toujours des éléments de R, quelque soit $x \in \mathbb{R}$. Les valeurs arrondies sont donc exactement représentables, ce qui est la propriété essentielle.

(2) Discuter l'ensemble $R = \mathbb{Q}$ qui n'est pas fermé dans $\mathbb{R}$. Que valent $\lfloor x \rfloor_{\mathbb{Q}}$ et $\lceil x \rceil_{\mathbb{Q}}$? Les valeurs ainsi « arrondies », sont-elles représentables sur machine ? Expliquer pourquoi cette approche, tellement utile en analyse et algèbre, ne convient pas pour le calcul numérique.

2.2. Arithmétique arrondie. Pour les nombres réels nous disposons des opérations élémentaires $+,-,*,/$ dont nous pouvons restreindre le domaine d'application à l'ensemble R :

$$+\colon R\times R\to\mathbb{R} \qquad *\colon R\times R\to\mathbb{R}$$
$$-\colon R\times R\to\mathbb{R} \qquad /\colon R\times R^*\to\mathbb{R}$$

Remarque. — Nous ne pouvons pas espérer que les images soient incluses dans R. Si jamais on avait un modèle de nombres machine R tel que $+,-,*,/$ prenaient toutes leurs valeurs dans R, alors R contiendrait le corps $\mathbb{Q}$ des nombres rationnels. Ce serait donc un sous-ensemble dense de $\mathbb{R}$, aussi inconvénient que $\mathbb{Q}$ pour le calcul arrondi. (Voir l'exercice précédent.) Ceci montre que le problème des opérations $+,-,*,/$ est inhérent au calcul arrondi, et non seulement un artefact d'une implémentation maladroite.

On fait donc appel aux arrondis introduits dans la définition précédente. On implémente l'addition, par exemple, comme l'addition exacte $+\colon R\times R\to\mathbb{R}$ suivie de l'arrondi spécifié $\mathbb{R}\to R$. Ainsi l'addition $+\colon\mathbb{R}\times\mathbb{R}\to\mathbb{R}$ est modélisée non par *une* mais par *cinq* opérations pour le calcul approché :

$$\lfloor+\rfloor,\ \lceil+\rceil,\ [+],\ ]+[,\ \lfloor+\rceil\colon R\times R\to R$$

Regardons plus généralement le calcul d'une fonction $f\colon\mathbb{R}\to\mathbb{R}$, comme $\exp\colon\mathbb{R}\to\mathbb{R}$. À nouveau on ne peut pas espérer que $f(R)\subset R$, il faut donc arrondir. La façon la plus honnête sera d'implémenter deux approximations $\lfloor f\rfloor,\lceil f\rceil\colon R\to R$ qui garantissent pour tout $x\in R$ un encadrement

$$\lfloor f\rfloor(x) \quad\le\quad f(x) \quad\le\quad \lceil f\rceil(x).$$

Étant donnée f, il nous reste évidemment le problème d'implémenter correctement $\lfloor f\rfloor$ et $\lceil f\rceil$ de manière efficace et avec un écart $\lceil f\rceil-\lfloor f\rfloor$ aussi petit que possible. C'est souvent faisable et permettra de *garantir* que l'encadrement final sera correct et satisfaisant.

2.3. Une implémentation « faite maison ». Après la discussion des modes d'arrondi, passons à une implémentation concrète ! Le programme XVI.1 donne une esquisse de la classe `RReal` modélisant des *nombres flottants fiables*, ou *reliably rounded real number* en anglais. Vous trouvez le code source complet dans le fichier `rreal.cc`, ainsi qu'un exemple d'utilisation dans `rreal-test.cc`.

L'idée est d'implémenter des nombres flottants en précision ℓ arbitraire. La présentation (1) a l'avantage de se baser uniquement sur les nombres entiers : les flottants sont stockés sous la forme $m\cdot 2^e$ avec deux entiers m (la mantisse) et e (l'exposant). Afin de pouvoir choisir la longueur de la mantisse (éventuellement très grande) il nous faut une implémentation de grands entiers. Nous nous servirons ici de la classe `mpz_class` de la bibliothèque GMP, qui fournit toutes les opérations nécessaires.

2.4. Comment arrondir ? On aura besoin d'une division arrondie : pour deux entiers a et $b\neq 0$ la fonction `eudiv(a,b,mode)` effectue une division euclidienne $a=qb+r$ avec $|r|<|b|$. On a donc $q=\lfloor\frac{a}{b}\rfloor_{\mathbb{Z}}$ ou $q=\lceil\frac{a}{b}\rceil_{\mathbb{Z}}$, et le quotient renvoyé est choisi suivant le mode d'arrondi spécifié.

Comme les entiers sont stockés par leur développement binaire, la division euclidienne par 2e est particulièrement efficace. (Rappeler pourquoi.) La fonction `eudiv2(a,e,mode)` effectue cette division de a par 2e telle que le quotient soit arrondi comme spécifié. Ceci permet de réduire une mantisse `m` à une longueur ℓ, simplement en posant `m= eudiv2(m,exces,mode)`. C'est justement l'objectif de la fonction `arrondir(mode,len)`.

Le constructeur `RReal(man,exp)` et la fonction `set(man,exp)` permettent de définir aisément un nombre de la forme $m\cdot 2^e$. Si en plus les paramètres `mode` et `len` sont spécifiés, on réduit la longueur de la mantisse à `len` bits en appliquant le mode d'arrondi spécifié. Si ces paramètres ne sont pas spécifiés, on prend les valeurs par défaut `mode= stdMode` et `len= stdLen`.

Finalement, la fonction `rallonger(len)` rajoute des bits zéro à la mantisse, alors que la fonction `remplir(len)` assure que la mantisse soit de longueur $\ge$ `len`.

2.5. Comment multiplier ? L'arithmétique arrondie implémente un principe net et simple : on effectue toute opération élémentaire d'abord de manière exacte, puis on arrondit la mantisse à la longueur prescrite. La multiplication de deux nombres à virgule flottante est particulièrement simple, voir le programme XVI.1. C'est presque trivial à un détail important près : la fonction `set` employée ici effectue automatiquement l'arrondi nécessaire. (Pourquoi est-ce important ?)

