OD AUTORÓW **1**
WSTĘP **4**
I. ROZWIĄZYWANIE UKŁADÓW RÓWNAŃ LINIOWYCH **9**
1. SKALARY 9
2. WEKTORY 9
3. MACIERZE 14
4. SLES, CZYLI ROZWIĄZYWANIE UKŁADÓW RÓWNAŃ LINIOWYCH 22
5. PRACA Z PETSC 24
6. PIERWSZY „POWAŻNY" PROGRAM UŻYWAJĄCY PETSC - ROZWIĄZYWANIE RÓWNANIA POISSONA 27
7. KSP, CZYLI ITERACYJNE METODY ROZWIĄZYWANIA RÓWNAŃ 29
8. PC, CZYLI PRECONDITIONERY 32
9. SZCZEGÓŁY O PRECONDITIONERZE AMS, KTÓRYCH NIE ZNAJDZIECIE W OFICJALNYM MANUALU 42
II. ROZWIĄZYWANIE UKŁADÓW RÓWNAŃ NIELINIOWYCH **47**
10. RÓWNANIA NIELINIOWE 47
11. SCHEMAT UŻYCIA SNESU DO ROZWIĄZANIA RÓWNANIA $F(x^*)=0$ 48
12. PIERWSZE ZADANIE NIELINIOWE W PETSC 48
13. NIELINIOWE ZAGADNIENIE RÓŻNICZKOWE W PETSC 56
14. CO JESZCZE WARTO WIEDZIEĆ O SNESIE? 65
15. PODSUMOWANIE SNESU 67
O CZYM NIE NAPISALIŚMY **70**
BIBLIOGRAFIA **71**

Od autorów

PETSc jest biblioteką procedur służących do rozwiązywania zagadnień powstałych w wyniku dyskretyzacji równań różniczkowych cząstkowych, rozwijaną od początku lat 90-tych przez B.Smith'a i in. w Argonne National Laboratory (USA). Programy używające PETSc mogą działać na wielu architekturach, od komputerów osobistych, po masywnie równoległe maszyny używające standardu MPI. PETSc jest przeznaczone dla naukowców zajmujących się *scientific computing*, jest też bardzo przydatne dla tworzenia i testowania własnych algorytmów rozwiązywania równań cząstkowych. Na przykład, przez ostatni rok z powodzeniem stosowaliśmy PETSc do testowania nowych metod dekompozycji obszaru oraz do rozwiązywania równań Naviera-Stokesa i mikropolarnych. Naszym zdaniem, PETSc ma wiele zalet sprawiających, że warto korzystać z tego pakietu.

Co znajdziemy w pakiecie PETSc?

Pakiet daje zbiór narzędzi służących do rozwiązywania zagadnień algebraicznych o dużym rozmiarze, zwłaszcza powstałych w wyniku dyskretyzacji równań różniczkowych cząstkowych. Poza wszystkim innym, PETSc jest także po prostu dobrą i bogatą biblioteką procedur numerycznych, opartych na takich standardach jak BLAS i LAPACK. Wśród funkcji oferowanych przez PETSc znajdują się gotowe i przetestowane procedury rozwiązywania wielkich układów:

- równań liniowych, metodami bezpośrednimi i iteracyjnymi;
- równań nieliniowych metodami typu Newtona;
- równań różniczkowych zwyczajnych metodami wielokrokowymi.

Wszystkie te procedury są niezależne od faktycznie używanych struktur danych. PETSc zorientowana jest na usprawnienie procesu tworzenia dużych aplikacji numerycznych działających na komputerach równoległych, dlatego w pakiecie znajdziemy:

- procedury wysokiego poziomu, służące do manipulacji obiektami w rodzaju wektor, macierz, itp., niezależne od tego, czy pracujemy na komputerze sekwencyjnym, czy równoległym (to bardzo wygodne dla programisty!);
- dostęp do procedur niskiego poziomu do precyzyjnego zarządzania rozproszeniem danych pomiędzy procesorami, sposobu komunikacji, itd.

Procedury związane z przetwarzaniem równoległym są oparte na powszechnie uznanym standardzie MPI, zaimplementowanym na większości obecnie używanych komputerów równoległych i stacji roboczych.

Przy tworzeniu kolejnych wersji biblioteki PETSc aktywnie uczestniczyli (i uczestniczą!) jej użytkownicy. Dlatego PETSc nie zapomina o wygodzie programisty i oferuje najrozmaitsze udogodnienia przydatne podczas uruchamiania aplikacji:

- zmianę parametrów i używanych metod bezpośrednio z linii komend;
- przyjazne przeglądarki do generowanych obiektów np. macierzy lub wektorów;
- system śledzenia (*profiling*) obliczeń zmiennoprzecinkowych i wykorzystania pamięci;
- rozległe możliwości wyłapywania błędów (*debugging*);
- środowisko graficzne.

Dlaczego warto poznać PETSc?

Już omówienie zawartości pakietu sugeruje, że jest ona całkiem smakowita. Obecnie skoncentrujemy się na omówieniu typów aplikacji, gdzie użycie PETSc może przynieść istotne korzyści.

Optymalizacja pracy programistycznej

Tworzenie dużych aplikacji, a takimi są najczęściej solvery równań różniczkowych cząstkowych, jest żmudne i czasochłonne. Stosowanie dobrych bibliotek prowadzi do bardziej zwartego kodu i oszczędza programiście pisania procedur znanych od dawna (w rodzaju metody gradientów sprzężonych). Wiadomo także, iż optymalizacja kodu na zadaną architekturę komputera jest trudna - wymaga specjalistycznej wiedzy i wielkiej praktyki - i prowadzi najczęściej do utraty przenośności kodu na inne architektury. Te problemy potęgują się (a nawet rosną wykładniczo szybko...) z chwilą, gdy nasz program ma dobrze działać na maszynie wieloprocesorowej (poza samą trudnością programowania, a następnie testowania i odpluskwiania programu, musimy także zadbać o komunikację, dystrybucję danych pomiędzy procesory i dobrą skalowalność). Biblioteka PETSc w pewnym stopniu uwalnia użytkownika od martwienia się o kodowanie (efektywnych) podstawowych solverów i obiektów. To pozwala skupić się na właściwym algorytmie i jego optymalizacji.

Pamiętajmy także, że ostateczną miarą rzeczywistej efektywności użytej metody rozwiązania danego problemu jest czas i koszt działania komputera *plus* czas i koszt pisania programu!

Eksperymenty numeryczne

Nawet użycie sekwencyjnej wersji PETSc, działającej na zwykłej stacji roboczej lub pececie, może być pożyteczne. Na przykład, jeżeli (z jakichkolwiek powodów) chcemy porównać kilka metod iteracyjnych rozwiązywania dyskretnego liniowego równania eliptycznego, to korzystając z PETSc jedyną trudną czynnością będzie konstrukcja macierzy sztywności i wektora prawej strony. Cała reszta jest już w pakiecie, w dodatku przyjemna w użyciu.

Często, badając nowe algorytmy dla równań cząstkowych chcielibyśmy przeprowadzić eksperymenty, porównujące nasz algorytm z innymi dostępnymi. Łatwo to osiągnąć używając PETSc, ze względu na mnogość zaimplementowanych solverów i łatwość dodawania własnych, oraz możliwości profilingu: możemy zbadać nie tylko liczbę iteracji, ale także wymagania pamięciowe, koszt obliczeń, itp. Co więcej, prowadząc eksperymenty możemy szybko weryfikować w praktyce pewne hipotezy.

Równoległość

Nie istnieje coś takiego jak prosta w implementacji aplikacja równoległa. Architektury masywnie równoległe bardzo istotnie różnią się między sobą i zazwyczaj wymagają od programu dużego dopasowania do własnych możliwości. Ponadto, programując taką maszynę musimy w jakiś sposób zdecydować, jak podzielić zadania między procesory oraz jak (i kiedy) je między sobą komunikować. Są to, niestety, detale dość niskiego poziomu (bliskie sprzętu), będące przyczyną wielu koszmarów dręczących tzw. zwykłych użytkowników maszyn równoległych (np. numeryków, którzy oczywiście chcieliby pisać algorytmy równoległe, ale na wyższym poziomie abstrakcji niż *„spakuj tablicę 10 liczb zmiennoprzecinkowych i wyślij do procesora nr 0”*. Każdy więc krok w stronę tworzenia procedur wysokiego poziomu dla naukowych obliczeń równoległych jest pożądany.

Jednym z istniejących rozwiązań jest język Fortran 90 oraz tworzony obecnie standard HPF. Fortran 90 w części zwalnia użytkownika od myślenia o równoległości, albowiem to kompilator może decydować, kiedy urównoleglić jakąś prostą operację. Tak na przykład działa kompilator F90 na Connection Machine CM-200. Podobne podejście stosuje np. Cray, gdzie jest dostępny tzw. zrównoleglacz kodu, dopisujący w programie odpowiednie dyrektywy dla kompilatora, powodujące wykorzystanie przetwarzania wektorowego lub równoległego.

Innym rozwiązaniem, jak sądzimy, bardziej elastycznym, jest wykorzystanie popularnego standardu MPI (Message Passing Interface) razem z „klasycznymi” językami programowania, takimi jak C czy Fortran. Właśnie ten drugi wariant wybrali autorzy PETSc, opierając na nim swoją implementację. PETSc daje z jednej strony dużo stosunkowo łatwych w użyciu narzędzi pozwalających na programowanie algorytmów równoległych bez potrzeby zagłębiania się w detale komunikacji między procesorami, itp. Z drugiej zaś strony, użytkownik wciąż może (jeśli chce) odzyskać pełną kontrolę nad realizacją przetwarzania równoległego i używać bezpośrednio funkcji standardu MPI.

Kiedy nie warto stosować PETSc?

Na pewno nie warto stosować PETSc do „małych” problemów. Wtedy lepiej skorzystać z Matlab'a lub Maple'a, czy nawet napisać własną procedurę. PETSc *musi* być pakietem zorientowanym na konkretny typ zagadnień (wielkie układy pochodzące z dyskretyzacji równań cząstkowych na maszynach równoległych), aby zachować balans pomiędzy efektywnością a użytecznością. Ostatecznie, nie będziemy przecież używać superkomputera do odwrócenia macierzy 2x2...

Wydaje się nam także, iż raczej nie warto używać PETSc do zadań algebry liniowej z macierzami „gęstymi” (nawet dużego wymiaru). Lepiej skorzystać bezpośrednio z optymalizowanych bibliotek numerycznych, zazwyczaj dostarczanych przez producenta razem z superkomputerem, albo korzystać wprost z BLASów lub procedur LAPACKa czy ScaLAPACKa.

W końcu, wymieńmy parę wad PETSc. Zazwyczaj użycie PETSc prowadzi do wygenerowania kodu wynikowego o sporej objętości. Jest to problem, ale z drugiej strony, nawet najmniejszy program w PETSc daje nam wciąż dostęp do jej funkcjonalności i np. licznych procedur wychwytujących błędy. Również szybkość działania niektórych procedur zostawia trochę do życzenia. Poważną wadą PETSc jest dość stroma krzywa uczenia się: nie sposób zacząć programować w PETSc bez zapoznania się z manualem, a ten z kolei nie jest niestety najłatwiejszą lekturą...

Dlaczego i dla kogo napisaliśmy ten podręcznik?

Zachęceni przez prof. M.Dryję, używamy PETSc od ponad roku. Jak wspomnieliśmy, lektura oficjalnego *manuala* [2] nie jest zbyt przyjemna. Ponadto, niektóre istotne kwestie są w *manualu* potraktowane skrótowo lub w ogóle pominięte. Wymagało to od nas eksperymentowania, przeglądania kodów z przykładami (PETSc przychodzi z zestawem ponad 100 przykładowych programów i pełnym zbiorem kodów źródłowych!) i badania sposobu implementacji konkretnych rozwiązań. Przyznajemy, że czwórka głównych autorów PETSc wykonała gigantyczną pracę (same kody źródłowe zajmują ok. 5 MB), a dostępne *man pages* [3] dla wszystkich funkcji i obiektów PETSc są satysfakcjonujące. Tylko dzięki temu oraz bezpośredniemu kontaktowi z autorami pakietu mogliśmy przebrnąć przez stromą drogę początkowej nauki PETSc, aby teraz móc cieszyć się jej funkcjonalnością i bogactwem, i aby... uczyć się dalej!

Niniejszy podręcznik ma na celu po pierwsze - zainteresowanie tym nowoczesnym i wygodnym pakietem szerszego grona studentów, a także osób zajmujących się obliczeniami naukowymi/eksperymentami numerycznymi w Polsce, a po drugie - oszczędzenie początkującemu użytkownikowi PETSc mozolnego przegryzania się przez oficjalny *manual* i *man pages*: ułatwienie wstępnego zapoznania się z pakietem i szybkie (w miarę bezbolesne) umożliwienie mu rozpoczęcia pisania własnych programów wykorzystujących PETSc.

Adresujemy go przede wszystkim do tych osób, które chcą prowadzić eksperymenty numeryczne w dziedzinie równań różniczkowych cząstkowych typu eliptycznego, liniowych bądź nieliniowych. Dlatego ograniczyliśmy się wyłącznie do wersji sekwencyjnej PETSc, kwestiom równoległości poświęcając jedynie kilka komentarzy i uwag. Mimo to i tak objętość niniejszego podręcznika dorównuje blisko połowie objętości oficjalnego *manuala*, choć dotyczy jedynie 20% jego zawartości. Szczególny nacisk położyliśmy na zilustrowanie omawianych funkcji możliwie licznymi przykładami praktycznych zastosowań. Nie wszystkie przykłady są programami dającymi się uruchomić - często dla większej przejrzystości kodu zrezygnowaliśmy z linii nie związanych bezpośrednio z omawianym zagadnieniem. W każdym rozdziale znajdują się przykłady podsumowujące i te dadzą się skompilować i uruchomić. Ponieważ ten podręcznik ma jedynie pomóc w rozpoczęciu pracy z PETSc, nie będą też prezentowane i omawiane techniki związane z optymalizacją kodu pod kątem obliczeń równoległych[1]. Jak wcześniej argumentowaliśmy, nawet w wersji sekwencyjnej (jednoprocesorowej) pakiet jest bardzo przydatnym narzędziem.

Jeśli będzie takie zapotrzebowanie, być może napiszemy kiedyś drugą część podręcznika, poświęconą technikom równoległym. Będziemy także wdzięczni za wszystkie uwagi dotyczące niniejszej wersji podręcznika. Prosimy je kierować pocztą elektroniczną na adres **`przykry@hydra.mimuw.edu.pl`**. Będziemy je uwzględniać w miarę możliwości w wersji elektronicznej podręcznika:

`http://hydra.mimuw.edu.pl/~przykry/petsc/podrecznik.html`

Aby korzystać z PETSc niezbędna jest znajomość w stopniu średnim programowania w języku C (podstawowe funkcje, wskaźniki, tablice i struktury) na poziomie standardu ANSI C. W zasadzie znajomość standardu Kernighan-Ritchie [12] też w zupełności wystarczy.

Podziękowanie

Chcielibyśmy podziękować prof. M.Dryji oraz mgr L.Marcinkowskiemu za sugestie i uwagi poczynione wobec wstępnej wersji niniejszego podręcznika. Podczas pracy nad redakcją podręcznika byliśmy częściowo finansowani z grantu KBN 2 P03A 009 09.

[1] Będą najwyżej sygnalizowane w przypisach

Wstęp

Załóżmy, że mamy do rozwiązania stacjonarne, eliptyczne równanie różniczkowe cząstkowe. Po dyskretyzacji metodą elementu skończonego lub różnic skończonych, prowadzi ono z reguły do wielkiego układu równań liniowych:

$$Ax = b$$

Aby rozwiązać powyższy układ przy pomocy komputera (być może równoległego) należy:

- Utworzyć wektory x i b i wpisać w wektor b odpowiednie wartości.
- Utworzyć i wpisać odpowiednie wartości w macierz A.
- Napisać procedury rozwiązywania takiego układu równań bądź metodą iteracyjną z preconditionerem, bądź bezpośrednią.
- Rozwiązać powyższy układ.

Wygląda to dosyć prosto, dopóki nie uzmysłowimy sobie, jakie problemy stoją przed programistą. Musi on, po pierwsze, zintegrować procedury rozwiązywania równań z pozostałą częścią programu. Po drugie, rozwiązać problem przechowywania danych w komputerze, z uwzględnieniem jego ewentualnej architektury równoległej. Po trzecie, wstawić wartości do macierzy i wektora prawej strony, uwzględniając rozproszenie danych po procesorach. Po czwarte, macierze pochodzące z dyskretyzacji równań różniczkowych mają z reguły strukturę rozrzedzoną, wobec czego trzeba rozwiązać problem efektywnego przechowywania takich macierzy w pamięci. Poza tym, dobrze byłoby móc testować i porównywać różne podejścia do problemu: zmieniać parametry, metody rozwiązywania układów równań, w szczególności - preconditionery. Program powinien wykorzystywać w pełni moc obliczeniową komputera, a dobrze, gdyby przy okazji był niezależny od platformy sprzętowej. Oczywiście, program powinien być wolny od błędów. Gdy spełnimy powyższe postulaty, okaże się, że program nabrał rozmiarów sporego pakietu. W tym przypadku mamy dwie możliwości - pisać samemu od podstaw własny pakiet lub wykorzystać istniejące oprogramowanie biblioteczne. Projekt PETSc jest rozwijany z myślą o osobach opowiadających się za tym drugim rozwiązaniem.

Inicjatorem projektu jest Barry Smith, który w 1991 roku napisał pierwszą wersję pakietu do rozwiązywania układów równań liniowych: PETSc 1.0. W ciągu kilku lat zestaw procedur przekształcił się w duży pakiet narzędzi do rozwiązywania równań różniczkowych cząstkowych, używany w wielu renomowanych ośrodkach obliczeniowych na świecie. Obecnie nad PETSc pracują Satish Balay, Willam Gropp, Lois Curfman McInnes i Barry Smith z Argonne National Laboratory (USA), wspomagani okazjonalnie przez grupy studentów - praktykantów. Akronim PETSc oznacza *Portable, Extensible Toolkit for Scientific computation*. Jak już wspomnieliśmy, pakiet jest w dużym stopniu niezależny od platformy sprzętowej. Programy można kompilować i uruchamiać na różnych architekturach, nie tracąc przy tym zbyt wiele z efektywności kodu, a jednocześnie wykorzystując potencjał konkretnego komputera. Obecnie obsługiwane architektury to:

- komputery osobiste klasy IBM PC, pracujące pod kontrolą systemów Linux, Windows 95 i NT 4.0;
- unixowe stacje robocze, z jednym lub kilkoma procesorami RISC: DEC Alpha, Silicon Graphics (także Origin i Power Challenge), Sun, IBM RS6000, itp.;
- sieć wyżej wymienionych komputerów połączonych Ethernetem;
- superkomputery masywnie równoległe (IBM SP2, Cray T3D/T3E, Convex Exemplar).

Zwróćmy uwagę, jak różne są te architektury! Osiągnięcie *pełnej*[2] przenośności kodu jest naprawdę imponującym dokonaniem. Można napisać program, którego poprawność przetestujemy na małym komputerze używając małego zadania, a potem uruchomimy dla poważnych obliczeń na superkomputerze. Poza tym mniejsze stacje robocze mają zwykle przyjaźniejszy interfejs użytkownika i narzędzia do pisania programów niż superkomputery. Podkreślają to także autorzy pakietu, którzy superkomputera (Cray T3E) używają jedynie po przetestowaniu kodu na stacjach roboczych Sun Sparcstation.

