Łukasz Czapliński
e-mail: czapl.luk@gmail.com

Pracownia z analizy numerycznej

Sprawozdanie do zadania **P2.2**

Prowadzący: dr Paweł Woźny

Wrocław, dnia 10 grudnia 2012 r.

1. Wstęp

1.1. Słowo wstępu

Numeryczne szukanie miejsca zerowego funkcji to temat wyjątkowo obszerny. Paradoksalnie może się jednak wydawać że wszystko zostało już w tym temacie omówione i zbadane. Znane są metody pozwalające bardzo szybko zwiększać dokładność, z jaką szacujemy miejsce zerowe (jak metoda Newtona), jak i pozwalające określać dobrze te początkowe przybliżenia (jak metoda bisekcji). Łącząc kilka takich metod otrzymujemy narzędzie, wydawać by się mogło, idealne. Dobrym przykładem jest tu metoda Brenta. Jednak takie konglomeraty, rozważające w każdym kroku, której ze swoich podmetod użyć mają zadaniczą wadę: procesor wykonuje je wolniej. O wiele szybciej liczone są te, które w każdym kroku pętli po prostu wyliczają kolejne przybliżenie i sprawdzają czy jest ono dość dokładne, zamiast najpierw wykonywać kilka "if'ów". W tej pracowni omówie 3 takie metody: dobrze znane metody stycznych i siecznych oraz mniej znaną odwrotnej interpolacji kwadratowej.

1.2. Opis metod

Praktycznie każdy, kto choć chwilę interesował się tematem analizy numerycznej zna metody siecznych i stycznych.

Metoda siecznych liczy kolejne przybliżenia ze wzoru:

$$x_{n+1} = x_n - \frac{y_n * (x_n - x_{n-1})}{y_n - y_{n-1}} \tag{1}$$

Z kolei metoda stycznych ze wzoru:

$$x_{n+1} = x_n - \frac{y_n}{y'_n} \tag{2}$$

Intuicyjnie oznacza to tyle, że przybliżamy funkcję $y = f(x)$ w otoczeniu x_n przez funkcję liniową (metoda siecznych wykorzystuje w tym celu 2 poprzednie wartości x, a metoda stycznych pochodną funkcji f), a następnie za kolejne przybliżenie x_{n+1} przyjmujemy miejsce zerowe tej funkcji liniowej. Poprzez analizę tych wzorów można też wykazać, że rząd zbieżnosci metody stycznych jest większy niż rząd zbieżnosci metody siecznych (rząd zbieżności określa jak bardzo kolejne zastosowanie metody zwiększy precyzję wyznaczonego przybliżenia miejsca zerwego).

Trzecia metoda, odwrotnej interpolacji kwadratowej, ma trochę bardziej zawiły wzór:

$$x_{n+1} = \frac{x_{n-2} * y_{n-1} * y_n}{((y_{n-2} - y_{n-1}) * (y_{n-2} - y_n))} + \frac{x_{n-1} * y_{n-2} * y_n}{((y_{n-1} - y_{n-2}) * (y_{n-1} - y_n))} + \frac{x_n * y_{n-2} * y_{n-1}}{((y_n - y_{n-2}) * (y_n - y_{n-2}))} \tag{3}$$

Odpowiada to przybliżeniu danej funkcji $y = f(x)$ przez funkcję kwadratową, wykorzystując 3 poprzednie wartości x, a następnie znalezieniu jej miejsca zerowego przez wyliczenie funkcji do niej odwrotnej $g = f^{-1}$ i wartości $g(0)$ (ponieważ $g = f^{-1} \Rightarrow x_0 = g(0) \Leftrightarrow f(x_0) = 0$). Tutaj już pojawia się pierwsza wątpliwość: przecież nie zawsze istnieje funkcja odwrotna do funkcji kwadratowej na zadanym przedziale, co wtedy? Kolejnym pytaniem jest: jeśli ta metoda działa, to jaki jest jej rząd zbieżności? Czy jest porównywalna z którą z poprzednich metod?