Programme XVI.1 La classe `RReal` — nombres flottants avec arrondi dirigé

```
// Quelques types et fonctions auxiliaires
enum Mode { RNDD, RNDU, RNDZ, RNDI, RNDN }; // modes d'arrondi
typedef mpz_class Man;      // type pour la mantisse (grand entier)
typedef mpz_class Exp;      // type pour l'exposant  (grand entier)
typedef int Len;            // type pour la longueur (petit entier)
Len length( const Man& a ); // longueur d'un entier = nombre de bits

// Division euclidienne de a par b, puis de a par 2^e, suivant le mode d'arrondi
Man eudiv( const Man& a, const Man& b, Mode mode= stdMode );
Man eudiv2( const Man& a, Exp exp, Mode mode= stdMode );

// La classe RReal implémente des nombres flottants fiables
class RReal
{
public:
  Man mantisse; // la mantisse
  Exp exposant; // l'exposant

  // Précision par défaut (variable statique, càd globale pour RReal)
  static Len stdLen;
  static Len len() { return stdLen; };
  static Len len( Len l ) { Len old= stdLen; stdLen= max(l,2l); return old; }

  // Mode d'arrondi par défaut (variable statique, càd globale pour RReal)
  static Mode stdMode;
  static Mode mode() { return stdMode; }
  static Mode mode( Mode m ) { Mode old= stdMode; stdMode= m; return old; }

  // La fonction suivante effectue l'arrondi comme spécifié par mode et len
  void arrondir( Mode mode= stdMode, Len len= stdLen )
    {
      if ( mantisse == 0 ) { exposant= 0; return; }; // cas particulier
      Len exces= length() - max( len, 2l );          // bits de trop
      if ( exces <= 0 ) return;                      // rien à raccourcir
      mantisse= eudiv2( mantisse, exces, mode );     // supprimer l'excès
      exposant+= exces;                              // compenser l'exposant
    }

  // Constructeur à partir d'une pair (mantisse, exposant) suivi d'arrondi
  RReal( Man man=0, Exp exp=0, Mode mode= stdMode, Len len= stdLen )
    : mantisse(man), exposant(exp) { arrondir( mode, len ); }

  // Affectation d'une pair (mantisse, exposant) suivi d'arrondi
  RReal& set( Man man=0, Exp exp=0, Mode mode= stdMode, Len len= stdLen )
    { mantisse= man; exposant= exp; arrondir( mode, len ); return *this; }

  // La fonction  rallonger  rajoute des bits zéro à la mantisse
  void rallonger( Len len )
    { if ( len > 0 ) { mantisse <<= len; exposant -= len; }; }

  // La fonction  remplir  assure que la mantisse soit de longueur >= len.
  void remplir( Len len= stdLen )
    { rallonger( len - length() ); }

  // Multiplication : multiplier les mantisses et additionner les exposants
  RReal& mul( const RReal& a, const RReal& b, Mode mode= stdMode, Len len= stdLen )
    { return set( a.mantisse * b.mantisse, a.exposant + b.exposant, mode, len ); }
};

// Multiplication sous forme d'opérateur, créant un objet temporaire
RReal operator* ( const RReal& a, const RReal& b )
{ RReal c; c.mul(a,b); return c; }
```

2.6. Comment diviser ? *Attention. —* La tentative suivante est vouée à l'échec :

```
RReal operator/ ( const RReal& a, const RReal& b )
{ return RReal( a.mantisse / b.mantisse, a.exposant - b.exposant ); }
```

Il faut d'abord rallonger la mantisse de `a` pour obtenir un quotient de précision suffisante ; ensuite `eudiv(a,b,mode)` calcule le quotient entier suivant le mode d'arrondi spécifié :

```
RReal& RReal::div( RReal a, const RReal& b, Mode mode= stdMode, Len len= stdLen )
{ a.remplir( len + b.length() );
  return set( eudiv(a.mantisse,b.mantisse,mode), a.exposant-b.exposant, mode, len ); }

RReal operator/ ( const RReal& a, const RReal& b )
{ RReal c; c.div(a,b); return c; }
```

2.7. Comment additionner ? L'addition est un peu moins évidente : elle nécessite d'abord d'égaliser les exposants. Plus explicitement, pour additionner $a = m_1 \cdot 2^{e_1}$ et $b = m_2 \cdot 2^{e_2}$ on détermine $e = \min\{e_1, e_2\}$ pour écrire $a = m_1' \cdot 2^e$ et $b = m_2' \cdot 2^e$ avec deux entiers $m_1' = m_1 \cdot 2^{e_1 - e}$ et $m_2' = m_2 \cdot 2^{e_2 - e}$. Sous cette forme on calcule ensuite $a + b = (m_1' + m_2') \cdot 2^e$, puis on arrondit le résultat le cas échéant.

```
RReal& RReal::add( RReal a, RReal b, Mode mode= stdMode, Len len= stdLen )
{ Exp minexp= min( a.exposant, b.exposant );
  a.rallonger( a.exposant - minexp );
  b.rallonger( b.exposant - minexp );
  return set( a.mantisse + b.mantisse, minexp, mode, len ); }

RReal operator+ ( const RReal& a, const RReal& b )
{ RReal c; c.add(a,b); return c; }
```

Exercice/P 2.3. Bien évidemment nos algorithmes pour le calcul arrondi sont susceptibles d'optimisation. Surtout l'égalisation des exposants est inutilement coûteuse si les deux exposants diffèrent de plus de `len` : de toute façon l'arrondi final ne gardera que les `len` premiers bits. Si vous voulez, vous pouvez optimiser l'addition de sorte que le décalage nécessaire n'excède jamais la longueur `len` souhaitée.

☞ Soustraction et comparaison sont similaires à l'addition. Voir le fichier `rreal.cc`, qui implémente aussi des opérateurs d'entrée-sortie en base 10. Munie de ces opérations de base, notre classe `RReal` est déjà opérationnelle, et les exemples suivants montrent quelques applications.

☞ Soulignons que l'implémentation de la classe `RReal` proposée ici a pour seul but d'illustrer le principe. Pour une application professionnelle il faudra encore la peaufiner ; typiquement on privilégiera une implémentation plus complète et plus soigneusement optimisée, comme MPFR et MPFI qui sont actuellement en cours de développement (voir `www.mpfr.org` ou `info mpfr` en local).

2.8. Newton-Héron revisité. Le programme `rreal-test.cc` calcule la racine d'un nombre réel par la méthode de Newton-Héron en profitant des arrondis dirigés (voir le programme XVI.2 ci-dessous). En l'occurrence on utilise le mode d'arrondi vers le haut globalement. On choisit une valeur initiale plus grande que la racine cherchée puis on applique l'itération de Newton tant que le résultat s'améliore. On peut ainsi garantir une majoration de la racine exacte cherchée.

Exercice/P 2.4. Implémenter la méthode de Newton-Héron pour calculer $\sqrt[n]{a}$ avec n quelconque :

```
RReal power( RReal a, int n, Mode mode= stdMode, Len len= stdLen );
RReal root( const RReal& a, int n, Mode mode= stdMode, Len len= stdLen );
```

Indication. — Dans la fonction `power` on devrait attraper le cas $n < 0$. Ensuite vous pouvez écrire une simple boucle ou bien une puissance dichotomique si vous envisagez appeler la puissance pour n assez grand. Les arrondis individuels sont tous donnés par le mode spécifié si $a \geq 0$; le cas $a < 0$ par contre est particulier et devrait être traité à part. Dans la fonction `root` on devrait attraper le cas $a < 0$, puis $n < 0$ ou $n = 0$ ou $n = 1$. Ensuite on peut itérer Newton-Héron comme dans le cas $n = 2$ ci-dessus. Pour chaque calcul intermédiaire veillez particulièrement à choisir le bon mode d'arrondi.

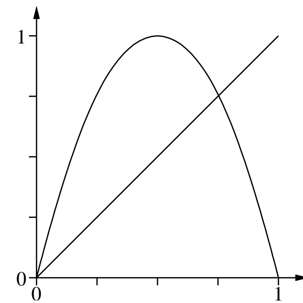
Programme XVI.2 Calcul fiable de la racine d'après Newton-Héron

```
RReal racine( const RReal& a, Mode mode= stdMode, Len len= stdLen )
{
  RReal u0, u1= (1+a)/2;          // Choix d'une valeur initiale.
  do {                            // L'itération de  u -> (u+a/u)/2 :
    u1.swap( u0 );                // échanger u0 et u1 (opération exacte),
    u1.div( a, u0, RNDU, len );   // u1 =  a / u0 , arrondi vers le haut,
    u1.add( u0, RNDU, len );      // u1 = u1 + u0 , arrondi vers le haut,
    u1.div( 2, RNDU, len );       // u1 = u1 / 2  , arrondi vers le haut.
  } while ( u1 < u0 );            // On continue tant que le résultat s'améliore.

  // On adapte la valeur finale selon le mode d'arrondi spécifié
  if( mode==RNDD || mode==RNDZ || mode==RNDN ) u1.div( a, u1, mode, len );
  return u1;
}
```

2.9. Instabilité numérique revisitée. L'itération de Newton-Héron pour calculer $\sqrt[n]{a}$ est, fort heureusement, numériquement stable : une petite perturbation de u_k (par une erreur d'arrondi, disons) restera petite, elle diminue même au cours des itérations suivantes. D'autres systèmes se comportent de manière contraire : des petites perturbations s'amplifient et « explosent » au cours des itérations suivantes. Ceci pose évidemment d'énormes problèmes pour le calcul numérique. On a déjà vu de tels exemples au chapitre XV, et nous allons regarder ici un exemple sur l'intervalle $[0,1]$.