Drugim przymiotnikiem z nazwy jest Extensibile (rozszerzalny) - autorzy położyli duży nacisk na to aby pakiet był modularny i można było także wykorzystywać, ostatnio bardzo modne, programowanie obiektowe. *„PETSc jest dowodem na to, że nowoczesne metody programowania mogą ułatwić rozwój wielkich aplikacji numerycznych w Fortranie, C i C++. Oprogramowanie, które tworzymy od kilku lat, rozwinęło się w potężny pakiet narzędzi do numerycznego rozwiązywania równań różniczkowych cząstkowych na superkomputerach”* [2]. Warto dodać, że PETSc na pewno będzie dalej rozwijane przez parę najbliższych lat, jako że autorom pakietu przyznano grant rządowy. Każdy użytkownik pakietu może zwrócić się do autorów o pomoc, ponadto ma do dyspozycji listę dyskusyjną oraz sporo

[2] Pełną przenośność kodu otrzymujemy między komputerami pracującymi pod kontrolą systemu operacyjnego UNIX. W przypadku Windows NT/95 mamy możliwość skompilowania i uruchomienia programu z zastrzeżeniem, że pewne procedury pomocnicze, np rysowanie wektora lub macierzy nie będą działały.

dodatkowych wydawnictw, głownie w formie elektronicznej. Szczegóły można znaleźć na stronie WWW poświęconej PETSc w Argonne National Laboratory: `http://www.mcs.anl.gov/petsc/`

Struktura PETSc

Pakiet PETSc jest zbiorem bibliotek napisanych w języku C. Programy używające PETSc można pisać w języku C (także C++) lub w Fortranie. Do zrozumienia zasad posługiwania się PETSc wystarczy w zasadzie podstawowa znajomość programowania w języku C. Funkcje PETSc wykorzystują, w sposób transparentny dla użytkownika, znane i popularne biblioteki numeryczne, takie jak BLAS [17] (podstawowe procedury obsługi wektorów i macierzy gęstych, zoptymalizowane pod kątem współczesnych architektur komputerowych), LAPACK [1] (nowoczesny pakiet algebry liniowej dla macierzy gęstych, stanowiący rozwinięcie powszechnie stosowanego LINPACKa) oraz MPI [9] (standaryzowany zestaw procedur ułatwiających programowanie równoległe, implementowany na większości masywnie równoległych komputerów, a także dla sieci stacji roboczych).

PETSc zawiera wiele składników podobnych do klas w C++. Każdy składnik dysponuje wydzieloną rodziną typów danych (na przykład wektorów) i operacji wykonywalnych na nich. Zarówno obiekty, jak operacje, opracowano na podstawie wieloletniego doświadczenia autorów (i użytkowników pakietu!) w rozwiązywaniu równań różniczkowych cząstkowych. Moduły PETSc operują obiektami takimi jak:

- wektory,
- macierze,
- iteracyjne i bezpośrednie solvery[3] układów równań liniowych,
- preconditionery dla metod iteracyjnych,
- solvery zadań nieliniowych (na przykład metoda Newtona) oraz solvery zadań ewolucyjnych,
- urządzenia graficzne takie jak okna czy strony,
- zbiory indeksów, w tym permutacje,
- tablice rozproszone, użyteczne przy programowaniu równoległym.

Weźmy na przykład typowy algorytm rozwiązywania eliptycznego równania różniczkowego cząstkowego. Okazuje się, że prawie na każdym jego kroku możemy wykorzystać PETSc.

Krok algorytmu	Wykorzystanie PETSc
1. Generuj triangulację	
2. Generuj macierze sztywności	Obiekty i operacje macierzowe (`Mat`)
3. Generuj prawą stronę równania i przybliżenie początkowe	Obiekty i operacje wektorowe (`Vec`)
4. Obejrzyj utworzone wektory i macierze	Przeglądarki obiektów PETSc (`Viewer`)
5. Generuj preconditioner	Obiekty i operacje macierzowe i wektorowe
6. Rozwiąż	Narzędzia do rozwiązywania równań liniowych i nieliniowych (`SLES, SNES`). Obiekty kontrolujące przebieg metod iteracyjnych (`KSP, PC`). Opcje wywołania i kontroli (`Options`).
7. Obejrzyj wyniki	

Każdy ze składników PETSc ma własny interfejs (zbiór wywołań procedur) i jedną lub więcej implementacji. Dzięki temu PETSc zapewnia przejrzysty i efektywny kod na różnych etapach rozwiązywania równań różniczkowych cząstkowych. Gwarantuje także łatwość testowania różnych podejść do rozwiązywanego problemu, to znaczy zastosowania konkretnych algorytmów czy preconditionerów.

W PETSc wszystkie funkcje nazywane są zgodnie z konwencją **`TypObiektuCzynność()`**, na przykład:

`MatCreate()` - generuje macierz
`MatMult()` - mnoży dwie macierze
`VecAXBY()` - wykonuje na wektorach operację $\alpha x+\beta y$
`OptionsGetInt()` - pobiera z opcji zmienną typu `int`

[3] to znaczy: „procedury służące do rozwiązywania..."; nie znamy dobrze brzmiącego polskiego odpowiednika słowa *solver*.

Nazwy są długie, co nie oszczędza nam stukania w klawiaturę, ale za to większość nazw funkcji jest zrozumiała i łatwa do zapamiętania, co wspomaga pisanie i czytanie programów korzystających z biblioteki PETSc.

Jak czytać ten podręcznik?

Wyszliśmy z założenia, że poznawanie PETSc należy zacząć od prostych przykładów i nieskomplikowanych zagadnień. Dlatego ograniczyliśmy się do wersji sekwencyjnej tej biblioteki, skupiając się na podstawowych problemach spotykanych w praktyce numerycznej: rozwiązywaniu układów równań.

Podręcznik składa się więc z dwóch części: w pierwszej wykładamy podstawy użytkowania PETSc i w oparciu o nie pokazujemy, jak można wykorzystać PETSc do rozwiązywania wielkich układów równań *liniowych* za pomocą nowoczesnych metod iteracyjnych. W drugiej części omawiamy bardziej zaawansowane techniki oraz metody wykorzystania PETSc do rozwiązywania układów równań *nieliniowych*. Obie części bogato ilustrujemy pouczającymi przykładami, związanymi najczęściej z eliptycznymi równaniami różniczkowymi cząstkowymi.

Wydaje się więc, że podręcznik ten trzeba czytać w sposób, hmm... monotoniczny, to znaczy „od początku do końca". Zachęcamy Czytelnika do pisania własnych programów i do testowania świeżo zdobytej wiedzy. W szczególności, dla dobrego wykonania ćwiczeń, którymi kończymy większość rozdziałów, warto jest zawsze zerknąć nie tylko do naszego podręcznika, ale też do *man pages* oraz oficjalnego *manuala*.

Nie da się ukryć, aby nauczyć się korzystać z PETSc, trzeba włożyć trochę wysiłku. Przede wszystkim, trzeba zacząć pisać programy. Dlatego zachęcamy do lektury poniższego pouczającego rozdziału.

Zaczynamy!

Poniżej przedstawiony jest najprostszy program używający PETSc, będący mutacją klasycznego przykładu „Hello, world!", pojawiającego się na początku wielu podręczników programowania:

`program1.c`

```
#include "sles.h"
static char help[]="Najprostszy program uzywajacy PETSc\n";

int main(int argc,char **args)
{
 PetscInitialize(&argc,&args,PETSC_NULL,help);
 PetscPrintf(MPI_COMM_WORLD,"PETSc działa i pozdrawia\n");
 PetscFinalize();
}
```

Uruchomienie powyższego programu spowoduje, jak łatwo się domyślić, wypisanie na konsolę tekstu:

```
hydra:/usr/home/students/bojko> program1
PETSc działa i pozdrawia
hydra:/usr/home/students/bojko> _
```

Dzięki deklaracji funkcji `main()` z argumentami `argc`, `args` oraz wywołaniu funkcji `PetscInitialize()`, tekst z linii komend zostanie przekazany bezpośrednio do programu. Dzięki temu, będziemy mogli używać rozmaitych parametrów wywołania programu (tzw. opcji), sterując wieloma parametrami PETSc i samego programu, bez konieczności nieustannego rekompilowania go.

Mimo, że kod źródłowy jest bardzo krótki, skompilowanie tego programu spowoduje wygenerowanie dość dużego pliku wynikowego, a to dlatego, że do każdego programu dołączany jest szereg procedur ułatwiających życie programistom i użytkownikom - chociażby zestaw procedur do śledzenia pracy programu. Na przykład, uruchomienie tego samego programu z opcją `-help` spowoduje wypisanie:

```
hydra: /usr/home/students/bojko> program1 -help
------------------------------------------------------------------------
PETSc Version 2.0.22, Released April 28, 1998.
The PETSc Team: Satish Balay, Bill Gropp, Lois Curfman McInnes, Barry Smith
 Bug reports, questions: petsc-maint@mcs.anl.gov
 Web page: http://www.mcs.anl.gov/petsc/
See docs/copyright.html for copyright information
See docs/changes.html for recent updates.
See docs/troubleshooting.html hints for problems.
See docs/manualpages/manualpages.html for help.
Libraries linked from /usr/local/petsc/petsc-2.0.22/lib/libO/solaris
------------------------------------------------------------------------
Options for all PETSc programs:
 -on_error_abort: cause an abort when an error is detected. Useful
       only when run in the debugger
```

```
 -on_error_attach_debugger [dbx,xxgdb,ups,noxterm]
       start the debugger (gdb by default) in new xterm
       unless noxterm is given
 -start_in_debugger [dbx,xxgdb,ups,noxterm]
       start all processes in the debugger
 -debugger_nodes [n1,n2,..] Nodes to start in debugger
 -debugger_pause [m] : delay (in seconds) to attach debugger
 -display display: Location where graphics and debuggers are displayed
 -no_signal_handler: do not trap error signals
 -mpi_return_on_error: MPI returns error code, rather than abort on internal
error
 -fp_trap: stop on floating point exceptions
           note on IBM RS6000 this slows run greatly
 -trdump: dump list of unfreed memory at conclusion
 -trmalloc: use our error checking malloc
 -trmalloc_off: don't use error checking malloc
 -trmalloc_nan: initialize memory locations with NaNs
 -trinfo: prints total memory usage
 -trdebug: enables extended checking for memory corruption
 -optionstable: dump list of options inputted
 -optionsleft: dump list of unused options
 -log[_all _summary]: logging objects and events
 -log_summary_exclude: <vec,mat,sles,snes>
 -log_trace [filename]: prints trace of all PETSc calls
 -log_info: print informative messages about the calculations
 -v: prints PETSc version number and release date
 -options_file <file>: reads options from file
------------------------------------------------
Najprostszy program uzywajacy PETSc
PETSc działa i pozdrawia
hydra:/usr/home/students/bojko> _
```

Widać więc, że opcja `-help` powoduje wypisanie własnych komunikatów PETSc dotyczących opcji dostępnych w składnikach pakietu używanych w naszym programie oraz naszego komunikatu ze stałej `help`. Wszystkie procedury udostępniane przez pakiet działają tylko między funkcjami `PetscInitialize()` i `PetscFinalize()`, więc część programu korzystająca z PETSc powinna zawierać się między nimi.

Zanim rozpoczniemy nasze bliskie spotkanie z PETSc, jeszcze krótko omówimy sposób kompilowania i uruchamiania programów korzystających z biblioteki PETSc[4]. Po pierwsze aby móc skompilować swój program należy ustawić zmienne środowiskowe `PETSC_DIR, PETSC_ARCH, BLAS_DIR, LAPACK_DIR`[5] jak następuje:

```
setenv PETSC_DIR /usr/local/petsc/petsc-2.0.22
setenv PETSC_ARCH solaris
setenv BLAS_DIR /usr/local/petsc/blaslapack
setenv LAPACK_DIR /usr/local/petsc/blaslapack
```

Najlepiej umieścić powyższe dyrektywy w pliku `.cshrc` w katalogu użytkownika. Poniżej podany jest plik `Makefile`, który musi znaleźć się w tym samym katalogu co plik źródłowy `program1.c`, dzięki któremu program będzie można skompilować używając polecenia `make program1`. Szczegóły dotyczące uruchamiania programów można znaleźć w rozdziale „Uruchamianie programów” na stronie 25.

Makefile

```
BOPT=O
include $(PETSC_DIR)/bmake/$(PETSC_ARCH)/base

CFLAG           = $(CPPFLAGS) -D__SDIR__='"$(LOCDIR)"' $(BS_INCLUDE)
LIBBASE         = libpetscsles
LOCDIR          = /usr/home/students/bojko/source/

program1:  program1.o
           -$(CLINKER) -o program1 program1.o  $(PETSC_LIB)
            $(RM) program1.o
```

[4] Omówimy tu sposób wykorzystania PETSc 2.0.22 na komputerze **`hydra.mimuw.edu.pl`**. Ponieważ instalacje (i wersje) mogą różnić się od siebie detalami należy skonsultować się z lokalnym administratorem lub lokalnym *guru* PETSc (często będą to różne osoby).

[5] Dokładny opis tych zmiennych można znaleźć w odpowiednim pliku w katalogu głównym PETSc, w przypadku `hydry.mimuw.edu.pl`. jest to katalog `/usr/local/petsc/petsc-XXX`, gdzie XXX jest numerem aktualnej wersji PETSc.

Oczywiście, parametr `LOCDIR` powinien być ustawiony na ścieżkę z Twoimi plikami źródłowymi.

Ćwiczenia

A) W katalogu `$PETSC_DIR/docs` znajdują się: *manual* [2] oraz *man pages* [3] dla PETSc, w formacie HTML. Odnajdź te pliki i przejrzyj ich zawartość przy użyciu jakiejś przeglądarki do stron WWW (np. Netscape). Będziesz ich bardzo często potrzebować, więc warto.

B) Ten podręcznik też ma swoje wcielenie internetowe. Łatwo stamtąd ściągniesz omawiane tu kody źródłowe i Makefile. Ponadto, właśnie tam będziemy umieszczać wszelkie poprawki i uaktualnienia. Zobacz `http://hydra.mimuw.edu.pl/~przykry/petsc/podrecznik.html`

C) Skompiluj i uruchom przykładowy program używający biblioteki PETSc. W przypadku kłopotów skontaktuj się z jakimś bardziej doświadczonym użytkownikiem pakietu. Warto poświęcić chwilę czasu na ustawienie (raz, a dobrze!) zmiennych środowiskowych i napisanie własnego `Makefile`. Zaprocentuje to w przyszłości!

D) Na przykładzie programu „PETSc działa i pozdrawia” zobacz, jak działają inne opcje PETSc, w szczególności: `-log`, `-v`, `-trinfo`.

I. Rozwiązywanie układów równań liniowych

W tej części zostanie omówiony sposób rozwiązywania układów algebraicznych równań liniowych przy użyciu pakietu PETSc. Na początku opisane zostaną podstawowe struktury w PETSc: skalary, wektory i macierze, potrzebne do zbudowania zagadnienia. Potem wprowadzony zostanie SLES - obiekt PETSc, pozwalający na rozwiązanie układu równań liniowych $Ax = b$. Pozwoli to na uruchomienie prostego programu rozwiązującego układ równań powstałych z dyskretyzacji zagadnienia Poissona

$$\begin{cases} -\Delta u(x,y) = f(x,y) & \text{dla } (x,y) \in \Omega = (0,1)^2 \\ u(x,y) = 0 & \text{dla } (x,y) \in \partial\Omega \end{cases}$$

metodą różnic skończonych.

Następnie omówione zostaną szczegółowo struktury PETSc pozwalające na wybór metod iteracyjnych i włączanie w nie preconditionerów (także własnych). Ostatni rozdział poświęcony zostanie na szczegółowy opis istniejącego w pakiecie preconditionera addytywnej metody Schwarza, wykorzystującego metodę dekompozycji obszaru.

1. Skalary

Styl rozwiązywania układów równań w pakiecie PETSc jest ściśle związany z intuicjami pochodzącymi z algebry liniowej, stąd definiowanie obiektów rozpoczyna się od podania „ciała" skalarów. Niniejszy podręcznik ograniczy się jedynie do zmiennych typu `double`[6] - tak jest standardowo zdefiniowany w PETSc typ `Scalar` (w pliku "`petsc.h`"). Nie jest to oczywiście ciało w ściśle algebraicznym sensie tego słowa, ale podobieństwo pojęcia jest zrozumiałe.

2. Wektory

2.1 Podstawy

Wektor, podobnie jak skalar, jest w PETSc typem danych. Do manipulacji wektorem używa się określonych funkcji dostępu. Aby otrzymać obiekt typu wektor, należy go najpierw *zadeklarować,* a potem *wygenerować*. Deklaracja jest prosta, PETSc zawiera typ `Vec`, zatem deklaracja:

```
...
Vec X;
...
```

spowoduje zadeklarowanie `X` jako zmiennej typu wektorowego.
Najprostszą funkcją służącą do wygenerowania wektora jest funkcja:

`VecCreate(MPI_Comm comm,int n, int N,Vec *X)`

Pierwszy argument to informacja dla MPI (Message Passing Interface) o sposobie komunikacji między procesami. Nie warto zagłębiać się w subtelności z tym związane. Ważne jest, że z argumentem `MPI_COMM_WORLD` funkcja działa poprawnie zarówno na maszynach wieloprocesorowych, jak i jednoprocesorowych i we wszystkich podobnych przypadkach rozważanych w tym podręczniku będzie używana ta właśnie wartość. Drugi argument, `n`, jest liczbą elementów wektora przechowywanych w procesorze, który wywołał funkcję; możemy podać wartość `PETSC_DECIDE`, gdy chcemy aby pakiet ustawił tę wielkość automatycznie. `PETSC_DECIDE` jest idealną wartością dla `n`, gdy program nasz ma działać przede wszystkim w wersji sekwencyjnej. Trzeci argument, `N`, jest liczbą współrzędnych całego wektora (można wstawić `PETSC_DETERMINE`, wtedy zostanie ona określona na podstawie liczby procesorów i wartości `n`). Ostatnim argumentem jest wskaźnik do zmiennej typu wektor. Wektor będzie składał się ze współrzędnych typu `Scalar`.

```
...
VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,100,&X);
...
```

Takie wywołanie funkcji `VecCreate()` spowoduje utworzenie wektora o 100 współrzędnych. Zwróćmy uwagę, że taki sposób tworzenia wektorów: osobno deklaracja, osobno wygenerowanie obiektu nie jest niczym nowym w języku C. Zwyczajne tablice w C można inicjalizować na dwa sposoby. Pierwszy sposób to podczas deklarowania zmiennych np. :

[6] Może to być także zmienna typu `float` lub `complex`.

```
...
double x[100];
...
```

a drugi sposób to dynamicznie:

```
...
double *x;
...
x=(double*)calloc(100,sizeof(double));
...
```

Sposób deklarowania wektorów (i nie tylko) w PETSc jest podobny do tej drugiej metody. Faktyczne przydzielenie pamięci następuje w momencie wywołania funkcji `VecCreate()`.

Funkcja `VecCreate()` jest zaprojektowana tak, aby automatycznie rozproszyć wektor pomiędzy dostępne procesory (procesy). Pakiet sam zadecyduje czy wygenerować wektor sekwencyjny, czy równoległy, a jeśli równoległy, to ile elementów wektora będzie przechowywanych w poszczególnych procesorach.[7]

Po wygenerowaniu wektora można przypisać wartości poszczególnym współrzędnym. Służą do tego dwie funkcje: **`VecSet(Scalar *val, Vec X)`** która nada wszystkim współrzędnym jednakową wartość `val` lub funkcja **`VecSetValue(Vec X, int idx, Scalar val, InsertMode mode)`**. Pierwszy argument tej funkcji to wektor w który wstawiamy wartość, drugi to numer współrzędnej, trzeci wartość na tej współrzędnej. Czwarty argument to tryb wstawiania wartości, Możliwe są dwie wartości: `ADD_VALUES` - dodaj wartość do istniejącej, `INSERT_VALUES` - wstaw nową wartość.

2.2 Przykład

W tym przykładzie utworzony zostanie wektor $x = (x_0,...,x_9)^T$, w który zostaną wstawione wartości

$$x_i = \begin{cases} \frac{1}{i} \ dla \ i > 0, \\ 1 \ dla \ i = 0, \end{cases}$$

a na końcu wektor zostanie wypisany na konsolę.

```
#include "sles.h"
static char help[]="przyklad2\n";

int main(int argc,char **args)
{
  Vec     x;
  Scalar val;
  int i,n=10;

  /* inicjalizacja */
  PetscInitialize(&argc,&args,PETSC_NULL,help);

  /* wygenerowanie wektora */
  VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,n,&x);

  /* przypisanie wartosci */
  for(i=0;i<n;i++)
  {
   if(i>0)val=(Scalar)1.0/i; else val=1.0;
   VecSetValue(x,i,val,INSERT_VALUES);
  }

  /* wypisuje wektor na konsolę */
  VecView(x,VIEWER_STDOUT_SELF);


  /* zakończenie */
```

To dostaniemy po uruchomieniu przykładu:

```
hydra:/usr/home/bin/> przyklad2
1
1
0.5
0.333333
0.25
0.2
0.166667
0.142857
0.125
0.111111
hydra:/usr/home/bin/> _
```

[7] Dostępne są też dwie funkcje, dzięki którym użytkownik może zadecydować jakiego typu wektory zostaną wygenerowane: `VecCreateSeq()` lub `VecCreateMPI()`. Dzięki drugiej z nich można ręcznie ustawić jak wektor będzie rozproszony pomiędzy procesory.

```
 VecDestroy(x);
 PetscFinalize();
 return(0);
}
```

Chyba warto odrobinę skomentować ten prościutki programik. Jak łatwo się domyślić, plik nagłówkowy `sles.h` zawiera wszystkie potrzebne definicje funkcji i obiektów PETSc, potrzebne w naszym programie. W rzeczywistości, pozwala on także na wykorzystanie całej funkcjonalności PETSc dotyczącej rozwiązywania równań liniowych.

