Obliczenia symboliczne w C/C++

Joanna Hałun
Aneta Mrózek

8 II 2016

Spis treści

1 Wstęp

Projekt polegał na omówieniu możliwości obliczeń symbolicznych w C++ przy użyciu bibilioteki SymbolicC++ i GiNaC.

2 SymbolicC ++

2.1 Wprowadzenie

SymbolicC ++ używa języka programowania C ++ i programowania obiektowego w celu opracowania systemu algebry komputerowej. Programowanie obiektowe jest podejściem do projektowania oprogramowania, które opiera się na klasach, a nie na procedurach. Takie podejście zwiększa modułowość i ukrywanie informacji. Projektowanie zorientowane obiektowo zapewnia wiele korzyści,np. zawiera zarówno dane, jak i funkcje, które działają na tych danych w jednym urządzeniu. Taką jednostkę (abstrakcyjny typ danych) nazywamy klasą. Używamy C ++ jako języka programowania obiektowego z następujących powodów: C ++ umożliwia wprowadzenie abstrakcyjnych typów danych. W ten sposób możemy wprowadzić typy danych wykorzystywanych w systemie algebry komputerowej jako abstrakcyjne typy danych. Język C ++ obsługuje kluczowe pojęcia programowania obiektowego: hermetyzacja, dziedziczenie, polimorfizm (w tym dynamiczne wiązanie) i przeciążenia operatora. Posiada dobre wsparcie dla dynamicznego zarządzania pamięcią i obsługuje zarówno proceduralne jak i obiektowe programowanie. SymbolicC ++ jest darmowym oprogramowaniem wydanym na warunkach GNU GPL. Najnowsza wersja pochodzi z 2010 roku.

2.2 Instalacja

Na początku przystępujemy do pobrania pliku ze strony `http://issc.uj.ac.za/symbolic/symbolic.html`, a następnie rozpakowujemy go i instalujemy. Poniżej kroki instalacji:

1. wget `http://issc.uj.ac.za/symbolic/sources/SymbolicC++3-3.34.tar.gz`
2. mkdir symbolicc++
3. rozpakowujemy pobrany plik

Instalacja tego pakietu przebiega bez najmniejszych problemów.

2.3 Programy

SymbolicC ++ jest używany poprzez dołaczanie pliku nagówkowego lub połączenie z bibiloteką. W celu uruchomienia programu tworzymy Makefila o poniższej zawartości:
GCC=g++-5 // używany kompilator
PROGRAM=symbolic.cpp // nazwa programu do kompilacji
LIB=../headers // lokalizacja głównej biblioteki symbolicc++ (symbolicc++.h)
run:
$(GCC) -I$(LIB) -o main.x $(PROGRAM)
(time ./main.x)

clean: main.x
-rm main.x
Plik ten umieszczamy w katalogu, w którym znajduje się program.

Wszystkie niezbędne klasy i definicje są uzyskiwane przez #include "symbolicc++.h" umieszczonego w pliku źródłowym. Rozpoczynając pisanie programu należy użyć następujących bibliotek i plików nagłówkowych:

- #include <iostream>
- using namespace std;
- #include "symbolicc++.h"

Dwie pierwsze należą do języka programowania C++, a ostatnia jest biblioteką symboliczną.
Przykładowy program wygląda natępująco:

```
#include <iostream>
using namespace std;
#include "symbolicc++.h"
int main(void)
{
Symbolic a("a");
Symbolic b = (a*a*a)+(a);
cout « b « endl;
return 0;
}
```

Po wykonaniu powyższego programu otrzymujemy wyrażenie.
W programie użyto typ danych dla nieokreślnych symboli matematycznych, porzez następujące zdefiniowanie: Symbolic a("a");

SymbolicC ++ przechowuje tak zdefiniowane zmienne jako stringi. Poniżej przedstawiamy kilka elementarnych działań matematycznych:
Symbolic a("a"),z("z"); // definiowanie zmiennych
Symbolic b = (a*a*a)+(a); // przypisanie zmiennej wyrażenia
cout « "b="« b[a==2] « endl; // podstawienie za "a" liczby 2 i w wyniku dostajemy 10
Symbolic c = exp(a) + 3; // definiowanie zmiennych przy użyciu funkcji matematycznych cout « "c="« c[a==10] « endl; // podstawienie za "a" liczby 10
cout « int (((z-2)*(z-2) - z*(z-4))« endl; // rozwiązuje wyrażenie algebraiczne i wyświetla tylko liczbę integer, gdy nie ma jej w równaniu to błąd
cout « double ((z-0.5)+(1.0-z))« endl; // rozwiązuje wyrażenie algebraiczne i wyświetla tylko liczbę double, gdy nie ma jej w równaniu to błąd