Regardons une des fonctions les plus simples exhibant un comportement instable, $f\colon [0,1] \to [0,1]$, $f(x) = 4x(1-x)$. On peut interpréter l'itération $x \mapsto f(x)$ comme un *modèle de population*, quoique très simplifié. Penser à une population de bactéries contrainte à un environnement fixé. Toutes les heures on mesure $x_k \in [0,1]$, la quantité de bactéries par rapport à la population maximale. Pour x petit ($x \approx 0$) on a $f(x) \approx 4x$, donc la population quadruple à peu près. Quand elle devient trop grande, la nourriture commence à manquer et la croissance ralentit. Pour $x \geq \frac{3}{4}$ on a une situation de surpopulation (famine) et la population décroît. C'est un cas particulier de la *fonction logistique*, cf. `fr.wikipedia.org/wiki/Fonction_logistique`.



Exemple numérique. — Commençons l'itération par la valeur initiale $x_0 = 0.1$ disons. Avec notre petit programme `instable-naif.cc`, utilisant le type `double`, on trouve $x_{24} \approx 0.44165$ (un jour) puis $x_{48} \approx 0.03768$ (deux jours). Cette valeur serait donc notre prévision d'ici deux jours.

Phénomène d'instabilité. — Évidemment il faut s'attendre à certaines erreurs, déjà provenant de la valeur initiale x_0 (le « comptage » des bactéries), puis des arrondis lors du calcul itéré de $x_{k+1} = f(x_k)$. Est-ce que ces petites erreurs jouent un rôle ici ? Pour le tester, commençons l'itération par une valeur initiale légèrement perturbée, disons $x'_0 = 0,1000000001$. Le résultat est assez désillusionnant :

$x_0 = 0,1000000000$	$x'_0 = 0,1000000001$
$x_1 = 0,3600000000$	$x'_1 = 0,3600000003$
$x_2 = 0,9216000000$	$x'_2 = 0,9216000004$
$\dots$	$\dots$
$x_{24} = 0,4416454191$	$x'_{24} = 0,4388692524$
$x_{25} = 0,9863789715$	$x'_{25} = 0,9850521268$
$x_{26} = 0,0537419842$	$x'_{26} = 0,0588977372$
$\dots$	$\dots$
$x_{48} = 0,0376796921$	$x'_{48} = 0,7981824730$
$x_{49} = 0,1450397316$	$x'_{49} = 0,6443488512$
$x_{50} = 0,4960128314$	$x'_{50} = 0,9166536366$

Exercice/M 2.5. Expliquer pourquoi $|f'(x)| > 1$ implique qu'une petite perturbation de x est amplifiée. Expliquer qualitativement le comportement observé dans l'exemple précédent. La fonction f a deux points fixes : sont-ils stables ou instables ? Peut-on espérer une convergence vers un de ces points fixes ?

Exercice/P 2.6. Peut-on faire confiance en les valeurs x_{24} et x_{48} calculées par notre programme ? Comme chaque itération de la fonction f est susceptible d'introduire une petite erreur d'arrondi, c'est au moins douteux. Le type `double` a une mantisse de 53 bits ; refaire le calcul avec le type `long double` qui a une mantisse de 64 bits. Qu'observez-vous ? Peut-on raisonnablement trouver la valeur exacte de x_{48} ?

On peut obtenir des encadrements fiables avec la classe `RReal` en contrôlant judicieusement le mode d'arrondi. Le programme `instable-rreal.cc` met en œuvre cette idée. (Le lire puis tester.) Certes, le calcul fiable ne peut pas enlever l'instabilité qui est une caractéristique de la fonction f. En l'occurrence les encadrements successifs deviennent de moins en moins précis, mais on peut toujours garantir leur correction. Ainsi tout s'affiche honnêtement sur les encadrements calculés : quand le calcul tourne mal, au moins on en est averti.

Remarque 2.7. Le phénomène d'instabilité est souvent appelé *comportement chaotique*, terme largement popularisé. Tous les systèmes ne sont pas chaotiques, heureusement, mais certains modèles (biologiques, physiques, météorologiques, etc.) le sont. Dans un tel cas on retrouve toutes les difficultés numériques de notre exemple minimaliste, et d'autres encore. Pensez-y quand vous consultez la météo pour le week-end.

3. Arithmétique d'intervalles

Les arrondis dirigés permettent d'encadrer tout nombre réel $a \in \mathbb{R}$ par une minoration $\underline{a}$ et une majoration $\overline{a}$ dans l'ensemble $R \subset \mathbb{R}$ des nombres machine. Ceci revient à remplacer la valeur exacte a par un intervalle $\mathbf{a} = [\underline{a}, \overline{a}]$ contenant a. C'est nécessaire quand a n'est pas exactement représentable ($a \notin R$) ou notre calcul n'aboutit pas à déterminer a plus précisément.

Le calcul séparé d'une minoration puis d'une majoration est souvent assez répétitif et la notation en C++ devient vite assez lourde. Il est plus commode d'encapsuler les deux bornes en un seul objet. L'idée est simple : au lieu des valeurs réelles $a \in \mathbb{R}$ on stocke et travaille les intervalles $\mathbf{a} = [\underline{a}, \overline{a}]$ avec $\underline{a}, \overline{a} \in R$.

3.1. Arithmétique d'intervalles idéalisée.

Dans la suite nous allons regarder des intervalles compacts $[a,b] = \{x \in \mathbb{R} \mid a \le x \le b\}$ avec $a \le b$ dans $\mathbb{R}$. Nous n'autorisons pas l'intervalle vide ici, ni des intervalles infinis comme $[a, +\infty[$ ou $]-\infty, b]$ voire $]-\infty, +\infty[$. Ces intervalles peuvent être bien utiles, mais pour simplifier nous ne regardons que des intervalles compacts non vides. Par un léger abus de langage on confondera l'intervalle $[a,a] = \{a\}$ avec le point $a \in \mathbb{R}$.

Notation. Dans la suite une lettre en gras comme $\mathbf{a}$ dénote un intervalle, $\mathbf{a} = [\underline{a}, \overline{a}]$, alors que la lettre soulignée $\underline{a} = \min \mathbf{a}$ et la lettre surlignée $\overline{a} = \max \mathbf{a}$ dénotent les extrémités de l'intervalle. Cette écriture est assez intuitive et évite tout conflit avec d'éventuels indices. Nous écrivons $\mathbf{a} \in \mathbb{R}$ si $\underline{a} = \overline{a}$.

Proposition 3.1. *Si $\mathbf{a} = [\underline{a}, \overline{a}]$ est un intervalle compact et que $f \colon \mathbb{R} \to \mathbb{R}$ est une fonction continue, alors l'image $f(\mathbf{a}) = \{f(a) \mid a \in \mathbf{a}\}$ est à nouveau un intervalle compact.* □

Exercice/M 3.2. Montrer la proposition, puis vérifier les formules suivantes :
- Si f est croissante, alors $f([\underline{a}, \overline{a}]) = [f(\underline{a}), f(\overline{a})]$.
- Si f est décroissante, alors $f([\underline{a}, \overline{a}]) = [f(\overline{a}), f(\underline{a})]$.

