Quasi Monte-Carlo

Exemples en C++

Vincent Lemaire
vincent.lemaire@upmc.fr

Surcharge d'opérateurs

Permet de redéfinir les opérateurs usuels pour une nouvelle `classe`.

- opérateurs unaires : `++`, `--`
- opérateurs d'affectation : `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- opérateurs arithmétiques (binaires) : `+`, `-`, `*`, `/`, `%`
- opérateurs de comparaison : `==`, `!=`, `<`, `>`, `<=`, `>=`
- opérateurs « informatiques » : `()`, `[]`, `*`, `&`, `->`, `new[]`, `delete[]`
- opérateurs de flux : `<<`, `>>`
- opérateurs de conversion : `double`, `int`, `char`, `...`

En général, la surcharge des opérateurs unaires, d'affectation, de conversion et « informatiques », se fait par des fonctions membres et la surcharge des opérateurs binaires (arithmétiques et comparaisons) se fait par des fonctions amies.

Obligatoirement, la surchage des opérateurs de flux se fait par des fonctions amies.

Surcharge par fonctions membres

Surcharge par une fonction membre de l'opérateur • : la fonction doit se nommer **operator•** et

$$\texttt{obj}\bullet \text{ et } \bullet\texttt{obj} \iff \texttt{obj.}\textbf{operator}\bullet\texttt{(...)}$$
$$\texttt{obj1}\bullet\texttt{obj2} \iff \texttt{obj1.}\textbf{operator}\bullet\texttt{(obj2)}$$

Surcharge par fonctions membres

Surcharge par une fonction membre de l'opérateur • : la fonction doit se nommer **operator•** et

obj• et •obj $\Longleftrightarrow$ obj.**operator•**(...)
obj1•obj2 $\Longleftrightarrow$ obj1.**operator•**(obj2)

Syntaxe pour l'opérateur d'affectation [] (unaire)

- Lecture : `elt` **`operator`**`[](`**`int`**`)` **`const`** ;
- Ecriture : `elt&` **`operator`**`[](`**`int`**`)` ;

Surcharge par fonctions membres

Surcharge par une fonction membre de l'opérateur • : la fonction doit se nommer **operator•** et

obj• et •obj $\iff$ obj.**operator•**(...)
obj1•obj2 $\iff$ obj1.**operator•**(obj2)

Syntaxe pour l'opérateur d'affectation `[]` (unaire)

- Lecture : `elt` **`operator`**`[](`**`int`**`)` **`const`**;
- Ecriture : `elt&` **`operator`**`[](`**`int`**`)`;

Syntaxe pour opérateurs ++ et `--` qui peuvent être préfixe (++n) ou suffixe (n++) : on utilise 2 fonctions différentes :

- Opérateur préfixe : `obj&` **`operator++`**`();`
- Opérateur suffixe : `obj` **`operator++`**`(`**`int`**`);`

Surcharge par fonctions amies

Surcharge par une fonction globale (souvent amie de la calsse) de l'opérateur • : la fonction doit se nommer **`operator•`** et

$$\texttt{obj}\bullet \text{ et } \bullet\texttt{obj} \iff \textbf{operator}\bullet\texttt{(obj)}$$
$$\texttt{obj1}\bullet\texttt{obj2} \iff \textbf{operator}\bullet\texttt{(obj1, obj2)}$$

Surcharge par fonctions amies

Surcharge par une fonction globale (souvent amie de la calsse) de l'opérateur $\bullet$: la fonction doit se nommer **`operator•`** et

$$\texttt{obj}\bullet \text{ et } \bullet\texttt{obj} \iff \textbf{\texttt{operator}}\bullet\texttt{(obj)}$$
$$\texttt{obj1}\bullet\texttt{obj2} \iff \textbf{\texttt{operator}}\bullet\texttt{(obj1, obj2)}$$

Syntaxe pour les opérateurs de flux << et >>

- Injection :
  `std::ostream& operator<<(std::ostream &o, const Obj &x)`
- Extraction :
  `std::istream& operator>>(std::istream &i, Obj &x)`

Koksma-Hlawka

Theorem (Koksma-Hlawka)

Soit $(\xi_n)_{n\geqslant 1}$ *une suite sur* $[0,1]^d$ *et* f *à variation* $V(f)$ *finie. Alors*

$$\left|\frac{1}{n}\sum_{k=1}^{n} f(\xi_k) - \int_{[0,1]^d} f(u)\mathrm{d}u\right| \leqslant V(f) D_n^*(\xi),$$

où $D_n^*(\xi)$ *est la discrépance de la suite* $(\xi_n)_{n\geqslant 1}$ *i.e.*

$$D_n^*(\xi) = \sup_{x\in[0,1]^d} \left|\frac{1}{n}\sum_{k=1}^{n} \mathbf{1}_{[0,x]}(\xi_k) - \prod_{i=1}^{d} x^i\right|$$

Koksma-Hlawka

Theorem (Koksma-Hlawka)

Soit $(\xi_n)_{n\geqslant 1}$ une suite sur $[0,1]^d$ et f à variation $V(f)$ finie. Alors

$$\left|\frac{1}{n}\sum_{k=1}^{n} f(\xi_k) - \int_{[0,1]^d} f(u)\mathrm{d}u\right| \leqslant V(f)D_n^*(\xi),$$

où $D_n^*(\xi)$ est la discrépance de la suite $(\xi_n)_{n\geqslant 1}$ i.e.

$$D_n^*(\xi) = \sup_{x\in[0,1]^d}\left|\frac{1}{n}\sum_{k=1}^{n}\mathbf{1}_{[0,x]}(\xi_k) - \prod_{i=1}^{d} x^i\right|$$

Remark : Si $(U_n)_{n\geqslant 1}$ est une suite *i.i.d.* uniformément distribuée sur $[0,1]^d$, alors (par la LLI)

$$\limsup_n \sqrt{\frac{2n}{\log(\log n)}}D_n^*(U) = 1 \quad p.s.$$

Dimension 1 - Van der Corput

La plupart des suites à discrépance faible repose sur la manipulation des coefficients de la décomposition $p-$adique de n.

Voici la construction de Van der Corput (dimension 1) :

- Soit p un nombre premier qui sert de base à la décomposition $p-$adique
- Décomposition de n :

$$n = a_0 + a_1 p + \dots a_r p^r, \quad 0 \leqslant a_i \leqslant p-1, a_r \neq 0,$$

- Construction de ξ_n :

$$\xi_n^{(p)} = \frac{a_0}{p} + \frac{a_1}{p^2} + \cdots + \frac{a_r}{p^{r+1}} \in [0,1].$$

Dimension 1 - Van der Corput

La plupart des suites à discrépance faible repose sur la manipulation des coefficients de la décomposition $p-$adique de n.

Voici la construction de Van der Corput (dimension 1) :

- Soit p un nombre premier qui sert de base à la décomposition $p-$adique
- Décomposition de n :

$$n = a_0 + a_1 p + \dots a_r p^r, \quad 0 \leqslant a_i \leqslant p-1, a_r \neq 0,$$

- Construction de ξ_n :

$$\xi_n^{(p)} = \frac{a_0}{p} + \frac{a_1}{p^2} + \cdots + \frac{a_r}{p^{r+1}} \in [0,1].$$

Discrépance :

$$D_n^*(\xi^{(p)}) \leqslant \frac{1}{n}\left(1 + (p-1)\frac{\log(pn)}{\log(p)}\right)$$

Classe `p_adic`

But : Ecrire une classe `p_adic` qui permet de manipuler facilement la décomposition $p-$adique de n et de calculer $\xi_n^{(p)}$.

- 3 constructeurs :
  - à partir des coefficients a_k, $k = 0, \ldots, r$ (et de p...)
  - à partir d'un entier n (et de la base p)
  - à partir d'un réel $x \in [0, 1]$ (et de la base p)
- 2 opérateurs de conversions :
  - vers **`double`**
  - vers **`int`**
- surcharge des opérateurs : +, ++ (préfixe et suffixe)

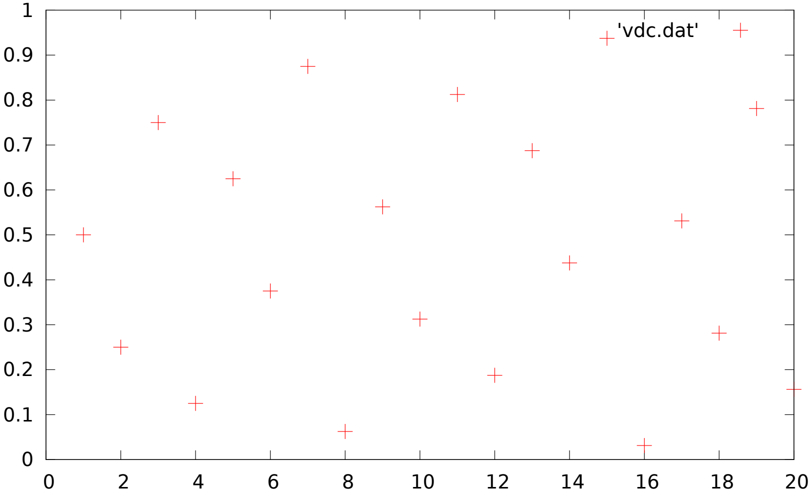
Classe `p_adic`

But : Ecrire une classe `p_adic` qui permet de manipuler facilement la décomposition $p-$adique de n et de calculer $\xi_n^{(p)}$.