1.3. O programie

Aby rzetelnie odpowiedzieć na te pytania, napisany został obszerny program testujący. Składa się on z kilku modułów. Pierwszy z nich odpowiada za parsowanie i obliczanie wartości funkcji. Znajduje się w pliku fhb.cpp. Pozwala na stworzenie struktury

`fhb::fun_wrap`

za pomocą konstruktora

`fhb::fun_wrap(string s)`

gdzie s zawiera wzór funkcji - więcej o nim dalej - i liczenie wartości tej funkcji w punkcie x za pomocą metody

`double fhb::fun_wrap::apply(double x)`

Kolejny, zawarty już w program.cpp odpowiada za implementację omawianych metod. Aby wymusić możliwie najlepsze warunki do porównania czasu działania metod, zostały one napisane w stylu funkcyjnym: każda z funkcji implementujących metodę składa się z procedury odpowiadającą za jeden krok obliczeń: mając daną funkcję i ciąg przybliżeń, zwraca ona kolejne przybliżenie. Nie ma tu spamiętywania: każda procedura wylicza wszystkie potrzebne wartości funkcji. Później wyjaśnię powód tak nieoptymalnego rozwiązania. Funkcja ta jest następnie iterowana (przez funkcję iterate), aby uzyskać ciąg kolejnych przybliżeń miejsca zerowego.
Nota o badaniu zbieżności w programie.
Warunki stopy zostały dobrane tak, żeby obliczenia zatrzymywały się, gdy obliczenia są zbieżne do jakiegokolwiek miejsca zerowego. Funkcja iterująca w żaden sposób nie bada odległości od przewidywanego przez nas miejsca zerwego.
Trzeci moduł to funkcja

`do_computations`

która odpowiada za komunikację programu z użytkownikiem. Przyjmuje jako argumenty metodę, jej nazwę oraz nazwę testu, który ma być wykonany, następnie wczytuje test z pliku

`"./testy/<nazwa_testu>.din"`

i wypisuje kolejno wyniki:

co?	gdzie? - nazwa pliku
ciąg przybliżeń miejsca zerowego	`"./wyniki/<nazwa_testu>/<nazwa_metody>.dw"`
rząd zbieżności dla kolejnych przybliżeń	`"./wyniki/<nazwa_testu>/<nazwa_metody>.dr"`
statystyki (czas, ilość iteracji, średni rząd zbieżności)	`"./wyniki/<nazwa_testu>/<nazwa_metody>.dt"`

Ogólnie sam program przyjmuje jako argument nazwę testu i wywołuje

`do_computations`

dla wszystkich 3 metod. Dodatkowo zamieszczone zostały skrypty (w folderze "./skrypty"). Główny z nich - replot.sh - upewnia się, że istnieje odpowiednia struktura katalogów (tworzy wymagane, jeśli ich nie ma), kompiluje program, wywołuje go dla każdego testu z foldery "./testy", następnie tworzy plik z poleceniem dla gnuplota, wywołuje gnuplota by stworzył wykresy i umieszcza je w folderze "./wykresy". Następnie wywołuje pdftex'a, by ponownie stworzył sprawozdanie (ze zmienionymi wykresami i tabelkami, np uaktualniania jest zbiorcza tabela wyników testów na stronie 4) i usuwa zbędne pliki.

1.3.1. O parserze

Używany parser z pliku

`fhb.cpp`

został napisany przez autora jako projekt na ANSI C zanim poznał on słowo "parser", więc naturalnie ma pewne ograniczenia:
1. Nie obsługuje on potęg. Tak więc x^2 należy podać jako $x * x$.
2. W funkcji może występować tylko jedna zmienna i należy ją oznaczyć jako x.
3. Niedozwolone są znaki białe - nie powinna wystąpić żadna spacja we wzorze funkcji.
4. Obsługiwane są tylko wbudowane funkcje: *sin*, *cos*, *sqrt*, *ln* i specjalna *mpi*. Każda z nich musi otrzymać jakiś argument - funkcję w nawiasach () lub samo x. Funkcja *mpi* zwraca zawsze PI, niezależnie od argumentu.
5. Poza tym argument - string może być dowolną (matematycznie poprawną) funkcją składającą się ze znaków: x, $*$, $/$, $+$, $-$, nawiasów () i stałych.