Plus généralement, si f est monotone par morceaux, on peut calculer l'image $f([\underline{a}, \overline{a}])$ en mettant bout à bout les intervalles sur lesquels f est monotone. Considérons par exemple $f \colon \mathbb{R} \to \mathbb{R}$, $f(x) = |x|$:
- Si $0 \le \underline{a} \le \overline{a}$ alors $f([\underline{a}, \overline{a}]) = [f(\underline{a}), f(\overline{a})]$.
- Si $\underline{a} \le \overline{a} \le 0$ alors $f([\underline{a}, \overline{a}]) = [f(\overline{a}), f(\underline{a})]$.
- Si $\underline{a} \le 0 \le \overline{a}$ alors $f([\underline{a}, \overline{a}]) = [0, c]$ avec $c = \max(f(\underline{a}), f(\overline{a}))$.

Les mêmes formules sont valables pour toute fonction f qui est décroissante sur $\mathbb{R}_-$ et croissante sur $\mathbb{R}_+$, comme par exemple $f(x) = x^n$ pour $n = 2, 4, 6, \dots$.

Définition 3.3. Pour deux intervalles $\mathbf{a}$ et $\mathbf{b}$ on définit les opérations élémentaires $+, -, *, /$ comme suit : si $\circ$ dénote une de ces opérations, alors on pose $\mathbf{a} \circ \mathbf{b} := \{a \circ b \mid a \in \mathbf{a}, b \in \mathbf{b}\}$.

Proposition 3.4. *L'arithmétique d'intervalles obéit aux règles suivantes :*

$$\mathbf{a}+\mathbf{b} = [\underline{a}+\underline{b}, \overline{a}+\overline{b}] \qquad \mathbf{a}*\mathbf{b} = [\ \min\{\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab}\}\ ,\ \max\{\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab}\}\ ]$$

$$-\mathbf{b} = [-\overline{b}, -\underline{b}] \qquad \mathbf{b}^{-1} = [\overline{b}^{-1}, \underline{b}^{-1}] \quad si \quad 0 \notin \mathbf{b}$$

$$\mathbf{a}-\mathbf{b} = \mathbf{a}+(-\mathbf{b}) = [\underline{a}-\overline{b}, \overline{a}-\underline{b}] \qquad \mathbf{a}/\mathbf{b} = \mathbf{a}*\mathbf{b}^{-1}$$

Exercice/M 3.5. Vérifier ces règles de calculs, dont seule la multiplication est délicate. Justifier d'abord que $\mathbf{c} = \mathbf{a}*\mathbf{b}$ est un intervalle compact, et que $\min \mathbf{c}$ et $\max \mathbf{c}$ sont toujours atteints dans un des quatre coins du rectangle $\mathbf{a}\times\mathbf{b} \subset \mathbb{R}^2$, donc $\underline{c} = \min\{\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab}\}$ et $\overline{c} = \max\{\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab}\}$.
Optimisation. — Pour une implémentation efficace on peut prédire quel(s) produit(s) réalisent le minimum ou maximum : l'intervalle $\mathbf{a}$ peut être ou strictement positif ($0 < \underline{a} \leq \overline{a}$, noté $0 < \mathbf{a}$), ou strictement négatif ($\underline{a} \leq \overline{a} < 0$, noté $\mathbf{a} < 0$), ou bien il contient zéro ($0 \in \mathbf{a}$ équivaut à $\underline{a} \leq 0 \leq \overline{a}$). De même pour l'intervalle $\mathbf{b}$; il y a donc au total 9 cas à distinguer. Dans 8 cas, le calcul de $\underline{c}$ et $\overline{c}$, respectivement, ne nécessite qu'une seule multiplication. Le dernier cas $\underline{a} < 0 < \overline{a}$ et $\underline{b} < 0 < \overline{b}$ est plus délicat et nécessite quatre multiplications.

Proposition 3.6. *L'arithmétique d'intervalles se réjouit des propriétés suivantes :*

Associativité :	$(\mathbf{a}+\mathbf{b})+\mathbf{c} = \mathbf{a}+(\mathbf{b}+\mathbf{c})$	$(\mathbf{a}*\mathbf{b})*\mathbf{c} = \mathbf{a}*(\mathbf{b}*\mathbf{c})$
Commutativité :	$\mathbf{a}+\mathbf{b} = \mathbf{b}+\mathbf{a}$	$\mathbf{a}*\mathbf{b} = \mathbf{b}*\mathbf{a}$
Élément neutre :	$\mathbf{0}+\mathbf{a} = \mathbf{a}+\mathbf{0} = \mathbf{a}$	$\mathbf{1}*\mathbf{a} = \mathbf{a}*\mathbf{1} = \mathbf{a}$
Sous-distributivité :	$\mathbf{a}*(\mathbf{b}+\mathbf{c}) \subseteq (\mathbf{a}*\mathbf{b})+(\mathbf{a}*\mathbf{c})$	

En général la dernière inclusion peut être stricte, mais on a égalité $\mathbf{a}*(\mathbf{b}+\mathbf{c}) = (\mathbf{a}*\mathbf{b})+(\mathbf{a}*\mathbf{c})$ *si* $\mathbf{a} \in \mathbb{R}$ *ou* $\mathbf{b}, \mathbf{c} \geq 0$ *ou* $\mathbf{b}, \mathbf{c} \leq 0$. *Pour un intervalle* $\mathbf{a}$ *qui n'est pas réduit à un point il n'existe pas d'intervalle inverse, ni pour l'addition ni pour la multiplication : on a* $\mathbf{a}+(-\mathbf{a}) \supsetneqq \mathbf{0}$ *et de même* $\mathbf{a}*\mathbf{a}^{-1} \supsetneqq \mathbf{1}$. □

Exercice/M 3.7. Les propositions précédentes montrent que l'arithmétique d'intervalles possède de bonnes propriétés, mais aussi qu'il faut s'habituer aux exceptions. Essayez de construire un exemple où $\mathbf{a}*(\mathbf{b}+\mathbf{c}) \neq (\mathbf{a}*\mathbf{b})+(\mathbf{a}*\mathbf{c})$.

Exercice/M 3.8. A-t-on toujours $\mathbf{a}+\mathbf{a} = 2\mathbf{a}$? Pour $n \in \mathbb{N}$ on définit $\mathbf{a}^n = \{a^n \mid a \in \mathbf{a}\}$. Montrer que $\mathbf{a}*\mathbf{a} = \mathbf{a}^2$, ou plus généralement $\mathbf{a}^m * \mathbf{a}^n = \mathbf{a}^{m+n}$, si $\mathbf{a} \geq 0$ ou $\mathbf{a} \leq 0$. A-t-on toujours $\mathbf{a}*\mathbf{a} \supseteq \mathbf{a}^2$, ou plus généralement $\mathbf{a}^m * \mathbf{a}^n \supseteq \mathbf{a}^{m+n}$? Construire un exemple où $\mathbf{a}*\mathbf{a} \neq \mathbf{a}^2$. Ceci montre que deux expressions algébriques peuvent définir la même application $\mathbb{R} \to \mathbb{R}$, mais différentes applications sur l'ensemble des intervalles.

