La récursion — un outil puissant

Peter Schlagheck

Université de Liège

La récursion

Une fonction (ou routine) peut appeler non seulement d'autres fonctions (ou routines) en cours de son exécution, mais aussi elle-même .

La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

```
int factoriel( int n )
{
  int nfac = 1;
  for ( int i = 1; i <= n; i++ )
    nfac *= i;
  return nfac;
}
```

La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 6;
  cout << factoriel( k ) << endl;
}
```

Exécution:

```
Segmentation fault
```

La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 6;
  cout << factoriel( k ) << endl;
}
```

Exécution:

```
720
```

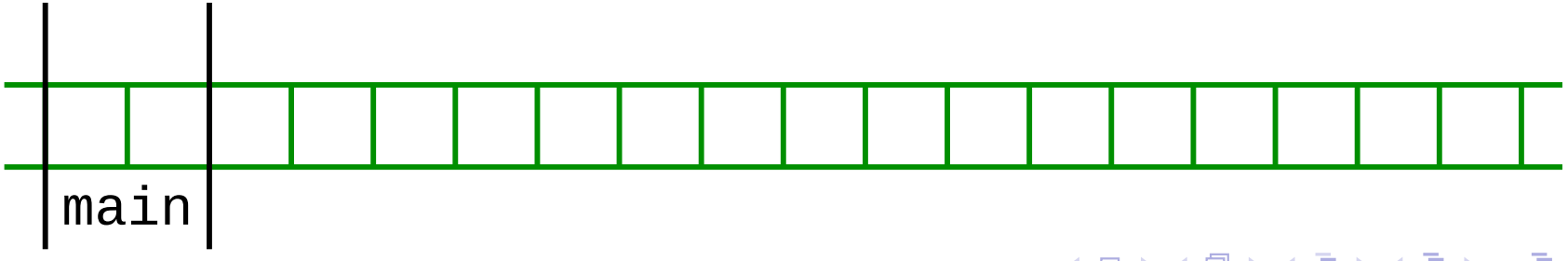
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



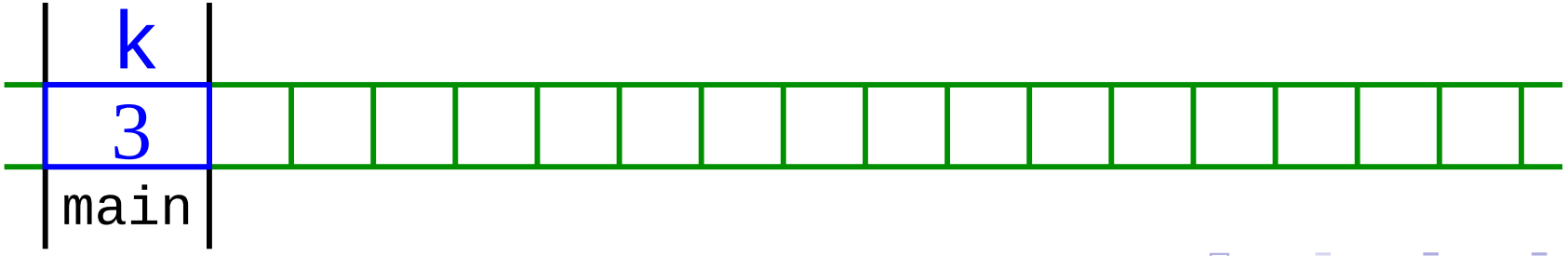
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



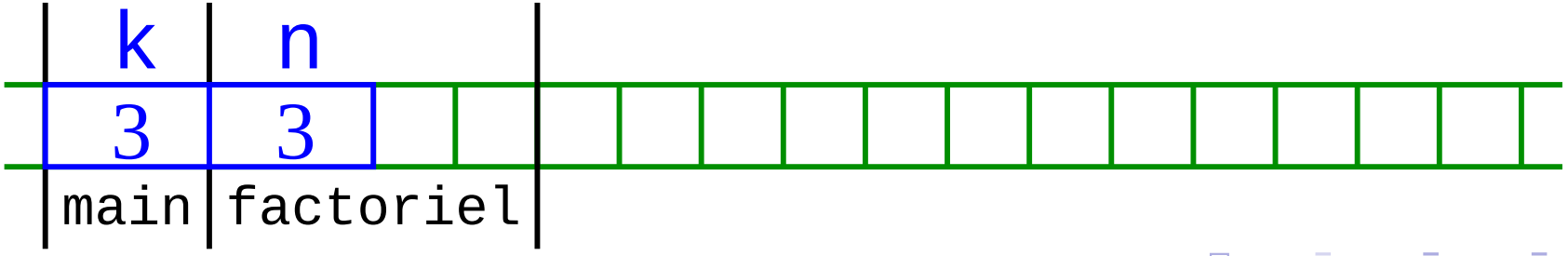
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



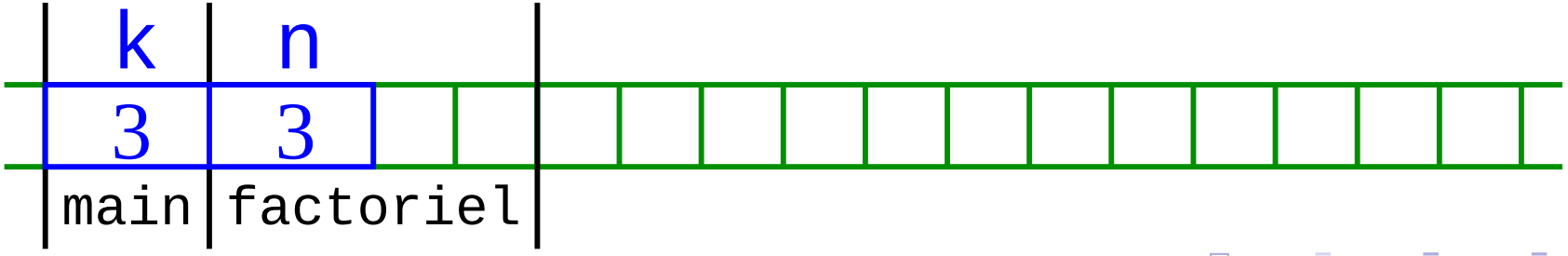
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



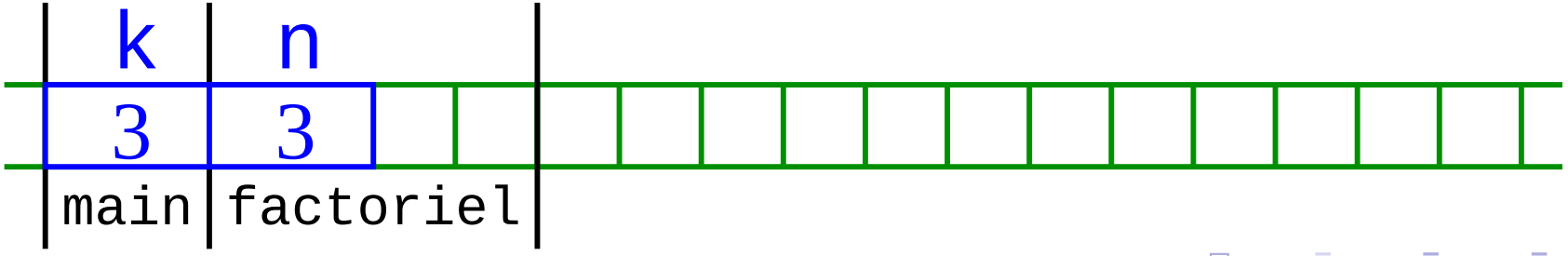
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 )  );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



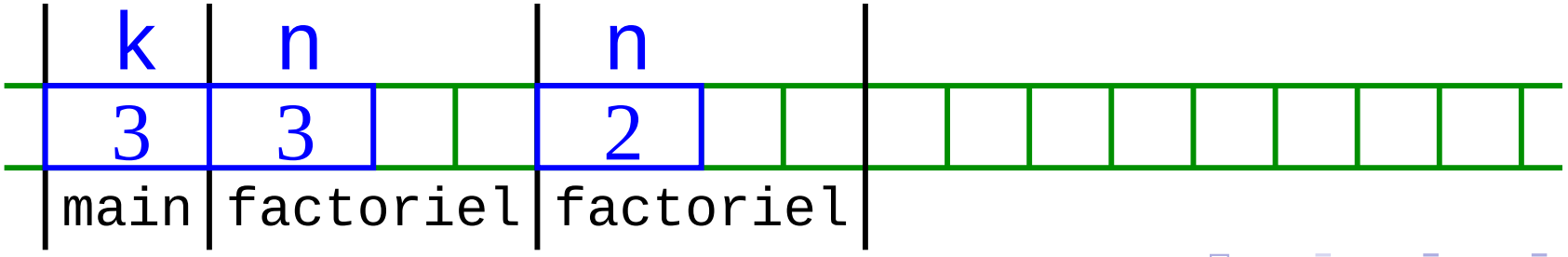
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



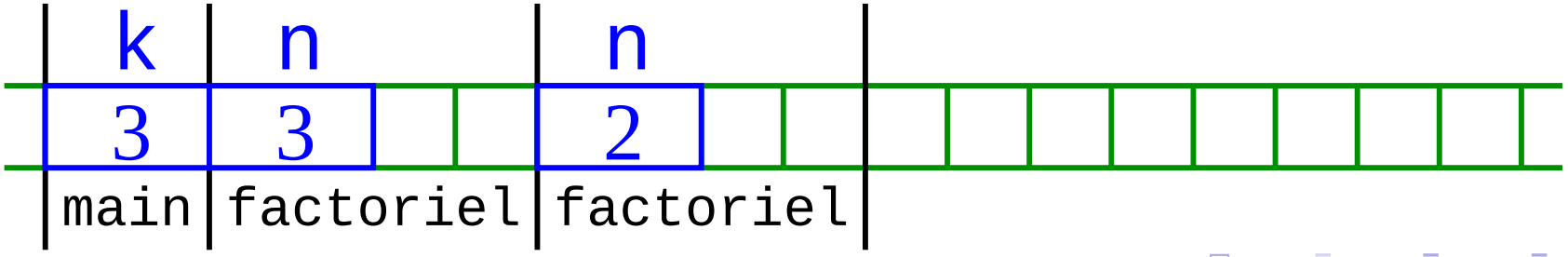
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



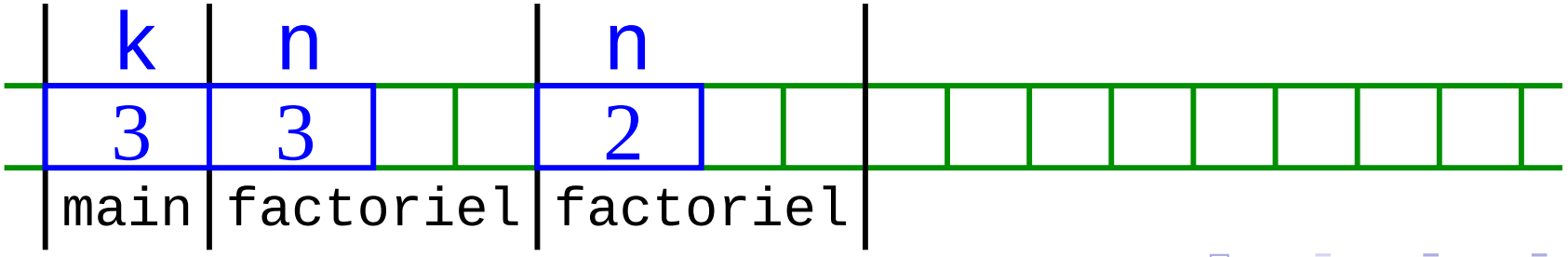
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 )  );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