1.3.2. Struktura testu

Test powinien się składać z 4 linii. Powinny one wyglądać następująco:

1. ciąg znaków - 3 liczby typu double (w notacji angielskiej, . zamiast ,) - kolejne przybliżenia miejsca zerwego. W kolejności od najmniej dokładnego do najbardziej. Oddzielone spacjami.
2. ciąg znaków - funkcja, której miejsce zerowe szukamy. Format taki, jak dla parsera.
3. ciąg znaków - pochodna powyższej funkcji. Format jw.
4. ciąg znaków - funkcja 0-argumentowa w formacie dla parsera - dokładne miejsce zerowe funkcji. Pozwala to wygodnie podawać miejsca zerowe funkcji trygonometrycznych. Program sprawdza po wczytaniu czy faktycznie funkcja ta daje stałą wartość dla różnych wyrażeń.

2. Wyniki doświadczenia

2.1. Ogólne spojrzenie na wyniki

Oto pełne wyniki przeprowadzonych doświadczeń.

n	Funkcja	Przybliżenia		
test0	x*x-9	0	1.5	2.5
test1	x*x+x-6	0.	-1.5	-2.
test2	sin(x)+1	0.	-1.	-1.5
test3	x+sin(x-3)-3	1.	1.8	2.7
test4	-x*x+4	1.	-1.	-1.5
test5	-x*x+9	2.	1.	0.
test6	sin(x)*(x+1)	1.	0.7	0.5
test7	x*x*x*x-16	1.5	1	-1
test8	x*x	1.5	1	0.6

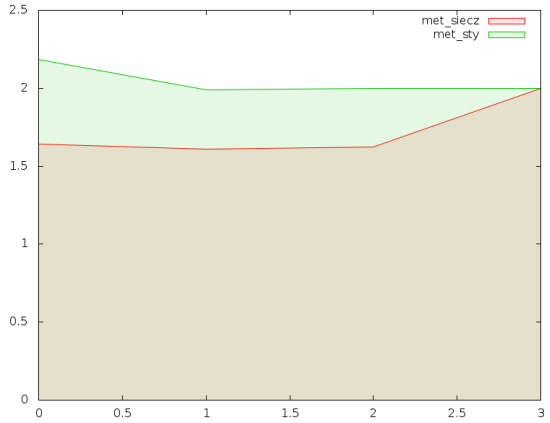
	met sty				met siecz				odwr interpol kw			
n	t	i	r	m	t	i	r	m	t	i	r	m
test0	25	8	1.99	3	82	10	1.82	3	40	10	1.93	3
test1	36	9	1.99	-3	73	11	1.63	-3	53	11	1.98	-3
test2	97	27	0.482	-1.57	318	103	0	-nan	126	33	0.621	-1.57
test3	58	7	2.51	3	121	8	2.17	3	56	9	1.84	3
test4	23	8	2.04	-2	66	10	1.72	-2	309	103	0	nan
test5	222	103	0	-nan	82	14	-0.00856	3	19	5	1.76	-3
test6	60	9	1.98	0	96	11	1.62	3.26e-28	65	10	1.85	3.85e-34
test7	51	13	2	-2	373	103	0	-nan	475	103	0	-nan
test8	88	43	1	5.46e-13	170	59	1	1.2e-12	263	52	1	1.11e-12

Przybliżenia	Szacunkowe wartości miejsca zerowego, od których zaczynać miały metody
t	Czas wykonywania obliczeń w mikrosekundach
i	Ilość wykonanych iteracji
	(wartość 103 oznacza, że nie udało się znaleźć dość dobrego przybliżenia)
r	Średnia wartość numerycznego rzędu zbieżności
m	Ostatnie przybliżenie miejsca zerowego