Po zadeklarowaniu, wygenerowaniu i przypisaniu wartości wektorowi `x`, na zakończenie programu wypisujemy go na ekran przy użyciu przeglądarki do wektorów `VecView()`.

2.3 Co jeszcze warto wiedzieć

W przypadku, gdy do wektora trzeba wstawić wiele różnych wartości, autorzy pakietu zalecają używanie funkcji **`VecSetValues(Vec x,int n,int *ind,int *val,InsertMode mode),`** która działa znacznie szybciej niż `VecSetValue()`. Funkcja `VecSetValues()` wstawia w wektor `x` `n` wartości z tablicy `val` w miejsca o indeksach zawartych w tablicy `ind`. Ostatni argument ma takie samo znaczenie jak w przypadku funkcji `VecInsertValue()`. Po wywołaniu funkcji `VecSetValues()` wartości, które mają zostać wstawione, zostają zbuforowane. Proces fizycznego wstawiania wartości w wektory rozpoczyna się dopiero po wywołaniu funkcji **`VecAssemblyBegin(Vec x)`**, a kończy po wywołaniu funkcji **`VecAssemblyEnd(Vec x)`**. Zostało to tak zaprojektowane, aby można było przeplatać obliczenia z przekazywaniem informacji między procesami. Między tymi dwoma funkcjami można wstawić dowolny ciąg instrukcji. Nie można jedynie mieszać dodawania wartości ze wstawianiem.

Aby wygenerować kilka takich samych wektorów (na przykład wektor rozwiązania równania i wektor prawej strony - oba są tego samego rozmiaru) można użyć funkcji **`VecDuplicate(Vec stary,Vec *nowy)`** lub **`VecDuplicateVecs(Vec stary,int n, Vec **nowe)`** pierwsza z tych funkcji utworzy wektor `nowy` o takiej samej strukturze jak wektor `stary`, to znaczy o tym samym rozmiarze i tak samo rozproszony pomiędzy procesorami. Druga utworzy `n` nowych wektorów o takiej samej strukturze jak `stary` i zapisuje w *tablicy* wektorów `nowe`. Powielenia struktury wektora nie należy mylić z kopiowaniem jego wartości. Do tego służy funkcja **`VecCopy(Vec x,Vec y)`**, kopiująca wartości z wektora `x` do `y`. Wektory `x` i `y` musza w tym wypadku mieć taką samą strukturę, co znaczy, że uprzednio muszą być utworzone, na przykład przy pomocy funkcji `VecCreate()` lub `VecDuplicate()`.

Nie istnieje, niestety, funkcja `VecGetValues()`, która pozwalałaby na pobranie wartości z wektora. W przypadku równoległym trzeba stosować specjalne funkcje zbierające i rozsyłające dane po procesorach. W przypadku sekwencyjnym można używać funkcji **`VecGetArray(Vec x,Scalar **val)`**, która zwraca wskaźnik do tablicy elementów wektora przechowywanych w procesorze[8] który wywołał tę procedurę. Po zakończeniu wszelkich operacji na tablicy elementów trzeba ją z powrotem wstawić do wektora funkcją **`VecRestoreArray(Vec x, Scalar **val)`**.

Przydatna jest też funkcja **`VecView(Vec v, Viewer viewer)`**, która pokazuje wektor `v` przy pomocy przeglądarki `viewer`. Istnieją trzy predefiniowane przeglądarki:

Viewer	OPIS
`VIEWER_STDOUT_SELF`	standardowa tekstowa; wypisuje wektor na konsolę
`VIEWER_STDOUT_WORLD`	standardowa tekstowa zsynchronizowana, przydatna w przypadku programów równoległych
`VIEWER_DRAWX_WORLD`	graficzna; rysuje wektor w oknie graficznym jako funkcję współrzędnej

Przeglądarki służą do oglądania obiektów w PETSc dla prawie każdego obiektu istnieje funkcja **`XXXView(XXX x,Viewer viewer)`**, gdzie `XXX` to typ obiektu (np. `Vec`, `Mat` itp.)[9]

Gdy wektor nie jest już potrzebny, należy zwolnić pamięć którą zajmuje funkcją **`VecDestroy(Vec x)`** lub - w przypadku tablicy wektorów - **`VecDestroyVecs(Vec *wektory,int n)`**.

[8] Dokładnie w procesie, na jednym procesorze może działać kilka procesów.

[9] Z tym zastrzeżeniem, że nie wszystkie obiekty można obejrzeć w przeglądarce graficznej.

2.4 Podstawowe operacje wektorowe

Ponieważ wektory są złożonymi obiektami, do wykonania nawet najprostszej operacji na wektorze będziemy używali specjalnych funkcji. Przykładowo, aby dodać/odjąć dwa wektory będziemy musieli skorzystać z funkcji `VecAXPY()` lub `VecWAXPY()`. Podobnie będzie, gdy zechcemy przekopiować jeden wektor na drugi. Poniżej przedstawiamy wybór funkcji, realizujących podstawowe operacje wektorowe.

NAZWA FUNKCJI	OPERACJA
VecAXPY(Scalar *a,Vec x,Vec y**);**	$\mathbf{y} \leftarrow a \cdot \mathbf{x} + \mathbf{y}$
VecAYPX(Scalar *a,Vec x,Vec y**);**	$\mathbf{y} \leftarrow \mathbf{x} + a \cdot \mathbf{y}$
VecWAXPY(Scalar *a,Vec x,Vec y,Vec w**);**	$\mathbf{w} \leftarrow a \cdot \mathbf{x} + \mathbf{y}$
VecScale(Scalar *a,Vec x**);**	$\mathbf{x} \leftarrow a \cdot \mathbf{x}$
VecDot(Vec x,Vec y,Scalar *r**);**	$r \leftarrow \overline{\mathbf{x}}^T \cdot \mathbf{y}$
VecTDot(Vec x,Vec y,Scalar *r**);**	$r \leftarrow \mathbf{x}^T \cdot \mathbf{y}$
VecNorm(Vec x,NormType norm,Double *r**);** norm ∈ {NORM_1,NORM_2,NORM_INFINITY}	$r \leftarrow \|\mathbf{x}\|_{norm}$
VecSum(Vec x, Scalar *r**);**	$r \leftarrow \sum x_i$
VecCopy(Vec x, Vec y**);**	$\mathbf{y} \leftarrow \mathbf{x}$
VecSwap(Vec x, Vec y**);**	$\mathbf{x} \leftrightarrow \mathbf{y}$
VecPointwiseMult(Vec x, Vec y, Vec w**);**	$w_i \leftarrow x_i * y_i$
VecPointwiseDivide(Vec x, Vec y, Vec w**);**	$w_i \leftarrow x_i / y_i$
VecMax(Vec x, int *idx, double *r**);**	$r \leftarrow \max x_i$
VecMin(Vec x, int *idx, double *r**);**	$r \leftarrow \min x_i$
VecAbs(Vec x**);**	$x_i \leftarrow \|x_i\|$
VecReciprocal(Vec x**);**	$x_i \leftarrow 1 / x_i$
VecShift(Scalar *s, Vec x**);**	$x_i \leftarrow s + x_i$

2.5 Podsumowanie

Powtórzmy zatem, co trzeba zrobić, aby posługiwać się obiektem typu wektor:

- Zadeklarować zmienną typu **`Vec`**
- Utworzyć wektor funkcją **`VecCreate()`**
- Wstawić wartości funkcją **`VecSetValues()`**, po czym wywołać funkcje **`VecAssemblyBegin()`** i **`VecAssemblyEnd()`**
- Po użyciu wektora, zwolnić zajmowaną przezeń pamięć funkcją **`VecDestroy()`**

2.6 Przykład podsumowujący

```
static char help[]="przyklad 4.6";
#include "sles.h"
#include <stdio.h>

int main(int argc,char **args)
{
  Vec     x,y;
```

```
  Scalar  val,*wartosci;
  int     i,n=5,*indeksy,flg;

  PetscInitialize(&argc,&args,PETSC_NULL,help);

  VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,n,&x);
  VecDuplicate(x,&y);
  indeksy=(int*)calloc(n/2,sizeof(int));
  wartosci=(Scalar*)calloc(n/2,sizeof(Scalar));
  val=0.5;
  VecSet(&val,x);
  VecView(x,VIEWER_STDOUT_SELF);
  PetscPrintf(MPI_COMM_WORLD,"\n");
  VecCopy(x,y);

/* przygotowanie danych */
  for(i=0;i<n/2;i++)
  {
   if(i>0)wartosci[i]=(Scalar)1/i;
   else wartosci[i]=1;
   indeksy[i]=2*i;
  }

/* Wstawienie danych do wektora */
  VecSetValues(y,n/2,indeksy,wartosci,ADD_VALUES);
  VecAssemblyBegin(y);
  VecAssemblyEnd(y);
  VecView(y,VIEWER_STDOUT_SELF);
  PetscPrintf(MPI_COMM_WORLD,"\n");

/* policzenie różnicy dwóch wektorów */
  val=-1;
  VecAXPY(&val,x,y);

/*wypisanie wektorów */
  VecView(y, VIEWER_STDOUT_SELF);
  PetscPrintf(MPI_COMM_WORLD,"\n");

/*zakończenie*/
  VecDestroy(x);
  VecDestroy(y);
  free(indeksy);
  free(wartosci);
  PetscFinalize();
  return(0);
 }
```

To dostaniemy po uruchomieniu przykładu:

```
hydra:/usr/home/> przyklad
0.5
0.5
0.5
0.5
0.5

1.5
0.5
1.5
0.5
0.5

1
0
1
0
0

hydra:/usr/home/> _
```

Ten program tworzy dwa wektory `x,y` o długości 5 (jakie są ich współrzędne?). Następnie, kładziemy $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{x}$, a następnie wektory są wypisywane na konsolę. Zwróćmy uwagę, w jaki sposób do odejmowania wektorów używamy funkcji `VecAXPY()`: realizując operację $\mathbf{y} \leftarrow (-1)\cdot\mathbf{x} + \mathbf{y}$!

2.7 Ćwiczenia

A) Utwórz na dwa sposoby (`VecSetValue()`+ pętla lub `VecSet()`) wektor o 10 współrzędnych wszystkich równych a) 1, b) π, Obejrzyj te wektory na konsoli i w okienku graficznym. (Uwaga: w pliku `<math.h>` jest zdefiniowana stała `M_PI` zawierająca przybliżenie π)

B) Utwórz wektor *x* o *N* współrzędnych równych $x_i = \sin \pi \frac{i}{N-1}$, $i = 0,...,N-1$. (Zauważ, że taki wektor można potraktować jako dyskretyzację funkcji $\sin \pi x$ na odcinku [0,1]). Obejrzyj ten wektor na konsoli i w okienku graficznym. Eksperymentuj dla różnych wartości *N*.

C) Utwórz wektor *y* o *N*x*N* współrzędnych $y_{i+N\cdot j} = \sin\left(\pi \frac{i}{N-1}\right)\cos\left(\pi \frac{j}{N-1}\right)$, $i, j = 0,...,N-1$. Przez analogię z po-

przednim można ten wektor traktować jako dyskretyzację funkcji $f(x,y) = \sin \pi x \cos \pi y$ na kwadracie $[0,1]^2$ z siatką o kroku *h = 1/(N-1)*. Obejrzyj *y* w okienku graficznym. Zobacz jak trudno zgadnąć, jaką funkcję dyskretyzuje *y*, stąd wniosek, że do takiej wizualizacji trzeba używać innych narzędzi. Dowiedz się, jakie programy wizualizacyjne są dostępne na Twoim komputerze i jak z nich korzystać.

D) Oblicz normy a) $\|\cdot\|_2$, b) $\|\cdot\|_1$, c) $\|\cdot\|_\infty$ wektorów *x* i *y* z poprzednich ćwiczeń. Jakich relacji między tymi normami można się spodziewać?

E) Zwróć uwagę, że dyskretna norma l_2 funkcji siatkowej reprezentowanej przez wektor *x* to odpowiednio przeskalowana norma euklidesowa tego wektora. Napisz procedurę, obliczającą normę l_2 wektora siatkowego *x* określonego na siatce o kroku *h* i *N* węzłach. Sprawdź na poprzednich przykładach, jak zachowuje się ta norma, gdy *N* rośnie. Porównaj to z normami: euklidesową i „max".

F) Wygeneruj wektor o długości 5 i losowych współrzędnych z przedziału [-1,1]. Będzie do tego potrzebna funkcja `PetscRandom()` której opis znajduje się w *man pages* [3].

G) Napisz funkcję `VecGetValues(Vec v,int n,int *ind, Scalar *val)`, która z wektora wyciąga `n` kolejnych wartości.

3. Macierze

Kolejną klasą obiektów, którą będziemy się zajmować, są macierze. PETSc operuje wieloma rodzajami macierzy, zarówno gęstych, jak i rozrzedzonych. Wszystkie rodzaje są ważne od strony implementacyjnej, ze względu na sposób przechowywania ich w pamięci. W tym podręczniku zostaną omówione pokrótce podstawowe typy macierzy przechowywanych w postaci tablic, oraz dodatkowo macierz powłokowa (ang. *shell matrix*): macierz zdefiniowana jako operator na wektorach bez podawania tablicy elementów (jak wiadomo, do wielu metod iteracyjnych nie trzeba jawnie podawać macierzy, a jedynie sposób w jaki macierz działa na wektorach).

Po zadeklarowaniu zmiennej jako macierz:

```
...
Mat A;
...
```

należy utworzyć, podobnie jak w przypadku wektorów, obiekt typu macierz. Robi się to przy pomocy funkcji: **`MatCreate(MPI_Comm comm,int m,int n,Mat *A)`**. Funkcja tworzy macierz `A` o rozmiarze `m`×`n`, gdzie `m` to liczba wierszy, a `n` kolumn. Wstawianie wartości do macierzy odbywa się analogicznie jak w przypadku wektorów, przy pomocy funkcji **`MatSetValue(Mat A,int row,int col,Scalar val,InsertMode mode)`**. Ta funkcja wpisuje w macierz `A` wartość `val` w miejsce `row,col`. Ostatni argument określa, podobnie jak w przypadku wektorów, sposób wstawiania wartości do macierzy i może przyjąć wartości `INSERT_VALUES` lub `ADD_VALUES`. Do zwolnienia pamięci zajmowanej przez macierz służy funkcja **`MatDestroy(Mat A)`**, która działa tak samo jak jej odpowiednik dla wektorów.

3.1 Przykład

W tym przykładzie zostanie utworzona, a potem pokazana w standardowej przeglądarce macierz 5x5:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}$$

```
#include "sles.h"

static char help[]="przykład_3\n";
int main(int argc,char **args)
{
  Mat     A;
  int i,n=5;
  /* inicjalizacja */
  PetscInitialize(&argc,&args,PETSC_NULL,help);
  /* wygenerowanie macierzy trójdiagonalnej*/
  MatCreate(MPI_COMM_WORLD,n,n,&A);
```

```
 /* wypełniamy macierz wierszami */
 for(i=0;i<n;i++)
 {
  MatSetValue(A,i,i,2,INSERT_VALUES);
  if(i<n-1)MatSetValue(A,i,i+1,1,INSERT_VALUES);
  if(i>0)MatSetValue(A,i,i-1,1,INSERT_VALUES);
 }
 MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
 MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);

 /* wypisuje macierz na konsolę */
 MatView(A,VIEWER_STDOUT_SELF);

 /* zakończenie */
 MatDestroy(A);
 PetscFinalize();
 return(0);
}
```

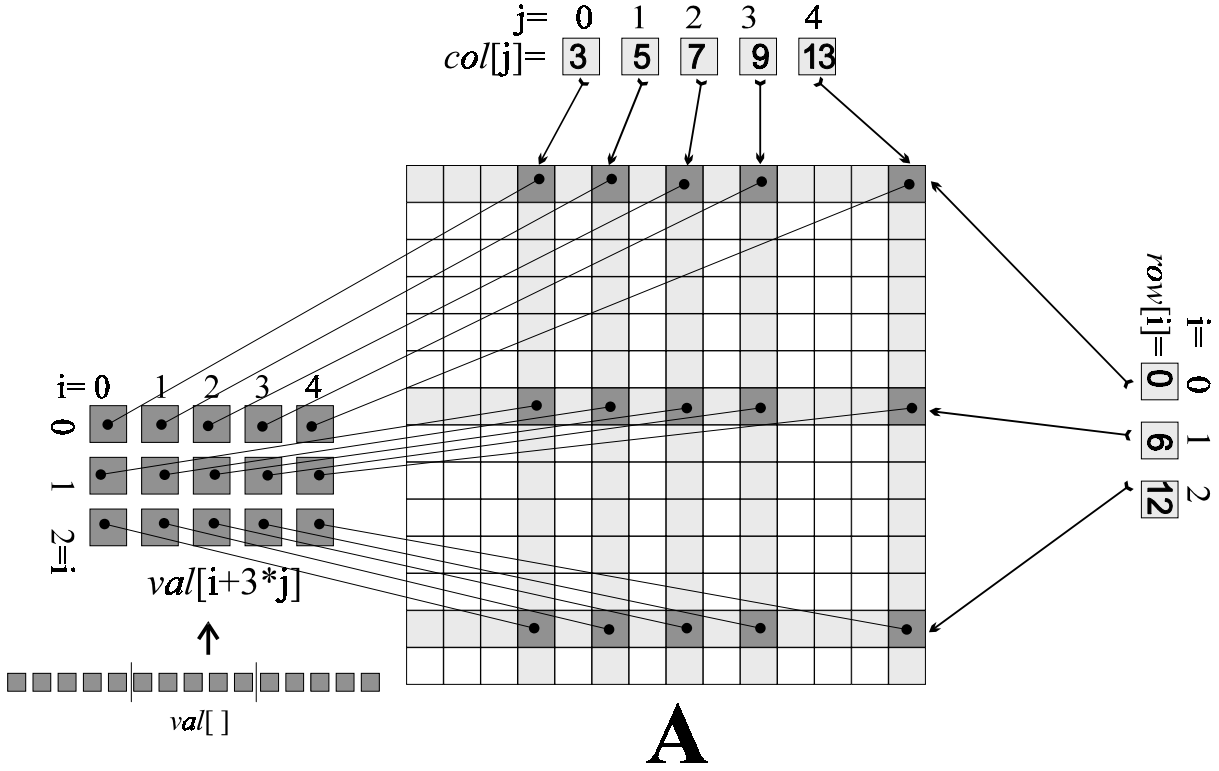
To dostaniemy po uruchomieniu przykładu:

```
hydra:/usr/home/bin/> przykład
row 0: 0 2  1 -1
row 1: 0 -1  1 2  2 -1
row 2: 1 -1  2 2  3 -1
row 3: 2 -1  3 2  4 -1
row 4: 3 1  4 2

hydra:/usr/home/bin/> _
```

Przyjrzyjmy się, jak działa przeglądarka macierzy `MatView()` w trybie tekstowym: macierz jest wypisywana wierszami. W każdym wierszu podawane są tylko niezerowe elementy (bardzo słusznie!); dlatego format wydruku jest następujący:

```
...
```



Rysunek 1. Działanie funkcji `MatSetValues()`.

```
row wiersz: kolumna wartość kolumna wartość kolumna wartość
....
```

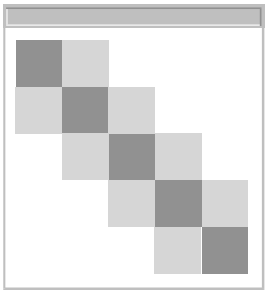
3.2 Co jeszcze warto wiedzieć

Podobnie jak w przypadku wektorów, autorzy pakietu zalecają wstawianie wartości do macierzy przy pomocy funkcji

`MatSetValues(Mat A,int m,int *row,int n,int *col,Scalar *val,InsertMode mode)`

Ta funkcja rozprasza w macierzy `A` macierz o rozmiarze `mxn` zapisaną w tablicy `val` w miejsca zapisane w tablicach `row` i `col`. Sposób działania tej funkcji wyjaśnia **Rysunek 1**. Aby wartości zostały fizycznie wpisane do macierzy należy, zawsze wywołać funkcje: **`MatAssemblyBegin(Mat A,MatAssemblyType type)`** i

MatAssemblyEnd(Mat A, MatAssemblyType type). Zmienna `type` może przyjąć dwie wartości: `MAT_FLUSH_ASSEMBLY` bądź `MAT_FINAL_ASSEMBLY`. Pierwsza wartość jest używana, gdy zmieniamy tryb wstawiania wartości (`INSERT_VALUES` na `ADD_VALUES`, bądź na odwrót), druga - gdy kończymy wstawianie wartości i macierz ma zostać użyta[10].