3.2. Arithmétique d'intervalles arrondie. Jusqu'à présent l'arithmétique de $\mathbb{R}$ a été utilisée pour décrire l'arithmétique d'intervalles. Une implémentation doit prendre en compte l'arithmétique arrondie disponible sur ordinateur : quand on implémente l'arithmétique d'intervalles sur ordinateur, il est naturel de stocker tout intervalle $\mathbf{a} = [\underline{a}, \overline{a}]$ par ses extrémités $\underline{a}$ et $\overline{a}$. Il s'agit de deux nombres à virgule flottante, et il faut passer des intervalles idéalisés aux intervalles arrondis :

- Pour un intervalle idéalisé $\mathbf{a} = [\underline{a}, \overline{a}]$ les extrémités $\underline{a}, \overline{a} \in \mathbb{R}$ ne sont en général pas exactement représentables par des nombres machines, c'est-à-dire $\underline{a} \notin R$ ou $\overline{a} \notin R$. On passe donc à un intervalle (légèrement) plus grand $\mathbf{b} = [\underline{b}, \overline{b}]$ dont les extrémités $\underline{b} = \lfloor \underline{a} \rfloor_R$ et $\overline{b} = \lceil \overline{a} \rceil_R$ sont des nombres machines. À noter que les arrondis dirigés assurent l'inclusion $\mathbf{a} \subseteq \mathbf{b}$: pour cela il faut arrondir $\underline{a}$ vers $-\infty$ et $\overline{a}$ vers $+\infty$. On parle de *l'arrondi vers l'extérieur*.
- Même si $\mathbf{a} = [\underline{a}, \overline{a}]$ est exactement représentable par des nombres machines $\underline{a}, \overline{a} \in R$, l'application d'une fonction f produit en général un intervalle $f(\mathbf{a})$ que ne l'est plus. Considérons une fonction croissante, comme $f(x) = \exp(x)$: au lieu de l'intervalle $f([\underline{a}, \overline{a}]) = [f(\underline{a}), f(\overline{a})]$ nous allons nous contenter d'un intervalle (légèrement) plus grand $\mathbf{b} = [\underline{b}, \overline{b}]$ dont les extrémités $\underline{b} \leq f(\underline{a})$ et $\overline{b} \geq f(\overline{a})$ sont des valeurs approchées, convenablement arrondies. À nouveau les arrondis dirigés vers l'extérieur assurent que $f(\mathbf{a}) \subseteq \mathbf{b}$.

☞ *L'arithmétique d'intervalles arrondie repose sur le principe que tout calcul retourne un encadrement garanti du résultat exact.* ✍

L'implémentation nécessite donc l'arrondi dirigé : en arrondissant vers l'extérieur on garantit toujours que l'intervalle calculé contient tous les résultats possibles à partir des données d'entrée.

3.3. Une implémentation « faite maison ». Afin d'avoir une écriture commode nous souhaitons implémenter une classe `Interval` qui permette d'écrire le code suivant :

```
Interval rayon, pi( 3.14, 3.15 );  // encadrement grossier mais correct de pi
cin >> rayon;                      // donnée initiale provenant d'une mesure
Interval aire= pi * r * r;         // calcul selon l'arithmétique d'intervalles
cout << "aire = " << aire << endl; // affichage du résultat sous forme d'intervalle
```

Le programme XVI.3 ci-dessous montre le début d'une telle classe, encore rudimentaire, qui met au point l'arithmétique d'intervalles selon les règles développées ci-dessus. Pour les arrondis dirigés nous utilisons notre classe `RReal` expliquée plus haut.

Remarque 3.9. Un objet de la classe `Interval` est donné par une paire `(mini,maxi)` où `mini` et `maxi` sont des nombres réels représentables par le type de base, en l'occurrence notre classe `RReal`. Vous constaterez une particularité dans le design de la classe `Interval` : nous exigeons toujours que `mini` $\le$ `maxi`. Afin de maintenir cet invariant nous sommes obligés de déclarer les éléments `mini` et `maxi` comme étant privés à la classe. Ceci empêche d'accéder directement à ces éléments – prudence oblige. Ensuite toutes nos opérations s'engagent à produire des résultats avec `mini` $\le$ `maxi`.

Exercice/P 3.10. Le fichier `interval.cc` contient une implémentation plus complète ; essayez de comprendre sa structure et de vous familiariser avec les diverses fonctions qu'elle offre. (Lire aussi puis tester `interval-test.cc`.) En particulier vous y trouverez une implémentation de la multiplication, qui minimise le nombre de multiplications à effectuer sur le type `RReal`. Essayez de comprendre puis de vérifier son fonctionnement (cf. l'exercice 3.5).

Remarque 3.11. Afin d'utiliser l'écriture usuelle des opérations élémentaires `a+b`, `a-b`, `a*b`, `a/b` il faut encore définir ces opérateurs en C++. Ceci n'est qu'une simple reformulation des fonctions déjà implémentées :

```
Interval add( const Interval& a, const Interval& b, Len len= stdLen )
{ Interval c; c.add( a, b, len ); return c; }

Interval operator+ ( const Interval& a, const Interval& b )
{ Interval c; c.add( a, b ); return c; }
```

À noter que sous cette forme on crée un objet temporaire, ici nommé `c`, puis on lui affecte la valeur calculée. Sous la première forme `add(a,b,len)` on peut optionnellement spécifier la longueur de la mantisse, c'est-à-dire la précision souhaitée. Sous la seconde forme `a+b` cette information n'apparaît plus explicitement : il est sous-entendu que l'on utilise la valeur spécifiée dans la variable globale `stdLen`.

Exercice/P 3.12. Après les opérations élémentaires, les fonctions usuels peuvent être implémentées pour les intervalles. Le programme `interval.cc` contient deux exemples faciles :

```
Interval sqr( const Interval& a );
Interval sqrt( const Interval& a );
```

La fonction `sqr` applique la fonction $x \mapsto x^2$ à un intervalle, alors que `sqrt` applique $x \mapsto \sqrt{x}$ pour $x \ge 0$. À noter que $x \mapsto x^2$ n'est que monotone par morceaux, donc il faut distinguer plusieurs cas. Rappelons aussi qu'en général $\mathbf{a}^2 \neq \mathbf{a} * \mathbf{a}$, il est donc fort utile de disposer des implémentations faites sur mesure pour les intervalles. Plus généralement, vous y trouverez une implémentation des fonctions $x \mapsto x^n$ et $x \mapsto \sqrt[n]{x}$:

```
Interval power( const Interval& a, long n );
Interval root( const Interval& a, long n );
```

3.4. Exemples d'utilisation.

Exercice/P 3.13. Les fichiers `somme-rreal.cc` et `somme-interval.cc` implémentent le calcul de $s_n = \sum_{k=1}^{n} \frac{1}{k^2}$ en utilisant les types `RReal` et `Interval`, respectivement. Lisez attentivement ces petits exemples pour vous familiariser avec l'usage de ces types. Vérifiez que c'est strictement la même démarche et le même résultat. La seule différence est que l'écriture sous forme d'intervalle est plus commode.

Exercice/P 3.14. Vérifier, tester, puis comparer `instable-rreal.cc` et `instable-interval.cc`.
Attention. — Appliquer la fonction $f\colon \mathbb{R} \to \mathbb{R}$, $f(x) = 4x(1-x)$ à un intervalle $\mathbf{a}$ n'est pas du tout équivalent à calculer $\mathbf{4} * \mathbf{a} * (\mathbf{1} - \mathbf{a})$. Discuter cette différence et l'illustrer par des exemples.