- 3 constructeurs :
  - à partir des coefficients a_k, $k = 0, \ldots, r$ (et de p...)
  - à partir d'un entier n (et de la base p)
  - à partir d'un réel $x \in [0, 1]$ (et de la base p)
- 2 opérateurs de conversions :
  - vers **`double`**
  - vers **`int`**
- surcharge des opérateurs : `+`, `++` (préfixe et suffixe)

```
int main() {
    ofstream file;
    file.open("vdc.dat");
    p_adic n(0,2);
    for (int k = 1; k <= 20; ++k) {
        file << k << "\t" << (double) ++n << endl;
    }
    file.close();
}
```

20 premiers points de $\xi^{(2)}$



Implémentation

Déclaration dans le fichier `p_adic.hpp` (sans les opérateurs arithmétiques) :

```
class p_adic {
public:
    typedef std::list<int> coeff;
    p_adic(coeff ak, coeff pk) : ak(ak),pk(pk),p(*(++pk.begin())){};
    p_adic(int n, int p = 2);
    p_adic(double x, int p = 2);
    operator int() {
        return std::inner_product(ak.begin(), ak.end(), pk.begin(), 0);
    }
    operator double() {
        return std::inner_product(ak.begin(), ak.end(), ++pk.begin(),
                   0.0, std::plus<double>(), std::divides<double>());
    }
    friend struct halton;
    friend struct kakutani;
    friend struct faure;
private:
    int p;
    coeff ak, pk;
};
```

Implémentation -2-

Définition d'un constructeur dans le fichier `p_adic.cpp` :

```
p_adic::p_adic(int n, int p) : p(p) {
    int puiss = 1;
    while (n > 0) {
        ak.push_back(n % p);
        pk.push_back(puiss);
        puiss *= p;
        n -= ak.back();
        n /= p;
    }
    pk.push_back(puiss);
};
```

Exercice : écrire le constructeur qui prend un **double** en argument.

Implémentation -3- (opérateurs ++)

Définition d'une fonction `increment()` (**`private`**) qui fait le travail :

```
void p_adic::increment() {
    coeff::iterator i = ak.begin();
    while ((i != ak.end()) && ((*i)+1 == p)) { (*i) = 0; i++; }
    if (i == ak.end()) {
        ak.push_back(1);
        pk.push_back(pk.back()*p);
    }
    else (*i) += 1;
};
```

Implémentation -3- (opérateurs ++)

Définition d'une fonction `increment()` (**`private`**) qui fait le travail :

```
void p_adic::increment() {
    coeff::iterator i = ak.begin();
    while ((i != ak.end()) && ((*i)+1 == p)) { (*i) = 0; i++; }
    if (i == ak.end()) {
        ak.push_back(1);
        pk.push_back(pk.back()*p);
    }
    else (*i) += 1;
};
```

Ecriture des opérateurs dans la classe (fichier `p_adic.hpp`)

```
p_adic operator++(int) {
    p_adic copie = *this;
    increment();
    return copie;
};
p_adic& operator++() {
    increment();
    return (*this);
};
```

Halton

Soit d la dimension, et $p_1, \ldots, p_d$ les d premiers nombres premiers.
La suite d'Halton est définie par

$$\Xi_n^{(d)} = (\xi_n^{(p_1)}, \ldots, \xi_n^{(p_d)})$$

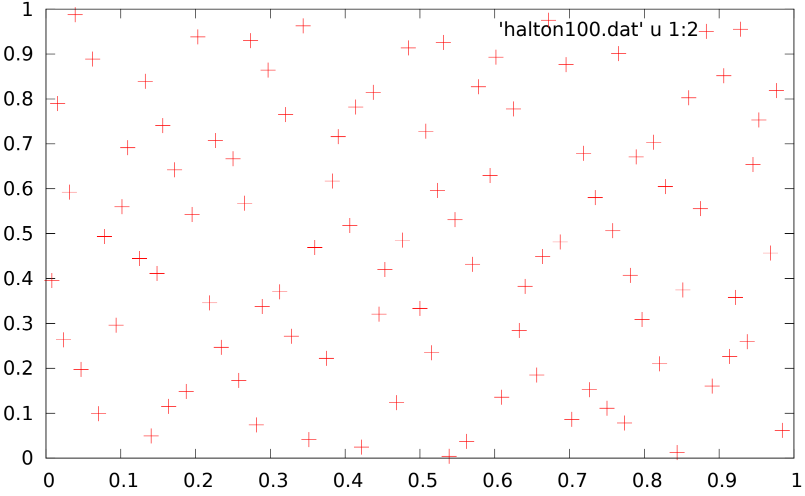
Discrépance :

$$D_n^*(\xi^{(p)}) \leqslant \frac{1}{n}\left(1 + \prod_{i=1}^{d}(p_i - 1)\frac{\log(p_i n)}{\log(p_i)}\right)$$

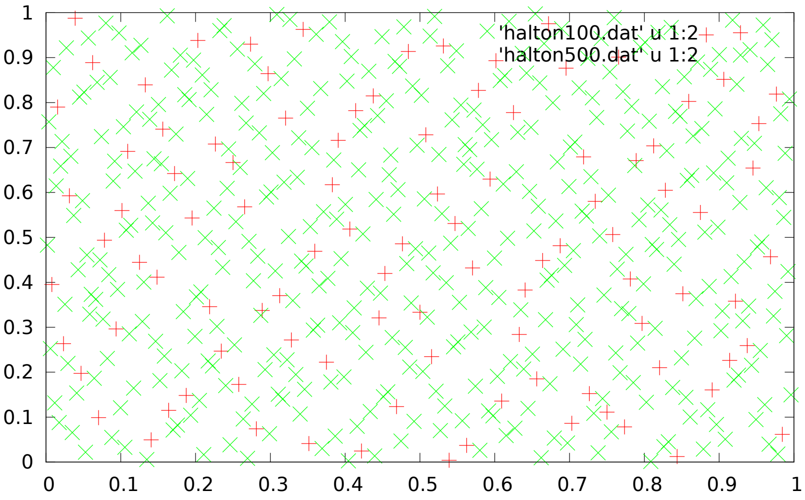
Implémentation

```
struct halton {
    typedef std::vector<double> result_type;
    typedef std::vector<p_adic> list_p_adic;
    halton(list_p_adic const & x) : nk(x), result(x.size()) {};
    halton(int dimension) : result(dimension) {
        for (int k = 0; k < dimension; k++)
            nk.push_back(p_adic((int) 1, primes[k]));
    };
    result_type operator()() {
        result_type::iterator ir = result.begin();
        list_p_adic::iterator ink = nk.begin();
        while (ink != nk.end()) {
            *ir++ = (double) (*ink++)++;
        }
        return result;
    }
private:
    list_p_adic nk;
    result_type result;
};
```

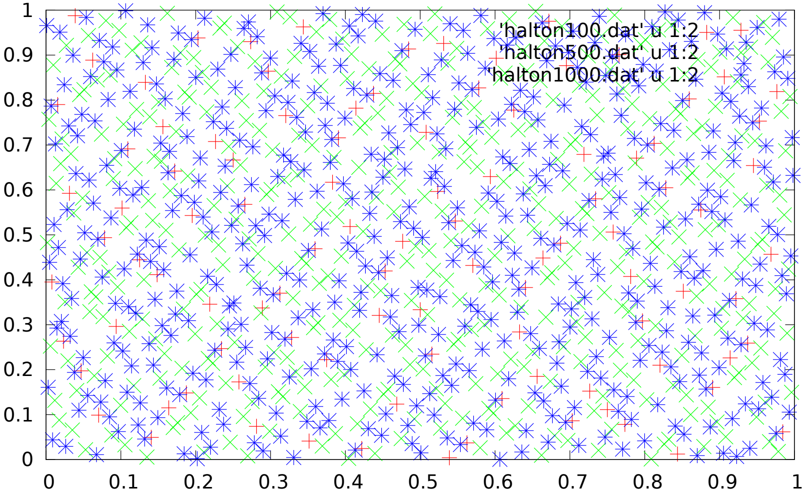
Halton, dimensions 1-2 (bases 2-3)



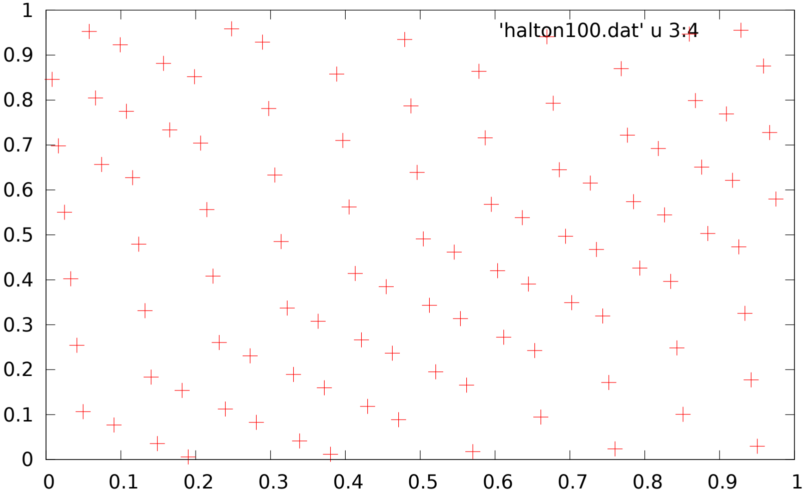
Halton, dimensions 1-2 (bases 2-3)



