Table des matières

1 Introduction

Cette note présente les éléments d'une spécification fonctionnelle du chaos polynomial. Elle se limite à description fonctionnelle des méthodes dites non intrusives où les développements en chaos polynomial est obtenu après avoir réalisé un certain nombre de simulations du code numérique.

2 Rappel mathématiques

Les polynômes de chaos ...

3 Méthodologie

3.1 Choix de la base

variables stochastiques ... ξ

3.2 Paramétrisation des paramètres incertains

La paramétrisation des paramètres incertains x consiste à représenter la variable aléatoire par une fonction
variables aléatoires $x = x(\xi)$

4 Premier exemple

Ce premier exemple est très simple. On suppose un modèle mathématique défini par une fonction analytique codée numériquement en C. Dans cet exemple, le modèle mathématique est le suivant :

$$Y = f(X_1, X_2) = X_1 X_2$$

où X_1, X_2 sont des variables aléatoires de densités respectives normale $\mathcal{N}(\mu = 1., \sigma = 0.5)$ et uniforme $\mathcal{U}(\min = 1., \max = 2.5)$. Nous allons approcher le modèle par un développement en chaos polynomial (hermite $\times$ legendre) de degré 2, valeur suffisante pour un développement exact. Le code C de la fonction analytique qui sera appelée, est :

```
=====================================
void Exemple1(double x[], double y[]) {
  y[0]=x[0]*x[1];
}
=====================================
```

4.1 Représentation des paramètres incertains

La première étape consiste à définir un ensemble permettant de regrouper les paramètres incertains du problème. Chaque paramètre incertain sera modélisé par une variable aléatoire.

```
================================================================
1 : SetRandomVariable * gu = new SetRandomVariable();
2 : gu->AddRandomVariable(new RandomVariable("Normale",1.,0.5));
3 : gu->AddRandomVariable(new RandomVariable("Uniforme",1.,2.5));
================================================================
```

1. On crée l'objet `gu` instance de la classe `SetRandomVariable` qui nous servira à regrouper l'ensemble des variables aléatoires modélisant les paramètres incertains du problème,
2. on y ajoute la première variable aléatoire X_1 de loi normale, de moyenne 1 et d'écart-type 0.5,
3. puis la seconde variable aléatoire X_2 de loi uniforme, répartie dans $[1., 2.5]$.

4.2 Base du chaos polynomial

Nous allons choisir naturellement pour cet exemple, un polynôme de chaos obtenu par tensorisation d'un polynôme de Hermite associé à la variable normale X_1, et d'un polynôme de Legendre associé à la variable uniforme X_2. Pour cela nous allons donc définir une base stochastique constituée d'un variable normale standardisée $\mathcal{N}(0, 1)$ et d'une variable uniforme standardisée $\mathcal{U}(0, 1)$. Cette base stochastique sera par la suite utilisée pour construire le polynôme de chaos.

```
=========================================================
1 : SetRandomVariable * gx = new SetRandomVariable();
2 : gx->AddRandomVariable(new RandomVariable("Normale"));
3 : gx->AddRandomVariable(new RandomVariable("Uniforme"));
4 : PolynomialChaos * pc = new PolynomialChaos(gx);
=========================================================
```

1. On crée l'objet `gx` instance de la classe `SetRandomVariable` pour regrouper l'ensemble des variables stochastiques,
2. la première est une variable normale standardisée $\mathcal{N}(0, 1)$,
3. et la seconde, une variable uniforme standardisée $\mathcal{U}(0, 1)$;

4. puis à partir de `gx` on crée le polynôme de chaos. On a donc ici crée un polynôme de chaos issu de la tensorisation d'un polynôme de **Hermite** (la première variable stochastique a une densité normale) et d'un polynôme de **Legendre** (la seconde variable stochastique a une densité uniforme).

4.3 Plan d'expériences numériques

Nous allons spécifier un plan d'expériences numérique à partir d'une quadrature de Gauss. Pour notre exemple (polynôme de degré 2), il suffit donc de spécifier un degré 2 pour la quadrature. La formule d'intégration nous permettra donc de calculer exactement (sans erreur d'approximation) les coefficents du polynôme de chaos.

```
=========================================
1 : int degre = 2;
2 : gx->BuildSample("Quadrature",degre);
3 : gu->BuildSample(gx);
=========================================
```

1. On spécifie le degré,
2. on calcule la quadrature tensorisée pour un degré 2,
3. puis on calcule le plan d'expériences numériques à réaliser. Le plan est obtenu par transformations automatiques isoprobabiliste entre les variables stochastiques et les paramètres incertains.