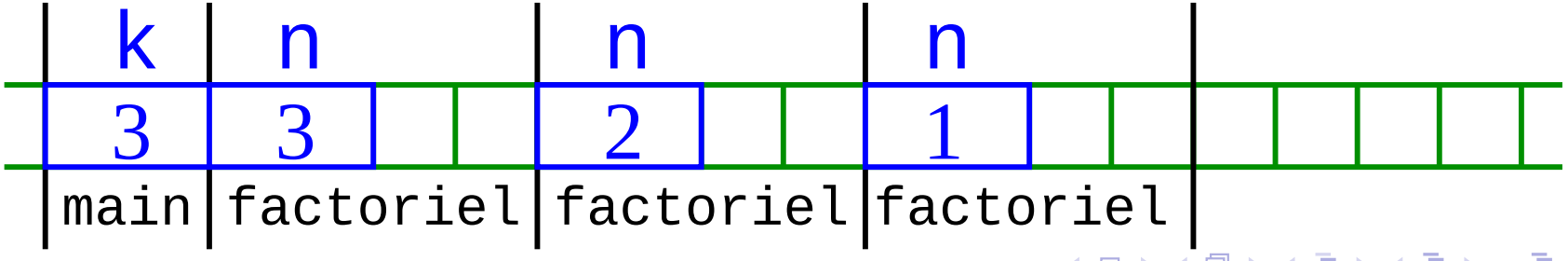
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



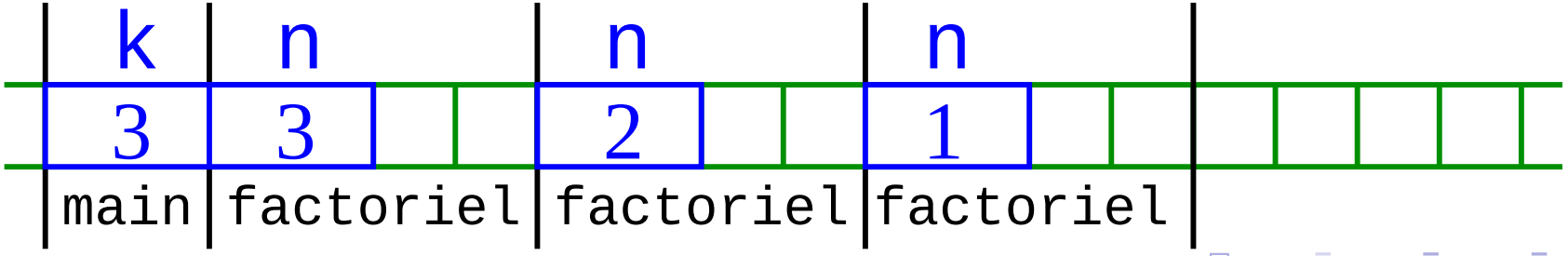
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



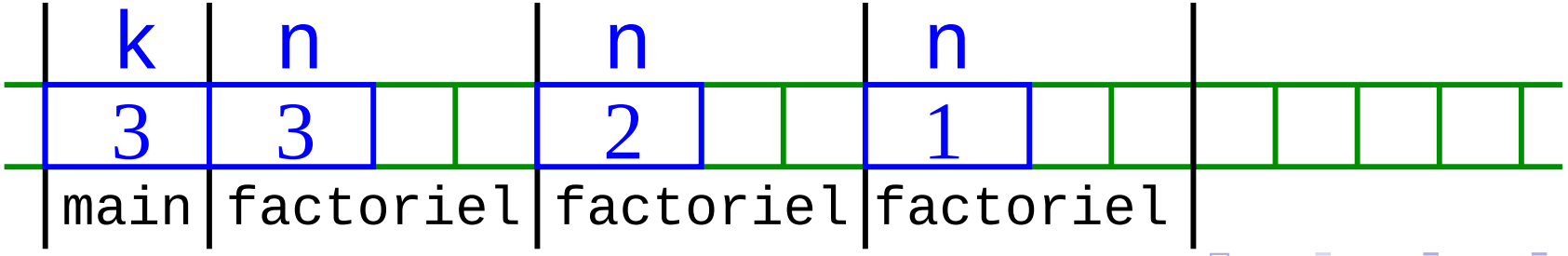
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 )  );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



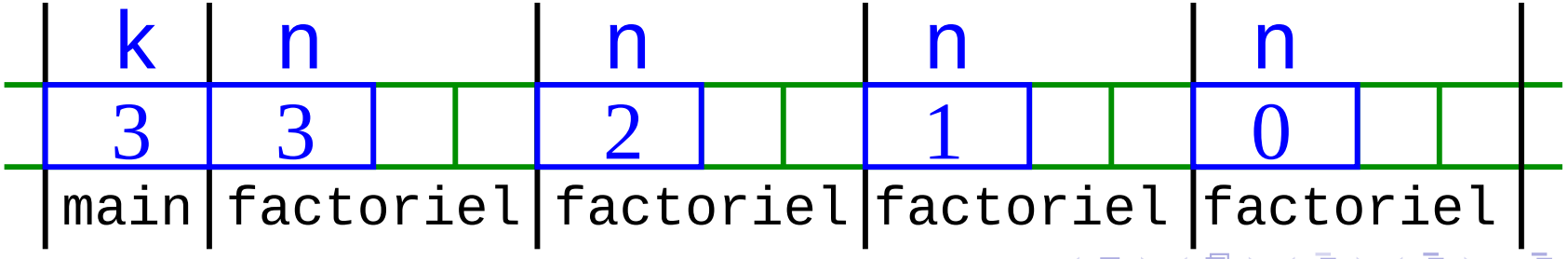
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



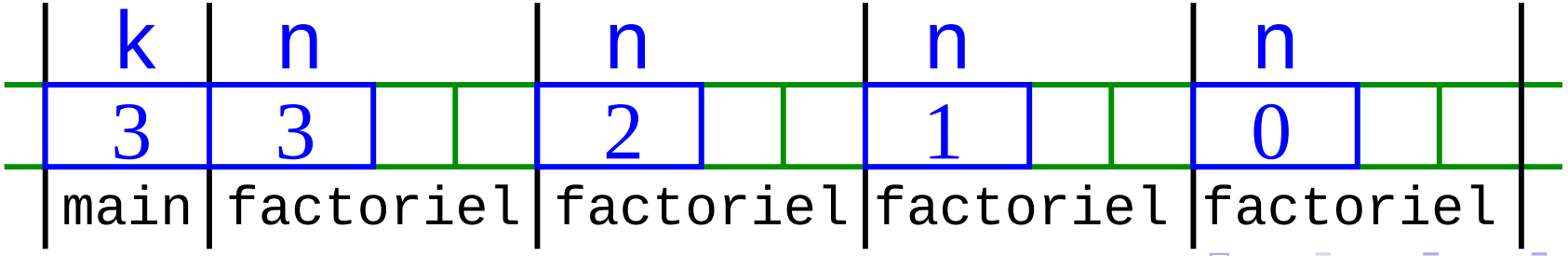
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



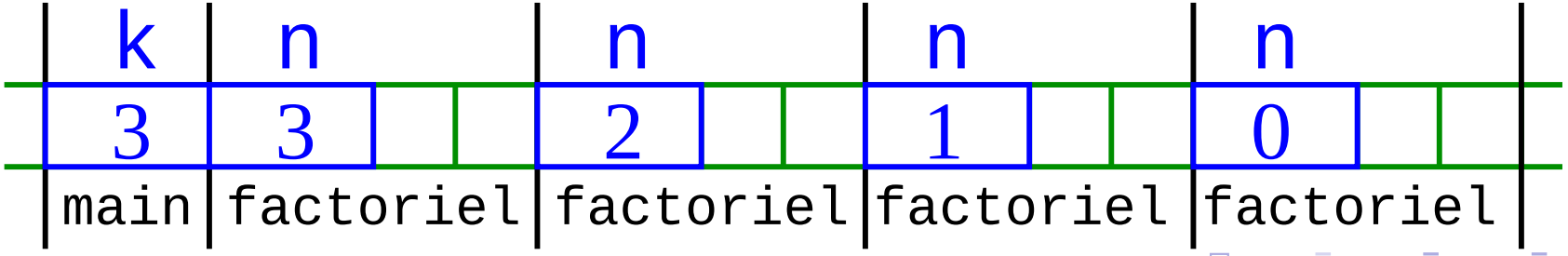
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



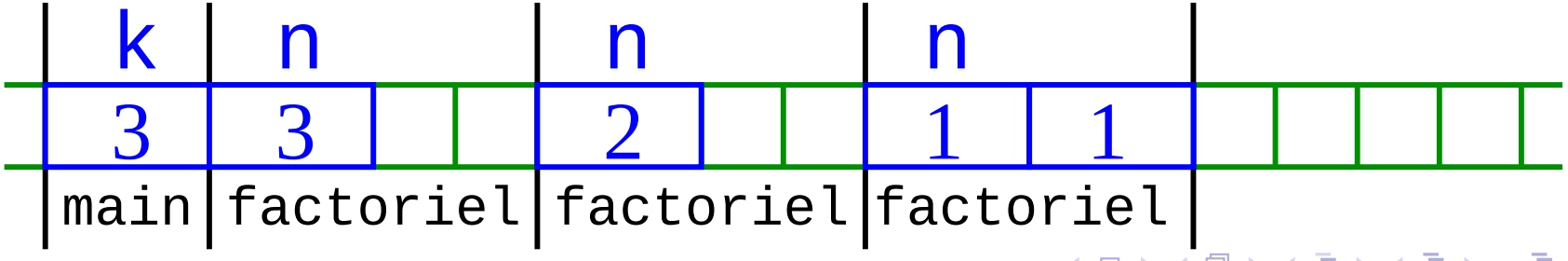
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



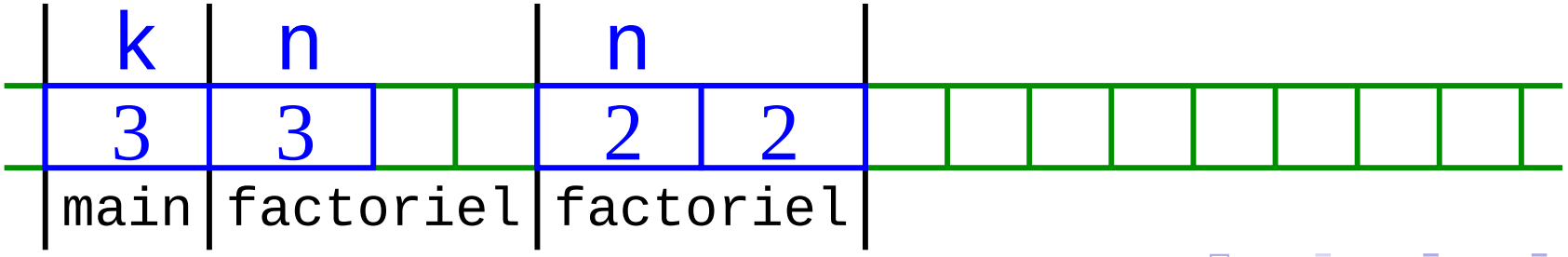
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



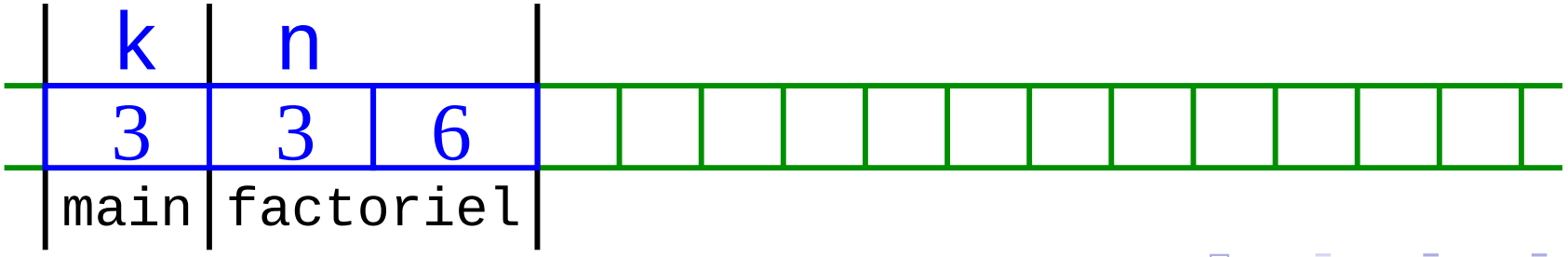
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