Rysunek 2. Macierz trójdiagonalna z poprzedniego przykładu w przeglądarce graficznej.

Funkcja `MatCreate()` tworzy szkielet macierzy rozrzedzonej pewnego (domyślnego) rodzaju. Użytkownik może też samemu wpłynąć na sposób generowania macierzy. Na przykład, można zaznaczyć jak duże kawałki macierzy mają być przechowywane w poszczególnych procesach, czy też, jak macierze mają być składowane: wierszami czy kolumnami. Szczegółowy opis można znaleźć w *manualu* [2] i *man pages* [3]. Dla potrzeb początkującego użytkownika przydatna jest jeszcze funkcja **MatCompress(Mat A)**, która poddaje macierz kompresji, tak aby zajmowała jak najmniej miejsca w pamięci, gdyż funkcja `MatCreate` alokuje 10 razy ilość wierszy macierzy, co z reguły jest za dużo. Warto zatem po zakończeniu wpisywania wartości i wywołaniu funkcji `MatAssemblyEnd()`, wywołać funkcję `MatCompress()`[11]. W przypadku gdy wiemy, że macierz zajmie więcej miejsca niż domyślne „10 razy ilość wierszy" trzeba użyć funkcji `MatCreateSeqAIJ()`, lub podobnej, ponieważ alokowanie dodatkowego miejsca w czasie wstawiania wartości do macierzy może spowodować nawet 50 krotne (!) spowolnienie działania funkcji.

Jak już wspomnieliśmy, dla macierzy istnieje funkcja **MatView(Mat A,Viewer view)**, która pokazuje macierz w przeglądarce. Argument `view` może m.in. przyjmować te same wartości co w przypadku funkcji `VecView()`. Na przykład, `MatView(A, VIEWER_DRAWX_WORLD)` pokazuje, w okienku graficznym, strukturę niezerowych miejsc macierzy. **Rysunek 2** pokazuje efekt przeglądania w trybie graficznym macierzy z poprzedniego przykładu.

3.3 Podstawowe operacje macierzowe

NAZWA FUNKCJI	OPERACJA
MatEqual(`Mat A,Mat B`**)**	sprawdza czy **A=B**
MatMult(`Mat A, Vec x, Vec y`**)**	$y = \mathbf{A} \cdot x$
MatMultAdd(`Mat A, Vec x,Vec y, Vec z`**)**	$z = \mathbf{A} \cdot x + y$
MatMultTrans(`Mat A, Vec x,Vec y`**)**	$y = \mathbf{A}^T \cdot x$
MatMultTransAdd(`Mat A, Vec x,Vec y, Vec z`**)**	$z = \mathbf{A}^T \cdot x + y$

3.4 Podsumowanie

Aby utworzyć macierz i wpisać w nią wartości należy:

- Zadeklarować zmienną typu **Mat**
- Utworzyć strukturę macierzy funkcją **MatCreate()** (lub inną funkcją, pozwalającą na podanie przypuszczalnego współczynnika zapełnienia macierzy).
- Wstawić do macierzy wartości funkcją **MatSetValues()**
- Wywołać funkcje **MatAssemblyBegin()** i **MatAssemblyEnd()** (z opcją `FINAL_ASSEMBLY`)
- Gdy macierz nie jest już potrzebna, zwolnić pamięć funkcją **MatDestroy()**

[10] Jeżeli na macierzy nie została wykonana operacja `FINAL_ASSEMBLY` to próba jej wykorzystania w programie skończy się komunikatem o błędzie.

[11] Można także użyć funkcji **MatCreateSeqAIJ(MPI_Comm comm,int m,int n,int nz,int *nnz, Mat *A)**, gdzie kolejne argumenty to: `comm` kanał komunikacji MPI, `m`. ilość wierszy macierzy, `n` ilość kolumn macierzy, `nz` ilość miejsc niezerowych w wierszu (stała dla wszystkich wierszy), `nnz` - tablica zawierająca liczbę elementów niezerowych w poszczególnych wierszach (lub `PETSC_NULL` gdy używamy tylko zmiennej `nz`).

3.5 Przykład podsumowujący

W tym przykładzie tworzona jest macierz *A* powstała z dyskretyzacji laplasjanu metodą różnic skończonych (elementu skończonego) na prostokącie[12]:

$$A=\begin{bmatrix} D & -I & & \\ -I & D & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & D \end{bmatrix}, \quad D=\begin{bmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{bmatrix}, \quad I=\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}$$

po czym mnożony jest przez nią wektor złożony z samych jedynek, a efekt zostaje wypisany na konsolę.

```
#include "sles.h"

int main(int argc,char **args)
{
  Vec     x,y;
  Mat     A;
  int     i,k,l,n=4,m=4,flg,col[5];
  Scalar  val[5];

/* inicjalizacja */
  PetscInitialize(&argc,&args,PETSC_NULL,PETSC_NULL);

/* wektory przydatne przy demonstracji */
  VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,n*m,&x);
  VecDuplicate(x,&y);
  val[0]=1.0;
  VecSet(val,x);

/* Tworzenie macierzy laplasjanu na prostokacie mxn */
  MatCreate(MPI_COMM_WORLD,n*m,n*m,&A);
  val[0] = -1.0; val[1] = -1.0; val[2] = 4.0;
  val[3]=-1.0;val[4]=-1.0;
  for (i=0; i<n*m; i++ )
  {
    col[0]=i+1; col[1]=i-1; col[2]=i;
    col[3]=i+m; col[4]=i-m;
    l=0;k=5;
    if(i<m)k=4;
    if((i+1)>(n-1)*m){k=4;col[3]=i-m;}
    if((i%m)==0){l=1;k=k-1;col[1]=i+1;}
    if((i+1)%m==0)if(i!=0){l=1;k=k-1;}
    MatSetValues(A,1,&i,k,&col[l],&val[l],INSERT_VALUES);
  }

/* Składanie macierzy*/
  MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);

/*Mnożenie wektora x przez macierz A (y=A*x)*/
  MatMult(A,x,y);

/*Wypisanie wektora y na konsolę*/
  VecView(y,VIEWER_STDOUT_SELF);
```

To dostaniemy po uruchomieniu przykładu:

```
hydra:/usr/home/>p5.6
2
1
1
2
1
0
0
1
1
0
0
1
2
1
1
2
hydra:/usr/home/> _
```

[12] Uważny czytelnik zapewne zauważy, że poprawna macierz dyskretyzacji laplasjanu metodą różnic skończonych powinna być jeszcze przemnożona przez czynnik $1/h^2$. Często jednak ten czynnik skalujący przenosi się na prawą stronę równania, dlatego w ćwiczeniu pomijamy go (jeśli jednak chcielibyśmy koniecznie go uwzględnić, po wygenerowaniu macierzy A jak w ćwiczeniu, powinniśmy przeskalować ją , używając w tym celu procedury `MatScale()` – w dalszej części podręcznika podajemy takie przykłady).

```
/* zakończenie */
  VecDestroy(x);
  MatDestroy(A);
  PetscFinalize();
  return(0);
 }
```

3.6 Macierze powłokowe

Jak wiadomo z podstawowego kursu algebry liniowej, macierz jednoznacznie definiuje operator liniowy, na przykład:

$$\mathbf{A} = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 4 & 0 \\ 5 & 6 & 7 \end{bmatrix}$$

jest macierzową reprezentacją (w standardowej bazie $\mathbf{R}^3$) następującego operatora:

$$\phi_{\mathbf{A}} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2x_1 + x_3 \\ 4x_2 \\ 5x_1 + 6x_2 + 7x_3 \end{pmatrix}$$

Taka definicja jest szczególnie ważna w przypadku macierzy rozrzedzonych, w których powtarzają się elementy, gdyż dzięki temu nie musimy tracić miejsca w pamięci na powtarzające się wartości. Na przykład macierz powstała z dyskretyzacji jednowymiarowego laplasjanu ma (z dokładnością do stałej) postać:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & & \ddots & & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}$$

Działanie tą macierzą na wektor można zapisać następująco:

$$\phi_{\mathbf{A}} \begin{pmatrix} x_1 \\ \vdots \\ x_x \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix},$$

gdzie

$$\begin{aligned} y_1 &= 2x_1 - x_2 \\ y_i &= 2x_i - x_{i-1} - x_{i+1} \quad dla \;\; i = 2,\ldots,n-1 \\ y_n &= 2x_n - x_{n-1} \end{aligned}$$

Autorzy pakietu przewidzieli możliwość definiowania macierzy w ten sposób. Użytkownik informuje pakiet jedynie o rozmiarze macierzy i podaje poszczególne funkcje realizujące operacje na tej macierzy (czy tą macierzą). Istotne jest to, że taka macierz jest traktowana *tak samo* jak inne rodzaje macierzy. Dopóki program nie usiłuje wykonać na nich operacji niewykonalnej, to znaczy nie zdefiniowanej przez użytkownika, to nie ma absolutnie żadnej różnicy w posługiwaniu się tą macierzą i macierzą „zwykłą”. Macierze definiowane w duchu operatorowym zwane są powłokowymi (ang. *shell matrix*).

Mając zadeklarowaną zmienną typu `Mat`, możemy utworzyć ją w postaci macierzy powłokowej. Do tego służy funkcja:

`MatShellCreate(MPI_Comm com,int m,int n,int M,int N,void *ctx,Mat *mat)`

Pierwszy argument to kanał komunikacyjny MPI (np. `MPI_COMM_WORLD`), drugi i trzeci argument (`m` i `n`) to rozmiar lokalny macierzy, to znaczy rozmiar części macierzy zapisanej w procesorze wywołującym tę funkcję. Czwarty i piąty argument (`M` i `N`) to globalny rozmiar macierzy. Oczywiście, w wersji sekwencyjnej będziemy mieli `n=N` i `m=M`; ponadto (zazwyczaj) nasze macierze będą kwadratowe, więc wtedy dodatkowo `m=n=M=N`. Szósty argument

(ctx) to wskaźnik do kontekstu związanego z macierzą[13], lub PETSC_NULL, gdy nie korzystamy z kontekstu. Ostatni argument to wskaźnik do zmiennej typu Mat z którą będzie powiązana macierz powłokowa.

Do definiowania operacji, którą można wykonać na macierzy powłokowej służy funkcja:
MatShellSetOperation(Mat mat,MatOperation op, void *f)
Pierwszy argument tej funkcji to zmienna powiązana z obiektem typu Mat, dla którego definiujemy operację. Drugi argument to nazwa operacji, a trzeci to wskaźnik[14] do funkcji wykonującej tą operację. Zdefiniować można 63 operacje, w poniższej tabeli podanych jest kilka ważniejszych:

NAZWA STAŁEJ	OPERACJA
MATOP_MULT	Pomnożenie wektora przez macierz
MATOP_EQUAL	Sprawdzenie czy dwie macierze są sobie równe
MATOP_GET_SIZE	Zwrócenie rozmiaru macierzy
MATOP_DESTROY	Zwolnienie pamięci zajmowanej przez macierz
MATOP_VIEW	Pokazanie macierzy
MATOP_NORM	Norma macierzy
MATOP_GET_DIAGONAL	Zwrócenie diagonali macierzy
MATOP_DIAGONAL_SCALE	Pomnożenie macierzy przez macierz diagonalną
MATOP_SET_VALUES	Wstawienie wartości do macierzy

Można więc samemu napisać funkcje wykonujące każdą z operacji na macierzach. To, czy dana funkcja została zdefiniowana dla macierzy powłokowej, sprawdzane jest dopiero przy próbie jej wywołania, wobec czego użytkownik nie musi definiować samemu wszystkich operacji, a jedynie te, które będą faktycznie wykorzystywane w programie. Na przykład wiadomo, że do iteracyjnego rozwiązywania układów równań często potrzeba jedynie wiedzy jak mnożyć wektor przez macierz. Możemy więc sami zdefiniować operację MATOP_MULT i rozwiązywać układy z tą macierzą metodami iteracyjnymi. Próba zastosowania bezpośredniego solvera do macierzy powłokowej zakończy się komunikatem o błędzie (chyba, że zdefiniowaliśmy operacje MATOP_LUFACTOR i MATOP_SOLVE.). Wszystkie operacje definiowane przez użytkownika muszą mieć taki sam format jak funkcje zdefiniowane w PETSc. Na przykład, jeżeli definiujemy operację mnożenia wektora przez macierz, to musi ona być zadeklarowana jak funkcja MatMult() czyli: **int mult(Mat A, Vec x, Vec y)**.

3.7 Przykłady

Przykład trywialny

W tym przykładzie zostanie utworzona powłokowa macierz identyczności (o rozmiarze 100x100). Mnożenie przez nią wektora daje ten sam wektor (użyliśmy funkcji VecCopy()).

```
#include "sles.h"
#define N=100

/**************************************************
            Procedura mnozenia przez macierz
**************************************************/

int mult(Mat A,Vec x,Vec y)
{
 VecCopy(x,y);
 return(0);
}

/**************************************************
                  Program glowny
**************************************************/
```

[13] Idea kontekstu zostanie szczegółowo wytłumaczona w rozdziale „Koncept kontekstu” na stronie 57.

[14] Formalnie jest to wskaźnik, brzmi to groźnie, ale w praktyce wskaźnikiem jest po prostu nazwa funkcji.

```
int main(int argc,char **args)
{
 Mat A;
 Vec v,b;
 PetscInitialize(&argc,&args,PETSC_NULL,PETSC_NULL);

 /* ...pominiete linie kodu: tworzenie wektora b i przypisanie mu
    wartosci... */

 /* tworzenie macierzy powłokowej */
 MatCreateShell(MPI_COMM_WORLD,N,N,N,N,PETSC_NULL,&A);
 MatShellSetOperation(A,MATOP_MULT,mult);

 /* zastosowanie */
 MatMult(A,v,b);
 PetscFinalize();
 return(0);
}
```

Dyskretny laplasjan

W kolejnym przykładzie, *A* będzie (z dokładnością do stałej) macierzą powstałą z dyskretyzacji jednowymiarowego równania Poissona z warunkami brzegowymi Dirichleta na odcinku, czyli macierzą trójdiagonalną z 2 na głównej przekątnej i 1 na kodiagonalach (patrz przykład ze strony 14). Przypomnijmy, że działanie tej macierzy na wektor można zapisać jako następujący operator:

$$\varphi\begin{pmatrix} x_0 \\ \vdots \\ \vdots \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} 2x_0 - x_1 \\ \vdots \\ 2x_i - x_{i-1} - x_{i+1} \\ \vdots \\ 2x_{n-1} - x_{n-2} \end{pmatrix}$$

```
#include "sles.h"
#include <stdio.h>

/*****************************************************
      Procedura mnozenia przez macierz
*****************************************************/

int mult(Mat A,Vec x,Vec y)
{
 int i,N,*idx;
 Scalar *wx,*wy;

 VecGetSize(x,&N);
 VecGetArray(x,&wx);
 VecGetArray(y,&wy);
 for(i=0;i<N;i++)
 {
  wy[i] = 2*wx[i];
  if(i>0) wy[i] -= wx[i-1];
  if(i<N) wy[i] -= wx[i+1];
 }
 VecRestoreArray(x,&wx);
 VecRestoreArray(y,&wy);
}


/*****************************************************
                    Program glowny
*****************************************************/
```

```
int main(int argc,char **args)
{
 Mat A;
 int N=10;
 Scalar val;
 Vec x,y;

 PetscInitialize(&argc,&args,PETSC_NULL,PETSC_NULL);

 /* tworzenie macierzy powłokowej */
 MatCreateShell(MPI_COMM_WORLD,N,N,N,N,0,&A);
 MatShellSetOperation(A,MATOP_MULT,mult);

 /* Wektory przydatne przy prezentacji */
 VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,N,&x);
 VecDuplicate(x,&y);
 val=1;
 VecSet(&val,x);

 /* uzycie macierzy powlokowej */
 MatMult(A,x,y);
 VecView(y,VIEWER_STDOUT_SELF);

 /* Zakończenie */
 VecDestroy(x); VecDestroy(y);
 MatDestroy(A);
 PetscFinalize();
 return(0);
}
```

To dostaniemy po uruchomieniu przykładu:

```
hydra:/usr/home/> przykład
1
0
0
0
0
0
0
0
0
1
hydra:/usr/home/> _
```

Powyższy program działa prawidłowo tylko w wersji sekwencyjnej, ponieważ funkcja `VecGetArray(x,&wx)` podaje wskaźnik do lokalnej tablicy wartości wektora x. Możliwe jest oczywiście napisanie w pełni równoległej funkcji mnożącej wektor przez macierz, ale oznaczałoby to zagłębianie się w subtelności związane z programowaniem równoległym z użyciem MPI, których nie będziemy omawiać w tym podręczniku.

3.8 Ćwiczenia

A) Uruchom program z przykładu trywialnego dla różnych *N* i dodaj przeglądarki wektorów, dzięki której będzie można obejrzeć wektor przed i po mnożeniu przez macierz. Proponujemy także zaimplementowanie innej macierzy diagonalnej w postaci macierzy powłokowej, np.:

$$A = \begin{bmatrix} 5 & & \\ & \ddots & \\ & & 5 \end{bmatrix}$$

B) Wygeneruj macierz *NxN*:

$$A = \begin{bmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}$$

obejrzyj ją w okienku, a następnie na konsoli. Naucz się odczytywać te widoki.

C) Przeskaluj macierz *A* o czynnik $\frac{1}{h^2}$, gdzie $h = \frac{1}{N+1}$. Napisz procedurę `GenLap1D(Mat A, int h)` która generuje macierz *NxN* dyskretyzacji 1-wymiarowego Laplasjanu.

D) Zrób to samo co w poprzednim ćwiczeniu, ale macierz zaimplementuj jako macierz powłokową.

E) Wygeneruj macierz dyskretyzacji operatora

$$(Lu)(x) = -u''(x) + u'(x)$$

metodą różnic skończonych. Wskazówka: użyj funkcji `GenLap1D()` a następnie dodaj wartości pochodzące z dyskretyzacji pierwszej pochodnej.

F) Obejrzyj macierz z poprzedniego ćwiczenia, następnie transponuj ją i znów obejrzyj.

G) Napisz funkcję tworzącą macierz dyskretyzacji pierwszej pochodnej różnicą centralną

$$B = \frac{1}{2h}\begin{bmatrix} 0 & 1 & & \\ -1 & 0 & \ddots & \\ & \ddots & \ddots & 1 \\ & & -1 & 0 \end{bmatrix}$$

Użyj obu macierzy do dyskretyzacji operatora L z ćwiczenia E) w wersji

$$L_h u = Au + Bu$$

(Najwygodniej zdefiniować go jako macierz powłokową!)

H) Napisz procedurę rozwiązywania układu z daną macierzą, metodą Richardsona z preconditionerem Jacobiego. Użyj funkcji `MatGetDiag()` do wyciągnięcia diagonali, a całą procedurę zaimplementuj używając funkcji wektorowych `VecAXPY()`, `VecScale()`, `VecPointwiseDivide()`.

I) Zaprogramuj własną przeglądarkę do macierzy/wektorów, używając procedur graficznych PETSc, tak, aby każde wywołanie otwierało nowe okno, utrzymujące się do końca pracy programu.

J) Procedura z ćwiczenia H) wymaga pewnych obiektów pomocniczych wektorów roboczych do przechowywania wyników pośrednich. Zastanów się jak to zgrabnie zaimplementować. Zajrzyj do rozdziału „Koncept kontekstu" na stronie 57.