4.4 Réalisation du plan d'expériences numériques

Cette partie du code illustre les appels au modèle mathématique. La fonction analytique est appelée pour chaque entrée du plan d'expériences numériques. Les sorties calculées sont récupérées par le polynôme.

```
=====================================================
1 : int np=gu->GetSize();
2 : pc->SetSizeTarget(np);
3 : int nx=pc->GetDimensionInput();
4 : int ny=pc->GetDimensionOutput();
5 : double input[nx], output[ny];
6 : for(int k=1;k<=np;k++) {
    a :  for(int i=1;i<=nx;i++) input[i-1]=gu->GetSample(k,i);
    b :  Exemple1(input,ouput);
    c :  for(int j=1;j<=ny;j++) pc->SetTarget(k,j,output[j-1]);
7 : }
=====================================================
```

1. on récupère la taille np du plan d'expériences à réaliser,
2. on alloue la mémoire nécessaire pour récupérer les sorties qui seront calculées par le modèle mathématique,
3. puis on récupère le nombre des entrées (nx=2)
4. et le nombre des sorties (ny=1),
5. on déclare de variables temporaires pour les entrées et de sorties du modèle mathématique ;
6. début de la boucle des évaluations numériques
   (a) saisie des entrées (input),
   (b) appel à la fonction analytique (modèle mathématique),
   (c) les sorties calculées (output) sont transférées au polynôme.
7. fin de la boucle

4.5 Calcul des coefficients du polynôme

Avant de calculer les coefficients du polynôme, nous devons évidemment définir son degré. Dans notre exemple, le degré a été précédemment défini lors de la quadrature (degre=2) :

```
================================================
1 : pc->SetDegree(degre);
2 : pc->ComputeChaosExpansion(gx,"Integration");
================================================
```

1. On spécifie le degré du polynôme,
2. puis on calcule les coefficients par intégration numérique.

A partir de cette étape, le polynôme de chaos est complétement instancié, il peut être utilisé comme un méta-modèle avec la particularité d'être adapté à l'ANOVA fonctionnelle.

4.6 Edition de l'analyse de sensibilité

Cette partie du programme illustre la manière de récupérer un certain nombre d'informations : moyenne, variance et indices de sensibilité :

```
================================================================
cout << "Mean     = " << pc->GetMean() << endl;
cout << "Variance = " << pc->GetVariance() << endl;
cout << "Indice de sensibilite du 1er ordre"<<endl;
cout << "    Variable X1 = " << pc->GetIndiceFirstOrder(1)<<endl;
cout << "    Variable X2 = " << pc->GetIndiceFirstOrder(2)<<endl;
cout << "Indice de sensibilite Totale"<<endl;
cout << "    Variable X1 = " << pc->GetIndiceTotalOrder(1)<<endl;
cout << "    Variable X2 = " << pc->GetIndiceTotalOrder(2)<<endl;
================================================================
```

Sur la sortie standard, on obtient les résultats suivants :

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Mean     = 1.75
Variance = 1
Indice de sensibilite du 1er ordre
    Variable X1 = 0.765625
    Variable X2 = 0.1875
Indice de sensibilite Totale
    Variable X1 = 0.8125
    Variable X2 = 0.234375
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

4.7 Programme complet

```
void CaseExemple1(int degre) {
  // Definition des variables physiques incertaines
  SetRandomVariable * gu = new SetRandomVariable();
  gu->AddRandomVariable(new RandomVariable("Normale",1.,0.5));
  gu->AddRandomVariable(new RandomVariable("Uniforme",1.,2.5));

  // Definition des variables stochastiques et du polynôme de chaos
  SetRandomVariable * gx = new SetRandomVariable();
```

```
  gx->AddRandomVariable(new RandomVariable("Normale"));
  gx->AddRandomVariable(new RandomVariable("Uniforme"));
  PolynomialChaos * pc = new PolynomialChaos(gx);

  // Calcul du plan d'experiences numeriques par Quadrature
  gx->BuildSample("Quadrature",degre);
  gu->BuildSample(gx);

  // Realisation du plan d'experiences numeriques
  int nx = pc->GetDimensionInput();  // nx=2
  int ny = pc->GetDimensionOutput(); // ny=1 (valeur par defaut)
  int np=gu->GetSize();              // taille du plan d'experiences
  pc->SetSizeTarget(np);             // allocation memoire
  double input[nx], output[ny];
  for(int k=1;k<=np;k++) {
    for(int i=0;i<nx;i++) input[i]=gu->GetSample(k,i+1);  // on recupere les entrees
    Exemple1(input,output);                               // appel au modele numerique
    for(int j=0;j<ny;j++) pc->SetTarget(k,j+1,output[j]); // transfert des sorties au pc
  }

  // Calcul du chaos polynomial
  pc->SetDegree(degre);
  pc->ComputeChaosExpansion(gx,"Integration");

  // Edition des resultats
  cout << "Mean     = " << pc->GetMean() << endl;
  cout << "Variance = " << pc->GetVariance() << endl;
  cout << "Indice de sensibilite du 1er ordre"<<endl;
  cout << "    Variable X1 = " << pc->GetIndiceFirstOrder(1)<<endl;
  cout << "    Variable X2 = " << pc->GetIndiceFirstOrder(2)<<endl;
  cout << "Indice de sensibilite Totale"<<endl;
  cout << "    Variable X1 = " << pc->GetIndiceTotalOrder(1)<<endl;
  cout << "    Variable X2 = " << pc->GetIndiceTotalOrder(2)<<endl;
}
```