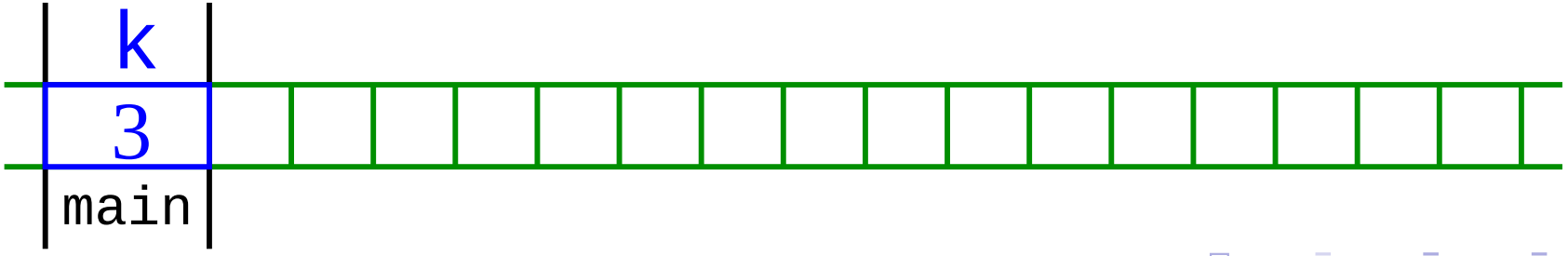
La récursion

Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$

Alternative: $n! = n \times (n-1)!$

```
int factoriel( int n )
{
  if ( n == 0 )
    return 1;
  return ( n * factoriel( n - 1 ) );
}

int main()
{
  int k = 3;
  cout << factoriel( k ) << endl;
}
```



La récursion

Une fonction (ou routine) peut appeler non seulement d'autres fonctions (ou routines) en cours de son exécution, mais aussi elle-même .

Ceci permet de résoudre certains problèmes de programmation compliqués par des programmes très compacts et élégants (et faciles à présenter dans le cadre d'un cours. . .)

. . . si on arrive à adapter convenablement sa manière de concevoir une stratégie de solution

et si le programme à écrire n'est pas soumis à des contraintes trop importantes concernant la mémoire sollicitée et son temps d'exécution.

Exemples

- Détermination d'une puissance entière :
  $x \stackrel{?}{=} 7^n$ pour un $n \in \mathbb{N}$
- Décomposition en produit de facteurs premiers :
  p.ex. $18 = 2 \times 3 \times 3$
- Permutations aléatoires :
  $\{1, 2, 3, 4, 5\} \mapsto \{4, 1, 5, 3, 2\}$
- Itérations multiples : $\sum_{i_1=1}^{N} \cdots \sum_{i_n=1}^{N} f(i_1, \ldots, i_n)$
- Problèmes de partition d'un ensemble
- Recherche des zéros
- Intégration numérique avec des intervalles adaptés
- Résolution des équations différentielles ordinaires

Détermination d'une puissance entière

$\longrightarrow$ Ecrire une fonction qui permet de déterminer si un entier x satisfait à l'équation

$$x = k^n$$

pour un $n \in \mathbb{N}$, où $k > 0$ est un entier donné.

Exemple : $2401 = 7^4$ (le saviez-vous ?)

Détermination d'une puissance entière

$\longrightarrow$ Ecrire une fonction qui permet de déterminer si un entier x satisfait à l'équation

$$x = k^n$$

pour un $n \in \mathbb{N}$, où $k > 0$ est un entier donné.

Stratégie générale :

$\rightarrow$ vérifier si x peut être divisé par k ;

$\rightarrow$ si oui, appliquer cette même stratégie pour x/k ;

$\rightarrow$ arrêt de récursion si $x/k = 1$.

Détermination d'une puissance entière

```
int puissance_de_k( int x, int k, int &n )
{
  if ( x % k )
    return 0;
  n++;
  if ( x == k )
    return 1;
  return ( puissance_de_k( x / k, k, n ) );
}
```

$\longrightarrow$ Il faut appeler cette fonction avec la valeur initiale $n = 0$ de son troisième argument.

$\longrightarrow$ Ce dernier contiendra l'exposant recherché si x est une puissance entière de k.

Détermination d'une puissance entière

```
int puissance_de_k( int x, int k, int &n )
{
  if ( x % k )
    return 0;
  n++;
  if ( x == k )
    return 1;
  return ( puissance_de_k( x / k, k, n ) );
}

int main()
{
  int x = 2401;
  int k = 7;
  int n = 0;
  if ( puissance_de_k( x, k, n ) )
    cout << x << " = " << k << "^" << n << endl;
}
```

Exécution : `2401 = 7^4`

Décomposition en facteurs premiers

$\longrightarrow$ Ecrire une fonction qui permet d'afficher la décomposition en produit de facteurs premiers pour un nombre n donné.

Exemple :
$285683662 = 2 \times 11 \times 11 \times 31 \times 113 \times 337$ (le saviez-vous ?)

Décomposition en facteurs premiers

$\longrightarrow$ Ecrire une fonction qui permet d'afficher la décomposition en produit de facteurs premiers pour un nombre n donné.

Stratégie générale :

$\rightarrow$ itérer l'entier $k = 2, 3, 4, \ldots$
et vérifier si n peut être divisé par k ;

$\rightarrow$ si oui, afficher k et recommencer l'itération pour n/k ;

$\rightarrow$ arrêt de récursion si $k > \sqrt{n}$;
dans ce cas, n est un nombre premier.

Décomposition en facteurs premiers

```
#include <iostream>
using namespace std;

void affiche_facteurs( int n )
{
  for ( int k = 2; k * k < n; k++ )
    if ( n % k == 0 )
    {
      cout << k << " ";
      affiche_facteurs( n / k );
      return;
    }
  cout << n << endl;
}

int main()
{
  int n = 60;
  affiche_facteurs( n );
}
```

Exécution :

```
2 2 3 5
```

Des permutations aléatoires

$\longrightarrow$ Déterminer une permutation aléatoire des nombres $\{1, 2, \ldots, N\}$ pour un entier N donné

(p.ex., afin d'établir un ordre de passage pour un examen oral)

Exemple : $\{1, 2, 3, 4, 5\} \mapsto \{4, 1, 5, 3, 2\}$ pour $N = 5$

Des permutations aléatoires

$\longrightarrow$ Déterminer une permutation aléatoire des nombres $\{1, 2, \ldots, N\}$ pour un entier N donné

Stratégie :

$\rightarrow$ définir un tableau à N entiers, initialisé à $\{0, \ldots, N-1\}$, qui contiendra la permutation à la fin ;

```
const int N = 20;
int perm[ N ];
for ( int i = 0; i < N; i++ )
  perm[ i ] = i;
```

Des permutations aléatoires

$\longrightarrow$ Déterminer une permutation aléatoire des nombres $\{1, 2, \ldots, N\}$ pour un entier N donné

Stratégie :

$\rightarrow$ définir un tableau à N entiers, initialisé à $\{0, \ldots, N-1\}$, qui contiendra la permutation à la fin ;

$\rightarrow$ choisir un nombre entier aléatoire $k \in \{0, \ldots, N-1\}$;

```
int random( int N )
{
  int n = rand();
  return ( n % N );
}
```

Des permutations aléatoires

$\longrightarrow$ Déterminer une permutation aléatoire des nombres $\{1, 2, \ldots, N\}$ pour un entier N donné

Stratégie :

$\rightarrow$ définir un tableau à N entiers, initialisé à $\{0, \ldots, N-1\}$, qui contiendra la permutation à la fin ;

$\rightarrow$ choisir un nombre entier aléatoire $k \in \{0, \ldots, N-1\}$;

```
int random( int N )
{
  int n = rand();
  if ( n >= ( RAND_MAX / N ) * N )
    return ( random( N ) );
  return ( n % N );
}
```

Des permutations aléatoires

$\longrightarrow$ Déterminer une permutation aléatoire des nombres $\{1, 2, \ldots, N\}$ pour un entier N donné

Stratégie :

$\rightarrow$ définir un tableau à N entiers, initialisé à $\{0, \ldots, N-1\}$, qui contiendra la permutation à la fin ;

$\rightarrow$ choisir un nombre entier aléatoire $k \in \{0, \ldots, N-1\}$;

$\rightarrow$ échanger l'élément $N-1$ du tableau avec son élément n ;

```
void echange( int &i, int &j )
{
  int k = i;
  i = j;
  j = k;
}
```

Des permutations aléatoires

$\longrightarrow$ Déterminer une permutation aléatoire des nombres $\{1, 2, \ldots, N\}$ pour un entier N donné

Stratégie :

$\rightarrow$ définir un tableau à N entiers, initialisé à $\{0, \ldots, N-1\}$, qui contiendra la permutation à la fin ;

$\rightarrow$ choisir un nombre entier aléatoire $k \in \{0, \ldots, N-1\}$;

$\rightarrow$ échanger l'élément $N-1$ du tableau avec son élément n ;

$\rightarrow$ refaire ces deux dernières opérations pour $N \mapsto N-1$ ...jusqu'à $N = 1$ (arrêt de récursion).

Des permutations aléatoires

```
void permutation( int perm[], int N )
{
  if ( N > 1 )
  {
    echange( perm[ random( N ) ], perm[ N - 1 ] );
    permutation( perm, N - 1 );
  }
}

int main()
{
  const int N = 20;
  int perm[ N ];
  for ( int i = 0; i < N; i++ )
    perm[ i ] = i;
  permutation( perm, N );
  for ( int i = 0; i < N; i++ )
    cout << perm[ i ] << " ";
  cout << endl;
}
```

Des permutations aléatoires

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int random( int N )
{ ... }

void echange( int &i, int &j )
{ ... }

void permutation( int perm[], int N )
{
  if ( N > 1 )
  {
    echange( perm[ random( N ) ], perm[ N - 1 ] );
    permutation( perm, N - 1 );
  }
}

int main()
{
  srand( time(0) );
  const int N = 20;
  int perm[ N ];
  for ( int i = 0; i < N; i++ )
    perm[ i ] = i;
  permutation( perm, N );
  for ( int i = 0; i < N; i++ )
    cout << perm[ i ] << " ";
  cout << endl;
}
```

Une routine de tri

```
void tri( double val[], int perm[], int N )
{
  for ( int i = 0; i < N - 1; i++ )
    if ( val[ perm[ i ] ] > val[ perm[ N - 1 ] ] )
      echange( perm[ i ], perm[ N - 1 ] );
  if ( N > 1 )
    tri( val, perm, N - 1 );
}
```

$\longrightarrow$ ordonner les valeurs du tableau `val`

$\longrightarrow$ accès aux valeurs originales :

`val[ 0 ], val[ 1 ], ...`

$\longrightarrow$ accès aux valeurs triées :

`val[ perm[ 0 ] ] < val[ perm[ 1 ] ] < ...`

Une routine de tri

```
#include <iostream>
#include <cmath>
#using namespace std;

void echange( int &i, int &j )
{
  ...
}

void tri( double val[], int perm[], int N )
{
  for ( int i = 0; i < N - 1; i++ )
    if ( val[ perm[ i ] ] > val[ perm[ N - 1 ] ] )
      echange( perm[ i ], perm[ N - 1 ] );
  if ( N > 1 )
    tri( val, perm, N - 1 );
}

int main()
{
  const int N = 5;
  int perm[ N ];
  for ( int i = 0; i < N; i++ )
    perm[ i ] = i;
  double val[ N ];
  for ( int i = 0; i < N; i++ )
    val[ i ] = sin( i );
  tri( val, perm, N );
  for ( int i = 0; i < N; i++ )
    cout << perm[ i ] << " " << val[ perm[ i ] ] << endl;
}
```

Exécution :