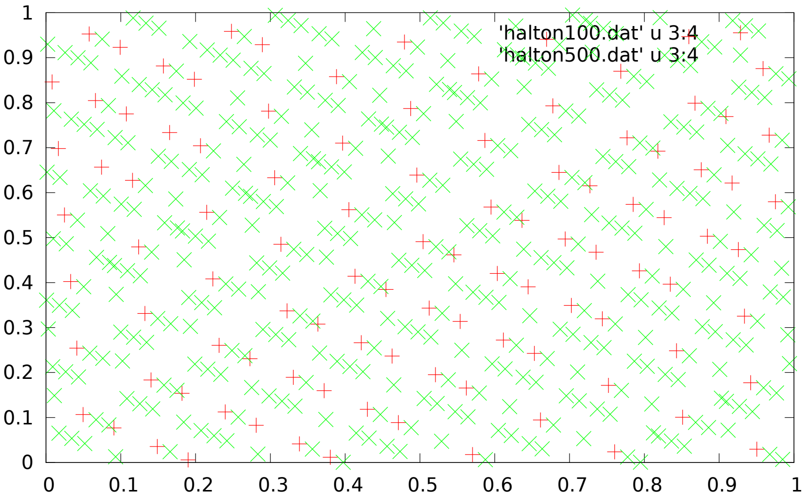
Halton, dimensions 1-2 (bases 2-3)



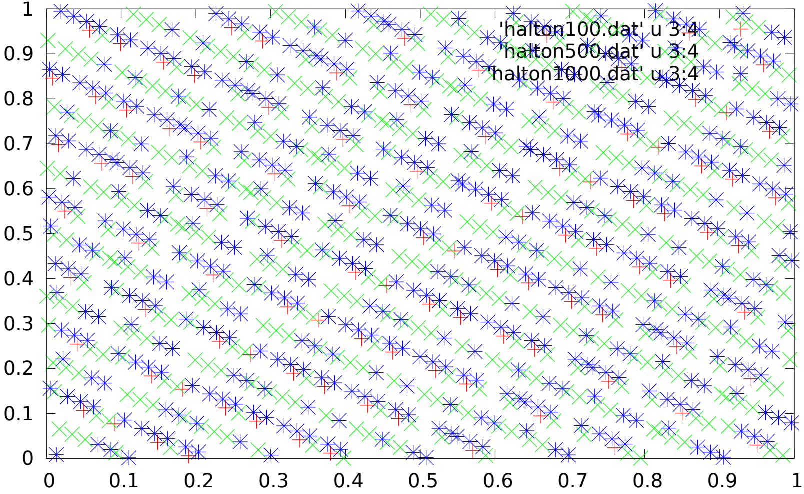
Halton, dimensions 5-6 (bases 11-13)



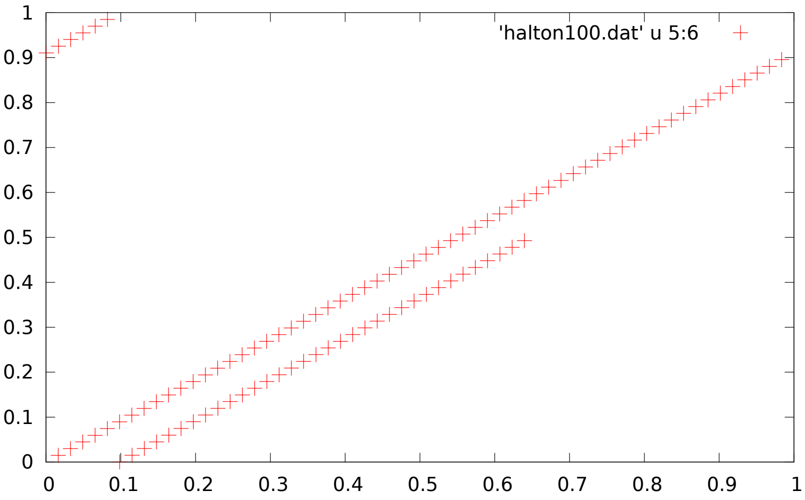
Halton, dimensions 5-6 (bases 11-13)



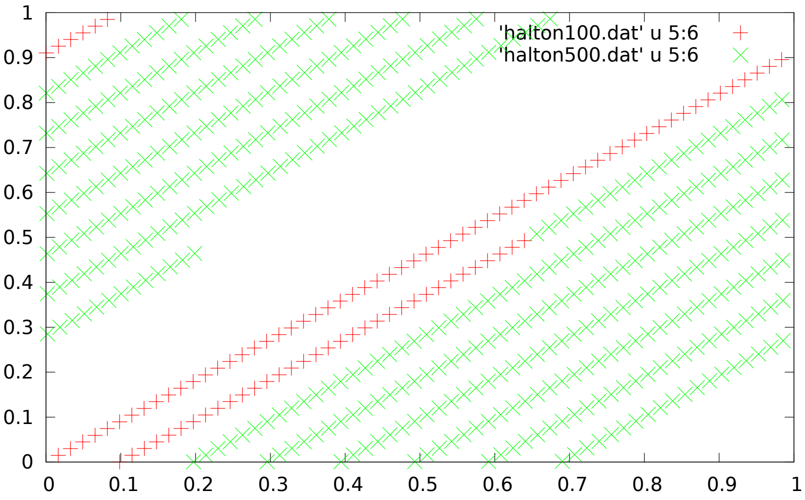
Halton, dimensions 5-6 (bases 11-13)



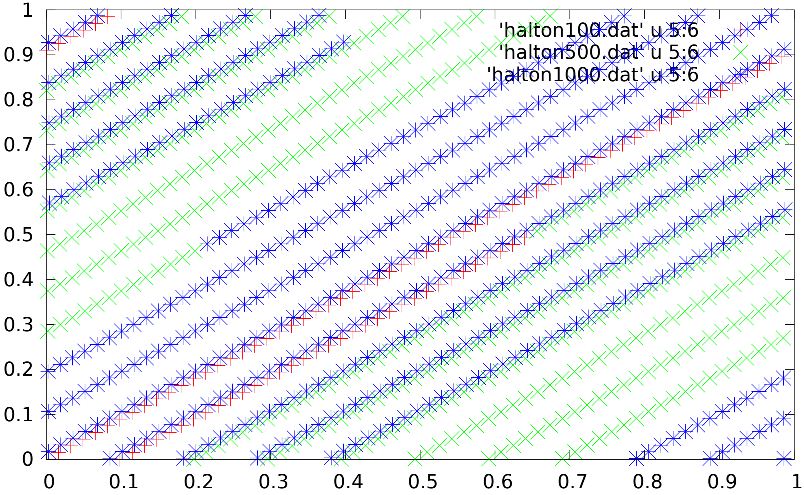
Halton, dimensions 18-19 (base 61-67)



Halton, dimensions 18-19 (base 61-67)



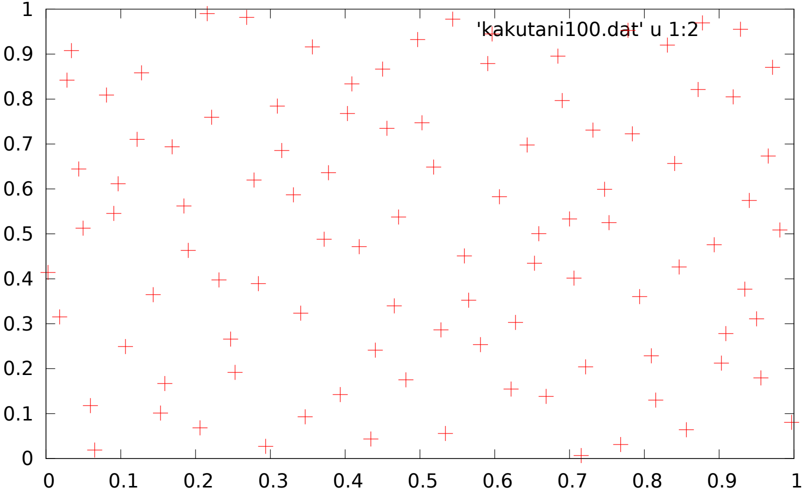
Halton, dimensions 18-19 (base 61-67)



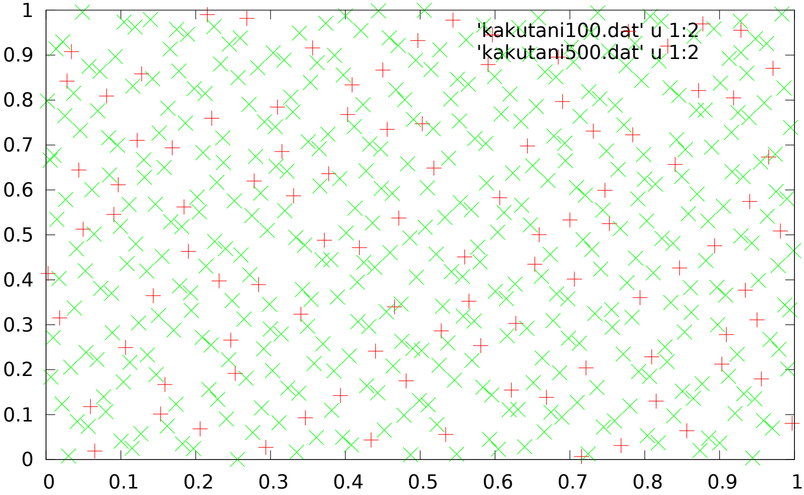
Kakutani - Généralisation d'Halton

Définition : cf. cours Gilles Pagès.