5 Cas test Ishigami

Le cas test de Ishigami est défini par le modèle suivant :

$$Y = f(X_1, X_2, X_3) = \sin(X1) + 7\sin(X_2)^2 + 0.1X_3^4\sin(X_1)$$

où X_1, X_2, X_3 sont des variables aléatoires de densité uniforme $\mathcal{U}(-\pi, +\pi)$. Nous allons approcher le modèle par un développement en chaos polynomial issu de la tensorisation de 3 polynômes de Legendre. Pour illustrer les cas d'utilisation à plusieurs sorties, nous avons ajouté à ce cas test, une seconde sortie égale au double de la première. Le code C de la fonction analytique qui sera appelée, est :

```
============================================
void Ishigami(double x[], double y[]) {
  double u1,u2,u3,s1,s2,s3,a,b;
  a=7.; b=0.1; u1=x[0];    u2=x[1];    u3=x[2];
  s1=sin(u1); s2=sin(u2); s3=sin(u3);
  y[0] = s1 + a *s2*s2 + b * u3*u3*u3*u3 * s1;
  y[1]=2.*y[0];
}
============================================
```

5.1 Représentation des paramètres incertains

La première étape consiste à définir un ensemble permettant de regrouper les paramètres incertains du problème. Chaque paramètre incertain sera modélisé par une variable aléatoire.

```
=========================================================================
1 : SetRandomVariable * gu = new SetRandomVariable();
2 : gu->AddRandomVariable(new RandomVariable("Uniforme",-M_PI,+M_PI));
2 : gu->AddRandomVariable(new RandomVariable("Uniforme",-M_PI,+M_PI));
2 : gu->AddRandomVariable(new RandomVariable("Uniforme",-M_PI,+M_PI));
=========================================================================
```

ou de façon plus concise :

```
=============================================================================================
1 : SetRandomVariable * gu = new SetRandomVariable();
2 : for(int i=0;i<3;i++) gu->AddRandomVariable(new RandomVariable("Uniforme",-M_PI,+M_PI));
=============================================================================================
```

1. On crée l'objet `gu` instance de la classe `SetRandomVariable` qui nous servira à regrouper l'ensemble des variables aléatoires modélisant les paramètres incertains du problème,
2. et on y ajoute les variables aléatoires $X_{i=1,2,3}$ de loi uniforme dans $[-\pi, +\pi]$.

La méthode `GetLog()` permet d'éditer les informations sur les variables du groupe. Par :

```
============
gu->GetLog();
============
```

on obtient sur la sortie standard :

```
>>>>>>>>>>>>>>>>>>>>>>>>
Uniforme:-3.14159:3.14159
Uniforme:-3.14159:3.14159
Uniforme:-3.14159:3.14159
<<<<<<<<<<<<<<<<<<<<<<<<
```

5.2 Base du chaos polynomial

Nous allons choisir naturellement pour cet exemple, un polynôme de chaos obtenu par tensorisation de 3 polynômes de Legendre. Nous allons donc créer un groupe de 3 variables stochastiques uniformes. La dimension stochastique (3) peut être récupérée par la méthode `GetDimension()` appliqué au groupe des paramètres incertains.

```
=======================================================
1 : int nx=gu->GetDimension();
2 : SetRandomVariable * gx = new SetRandomVariable(nx);
3 : gx->BuildSample(methode, degre);
4 : gu->BuildSample(gx);
=======================================================
```

1. On récupère le nombre de paramètres incertains, ici `nx`=3
2. puis on définit un groupe de `nx` variables stochastiques uniformes. Par défaut le constructeur crée des variables uniformes lorsqu'on passe en argument un entier.
3. On calcule le plan d'expériences pour les variables stochastiques en précisant la méthode et le degré. La méthode pourra être choisie parmi les méthodes d'intégration ("Quadrature","Petras"), aléatoires ("MontCarlo", "Lhs") ou Quasi Monte Carlo ("QmcSobol"). Pour les 3 dernières méthodes, on devra définir le nombre de tirages et non pas le degré. Pour la méthode "Petras", basée surldes points de Clenshaw-Curtis le degré défini correspond à une formule exacte que dans le cas où le polynôme de chaos est issue d'une tensorisation de polynômes de Legendre.
4. Puis on calcule le plan d'expériences numériques à réaliser par transformations iso-probabilistes.

Pour récupérer la taille du plan d'expériences, on peut appliquer la méthode `GetSize()` sur le groupe `gu` ou sur le groupe `gx` puisqu'ils sont équivalents (même nombre de variables aléatoires et même nombre de réalisations). Pour éditer sur la sortie standard la taille du plan :

```
============================================================
cout << "Nombre de simulations = " << gx->GetSize() << endl;
============================================================
```

5.3 Sauvegardes des plans d'expériences

On peut également sauvegarder les 2 plans d'expériences :

```
================================
gu->Save("PlanExperiences.dat");
gx->Save("PlanStochastique.dat");
================================
```

Le fichier `PlanExperiences.dat` contenant le plan d'expériences numériques se présente avec une entête suivie des réalisations des paramètres incertains. La $3^{ième}$ colonne correspond aux poids de la quadrature :