```
4 -0.756802
0 0
3 0.14112
1 0.841471
2 0.909297
```

Des itérations multiples

$\longrightarrow$ Calculer une intégrale dans un espace à N dimensions

$$I = \int dr_1 \cdots \int dr_N f(r_1, \ldots, r_N)$$

Des itérations multiples

$\longrightarrow$ Calculer une intégrale dans un espace à N dimensions par une somme de Riemann

$$I = \sum_{k_1=-K}^{K} \cdots \sum_{k_N=-K}^{K} f(k_1, \ldots, k_N)$$

. . . où N et K pourront être facilement modifiés par l'utilisateur :

```
const int N = 13;
int K = 100;
double integral = 0;
int k[ N ];
for ( k[ 0 ] = - K; k[ 0 ] <= K; k[ 0 ]++ )
for ( k[ 1 ] = - K; k[ 1 ] <= K; k[ 1 ]++ )
...
for ( k[ N - 1 ] = - K; k[ N - 1 ] <= K; k[ N - 1 ]++ )
  integral += f( k );
```

ça ne marchera pas comme ça . . .

Des itérations multiples

$\longrightarrow$ Calculer une intégrale dans un espace à N dimensions par une somme de Riemann

$$I = \sum_{k_1=-K}^{K} \cdots \sum_{k_N=-K}^{K} f(k_1, \ldots, k_N)$$

$\longrightarrow$ utiliser la récursion :

```
double integrale( int k[], int N, int n, int K )
{
  if ( n == 0 )
    return ( f( k, N ) );
  double integr = 0;
  for ( k[ n - 1 ] = -K; k[ n - 1 ] <= K; k[ n - 1 ]++ )
    integr += integrale( k, N, n - 1, K );
  return integr;
}
```

$\longrightarrow$ appel de la fonction `integrale` avec $n = N$

Des itérations multiples

Exemple :

$$
\begin{aligned}
(10\sqrt{\pi})^N &= \int_{-\infty}^{\infty} dr_1 \cdots \int_{-\infty}^{\infty} dr_N \, e^{-(r_1^2+\ldots+r_N^2)/100} \\
&\simeq \sum_{k_1=-K}^{K} \cdots \sum_{k_N=-K}^{K} e^{-(k_1^2+\ldots+k_N^2)/100}
\end{aligned}
$$

```
double f( int k[], int N )
{
  double x = 0;
  for ( int n = 0; n < N; n++ )
    x += k[ n ] * k[ n ];
  return ( exp( - x / 100 ) );
}
```

Des itérations multiples

```
#include <iostream>
#include <cmath>
using namespace std;

double f( int k[], int N )
{
  double x = 0;
  for ( int n = 0; n < N; n++ )
    x += k[ n ] * k[ n ];
  return ( exp( - x / 100 ) );
}

double integrale( int k[], int N, int n, int K )
{
  if ( n == 0 )
    return ( f( k, N ) );
  double integr = 0;
  for ( k[ n - 1 ] = -K; k[ n - 1 ] <= K; k[ n - 1 ]++ )
    integr += integrale( k, N, n - 1, K );
  return integr;
}

int main()
{
  const int N = 3,
  int K = 100;
  int k[ N ];
  cout << integrale( k, N, N, K ) << endl;
}
```

Exécution :

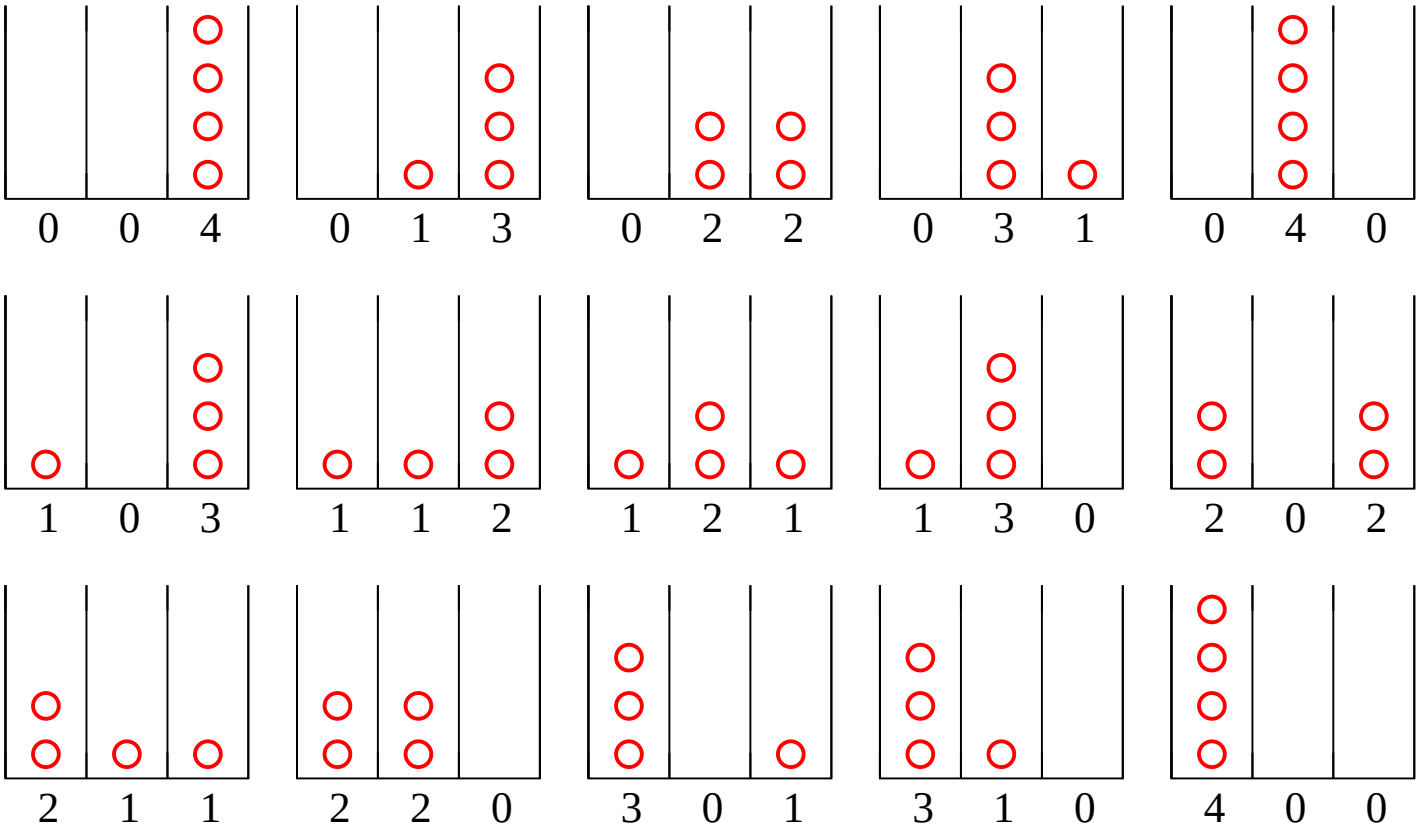
5568.33

$= (10\sqrt{\pi})^3$

Partitions d'un ensemble

$\longrightarrow$ Lister toutes les possibilités de distribuer N objets (indiscernables) dans L cases

Exemple pour $N = 4$ et $L = 3$:



Partitions d'un ensemble

$\longrightarrow$ Lister toutes les possibilités de distribuer N objets (indiscernables) dans L cases

Stratégie générale :

$\rightarrow$ itérer sur toutes le possibilités de mettre $n = 0, 1, \ldots, N$ objets dans la première case

$\rightarrow$ pour chaque n, appliquer cette stratégie à la case suivante pour les $N - n$ objets restants

$\rightarrow$ fin de récursion à la dernière case : y mettre les objets restants et afficher les populations de toutes les cases

Partitions d'un ensemble

```
void ds( int p[], int N, int L, int l )
{
  if ( l < L - 1 )
    for ( int n = 0; n <= N; n++ )
    {
      p[ l ] = n;
      ds( p, N - n, L, l + 1 );
    }
  else
  {
    p[ l ] = N;
    for ( int j = 0; j < L; j++ )
      cout << p[ j ] << " ";
    cout << endl;
  }
}
```

$\longrightarrow$ à appeler avec $l = 0$
dans le dernier argument

Partitions d'un ensemble

```
#include <iostream>
using namespace std;

void ds( int p[], int N, int L, int l )
{
  if ( l < L - 1 )
    for ( int n = 0; n <= N; n++ )
    {
      p[ l ] = n;
      ds( p, N - n, L, l + 1 );
    }
  else
  {
    p[ l ] = N;
    for ( int j = 0; j < L; j++ )
      cout << p[ j ] << " ";
    cout << endl;
  }
}

int main()
{
  const int L = 3;
  int N = 4;
  int pop[ L ];
  ds( pop, N, L, 0 );
}
```

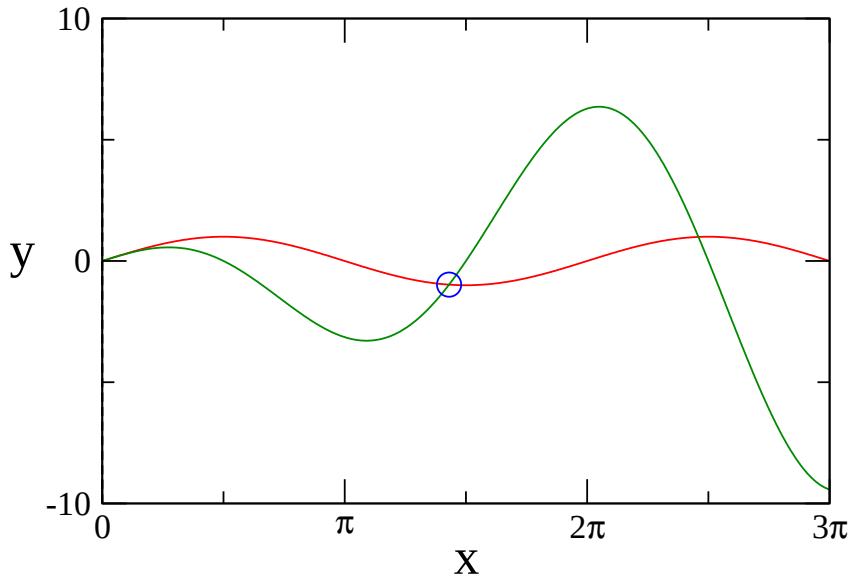
Exécution :