Jak widzimy SymbolicC++ wylicza wyrażenia algebraiczne tzn. np. redukuje wyrazy podobne.
Zaprezentujemy teraz wielomian trzeciego stopnia oraz możliwość określenia współczynników wyrażenia poprzez użycie następującego polecenia nazwa.coeff():
Symbolic s("s"), p("p");
Symbolic wiel = (4*s-7*p)*(4*s-7*p)*(4*s-7*p); // definiujemy wielomian w postaci wzoru skróconego mnożenia
cout « "wielomian = " « wiel « endl; // wynik z poprzedniej linijki po zastosowaniu wzoru skróconego mnożenia
cout « wiel.coeff(p*p*p) « endl; // współczynnki przy p do 3
cout « wiel.coeff(s,2) « endl;// drugi sposób zapisania potęgi
cout « wiel.coeff(s) « endl; // współczynnki przy s
cout « wiel.coeff(s*p) « endl; // współczynnki przy sp
cout « wiel.coeff(p,0) « endl;// współczynnki tam gdzie p=0
cout « wiel.coeff(s*p,0) « endl;// współczynnki tam gdzie sp=0

Podstawowe całki obsługiwane są za pomocą funkcji integrate(). Poniżej prezentujemy kilka przykładów:
Symbolic w("w");
Symbolic cal = (w*w*w);
Symbolic cal1 = (cos(w)+ (w*w));
Symbolic cal2 = ((((w*w)-1)*((w*w)-1)*((w*w)-1))/(w));
Funkcja integrate() przyjmuje dwie wartości pierwsza to funkcja do scałkowania, a druga to argument po którym ma całkować.
cout « integrate(cal,w) « endl; // wyświetal wynik całki cal
cout « integrate(cal1,w) « endl; // wyświetal wynik całki cal1
cout « integrate(cal2,w) « endl; // wyświetal wynik całki cal2

Podstawowe pochodne obsługiwane są za pomocą funkcji df(). Poniżej kilka przykładów:
Symbolic r("r"), k("k"), l("l");
cout « df(cos(r*r) - (r*r)*sin(r),r)« endl; // wyliczenie zwykłej pochodnej Symbolic poch = (2*(k*k)*(l)-(5*(k*k)*(l*l*l))); // zdefiniowanie funkcji dwóch zmiennych
cout « df(poch,k) « endl; // liczenie pochodnej cząstkowej po "k"
cout « df(poch,l) « endl; // liczenie pochodnej cząstkowej po "l"
cout « df(df(poch,k),k) « endl; // liczenie drugiej pochodnej cząstkowej po "k"

Macierze
Do tworzenia macierzy niezbędne jest dołączenie pliku nagłówkowego #include "matrix.h". Numeracja kolumn i wierszy w macierzach zaczyna się od 0. Poniżej przykłady tworzenia macierzy i elementarne operacje na nich:

int n = 2; // liczba typu integer, która definiuje nam wymiar macierzy

Matrix <Symbolic> A(n,n); // tworzymy macierz 2 na 2
A[0][0] = 1; A[0][1] = 1; // podstawienie danych do odpowiednich współczynników macierzy
A[1][0] = 2; A[1][1] = 1;

Matrix<Symbolic> B(n,n);
B[0][0] = 1; B[0][1] = 9;
B[1][0] = 0; B[1][1] = 8;

cout « A+B « endl; // dodawnaie macierzy
cout « A-B « endl; //odejmownia macierzy
cout « A*B« endl; // mnożenie macierzy
cout « B.inverse() « endl; // macierz odwrotna
cout « det(A) « endl; // wyznacznik macierzy

Symbolic X;
Symbolic alf("alf");
X = ((cos(alf), sin(alf)),
(-sin(alf), cos(alf)));
cout « X « endl; // wyświetal macierz
cout « X.transpose() « endl; // transpozycja macierzy
cout « A.kron(B) « endl; // Kronecer
cout « A.dsum(B)« endl; // macierze na diagonali a pozostał wypełnione są zerami

Wielomiany Lagrange'a opisane są poniższym wyrażeniem:

