Hornerovo schéma

- je algoritmus výpočtu hodnoty polynomu $P(\mathrm{x})$ v bodě x_0

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

  – eliminuje výpočet i-té mocniny převodem na postupné násobení

Algoritmus výpočtu

Vstup:

- polynom stupně n:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- bod x_0

Výstup:

- hodnota polynomu v bodě x_0 $P(x_0)$

Hornerovo schéma

- princip

$$P(x) = 3x^3 + 2x^2 - 2x + 5 = (3x^2 + 2x - 2)x + 5 =$$

$$= ((3x + 2)x - 2)x + 5$$

- hodnotu polynomu v bodě $x_0=2$ $P(2)$ spočítáme:

$$3 \cdot 2 + 2 = 8$$

$$8 \cdot 2 - 2 = 14$$

$$14 \cdot 2 + 5 = 33$$

Hornerovo schéma

$$P(x) = 3x^3 + 2x^2 - 2x + 5$$

$$P(2) = ?$$

3	2	-2	5
3*2	+2 =8		
	8*2	-2=14	
		14*2	+5 = 33

Algoritmus výpočtu

1. Polož $i = n$, $s = a_n$
2. Je-li $i = 0$, konec
3. $s = s \cdot x_0 + a_{\mathrm{i-1}}$
4. $i = i - 1$
5. Jdi na krok 2

- v s je hodnota polynomu v bodě x_0

Příklad

$$P(x) = 5x^5 - 3x^4 + 2x^3 + 7x^2 - x + 2$$

Vypočítejte hodnotu polynomu v bodě

$$x_0 = 3$$

Eratostenovo síto

- algoritmus určí všechna prvočísla menší než zadané n
- vychází z následující úvahy:
  - prvočíslo je dělitelné pouze sebou samým nebo jedničkou, není tedy násobkem žádného menšího přirozeného čísla
  - máme-li např. vypsat všechna prvočísla od 1 do 100, napíšeme si všechna čísla do řádku a postupně vyškrtáme násobky 2, 3, ..., n/2; zbylá čísla jsou prvočísla

Eratostenovo síto

2 3 4 5 6 7 8 9 10

- vyškrtáme násobky 2:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~

- vyškrtáme násobky 3:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~

- protože je 4 již škrtnutá, násobky 4 škrtat nemusíme, již jsou vyškrtnuté
- vyškrtáme násobky 5:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~

- prvočísla jsou: 2, 3, 5, 7

Eratostenovo síto

- implementace
  - pomocí pole přirozených čísel, resp. booleovských hodnot
    - prvek pole A[i] má hodnotu 1, je-li číslo i prvočíslo, jinak má hodnotu 0
  - na počátku inicializujeme prvky na hodnotu 1 (předpokládáme, že všechna čísla jsou prvočísla)
  - v cyklu přiřazujeme prvkům pole s indexy odpovídající násobkům čísel 2, 3 , ... , n/2 hodnotu 0
    - algoritmus zrychlit tím, že je-li A[j]=0, již nenulujeme prvky s indexy odpovídající násobkům čísla i (jsou již vynulovány)
  - vypíšeme indexy prvků pole, kde A[j]=1

STL

- vektory a matice reprezentujeme statickým nebo dynamickým polem
  - problémy s velikostí (přetečením), neznáme-li předem velikost dat, nutnost realokace
- v C++ existuje knihovna STL (standard template llibrary), která implementuje jako šablony vektor a ostatní ADT (fronta, zásobník)

STL

- STL je knihovna šablon (generických tříd) implementující zmíněné datové typy
  - fronta, zásobník, seznam, vektor
- je součástí normy C++

- v STL jsou definovány tři skupiny generických tříd:
  - **kontejnery**
    - představují vlastní ADT – vektor, zásobník, fronta, string
  - **iterátory**
    - třídy, jejichž metody umožňují procházet kontejnery, např. nastavit se na první nebo poslední prvek, postoupit na další atd.
  - **algoritmy**
    - provádějí inicializaci, třídění, vyhledávání nad kontejnery

- každý kontejner má definován vlastní alokátor
  - alokátor řídí přidělování paměti pro kontejnery
  - standardní alokátor je instance třídy *allocator*, která je definována v STL
  - uživatel si může definovat vlastní alokátory

Kontejnery

Kontejner	Popis	Hlav. soubor
bitset	sada bitů	<bitset>
deque	oboustr. zakončená fronta	<deque>
list	lineární seznam	<list>
map	ukládá páry klíč/hodnota	<map>
multimap	ukládá páry klíč/hodnoty	<multimap>
multiset	sada (množina)	<multiset>
priority_queue	prioritní fronta	<queue>
queue	fronta	<queue>
set	sada, každý prvek jednozn.	<set>
stack	zásobník	<stack>
vector	vektor (dynamické pole)	<vector>