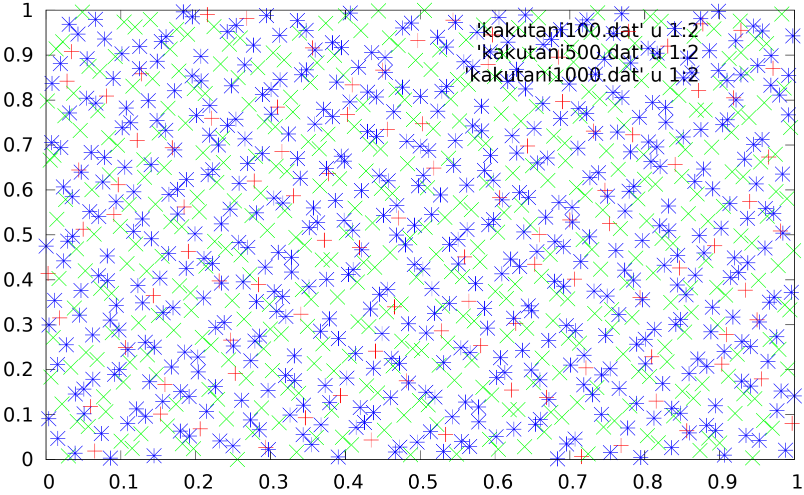
Kakutani, dimensions 1-2 (bases 2-3)



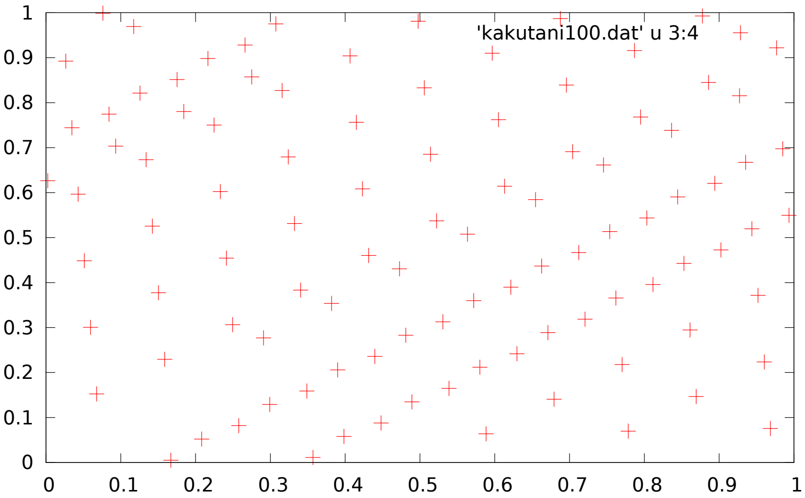
Kakutani, dimensions 1-2 (bases 2-3)



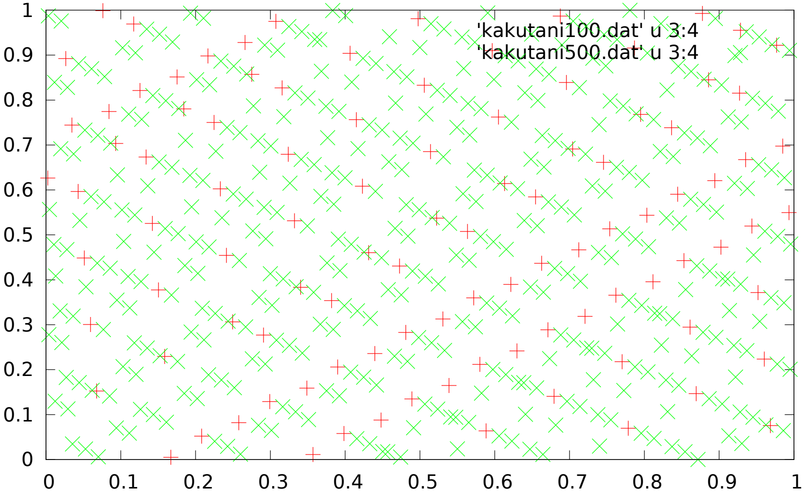
Kakutani, dimensions 1-2 (bases 2-3)



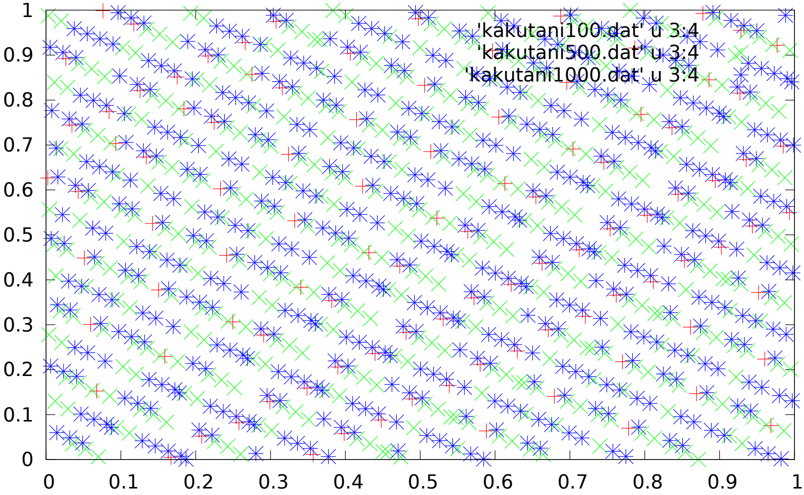
Kakutani, dimensions 5-6 (bases 11-13)



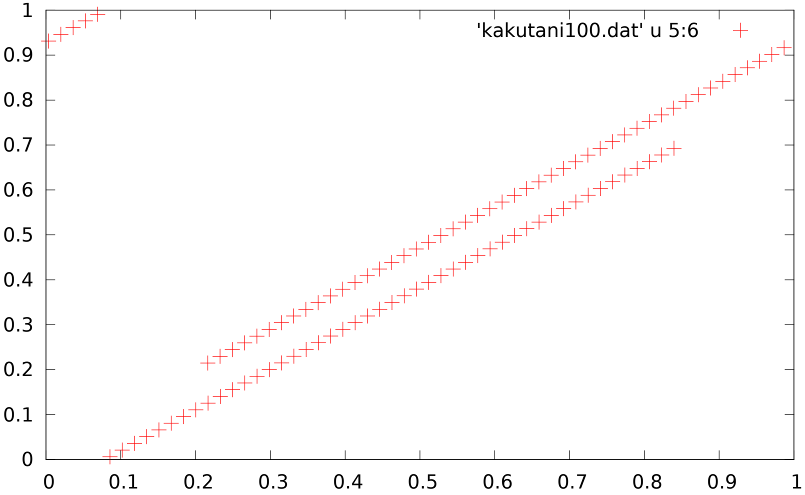
Kakutani, dimensions 5-6 (bases 11-13)



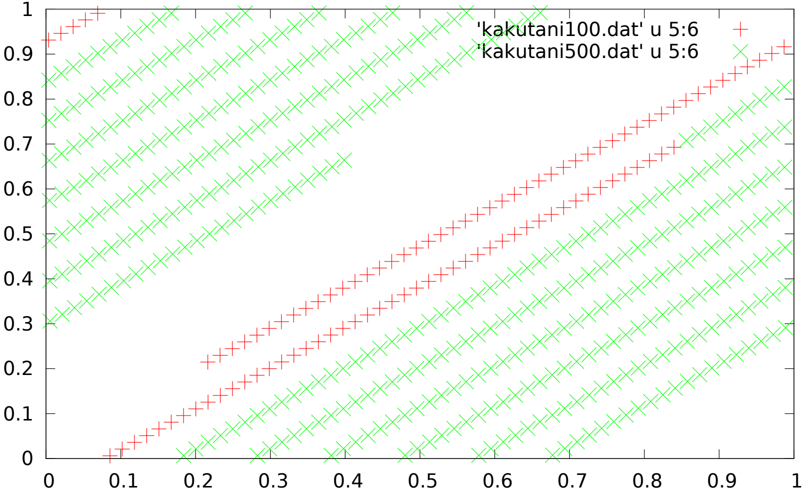
Kakutani, dimensions 5-6 (bases 11-13)



