Multiprécision sur HP Prime - Partie 1

La HP Prime est très puissante mais ne sait pas calculer sur des nombres flottants avec plus de 15 chiffres de précision. Une boîte à outils permet de dépasser cette limite – par gege

Aujourd'hui nous allons créer les fonctions arithmétiques de base, les fonctions transcendantes feront l'objet d'un second article un peu plus loin dans cette Gazette.

Objectif

L'objectif est qu'on puisse disposer des fonctions scientifiques usuelles calculables avec une précision que l'utilisateur puisse spécifier.

On ne s'intéresse pas aux fonctions sur les nombres complexes qui s'en dérivent facilement, ni aux fonctions supérieures (Gamma, Zéta, Lambert) dont la nature non-linéaire et l'absence de formules de réduction rend le calcul précis difficile dans le cas général.

On ne propose pas non plus de fonctions de résolution d'équation ou de calcul matriciel.

Représentation des nombres

Pour stocker les chiffres il faut une forme de tableau contenant des nombres « standard » de la machine. On a choisi les listes.

On peut représenter les nombres en virgule fixe ou flottante, mais pour les raisons habituelles on a opté pour une représentation (mantisse, exposant). L'exposant peut être représenté par un seul nombre, il n'a pas besoin d'être lui-même en multi-précision. Comme il sera toujours présent je le place en tant que premier élément de la liste, il est facilement accessible.

Le reste de la liste représente les digits dans une base qui sera naturellement une puissance de 10.

Afin de pouvoir faire des produits de ces 'digits' sans perte de précision, sur cette machine avec 12 chiffres significatifs on a choisi d'utiliser la base 10^6, soit 6 chiffres par digit. En effet lors de la multiplication par exemple on fait des sommes de produits et accepter 7 digits aurait provoqué des problèmes.

La valeur 10^6 est stockée dans une variable nommée **m**. Par ailleurs le nombre 6 est présent à de nombreux endroits de la librairie, sous diverses formes.

Autant l'exposant peut porter son propre signe, autant il est plus difficile de stocker le signe de la mantisse de façon cohérente. Nous avons choisi la convention que si le premier élément (qui contient l'exposant) possède une partie fractionnaire, alors le nombre représenté est négatif. Cette convention n'est pas excellente, mais j'en cherche une meilleure.

Partout où on utilise un nombre négatif dans la librairie, la partie fractionnaire employée est 0.1, mais ce n'est pas obligatoire.

Enfin, où placer conventionnellement la virgule sur la mantisse ?

En général elle est avant le premier chiffre, mais j'ai choisi de la mettre après le dernier.

Aussi les représentations suivantes sont :

```
{0,123456,789012}     →    123456 789012
{-3,123456,789012}    →    0,000000 123456 789012
{2.1,12345,12}        →    -12345 000012 000000 000000
{-1.1,1234,78,90123}  →    -1234,000078 090123
```

Zéro est spécial et on considère être zéro tout nombre dont la liste est de taille 1 ou dont le second élément est zéro. Partout où on a besoin de zéro, on utilise {0}.

Briques de base

On convient de préfixer toutes les fonctions de la lettre x pour les regrouper.

Les fonctions utilitaires et pas utiles à priori hors de la librairie, sont préfixées par xx.

Le programme **xinit** doit être appelé avant utilisation de la librairie, il initialise les variables indispensables. Pour l'instant il n'en existe qu'une, la variable **m** qui contient toujours 10^6, elle est utilisée pour le report des retenues.

```
EXPORT m;

EXPORT xinit()
BEGIN
1000000m;
END;
```

La précision d'un nombre n'est pas imposée, ne dépend que de la taille de la liste qui le représente. Pour les fonctions transcendantes calculées par la méthode CORDIC, qui utilise des tables de constantes, on conviendra qu'elles auront une précision fixe.

Pour manipuler le signe on a les fonctions suivantes :

xabs renvoie le nombre en forçant son signe à '+' :

```
EXPORT xabs(f)
BEGIN
IP(f(1))f(1);RETU RN f;
END;
```

On élimine tout simplement la partie fractionnaire du premier élément. Il serait plus efficace d'éviter le passage en paramètre de la liste et son renvoi comme résultat, mais c'est le seul moyen de pouvoir inclure **xabs** dans une expression, la Prime ne permettant pas le passage par référence.

xneg inverse le signe du nombre

```
EXPORT xneg(f)
BEGIN
LOCAL k;f(1)k;
when(0==FP(k),k+when(0<k,.1,−.1),IP(k))f(1);
RETURN f;
END;
```

Selon que le premier élément contient déjà ou pas une partie décimale, on l'enlève ou on l'ajoute (en tenant compte de son signe).

xsign renvoie +1, 0 ou -1 selon que le nombre est supérieur, égal, ou inférieur à zéro.

```
EXPORT xsign(f)
BEGIN
IF 2>SIZE(f) THEN RETURN 0;END;
IF 0==f(2) THEN RETURN 0;END;
RETURN when(0==FP(f(1)),1,−1);
END;
```

On suit la définition. Noter que combiner les deux premiers tests planterait sur une liste de taille 1, la Prime calcule la seconde clause d'un ***and*** même si le premier a renvoyé ***false*** !

xxnorm normalise un nombre en conservant sa valeur numérique mais en normalisant les éléments afin qu'ils soient compris entre 0 et **m**, et supprime les digits nuls en début et fin de mantisse.