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Type         : Petras
# Simulations  : 495
# Variables    : 3
# DegreMax     : 8
# PDF          : Uniforme -3.14159 3.14159
# PDF          : Uniforme -3.14159 3.14159
# PDF          : Uniforme -3.14159 3.14159
```

```
  0.00000000e+00  0.00000000e+00  0.00000000e+00 -4.48251380e-02
  0.00000000e+00  0.00000000e+00  2.43346721e+00 -2.73626983e-02
  0.00000000e+00  0.00000000e+00 -2.43346721e+00 -2.73626983e-02
  ..........................................................
 -3.12221519e+00 -2.43346721e+00  2.43346721e+00  6.55930541e-04
 -3.12221519e+00 -2.43346721e+00 -2.43346721e+00  6.55930541e-04
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

L'entête précise :

1. le type de la méthode,
2. la taille de l'échantillon,
3. la dimension stochastique,
4. le degré maximal autorisé,
5. les lois des variables aléatoires modélisant paramètres physiques incertains.

La sauvegarde du plan stochastique est similaire. Les changements portent sur les lois (`PDF`) et les réalisations. Le fichier `PlanStochastique.dat` se présente ainsi :

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Type         : Petras
# Simulations  : 495
# Variables    : 3
# DegreMax     : 8
# PDF          : Uniforme 0 1
# PDF          : Uniforme 0 1
# PDF          : Uniforme 0 1

  5.00000000e-01  5.00000000e-01  5.00000000e-01 -4.48251380e-02
  5.00000000e-01  5.00000000e-01  8.87298335e-01 -2.73626983e-02
  5.00000000e-01  5.00000000e-01  1.12701665e-01 -2.73626983e-02
  ..........................................................
  3.08401839e-03  1.12701665e-01  8.87298335e-01  6.55930541e-04
  3.08401839e-03  1.12701665e-01  1.12701665e-01  6.55930541e-04
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

5.4 Polynôme de chaos

Le polynôme est construit à partir du groupe des variables stochastiques. On devra égalemen indiquer le nombre de sorties car il est différent de 1 (valeur par dafut).

```
=====================================================
1 : int ny=2;
2 : PolynomialChaos * pc = new PolynomialChaos(gx,ny);
=====================================================
```

1. On précise le nombre de sorties,
2. on construit le polynôme de chaos à partir du groupe stochastique `gx` et en précisant le nombre de sorties.

Dans cet exemple, le polynôme de chaos est donc construit par tensorisation de 3 polynômes de Legendre (car `gx` regroupe 3 variables uniformes) et aura deux jeux de coefficients puisqu'il y a deux sorties à analyser (`ny`=2).

5.5 Réalisation du plan d'expériences numériques

```
============================================================
1 : int np=gu->GetSize();
2 : pc->SetSizeTarget(np);
3 : int ne = pc->GetDimensionInput();
4 : int ns = pc->GetDimensionOutput();
5 : double input[ne], output[ns];
6 : for(int k=1;k<=np;k++) {
    a :  for(int i=1;i<=ne;i++) input[i-1]=gu->GetSample(k,i);
    b :  Ishigami(input,ouput);
    c :  for(int j=1;j<=ns;j++) pc->SetTarget(k,j,output[j-1]);
7 : }
============================================================
```

1. on récupère la taille np du plan d'expériences à réaliser,
2. on alloue la mémoire nécessaire pour récupérer les sorties qui seront calculées par le modèle mathématique,
3. comment récupérer la taille des entrées (égale à nx=3),
4. et celle des sorties (égale à ny=2),
5. on déclare de variables temporaires pour les entrées et de sorties du modèle mathématique ;
6. début de la boucle des évaluations numériques
   (a) saisie des entrées (input),
   (b) appel à la fonction analytique (modèle mathématique),
   (c) les sorties calculées (output) sont transférées au polynôme.
7. fin de la boucle

5.6 Calcul des coefficients du polynôme

Avant de calculer les coefficients du polynôme, nous devons évidemment définir son degré.

```
================================================
1 : pc->SetDegree(degre);
2 : pc->ComputeChaosExpansion(gx,"Integration");
================================================
```

1. On spécifie le degré du polynôme,
2. puis on calcule les coefficients par intégration numérique.

A partir de cette étape, le polynôme de chaos est complétement instancié, il peut être utilisé comme un méta-modèle avec la particularité d'être adapté à l'ANOVA fonctionnelle.

5.7 Sauvegarde du polynôme

La méthode `Save()` appliquée au polynôme permet de sauvegarder le polynôme dans un fichier pour une éventuelle utilisation ultérieure via le constructeur avec pour argument le nom du fichier.

```
====================================
pc->Save((char *) "NispIshigami.dat");
====================================
```

Le fichier `NispIshigami.dat`

```
================================================================================================
nx= 3 Legendre Legendre Legendre no= 8 p= 164 ny= 2 Coefficients[1]= 3.500000e+00 1.625418e+00
..................... Coefficients[2]= 6.999999e+00 3.250836e+00 .....................
================================================================================================
```

La sémantique est la suivante :

1. la dimension stochastique nx=3,
2. les polynômes orthogonaux,
3. le degré no=8,
4. le nombre de coefficients est 165 = p+1,
5. le nombre des sorties ny=2,
6. puis la liste des valeurs des coefficients pour les sorties j=1,2. Le premier coefficient correspond à la moyenne : 3.5 pour la première sortie, et 6.999999 pour la seconde.