K) Trudne zadanie, ale warto. Zaimplementuj procedurę generującą macierz sztywności dla dyskretyzacji równania eliptycznego drugiego rzędu na zadanej triangulacji MES i rozmaitych elementów skończonych. Oczywiście, zgodnie z prawidłami sztuki, najpierw skonstruuj lokalne macierze sztywności, które potem złożysz w globalną macierz sztywności w procesie *assemblacji*. Znakomicie pasuje tu funkcja `MatSetValues()`.

4. SLES, czyli rozwiązywanie układów równań liniowych

Pod tą nazwą kryje się główny mechanizm służący do rozwiązywania układów równań liniowych (ma on bliźniaczy odpowiednik dla równań nieliniowych: SNES). Skrót SLES oznacza *Solvers for Linear Equation Sets* - czyli solvery dla układów równań liniowych. SLES jest zarówno nazwą części pakietu jak i obiektu funkcjonującego w PETSc. Przypuśćmy, że mamy do rozwiązania układ równań:

$$Ax=b.$$

Aby go rozwiązać, musimy utworzyć wektory rozwiązania i prawej strony oraz macierz zadania, a następnie spiąć te wszystkie dane ze sobą i zlecić pakietowi rozwiązanie tego układu. Do tego służą właśnie obiekty typu SLES.

Aby utworzyć obiekt typu SLES trzeba, podobnie jak w poprzednich przypadkach, najpierw zadeklarować odpowiednią zmienną:

```
...
SLES MojSles;
...
```

a następnie utworzyć obiekt typu SLES funkcją **`SLESCreate(MPI_Comm comm,SLES *MojSles)`**.

Pierwszy argument funkcji `SLESCreate()` to, podobnie jak w przypadku innych obiektów, kanał komunikatora MPI, u nas zawsze będzie to `MPI_COMM_WORLD`. Drugi to wskaźnik do uprzednio zadeklarowanej zmiennej typu SLES.

Następną czynnością, jaką należy wykonać, jest przekazanie macierzy zadania do SLESu. Robi się to przy pomocy funkcji:

`SLESSetOperators(SLES MojSles,Mat A,Mat P, MatStructure flag);`

Pierwszym argumentem funkcji jest obiekt typu SLES dla którego ustawiamy operatory. Drugim macierz zadania `A`. Trzecim argumentem jest macierz `P` z której będzie tworzony preconditioner (zazwyczaj wystarczy podać macierz

zadania[15]). Nie ma znaczenia czy macierze są „normalne” to znaczy przechowywane w postaci tablicy elementów typu `Scalar`, czy są to macierze powłokowe. Czwartym argumentem jest informacja o strukturze macierzy preconditionera `P`, pozwalająca na wyeliminowanie niepotrzebnej pracy w trakcie iteracyjnego rozwiązywania równań. Możliwe wartości parametru `flag` to:

WARTOŚĆ	OPIS
SAME_PRECONDITIONER	Macierz `P` jest cały czas taka sama w czasie procesu iteracyjnego. Ta opcja jest dla tych, którzy używają różnych macierzy `A` i `P`. Opcja ta np. oszczędza pracę przy niekompletnym rozkładzie Choleskiego stosowanym jako preconditioner: rozkład jest robiony tylko raz.
SAME_NONZERO_PATTERN	w trakcie iteracji macierz `P` ma taką samą strukturę miejsc niezerowych
DIFFERENT_NONZERO_PATTERN	w trakcie iteracji macierz `P` zmienia strukturę

Wywołanie **`SLESSetFromOptions(SLES MojSles)`** pozwala[16] na kontrolowanie prawie wszystkich możliwości SLESu z linii komend, bez konieczności rekompilowania programu.

Powyższe przygotowania wystarczają, aby SLES rozwiązał układ równań z macierzą `A` i prawą stroną `b`. W tym celu należy wywołać funkcję **`SLESSolve(SLES MojSles,Vec b,Vec x,int *its).`** Wynik zostanie zapisany w wektorze `x`.

W zmiennej `its` zostanie zapisana liczba wykonanych iteracji. Po zakończeniu obliczeń należy zwolnić pamięć zajmowaną przez SLES funkcją: **`SLESDestroy(SLES MojSles),`** gdzie argumentem jest (niepotrzebny już) obiekt `MojSles`. Ważne jest, że jeden SLES może służyć do rozwiązania wielu układów równań, na przykład kilku układów z tą samą macierzą i inną prawą stroną.

4.1 Podsumowanie

Aby rozwiązać układ równań liniowych $Ax=b$ należy:

- Zadeklarować zmienne `Mat A; Vec b,x;` i `SLES MojSles;`
- Utworzyć wektor rozwiązania `x` i wektor prawej `b` strony, a następnie wpisać w `b` odpowiednie wartości (szczegóły w rozdziale 2 na stronie 9)
- Utworzyć macierz zadania `A` i wpisać w nią odpowiednie wartości.
- Utworzyć obiekt typu SLES funkcją **`SLESCreate()`**
- Przekazać macierz zadania i preconditionera do SLESu funkcją **`SLESSetOperators()`**
- Rozwiązać zadanie funkcją **`SLESSolve()`**
- Po rozwiązaniu, gdy już obiekt jest niepotrzebny, należy zwolnić pamięć funkcją **`SLESDestroy()`**

4.2 Przykład

```
#include "sles.h"

int main(int argc,char **args)
{
  Vec    x,b;
  Mat    A;            /* macierz układu równań */
  SLES   MojSles;         /* kontekst SLESu */
  int    its;
  Scalar val;
```

[15] Informacja w manualu na ten temat jest tak lakoniczna, że trudno wywnioskować kiedy te macierze miałyby być różne. Dość wiedzieć, że gdy wstawimy na obu miejscach tę samą macierz to wszystko będzie działało sprawnie.

[16] przy pewnych dodatkowych liniach kodu, szczegóły można znaleźć w rozdziale „Co jeszcze trzeba wiedzieć?” na stronie 25.

```
  PetscInitialize(&argc,&args,PETSC_NULL,PETSC_NULL);

/* Tworzenie kontekstu SLES */
  SLESCreate(MPI_COMM_WORLD,&MojSles);

/* tutaj trzeba utworzyć wektory (str. 12)
   i macierze, (str. 17) */

/*Ustawienie operatorów */
  SLESSetOperators(MojSles,A,A,DIFFERENT_NONZERO_PATTERN);
  SLESSetFromOptions(MojSles);

/* rozwiązanie układu liniowego */
  SLESSolve(MojSles,b,x,&its);

/*Tu z kolei należy zwolnić pamięć*/
  SLESDestroy(MojSles);
  PetscFinalize();
  return(0);
}
```

4.3 Ćwiczenia

A) Rozwiąż równanie

$$-u''(x)+u(x)=f(x),\ x\in(0,\pi),$$
$$u(0)=a,\ u(\pi)=b,$$

na siatce o kroku h. Testuj błąd rozwiązania w normie *max* i l_2 dla f odpowiadającym rozwiązaniu dokładnemu a) $sin(x)$, b) x^2+y^2. Sprawdź, jak zmienia się liczba iteracji w zależności od h. Obejrzyj wektory rozwiązania i błędu w okienku graficznym.

B) Rozwiąż równanie

$$-u''(x)+u'(x)=f(x),\ x\in(0,\pi),$$
$$u(0)=a,\ u(\pi)=b.$$

Eksperymentuj z różnymi sposobami implementacji macierzy (klasycznie i powłokowo).

5. Praca z PETSc

5.1 Kompilowanie programów

W środowisku UNIX programista ma do dyspozycji program zwany `make` - pozwalający na szybkie i wygodne kompilowanie programów. Ponieważ kompilator i linker z reguły wymagają ustawienia wielu opcji i podania licznych ścieżek dostępu, nie byłoby wygodnie za każdym razem podawać wszystkie potrzebne parametry wywołania kompilatora w linii komend. Do zapanowania nad tymi parametrami służy właśnie program `make`, a ściślej, plik `Makefile`, zawierający potrzebne parametry kompilacji. Skompilowanie programu sprowadza się wówczas do wydania komendy

```
hydra:/usr/home/bin> make moj_program
```

Aby jednak to działało, należy jednak napisać własny plik tekstowy nazwany `Makefile` w którym podane zostaną (ale tylko raz) wszystkie parametry potrzebne do kompilowania i linkowania programu. Przykładowy plik `Makefile`, pozwalający na skompilowanie programu z dołączeniem bibliotek PETSc podajemy poniżej. Zmienna `BOPT` mówi jaka wersja programu ma być tworzona. `BOPT=O` oznacza program zoptymalizowany, `BOPT=g` oznacza wersję do odpluskwiania (debuggingu). Ważne jest, żeby gotowy (tzn. odpluskwiony) program kompilować z opcją `BOPT=O`, gdyż to daje kilkukrotnie mniejszy (około 5 razy) i szybszy kod wynikowy. Oczywiście, aby w ogóle dało się skompilować program, muszą być ustawione zmienne środowiskowe opisane w rozdziale „Zaczynamy” na stronie 7. Tworzenie `Makefile`'ów jest odrębną sztuką. Zamiast ją zgłebiać, proponujemy pisanie własnych przez prostą modyfikację poniższego:

Makefile

```
BOPT=O
include $(PETSC_DIR)/bmake/$(PETSC_ARCH)/base

CFLAG           = $(CPPFLAGS) -D__SDIR__='"$(LOCDIR)"' $(BS_INCLUDE)
LIBBASE         = libpetscsles
LOCDIR          = /usr/home/students/bojko/source/

moj_program: moj_program.o
             $(CLINKER) -o moj_program moj_program.o  $(PETSC_LIB)
              $(RM) moj_program.o
```

Bardziej rozbudowane `Makefile`, umożliwiające kompilację kilku różnych programów, znajdują się na stronach 41 i 55.

5.2 Uruchamianie programów

Są dwa sposoby uruchamiania programów w PETSc. Jeżeli uruchamiamy program na maszynie sekwencyjnej to wystarczy podać nazwę programu (powiedzmy, że nazywa się on `moj_program`) i opcje jego wywołania:

```
hydra:/usr/home/bin> moj_program [opcje]
```

Drugi sposób stosujemy, gdy uruchamiamy program na maszynie wieloprocesorowej z zainstalowaną biblioteką MPI. Wówczas program uruchamia się komendą:

```
hydra:/usr/home/bin> mpirun -np 4 moj_program [opcje]
```

Opcja `-np 4` oznacza liczbę procesorów, na których będzie uruchomiony `moj_program` (w tym wypadku 4). Dalej podajemy nazwę programu i ewentualne opcje samego programu.

5.3 Co jeszcze trzeba wiedzieć?

Do używania dotychczas wymienionych funkcji (działających na wektorach, macierzach i SLESie) wystarczy umieścić na początku programu następujący plik nagłówkowy:

```
#include "sles.h"
```

Zawiera on deklaracje wszystkich typów zmiennych i funkcji potrzebnych do rozwiązywania układów równań liniowych oraz innych podstawowych funkcji PETSc (np. przeglądarek, tworzenia wektorów i macierzy, itp.).

Jak już wspomnieliśmy, dzięki użyciu funkcji `main()` z argumentami `argc`, `args` oraz wywołaniu funkcji `PetscInitialize()`, tekst z linii komend zostanie przekazany bezpośrednio do programu i potraktowany jako opcje[17], co pozwala na sterowanie wieloma parametrami programu. Można zmieniać rozmiar zadania, podawać dodatkowe parametry itp.

W pełnym wykorzystaniu opcji pomocne będą funkcje: **`OptionsGetXXX(char* pre, char *name, XXX *zmienna,int *flag)`** gdzie w miejscu `XXX` wstawiamy typ wartości podawanej z linii komend: `Int`, `Double`, `Scalar`, `String`, `DoubleArray`. Na przykład funkcja

```
....
OptionsGetInt(PETSC_NULL,"-N",&rozmiar);
....
```

pobiera z linii komend zmienną typu `int` poprzedzoną symbolem `-N` i zapisuje w zmiennej `rozmiar` (której adres jest podany jako ostatni argument funkcji)

Argumenty funkcji **`OptionsGetXXX()`** mają następujące znaczenie: pierwszy to przedrostek, u nas będzie zawsze ustawiony na `PETSC_NULL`. Drugi to poszukiwany ciąg znaków, a trzeci argument to wskaźnik zmiennej do której będzie przekazywany wynik. Ostatni argument to wskaźnik do zmiennej, w której zapisane zostanie czy przypisanie wartości miało miejsce, czy nie. Przydatna jest też funkcja **`OptionsHasName(char* pre, char *name,int *flag)`**, która sprawdza, czy dany ciąg znaków `name` pojawił się w linii komend i wynik zapisuje w zmiennej `flag`.

[17] Options - to także pełnoprawny obiekt w PETSc, z własnym zbiorem funkcji. Podany sposób inicjalizacji PETSc tworzy obiekt - opcje, który będzie funkcjonował w programie aż do wywołania funkcji `PETSCFinalize()`.

Ważnym ułatwieniem w pisaniu programów jest to, że wszystkie funkcje w PETSc zwracają kody błędów. Jest też makro **CHKERRA(int ierr)**, która w przypadku wystąpienia błędu poda numer linii programu i nazwę pliku z kodem źródłowym oraz krótki opis błędu. Jeżeli zatem każdą funkcję PETSc wywołujemy następująco:

```
...
ierr = VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,N,&v); CHKERRA(ierr);
ierr = VecSet(&val,v); CHKERRA(ierr);
...
```

to bardzo łatwo będziemy mogli zlokalizować i określić ewentualny błąd. W podprogramach będziemy używali analogicznego makra **CHKERRQ(int ierr)**. W przykładowych programach przytoczonych do tej pory, nie zastosowaliśmy takiej konwencji ze względu na przejrzystość kodu źródłowego.

5.4 Przykłady

Opcje PETSc.

Standardowo PETSc zapewnia bardzo szeroką obsługę programu z linii komend. Na przykład po uruchomieniu programu z opcją `-h` lub `-help` zostanie wypisana pomoc do samego pakietu, oraz każdego komponentu PETSc dla którego była wywołana w programie funkcja `[...]SetFromOptions()` (Na przykład `SLESSetFromOptions(sles)`)

Opcja **-sles_view** spowoduje wypisanie na konsolę informacji na temat SLESu

Opcja **-vec_view** spowoduje, że po wywołaniu w programie funkcji `VecAssemblyBegin(v)` zostanie wypisany na konsolę wektor v. Podobna do niej **-vec_view_draw** narysuje w oknie graficznym wektor jako funkcję współrzędnej.

Opcja **-mat_view** wypisuje na konsolę każdą kolejną macierz, dla której wywołano w programie funkcję `MatAssemblyBegin( ... ,FINAL_ASSEMBLY)`. Podobna do niej **-mat_view_draw** narysuje w oknie graficznym strukturę miejsc niezerowych macierzy.

Opcja **-draw_pause <**`n`**>** spowoduje zatrzymanie przeglądarki graficznej (zarówno wywoływanej opcją z linii komend jak i z programu) na `n` sekund. Jeżeli `n=0` to przeglądarka czeka na kliknięcie klawiszem myszki. To bardzo przydatna opcja, gdyż często przeglądarka jest zamykana zaraz po zakończeniu rysowania, co w wielu przypadkach nie wystarcza nawet na zauważenie, że okno pojawiło się na ekranie.

Opcją **-ksp_type <**`typ`**>** możemy zmienić rodzaj metody iteracyjnej używanej do rozwiązywania układu równań (szczegóły na stronie 29).

Inne opcje będziemy omawiać na bieżąco. Opis opcji dostępnych w każdym z programów jest wypisywany gdy uruchamiamy program z opcją `-help`.

Opcje użytkownika.

Poniżej omówimy na przykładzie sposób łatwego przekazywania parametrów do programu w PETSc. Rozważmy krótki program:

```
#include "sles.h"
static char help[]="Pomoc";
int main(int argc,char **args)
{
 int N=5,pc_flag=0;
 double af=0.1;

 PetscInitialize(&argc,&args,PETSC_NULL,help);
 OptionsGetInt(PETSC_NULL,"-N",&N);
 OptionsGetDouble(PETSC_NULL,"-alfa",&af);
 OptionsHasName(PETSC_NULL,"-bez_pc",&pc_flag);
 PetscPrintf(MPI_COMM_WORLD,"rozmiar=%i,af=%d,pc_flag=%i\n",N,af,pc_flag);
 PetscFinalize();

 return(0);
}
```

Uruchomienie tego programu (w wersji jednoprocesorowej) z następującymi opcjami

```
hydra:/usr/home/bin/> program -help -N 10 -alfa 0.22
```

spowoduje przypisanie zmiennej `rozmiar` wartości 10, zmiennej `af` wartości 0.22, oraz zostanie wypisana pomoc do pakietu z uwzględnieniem stałej tekstowej `help` (patrz wyżej).

```
hydra:/usr/home/bin/> program -N 23 -bez_pc
```

spowoduje przypisanie zmiennej `rozmiar` wartości 23, ustawienia `pc_flag` na `TRUE`, a zmienna `af` pozostanie bez zmian, tzn. równa 0.1.

5.5 Ćwiczenie

A) Wzbogać swoje dotychczasowe programy o możliwość ustawiania kluczowych parametrów z linii komend.

6. Pierwszy „poważny" program używający PETSc - rozwiązywanie równania Poissona

Dotychczas zdobyta przez nas wiedza pozwala na uruchomienie „prawdziwego" programu używającego PETSc. Jak to zostało zapowiedziane, program będzie rozwiązywał układ równań powstałych z dyskretyzacji metodą różnic skończonych zadania Poissona na kwadracie jednostkowym:

$$\begin{cases} -\Delta u(x) = f(x) \; dla \; x \in \Omega = (0,1)^2 \\ u|_{\partial\Omega} = 0 \end{cases}$$

przy czym rozważymy przypadek gdy $f(x) \equiv 1$.

Jak wiemy, dyskretyzacja taka prowadzi do następującej postaci macierzowej (patrz [6]):

$$\mathbf{A}\cdot x = b$$

gdzie macierz A ma postać:

$$A = \underbrace{\begin{bmatrix} D & -I & & \\ -I & D & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & D \end{bmatrix}}_{N^2} \Bigg\} N^2, \; D = \frac{1}{h^2}\underbrace{\begin{bmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{bmatrix}}_{N} \Bigg\} N, \; I = \frac{1}{h^2}\underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}}_{N} \Bigg\} N$$

```
static char help[] = "Rozwiązuje pięciodiagonalny układ z użyciem SLES.\n\n";

#include "sles.h"
#include <stdio.h>

int main(int argc,char **args)
{
  Vec     x, b, u;
  Mat     A;            /* macierz układu równań */
  SLES    sles;         /* kontekst SLESu */
  int     ierr, i, n = 10, m = 10,col[5], its,flg,k,l;
  Scalar  minus_one = -1.0, one = 1.0, val[5],h;
  double  norm;

  PetscInitialize(&argc,&args,PETSC_NULL,help);
  OptionsGetInt(PETSC_NULL,"-n",&n,&flg);
  OptionsGetInt(PETSC_NULL,"-m",&m,&flg);

/* Tworzenie wektorów */
  VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,n*m,&x);
  VecDuplicate(x,&b);
  VecDuplicate(x,&u);
/* Wartości wektora prawej strony */
  val[0]=1;
  VecSet(val,b);
```

```
/* Tworzenie macierzy laplasjanu na prostokacie mxn */
  MatCreate(MPI_COMM_WORLD,n*m,n*m,&A);
  h=1/(m*n+1);
  val[0] = -1.0/h; val[1] = -1.0/h; val[2] = 4.0/h;
  val[3] = -1.0/h; val[4] = -1.0/h;
  for (i=0; i<n*m; i++ )
  {
    col[0]=i+1; col[1]=i-1; col[2]=i;
    col[3]=i+m; col[4]=i-m;
    l=0;k=5;
    if(i<m) k=4;
    if((i+1)>(n-1)*m){k=4;col[3]=i-m;}
    if((i%m)==0){l=1;k=k-1;col[1]=i+1;}
    if((i+1)%m==0)if(i!=0){l=1;k=k-1;}
    MatSetValues(A,1,&i,k,&col[l],&val[l],INSERT_VALUES);
  }
  MatSetValues(A,1,&i,k,&col[l],&val[l],INSERT_VALUES);
  MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);

/* Tworzenie kontekstu SLES */
  SLESCreate(MPI_COMM_WORLD,&sles);

/*Ustawienie operatorów */
  SLESSetOperators(sles,A,A,DIFFERENT_NONZERO_PATTERN);
  SLESSetFromOptions(sles);

/* rozwiązanie układu liniowego */
  SLESSolve(sles,b,x,&its);

/* Sprawdzenie błedu */
  VecAXPY(&minus_one,u,x);
  VecNorm(x,NORM_2,&norm);
  PetscPrintf(MPI_COMM_WORLD,"Norma błędu %g, %d iteracji\n",norm,its);

/* Zwolnienie przestrzeni roboczej */
  VecDestroy(x);VecDestroy(u);
  VecDestroy(b);MatDestroy(A);
  SLESDestroy(sles);
  PetscFinalize();
  return(0);
}
```

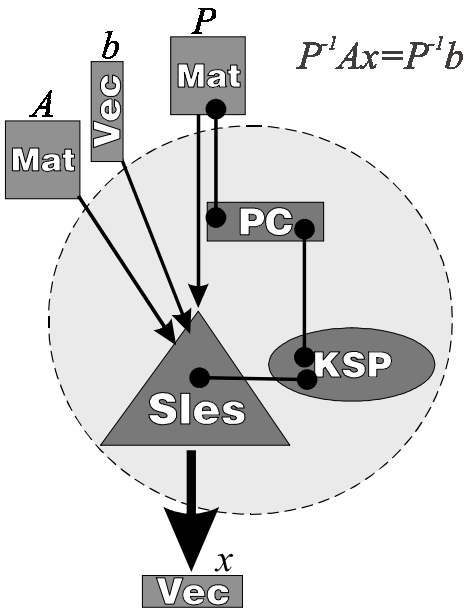
Po uruchomieniu mamy:

```
hydra:/usr/home/students/bojko/> przyklad

Norma błędu 0.00893566, 65 iteracji

hydra:/usr/home/students/bojko/>
```

7. KSP, czyli iteracyjne metody rozwiązywania równań



Rysunek 3. Powiązania SLESu z innymi obiektami PETSc.