Comme elle n'est pas normalement appelée par l'utilisateur, son nom commence par xx et elle n'est pas EXPORTée.

```
xxnorm(f)
BEGIN
LOCAL a,b,r;
SIZE(f)a;0r;
FOR a FROM a DOWNTO 2 DO r+f(a)r;
r MOD mf(a);FLOOR(r/m)r;END;
IF 0≠r THEN CONCAT({f(1)},f)f;rf(2);
ELSE 2a;WHILE 0==f(a) DO a+1a;END;
CONCAT({f(1)},f({a,SIZE(f)}))f;END;
SIZE(f)a;ab;
WHILE a>1 AND 0≥f(a) DO a-1a;END;
IF a<b THEN f({1,a})f;
f(1)+b-af(1);END;
IF 1==a THEN {0}f;END;
RETURN f;
END;
```

Conversion de format

On veut pouvoir convertir de et vers une chaîne de caractères.

xtof convertit une chaîne de caractères en nombre multi-précision. On définit en passant une fonction utilitaire

```
xxmaklst(n)
BEGIN
RETURN MAKELIST(0,X,1,n);
END;

EXPORT xtof(s)
BEGIN
LOCAL f,a,b,c,d;0c;
IF "−"==LEFT(s,1) THEN MID(s,2)s;0.1c;END;
SIZE(s)d;INSTRING(s,".")a;
IF 0==a THEN 1+da;END;
xxmaklst(IP((a+10)/6))f;
IF 1<a THEN
FOR b FROM a-6 DOWNTO -5 STEP 6 DO
EXPR(MID(s,MAX(b,1),MIN(6,5+b)))f(IP((b+16)/6
));
END;END;
IP((a-d-5)/6)b;b+when(0>b,-c,c)f(1);
s+"00000"s;
FOR a FROM a+1 TO d STEP 6 DO
CONCAT(f,{EXPR(MID(s,a,6))})f;END;
RETURN xxnorm(f);
END;
```

On remarque que dans tout le code les variables scalaires sont toujours nommées a,b,c,d tandis que les nombres multi-précision sont désignés f,g,h,w,x,y,z.

xtostr réalise la conversion inverse

```
EXPORT xtostr(f)
BEGIN
LOCAL a,b,c,d,s,w;
IF −1==xsign(f) THEN
"−"s;xneg(f)f ELSE ""s;END;
SIZE(f)d;f(1)a;
IF a+d<2 THEN s+"."s;
FOR b FROM a+d TO 0 DO
s+"000000"s;END;
FOR b FROM 2 TO d DO
```

```
"00000"+STRING(f(b))w;
RIGHT(w,6)w;s+ws;END;
ELSE
s+STRING(f(2))s;
FOR b FROM 3 TO d DO
IF a+d+1==b THEN s+"."s;END;
s+RIGHT("00000"+STRING(f(b)),6)s;
END;
FOR b FROM 1 TO a DO s+"000000"s;END;
END;RETURN s;
END;
```

Fonctions arithmétiques

On aura besoin de tronquer la partie entière :

Partie fractionnaire : **xfpart**

```
EXPORT xfpart(f)
BEGIN
LOCAL a,b;SIZE(f)b;IP(f(1))+ba;
IF a≥b THEN RETURN {0};END;
IF 2>a THEN RETURN f;END;
f(1)b;f({a,SIZE(f)})f;bf(1);
RETURN f;END;
```

Partie entière : **xipart**

```
EXPORT xipart(f)
BEGIN
LOCAL a,b;SIZE(f)b;IP(f(1))+ba;
IF a≥b THEN RETURN f;END;
IF 2>a THEN RETURN {0};END;
ABS(FP(f(1)))f(1);RETURN f({1,a});
END;
```

Comparaison : **xcomp** qui n'utilise pas la soustraction, laquelle serait bien plus lente.

En effet on peut facilement comparer deux nombres de signes différents, ou qui sont de même signe mais ont un exposant différent.

```
EXPORT xcomp(f,g)
BEGIN
LOCAL a,b,c;xsign(f)a;xsign(g)b;
IF a≠b THEN RETURN sign(a-b);END;
IF 0==a THEN RETURN 0;END;
IP(f(1))+SIZE(f)b;IP(g(1))+SIZE(g)c;
IF b≠c THEN RETURN a*sign(b-c);END;
FOR b FROM 2 TO MIN(SIZE(f),SIZE(g)) DO
f(b)-g(b)c;IF 0≠c THEN RETURN a*SIGN(c);END;
END;
SIZE(f)-SIZE(g)b;
RETURN when(0==b,0,a*SIGN(b));
END;
```

On peut maintenant s'attaquer aux fonctions arithmétiques : **xadd** pour l'addition.

L'addition de deux nombres dépend de leur signe, si c'est le même c'est vraiment une addition, sinon une soustraction. On appelle donc deux sous-fonctions utilitaires qui additionnent ou soustraient en valeur absolue.