```
0 0 4
0 1 3
0 2 2
0 3 1
0 4 0
1 0 3
1 1 2
1 2 1
1 3 0
2 0 2
2 1 1
2 2 0
3 0 1
3 1 0
4 0 0
```

Recherche des zéros

Exemple :
Déterminer la plus petite solution positive de l'équation

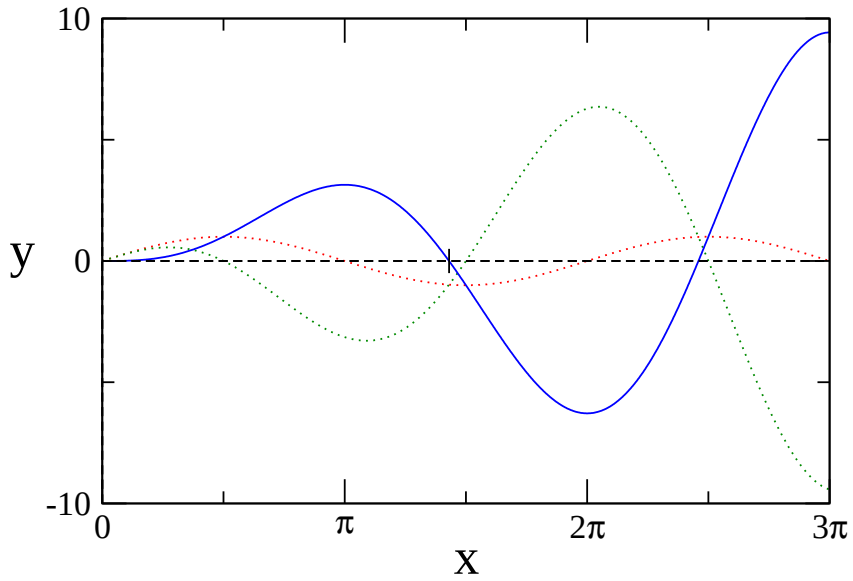
$$\sin(x) = x\cos(x)$$



Recherche des zéros

Exemple :
Déterminer la plus petite solution positive de l'équation

$$\sin(x) = x\cos(x)$$



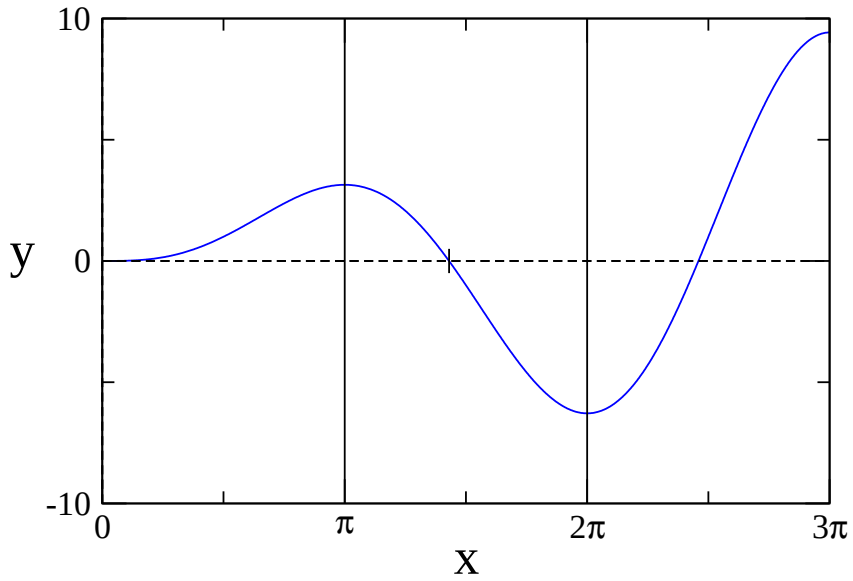
$\longrightarrow$ Déterminer le plus petit zéro positif de la fonction

$$f(x) = \sin(x) - x\cos(x)$$

Recherche des zéros

Exemple :
Déterminer la plus petite solution positive de l'équation

$$\sin(x) = x\cos(x)$$



$\longrightarrow$ Déterminer le plus petit zéro positif de la fonction

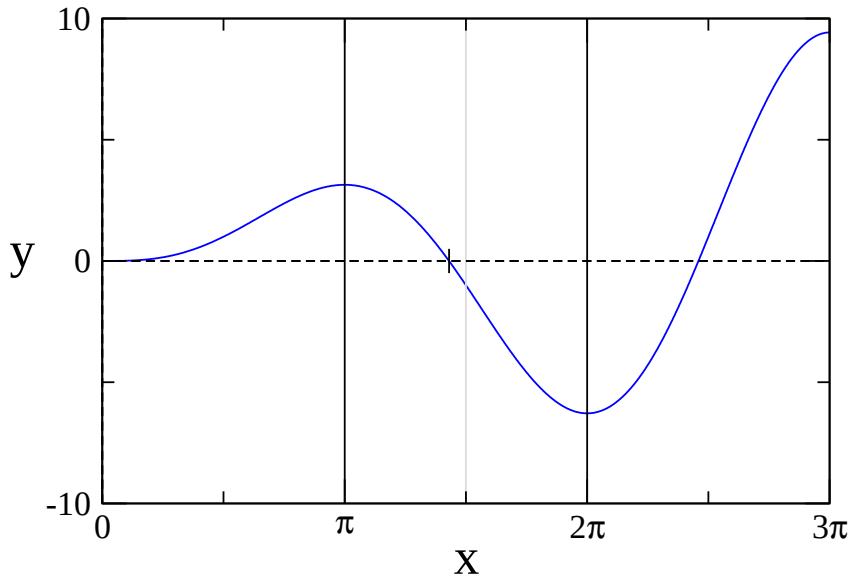
$$f(x) = \sin(x) - x\cos(x)$$

par une recherche binaire

Recherche des zéros

Exemple :
Déterminer la plus petite solution positive de l'équation

$$\sin(x) = x\cos(x)$$



$\longrightarrow$ Déterminer le plus petit zéro positif de la fonction

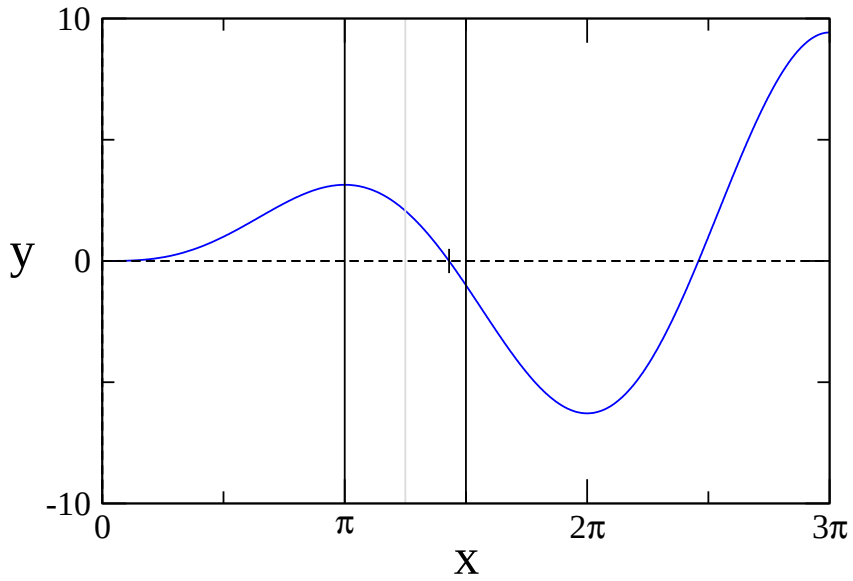
$$f(x) = \sin(x) - x\cos(x)$$

par une recherche binaire

Recherche des zéros

Exemple :
Déterminer la plus petite solution positive de l'équation

$$\sin(x) = x\cos(x)$$



$\longrightarrow$ Déterminer le plus petit zéro positif de la fonction

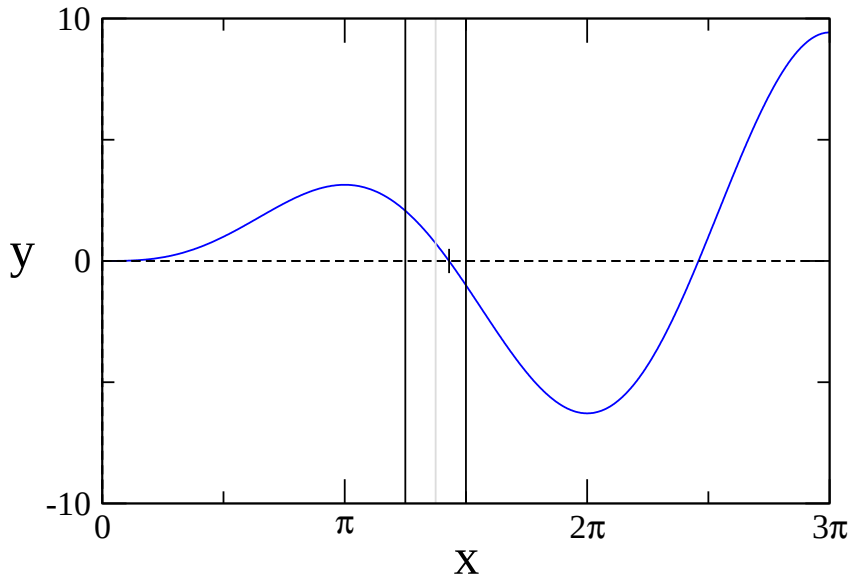
$$f(x) = \sin(x) - x\cos(x)$$

par une recherche binaire

Recherche des zéros

Exemple :
Déterminer la plus petite solution positive de l'équation

$$\sin(x) = x\cos(x)$$



$\longrightarrow$ Déterminer le plus petit zéro positif de la fonction

$$f(x) = \sin(x) - x\cos(x)$$

par une recherche binaire

Recherche des zéros

```
double zero( double a, double fa, double b, double fb,
             double eps )
{
  double x = ( a + b ) / 2;
  double fx = fonction( x );
  if ( ( fabs( x - a ) < eps ) || ( fabs( fx ) < eps ) )
    return x;
  if ( fa * fx < 0 )
    return ( zero( a, fa, x, fx, eps ) );
  return ( zero( x, fx, b, fb, eps ) );
}
```

⟶ à appeler avec `fa = fonction( a )` et `fb = fonction( b )`

⟶ `eps` = précision numérique du calcul

Recherche des zéros

```
#include <iostream>
#include <cmath>
using namespace std;

double fonction( double x )
{
  return ( sin( x ) - x * cos( x ) );
}

double zero( double a, double fa, double b, double fb, double eps )
{
  double x = ( a + b ) / 2;
  double fx = fonction( x );
  if ( ( fabs( x - a ) < eps ) || ( fabs( fx ) < eps ) )
    return x;
  if ( fa * fx < 0 )
    return ( zero( a, fa, x, fx, eps ) );
  return ( zero( x, fx, b, fb, eps ) );
}

int main()
{
  double a = M_PI;
  double b = 2 * M_PI;
  cout << zero( a, fonction( a ), b, fonction( b ), 1e-10 ) << endl;
}
```

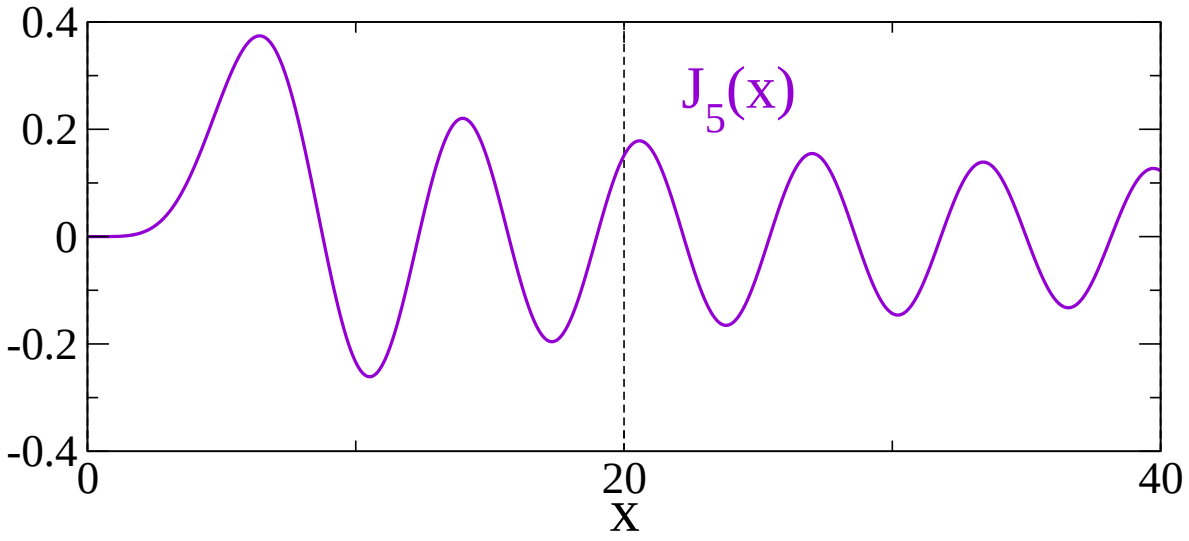
Exécution :