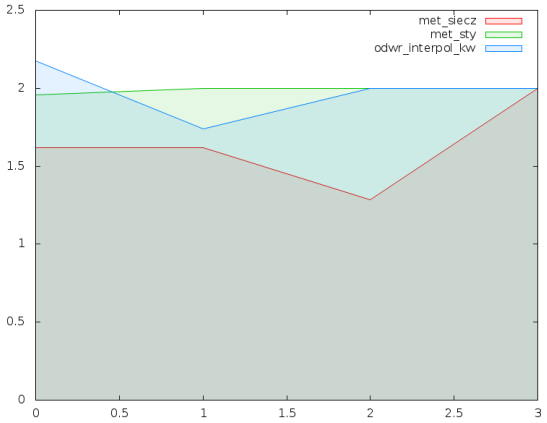
Patrząc ogólnie na powyższe wyniki można zauważyć, że metoda stycznych w większości przypadków jest o wiele lepsza od metody stycznych czy też odwrotnej interpolacji kwadratowej. Potwierdza to spojrzenie na wykresy 1 i 2.

Ciekawsze jednak wydają się te przypadki, w których któraś z metod okazała się zbyt wolno zbieżna (test2, test4, test5, test6, test7). Przyjrzyjmy się im dokładniej.

Wykresy można znaleźć w katalogu "./wykresy". Dla każdego testu znajduje się tam wykres $y = x_n$, rzędu



Rysunek 1: Numeryczny rząd zbieżności w teście4



Rysunek 2: Numeryczny rząd zbieżności w teście1

zbieżności oraz wartości, które znajdują się w tabelce powyżej. Z kolei numeryczne wartości są w plikach wymienionych przy opisie programu.

2.2. Brak zbieżności metod

Widzimy, że każda z metod dla pewnych warunków startowych nie jest zbieżna do miejsca zerowego (czasem, jak w teście 5 bywa zbieżna do innego miejsca zerowego). Metoda siecznych nie jest zbieżna, gdy prosta przechodząca przez punkty $(x_n, f(x_n))$ i $(x_{n+1}, f(x_{n+1}))$ jest równoległa do ox. Metoda stycznych - gdy $f'(x_n) = 0$. Z kolei odwrotna interpolacja kwadratowa - gdy nie ma funkcji odwrotnej $g = y^{-1}$ na zadanym przedziale (gdzie $y \in \Pi_2$ interpoluje f w x_n, x_{n-1}, x_{n-2}). Widać tu podobieństwo metody siecznych i odwrotnej interpolacji kwadratowej - obie mają duże problemy z funkcjami, które nie są rożnowartościowe w zadanym otoczeniu miejsca zerowego. Mogłoby się wydawać, że odwrotna interpolacja kwadratowa, dzięki temu, że bierze więcej punktów powinna być bardziej odporna na perfidny ich wybór. Jak się okazuje, tak nie jest: była ona zawodna w większości przypadków, w których zawodziła metoda siecznych, a także w innych. Metoda stycznych przeszła wszystkie testy, które nie były specjalnie w nią wymierzone. Na korzyść odwrotnej interpolacji kwadratowej przemawia fakt, że jej rząd zbieżności był lepszy niż metody siecznych. Z tego powodu jej wykonanie było zwykle około 1,5-2x szybsze niż metody siecznych.

3. Zakończenie

Podsumowując, metoda odwrotnej interpolacji kwadratowej jest przydatna, gdy z jakichś powodów nie możemy zastosować metody stycznych (np nie znamy lub nie możemy policzyć pochodnej funkcji). Jest ona szybciej zbieżna niż metoda siecznych, lecz bardziej kapryśna niż metoda stycznych. Nawet bez spamiętywania jej wykonanie jest szybsze niż metody siecznych. Zastanawiające jest więc, czy metoda odwrotnej interpolacji sześciennej nie okazałaby się jeszcze lepsza, dodatkowo będąc pozbawioną wady braku funkcji odwrotnej w liczbach rzeczywistych. Problemem mogłoby być liczenie współczynników, lecz to temat na inną pracownię.