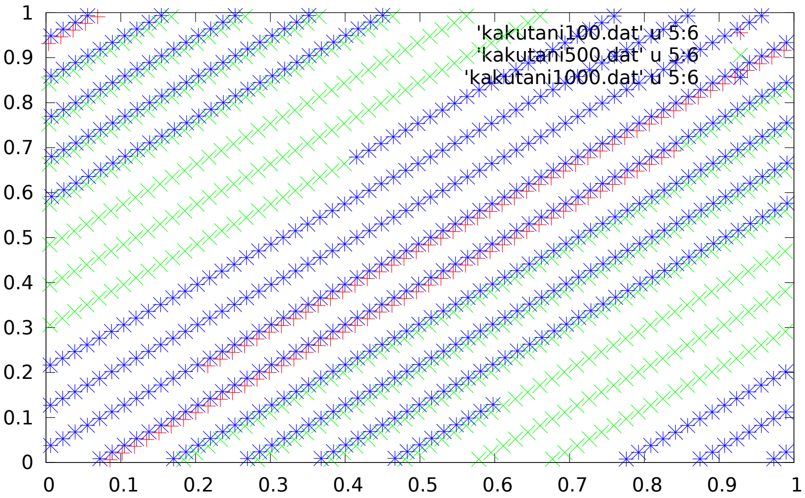
Kakutani, dimensions 18-19 (base 61-67)



Kakutani, dimensions 18-19 (base 61-67)



Kakutani, dimensions 18-19 (base 61-67)



Suite de Faure

Soit p le plus petit nombre premier plus grand que d.
La suite de Faure est définie par

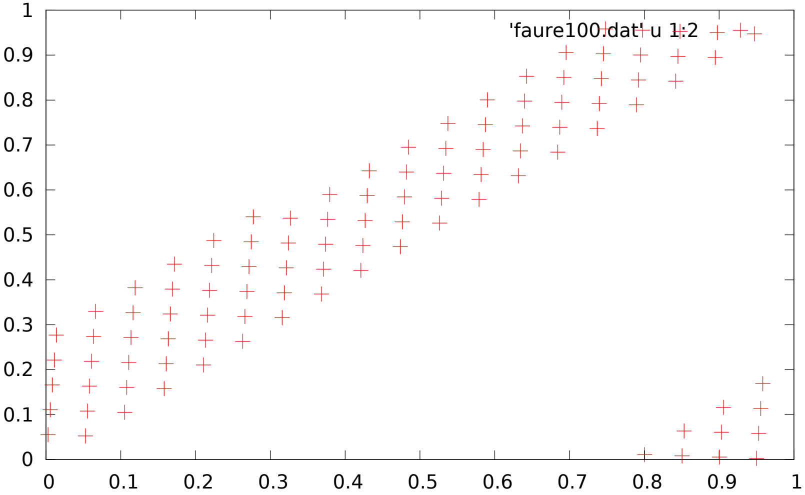
$$\xi_n = \left(\xi_n^{(p)}, C_p(\xi_n^{(p)}), \ldots, C_p^{d-1}(\xi_n^{(p)}) \right),$$

où pour tout $u \in [0,1]$, $u = \sum_k a_k p^{-(k+1)}$

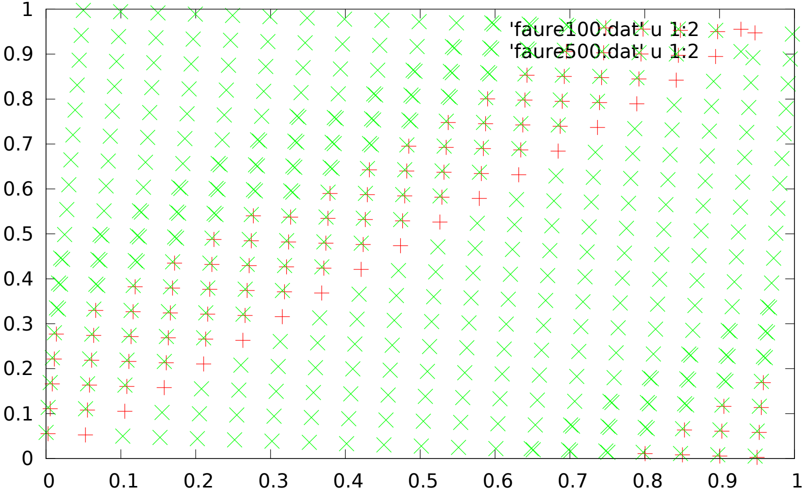
$$C_p(u) = \sum_k b_k p^{-(k+1)}, \quad \text{avec} \quad b_k = \sum_{j \geqslant k} C_k^j a_j \text{ mod } p.$$

Remarque : Dans les exemples suivants, la suite de Faure est construite sur $p = 19$.
Faire le graphique de la dimension 1-2 lorsque $p = 3$.