Kolejnym komponentem PETSc jest KSP (*Krylov Subspace Procedures*) - mówiąc w uproszczeniu, jest to zestaw procedur do rozwiązywania równań liniowych metodami iteracyjnymi. Obiekt ten jest integralną częścią SLESu, choć nie musi być oddzielnie definiowany przez użytkownika. **Rysunek 3** pokazuje powiązania SLESu z innymi obiektami PETSc. Na zewnątrz koła pokazane są obiekty, które użytkownik musi zdefiniować. Wewnątrz koła pokazane są te obiekty związane ze SLESem, których użytkownik nie musi definiować samemu (to znaczy deklarować i tworzyć): KSP i PC. Dwa następne rozdziały będą poświęcone tym właśnie obiektom. Gdy tworzymy obiekt typu SLES, automatycznie generowany jest także obiekt typu KSP związany ze SLESem. Dokładniej, podczas wywołania funkcji `SLESSolve()` lub `SLESSetUp()`, automatycznie przygotowywany jest, między innymi, obiekt typu KSP i ustawiane wszystkie parametry potrzebne do prawidłowego działania procedur KSP. Mówiąc ogólnie, to właśnie parametry KSP decydują, jaka metoda zostanie wykorzystana do rozwiązywania równania. Do wyboru mamy między innymi: metody iteracyjne Richardsona, Czebyszewa, CG i GMRES (szczegółowe opisy wyżej wymienionych metod można znaleźć w [11], [8] i [10]). Domyślnie używana jest metoda GMRES z restartem po 30 iteracjach. Metodę rozwiązywania możemy zmienić albo wpisując odpowiednie funkcje do kodu źródłowego, albo uruchamiając program z odpowiednimi opcjami linii komend (zakładając, że zostały spełnione warunki opisane w rozdziale 7).

Jeżeli chcemy wybrać metodę z poziomu kodu źródłowego, trzeba to zrobić w następujący sposób: zadeklarować zmienną typu KSP i powiązać ją z obiektem KSP ze SLESu funkcją **`SLESGetKSP(SLES sles,KSP *ksp)`**, a potem użyć funkcji **`KSPSetType(KSP ksp, KSPType itmethod)`**. Zmienna `sles` jest związana z obiektem SLES, z którego wydobywamy obiekt KSP, z kolei `*ksp` to wskaźniki do obiektu KSP. `itmethod` jest stałą PETSc oznaczającą rodzaj metody. Spis stałych znajduje się w tabelce poniżej. Jeżeli po ustawieniu rodzaju metody z poziomu kodu źródłowego funkcją `KSPSetType()` wywołamy funkcję `SLESSetFromOptions()` to dodatkowo będziemy w stanie zmienić metodę opcją z linii komend, ale domyślną metodą będzie metoda ustawiona z poziomu kodu źródłowego.

Można też wybrać metodę uruchamiając program z opcją **`-ksp_type`** `<metoda>`. Poniższa tabelka podaje wszystkie stałe, predefiniowane w PETSc i opcje, którymi wybieramy metodę z linii komend

STAŁA PETSc	OPCJA Z LINII KOMEND	Wektor residualny używany do testowania zbieżności
KSPRICHARDSON	richardson	Normalny
KSPCHEBYSHEV	chebyshev	Normalny
KSPCG	cg	Normalny
KSPGMRES	gmres	Przewarunkowany
KSPTCQMR	tcqmr	Przewarunkowany
KSPBCGS	bcgs	Przewarunkowany
KSPCGS	cgs	Przewarunkowany
KSPTFQMR	tfqmr	Przewarunkowany
KSPCR	cr	Przewarunkowany
KSPLSQR	lsqr	Przewarunkowany
KSPPREONLY	preonly	Przewarunkowany

Ostatnia metoda w tej tabelce: `KSPPREONLY`, oznacza metodę, w której do liczenia wektora residualnego użyty zostanie jedynie preconditioner. Tego typu metodę wybieramy, gdy zamierzamy użyć metody bezpośredniej - szcze-

góły w następnym rozdziale.

7.1 Podsumowanie

Wyboru metody iteracyjnej, która posłuży SLESowi do rozwiązywania układu równań możemy dokonać dwojako:

- W linii komend przy uruchamianiu programu podać opcję `-ksp_type <typ>` (patrz tabelka powyżej). Należy pamiętać o wywołaniu w programie funkcji `SLESSetFromOptions()`.

Drugim sposobem jest ustawienie metody domyślnej z poziomu kodu źródłowego funkcją `KSPSetType()`. W tym przypadku należy kolejno:

- zadeklarować zmienną typu KSP
- wydobyć funkcją `SLESGetKSP()` z obiektu typu SLES obiekt KSP wiążąc go z uprzednio zadeklarowaną zmienną typu KSP
- ustawić metodę iteracyjną funkcją `KSPSetType()`

7.2 Przykład

```
#include "sles.h"
int main(int argc,char **args)
{
 Vec x,b;
 SLES sles;
 KSP  ksp;
 int its;

 /* inicjalizacja */
 PetscInitialize(&argc,&args,PETSC_NULL,PETSC_NULL);
 SLESCreate(MPI_COMM_WORLD,&sles);

 /* wybór metody KSP */
 SLESGetKSP(sles,&ksp);
 KSPSetType(ksp,KSPRICHARDSON);

 /* W tej części programu muszą zostać ustawone wszystkie parametry SLESu, */
 /* omówione w poprzednich rozdziałach. Dla większej przejrzystości        */
 /* przykładu zostały pominięte                                            */

 /* rozwiaz! */
 SLESSetFromOptions(sles);
 SLESSolve(sles,x,b,&its);

/* zakończenie */
 SLESDestroy(sles);
 PetscFinalize();
 return(0);
}
```

Domyślnie, ustawiona zostaje metoda Richardsona. Można ją zmienić, używając opcji `-ksp_type <typ>`.

7.3 Warunek stopu

Domyślny warunek stopu metody iteracyjnej oparty jest na normie euklidesowej wektora residualnego. Zbieżność (lub rozbieżność) stwierdzana jest na podstawie trzech wielkości: relatywnego zmniejszenia się normy wektora residualnego - **`rtol`**, bezwzględnej normy wektora residualnego - **`atol`** lub względnego zwiększenia wektora residualnego powyżej zadanego progu - **`dtol`**. Dokładniej, proces iteracyjny zatrzymuje się gdy zostanie spełniony jeden z poniższych warunków:

- $\frac{\|r_k\|_2}{\|r_0\|_2} < \texttt{rtol}$
- $\|r_k\|_2 < \texttt{atol}$

- $\|r_k\| > \texttt{dtol} * \|r_0\|$ (w tym przypadku stwierdzona zostaje rozbieżność)
- przekroczono zadaną liczbę iteracji **maxits.**

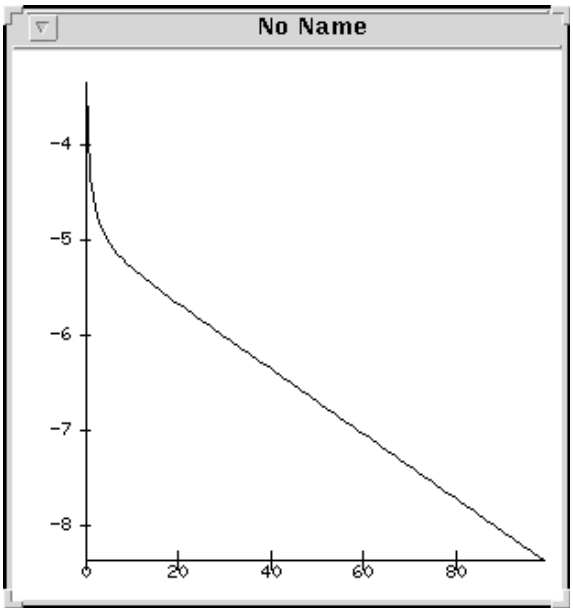
Jeżeli stwierdzona została rozbieżność, to liczba iteracji zwracana przez `SLESSolve()` jest poprzedzona znakiem minus. Wszystkie te wielkości mogą być ustawione w programie przy pomocy funkcji:

KSPSetTolerances(KSP ksp,double rtol,double atol,double dtol,int maxits).

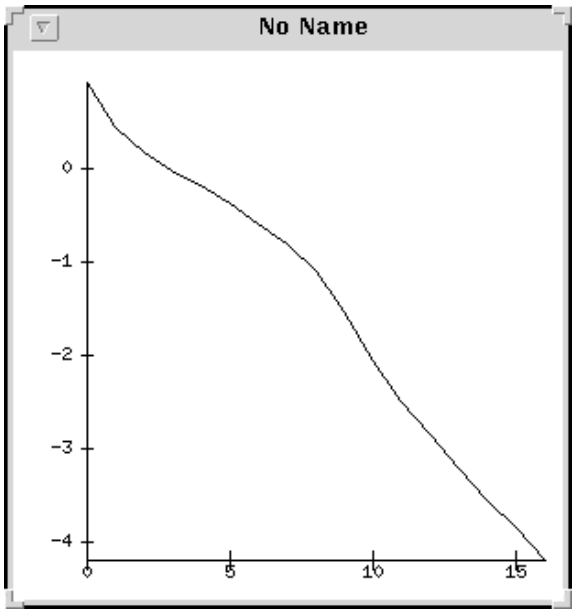
Te same parametry można też ustawić w linii komend podając opcje **-ksp_rtol** <rtol>, **-ksp_atol** <atol>, **-ksp_dtol** <dtol>, **-ksp_max_it** <its>. Wartości domyślne wynoszą

`rtol` $= 10^{-5}$, `atol` $= 10^{-50}$, `dtol` $= 10^{5}$ i `maxits` $= 10^{5}$.

Inne przydatne opcje dla KSP to **-ksp_monitor** i **-ksp_xmonitor**[18], które na bieżąco śledzą (odpowiednio: w trybie tekstowym lub graficznym) przebieg procesu iteracyjnego rozwiązywania równania. W pierwszym przypadku na każdym kroku zostaje wypisana na konsolę norma wektora residualnego. W drugim przypadku przebieg zbieżności obserwujemy w oknie graficznym, zob. **Rysunek 4** i **5**.



Rysunek 4. Przebieg zbieżności dla metody Richardsona. Zauważ, że metoda GMRES (obok) jest szybsza!



Rysunek 5. Przebieg zbieżności procesu iteracyjnego dla metody GMRES.

7.4 Warto wiedzieć, że ...

W metodach CG i GMRES można włączyć opcję liczenia skrajnych wartości własnych operatora według algorytmu Lanczosa (dla CG) lub Arnoldiego (dla GMRES). Jest to przydatne, jeżeli chcemy w ten sposób w sposób przybliżony wyznaczyć uwarunkowanie macierzy.[19] Używa się do tego funkcji `KSPSetComputeSingularValues(KSP ksp)`. Należy ją wywołać przed wywołaniem funkcji `SLESSolve()` (dokładnie przed wywołaniem funkcji `KSPSetUp()`, ale automatycznie wywołuje ją funkcja `SLESSetUp()`, a tą z kolei - `SLESSolve()`...).

Po wywołaniu funkcji `SLESSolve()` należy wywołać funkcję **KSPComputeExtremeSingularValues(KSP ksp, double *emax, double *emin)**, która podaje najmniejszą i największą wartość własną operatora użytego w konkretnej metodzie iteracyjnej.

```
KSP ksp;
double emax,emin;
```

[18] W wersji 2.0.17 stosowany był domyślny rozmiar okienka graficznego, w wersji 2.0.21 (nie wiadomo czemu) trzeba podać tą opcję wraz z parametrami okna czyli: **-ksp_xmonitor 0 0 200 100** spowoduje wyświetlenie okna z lewym górnym rogiem w punkcie (0,0) o szerokości 200 i wyskości 100 pikseli. To może cały czas zmieniać się w zależności od wersji. Sprawdź, jak to działa w *obecnej* wersji!

[19] Jak wiadomo, $cond(\mathbf{A}) = \dfrac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$ dla $A=A^T>0$ (pojęcie współczynnika uwarunkowania macierzy i jego znaczenie dla iteracyjnego rozwiązywania układów liniowych jest wytłumaczone w [6]).

```
...
SLESGetKSP(sles,&ksp);
KSPSetMethod(ksp,KSPCG);
KSPSetComputeSingularValues(ksp);
SLESSolve(sles);
KSPComputeExtremeSingularValues(ksp,&emax,&emin);
PetscPrintf(MPI_COMM_WORLD,"Uwarunkowanie operatora wynosi %d\n",emax/emin);
...
```

Można także wykorzystać opcję PETSc `-ksp_plot_eigenvalues`, która powinna wyświetlić wyliczony w/w metodami przybliżony rozkład wartości własnych macierzy naszego układu.

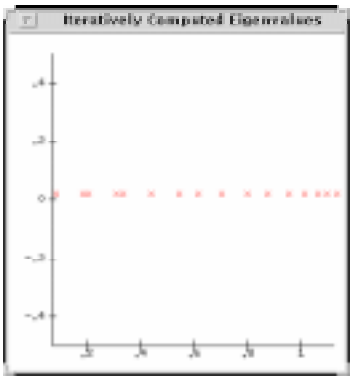
7.5 Ćwiczenia

A) Porównaj swoją metodę Richardsona z zaimplementowaną w PETSc.

B) Używając `-ksp_type <typ>` i `-ksp_xmonitor`, zobacz, która z dostępnych metod iteracyjnych najszybciej rozwiązuje jednowymiarowe zagadnienie Poissona.

C) Zbadaj, jak parametry zbieżności `rtol` i `atol` wpływają na liczbę iteracji i *błąd* rozwiązania.

A) Użyj `KSPComputeEigenvalues()` dla oszacowania uwarunkowania macierzy 1-wymiarowego laplasjanu. Porównaj wynik z prawdziwymi wartościami.



Rysunek 6. Wynik działania opcji `-ksp_plot_eigenvalues`.

8. PC, czyli preconditionery

Jedną z zalet pakietu PETSc jest łatwość używania preconditionerów przy rozwiązywaniu układów równań liniowych. PETSc ma wbudowane wiele gotowych preconditionerów; co więcej, można łatwo dodawać własne. Idea preconditioningu[20] dla układu równań liniowych

$$Ax = b$$

polega na zastąpieniu tego układu równoważnym, mnożąc układ wyjściowy przez odwrotność pewnej macierzy B, zwanej preconditionerem:

$$B^{-1}Ax = B^{-1}b\,.$$

Jeżeli teraz zaczniemy rozwiązywać powyższy układ metodą iteracyjną, np. metodą iteracji prostej to otrzymujemy następujące wzory na kolejne przybliżenia rozwiązania:

$$x_{k+1} = x_k - B^{-1}Ax_k + B^{-1}b = x_k - B^{-1}(Ax_k - b)\,.$$

Oznaczając $r_k = Ax_k - b$ mamy:

$$x_{k+1} = x_k - B^{-1}r_k\,,$$

czyli w każdym kroku iteracyjnym mnożymy wektor residualny przez odwrotność macierzy B: po prostu rozwiązujemy układ z macierzą preconditionera. Od macierzy B żądamy:

- po pierwsze, aby łatwo się odwracała (to znaczy, żeby koszt rozwiązania układu z macierzą B był istotnie niższy od kosztu rozwiązania wyjściowego układu z macierzą A) - żeby łatwo było obliczać poprawkę $B^{-1}r_k$;
- po drugie, aby macierz $B^{-1}A$ miała istotnie niższy współczynnik uwarunkowania niż sama macierz A: jak wiadomo, prędkość zbieżności metod iteracyjnych zazwyczaj silnie zależy do współczynnika uwarunkowania macierzy układu; zmniejszenie tego współczynnika poważnie wpływa na obniżenie liczby iteracji i tym samym - kosztu rozwiązania całego układu równań; szczegóły można znaleźć w [6].

Obecnie poszukiwanie coraz efektywniejszych preconditionerów, zwłaszcza dla najnowocześniejszych architektur komputerowych (np. masywnie równoległych), stanowi jeden z ważniejszych obszarów zainteresowań analizy numerycznej, o bezpośrednim zastosowaniu w praktycznych symulacjach komputerowych.

---

[20] *Preconditioning*, czyli wstępne poprawienie współczynnika uwarunkowania (*condition number*). Czasami używa się polskiego „przewarunkowanie". Od tego słowa bierze się *preconditioner* - co można przetłumaczyć jako „przewarunkowywacz". O ile czasownik brzmi w miarę przyzwoicie po polsku, to rzeczownik - co najmniej dziwnie. Dlatego będziemy konsekwentnie trzymali się formy czysto angielskiej. Podobny kłopot mieliśmy ze słowem *solver*.

Do kontroli preconditionera służy obiekt typu PC, który jest częścią obiektu KSP. Intuicyjnie wydaje się to oczywiste: konkretna metoda iteracyjna praktycznie zawsze musi być stosowana z preconditionerem, na przykład możemy wybrać metodę sprzężonych gradientów z preconditionerem Addytywnej Metody Schwarza. W tym rozdziale zostaną omówione podstawowe zasady korzystania z PC, następnie typy preconditionerów zawarte w pakiecie oraz, dokładnie, metoda tworzenia własnych preconditionerów.

W świetle tego co przed chwilą stwierdziliśmy, każdy utworzony obiekt typu KSP i SLES zawiera już obiekt PC. Aby uzyskać do niego bezpośredni dostęp należy skojarzyć go ze zmienną typu PC przy pomocy funkcji **`SLESGetPC(`**`SLES sles,PC *pc`**`)`**;

```
SLES sles;
PC pc;
...
SLESCreate(MPI_COMM_WORLD,&sles);
SLESGetPC(sles,&pc);
...
```

Teraz możemy już działać na obiekcie PC, stowarzyszonym z obiektem typu SLES, który zostanie uwzględniony w trakcie rozwiązywania zadania, czyli po wywołaniu funkcji `SLESSolve()`. Jak widać, nie musimy najpierw wydobywać ze SLESu KSP, aby potem dotrzeć do PC. Formalnie PC jest częścią KSP związanego ze SLESem, a nie samego SLESu, ale ze względu na powszechność stosowania preconditionerów w trakcie rozwiązywania równań liniowych autorzy pakietu napisali także funkcję `SLESGetPC()`.