5.8 Génération du code C du méta-modèle polynôme

La méthode `GenerateCode()` appliquée au polynôme permet de générer le code C du méta-modèle associé au polynôme de chaos et de le sauvegarder dans un fichier.

```
=========================================================================
pc->GenerateCode((char *) "NispIshigami.C", (char *) "NispIshigami");
=========================================================================
```

Le code sera sauvegardé dans le fichier `NispIshigami.C`. La fonction d'appel est `NispIshigami(input,output)` où `input` est l'adresse d'un tableau de double représdentant les entrées et output celle des sorties.
Le fichier `NispIshigami.C` se présente sous la forme :

```
=============================================================================
#include <math.h>
double NispIshigami_beta[2][165]={
3.5,1.62542,3.71599e-16,1.39743e-16, ..........};
int NispIshigami_indmul[165][3]={
0,0,0,1,0,0,0,1,0,0,0,1,2,0,0,1,1,0,1,.........};
void NispIshigami_legendre(double *phi,double x, int no) {
   int i;
   x=2.*x-1.;
   phi[0]=1.; phi[1]=x;
   for(i=1;i<no;i++)    phi[i+1]= ((2.*i+1.) * x * phi[i] - i * phi[i-1]) / (i+1.);
   for(i=0;i<=no;i++)   phi[i]  = phi[i] * sqrt(2.* i + 1.);
}
void NispIshigami(double *x, double *y) {
   int i,j,k,nx,ny,no,p;
   nx=3;ny=2;no=8;p=164;
   double psi[165],phi[3][9],xi[3],s;
   for(i=0;i<nx;i++) xi[i]=x[i];
   NispIshigami_legendre(phi[0],xi[0],8);
   NispIshigami_legendre(phi[1],xi[1],8);
   NispIshigami_legendre(phi[2],xi[2],8);
   for(k=0;k<=p;k++) {
      for(psi[k]=1.,i=0;i<nx;i++) psi[k]=psi[k]*phi[i][NispIshigami_indmul[k][i]];
   }
   for(j=0;j<ny;j++) {
```

```
      for(s=0.,k=0;k<=p;k++) s+=NispIshigami_beta[j][k]*psi[k];
      y[j]=s;
   }
}
================================================================================
```

5.9 Edition des moyennes et variances

```
================================================================================================
cout << "Moyenne"<< endl;
for(int j=1;j<=ny;j++) cout << "  Moyenne["  <<j << "] =" << pc->GetMean(j)     << endl;
cout << "Variance"<< endl;
for(int j=1;j<=ny;j++) cout << "  Variance[" <<j << "] =" << pc->GetVariance(j) << endl;
================================================================================================
```

Sur la sortie standard on obtient :

```
>>>>>>>>>>>>>>>>>>>>
Moyenne
  Moyenne[1] =3.5
  Moyenne[2] =7
Variance
  Variance[1]=13.8849
  Variance[2]=55.5396
<<<<<<<<<<<<<<<<<<<<
```

5.10 Matrices des covariances et corrélations

On obtient les élements de la matrice des covariances ou des corrélations entre les variables i et j par les méthodes `GetCovariance(i,j)` et `GetCorrealtion(i,j)`.

```
================================================================
cout << "MATRICE Covariances" << endl;
for(int i=1;i<=ns;i++) {
  for(int j=1;j<=ns;j++) cout << pc->GetCovariance(i,j) << " ";
  cout << endl;
}
cout << "MATRICE Correlations" << endl;
for(int i=1;i<=ns;i++) {
  for(int j=1;j<=ns;j++) cout << pc->GetCorrelation(i,j) << " ";
  cout << endl;
}
================================================================
```

5.11 Edition des indices de sensibilité du premier ordre et des indices de sensibilité totale

```
================================================================================================================
for(int j=1;j<=ny;j++) {
  cout << "Indice de sensibilite - Output[" << j << "]" <<endl;
  cout << "  Premier ordre ***"<< endl;
  for(int i=1;i<=nx;i++) cout << "  Variable["<< i << "]= " << pc->GetIndiceFirstOrder(i,j) << endl;
  cout << "  Totale **********"<< endl;
```

```
  for(int i=1;i<=nx;i++) cout << "  Variable["<< i << "]= " << pc->GetIndiceTotalOrder(i,j) << endl;
}
=============================================================================================================
```

Sur la sortie standard on obtient :

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Indice de sensibilite - Output[1]
  Premier ordre ***
  Variable[1]= 0.312994
  Variable[2]= 0.444213
  Variable[3]= 5.02499e-25
  Totale **********
  Variable[1]= 0.555787
  Variable[2]= 0.444213
  Variable[3]= 0.242794
Indice de sensibilite - Output[2]
  Premier ordre ***
  Variable[1]= 0.312994
  Variable[2]= 0.444213
  Variable[3]= 5.02499e-25
  Totale **********
  Variable[1]= 0.555787
  Variable[2]= 0.444213
  Variable[3]= 0.242794
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