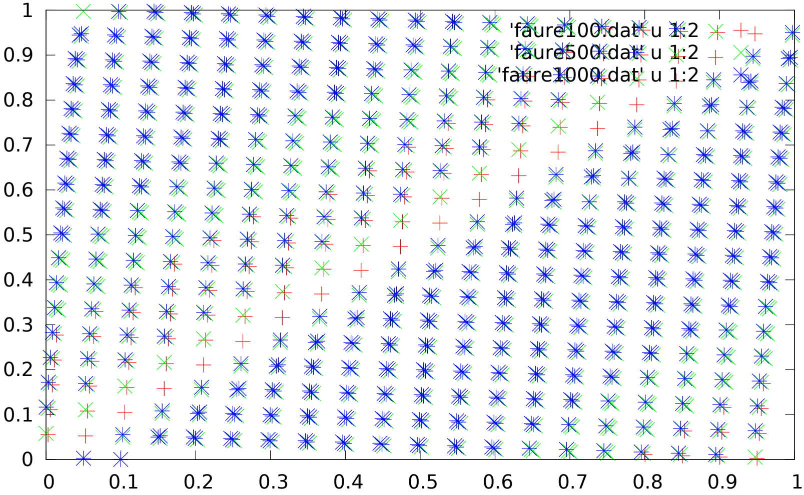
Faure, dimensions 1-2



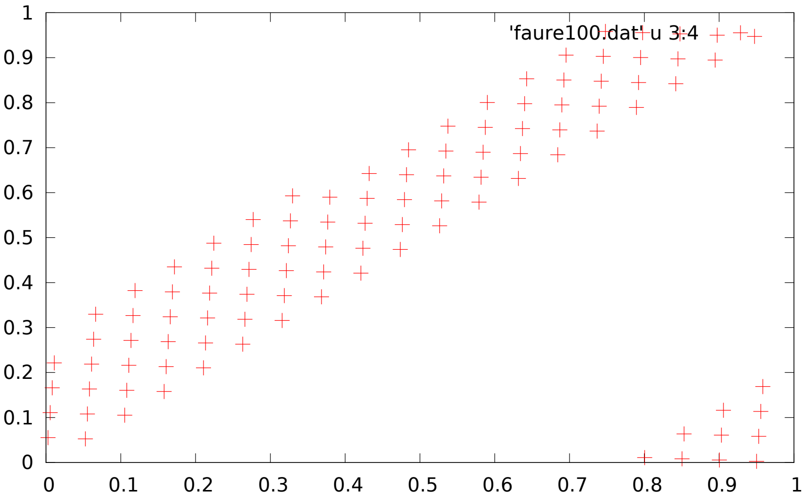
Faure, dimensions 1-2



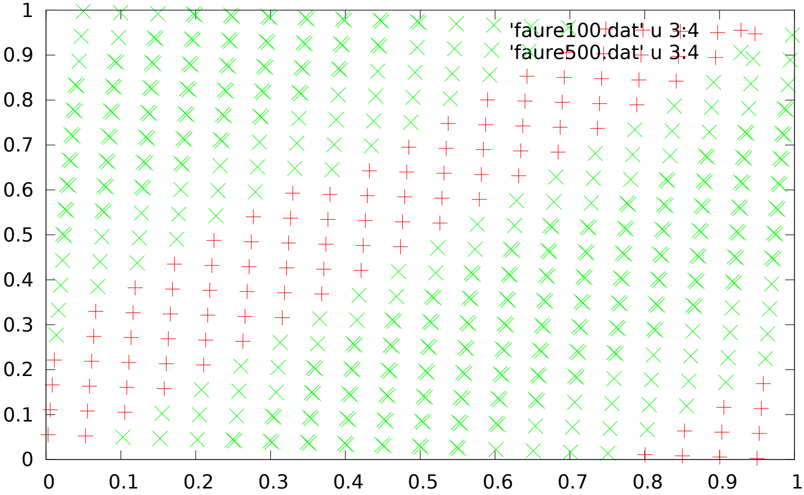
Faure, dimensions 1-2



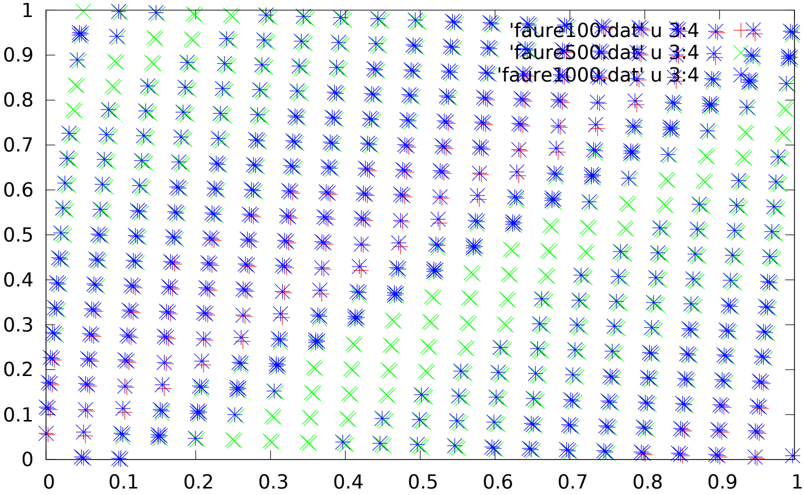
Faure, dimensions 5-6



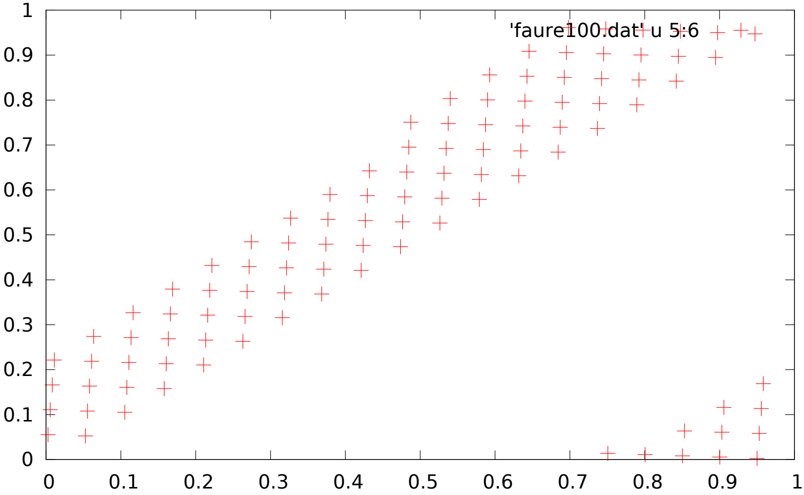
Faure, dimensions 5-6



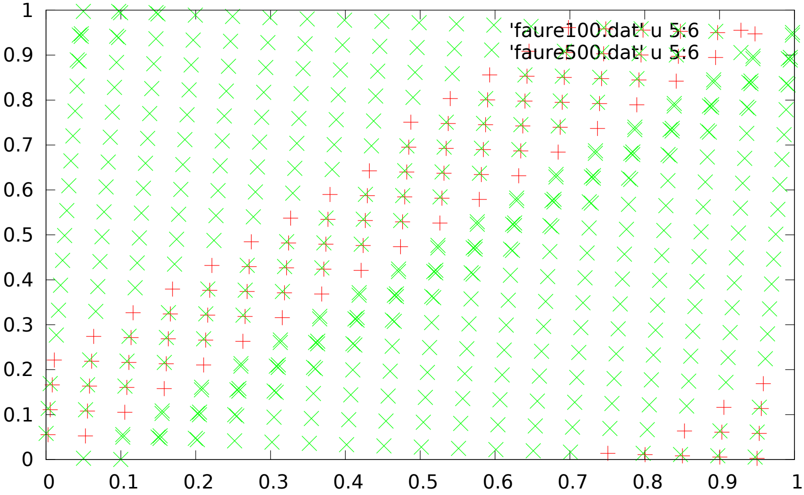
Faure, dimensions 5-6



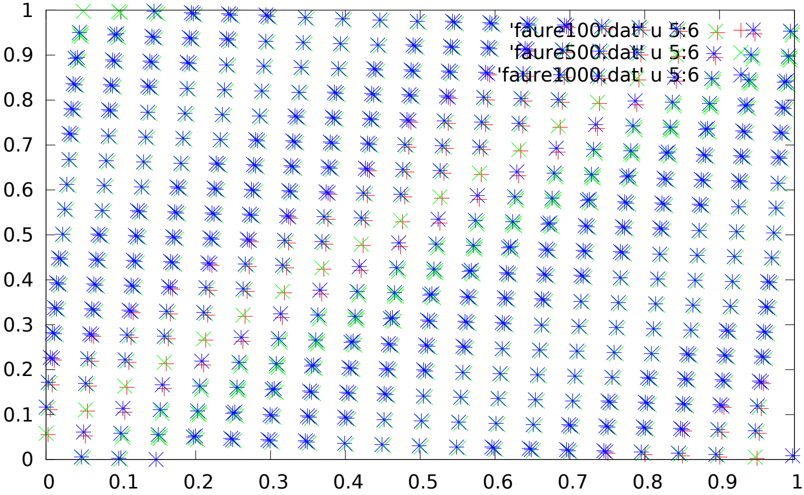
Faure, dimensions 18-19



Faure, dimensions 18-19



Faure, dimensions 18-19



Suite de Sobol, utilisation de la GSL -1-

Implémentation depuis le code de la GSL (Gnu Scientific Library) disponible sur
`https://www.gnu.org/software/gsl/`
Version utilisée : 1.16

Descriptions des fonctions de la GSL en C

- `gsl_qrng * gsl_qrng_alloc (const gsl_qrng_type * T, unsigned d)`
  This function returns a pointer to a newly-created instance of a quasi-random sequence generator of type T and dimension d.
- `void gsl_qrng_free (gsl_qrng * q)`
  This function frees all the memory associated with the generator q.
- `void gsl_qrng_init (gsl_qrng * q)`
  This function reinitializes the generator q to its starting point. Note that quasi-random sequences do not use a seed and always produce the same set of values.
- `gsl_qrng_sobol` générateur de type `gsl_qrng_type *`
  This generator uses the Sobol sequence described in *Antonov, Saleev, USSR Comput. Maths. Math. Phys. 19, 252 (1980)*. It is valid up to 40 dimensions.

Suite de Sobol, utilisation de la GSL -2-

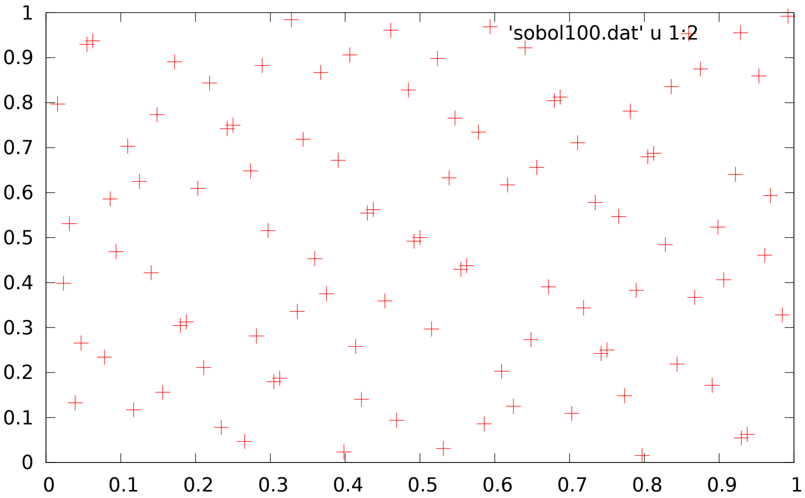
- `int gsl_qrng_get (const gsl_qrng * q, double x[])`
  This function stores the next point from the sequence generator `q` in the array `x`. The space available for `x` must match the dimension of the generator. The point `x` will lie in the range `0 < x_i < 1` for each `x_i`.
- `int gsl_qrng_memcpy (gsl_qrng * dest, const gsl_qrng * src)`
  This function copies the quasi-random sequence generator src into the pre-existing generator dest, making dest into an exact copy of src. The two generators must be of the same type.
- `gsl_qrng * gsl_qrng_clone (const gsl_qrng * q)`
  This function returns a pointer to a newly created generator which is an exact copy of the generator q.