Najczęściej wykorzystywaną operacją na PC jest **`PCSetType(PC pc,PCType type)`** - funkcja ustawiająca typ preconditionera. Pierwszy argument to obiekt PC dla którego ustawiamy typ, drugim jest jeden z predefiniowanych typów:

OPIS PRECONDITIONERA	NAZWA STAŁEJ	OPCJA Z LINII KOMEND
Jacobi	PCJACOBI	jacobi
Blokowy Jacobi	PCBJACOBI	bjacobi
Blokowy GaussSeidel (tylko sekwencyjny)	PCBGS	bgs
SOR (i SSOR)	PCSOR	sor
SOR z trickiem Eisenstat'a	PCEISENSTAT	eisenstat
Niekompletny rozkład Choleskiego	PCICC	icc
Niekompletny rozkład LU	PCILU	ilu
Addytywna metoda Schwarza	PCASM	asm
Eliminacja Gaussa LU	PCLU	lu
Bez preconditionera	PCMINUS_ONE	minus_one
Zdefiniowany przez użytkownika	PCSHELL	shell

Preconditioner PCLU jest po prostu metodą eliminacji Gaussa *bez* wyboru elementu głównego - jest to metoda bezpośrednia. Uczynienie jej preconditionerem pozwoliło na wspólny schemat wywołania metod iteracyjnych i metod bezpośrednich. Preconditionera PCLU używa się z metodą KSPPREONLY, czyli bez metody iteracyjnej - stosujemy jedynie preconditioner. Zatem, jeśli chcemy rozwiązać układ równań bezpośrednio, można to zrobić tak:

```
SLES sles;
KSP ksp;
PC pc;
...
SLESCreate(MPI_COM_WORLD,&sles);
...
SLESGetKSP(sles,&ksp);
KSPSetType(ksp,KSPPREONLY);
SLESGetPC(sles,&pc);
PCSetType(pc,LU);
...
SLESSolve(sles);
```

Jeżeli natomiast chcemy jako preconditioner wykorzystać niekompletny rozkład Choleskiego, to trzeba zrobić tak:

```
...
SLESCreate(MPI_COMM_WORLD,&sles);
SLESGetPC(sles,&pc);
PCSetType(pc,PCICC);
...
```

Zmieniliśmy tylko preconditioner, a metoda iteracyjna pozostaje bez zmian. Każdy z typów preconditionera ma zbiór sobie właściwych funkcji, służących do ustawiania parametrów właściwych tylko dla niego. Na przykład, dla addytywnej metody Schwarza mamy funkcje:

FUNKCJA	OPIS
`PCASMGetSubSLES()`	Wydobywa z preconditionera tablicę SLESów służących do rozwiązywania zadania w podobszarach.
`PCASMSetLocalSubdomains()`	Ustawia ile obszarów obsługuje procesor wywołujący tę funkcję.
`PCASMSetOverlap()`	Ustawia szerokość zakładki między obszarami.
`PCASMSetTotalSubdomains()`	Ustawia całkowitą liczbę podobszarów.
`PCASMSetType()`	Ustawia rodzaj metody Schwarza – sposób w jaki dane będą przekazywane między procesorami.

Bardzo skrócony opis dotyczący funkcji regulujących konkretne preconitionery można znaleźć w *manualu*, szczegóły dotyczące składni wszystkich procedur - w *man pages*.

8.1 Podsumowanie

Ustawić preconditioner według własnego uznania można na dwa sposoby:

- W linii komend przy uruchamianiu programu podać opcję `-pc_type <typ>`. Należy pamiętać o wywołaniu funkcji `SLESSetFromOptions()`.

Drugim sposobem jest ustawienie preconditionera z poziomu kodu źródłowego funkcją `PCSetType()`. W tym celu należy:

- zadeklarować zmienną typu PC
- wydobyć ze SLESu obiekt typu PC funkcją `SLESGetPC()`
- ustawić preconditioner funkcją `PCSetType()`

8.2 Tworzenie własnych preconditionerów

W PETSc mamy możliwość łatwego budowania nowych preconditionerów dla konkretnych metod iteracyjnych. Do tworzenia własnych preconditionerów służy typ preconditionera `PCSHELL`. Jak sama nazwa wskazuje, będzie on definiowany analogicznie jak macierze powłokowe. Dlatego wraz z takim preconditionerem trzeba zdefiniować sposób, w jaki działa on na wektor residualny, funkcją **`PCShellSetApply(PC pc, int (*wlasny_pc)(void*,Vec,Vec),void *ptr).`** Pierwszy argument tej funkcji to zmienna typu PC obsługująca preconditioner. Drugi to wskaźnik do funkcji napisanej przez użytkownika działającej na wektor residualny. Trzeci argument to wskaźnik do danych zewnętrznych potrzebnych funkcji `wlasny_pc()` (oczywiście, gdy funkcja ich nie potrzebuje, wstawiamy `PETSC_NULL`). Funkcja `wlasny_pc()` musi mieć trzy argumenty. Pierwszy to wskaźnik do kontekstu używanego przez funkcję, może być `PETSC_NULL`. Druga i trzecia zmienna to odpowiednio wektory: residualny, otrzymywany przez preconditioner, i wynik działania na wektorze residualnym.

Tak więc, aby wstawić do SLESu własny preconditioner należy:

- zadeklarować zmienną typu PC
- zdefiniować funkcję działającą preconditionerem na wektor residualny (zwróćmy uwagę, że preconditioner działa analogicznie jak macierz typu powłokowego)
- wydobyć funkcją `SLESGetPC()` z obiektu typu SLES obiekt PC wiążąc go z uprzednio zadeklarowaną zmienną typu PC
- ustawić preconditioner na powłokowy (`PCSHELL`) funkcją `PCSetType()`
- przekazać do preconditionera funkcję realizującą jego działanie na wektor residualny: `PCShellSetApply()`

8.3 Przykłady

Przykład trywialny

Ten program włącza własny „preconditioner" do SLESu. W charakterze preconditionera użyjemy macierzy $B=I$. Nie jest to oczywiście macierz, która będzie w jakikolwiek poprawiała własności spektralne układu, ale za to mnożenie przez B^{-1} jest szczególnie tanie... polegać będzie po prostu na skopiowaniu wektora. Za każdym razem, gdy zostanie wywołana funkcja `SLESSolve()` wektor residualny będzie przechodził przez funkcję `Pierwszy_pc` (i nic z nim się nie będzie dzialo - funkcja `VecCopy()`). Funkcja `PCShellSetName(pc,"Mój pierwszy preconditioner")` nadaje nazwę preconditionerowi powłokowemu. Przy wypisywaniu informacji na temat SLESu pojawi się właśnie ta nazwa. Pliki `common.h` i `common.c` są podane w przykładzie podsumowującym w następnym rozdziale; znajduje się tam także `Makefile` służący do skompilowania przykładów.

`przyklad8_1.c`

```
#include "common.h"
static char help[] ="Preconditioner trywialny\n";

/**********************************************
        Działanie preconditionera
**********************************************/

int Pierwszy_pc(void *ctx,Vec v0,Vec v1)
{
 VecCopy(v0,v1);
 return(0);
}

/**********************************************
           Program glowny
**********************************************/

int main(int argc,char **args)
{
 PetscInitialize(&argc,&args,PETSC_NULL,help);
 Przygotuj_zadanie(N,&sles,&A,&u0,&u,&f);

/*ustawienie preconditionera powłokowego */
 SLESGetPC(sles,&pc);
 PCSetType(pc,PCSHELL);
 PCShellSetApply(pc,Pierwszy_pc,PETSC_NULL);
 PCShellSetName(pc,"Mój pierwszy preconditioner");

/*rozwiązanie zadania */
 Rozwiaz(sles);

/*Zwalnianie pamięci*/
 Zwolnij_pamiec(sles,A,u,u0,f);
 PetscFinalize();
 return(0);
}
```

Po uruchomieniu mamy:

```
hydra:/usr/home/bin/> przyklad

Iteracje = 351
 Blad    = 6.364062e-04

hydra:/usr/home/bin/>
```

Aby zobaczyć efekt `PCShellSetName()` trzeba uruchomić program z opcją `-sles_view`.

Preconditioner Richardsona.

Ten preconditioner stosuje przeskalowanie wektora w stylu poprawki Richardsona, to znaczy, jako preconditioner B wybieramy macierz:

$$B = \begin{bmatrix} \tau^{-1} & & \\ & \ddots & \\ & & \tau^{-1} \end{bmatrix} = \tau^{-1} \cdot I ,$$

dla odpowiednio dobranej wartości parametru τ. Jak wiadomo, najlepszy parametr metody Richardsona uzyskuje się, korzystając z najmniejszej i największej wartości własnej macierzy (opis metody Richardsona można znaleźć w [6]). Dlatego jako kontekst podane zostaje oszacowanie na spektrum macierzy. Podobnie jak w poprzednim przypadku, pliki `common.h` i `common.c` są podane w następnym rozdziale.

przyklad8_2.c

```
#include "common.h"
static char help[] ="Preconditioner Richardsona\n";

/*************************************************
          Kontekst dla preconditionera
*************************************************/

typedef struct {
   double lambdamin;
double lambdamax;} Kontekst;

Kontekst k_pc;

/*************************************************
          Dzialanie preconditionera
*************************************************/

int Richardson_pc(void *ctx,Vec v0,Vec v1)
{
 Scalar val=1;
 Kontekst *kn=(Kontekst*)ctx;
 val=2/(kn->lambdamin+kn->lambdamax);
 VecCopy(v0,v1);
 VecScale(&val,v1);
 return(0);
}

/*************************************************
          Program glowny
*************************************************/

int main(int argc,char **args)
{
 PetscInitialize(&argc,&args,PETSC_NULL,help);
 Przygotuj_zadanie(N,&sles,&A,&u0,&u,&f);
 SLESGetPC(sles,&pc);
 PCSetType(pc,PCSHELL);
 PCShellSetApply(pc,Richardson_pc,(void*)&k_pc);
 PCShellSetName(pc,"Preconditioner Richardsona");
 k_pc.lambdamin=1.0/N;
 k_pc.lambdamax=1.0;
 Rozwiaz(sles);
 Zwolnij_pamiec(sles,A,u0,u,f);
 PetscFinalize();
 return(0);
}
```

Po uruchomieniu mamy:

```
hydra:/usr/home/bin/> przyklad

Iteracje = 80
 Blad    = 4.366761e-04

hydra:/usr/home/bin/>
```

Preconditioner Jacobi'ego.

Teraz zaimplementujemy preconditioner Jacobi'ego, mnożący wektor residualny przez odwrotność diagonali operatora. Macierz preconditionera ma zatem następującą postać:

$$B = \begin{bmatrix} a_{11} & & \\ & \ddots & \\ & & a_{nn} \end{bmatrix}$$

gdzie a_{ii} oznaczają elementy diagonalne macierzy zadania A. W implementacji tego preconditionera skorzystamy

więc ze zmiennej kontekstowej: jako kontekst podamy główną diagonalę macierzy zadania. Zwróćmy uwagę na funkcję `PrzygotujPreconditioner()` - wykonuje on podobną pracę, co w standardowym przypadku funkcja `PCSetUp()`.

przyklad8_3.c

```
#include "common.h"
static char help[] ="Preconditioner Jacobi'ego";

/**********************************************
    Inicjalizacja kontekstu preconditionera
**********************************************/

int PrzygotujPreconditioner(Mat A,Vec *k_pc)
{
 int m,n;
 MatGetSize(A,&m,&n);
 If(m!=n)
 {
  PetscPrintf(MPI_COMM_WORLD,"Hej!! Macierz nie jest kwadratowa\n");
 }
 VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,m,k_pc);
 MatGetDiagonal(A,(*k_pc));
 return(0);
}

/**********************************************
          Dzialanie preconditionera
**********************************************/

int wlasny_pc(void *ctx,Vec v0,Vec v1)
{
 Vec *k_pc=(Vec*)ctx;
/* Dzialanie preconditionerem Jacobiego */
 VecPointwiseDivide(v0,*k_pc,v1);
 return(0);
}

/**********************************************
             Program glowny
**********************************************/

int main(int argc,char **args)
{
 Vec diag;

 PetscInitialize(&argc,&args,PETSC_NULL,help);
 Przygotuj_zadanie(N,&sles,&A,&u0,&u,&f);
 SLESGetPC(sles,&pc);
 PCSetType(pc,PCSHELL);
 PrzygotujPreconditioner(A,&diag);
 PCShellSetApply(pc,wlasny_pc,&diag);
 PCShellSetName(pc,"Preconditioner Jacobi'ego");
 Rozwiaz(sles);
 PetscFinalize(); return(0);
}
```

Po uruchomieniu mamy:

```
hydra:/usr/home/bin/> przyklad

Iteracje = 63
 Blad    = 4.364761e-04

hydra:/usr/home/bin/>
```

Jak widać, macierz *B* spełnia postulaty preconditionera: jest łatwo odwracalna (jak każda macierz diagonalna) i jednocześnie poprawia współczynnik uwarunkowania zadania, co widać po liczbie wykonanych iteracji.

8.4 Przykład podsumowujący: addytywna metoda Schwarza z grubą siatką

Ten przykład demonstruje zastosowanie znacznie bardziej wyrafinowanego preconditionera - addytywną metodę Schwarza z grubą siatką. W PETSc jest gotowy preconditioner Schwarza, ale działający jedynie na siatkach jedno-

rodnych na prostokącie. Gdy chcemy użyć preconditionera z gruba siatką, musimy napisać własny.

przyklad8_4.c

```
#include "common.h"

static char help[] ="Addytywny preconditioner Schwarza z grub¹ siatk¹
                                Marcin Bojko";

/* Tutaj definiujemy strukture, która bedzie kontekstem preconditionera */
typedef struct {
 Mat I,Ag;               /* macierz interpolacji i zadania na grubej siatce */
 int rozmiar,mg;         /* mg to rozmiar grubej siatki*/
 int m,M,zakl;           /* M to iloœæ podobszarów w jednym kierunku */
                         /* wszystkich podobszarów bêdzie M*M */
 IS *boxy;               /* tu bêdzie sk³adowana dekompozycja */
 Vec fg,ug;              /* pomocnicze wektory zadania na grubej siatce */
 Scalar value[3],val;
 SLES slesg;             /* SLES dla zadania na gubej siatce */
 PC pc_box;              /* preconditioner Schwarza na boxach */
} asm_ctx;

int Preconditioner(void *pointer,Vec u0,Vec u1);

void Przygotuj_macierz_interpolacji(asm_ctx *ctx)
{
 int i,j,row,col,r,k,l;
 int mg=ctx->mg,m=ctx->m,rozmiar=ctx->rozmiar;

 Scalar val;
 MatCreate(MPI_COMM_WORLD,rozmiar*rozmiar,mg*mg,&ctx->I);
 for(j=0;j<mg;j++)
 for(i=0;i<mg;i++)
 {
  col=i+mg*j;
  row=r=(m-1)*rozmiar+m-1+i*m+j*m*rozmiar;
  val=1;
  MatSetValues(ctx->I,1,&row,1,&col,&val,INSERT_VALUES);
  for(k=1;k<m;k++)
  {
   val=(Scalar)(m-k)/m;
   for(l=0;l<6*k;l++)
   {
    if((l>1)&&(l<k))row=r-k*rozmiar+l;
    if((l>k-1)&&(l<2*k))row=r-k*rozmiar+k+(l-k)*rozmiar;
    if((l>2*k-1)&&(l<3*k))row=r+k+(l-2*k)*(rozmiar-11);
    if((l>3*k-1)&&(l<4*k))row=r+k*rozmiar-(l-3*k);
    if((l>4*k-1)&&(l<5*k))row=r+k*(rozmiar-1)-(l-4*k)*rozmiar;
    if((l>5*k-1)&&(l<6*k))row=r-k-(l-5*k)*(rozmiar-1);
    MatSetValues(ctx->I,1,&row,1,&col,&val,INSERT_VALUES);
   }
  }
 }
 MatAssemblyBegin(ctx->I,MAT_FINAL_ASSEMBLY);
 MatAssemblyEnd(ctx->I,MAT_FINAL_ASSEMBLY);
 MatCompress(ctx->I);
}

Przygotuj_preconditioner(asm_ctx *ctx,PC *pc_shell)
{
 int flag;
 int rozmiar=ctx->rozmiar,M=ctx->M;
 int zakl=ctx->zakl,mg=ctx->mg;
 PC pcg;
 KSP ksp;
```

```
 /*Przygotowanie preconditionera Schwarza */
 PCCreate(MPI_COMM_WORLD,&ctx->pc_box);
 PCSetType(ctx->pc_box,PCASM);
 PCASMCreateSubdomains2D(rozmiar,rozmiar,M,M,1,zakl,&i,&ctx->boxy);
 PCASMSetLocalSubdomains(ctx->pc_box,i,ctx->boxy);
 PCSetOperators(ctx->pc_box,A,A,SAME_PRECONDITIONER);
 PCSetVector(ctx->pc_box,u);
 PCASMSetOverlap(ctx->pc_box,0);
 PCSetUp(ctx->pc_box);

 /* gruba siatka */
 VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,mg*mg,&ctx->fg);
 VecDuplicate(ctx->fg,&ctx->ug);
 Przygotuj_macierz_laplasjanu(&ctx->Ag,mg,mg);
 SLESCreate(MPI_COMM_WORLD,&ctx->slesg);
 SLESSetOperators(ctx->slesg,ctx->Ag,ctx->Ag,SAME_PRECONDITIONER);
 SLESGetPC(ctx->slesg,&pcg);
 PCSetType(pcg,PCLU);
 SLESGetKSP(ctx->slesg,&ksp);
 KSPSetType(ksp,KSPPREONLY);
 Przygotuj_macierz_interpolacji(ctx);
}

int main(int argc,char **args)
{
 KSP ksp;
 PC pc_shell;
 asm_ctx ams;
 double norma,emin,emax;
 int i,flag;
 PetscInitialize(&argc,&args,PETSC_NULL,help);
 ams.M=4;
 ams.mg=7;
 ams.zakl=1;
 ams.m=8;
 OptionsGetInt(PETSC_NULL,"M",&ams.M,&flag);
 OptionsGetInt(PETSC_NULL,"mg",&ams.mg,&flag);
 OptionsGetInt(PETSC_NULL,"m",&ams.m,&flag);
 OptionsGetInt(PETSC_NULL,"zakl",&ams.zakl,&flag);
 ams.rozmiar=(ams.mg+1)*ams.m-1;

 Przygotuj_zadanie(ams.rozmiar,&sles,&A,&u0,&u,&f);
 SLESGetKSP(sles,&ksp);
 SLESGetPC(sles,&pc_shell);
 Przygotuj_preconditioner(&ams,&pc_shell);
 PCSetType(pc_shell,PCSHELL);
 PCShellSetApply(pc_shell,Preconditioner,&ams);
 PCShellSetName(pc_shell,"asmbox");
 KSPSetType(ksp,KSPGMRES);
 KSPSetComputeSingularValues(ksp);

 SLESSetFromOptions(sles);
 Rozwiaz(sles);
 KSPComputeExtremeSingularValues(ksp,&emax,&emin);
 PetscFPrintf(MPI_COMM_WORLD,stderr," Uwarunkowanie  = %e\n\n",emax/emin);

 SLESDestroy(ams.slesg);
 MatDestroy(ams.Ag);
 VecDestroy(ams.ug);
 VecDestroy(ams.fg);
 MatDestroy(ams.I);
 Zwolnij_pamiec(sles,A,u0,u,f);
 PetscFinalize(); return(0);
}
```

```
int Preconditioner(void *pointer,Vec v0 ,Vec v1)
{
 asm_ctx *ams=(asm_ctx*)pointer;
 int its;
 /*rozwi¹zanie na boxach */
 PCApply(ams->pc_box,v0,v1);
 /* rozwiazanie na grubej siatce */
 MatMultTrans(ams->I,v0,ams->fg);
 SLESSolve(ams->slesg,ams->fg,ams->ug,&its);
 VecScale(&val,ams->ug);
 MatMultAdd(ams->I,ams->ug,v1,v1);
 return(0);
}
```

Po uruchomieniu mamy:

```
hydra:/usr/home/bin/> przykład

Liczba iteracji = 24
 Blad           = 6.442081e-04
 Uwarunkowanie  = 5.804242e+01

hydra:/usr/home/bin/>
```

Poniżej przytaczamy pliki źródłowe common.c i common.h, z których korzystaliśmy w niniejszym rozdziale.