On identifie si un des termes est nul, dans ce cas le résultat est immédiat.

```
EXPORT xadd(f,g)
BEGIN
IF 0==xsign(f) THEN RETURN g;END;
IF 0==xsign(g) THEN RETURN f;END;
IF 0<xsign(f) THEN
IF 0<xsign(g) THEN RETURN xxadd(f,g);
ELSE xneg(g)g;RETURN xsub(f,g);END;
ELSE xneg(f)f;
IF 0<xsign(g) THEN RETURN xsub(f,g);
ELSE xneg(g)g;xxadd(f,g)f;
RETURN xneg(f);END;
END;END;
```

Soustraction : **xsub**

```
EXPORT xsub(f,g)
BEGIN
LOCAL a,b,c,d;xsign(f)a;
IF 0==a THEN RETURN xneg(g);END;
xsign(g)b;IF 0==b THEN RETURN f;END;
IF a≠b THEN
IF 0>a THEN xabs(f)f ELSE xabs(g)g;END;
xxadd(f,g)f;IF 0>a THEN xneg(f)f END;
ELSE
IF 0>a THEN xabs(f)f;xabs(g)g;END;
IF xcomp(f,g)>0 THEN
xxsub(f,g)f;IF 0>a THEN xneg(f)f;END;
ELSE
xxsub(g,f)f;IF 0<a THEN xneg(f)f;END;
END;
END;RETURN f;
END;
```

Voici les fonctions utilitaires qui font l'opération réelle :

xxadd commence par ajuster la taille des listes pour qu'elles soient identiques, ensuite la somme de listes suffit mais on doit restaurer l'exposant.

```
xxadd(f,g)
BEGIN
LOCAL a,b,c,d;
f(1)a;0f(1);a+SIZE(f)b;
g(1)c;0g(1);c+SIZE(g)d;
IF a>c THEN CONCAT(f,xxmaklst(a-c))f;END;
IF c>a THEN CONCAT(g,xxmaklst(c-a))g;END;
IF b>d THEN CONCAT(xxmaklst(b-d),g)g;END;
IF d>b THEN CONCAT(xxmaklst(d-b),f)f;END;
f+gf;MIN(a,c)f(1);RETURN xxnorm(f);
END;
```

xxsub est similaire, la fonction **xadd** ou **xsub** qui l'appelle s'est d'abord assurée que le premier terme est plus grand que le second. Le résultat est donc forcément positif.

```
xxsub(f,g)
BEGIN
LOCAL a,b,c,d;
f(1)a;0f(1);a+SIZE(f)b;
g(1)c;0g(1);c+SIZE(g)d;
IF a>c THEN CONCAT(f,xxmaklst(a-c))f;END;
IF c>a THEN CONCAT(g,xxmaklst(c-a))g;END;
IF b>d THEN CONCAT(xxmaklst(b-d),g)g;END;
IF d>b THEN CONCAT(xxmaklst(d-b),f)f;END;
f-gf;MIN(a,c)f(1);
0c;FOR a FROM SIZE(f) DOWNTO 2 DO
c+f(a)c;c MOD mf(a);FLOOR(c/m)c;END;
RETURN xxnorm(f);
END;
```

On passe à un peu plus compliqué, la multiplication **xmul**. On regarde d'abord le cas ou au moins un terme est nul, sinon on dimensionne un nombre multi-précision avec suffisamment de termes, et on fait le produit terme à terme. Les sommes de produits accumulés font au plus 12 chiffres, et le tout est finalement normalisé.

```
EXPORT xmul(f,g)
BEGIN
LOCAL a,b,c,d,h;xsign(f)*xsign(g)c;
IF 0==c THEN RETURN {0};END;
SIZE(f)a;SIZE(g)b;xxmaklst(a+b-2)h;
IP(f(1))+IP(g(1))d;
IF 0>c THEN when(0>d,−.1,.1)+dd;END;
dh(1);
FOR c FROM a DOWNTO 2 DO
FOR d FROM b DOWNTO 2 DO
h(c+d-2)+f(c)*g(d)h(c+d-2);END;END;
RETURN xxnorm(h);
END;
```

La fonction **xmulk** permet la multiplication par un scalaire, utile car bien plus rapide quand un des termes n'est pas en multiprécision :

```
EXPORT xmulk(f,k)
BEGIN
LOCAL a;
IF 0==k THEN RETURN {0};END;
IF 0==xsign(f) THEN RETURN {0};END;
IF 0>k THEN −kk;xneg(f)f;END;
FOR a FROM SIZE(f) DOWNTO 2 DO
f(a)*kf(a);END;RETURN xxnorm(f);
END;
```

La division est **xdiv**.

Il est nécessaire de donner en troisième paramètre la taille du quotient souhaité, car une division ne tombe en général pas juste et on pourrait la poursuivre indéfiniment. Ce paramètre est le nombre de tranches souhaité, un nombre entier.

On teste le cas du diviseur ou du dividende nul, et si pas dans ce cas on dimensionne le résultat.

Ensuite on élargit la plus petite mantisse pour que les nombres aient la même largeur.

On démarre une boucle pour trouver tous les 'chiffres' (6 chiffres) du quotient multi-précision.

Pour chacun, en utilisant les premiers termes des nombres, on estime un quotient partiel qu'on essaye en soustrayant (dividende − quotient * diviseur).

Si le résultat après propagation des retenues est négatif, on corrige en diminuant le quotient estimé et on boucle.

Sinon, ce 'chiffre' est trouvé et on passe au suivant.