Suite de Sobol, exemple en GSL

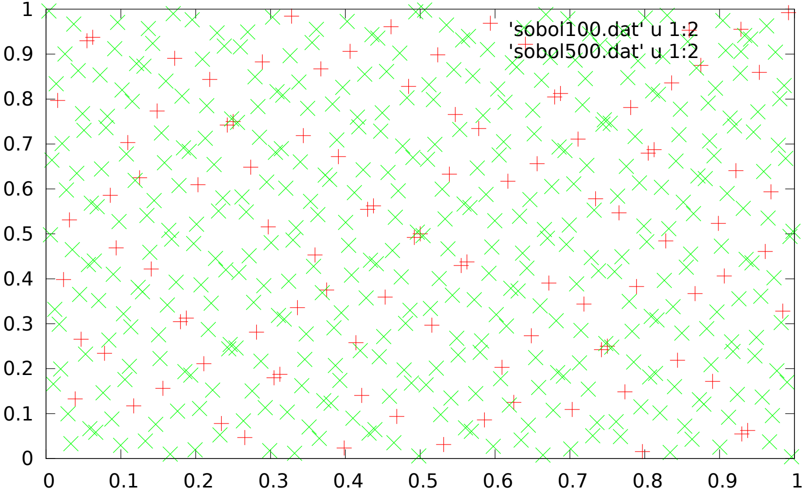
```
#include <stdio.h>
#include <gsl/gsl_qrng.h>

int main (void) {
    int i;
    gsl_qrng * q = gsl_qrng_alloc (gsl_qrng_sobol, 2);

    for (i = 0; i < 1024; i++) {
        double v[2];
        gsl_qrng_get (q, v);
        printf ("%.5f %.5f\n", v[0], v[1]);
    }

    gsl_qrng_free (q);
    return 0;
}
```

Exercice : écrire une classe `sobol` en C++ qui encapsule ces fontions de la GSL

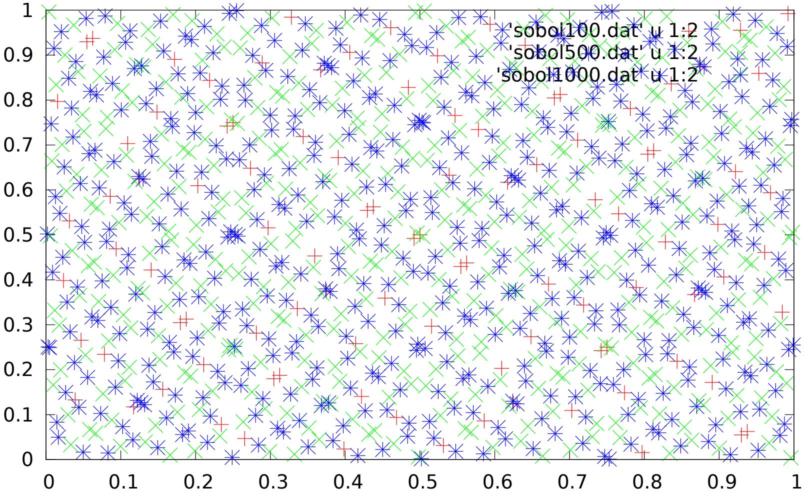
Sobol, dimensions 1-2



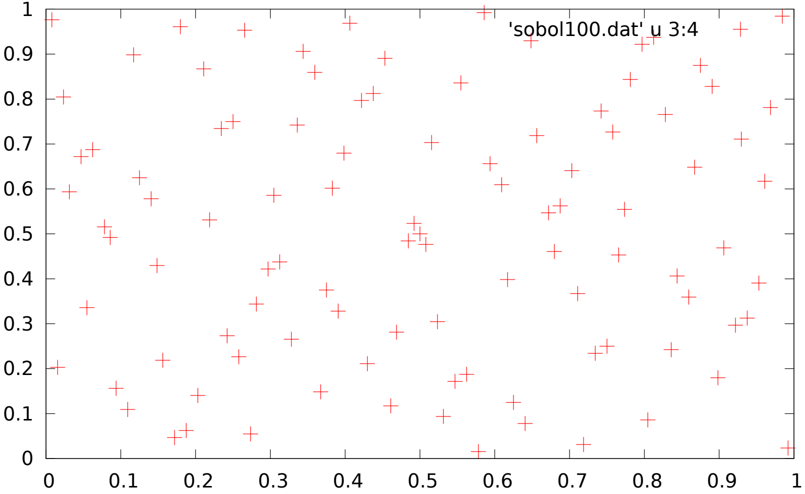
Sobol, dimensions 1-2



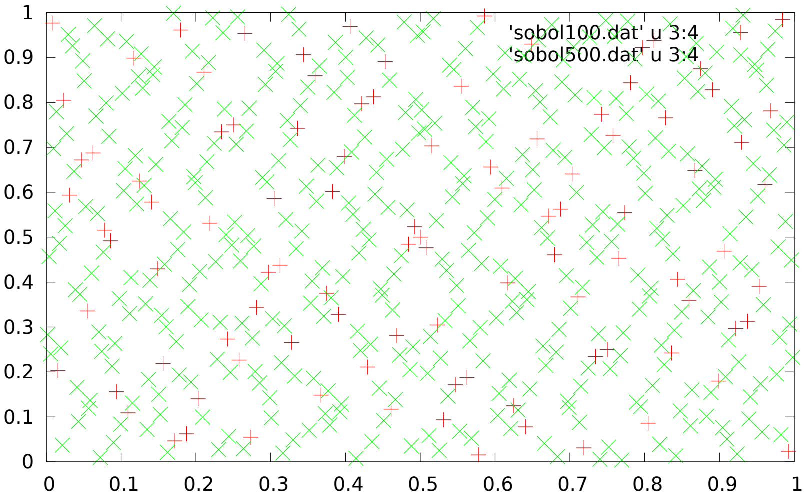
Sobol, dimensions 1-2



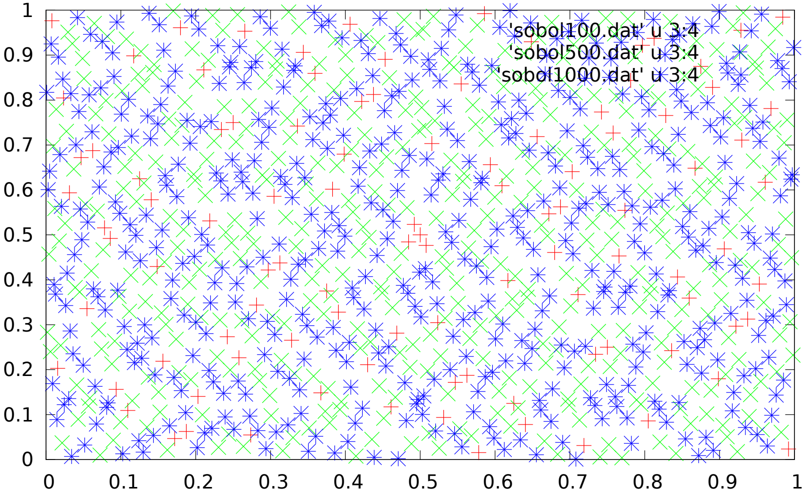
Sobol, dimensions 5-6



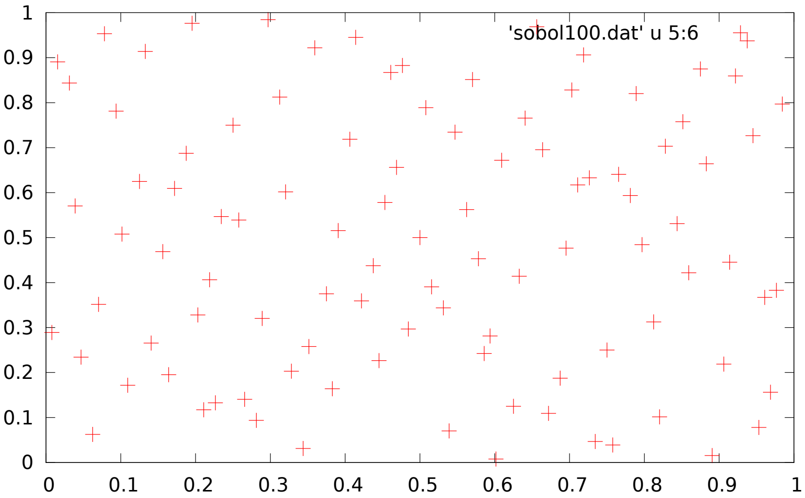
Sobol, dimensions 5-6



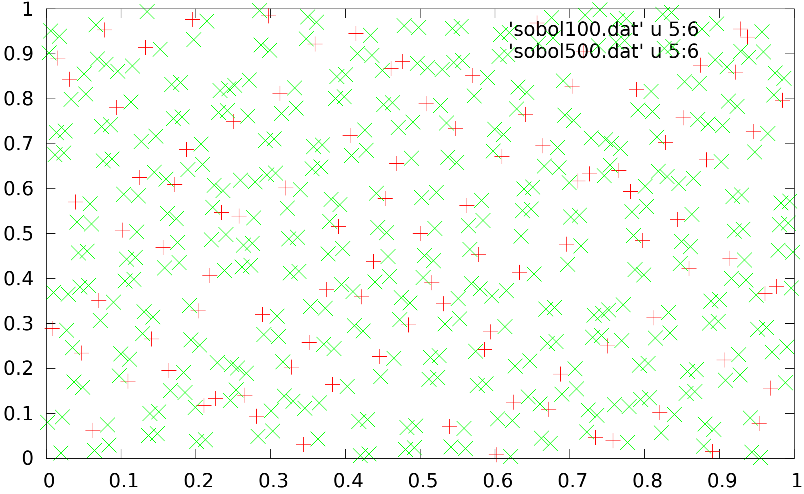
Sobol, dimensions 5-6



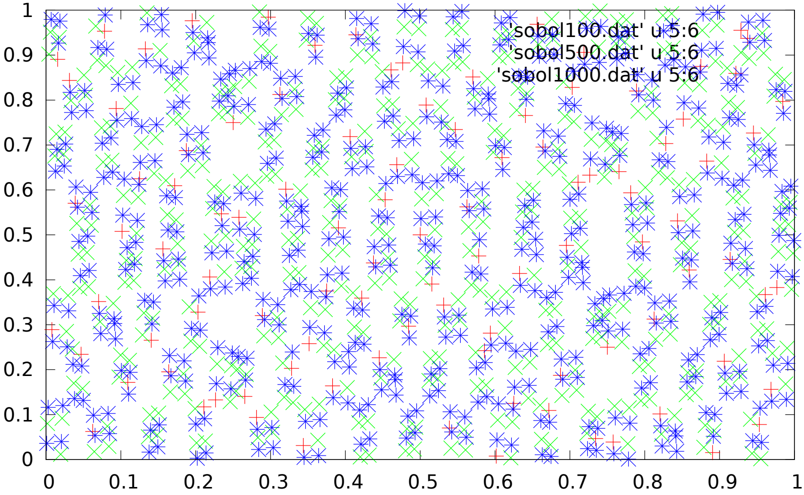
Sobol, dimensions 18-19



Sobol, dimensions 18-19



Sobol, dimensions 18-19



Type gsl_function

Le type gsl_function est une structure qui contient 2 champs :