$\prod_{0\leq m\leq km\neq j} \frac{x-x_m}{x_j-x_m} = \frac{(x-x_0)}{(x_j-x_0)} \cdots \frac{(x-x_{j-1})}{(x_j-x_{j-1})} \frac{(x-x_{j+1})}{(x_j-x_{j+1})} \cdots \frac{(x-x_k)}{(x_j-x_k)}$
Kod programu:

```
#include <iostream>
#include "symbolicc++.h"
using namespace std;

int main(void)
{
Symbolic x("x"), x0("x0"), x1("x1"), x2("x2");
Symbolic f = x*x*x;
Symbolic f_x0("f_x0");
Symbolic f_x1("f_x1");
Symbolic f_x2("f_x2");
Symbolic L = (((f_x0) *(((x-x1)/(x0-x1))*((x-x2)/(x0-x2))))+((f_x1) *(((x-x0)/(x1-
x0))*((x-x2)/(x1-x2))))+((f_x2) *(((x-x0)/(x2-x0))*((x-x1)/(x2-x1)))));
cout « "L="« L[x0==1,x1==2,x2==3,f_x0==1,f_x1==8,f_x2==27] « endl;
return 0;
}
```

3 GiNaC

3.1 Wprowadzenie

GiNaC to darmowy system algebry komputerowej wydany na licencji GNU GPL. Nazwa jest akronimem rekurencyjnym "GiNaC nie jest CAS". Wadą GiNaCka w porównaniu z innymi systemami algebry komputerowej jest to, że interfejs nie zapewnia wysokiego poziomu interakcji z użytkownikem. GiNaC jest biblioteką C++, która pozwala użytkownikowi na wykonywanie operacji algebraicznych z poziomu języka C++. Używa on biblioteki CLN do implementacji dużych liczb całkowitych. Wykorzystuje się go w szczególności do teoretycznych obliczeń w fizyce wysokich energii, a także poza zagadnieniami fizycznymi. Stał się on podstawą w kilku projektach open-source: rozszerzenie GNU Octave, symulator do obrazowania metodą rezonansu magnetycznego, wspomaganie dla wyrażeń symbolicznych w Sage. Pierwsza wersja wydana w 2001 roku zawierała wszystkie niezbędne funkcje przeznaczone do wykonywania operacji algebraicznych. Od tego czasu GiNaC jest ciągle ulepszany i rozszerzany. Najnowsza wersja jest z grudnia 2015 roku - 1.6.6.

3.2 Instalacja

Przed ręcznym instalowaniem GiNaCka trzeba pobrać bibilotekę CLN i zainstalować ją w systemie. Bez tej biblioteki GiNaC jest bezwartościowy.
Instalacja biblioteki CLN:

1. ./configure
2. make
3. make check - sprawdza czy poprzednie kroki zostały poprawnie wykonane
4. make install

Po poprawnym zainstalowaniu bibiloteki CLN pobieramy ze strony git://www.ginac.de/ginac.git paczkę ginac-1.6.6.tar.bz2, którą następnie rozpakowujemy, a później instalujemy.

3.3 Programy

Aby uruchomić program musimy najpierw wejść do katalogu, w którym on się znajduje. Uruchomienie programu odbywa się poprzez wpisanie w terminalu komendy: g++ -o nazwaprogramuwynikowego -Wl, –no-as-needed' pkg-config –cflags –libsginac' nazwaprogramu.cpp. Nazwaprogramu - program, który chcemy skompilować, a nazwaprogramuwynikowego to wynikowy program po kompilacji (w zależności od użytkownika nazwy te mogą być takie same). Po poprawnej kompilacji wpisujemy komende: ./nazwaprogramuwynikowego. Innym sposobem (wygodniejszym) jest Makefile'a, w którym podajemy wspomnianą wcześniej komende do kompilacji programu. Makefile'a musi się znajdować w katalogu, w którym jest program. Uruchomienie go w konsoli odbywa się poprzez komende make run. Poniżej przykładowy kod Makefile'a, którego używamy do kompilacji:

```
CC=g++ // używany kompilator
PROGRAM=ginac.cpp // nazwa programu do kompilacji
CFLAGS=-Wl,–no-as-needed ‘pkg-config –cflags –libs ginac‘

run:
$(CC) -o ginac $(CFLAGS) $(PROGRAM)
(time ./ginac)

clean:
rm *.o ginac
```

Rozpoczynając pisanie programu należy użyć następujących bibliotek:

- #include <ctime>
- #include <iostream>
- using namespace std;
- #include <ginac/ginac.h>
- using namespace GiNaC;