Iterátory

- objekty, jejichž metody slouží k „pohybu“ po prvcích kontejnerů
  - umožňují např. stejným způsobem procházet vektor, frontu od začátku do konce, od konce k začátku atd.
  - pracuje se s nimi jako s ukazateli (mají přetížené operátory *,++,--) – lze je dekrementovat, inkrementovat,…

- typy iterátorů:
  - dopředný (ForIter)
    - ukládá a získává hodnoty, pohyb vpřed
  - obousměrný (BiIter)
    - ukládá a získává hodnoty, pohyb vpřed i vzad
  - vstupní (InIter)
    - získává, ale neukládá hodnoty, pohyb pouze vpřed
  - výstupní (OutIter)
    - ukládá, ale nezískává hodnoty, pohyb pouze vpřed
  - přímý přístup (RandIter)
    - získává a ukládá hodnoty, přímý přístup k prvkům jako u pole

Poznámky:

- dopředný
  - nemá přetížený operátor `--`
- vstupní
  - nemůže být použit na levé straně přiřazovacího příkazu

Vektor

- deklarace v souboru <vector>:

```
template<class _Ty,class _Ax = allocator<_Ty> >
class vector
```

- konstruktory

```
vector(const _Alloc& _Al)
vector(size_type _Count, const _Ty& _Val)
vector(size_type _Count, const _Ty& _Val, const
    _Alloc& _Al)
```

Příklad metod

`reference front()`

- vrací odkaz na první prvek vektoru (reference je datový typ představující odkaz)
- `typedef T& reference;`

`void clear()`

- odstraní všechny prvku vektoru

`iterator insert(iterator _Where, const _Ty& _Val)`

- vloží prvek Val před (?) prvek označený Where
- insert je 3x přetíženo

- `void insert(iterator _Where, const Ty& val = Ty());`
  - vloží 1 prvek
- `void insert(iterator _Where, size_type n, const Ty& val = Ty())`
  - vloží n kopií prvku
- `template<class InputIterator> void insert(iterator _Where, InputIterator zacatek, InputIterator konec);`
  - vloží prvky z jiného iterátoru

`reference back()`

– vrací referenci na poslední prvek

`void push_back()`

– vloží nový prvek "za" konec

`void pop_back()`

– odebere prvek z konce vektoru

Použití

```
#include <vector>
using namespace std;
void main(void)
{
  vector<int> v;
  // vytvori prazdny vektor
  for(int i=0;i<10;i++)  v.push_back(i);
  for(int i=0;i<10;i++)
    cout << v[i] << ' ';
}
```

Použití

```
include <vector>
using namespace std;
void main(void)
{
  vector<int> v(10);
  // vytvori vektor o 10 polozkach
  for(int i=0;i<10;i++)  v[i] = i;
  for(int i=0;i<10;i++)
    cout << v[i] << ' ';
}
```

Použití

```
include <vector>
using namespace std;
void main(void)
{
  vector<int> v(10,5);
  // nastavi vsechny prvky vektoru na 5
  for(int i=0;i<10;i++)  v[i] = i;
  for(int i=0;i<10;i++)
    cout << v[i] << ' ';
  cout << "\nPocet prvku ve vektoru: ";
  cout << v.size();
}
```

- přístup přes iterátor

```
vector<int>::iterator p=v.begin();
while(p != v.end())
{
  cout << *p << ' ';
  p++;
}
```

Poznámky:

- všechny kontejnery jsou ve jmenném prostoru std
- parametrem <bitset> je počet bitů, nikoliv typ

```
template <size_t N>
class  bitset
{
…
}
```

- každý kontejner má veřejné metody:
  - begin – vrací iterátor na počáteční prvek.
  - empty - vrací true, jestliže je kontejner prázdný.
  - end – vrací iterátor za poslední prvek
  - max_size - vrací maximální možnou velikost kontejneru.
  - size - vrací aktuální velikost kontejneru.
  - swap - zajistí výměnu prvků s jiným kontejnerem

Algoritmy

- funkce jsou definovány v hlavičkovém souboru `algorithm`
- dělení
  - algoritmy nepracující s kontejnery
    - min, max, swap – parametry jsou reference na typ
  - kopírující algoritmy
    - copy, copy_backward – kopírují části kontejnerů určené iterátory

- přesouvací algoritmy
  - remove, remove_if – odstraňují prvky
  - remove_copy, remove_copy_if – kopíruje prvky do jiného kontejneru
- vyhledávací algoritmy
  - find
- a další
  - generování permutací
    - `next_permutation(BidIt f, BidIt l)`