Quand c'est fini on normalise essentiellement pour supprimer les zéro en début de quotient.

```
EXPORT xdiv(f,g,p)
BEGIN
LOCAL a,b,c,d,h,q;
// Cas particuliers
xsign(g)b;IF 0==b THEN RETURN "undef";END;
xsign(f)a;IF 0==a THEN RETURN {0};END;
// Exposant
IP(f(1))+SIZE(f)-IP(g(1))-SIZE(g)-p+1c;
0f(1);0g(1);p+1p;xxmaklst(p)h;
IF 0>a*b THEN c+when(0>c,−.1,.1)c;END;
ch(1);SIZE(f)-SIZE(g)d;
IF 0<d THEN CONCAT(g,xxmaklst(d))g;END;
IF 0>d THEN CONCAT(f,xxmaklst(−d))f;END;
FOR a FROM 2 TO p DO
IP((f(1)*m+f(2))/g(2))b;
REPEAT f-b*gq;0c;
FOR d FROM SIZE(q) DOWNTO 1 DO
c+q(d)c;c MOD mq(d);FLOOR(c/m)c;END;
IF 0>c THEN
b-MAX(1,IP((m-q(2))/(2*g(2))))b;END;
UNTIL 0≤c;
CONCAT(q({2,SIZE(q)}),{0})f;
IF 0==b AND 2==a THEN 1a;h(1)-1h(1);
ELSE bh(a);END;
IF 0==ΣLIST(f) THEN BREAK;END;
END;RETURN xxnorm(h);
END;
```

On a aussi la division par un scalaire **xdivk**, plus simple et plus rapide.

On n'a pas le problème d'estimer les chiffres du quotient, c'est direct, mais il faut tenir compte du signe et dimensionner le résultat :

```
EXPORT xdivk(f,k,p)
BEGIN
LOCAL a,b;
IF 0==k THEN RETURN "undef";END;
IF 0>k THEN −kk;xneg(f)f;END;
IF f(2)<k THEN p+1p;END;
p+1-SIZE(f)a;
```

```
IF 0>a THEN f({1,p+1})f;END;
IF 0<a THEN CONCAT(f,xxmaklst(a))f;END;
IF 0≠a THEN xsign(f)b;IP(f(1))-aa;
IF 0>b THEN a+when(0>a,−.1,.1)f(1);
ELSE af(1);END;
END;
0b;FOR a FROM 2 TO p+1 DO b*m+f(a)b;
IP(b/k)f(a);b MOD kb;END;
RETURN xxnorm(f);
END;
```

Fonctions dérivées des fonctions arithmétiques

Ces fonctions reposent sur les seules fonctions arithmétiques qui précèdent.

Le modulo (reste par la division) est **xmod**, on applique sans se poser de question la formule X – Y * int(X/Y)

```
EXPORT xmod(f,g)
BEGIN
LOCAL h,p;
IP(f(1))+SIZE(f)-IP(g(1))+SIZE(g)+2p;
xdiv(f,g,p)h;xipart(h)h;
xmul(h,g)h;xsub(f,h)h;RETURN h;
END;
```

La racine carrée est **xsqrt**. On applique la classique méthode de Newton, qui n'est pas efficace car impliquant une division, mais la convergence est rapide en pratique.

Là encore il faut passer un argument supplémentaire spécifiant le nombre de tranches que l'on souhaite.

On filtre les valeurs négatives, puis on calcule l'exposant qui sera la moitié de celui de la valeur en entrée. Il y a un ajustement à faire si cet exposant de départ n'est pas pair.

Ensuite on traite la mantisse.

```
EXPORT xsqrt(f,p)
BEGIN
LOCAL a,b,g,h;
xsign(f)a;
IF 0>a THEN RETURN "undef";END;
IF 0==a THEN RETURN {0};END;
f(1)+SIZE(f)-2a;a-(a MOD 2)a;
f(1)-af(1);a/2a;fg;REPEAT gh;
xdiv(f,g,p)b;xxadd(g,b)g;
xdivk(g,2,p)g;xcomp(g,h)b;
UNTIL 0==b;g(1)+ag(1);RETURN g;
END;
```

Comme sur Prime toute fonction utilisée doit avoir été déclarée ou définie précédemment dans le programme, il faut mettre les fonctions dans l'ordre, celui-ci fonctionne (de haut en bas puis de gauche à droite):

EXPORT m	xtof	xxsub *	xdivk
xabs	xtostr	xsub	xmod
xneg	xfpart	xadd	xsqrt
xsign	xipart	xmul	xinit
xxnorm *	xcomp	xmulk	
xxmaklst *	xxadd *	xdiv	

* non exportée

Essayons tout ça

Inutile d'essayer de créer une bibliothèque multi-précision si on n'a pas un système de test très solide !

Voici le code qui a permis la mise au point, il vous permettra de contrôler que tout va bien.

Tous les tests sont mis en commentaire, ce qui permet de n'activer que celui ou ceux que l'on veut effectuer.