Trzy pierwsze są standardowymi plikami nagłówkowymi języka programowania C++, a dwie pozostał służą do uruchomienia biblioteki GiNaC.
Przykładowy program wygląda następująco:

```
#include <ctime>
#include <iostream>
using namespace std;
#include <ginac/ginac.h>
using namespace GiNaC;
int main()
{
symbol x("x"), y("y");
ex Ex1 = 6*x-2*y+3*y-2+x;
std::cout « Ex1 « std::endl; return 0;
}
```

Po wykonaniu powyższego programu otrzymujemy: 7x + y -2.
W programie użyto typ danych dla nieokreślnych symboli matematycznych, porzez następujące zdefiniowanie: symbol x("x"), y("y");
GiNaC przechowuje tak zdefiniowane zmienne jako stringi. Równania przechowywane są w innych typach danych ex, pozwala to odpowiednie zmienne zawierać w dowolne wyrażenia symboliczne dla wartości numerycznych. Pod zdefiniowane zmienne możemy podstawiać wartości liczbowe.
GiNaC wykonuje kilka automatycznych przekształceń na wyrażeniach, aby je uprościć i umieścić je w postaci kanonicznej, poniżej kilka przykładów:
ex Ex2 = y-y; - w wyniku otrzymujemy 0,
ex Ex3 = x*x/x; - w wyniku otrzymujemy x,
ex Ex4 = cos(2*Pi); - w wyniku otrzymujemy 1.
Wymienione typy danych są najważniejszymi typami w GiNaCku.
Wyrażenia algebraiczne w GiNaCku mogą składać się także z liczb, takich jak np. liczby zmiennno przecinkowe. Użytkownik może utworzyć obiekt klasy numerycznej na kilka sposobów:

- numeric two = 2; // zmienna typu integer

- numeric r(2,3); // ułamek
- numeric e(2.71828); // zmienna typu float

Poniżej przedstawiamy kilka przykładów działań matematycznych z użyciem zmiennej typu integer:
numeric a = 2; // deklaracja zmiennej
numeric b = -3;
numeric c = 4;
numeric n = 10;
numeric add = a+b; // -1
numeric sub = a-b; // 5
numeric mul = a*b; // -6
numeric div = c/a; // 2
Kolejnym ważnym przykładem są funkcje matematyczne, które mogą być stosowane dla danych liczbowych. Przykłady:
numeric pot = pow(b,a); // potęgowanie
numeric pierw = sqrt(c); // pierwiastkowanie
numeric wbz = abs(b); // wartość bezwzględna
numeric sil = factorial(n); // silnia
numeric cf = fibonacci(n); // ciąg fibonacciego
numeric inv = inverse(a); // odwrotność liczby a
Oprócz powyższych funkcji możemy użyć także: log(z), funkcje trygonometryczne i wiele innnych.
Lista klasy GiNaC służy do przytrzymywania list dowolnych wyrażeń. Nie są one tak wszechobecne jak w wielu innych pakietach algebry komputerowej, ale trzeba mieć podstawową wiedzę o nich. Numeracja poszczególnych elementów w liście zaczyna się od 0. Lista może być konstrułowana w następujący sposób:
{
symbol x("x"), y("y");
lst l; // deklaracja listy "l"
l = x, 4, x-y, y, y+x;
cout « l.nops() « endl; // wyświetla ile elementów jest w liście
cout « l.op(4)« " " « l[1]«endl; // dostęp do poszczególnych elementów z listy
...
Modyfikacja list:
k = l[2] = 15; // podstawianie nowej wartości za drugi element w liście
p = l.let_op(2) = x; // drugi sposób podstawienia
s = l.append(3*y); // dodanie na koniec listy nowego elementu
f = l.remove_last(); // usuwa ostatni element z listy
t = l.remove_all(); // usuwa całą liste
Sortowanie list: lst l1, l2, s1, s2;

```
l1 = x, 2, y, x+y;
l2 = 2, x+y, x, y;
s1 = l1.sort();
s2 = l2.sort();
```
Po wykonaniu operacji sortowania listy l1 i l2 są sobie równe.
Symboliczne całkowanie wymaga użycia klasy integral. Poniżej przykład:
```
symbol intf("inft");
intf = integral(x,0,1,x*x).eval_integ(); // oblicznie całki z funkcji x*x w granicy od 0 do 1
```
Możemy również obliczać pochodne zarówno pierwszego jak i wyższych rzędów oraz pochodne cząstkowe. Poniżej przykłady:
```
{
symbol z("z");
ex P = pow(x, 5) + pow(x, 2) + y;
cout « P.diff(x,2) « endl; // wyświetla drugą pochodną cząstkową po x
cout « P.diff(y) « endl; // wyświetla pochodną cząstkową po y
cout « P.diff(z) « endl;// wyświetla pochodną cząstkową po z
```