Programme XVI.3 La classe `Interval` — arithmétique d'intervalles

```
class Interval
{
private:
  // Données privées: le minimum et le maximum de l'intervalle en question
  RReal mini, maxi;

  // Affectation sans contrôle (à usage privé uniquement)
  Interval& assign( const RReal& a, const RReal& b )
    { mini= a; maxi= b; return *this; }

public:
  // Constructeur à partir d'un intervalle (contrôle inutile)
  Interval( const Interval& interval )
    : mini( interval.mini ), maxi( interval.maxi ) {}

  // Constructeur à partir de deux bornes (avec contrôle)
  Interval( const RReal& a, const RReal& b )
    : mini(a), maxi(b) { if( mini > maxi ) mini.swap( maxi ); }

  // Constructeur à partir de deux bornes (avec contrôle)
  Interval( double a, double b )
    { if(a>b) std::swap(a,b); mini= RReal(a,RNDD); maxi= RReal(b,RNDU); }

  // Accès contrôlé aux informations -- en lecture seulement !
  const RReal& min() const { return mini; }   // lire le minimum
  const RReal& max() const { return maxi; }   // lire le maximum

  // Addition de deux intervalles
  Interval& add( const Interval& a, const Interval& b, Len len= stdLen )
    { return assign( Numeric::add( a.mini, b.mini, RNDD, len ),
                     Numeric::add( a.maxi, b.maxi, RNDU, len ) ); }

  // Soustraction de deux intervalles
  Interval& sub( const Interval& a, const Interval& b, Len len= stdLen )
    { return assign( Numeric::sub( a.mini, b.maxi, RNDD, len ),
                     Numeric::sub( a.maxi, b.mini, RNDU, len ) ); }

  // La multiplication de deux intervalles
  Interval& mul( const Interval& a, const Interval& b, Len len= stdLen );

  // Inversion d'un intervalle ne contenant pas zéro
  Interval& inv( const Interval& a, Len len= stdLen )
    {
      if( sign(a.mini) <= 0 && sign(a.maxi) >= 0 )
        { cerr << "Interval division by zero!" << endl; exit(1); }
      return assign( Numeric::div( 1, a.maxi, RNDD, len ),
                     Numeric::div( 1, a.mini, RNDU, len ) );
    }

  // La division a/b est ramenée à une multiplication  a * (1/b).
  Interval& div( const Interval& a, const Interval& b, Len len= stdLen )
    { return mul( a, inv( b, len ), len ); }

  // L'entrée-sortie
  void write( ostream& out ) const;
  void read( istream& in );
};
```

4. Applications

Ce chapitre met à votre disposition les classes `RReal` et `Interval` pour le calcul arrondi en précision arbitraire avec des arrondis dirigés. Ceci permet d'établir des résultats numériques rigoureux, c'est-à-dire mathématiquement prouvables. Ces outils sont particulièrement bien adaptés pour évaluer des séries, dont ce paragraphe vous propose quelques exemples classiques. Dans les exercices suivants vous pouvez donc expérimenter avec nos classes `RReal` et `Interval` faites maison.

4.1. Retour sur les problèmes du chapitre XV. Avec nos nouveaux outils, vous pouvez reprendre quelques calculs du chapitre XV et les réécrire plus savamment comme calculs fiables : l'instabilité de Fibonacci (§2), l'évaluation du polynôme de Rump (§4.5), puis les pièges à éviter (§5).

Question 4.1. Analysez en particulier le comportement de l'arithmétique d'intervalles lors d'une perte de chiffres significatifs (phénomène d'annulation, voir chapitre XV,§5.4). Le problème persiste-t-il ? Quel est alors l'avantage de l'arithmétique d'intervalles dans ce cas ?

Exercice/P 4.2. Les équations quadratiques sont reprises dans `chap14/quadratique.cc`. Testez puis discutez quels problèmes sont ainsi résolus, et lesquels persistent ou s'ajoutent... Optimisez ce programme pour qu'il devienne robuste et produise des résultats satisfaisants.

4.2. La fonction zéta de Riemann.

Exercice/P 4.3. La série $\sum \frac{1}{k}$ diverge, c'est-à-dire les sommes partielles $s_n = \sum_{k=1}^{n} \frac{1}{k}$ croissent sans borne. Pourtant elles deviennent stationnaires quand on les calcule naïvement sur ordinateur ! Essayez de prédire la valeur stationnaire pour le type `float`. Le vérifier avec le programme `chap13/divergence.cc`.

Ce phénomène bizarre persistera-t-il quand on encadre s_n avec la classe `RReal`, ou plus commodément avec `Interval` ? Essayez de prédire le résultat, puis testez-le empiriquement si cela vous intéresse. Pour imiter le type `float` on prendra des mantisses de 24 bits, mais vous pouvez varier ce choix.

☞ *Se méfier d'une apparente « convergence numérique » sans contrôle d'erreur.* ✍

Exercice/P 4.4. La série $\sum_{k=1}^{\infty} \frac{1}{k^2}$ converge. Pour approcher la limite, en utilisant le type `float`, calculer $\sum_{1}^{n} \frac{1}{k^2}$ dans le sens des indices croissants, puis $\sum_{n}^{1} \frac{1}{k^2}$ dans le sens des indices décroissants (voir `somme-naive.cc`). Vu la nature des erreurs d'arrondi, quelle approche vous semble plus exacte ?

Comparer avec les encadrements fiables : lire puis tester les deux programmes `somme-rreal.cc` et `somme-interval.cc`. Les deux encadrements ainsi obtenus sont corrects, mais l'un est plus fin que l'autre ! Le tester puis expliquer le résultat.

☞ *Lors d'une sommation numérique l'ordre des termes peut influencer le résultat.* ✍

Exercice/M 4.5. Afin d'encadrer la valeur de $\zeta(2) = \sum_{k=1}^{\infty} \frac{1}{k^2}$ il ne suffit évidemment pas de calculer la somme partielle $s_n = \sum_{k=1}^{\infty} \frac{1}{k^2}$, il faut aussi majorer le reste $r_n = \sum_{k=n+1}^{\infty} \frac{1}{k^2}$. Montrer que $\frac{1}{n+1} < r_n < \frac{1}{n}$:

(1) via l'encadrement $\frac{1}{k} - \frac{1}{k+1} < \frac{1}{k^2} < \frac{1}{k-1} - \frac{1}{k}$ puis une somme télescopique,

(2) via l'encadrement $\int_{k}^{k+1} \frac{1}{x^2} dx < \frac{1}{k^2} < \int_{k-1}^{k} \frac{1}{x^2} dx$ puis la somme des intégrales.

En déduire un programme qui calcule un encadrement fiable de $\zeta(2)$ en fonction de n. (On sait d'ailleurs que $\zeta(2) = \pi^2/6$, mais ce beau résultat ne joue pas de rôle ici.)

☞ *C'est la majoration du reste qui fait d'une série $\sum_{k=1}^{\infty} a_k$ une méthode praticable pour calculer une valeur approchée de la somme.* ✍

Exercice/M 4.6. En généralisant l'exercice précédent, écrire un programme qui calcule un encadrement de $\zeta(3) = \sum_{k=1}^{\infty} \frac{1}{k^3}$ en fonction de n. Si vous voulez, vous pouvez étendre cette approche afin d'encadrer $\zeta(s)$ pour $s = 2, 3, 4, 5, \ldots$. Est-ce praticable pour tout $s > 1$ et $\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s}$?

4.3. Séries alternées : sin, cos, arctan, etc. Pour une suite $(u_k)_{k\in\mathbb{N}}$ de nombres réels $u_k \in \mathbb{R}$ on écrit $u_k \searrow u$ si la suite est décroissante, $u_0 \ge u_1 \ge u_2 \ge \dots$, avec pour limite $\lim u_k = \inf u_k = u$. De manière analogue on écrit $u_k \nearrow u$ si la suite est croissante, $u_0 \le u_1 \le u_2 \le \dots$, avec $\lim u_k = \sup u_k = u$.

Théorème 4.7. *Si $u_k \searrow 0$ alors la série alternée $\sum_{k=0}^{\infty}(-1)^k u_k$ converge, c'est-à-dire les sommes partielles $s_n = \sum_{k=0}^{n}(-1)^k u_k$ convergent vers un nombre réel $s \in \mathbb{R}$. Plus précisément on a $s_{2n} \searrow s$ et $s_{2n+1} \nearrow s$. Ainsi on obtient des encadrements $s_{2n-1} \le s \le s_{2n}$ de plus en plus fins de la valeur $s = \sum_{k=0}^{\infty}(-1)^k u_k$.*

Exercice/M 4.8. Montrer ce théorème.