Grâce à la première ligne, on peut le placer dans un programme séparé de la librairie.

```
xinit();

TST(U) BEGIN
PRINT("*"+U);PRINT(""+STRING(EXPR(U)));
END;

STST(s) BEGIN
LOCAL f;EXPR(s)f;IF 6==TYPE(f) THEN
PRINT(""+STRING(f)+":"+xtostr(f));
ELSE PRINT(""+STRING(f));END;END;

TST1(s,f) BEGIN
s+"("+STRING(f)+")"s;
PRINT("*"+s+":"+xtostr(f));STST(s);END;

TST2(s,f,g) BEGIN
s+"("+STRING(f)+","+STRING(g)+")"s;
PRINT("*"+s);PRINT(":"+xtostr(f)+":"+xtostr(g));
STST(s);END;

TST3(s,f,g,p) BEGIN
s+"("+STRING(f)+","+STRING(g)+","+STRING(p)
+")"s;
PRINT("*"+s);PRINT(":"+xtostr(f)+":"+xtostr(g)
+":"+STRING(p));
STST(s);END;

TST2s(s,f,p) BEGIN
s+"("+STRING(f)+","+STRING(p)+")"s;
PRINT("*"+s);PRINT(":"+xtostr(f)
+":"+STRING(p));
STST(s);END;

TST3ss(s,f,k,p) BEGIN
s+"("+STRING(f)+","+STRING(k)+","+STRING(p)
+")"s;
PRINT("*"+s);PRINT(":"+xtostr(f)+":"+STRING(k)
+":"+STRING(p));
```

```
STST(s);END;

EXPORT Multiprecision()
BEGIN
xinit(4);PRINT();
//TST2("xxadd",{−3,1,414213,562373,190097},
{−2,1,414231,562373});
//TST1("xxnorm",{−3.1,1,414213,562373,0});
//TST1("xsign",{−3.1,2,6});
//TST1("xsign",{3.1,2,6});
//TST1("xsign",{3,0});
//TST1("xxnorm",
{2,0,0,794653824,215,9645781,0,0});
//TST1("xxnorm",
{−1,20610,255085212,315204834});
//TST("xtof(".123")");
//TST("xtof("−224455445.42424544")");
//TST("xtof("−00.002115002120242421")");
//TST("xtof("77943132849972424544")");
//TST("xtostr({−2.1,58,54,22,17})");
//TST("xtostr({0})");
//TST("xtostr({−3,12,345678,901234})");
//TST("xtostr({0.1,2552,22544,255})");
//TST1("xfpart",{−2.1,2552,22544,255});
//TST1("xfpart",{−1.1,2552,22544,255});
//TST1("xfpart",{−5.1,2552,22544});
//TST1("xipart",{−2.1,2552,22544,255});
//TST1("xipart",{−1.1,2552,22544,255});
//TST1("xipart",{−2.1,257752,822544});
//TST2("xcomp",{−2.1,257752,822544},{2.1,3});
//TST2("xcomp",{−1.1,257752},{−1,257752});
//TST2("xcomp",{−2.1,257752,822544},
{−1.1,257752});
//TST2("xcomp",{−1,257752},
{−2,257752,822544});
//TST2("xcomp",{−2.1,257752,822544},
{−2.1,257752,822544});
//TST2("xcomp",{2,257752,822544},
{3,257752});
//TST1("xfpart",{−2.1,257752,822544});
//TST1("xfpart",{−1,257752,822544});
//TST2("xadd",{0,45,556899},{−1,458,566});
//TST2("xadd",{.1,45,556899},{−1,458,566});
//TST2("xadd",{.1,45,556899},{−1.1,458,566});
//TST2("xsub",{0,45,556899},{−1,458,566});
//TST2("xsub",{.1,45,556899},{−1,458,566});
//TST2("xsub",{.1,45,556899},{−1.1,458,566});
//TST2("xmul",{.1,45,556899},{−1.1,458,566});
//TST2("xmul",{−2.1,45,55699,335},
{1,458,444566,1125});
//TST("xmulk({−2.1,45886,55699,335458,444566
,1125},1133)");
//TST3("xdiv",{0,12},{0,7},5);
//TST3("xdiv",{−1,125698,588002},
{−1,32,117041},5);
//TST3("xdiv",{−2.1,45,55699,335},
{−1,458,444566,1125},3);
//TST3ss("xdivk",{-
3,2,833333,333333,333333},2,4);
//TST3("xdiv",{0,2},
{−3,1,414213,562373,190097},4);
//TST2s("xsqrt",{0,2},4);
RETURN 0;
END;
```

Il faut quand même donner un exemple, calculons pi par la méthode de Viète

```
EXPORT Viete(n,p)
BEGIN
LOCAL a,b,d;
xinit();xtof("2")d;da;
FOR n FROM n DOWNTO 0 DO
xsqrt(xadd(b,d),p)b;
xdiv(xmulk(a,2),b,p)a;
END;
RETURN xtostr(a);
END;
```

Le premier paramètre est le nombre de boucles, le second la précision en nombre de tranches de 6 chiffres comme d'habitude. A noter qu'une des tranches contiendra la partie entière (oui c'est 3 !).

viete(50,6) renvoie
"3.14159265358979323846264338330 9"

Conclusion provisoire

Dans la prochaine partie on s'attaquera aux fonctions transcendantes (log, exp) et trigonométriques ainsi que leurs dérivées simples. Nous entrerons dans le royaume de CORDIC et recourrons à un certain niveau de magie noire...

En attendant, on constate déjà que la Prime, grâce à sa puissance brute, peut calculer efficacement les fonctions arithmétiques et rendre utilisable une implémentation plutôt basique dans le langage utilisateur normal.

À une époque où la moindre calculatrice de supermarché offre parfois 24 ou 32 chiffres de précision, l'absence d'une option de précision réglable sur les modèles haut de gamme ne se justifie plus que par des « impératifs » de pédagogie à destination des écoliers et étudiants.

Dommage que sur la Prime, avec son mode CAS séparé, HP n'aie pas ménagé un espace d'exploration mathématique à destination d'une frange de passionnés.

Nous pouvons maintenant corriger ce problème. □

Multiprécision sur HP Prime – Partie 2

La HP Prime peut calculer les fonctions arithmétiques en nombres flottants sur plus de 15 chiffres de précision depuis l'article précédent. C'est maintenant au tour des fonctions transcendantes. – par gege

Nous allons créer les fonctions transcendantes selon les mêmes principes que les fonctions arithmétiques décrites dans l'article précédent.

Objectif

L'objectif est de calculer logarithme, exponentielle et les lignes trigonométriques directes et inverses.

Pour cela on s'appuie sur les fonctions arithmétiques décrites précédemment, et sur la méthode CORDIC. Nous ne tentons pas de décrire cette méthode, beaucoup le font et parfois correctement.