```
4.49341
```

Intégration numérique

Exemple : Calcul numérique de la fonction de Bessel

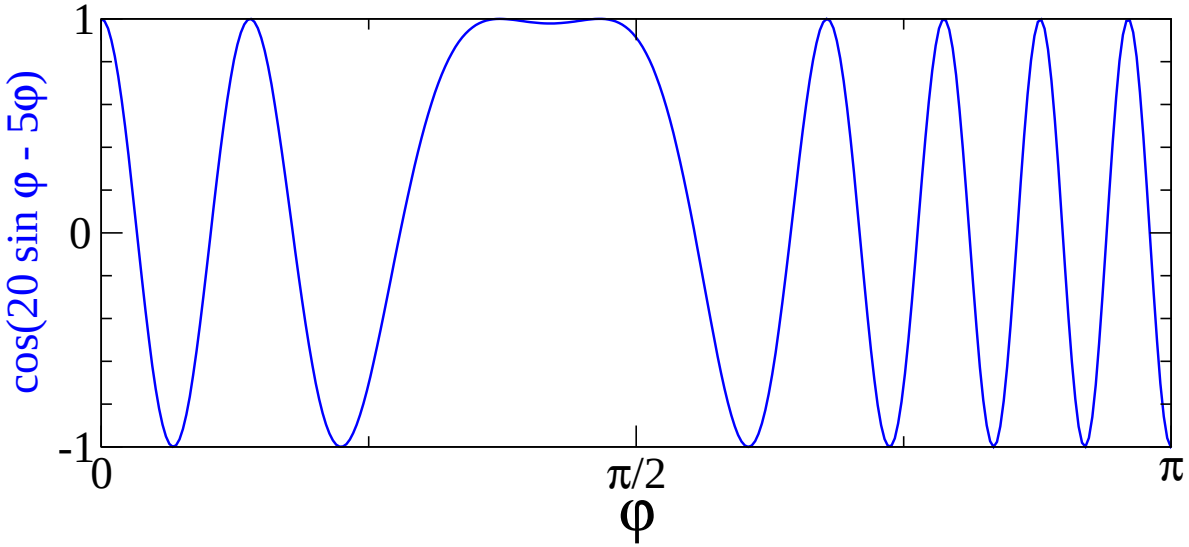
$$J_n(x) = \frac{1}{\pi} \int_0^{\pi} \cos(x \sin \varphi - n\varphi) d\varphi$$ p.ex. pour $n = 5$ et $x = 20$



Intégration numérique

Exemple : Calcul numérique de la fonction de Bessel

$$J_n(x) = \frac{1}{\pi} \int_0^{\pi} \cos(x \sin \varphi - n\varphi) d\varphi$$ p.ex. pour $n = 5$ et $x = 20$

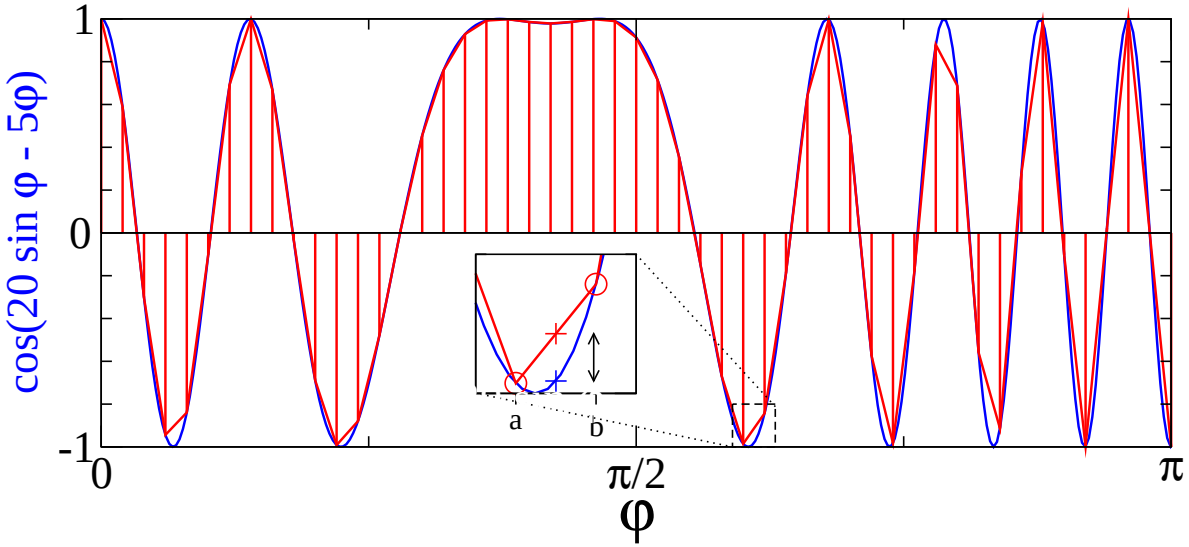


. . . avec la méthode des trapèzes

Intégration numérique

Exemple : Calcul numérique de la fonction de Bessel

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin\varphi - n\varphi)d\varphi \quad \text{p.ex. pour } n = 5 \text{ et } x = 20$$



. . . avec la méthode des trapèzes

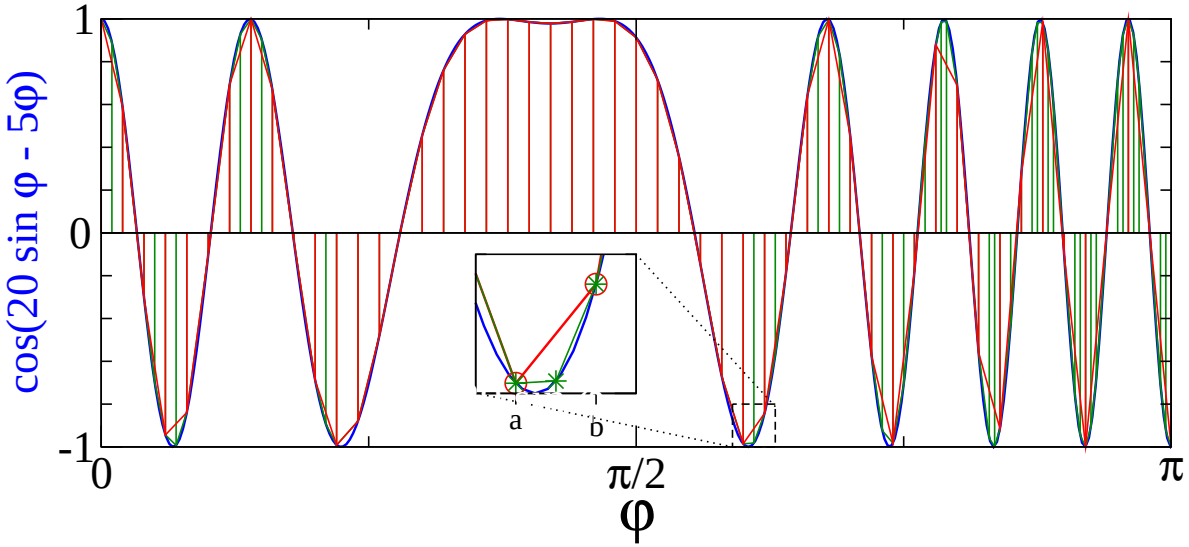
$\longrightarrow$ raffiner les intervalles là où la courbure est importante :

$\left| f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2} \right| > \epsilon$ pour un seuil de précision $\epsilon > 0$

Intégration numérique

Exemple : Calcul numérique de la fonction de Bessel

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin\varphi - n\varphi)d\varphi \quad \text{p.ex. pour } n = 5 \text{ et } x = 20$$



. . . avec la méthode des trapèzes

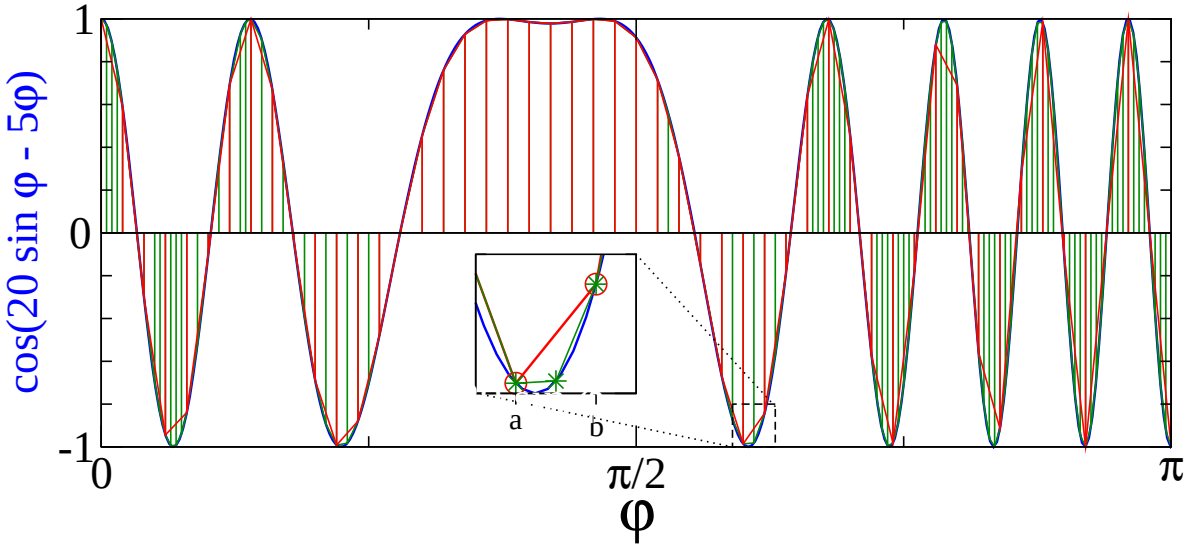
$\longrightarrow$ raffiner les intervalles là où la courbure est importante :

$\left| f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2} \right| > \epsilon$ pour un seuil de précision $\epsilon = 0.05$

Intégration numérique

Exemple : Calcul numérique de la fonction de Bessel

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin\varphi - n\varphi)d\varphi \quad \text{p.ex. pour } n = 5 \text{ et } x = 20$$



. . . avec la méthode des trapèzes

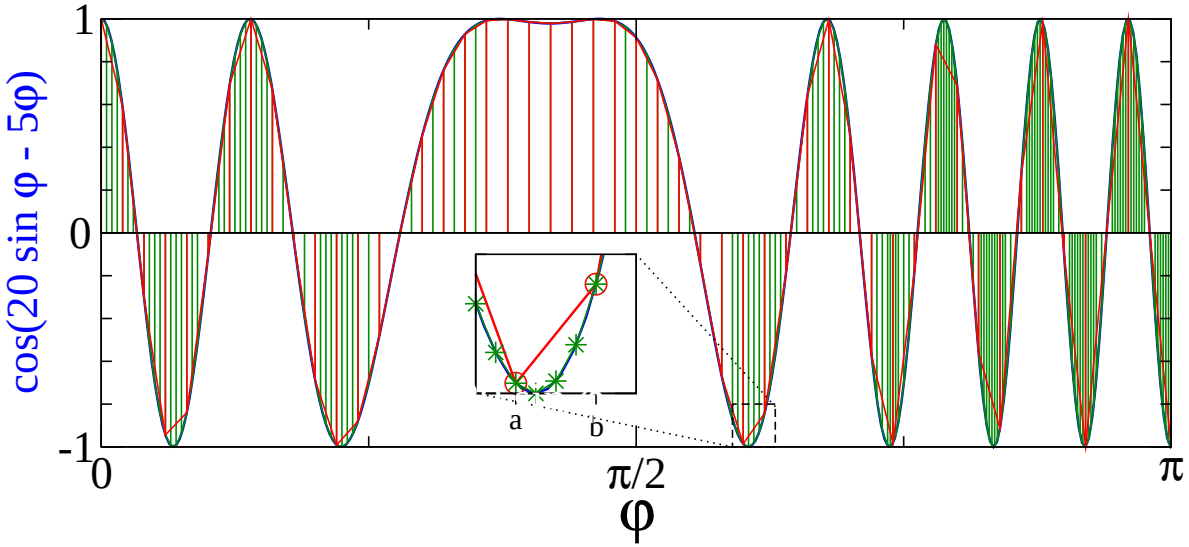
$\longrightarrow$ raffiner les intervalles là où la courbure est importante :

$\left| f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2} \right| > \epsilon$ pour un seuil de précision $\epsilon = 0.02$

Intégration numérique

Exemple : Calcul numérique de la fonction de Bessel

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin\varphi - n\varphi)d\varphi \quad \text{p.ex. pour } n = 5 \text{ et } x = 20$$



. . . avec la méthode des trapèzes

$\longrightarrow$ raffiner les intervalles là où la courbure est importante :

$\left| f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2} \right| > \epsilon$ pour un seuil de précision $\epsilon = 0.01$

Intégration numérique