Exercice/P 4.9. Pour appliquer ce théorème, par exemple à la série $\sum_{k=1}^{\infty}(-1)^{k-1}\frac{1}{\sqrt{k}}$, on pourra calculer le terme général par la fonction

```
Interval terme_general( long n ) { return 1/sqrt(Interval(n)); }
```

Comme les séries alternées sont assez fréquentes, il sera utile de disposer d'une fonction générique

```
Interval somme_alternee( long debut, long fin );
```

qui donne un encadrement prouvable de $s = \sum_{k=k_0}^{\infty}(-1)^{k-k_0}u_k$. Le choix de k_0 est commode si vous voulez sommer à partir d'un indice quelconque, en l'occurrence $k_0 = 1$ et non $k_0 = 0$. Pour simplifier on pourra sommer un nombre pair de terme, puis ajouter la marge d'erreur obtenue par le terme suivant.

Plus souvent il est préférable de prescrire une borne $\varepsilon > 0$ et de sommer jusqu'à ce que $u_{n+1} \le \varepsilon$. Implémenter une telle fonction

```
Interval somme_alternee( long debut, const RReal& eps );
```

Si possible, veillez à ne pas évaluer inutilement la fonction `terme_general`.

Exercice/P 4.10. Implémenter les fonctions

$$\cos(x) = \sum_{k=0}^{\infty}(-1)^k \frac{x^{2k}}{(2k)!} \qquad \text{et} \qquad \sin(x) = \sum_{k=0}^{\infty}(-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

pour $x \in [0,1]$ à une précision $\varepsilon = 2^{-\texttt{len}}$ près. En supposant que nous disposons d'une bonne approximation de π, comment implémenter $\sin(x)$ et $\cos(x)$ efficacement pour tout $x \in \mathbb{R}$?

Exercice/P 4.11. On se propose de calculer $g(x) = \int_0^x e^{-t^2}dt$. Développer $f(t) = e^{-t^2}$ en une série entière, intégrer terme par terme pour obtenir la série entière pour la fonction primitive g avec $g' = f$ et $g(0) = 0$. Justifier ce résultat. Implémenter ainsi le calcul de $g(x)$ pour $x \in [0,1]$ à une précision $\varepsilon = 2^{-\texttt{len}}$ près.

Exercice/P 4.12. Pour $x \in [-1,1]$ montrer que la série $\sum_{k=1}^{\infty}(-1)^{k+1}\frac{x^{2k-1}}{2k-1}$ converge vers $\arctan(x)$: développer $\frac{d}{dx}\arctan(x) = \frac{1}{1+x^2}$ en une série entière, puis intégrer terme par terme pour obtenir la série entière de la fonction primitive $\arctan(x)$. Justifier ce résultat pour $-1 < x < 1$, puis aux extrémités $x = \pm 1$.

Est-ce une méthode praticable pour calculer $\arctan(x)$ avec $x \in [0, \frac{1}{2}]$? Implémenter ce calcul à une précision $\varepsilon = 2^{-\texttt{len}}$ près. Combien de termes sont nécessaires ? Y a-t-il un problème d'annulation de termes consécutifs et de perte de chiffres significatifs ? Est-ce encore praticable proche de l'extrémité $x = 1$? Comment implémenter $\arctan(x)$ efficacement pour tout $x \in \mathbb{R}$?

La sommation numérique est efficace seulement si la série converge rapidement.

4.4. Calcul de π.

Exercice/P 4.13. Dans le projet IV on a développé un calcul de $\pi = 2\sum_{k=0}^{\infty}\frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)}$ à 10000 décimales exactes — avec preuve de correction ! Vous pouvez refaire, si vous voulez, un tel calcul avec un type de nombres à virgule flottante convenable. *Indication.* — La série peut être sommée dans les deux sens. Elle peut aussi être évaluée par la méthode de Horner. Choisir une de ces méthodes, ou bien tester les toutes.

Exercice/P 4.14. Vérifier que $x := e^{\pi\sqrt{163}} \approx 262\,537\,412\,640\,768\,743,999\,999\,999\,999\dots$ est invraisemblablement proche de l'entier $y := 262\,537\,412\,640\,768\,744$. On soupçonne néanmoins que $x \ne y$: peut-on implémenter un calcul qui permet de prouver que $x \ne y$? Si l'on avait $x = y$, serait-il possible de prouver cette égalité par un calcul numérique sur ordinateur ?

PROJET XVI

Calcul fiable de exp et log

Objectif

- ▶ Implémenter correctement et efficacement les fonctions exp et log.
- ▶ Majorer l'erreur de l'approximation afin de garantir un encadrement.
- ▶ Garantir un calcul numérique fiable grâce aux arrondis dirigés.

On appelle *usuelles* les fonctions exp et log, les fonctions trigonométriques sin, cos, tan avec leurs inverses arcsin, arccos, arctan, ainsi que les fonctions hyperboliques sinh, cosh, tanh avec leurs inverses asinh, acosh, atanh (et d'autres encore selon le contexte). Toute bibliothèque numérique qui se respecte offre ces fonctions, comme par exemple `<cmath>` pour le type `double`. Dans ce projet on implémentera les deux premières fonctions, exp et log, certes les plus simples mais suffisamment représentatives pour toute la famille. En choisissant judicieusement les modes d'arrondi on peut garantir la correction du résultat sous forme d'un encadrement fiable, à n'importe quelle précision souhaitée.

1. Calcul de l'exponentielle

On implémente d'abord la fonction $\exp\colon \mathbb{R} \to \mathbb{R}_+$ définie par $\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$. Comme on ne peut calculer que des sommes finies, on évaluera le polynôme $s_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}$ pour un certain rang n, encore à déterminer. Pour le calcul numérique deux problèmes se posent. Si $|x|$ est grand, les premiers termes croissent avant que la factorielle $k!$ ne l'emporte. Pire encore, pour $x < 0$ les termes sont de signes alternés, ce qui entraîne de sérieuses difficultés d'annulation (perte de chiffres significatifs, voir chapitre XV, §5.5).

Exercice/M 1.1 (réduction de l'argument)**.** Pour les raisons évoquées on évaluera la série seulement pour $x \in [0,1]$ où le comportement numérique est excellent. Notre première tâche sera de majorer le reste $r_n(x) = \sum_{k=n+1}^{\infty} \frac{x^k}{k!}$. Évidemment on a $r_n(x) \geq \frac{x^{n+1}}{(n+1)!}$; montrer la majoration $r_n(x) \leq \frac{2x^{n+1}}{(n+1)!} =: \varepsilon_n$.

Exercice/P 1.2 (choix du rang n)**.** Pour $x \in [0,1]$ on sait que $\exp(x) \in [1,3]$. Afin de calculer $\exp(x)$ avec une mantisse de longueur `len`, l'écart toléré sera donc $\varepsilon = 2^{-\texttt{len}}$. (*Astuce.* — Comment construire *sans calcul* ce nombre ε en utilisant le type `RReal` ?) Le choix $n = \max\{5, \texttt{len}\}$ suffira largement mais sera inutilement grand. Écrire une boucle qui calcule successivement ε_n et qui détermine le plus petit n tel que $\varepsilon_n \leq \varepsilon$. (*Astuce.* — Le passage de ε_{n-1} à ε_n ne nécessite que deux opérations arithmétiques.)