5.12 Edition de la décomposition fonctionnelle de la variance

Pour chaque sortie, on peut éditer la décomposition fonctionnelle de la variance par la méthode `GetAnova()`. Pour obtenir les termes par valeurs croissantes de la part de la variance, on utilisera la méthode `GetAnovaOrdered()`. Enfin si on souhaite éditer uniquement les termes qui expliquent une fraction de la variance on utilisera la même méthode `GetAnovaOrdered()` mais en spécifiant la fraction. Par exemple pour obtenir la décompostion de l'ANOVA expliquant 95% de la variance de la sortie j il suffira de faire `GetAnovaOrdered(0.95,j)`. Pour notre exemple, on pourra faire :

```
=====================================================================
for(int j=1;j<ns;j++) {
  cout << "Output " << j << " ****"<< endl;
  cout << "ANOVA fonctionnelle"<< endl;
  pc->GetAnova(j);
  cout << "ANOVA fonctionnelle ordonnee"  << endl;
  pc->GetAnovaOrdered(j);
  cout << "ANOVA fonctionnelle ordonnee et seuillee a 95%"  << endl;
  pc->GetAnovaOrdered((double) 0.95,j);
}
=====================================================================
```

Et su la sortie standard on obtient :

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Output 1 ****
ANOVA fonctionnelle
1 0 0  : 0.312994
```

```
0 1 0  : 0.444213
1 1 0  : 2.00958e-25
0 0 1  : 5.02499e-25
1 0 1  : 0.242794
0 1 1  : 9.48666e-26
1 1 1  : 4.24684e-26
ANOVA fonctionnelle ordonnee
0 1 0  : 0.444213 : cumul = 0.444213
1 0 0  : 0.312994 : cumul = 0.757206
1 0 1  : 0.242794 : cumul = 1
0 0 1  : 5.02499e-25 : cumul = 1
1 1 0  : 2.00958e-25 : cumul = 1
0 1 1  : 9.48666e-26 : cumul = 1
1 1 1  : 4.24684e-26 : cumul = 1
ANOVA fonctionnelle ordonnee et seuillee a 95%
0 1 0  : 0.444213 : cumul = 0.444213
1 0 0  : 0.312994 : cumul = 0.757206
1 0 1  : 0.242794 : cumul = 1
Output 2 ****
ANOVA fonctionnelle
1 0 0  : 0.312994
0 1 0  : 0.444213
1 1 0  : 2.00958e-25
0 0 1  : 5.02499e-25
1 0 1  : 0.242794
0 1 1  : 9.48666e-26
1 1 1  : 4.24684e-26
ANOVA fonctionnelle ordonnee
0 1 0  : 0.444213 : cumul = 0.444213
1 0 0  : 0.312994 : cumul = 0.757206
1 0 1  : 0.242794 : cumul = 1
0 0 1  : 5.02499e-25 : cumul = 1
1 1 0  : 2.00958e-25 : cumul = 1
0 1 1  : 9.48666e-26 : cumul = 1
1 1 1  : 4.24684e-26 : cumul = 1
ANOVA fonctionnelle ordonnee et seuillee a 95%
0 1 0  : 0.444213 : cumul = 0.444213
1 0 0  : 0.312994 : cumul = 0.757206
1 0 1  : 0.242794 : cumul = 1
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

La dernière ligne nous indique que la part de variance expliquée par l'interaction des variables X_1 et X_3 est 0.242794 correspondant à la variance est expliquée par le groupe (X_1, X_3) lorsque ces variables agissent ensemble, c'est à dire en interaction.
En analysant les 3 dernières lignes on vérifie que la décomposition de la fonction de Ishigami se limite à 3 termes car le cumul est égal à 1 :

- une fonction de la seule variable X_2 expliquant 0.444213 de la variance, valeur correspondant à l'indice du premier ordre de X_2,
- une fonction de la seule variable X_1 expliquant 0.312994 de la variance, valeur correspondant à l'indice du premire ordre de X_1,
- une fonction du groupe des variables (X_1, X_3) expliquant le reste de la variance c'est à dire 0.242794.

5.13 Indices de sensibilité d'un groupe de variables

Les valeurs précédentes sur les différents indices de sensibilité sur un groupe quelconque de variables peuvent également être récupérées par deux méthodes.
- `GetGroupIndice(j)` pour récupérer la part de la variance de la sortie j expliquée par un groupe de variables,
- `GetGroupIndiceIntercation(j)` pour récupérer la part de la variance de la sortie j expliquée par un groupe de variables agissant ensemble en intercation.