```
double integ( double a, double fa, double b, double fb,
              double eps )
{
  double x = ( a + b ) / 2;
  double f = ( fa + fb ) / 2;
  double fx = fonction( x );
  if ( fabs( f - fx ) < eps )
    return ( f * ( b - a ) );
  return ( integ( a, fa, x, fx, eps ) +
           integ( x, fx, b, fb, eps ) );
}
```

$\longrightarrow$ à appeler avec `fa = fonction( a )` et `fb = fonction( b )`

$\longrightarrow$ `eps` = précision numérique du calcul

Intégration numérique

```
#include <iostream>
#include <cmath>
using namespace std;

double fonction( double x )
{
  return ( cos( 20 * sin( x ) - 5 * x ) );
}

double integ( double a, double fa, double b, double fb, double eps )
{
  double x = ( a + b ) / 2;
  double f = ( fa + fb ) / 2;
  double fx = fonction( x );
  if ( fabs( f - fx ) < eps )
    return ( f * ( b - a ) );
  return ( integ( a, fa, x, fx, eps ) + integ( x, fx, b, fb, eps ) );
}

int main()
{
  double eps = 0.001;
  double a = 0;
  double b = M_PI;
  cout << integ( a, fonction( a ), b, fonction( b ), eps ) / M_PI << endl;
}
```

Exécution : `0.151103` $\simeq 0.151169768 = J_5(20)$

$\longrightarrow$ obtenu avec 1000 évaluations de la fonction

Résolution des équations différentielles

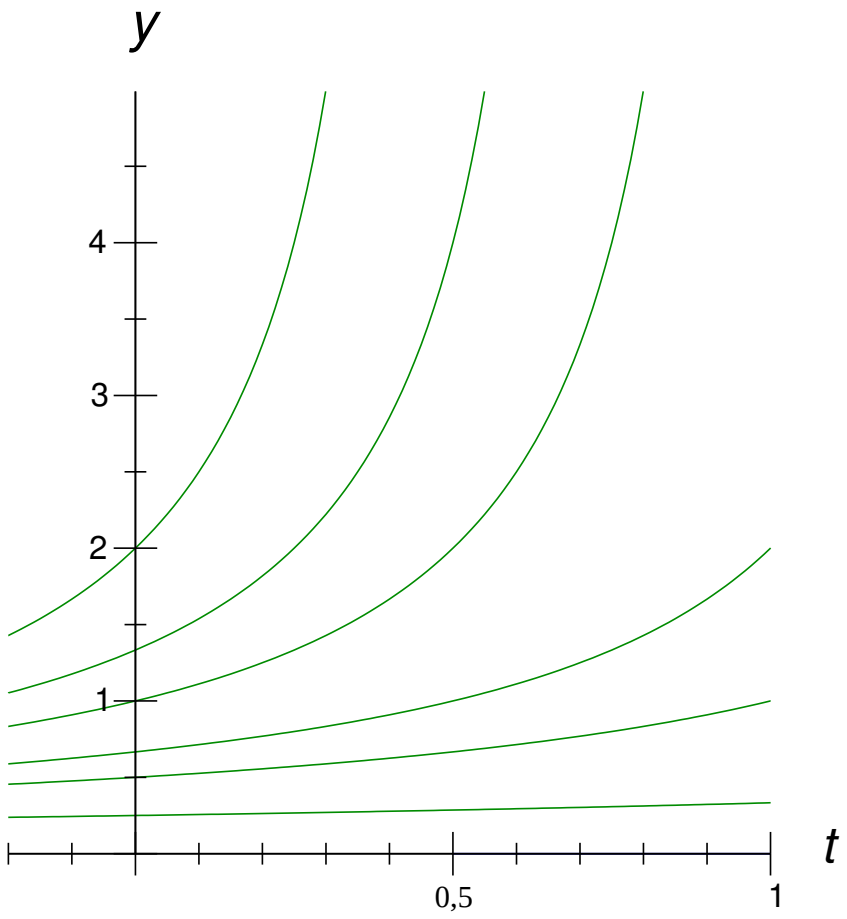
$$\dot{y}(t) = f(y(t), t)$$

pour la condition initiale

$$y(0) = y_0$$

Exemple: $f(y, t) = y^2$

$$\Rightarrow y(t) = \frac{y_0}{1 - y_0 t}$$



Résolution des équations différentielles

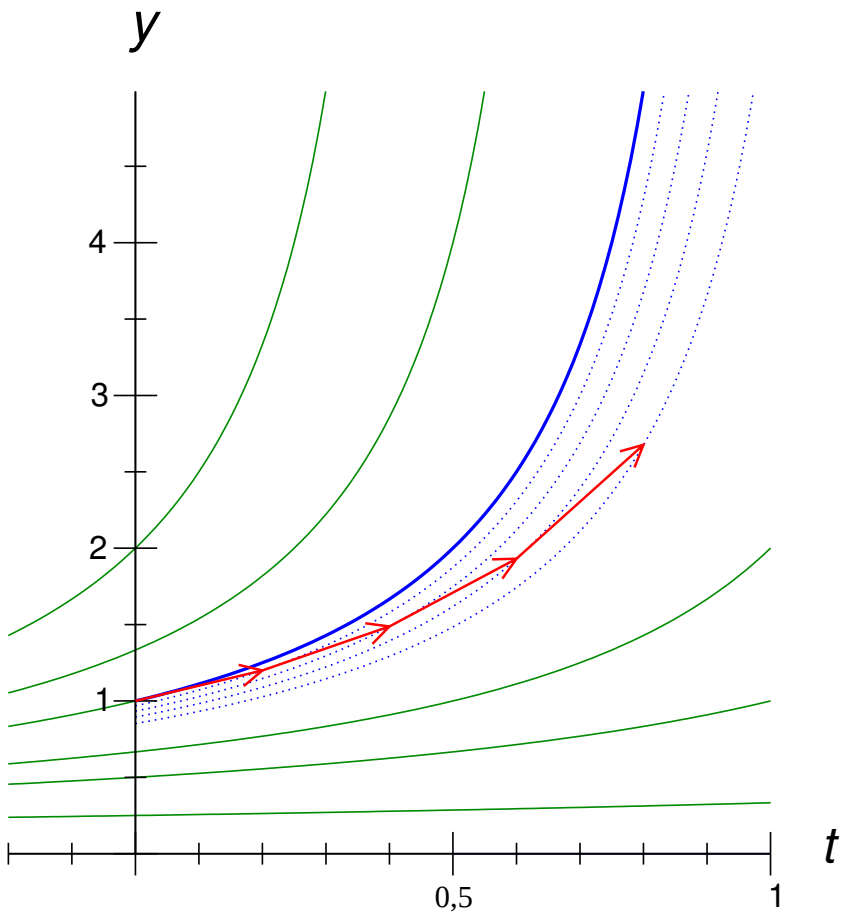
$$\dot{y}(t) = f(y(t), t)$$

pour la condition initiale

$$y(0) = y_0$$

Méthode d'Euler :

$$\begin{aligned} y(t+\delta) &\simeq y(t) + \delta\, \dot{y}(t) \\ &= y(t) + \delta\, f(y(t), t) \end{aligned}$$



$\delta = 0.2$

Résolution des équations différentielles

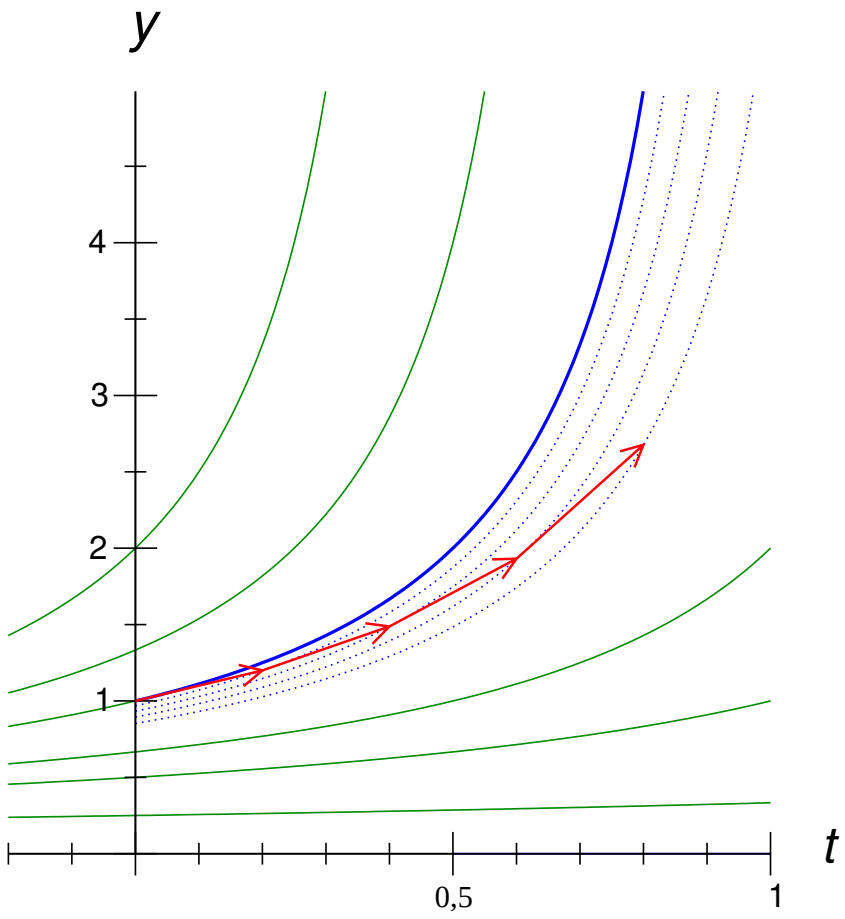
$$\dot{y}(t) = f(y(t), t)$$

pour la condition initiale

$$y(0) = y_0$$

Méthode d'Euler :

$$\begin{aligned} y(t+\delta) &\simeq y(t) + \delta\, \dot{y}(t) \\ &= y(t) + \delta\, f(y(t), t) \end{aligned}$$



$\longrightarrow$ réduire le pas δ lorsque la courbure de la solution $\ddot{y}(t) \propto f(y(t+\delta), t+\delta) - f(y(t), t)$ devient trop importante

Résolution des équations différentielles

La méthode d'Euler ...