Macierz jest dwuwymiarową tablicą wyrażeń. Elementy macierzy o m wierszach i n kolumnach, dostępne są z dwoma indeksami, pierwszy w zakresie od 0 do m-1, drugi od 0 do n-1. Istnieje kilka sposobów, aby skonstruować macierz z/bez gotowych elementów. Jeden ze sposobów tworzenia macierzy przedstawiony jest poniżej:
```
matrix m(3,3); // w nawiasie deklarujemy ilość kolumny i wierszy
m = 11, 12, 13, // deklarujemy macierz
21, 22, 23,
31, 32, 33;
cout « reduced_matrix(m, 1, 1) « endl; // usunięcie pierwszej kolumny i pierwszego wiersza z macierzy m
cout« unit_matrix(3) « endl; // tworzy macierz jednostkową
```
Operacje na macierzach:
```
matrix A(2,2), B(2,2), C(2,2); // deklaracja macierzy 2 2
A = 1, 3,
2, 4;
B = -1, 0,
-2, 1;
C = 8, 4,
2, 1;
matrix dod = A.add(B);// dodawanie macierzy
matrix nn = C.pow(3);// potęgowanie macierzy
matrix result = C.mul(B).sub(A.mul_scalar(3)); // pierwszy sposób
```

```
ex r = C*B - 3*A;
cout « r.evalm() « endl; // drugi sposób
matrix trans = transpose(A); // transpozyjca macierzy
```

Wielomiany Hermite'a to wielomiany o współczynnikach rzeczywistych, będące rozwiązaniem poniższego równania.

$H\_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$
Równanie to występuje także w innych postaciach. Poniżej kod programu w wyniku którego otrzymujemy kilka początkowych wielomianów.

```
#include <ctime>
#include <iostream>
using namespace std;
#include <ginac/ginac.h>
using namespace GiNaC;
ex HerWiel(const symbol & x, int n)
{
ex Her = exp(-pow(x,2));
ex H = ((pow(-1,n))*(exp(pow(x,2)))*(pow(Her,n)));
}

int main()
{ symbol h("h");
for(int i=0; i<10; ++i)
cout « "H_ "« i « "(h) == " « HerWiel(h,i) «endl;
return 0;
}
```

4 SymbolicC ++ vs GiNaC

Zarówno SymbilicC ++ jak i GiNaC są darmowym oprogramowaniem opartym na licencji GNU GPL. SymbolicC ++ bazuje na C++. GiNaC stosuje zintegrowane obliczenia symboliczne w programach C++ i kładzie duży nacisk na interoperacyjność. Jeżeli chodzi o różnice w pisaniu programów to są one następujące:

- definicja zmiennych w symbolicc++ to Symbolic x("x") a w GiNaC symbol x("x")

- deklaracja macierzy w symbolicc++ Matrix<Symbolic>A(2,2) a w GiNaC matrix A(2,2)
- tworzenie macierzy o swoich parametrach wygląda w symbolicc:
  A[0][0] = 1; A[0][1] = 1;
  A[1][0] = 2; A[1][1] = 1;
  a w GiNaC:
  A = 1, 3,
  2, 4;
- funkcja do obliczania pochodnych w symbolicc++: df() a w GiNaC: .diff()
- funkcja do obliczania całek w symbolicc++: integrate() a w GiNaC: integral().eval_integ()

GiNaC zawiera typ danych: ex - zapis równań; numeric - podaje dane liczbowe; oprócz tego możemy definiować listy. Najnowsza wersja GiNaCka jest z 2015 roku a SymbilicC ++ z 2010.

5 Podsumowanie

Naszym zdaniem lepiej pracuje się w GiNaCku, ponieważ jest bardziej zrozumiały, nie miałyśmy z nim żadnych problemów. Natomiast przy pracy z symbolikiem napotkałyśmy dużo trudności m.in. problem z kompilowaniem plików (error 139), który polegał na tym, że przy bardziej skomplikowanych obliczeniach wymagał nowszej wersji kompilatora lub większej ilości pamięci.

Literatura

[1] GiNaC `http://www.ginac.de`

[2] GiNaC `https://en.wikipedia.org/wiki/GiNaC`

[3] SymbolicC++ `http://issc.uj.ac.za/symbolic/symbolic.html`

[4] SymbolicC++ `https://en.wikipedia.org/wiki/SymbolicC`