- pěkný přehled je na

  http://www.cplusplus.com/

GSL

- GNU Scientific Library
- numerická knihovna pro programátory v C/C++
  - převažují rysy klasického C
- aktuální verze 1.15 (květen 2011)
  - verze pro Linux i se zdrojovými kódy
  - verze pro MS Windows - 1.8
    - pro překladač GCC

Webové stránky

- hlavní stránka
  - http://www.gnu.org/software/gsl/
- zkompilovaná verze pro MS Windows
  - http://www.cygwin.com

Oblasti využití

- komplexní čísla
- hledání kořenů polynomů
- vektory a matice
- generování permutací
- lineární algebra
- vlastní čísla matic
- FFT (Fast Fourier Transforms)
- speciální generátory náhodných čísel

Oblasti využití

- statistika – histogramy
- integrace metodou Monte Carlo
- simulované žíhání
- řešení diferenciálních rovnic
- interpolace
- Čebyševova aproximace
- hledání kořenů

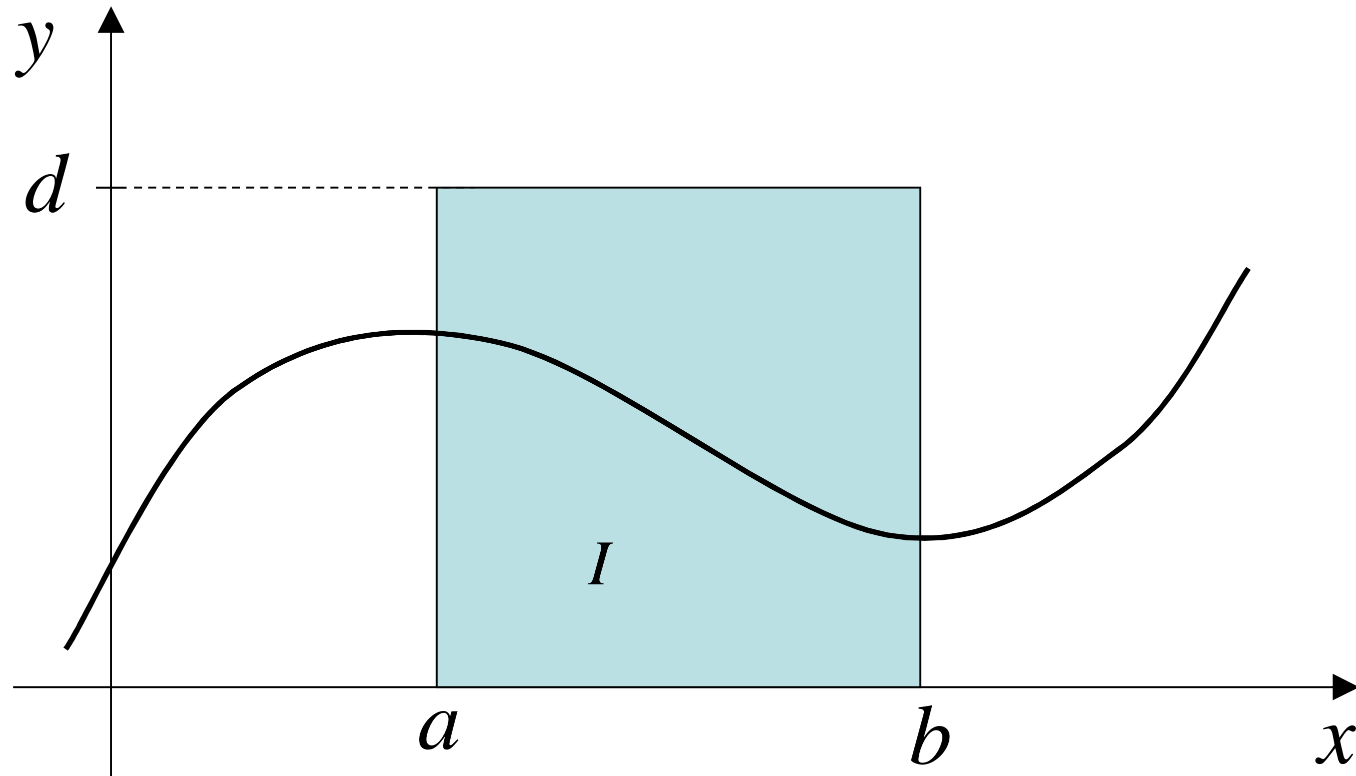
Oblasti využití

- minimalizace metodou nejmenších čtverců
- spline funkce

Integrace metodou Monte Carlo

- stochastická metoda
  - založena na sérii náhodných pokusů

Integrace metodou Monte Carlo



$$\frac{I}{S} = \frac{N_I}{N}$$

$$I = \frac{N_I}{N}(b-a) \cdot d$$

Integrace metodou Monte Carlo

- generuji náhodně s rovnoměrným rozložením body do "modrého obdélníka"
- poměr plochy pod křivkou I ku ploše obdélníka S je roven poměru počtu bodů N_I, které padnou pod křivku, ku celkovému počtu vygenerovaných bodů N