Calcul avec des tables

La méthode CORDIC utilise des tables de constantes, donc autant en arithmétique on pouvait calculer à la précision que l'on veut en la faisant varier durant le calcul, autant il va falloir la choisir à l'avance pour nos nouvelles fonctions.

Pour pouvoir la choisir, il devra être possible de recalculer les tables à la demande par appel d'une fonction spéciale.

Cette fonction s'appelle **xsetp** et prend un seul paramètre, qui est le nombre de tranches souhaitées. Le nombre de tranches est conservé dans la variable globale **pre**, qu'il ne faut jamais changer hors du programme **xsetp**.

Par exemple **xsetp**(6) permet le calcul des fonctions transcendantes avec au plus 36 chiffres de précision.

Trois autres variables globales **tlog**, **log10** et **ttan** seront utilisées pour logarithme, exponentielle et tangente.

Pour rendre le développement plus agréable, les nouvelles fonctions sont logées dans un programme séparé, mais il faut rendre globales **xxnorm**, **xxadd**, **xxsub** et **xxmaklst** en les préfixant par EXPORT dans la librairie arithmétique, parce qu'on va les utiliser. On doit aussi dans cette dernière ajouter au tout début

```
xsetp();
```

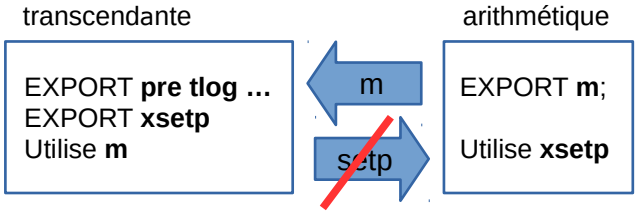
Car on veut que **xinit** appelle **xsetp** et recalcule les tables Cordic. On la modifie selon

```
EXPORT xinit(n)
BEGIN
1000000m;xsetp(n);
END;
```

La nouvelle librairie commence donc par :

```
EXPORT pre,tlog,log10,ttan;
```

En effet et si on se penche un instant sur le comportement de la Prime pour rendre disponibles des variables ou fonctions pour d'autres programmes, voici ce qui se passe :



Tant qu'un module n'a pas été validé par la Prime, ses variables exportées n'existent pas, donc si on les utilise dans un autre module celui-ci fera une « Syntax Error ».

Ici on compile l'arithmétique avant le module transcendant, il est donc nécessaire d'y déclarer **xsetp** car l'autre n'est pas encore compilé, puisqu'il dépend de **m**.

Pour travailler avec des listes de taille fixe, on a besoin d'un fonction utilitaire capable de tronquer la représentation d'un flottant en une liste de taille choisie

```
xxtrunc(f,p)
BEGIN
LOCAL a,b;
IF 0==xsign(f) THEN
RETURN xxmaklst(p+1);END;
p+1-SIZE(f)a;
IF 0==a THEN RETURN f;END;
IF 0>a THEN f({1,p+1})f;
ELSE CONCAT(f,xxmaklst(a))f;END;
FP(f(1))b;IP(f(1))-aa;
IF 0≠b THEN a+when(0>a,−.1,.1)a;END;
af(1);RETURN f;
END;
```

Logarithme

Les valeurs de la table des logs sont les logarithmes des nombres $1+10^{-p}$, on aura besoin de ces nombres 10^p

```
EXPORT xx10mp(p)
BEGIN
LOCAL a;p MOD 6a;RETURN {(p-a)/6,10^a};
END;
```

xlog calcule le logarithme naturel (base e).

On suppose disposer d'une table pré-calculée **tlog** (variable globale) dans laquelle sont stockées les valeurs des logarithmes nécessaires à l'algorithme Cordic.

```
EXPORT xlog(f)
BEGIN
LOCAL a,b,c,g,y,z,t;
IF 0≥xsign(f) THEN RETURN "log<0";END;
f(1)+SIZE(f)-1c;
IF 0≠c THEN f(1)-cf(1);6*cc;END;
1a;10*f(2)b;
WHILE m>b DO 10*bb;10*aa;c-1c;END;
IF 1≠a THEN xmulk(f,a)f;END;
{0}g;0a;
REPEAT xxadd({0,1},xx10mp(−a))y;
REPEAT xmul(f,y)z;
xxnorm(xxtrunc(z,pre+1))z;
xcomp(z,{0,1})b;
IF 0<b THEN BREAK;END;
IF 3*pre<a THEN xx10mp(−a)t;ELSE
tlog({1+a*(pre+1),(1+a)*(pre+1)})t;
END;
IF 0==xsign(t) THEN BREAK;END;
zf;xxadd(g,t)g;
IF 0==b THEN
 IF 0≠c THEN xmulk(log10,c)z;
 xxsub(z,g)g;ELSE xneg(g)g;END;
 RETURN g;
END;
UNTIL 1==0;
a+1a
UNTIL 6*pre≤a;
 IF 0≠c THEN xmulk(log10,c)z;
 xxsub(z,g)g;ELSE xneg(g)g;END;
 RETURN g;
END;
```

En gros on teste les cas zéro ou négatif, puis on prend l'exposant E, qui contribuera pour 6*E*log(10) car l'exposant est en fait le nombre de tranches, chacune ayant 6 chiffres. La valeur de log(10) est dans la variable **log10**.

Puis on multiplie par 10 si nécessaire un nombre suffisant de fois pour que le premier 'chiffre' comporte six chiffres (exemple : {123, 456789} est transformé en {123456, 789000}).