Exercice/P 1.3 (implémentation, cas restreint)**.** En utilisant les exercices précédents, écrire une fonction

```
RReal exp1( const RReal& x, Mode mode= stdMode, Len len= stdLen )
```

qui calcule une approximation de $\exp(x)$ pour $x \in [0,1]$ en choisissant judicieusement les arrondis.
Indication. — Afin d'évaluer $s_n(x)$ on utilisera la méthode de Horner :

$$s_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!} = \left(\left(\dots\left(\left(\left(\frac{x}{n}+1\right)\frac{x}{n-1}+1\right)\frac{x}{n-2}+1\right)\dots\right)\frac{x}{2}+1\right)\frac{x}{1}+1$$

Tous les facteurs $\frac{x}{k}$ sont positifs, donc pour obtenir un certain arrondi `RNDD`, `RNDU`, `RNDZ`, ou `RNDI` dans le résultat final, il suffit d'appliquer ce même mode d'arrondi à chaque opération élémentaire intermédiaire.
Attention. — Pour `RNDU` et `RNDI` n'oubliez pas d'ajouter la marge d'erreur ε_n ou ε à la fin de la fonction.

Exercice/P 1.4 (implémentation, cas général)**.** Afin de calculer $\exp(x)$ pour $x \in \mathbb{R}$ quelconque on se ramène au cas restreint où x est dans l'intervalle $[0,1]$:

(1) Pour $x < 0$ on utilise $\exp(x) = 1/\exp(-x)$. *Astuce.* — Pour un mode d'arrondi `mode` l'inverse est donné par `!mode` : ceci échange `RNDD` et `RNDU`, ainsi que `RNDZ` et `RNDI`.

(2) Pour $x \geq 1$ on applique $\exp(x) = \exp(x/2^k)^{(2^k)}$. *Astuce.* — La fonction `x.magnitude()` aidera à déterminer k. Elle est définie et documentée dans le fichier `rreal.cc`. La division par 2^k n'est qu'une simple soustraction des exposants. La puissance $y^{(2^k)}$ ne nécessite que k opérations.

En tenant compte du mode d'arrondi choisi, écrire ainsi deux fonctions

```
RReal exp( RReal x, Mode mode= stdMode, Len len= stdLen );  // entre 15 et 20 lignes
Interval exp( const Interval& a, Len len= stdLen );         // une seule ligne suffit
```

Vérification. — Encadrer $e = \exp(1)$ à 100 décimales, soit 330 bits environ. Pour vérification comparer avec les résultats du chapitre IV, §2, ou une autre source suffisamment sérieuse. De même, par souci de tester votre implémentation, encadrer aussi d'autres valeurs, comme e^{100} et e^{-100} par exemple.

Exercice/P 1.5. Calculer un encadrement fiable de $\sqrt{163}$, de π, puis de $x := \exp(\pi\sqrt{163})$. Adapter la précision afin de prouver que x n'est pas un nombre entier. (Voir l'exercice 4.14 du chapitre XVI,)

2. Calcul du logarithme

Comme inverse de l'exponentielle on se propose d'implémenter la fonction $\log\colon \mathbb{R}_+ \to \mathbb{R}$. Pour tout $t \in]-1,+1[$ on a $\log(1+t) = \sum_{k=1}^{+\infty} (-1)^{k+1} \frac{t^k}{k}$, mais cette série converge trop lentement proche des bords ± 1 (voir le chapitre XV, exercices 6.6 et 6.7). Afin d'accélérer la convergence on passe donc à la série

$$\log\left(\frac{1+t}{1-t}\right) = 2\left(t + \frac{t^3}{3} + \frac{t^5}{5} + \dots\right) = 2t \sum_{k=0}^{\infty} \frac{t^{2k}}{2k+1}$$

Exercice/M 2.1 (transformation entre x et t)**.** Pour $x \in \mathbb{R}_+$ expliciter l'unique valeur $t \in]-1,+1[$ telle que $x = \frac{1+t}{1-t}$. Implémenter ce calcul $x \mapsto t$ en tenant compte du mode d'arrondi choisi.

Dans la suite on considérera seulement $x \in [1,2]$, ce qui se traduit en $t \in [0, \frac{1}{3}]$ (le vérifier). Sur cette intervalle notre série converge très rapidement : un peu mieux que la série géométrique à base $t^2 \leq \frac{1}{9}$.

Exercice/M 2.2 (majoration de l'erreur)**.** Pour $t \geq 0$ notre série donne la valeur $\log\left(\frac{1+t}{1-t}\right) \geq 2t$. Pour une mantisse de longueur `len` on tolère donc une erreur absolue de $2t \cdot 2^{-\mathtt{len}}$. D'autre part la somme partielle $s_n(t) = 2t \sum_{k=0}^{n} \frac{t^{2k}}{2k+1}$ laisse une erreur $r_n(t) = 2t \sum_{k=n+1}^{+\infty} \frac{t^{2k}}{2k+1}$. Donner une majoration commode ε_n de $\sum_{k=n+1}^{+\infty} \frac{t^{2k}}{2k+1}$. Bien que le choix $n = \mathtt{len}/3 - 1$ suffise toujours, on peut faire mieux si t est petit. Comment déterminer efficacement le plus petit indice n tel que $\varepsilon_n < 2^{-\mathtt{len}}$?

Exercice/P 2.3 (implémentation, cas restreint)**.** En utilisant les exercices précédents, écrire une fonction

```
RReal log1( const RReal& x, Mode mode= stdMode, Len len= stdLen )
```

qui calcule une approximation de $\log(x)$ pour $x \in [1,2]$ en tenant compte du mode d'arrondi choisi.

Indication. — Comme toujours il vaut mieux évaluer $s_n(t) = 2t \sum_{k=0}^{n} \frac{t^{2k}}{2k+1}$ par la méthode de Horner. Pour `RNDU` et `RNDI` il faut ajouter la majoration du reste à la fin.

Vérification. — Encadrer ainsi $\log(2)$ à 100 décimales. Comparer avec la valeur fournie par la bibliothèque `cmath` ; combien de décimales sont exactes ? Si possible comparer avec une source plus sérieuse.

Exercice/P 2.4 (implémentation, cas général)**.** Afin de calculer $\log(x)$ pour $x \in \mathbb{R}_+$ quelconque on se ramène au cas restreint où x est dans l'intervalle $[1,2]$:

(1) Pour $x < 1$ on utilise $\log(x) = -\log(1/x)$.

(2) Pour $x \geq 2$ on applique $\log(x) = \log(x/2^k) + k\log(2)$.

En déduire des fonctions

```
RReal log( RReal x, Mode mode= stdMode, Len len= stdLen );  // entre 15 et 20 lignes
Interval log( const Interval& a, Len len= stdLen );         // une seule ligne suffit
```

Vérification. — Tester si les implémentations de exp et log donnent des encadrements cohérents : pour $x \in \mathbb{R}$ calculer un encadrement $\underline{y} \lessapprox \exp(x) \lessapprox \overline{y}$, puis calculer $\underline{z} \lessapprox \log(\underline{y})$ arrondi vers le bas et $\overline{z} \gtrapprox \log(\overline{y})$ arrondi vers le haut. Trouve-t-on $\underline{z} \leq x \leq \overline{z}$ comme il faut ? L'écart $\overline{z} - \underline{z}$ est-il acceptable ?

Exercice/P 2.5. Encadrer $\sqrt{2}$ à 100 décimales en calculant $2^{\frac{1}{2}} = \exp(\log(2)/2)$. Comparer avec la méthode de Newton-Héron ; quelle méthode vous semble préférable quant à la précision et l'efficacité ?

Exercice/P 2.6. À partir des fonctions `exp` et `log` écrire une fonction

```
RReal power( const RReal& x, const RReal& y, Mode mode= stdMode, Len len= stdLen );
Interval power( const Interval& x, const Interval& y, Len len= stdLen );
```

qui calcule x^y en tenant compte du mode d'arrondi choisi. (Attraper d'abord les exceptions.)