Il faut au préalable spécifier le groupe des variables pour lequelles nous allons demander la part de la variance expliquée par ces variables :
- `SetGroupEmpty()` pour rendre vide (si besoin) le groupe courant de variables,
- `SetGroupAddVar(i)` pour ajouter la variable de rang $i = 1, 2, \ldots$ au groupe,

Par exemple pour analyser la part de la variance expliquée par le groupe (X_1, X_3) :

```
cout << "GROUPE de VARIABLES" << endl;
for(int j=1;j<=ns;j++) {
  pc->SetGroupEmpty();
  pc->SetGroupAddVar(1);
  pc->SetGroupAddVar(3);
  cout << "Output " << j << " ****"<<endl;
  cout << "   Indice du groupe (1,3) " << pc->GetGroupIndice(j) << endl;
  cout << "   Indice du groupe (1,3) Interaction " << pc->GetGroupIndiceInteraction(j) << endl;
}
```

Sur la sortie standard on obtient :

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
GROUPE de VARIABLES
Output 1 ****
   Indice du groupe (1,3) 0.555787
   Indice du groupe (1,3) Interaction 0.242794
Output 2 ****
   Indice du groupe (1,3) 0.555787
   Indice du groupe (1,3) Interaction 0.242794
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

On vérifie que que les valeurs obtenues sont identiques pour les 2 sorties (la seconde sortie est le double de la première). On vérifie que la valeur de l'indice pour le groupe de variables (X_1, X_3) est bien égale à la somme des valeurs précédemment obtenues par l'ANOVA fonctionnelle ordonnée (3 dernières lignes) : $0.555787 \simeq 0.312994 + 0.242794$. On retrouve également la valeur 0.242794 de l'indice de sensibilité du groupe lorsque les variables (X_1, X_3) agissent ensemble.

5.14 Quantiles et probabilité de dépassement de seuils

Les quantiles empiriques sur le méta-modèle peuvent être obtenus après avoir réalisé un échantillon. Par exemple pour obtenir le quantile à 95% de la sortie de rang 1 à partir d'un tirage de type Monte Carlo de 10000 réalisations :

```
================================================
cout << "QUANTILE" << endl;
InitSeed(0); // necessite #include ‘‘nisp_util.h’’
pc->BuildSample("MonteCarlo",10000);
cout << pc->GetQuantile((double) 0.95) << endl;
================================================
```

On peut également obtenir une estimation de la probabilité de dépassement d'un seuil par la méthode `InvQuantile(seuil)`. Et donc en faisant :

```
============================================================
cout << pc->GetInvQuantile(pc->GetQuantile((double) 0.95)) << endl;
============================================================
```

on devrait btenir une valeur très proche de 0.95.

6 Application Programming Interface

6.1 Variables aléatoires - Classe RandomVariable

6.1.1 Constructeurs

RandomVariable("Normale")	normale, moyenne $\mu = 0$, écart type $\sigma = 1$
RandomVariable("Normale",double m,double s)	normale, moyenne $\mu = m$, écart type $\sigma = s$
RandomVariable("Uniforme")	uniforme, $[0, 1]$
RandomVariable("Uniforme",double min,double max)	uniforme, $[\min, \max]$
RandomVariable("Exponentielle")	exponentielle, paramètre $\lambda = 1$
RandomVariable("Exponentielle",double lambda)	exponentielle, paramètre λ=lambda
RandomVariable("LogNormale",double m, double s)	log normale de moyenne XXXXXXX
RandomVariable("LogUniforme",double min,double max)	log uniforme, $[\min, \max]$

6.1.2 Réalisations

double GetValue()	une réalisation de la variable
double GetValue(RandomVariable *va, double x)	une réalisation à partir de la valeur `x` de la variable aléatoire `*va` obtenue par transformation probabiliste

6.1.3 Informations

void GetLog()	édition sur la sortie standard des informations sur la variable : loi, paramètres

6.2 Groupes de variables aléatoires - Classe SetRandomVariable

6.2.1 Constructeurs

SetRandomVariable()	par défaut
SetRandomVariable(int n)	un groupe de `n` variables uniformes $[0, 1]$
SetRandomVariable(char *fichier)	un groupe à partir d'une sauvegarde préalable dans un fichier

6.2.2 Ajout d'une variable aléatoire

void AddRandomVariable(RandomVariable *va)	ajout d'une variable aléatoire *va

6.2.3 Réalisation d'un échantillon

void BuildSample("MonteCarlo", int n)	échantillon de taille `n` par tirages indépendants
void BuildSample("Lhs", int n)	échantillon de taille `n` par Latin Hypercube
void BuildSample("QmcSobol", int n)	échantillon de taille `n` par suite à discrépance faible de Sobol
void BuildSample("Quadrature", int niveau)	échantillon par Quadrature tensorisée, formule exacte pour un polynôme de degré `niveau`-1
void BuildSample("Cubature", int niveau)	échantillon par Cubature formule exacte pour un polynôme de degré `niveau`
void BuildSample(SetRandomVariable *gva)	réalisation d'un échantillon à partir d'un échantillon d'un autre groupe de variables (transformations probabilistes)

6.2.4 Définition d'un échantillon réalisé par un autre composant (Open Turns, Uranie)

void SetSample(string type, int size)	allocation mémoire afin de récupérer un échantillon en spécifiant le `type` et la taille `size`
void SetSample(int k, int i, double value)	affecte la valeur `value` à l'exemple `k` de la variable `i`

6.2.5 Information sur le groupe

int GetDimension() ;	nombre de variables aléatoires du groupe (0 par défaut)
int GetSize() ;	taille de l'échantillon (0 par défaut)
void GetLog() ;	édition sur la sortie standard des informations sur les variables aléatoires du groupe