Ensuite c'est l'algorithme Cordic classique, on multiplie l'argument de départ par des coefficients de la forme $(1+10^{-i})$ tout en accumulant par ailleurs les logarithmes de ces termes, jusqu'à ce que l'argument soit très proche de 1.

On remarque que l'absence du GOTO dans le langage de la Prime rend l'écriture, la mise au point et la lecture de ce code particulièrement difficile. Est-il possible que les adversaires du GOTO ne programment pas d'algorithmes sophistiqués ?

xexp est l'exponentielle, c'est l'inverse du log : au lieu de multiplier l'argument par des coefficients en accumulant leurs logarithmes, on soustrait les logarithmes des coefficients, en multipliant les coefficients entre eux.

```
EXPORT xexp(f)
BEGIN
LOCAL a,b,c,g,h,y;
{0,1}g;
IF 0==xsign(f) THEN RETURN g;END;
xipart(xdiv(f,log10,1))h;xsign(h)a;
IF 0==a THEN 0c;ELSE
a*h(2)*m^IP(h(1))c;
IF 0>c THEN c-1c;END;
xmulk(log10,c)h;xsub(f,h)f;
END;
1a;
REPEAT
IF a>3*pre THEN xx10mp(−a)y;ELSE
tlog({1+a*(pre+1),(1+a)*(pre+1)})y;
END;
REPEAT
xcomp(f,y)b;IF b<0 THEN BREAK;END;
xsub(f,y)f;xxadd({0,1},xx10mp(-a))h;
xmul(g,h)g;xxtrunc(g,pre+1)g;
IF b==0 THEN
IF 0≠c THEN xmul(g,xx10mp(c))g;END;
RETURN g;END;
UNTIL 1==0;
a+1a;
UNTIL 6*pre≤a;
IF 0≠c THEN xmul(g,xx10mp(c))g;END;
RETURN g;
END;
```

Trigonométrie

xpi renvoie la constante π. Ici cette valeur est stockée dans le programme, et on se contente de la tronquer à la précision qui a été choisie lors de l'initialisation des fonctions transcendantes

```
EXPORT xpi()
BEGIN
LOCAL p,f;
{0,3,141592,653589,793238,
462643,383279,502884}f;
MIN(7,pre)p;1-pf(1);
```

```
RETURN f({1,p+1});
END;
```

On pourrait faire beaucoup mieux en le recalculant comme 4.arctan(1) dans **xsetp**, et en stockant dans une variable globale.

xatan calcule l'arc tangente, selon le même principe que le logarithme.

```
EXPORT xatan(f)
BEGIN
LOCAL a,b,c,g,h,v,w,y;
xsign(f)c;IF 0==c THEN RETURN {0};END;
IF 0>c THEN xabs(f)f;END;
IF xcomp({0,1},f)<0 THEN
xdiv({0,1},f,pre)f;2-cc;END;
{0,1}g;0a;{0}y;
REPEAT IF a>2*pre THEN xx10mp(−a)w;
ELSE
ttan({1+a*(pre+1),
(1+a)*(pre+1)})w;ttan({1+a*(pre+1),
(1+a)*(pre+1)})w;
END;
IF 0≠w(1) THEN
REPEAT
xx10mp(−a)h;xmul(g,h)h;xcomp(f,h)b;
IF 0>b THEN BREAK;END;
xxsub(f,h)h;xxadd(y,w)y;
IF 0==b THEN 2*prea;BREAK;END;
xx10mp(−a)v;xmul(f,v)v;xxadd(v,g)g;
xxtrunc(g,pre+1)g;xxtrunc(h,pre+1)f
UNTIL 1==0;
END;
a+1a;
UNTIL 2*pre≤a;
IF 0≠b THEN
xdiv(f,g,pre)h;xadd(y,h)y;END;
IF 2==ABS(c) THEN
xdiv(xpi,{0,2},pre)h;
xxsub(h,f)f;END;
IF 0>c THEN xneg(y)y;END;
RETURN y;
END;
```

xtan est la tangente. On remarque qu'on n'a pas traité le problème de normaliser l'argument entre]-π/2, π/2[, à utiliser avec prudence donc, il faudrait ajouter le mécanisme manquant.

```
EXPORT xtan(f)
BEGIN
LOCAL e,g,u,w;
0e;{0}g;
REPEAT
IF e>2*pre THEN xx10mp(−e)w;ELSE
ttan({1+e*(pre+1),(1+e)*(pre+1)})w;
END;
REPEAT
IF xcomp(f,w)<0 THEN BREAK;END;
xxsub(f,w)f;
IF 0==xsign(f) THEN RETURN g;END;
xxadd(g,xx10mp(−e))u;
xmul(g,xx10mp(−e))g;xxsub({0,1},g)g;
xdiv(u,g,pre)g;
UNTIL 1==0;
e+1e;
UNTIL 2*pre≤e;
xxadd(g,f)u;xmul(g,f)g;
xxsub({0,1},g)g;xdiv(u,g,pre)g;
RETURN g;
END;
```

xsin est le sinus, on utilise la formule immédiate :

$$\sin(x)=\frac{\tan(x)}{\sqrt{1+\tan(x)^2}}$$

```
EXPORT xsin(f)
BEGIN
LOCAL g;
xtan(f)g;xmul(g,g)f;
xxadd(f,{0,1})f;xsqrt(e,pre)f;
xdiv(g,f,pre)g;RETURN g;
END;
```

Pour les autres lignes trigonométriques on peut utiliser les formules suivantes :

$$\cos(x)=\sin\left(\frac{\pi}{2}-x\right)$$

$$\arcsin(x)=\arctan\left(\frac{x}{\sqrt{(1-x^2)}}\right)$$

$$\arccos(x)=\frac{\pi}{2}-\arcsin(x)$$

Calcul des tables CORDIC

xsetp recalcule les tables de constantes pour les fonctions transcendantes, elle est appelée uniquement par **xinit**, mais on peut la rappeler à tout moment pour changer la précision.