Hlavičkové soubory

- `gsl_math.h`
- `gsl_monte.h`
- `gsl_monte_plain.h`
- `gsl_monte_miser.h`
- `gsl_monte_vegas.h`

Obecný princip práce z knihovnou

1. volání inicializačních a alokačních funkcí
2. použití vlastních funkcí numerickým metod
3. volání dealokačních funkcí

Funkce pro základní metodu

Inicializace

```
gsl_monte_plain_state *gsl_monte_plain_alloc (size_t dim)
```

- parametry
  - `dim` – dimenze funkce, kterou integruji
- návratová hodnota
  - ukazatel na strukturu `gsl_monte_plain_state`

Funkce pro základní metodu

Výpočet

```
int gsl_monte_plain_integrate (gsl_monte_function * f,
const double xl[], const double xu[], size_t dim,
size_t calls, gsl_rng * r, gsl_monte_plain_state * s,
double * result, double * abserr)
```

- parametry
  - `f` – ukazatel na funkci, kterou integruji (speciální struktura knihovny)
  - `xl, xu` – interval, na kterém integruji
  - `dim` – dimenze problému
  - `calls` – počet pokusů
  - `r` – ukazatel na generátor náhodných čísel
  - `s` – ukazatel na strukturu `gsl_monte_plain_state` vytvořenou voláním funkce **`gsl_monte_plain_alloc`**
  - `result` – hodnota integrálu
  - `abserr` – odhad chyby
- návratová hodnota
  - 0 – bez chyby, jinak chyba

Funkce pro základní metodu

Dealokace

- **void gsl_monte_plain_free** (*gsl_monte_plain_state * s*)

- praktická ukázka na cvičení

Integrace jednodimenzionální funkce

- funkce $y = x^2$ na intervalu <0,2>
- definuji funkci

```
double  funkce(double *x, size_t dim, void *params)
{
  return x[0]*x[0];
}
```

Integrace jednodimenzionální funkce

- hlavní program

```
int main (void)
{
  double integral, err;

  double xl[1] = { 0 };
  double xu[1] = { 2 };

  const gsl_rng_type *T;
  gsl_rng *r;
  gsl_monte_function G = { &funkce, 1, 0 };
```

Integrace jednodimenzionální funkce

- hlavní program

```
size_t pocet_volani = 100000;

gsl_rng_env_setup ();
T = gsl_rng_default;
r = gsl_rng_alloc(T);

gsl_monte_plain_state *s = gsl_monte_plain_alloc(1);
gsl_monte_plain_integrate (&G, xl, xu, 1,
            pocet_volani, r, s,&integral, &err);
gsl_monte_plain_free (s);
```

Integrace jednodimenzionální funkce

- hlavní program

```
  cout << "Integral = " << integral << endl;
  cout << "Odhad chyby = " << err << endl;

  return 0;
}
```

Integrace dvouimenzionální funkce

- funkce $z = x^2 + y^2$ na intervalu <0,2> x <0,1>

- definuji funkci

```
double  funkce(double *x, size_t dim, void *params)
{
  return x[0]*x[0]+x[1]*x[1];
}
```

Integrace jednodimenzionální funkce

- hlavní program

```
int main (void)
{
  double integral, err;

  double xl[2] = { 0,0 };
  double xu[2] = { 2,1 };

  const gsl_rng_type *T;
  gsl_rng *r;
  gsl_monte_function G = { &funkce, 2, 0 };
```

Integrace jednodimenzionální funkce

- hlavní program

```
size_t pocet_volani = 100000;

gsl_rng_env_setup ();
T = gsl_rng_default;
r = gsl_rng_alloc(T);

gsl_monte_plain_state *s = gsl_monte_plain_alloc(2);
gsl_monte_plain_integrate (&G, xl, xu, 2,
            pocet_volani, r, s,&integral, &err);
gsl_monte_plain_free (s);
```

Integrace jednodimenzionální funkce

- hlavní program

```
  cout << "Integral = " << integral << endl;
  cout << "Odhad chyby = " << err << endl;

  return 0;
}
```