6.2.6 Récupération des réalisations de l'échantillon

double GetSample(int k, int i) ;	valeur de la variable `i` de la réalisation `k`

6.2.7 Sauvegarde du groupe

void Save("fichier") ;	sauvegarde du groupe dans un fichier y compris la sauvegarde éventuelle de l'échantillon

6.2.8 Effacement d'un échantillon et libération mémoire

void FreeMemory() ;	effacement de l"échantillon éventuel du groupe et libération mémoire

6.3 Polynômes de chaos - Classe PolynomialChaos

6.3.1 Constructeurs

PolynomialChaos(SetRandomVariable *gvx)	à partir d'un groupe de variables aléatoires standardisées, et par défaut le nombre de sorties est 1.
PolynomialChaos(SetRandomVariable *gvx, int ny)	à partir d'un groupe de variables stochastiques et en précisant le nombre de sorties `ny`
PolynomialChaos(char *fichier) ;	à partir d'une sauvegarde préalable dans un fichier

6.3.2 Dimensions

void SetDimensionOutput(int ny) ;	définition du nombre de sorties est `ny`
int GetDimensionOutput()	récupére le nombre de sorties
int GetDimensionInput()	récupère le nombre d'entrées
int GetDimensionExpansion()	récupère le nombre de coefficients (n-1)

6.3.3 Degré du polynôme

void SetDegree(int degre) ;	définition du degré
int GetDegree()	récupère le degré

6.3.4 Instances du polynôme

void SetInput(int i, double x) ;	affecte la valeur `x` à l'entrée `i`
void ComputeOutput() ;	calcul des sorties
double GetOutput(int j) ;	récupération de la sortie, par défaut `j`=1

6.3.5 Transfert des sorties du modèle numérique au polynôme

void ReadTarget(char *fichier) ;	Lecture d'un fichier des valeurs cibles que le polynôme devra approcher
void SetSizeTarget(int np) ;	définition du nombre des valeurs cibles
void SetTarget(int k, int j, double output) ;	définition de la valeur cible `output` pour la sortie `j` de l'exemple `k`
void FreeMemoryTarget() ;	libération mémoire de la zone des cibles

6.3.6 Calcul des coefficients du polynôme

void ComputeChaosExpansion(SetRandomVariable *gvx,"Integration")	Calcul des coefficients par `Intégration`, rappel du groupe des variables stochastiques pour récupérer l'échnatillon
void ComputeChaosExpansion(SetRandomVariable *gvx,"Regression")	Calcul des coefficients par `Régression`, rappel du groupe des variables stochastiques pour récupérer l'échnatillon

6.3.7 Moyennes, variances, covariances et corrélations

double GetMean(int j)	moyenne de la sortie `j`
double GetVariance(int j)	variance de la sortie `j`
double GetCovariance(int i, int i)	covariance des sorties `i,j`
double GetCorrelation(int i, int i)	coefficient de corrélation des sorties `i,j`

6.3.8 Indices de sensibilité : premier ordre et totale

double GetIndiceFirstOrder(int j)	Indice de sensibilité du premier ordre de la variable `i`
double GetIndiceTotalOrder(int j)	Indice de sensibilité toptale de la variable `i`

6.3.9 Indices de sensibilité d'un groupe de variables

void SetGroupAddVar(int i)	ajout de la variable `i` dans le groupe
void SetGroupEmpty() ;	le groupe est mis à vide
double GetGroupIndice(int j)	indice de sensibilité dû à l'ensemble des variables du groupe sur la sortie `j`. Par défaut $\texttt{j} = 1$
double GetGroupIndiceInteraction(int j)	Indice de sensibilité dû à l'interaction des variables du groupe sur la sortie `j`. Par défaut $\texttt{j} = 1$

6.3.10 Anova fonctionnelle

void GetAnova(int j)	Edition de l'ANOVA fonctionnelle pour la sortie `j`. Par défaut $\texttt{j} = 1$
void GetAnovaOrdered(double seuil, int j)	Edition ordonnée de l'ANOVA fonctionnelle pour la sortie `j`, de la plus grande valeur à la plus petite. Par défaut $\texttt{seuil} = 1.$ et $\texttt{j} = 1$

6.3.11 Sauvegarde

void Save(char *fichier)	Sauvegarde du polynôme dans un fichier

6.3.12 Génération du code source

void GenerateCode(char *fichier, char *name)	Sauvegarde du code source du polynôme dans un fichier. Le nom d'appel de la fonction C est donné par la chaîne `name`.

6.3.13 Quantiles et probabilités de dépassement de seuils

void BuildSample(string methode,int np,int order=1)	Tirages d'un échantillon par une méthode aléatoire MonteCarlo", "Lhs") de taille `np`. Pour les quantiles, order = 1 (valeur par défaut).
void Quantile(double alpha, int j=1)	Valeur du quantile d'ordre $\alpha \in [0,1]$ pour la sortie j (par défaut à 1)
void InvQuantile(double seuil, int j=1)	Estimation de la probabilité de dépassement de la valeur seuil pour la sortie j (par défaut à 1).