Le seul paramètre est le nombre de tranches pour les constantes, ce qui conditionne donc la précision de ces fonctions.

```
EXPORT xsetp(p)
BEGIN
LOCAL a,b,f,g;
ppre;
{}tlog;
FOR a FROM 3*p DOWNTO 6*p/5 STEP 1 DO
xx10mp(−a)f;
FOR b FROM 2 TO 6*p/a DO
xx10mp(−b*a)g;xdivk(g,b,p)g;
IF 1==b MOD 2 THEN xxadd(f,g)f
ELSE xxsub(f,g)f END;
```

```
END;
xxtrunc(f,p)f;CONCAT(f,tlog)tlog;
END;
CONCAT(xxmaklst((a+1)*(p+1)),tlog)tlog;
FOR a FROM a DOWNTO 0 STEP 1 DO
xxadd({0,1},xx10mp(−a))f;
xdiv({0,1},f,p)f;xneg(xlog(f))f;
xxtrunc(f,p)f;IF 0<a THEN
CONCAT(tlog({1,a*(p+1)}),f)f;END;
CONCAT(f,tlog({1+(a+1)*(p+1),
SIZE(tlog)}))tlog;END;
xneg(xlog({−1,100000}))log10;
RETURN pre;
{}ttan;
FOR a FROM 2*p DOWNTO 6*p/5 STEP 1 DO
xx10mp(−a)f;
FOR b FROM 3 TO 6*p/a STEP 2 DO
xx10mp(−b*a)g;xdivk(g,b,p)g;
IF 1==b MOD 4 THEN xxadd(f,g)f
ELSE xxsub(f,g)f END;
END;
xxtrunc(f,p)f;CONCAT(f,ttan)ttan;
END;
```

Les tables **tlog** et **ttan** sont des listes contenant successivement les listes représentant les constantes CORDIC.

On utilise la syntaxe de la Prime L({a,b}) qui renvoie la sous-liste de L constituée des éléments du a-ième au b-ième.

tlog contient 3x**pre** valeurs, et **ttan** en contient 2x**pre**.

Leur mode de construction est identique :

- utiliser le développement limité en zéro pour les petites valeurs ($1+10^{-p}$ avec p 'grand')

- appeler la fonction elle-même pour les autres valeurs, mais dans **xlog** ou **xatan** on a ménagé un test qui détecte que la table n'est pas encore complète (c'est IF 0==xsign(t) THEN BREAK;END; ou IF 0≠w(1)) et dans ce cas la fonction utilise les constantes suivantes de la table pour calculer.

Les temps d'initialisation sont : (3)=12s, (6)=58s, (10)=183s.

Applications

Le nombre $e^{\pi*\sqrt{163}}$ est très proche d'un entier. Malheureusement, il est surtout proche de 2.10^{17}, incalculable sur la plupart des machines… jusqu'à maintenant !

xsetp(10);**xtostr**(**xexp**(**xmul**(**xpi**,**xsqrt**(**xtof**("163"),10))))

Renvoie :

262537412640768743.99999999998199922023213745378602533066532 5

(les décimales en rouge sont inexactes)

Avec xinit(10), on observe dans log10 la valeur : 2.30258509299404568401799145468436420760110148862813624242733691045133332099998 soit 42 décimales exactes sur 78

Calculons $\left(1+\tan\left(\frac{\pi}{8}\right)\right)^2-2=0$

xadd(**xtof**("1"),**xtan**(**xdiv**(**xpi**,**xtof**("8"),10)))
f

xtostr(**xsub**(**xmul**(f,f),**xtof**("2")))

Donne

0,00000000000000000000000000000000010918320760640497890257950391440050990685430076480566347003485018106649235414321531876

Amusant, non ?

Conclusions

Mettre au point ce genre de librairie est assez long du fait des très nombreux cas particuliers qui se présentent progressivement.

Il faut souligner l'intérêt du debugger de la Prime, qui est plutôt pratique même si on aurait préféré pouvoir le piloter par des touches du clavier et non via l'écran.

Le périple n'en est pas à son terme, loin s'en faut.

- Il manque encore des fonctions (arcsin etc), et la normalisation des arguments trigonométriques reste à faire.
- L'implémentation de π est laide, à revoir.
- On pourrait aussi ajouter les nombres complexes mais ils sont en fait peu utilisés et ce serait un simple formulaire.
- Du côté des fonctions plus complexes (Béta, Lambert etc), il n'y a pas d'algorithmes pour lesquels avoir accès à la représentation interne des nombres serait un avantage.
- Par contre, simplifier l'écriture des calculs présente un grand intérêt, que diriez-vous de taper : xexpr("exp(pi*sqrt(163))") et d'obtenir directement le résultat ? Cela fera l'objet d'un prochain article !

Nous n'avons pas discuté de la **précision** des résultats, et il subsiste certains **problèmes**. Ce sujet est profond et subtil, on en reparlera aussi dans le prochain article.

Reste que la Prime est maintenant capable de calculer les fonctions transcendantes avec la précision que vous voulez, et cela relativement rapidement.

Il n'y a plus qu'à en faire bon usage. ■