```
double derivee( double y, double t )
{
  return ( y * y );
}

void propa( double &y, double t, double dt, double eps )
{
  double dy = derivee( y, t );
  y += dt * dy;
}
```

Résolution des équations différentielles

La méthode d'Euler . . . avec adapation du pas si besoin :

```
double derivee( double y, double t )
{
  return ( y * y );
}

void propa( double &y, double t, double dt, double eps )
{
  double dy = derivee( y, t );
  double y1 = y + dt * dy;
  double dy1 = derivee( y1, t + dt );
  if ( fabs( dy1 - dy ) > eps )
  {
    propa( y, t, dt / 2, eps );
    propa( y, t + dt / 2, dt / 2, eps );
  }
  else
    y = y1;
}
```

Résolution des équations différentielles

La méthode d'Euler . . . avec adapation du pas si besoin :

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

double derivee( double y, double t )
{
  return ( y * y );
}

void propa( double &y, double t, double dt, double eps )
{
  ...
}

int main()
{
  double y = 1;
  double dt = 0.2;
  int Nt = 4;
  double eps = 0.1;
  ofstream out( "solution.txt" );
  for ( int i = 0; i < Nt; i++ )
  {
    propa( y, i * dt, dt, eps );
    out << ( i + 1 ) * dt << " " << y << endl;
  }
}
```

Résolution des équations différentielles

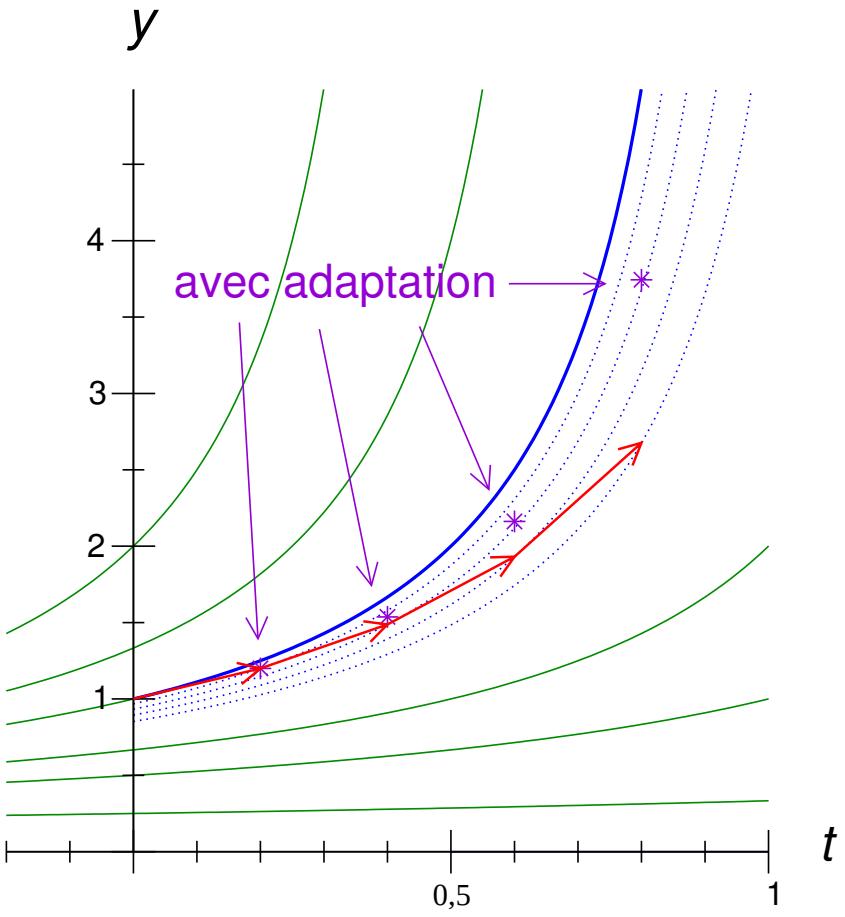
$$\dot{y}(t) = y^2(t)$$

pour la condition initiale

$$y(0) = 1$$

Calcul avec `dt = 0.2` et

`eps = 0.5`



Résolution des équations différentielles

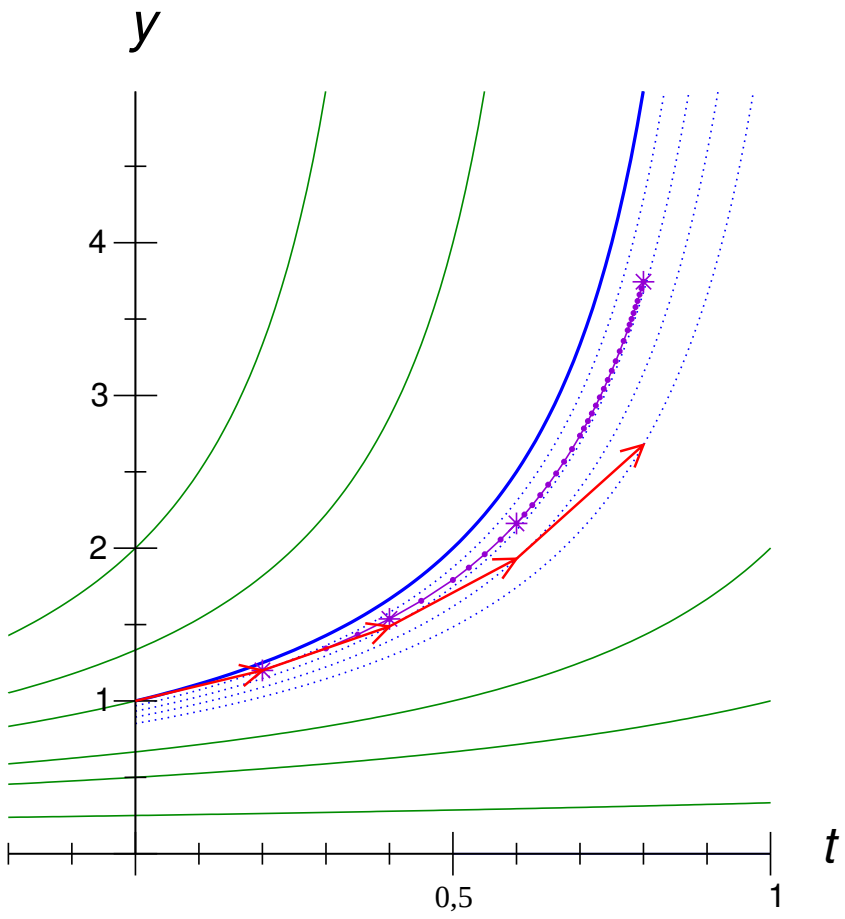
$$\dot{y}(t) = y^2(t)$$

pour la condition initiale

$$y(0) = 1$$

Calcul avec `dt = 0.2` et

`eps = 0.5`



$\longrightarrow$ 38 pas de propagation intermédiaires
(notamment là où la courbure est la plus importante)

Résolution des équations différentielles

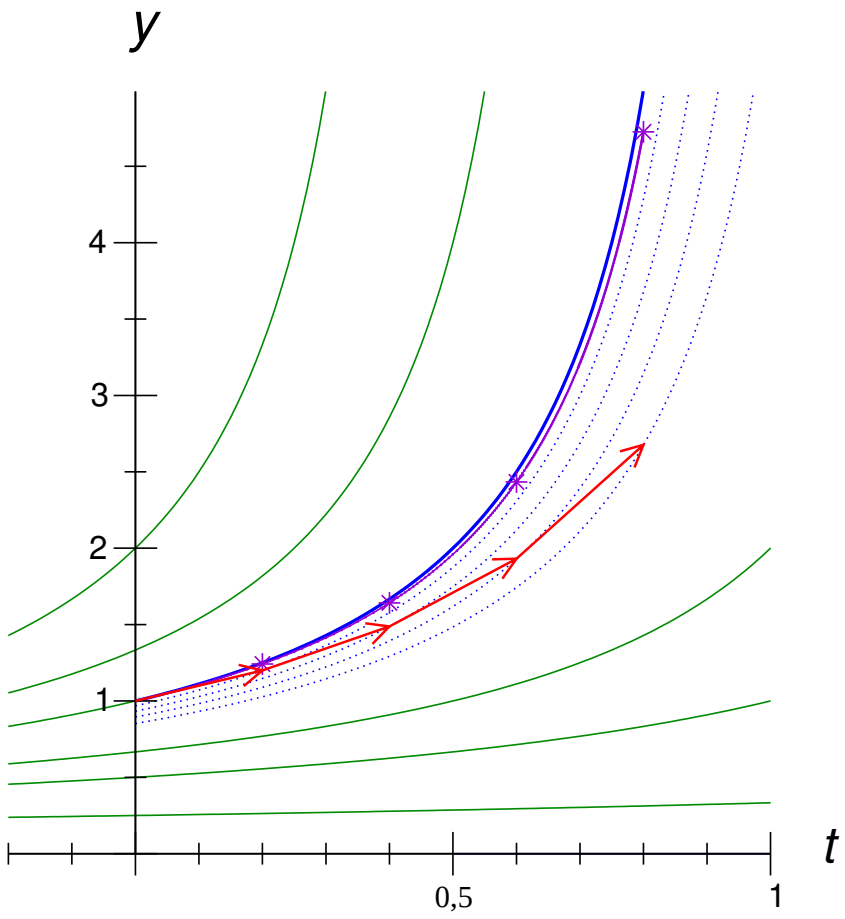
$$\dot{y}(t) = y^2(t)$$

pour la condition initiale

$$y(0) = 1$$

Calcul avec `dt = 0.2` et

`eps = 0.1`



$\longrightarrow$ 308 pas de propagation intermédiaires

Résolution des équations différentielles

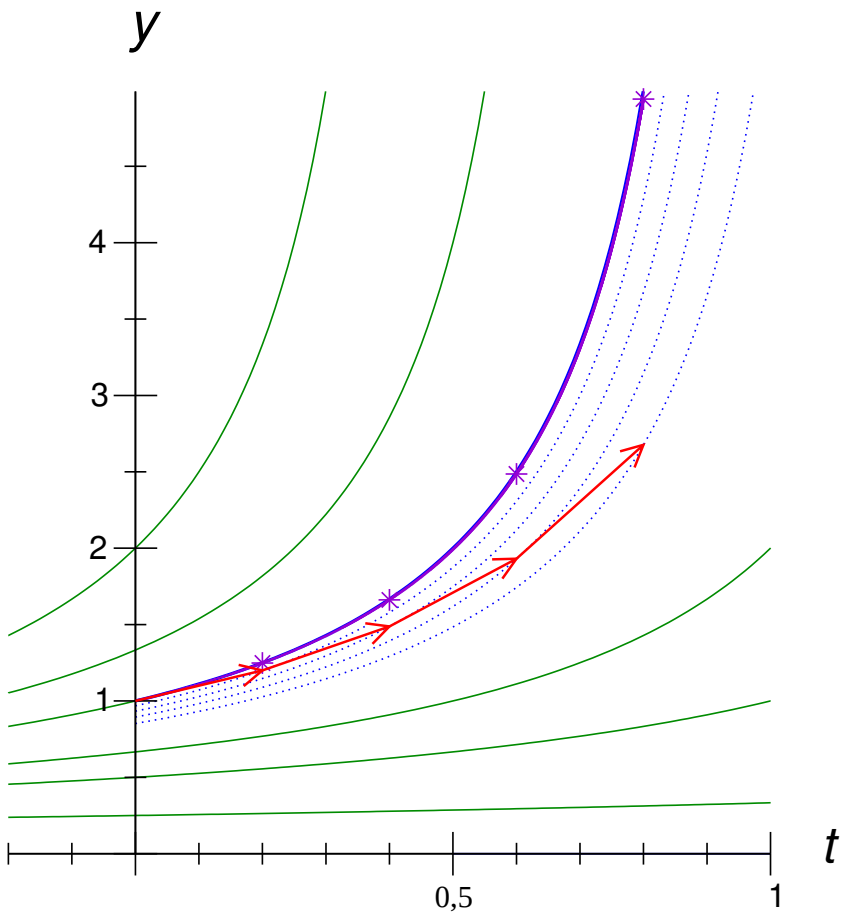
$$\dot{y}(t) = y^2(t)$$

pour la condition initiale

$$y(0) = 1$$

Calcul avec `dt = 0.2` et

`eps = 0.02`



$\longrightarrow$ 1695 pas de propagation intermédiaires

Conclusion

La récursion peut être un outil puissant permettant de résoudre certaines tâches de programmation assez facilement . . .

. . . mais elle n'est pas forcément la solution miracle pour tous les problèmes de programmation et peut s'avérer dangereuse pour des calculs numériques qui sont très exigeants en besoin de mémoire et/ou temps d'exécution.

Calcul du déterminant d'une matrice $N \times N$ par récursion :

Formule de Laplace : $\det A = \sum_{i=1}^{N} a_{i1} \det(A'_{i1})$

Conclusion

La récursion peut être un outil puissant permettant de résoudre certaines tâches de programmation assez facilement . . .

. . . mais elle n'est pas forcément la solution miracle pour tous les problèmes de programmation et peut s'avérer dangereuse pour des calculs numériques qui sont très exigeants en besoin de mémoire et/ou temps d'exécution.

Calcul du déterminant d'une matrice $N \times N$ par récursion :

$$\longrightarrow N! \sum_{n=2}^{N} \frac{2n-1}{n!} \simeq eN! \text{ } \textit{floating point operations}$$

(= opérations arithmétiques avec des nombres flottants)

Pour une matrice 30×30, le meilleur ordinateur du monde (capable d'effectuer 10^{17} *floating point operations* par seconde) mettrait plus que 200 millions ans pour finir ce calcul.