- function un pointeur de type
  `double (*function)(double x, void * params)`
- params de type `void *`

Par exemple pour coder la fonction paramétrique

$$f(x) = ax^2 + bx + c$$

avec $a = 3$, $b = 2$ et $c = 1$ on peut définir une gsl_function F :

```
struct my_f_params { double a; double b; double c; };
double my_f (double x, void * p) {
   struct my_f_params * pa = (struct my_f_params *)p;
   return  (pa->a * x + pa->b) * x + pa->c;
}

gsl_function F;
struct my_f_params params = { 3.0, 2.0, 1.0 };
F.function = &my_f;
F.params = &params;
```

Exemple d'intégration numérique en GSL

```c
#include <gsl/gsl_integration.h>

double f(double x, void * params) {
  double alpha = *(double *) params;
  return log(alpha*x) / sqrt(x);
}

int main(void) {
  gsl_integration_workspace * w = gsl_integration_workspace_alloc(1000);
  double result, error;
  double alpha = 1.0;
  gsl_function F;
  F.function = &f;
  F.params = &alpha;
  gsl_integration_qags(&F, 0, 1, 0, 1e-7, 1000,
                       w, &result, &error);
  printf("result          = % .18f\n", result);
  printf("estimated error = % .18f\n", error);
  gsl_integration_workspace_free(w);
  return 0;
}
```

Lambda function

On peut voir une Lambda fonction comme un objet fonctionnel dont l'écriture syntaxique est plus épurée. Prenons l'exemple suivant :

```
struct compare {
    bool operator()(int a, int b) const { return abs(a) < abs(b); }
}
vector<int> v = {50, -10, 20, -30};
std::sort(v.begin(), v.end());  // the default sort
std::sort(v.begin(), v.end(), compare());
```

On peut remplacer l'objet `compare` par une lambda function dont la syntaxe est la suivante :

```
std::sort(v.begin(), v.end(), [](int a, int b) {return abs(a)<abs(b);});
```

On peut stocker une lambda function dans un objet `std::function`.

```
std::function<bool(int, int)> f2 =
    [](int a, int b) {return abs(a) < abs(b);};
auto f = [](int a, int b) { return abs(a) < abs(b); };
```

Capture and paramètres

Les crochets [ ] sont obligatoires et définissent une liste de capture indiquant si des variables locales doivent être des paramètres de la lambda function. Voici les options

- [ ] liste vide, aucun paramètre capturé
- [=] toutes les variables locales utilisées dans le code de la lambda sont capturées *par copie*
- [&] toutes les variables locales utilisées dans le code de la lambda sont capturées *par référence*
- [x] uniquement la variable x capturée par copie
- [x, &y] la variable x capturée par copie et la variable y capturée par référence

Exemple :

```
void print_modulo(const vector<int>& v, ostream& os, int m) {
    for_each(v.begin(), v.end(),
      [&os,m](int x) { if (x%m==0) os << x << '\n'; });}
```

Functor équivalent à la lambda function précédente

```
class Modulo_print {
    ostream& os; // members to hold the capture list
    int m;
  public:
    Modulo_print(ostream& s, int mm) : os(s), m(mm) {}
    void operator()(int x) const { if (x%m==0) os << x << '\n'; }
};

void print_modulo(const vector<int>& v, ostream& os, int m) {
   for_each(v.begin(), v.end(), Modulo_print(os,m));
}
```

Functor équivalent à la lambda function précédente

```
class Modulo_print {
    ostream& os; // members to hold the capture list
    int m;
  public:
    Modulo_print(ostream& s, int mm) : os(s), m(mm) {}
    void operator()(int x) const { if (x%m==0) os << x << '\n'; }
};

void print_modulo(const vector<int>& v, ostream& os, int m) {
   for_each(v.begin(), v.end(), Modulo_print(os,m));
}
```

Une lambda function par défaut est **const** (cf. exemple précédent) et ne modifie pas les paramètres capturés. On peut changer ce comportement en déclarant la lambda **mutable**. Par exemple

```
void algo(vector<int> & v) {
    int count = v.size();
    std::generate(v.begin(), v.end(),
        [count]() mutable { return --count; });
}
```

Lambda function et pointeur de fonction

Important : Une lambda function sans paramètre (qui ne capture rien) peut être assigné à un pointeur de fonction. Par exemple

```
double (*p1)(double) = [](double a) { return sqrt(a); }
```

Mais attention, il faut vraiment une liste de capture vide et que les types correspondent :

```
double (*p2)(double) = [&](double a) { return sqrt(a); }  // Erreur
double (*p3)(int) = [](double a) { return sqrt(a); }      // Erreur
```

Si le type de retour d'une lambda ne peut pas être inféré par le compilateur il y a une erreur (à la compilation) et il faut indiquer le type avec la syntaxe suivante :

```
double y;
auto z = [y]() -> int { if (y) return 1; else return 2; };
double (*p)(int) = z;
```

Attention, on ne peut pas assigner un objet `std::function` à un pointeur de fonction...

Appel du code intégration numérique de la GSL avec un objet fonctionnel

Au tableau !

Surcharge d'opérateurs
- Par fonctions membres
- Par fonctions amies

QMC - Discrépance
- Koksma-Hlawka
- Dimension 1 - Van der Corput

Classe `p_adic`

Dimension supérieure
- Halton
- Kakutani
- Faure
- Suite de Sobol, intégration de la GSL
- Un peu plus sur la GSL, `gsl_function` et Lambda function en C++11

Pour compléter la classe `sobol` en C++11
- Référence sur rvalue
- Constructeur de déplacement
- `std::move` et `std::forward`
- Retour à `sobol`

Référence sur *rvalue*

- *lvalue* : expression qui a un nom, une adresse, c'est donc une variable qui peut se positionner à gauche du signe d'affectation *left value* Référence sur une `lvalue` : symbole &
- *rvalue* : expression anonyme, qui n'est pas une *lvalue* Référence sur une `rvalue` : symbole &

Exemples :

```
int && r = 4;
```

```
struct X {
// définition d'une classe
};
X && r = X();
```

L'intérêt de ces références sur *rvalue* est principalement la mise en place de

- Constructeur de déplacement (move constructor et move operator)
- Transfert parfait (perfect forwarding) d'arguments

Constructeur de déplacement

Au tableau, classe `vecteur`

Fonction `std::move`

```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t );
```

La fonction générique `std::move` renvoie une *rvalue* référence sur son argument (conversion vers une *rvalue* reference).
Attention : on dit explicitement que l'objet passé en argument ne sera plus jamais utilisé : son contenu est détruit.

Exemple d'utilisation :

```
template<class T>
void swap(T & a, T & b) {
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
```

Le code précédent fonctionne si la classe T possède un opérateur d'affectation de déplacement...

Fonction `std::forward`

```
template< class T >
T&& forward( typename std::remove_reference<T>::type& t );
template< class T >
T&& forward( typename std::remove_reference<T>::type&& t );
```

La fonction générique `std::forward` produit une référence sur *rvalue* uniquement si l'argument est de type `rvalue`.

Si l'argument est une variable, celle-ci n'est pas modifiée par l'appel de `std::forward` contrairement à `std::move`.

Exemple pour distinguer `std::move` et `std::forward`

```
#include <iostream>

void overloaded(int const & arg) { std::cout << "by lvalue\n"; }
void overloaded(int && arg) { std::cout << "by rvalue\n"; }

template<typename T>
void forwarding(T && arg) {
    std::cout << "via std::forward: ";
    overloaded( std::forward<T>(arg) );
    std::cout << "via std::move: ";
    overloaded( std::move(arg) );
    std::cout << "by simple passing: ";
    overloaded( arg );
    std::cout << std::endl;
}

int main() {
    std::cout << "initial caller passes rvalue:\n";
    forwarding( 5 );
    std::cout << "initial caller passes lvalue:\n";
    int x = 5;
    forwarding( x );
    return 0;
}
```

Exercice : écrire le constructeur et l'opérateur `move` pour la classe `sobol`