common.h

```
#include "sles.h"

Vec u,u0,f;
Mat A;
SLES sles;
PC pc;
int N=100,its,i;
Scalar val;
double norma;

int Rozwiaz(SLES sles);
int Przygotuj_zadanie(int rozmiar,SLES *sles,Mat *A,Vec *u0,Vec *u,Vec *f);
int Zwolnij_pamiec(SLES sles,Mat A,Vec u0,Vec u,Vec f);
int Przygotuj_macierz_laplasjanu(Mat *A,int roz_x,int roz_y);
```

common.c

```
#include "common.h"

int Przygotuj_macierz_laplasjanu(Mat *A,int roz_x,int roz_y)
{
 Scalar val;
 int n,m,i,j;
 n=roz_x*roz_y;
 MatCreateSeqAIJ(MPI_COMM_WORLD,n,n,5,PETSC_NULL,A);;
 for(i=0;i<n;i++)
 {
  val=4;
  MatSetValues(*A,1,&i,1,&i,&val,INSERT_VALUES);;
     /* elementy na dalekich kodiagonalach */
  val=-1;
  j=i+roz_y;
  if(j<n)
  {
   MatSetValues(*A,1,&i,1,&j,&val,INSERT_VALUES);;
   MatSetValues(*A,1,&j,1,&i,&val,INSERT_VALUES);;
  }
```

```
    /* elementy na bliskich kodiagonalach */
  j=i+1;
  val=-1;
  if(j<n)if(((i+1)%roz_x)!=0||(i==0))
  {
   MatSetValues(*A,1,&i,1,&j,&val,INSERT_VALUES);;
   MatSetValues(*A,1,&j,1,&i,&val,INSERT_VALUES);;
  }
 }
 MatAssemblyBegin(*A,MAT_FINAL_ASSEMBLY); ;
 MatAssemblyEnd(*A,MAT_FINAL_ASSEMBLY); ;
 MatCompress(*A);
 return(0);
}

/*Przygotowanie zadania testowego */
int Przygotuj_zadanie_testowe(int N,SLES *sles,Mat *A,Vec *u0,Vec *u,Vec *f)
{
 PetscRandom rctx;
 VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,N*N,f);
 VecDuplicate(*f,u);
 VecDuplicate(*f,u0);
 Przygotuj_macierz_laplasjanu(A,N,N);
 SLESCreate(MPI_COMM_WORLD,sles);
 SLESSetOperators(*sles,*A,*A,SAME_PRECONDITIONER);
 PetscRandomCreate(MPI_COMM_WORLD,RANDOM_DEFAULT,&rctx);
 VecSetRandom(rctx,*u0);
 PetscRandomDestroy(rctx);
 MatMult(*A,*u0,*f);
 return(0);
}

/* Zwolnienie pamięci po zadaniu testowym */
int Zniszcz_zadanie_testowe(SLES sles,Mat A,Vec u0,Vec u,Vec f)
{
 SLESDestroy(sles);
 VecDestroy(u0);
 VecDestroy(u);
 VecDestroy(f);
 MatDestroy(A);
}
int Rozwiaz(SLES sles)
{
 SLESSetFromOptions(sles);
 SLESSolve(sles,f,u,&its);

/*Podanie wynikow*/
 val=-1;
 VecAXPY(&val,u,u0);
 VecNorm(u0,NORM_2,&norma);
 PetscFPrintf(MPI_COMM_WORLD,stderr,"\nLiczba iteracji = %i\n",its);
 PetscFPrintf(MPI_COMM_WORLD,stderr," Blad           = %e\n",(double)norma);
}
```

`Makefile` (do skompilowania powyższych przykładów)

```
BOPT=O
include $(PETSC_DIR)/bmake/$(PETSC_ARCH)/base

CFLAG          = $(CPPFLAGS) -D__SDIR__='"$(LOCDIR)"' $(BS_INCLUDE)
LIBBASE        = libpetscsles
LOCDIR         = /home/students/bojko/source/

wszystko: przykład8_1 przykład8_2 przykład8_3 przykład8_4
```

```
przykład8_1: przyklad8_1.o
             $(CLINKER) -o przyklad8_1 przyklad8_1.o  common.o
                   $(PETSC_LIB)
             $(RM) przyklad8_1.o

przyklad8_2: przyklad8_2.o
             $(CLINKER) -o przyklad8_2 przyklad8_2.o common.o
                   $(PETSC_LIB)
              $(RM) przyklad8_2.o

przyklad8_3: przyklad8_3.o
             $(CLINKER) -o przyklad8_3 przyklad8_3.o common.o
                   $(PETSC_LIB)
              $(RM) przyklad8_3.o

przyklad8_4: przyklad8_4.o
             $(CLINKER) -o przyklad8_4 przyklad8_4.o common.o
                   $(PETSC_LIB)
              $(RM) przyklad8_4.o
```

Aby skompilować powyższe przykłady należy uruchomić program `make` z nazwą pliku, który chcemy skompilować, na przykład:

```
hydra:/usr/home/students/bojko/>make przyklad8_1
```

Plik wykonywalny otrzyma nazwę `przyklad8_1`, więc będziemy go uruchamiać, pisząc

```
hydra:/usr/home/students/bojko/>przyklad8_1
```

Można też od razu skompilować wszystkie programy (pod warunkiem, że wszystkie pliku źródłowe są w jednym katalogu) poleceniem:

```
hydra:/usr/home/students/bojko/>make wszystko
```

9. Szczegóły o preconditionerze AMS, których nie znajdziecie w oficjalnym manualu

W tym rozdziale zostanie szczegółowo omówiony preconditioner addytywnej metody Schwarza w wersji zaimplementowanej w pakiecie PETSc. Informacje tutaj zawarte istotne są tylko dla tych czytelników, którzy chcą używać preconditionera Schwarza - część z nich została uzyskana na drodze prób i błędów, a część pochodzi dzięki konsultacji z autorami pakietu (autorzy szybko i chętnie odpowiadają na pytania których nie ma w FAQ). Czytelnik, któremu bardziej zależy na ogólnym poznaniu pakietu PETSc, może lekturę poniższego rozdziału odłożyć na później.

Załóżmy, że mamy już wygenerowany obiekt PC (nieważne, czy wydobyty ze SLESu funkcją `SLESGetPC()`, czy wygenerowany funkcją `PCCreate()`) i ustawione operatory. Aby ustawić preconditioner Schwarza, należy wywołać funkcję: **`PCSetType(pc,PCASM)`**. Funkcje służące do sterowania tym preconditionerem pozwalają na automatyzację tworzenia jego struktury. W zasadzie jedyną rzeczą, jaką użytkownik musi sam zaimplementować, jest dekompozycja obszaru.

Funkcje ASM można podzielić na trzy zasadnicze typy:

- Służące do generowania i obsługi dekompozycji: `PCASMCreateSubdomains2D()`, `PCASMSetOverlap()`.
- Służące do kontroli sposobu działania: `PCASMGetSubSLES()`, `PCASMSetType()`.
- Służące do kontroli implementacji równoległej `PCASMSetLocalSubdomains()` i `PCASMSetTotalSubdomains()`.

9.1 Zbiory indeksów

Strukturą potrzebną do obsługi dekompozycji obszaru są zbiory indeksów (ang. *Index Sets*). Jak wszystkie obiekty w PETSc, zbiory indeksów muszą być powiązane ze zmienną określonego typu, w tym wypadku `IS`. Na potrzeby preconditionera Schwarza trzeba omówić funkcję **`ISCreateGeneral(MPI_Comm comm,int n,int *idx,IS *is)`** Pierwszy argument to jak zwykle komunikator MPI. Drugi argument to liczba indeksów w generowanym IS'ie. Trzeci argument to tablica zmiennych typu `int`, w której zapisane są po kolei numery indeksów,

które chcemy wstawić do generowanego IS'a, a czwarty argument to wskaźnik do zmiennej IS, z którą będzie powiązany generowany obiekt. Inną funkcją służącą do generowania zbiorów indeksów jest **`ISCreateStride(MPI_Comm comm,int n,int first, int step, IS *is).`** Ta funkcja służy do generowania sekwencji indeksów o określonym kroku. Pierwszy i drugi argument mają takie samo znaczenie jak w poprzedniej funkcji. Trzeci to pierwszy indeks w ciągu, a drugi krok przyrostu kolejnych wyrazów.

Przykład

```
...
#define MP MPI_COMM_WORLD
IS is;
Int indeksy[4]={0,5,12,17,22};
...
ISCreateGeneral(MP,5, indeksy,&is);
...
```

W tym przypadku wygenerowany zostanie zbiór indeksów {0,5,12,17,22}

Przykład

```
...
#define MP MPI_COMM_WORLD
IS is;
...
ISCreateStride(MP,5,2,3,&is);
...
```

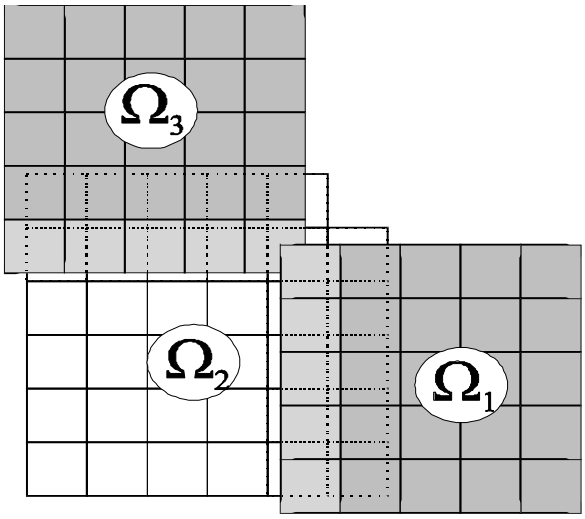
W tym przypadku wygenerowany zostanie zbiór indeksów {2,5,8,11,14} Tak jakby wstawić tablicę wygenerowaną następująco:

```
for(i=0;i<n;i++)indeksy[i]=2+i*3;
```

9.2 Dekompozycja

Dekompozycję obszaru podaje się jako tablicę zbiorów indeksów (IS), gdzie każdy zbiór indeksów podaje numery współrzędnych w jednym podobszarze. Przypuśćmy, że mamy obszar L-kształtny. **Rysunek 7,** pokazuje jak ponumerowane są węzły siatki. Cała funkcja siatkowa na tym obszarze zapisywana jest jako wektor o 48 współrzędnych. Wprowadzamy dekompozycję na trzy podobszary jak na **Rysunku 8**. Dekompozycję w tym przypadku zapisujemy więc jako 3-elementową tablicę zmiennych typu IS. Będą to następujące zbiory:

- is[0]←{19,..,23,27,...,31,35,...,39,43,...,47 }
- is[1]←{16,..,20,24,..28,32..,36,40,...,44}



Rysunek 8. Dekompozycja obszaru.

	0	1	2	3				
	4	5	6	7				
	8	9	10	11				
	12	13	14	15				
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39
	40	41	42	43	44	45	46	47

Rysunek 7. Numeracja węzłów siatki.

- is[3]←{0,...,19}

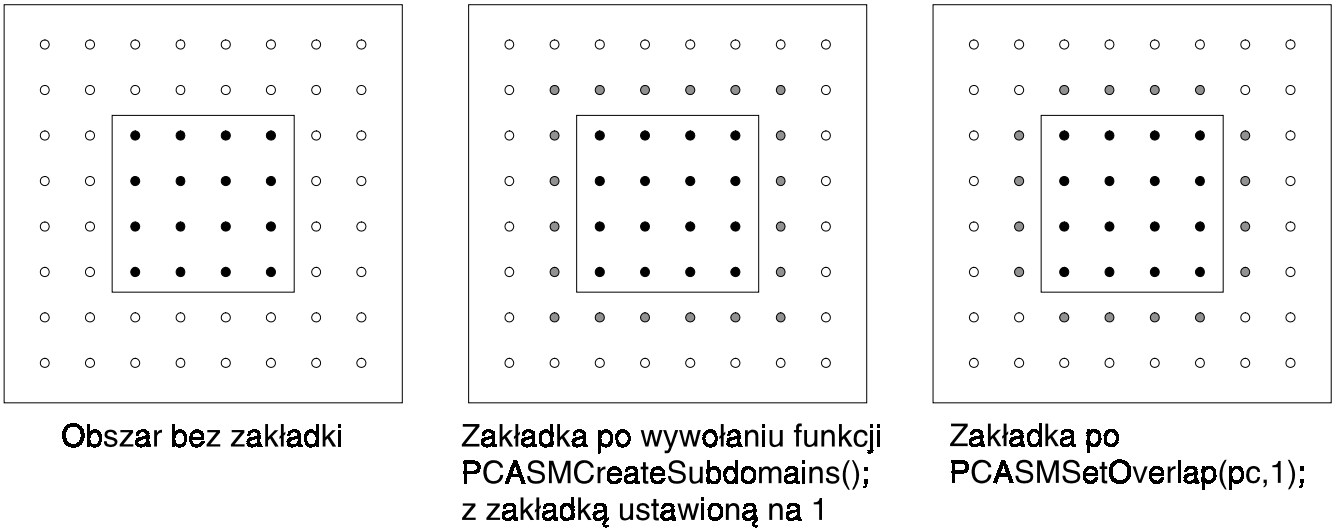
Przy podawaniu dekompozycji musimy sami zdecydować, czy zdajemy się na funkcję automatycznie ustawiającą wielkość zakładki, czy zakładkę definiujemy sami i takie zbiory indeksów musimy wygenerować. W powyższym przykładzie dekompozycja została zdefiniowana z zakładkami, bo tak uzyskujemy pełną kontrolę nad geometrią podobszarów. Zbiór indeksów wstawiamy do PC przy pomocy funkcji **`PCASMSetTotalSubdomains(PC pc, int N, IS *is).`** Pierwszy argument to zmienna typu PC służąca do kontroli obiektu, drugi to całkowita liczba podobszarów a trzeci to tablica zbiorów indeksów opisująca dekompozycję. Ta funkcja automatycznie rozdysponowuje podobszary pomiędzy procesory. Można też samemu zdecydować ile podobszarów będzie w poszczególnych procesorach przy pomocy funkcji **`PCASMSetLocalSubdomains(PC pc, int n, IS *is)`** Pierwszy argument to zmienna typu PC powiązana z obiektem, drugi to liczba podobszarów w danym procesorze, a trzeci to tablica zbiorów indeksów z podobszarami dla tego procesora. Ostatnia zmienna może przyjąć wartość PETSC_NULL, gdy PETSc ma samo zdecydować, które podobszary mają trafić do danego procesora. Gdy powyższe funkcje są wywoływane w programie wykonywanym wielowątkowo, to muszą być wywołane przez wszystkie procesy w obrębie komunikatora MPI, przy czym pierwsza musi być wywołana z tą samą we wszystkich wątkach tablicą zbiorów indeksów.

Dodatkowo autorzy dołączyli do pakietu funkcję **PCASMCreateSubdomains2D(int m,int n,int M,int N,int dof,int overlap,int *Nsub,IS **is)** która służy do utworzenia dekompozycji prostokątnego obszaru dwuwymiarowego z regularną siatką na `M*N` prostokątnych podobszarów. Argumenty tej funkcji to po kolei: rozmiar siatki w poziomie, rozmiar siatki w pionie, liczba podobszarów w poziomie, liczba podobszarów w pionie, liczba stopni swobody siatki, szerokość zakładki, wskaźnik do zmiennej, w której znajdzie się całkowita liczba utworzonych podobszarów i ostatni argument to wskaźnik do tablicy zbiorów indeksów, która zostanie wygenerowana przez tę funkcję.

9.3 Zakładka

Następnym krokiem, który trzeba wykonać jest ustawienie wielkości zakładki pomiędzy podobszarami. Robi się to funkcją **PCASMSetOverlap(PC pc, int ovl).** Pierwszy argument tej funkcji to zmienna typu PC powiązana z obiektem, a drugi to szerokość zakładki w punktach. Ta funkcja spowoduje, że zbiory indeksów opisujące dekompozycję zostaną automatycznie rozszerzone o zakładkę, ale dopiero po wywoływaniu funkcji `PCSetUp()`[21]. Jeżeli zdefiniowaliśmy dekompozycję z zakładkami należy wywołać funkcję **PCASMSetOverlap(pc,0)**, bo inaczej obszary zostaną rozszerzone o dodatkową zakładkę.

Zakładka generowana automatycznie tworzona jest na podstawie niezerowej struktury macierzy. Konkretne problemy mogą prowadzić do innych struktur zakładki, na przykład funkcja `PCASMCreateSubdomains2D()` tworzy zakładkę w innym kształcie niż zakładka utworzona automatycznie (**Rysunek 9**), dlatego dobrze jest samemu zdefiniować dekompozycję wraz z zakładkami.



Rysunek 9. Działanie funkcji `PCASMSetOverlap()`

9.4 Wybór typu AMS

W PETSc są zaimplementowane cztery typy AMS:

`PC_ASM_BASIC` metoda podstawowa pełna interpolacja i restrykcja

`PC_ASM_RESTRICT` pełna restrykcja, lokalna interpolacja w procesorze

`PC_ASM_INTERPOLATE` pełna interpolacja, lokalna restrykcja w procesorze

`PC_ASM_NONE` lokalna restrykcja i interpolacja w procesorze

Wszystkie te metody związane są ściśle z implementacją równoległą, to znaczy wszelkie obcięcia i interpolacje odnoszą się do danych pochodzących z innych procesorów. Jeżeli jeden procesor ma kilka podobszarów to w obrębie jego danych stosowana jest zawsze metoda podstawowa - i słusznie, bo wszystkie innowacje mają na celu wyeliminowanie zbędnej komunikacji, która często jest kosztowna, stąd w obrębie jednego procesora nie warto stosować tych metod. Wszystkie metody rozwiązują układ bez grubej siatki.

[21] Jeżeli użytkownik nie wywołał tej funkcji explicite, zostanie ona wywołana na początku wykonywania funkcji `SLESSolve()`.

9.5 Podsumowanie

Sposób tworzenia preconditionera różni się zasadniczo w zależności od tego, czy włączamy go do SLESu jako metodę PCASM, czy jako PCSHELL. W pierwszym przypadku:

1) Deklarujemy zmienną typu PC: `PC pc;`
2) Wydobywamy ze SLESu obiekt PC funkcją `SLESGetPC(sles,&pc)`
3) Ustawiamy typ preconditionera na ASM przy pomocy funkcji `PCSetType(pc,PCASM)`
4) Definiujemy dekompozycję jako zbiór indeksów, można to zrobić na przykład funkcją `PCASMCreateSubdomains2D()`, lub samodzielnie
5) Wstawiamy dekompozycję do PC funkcją `PCASMSetTotalSubdomains(pc,nsub,dekomp)`, gdzie `nsub` to liczba podobszarów a `dekomp` tablica zbiorów indeksów.
6) Ustawiamy zakładkę funkcją `PCASMSetOverlap(pc,zakl)`. Należy pamiętać o tym, że niezerowa wartość zmiennej `zakl` spowoduje automatyczne wygenerowanie dodatkowej zakładki o tej szerokości. Zatem gdy dekompozycja została podana z zakładkami należy zmienną `zakl` ustawić na 0.

Gdy tworzymy własny preconditioner oparty na metodzie dekompozycji obszaru, na przykład powłokowy (*shell*) należy wygenerować go w całości, to znaczy wywołać szereg funkcji, które w poprzednim przypadku są automatycznie wywoływane w funkcji `SLESSolve()`. Takie postępowanie jest niezbędne, gdy chcemy utworzyć preconditioner AMS z grubą siatką, gdyż dorzucenie grubej siatki wymaga dodatkowego operatora restrykcji-interpolacji, który ma inną strukturę niż analogiczne operatory dla podobszarów. Czynności którymi postępowanie w tym wypadku różni się od poprzedniego to:

- W punkcie 2 obiekt PC trzeba samemu wykreować funkcją `PCCreate(&pc)`
- Po punkcie 6 trzeba samemu wywołać funkcję `PCSetUp(pc)`. Spowoduje to utworzenie zakładek, przygotowanie SLESów wektorów dla podobszarów oraz obiektów przerzucających dane między wektorami (Scatter) dla zadań na podobszarach.
- Preconditioner stosujemy używając funkcji `PCApply(PC pc,Vec v0, Vec v1)`

Taki sposób postępowania został przedstawiony w przykładzie ze strony 37.

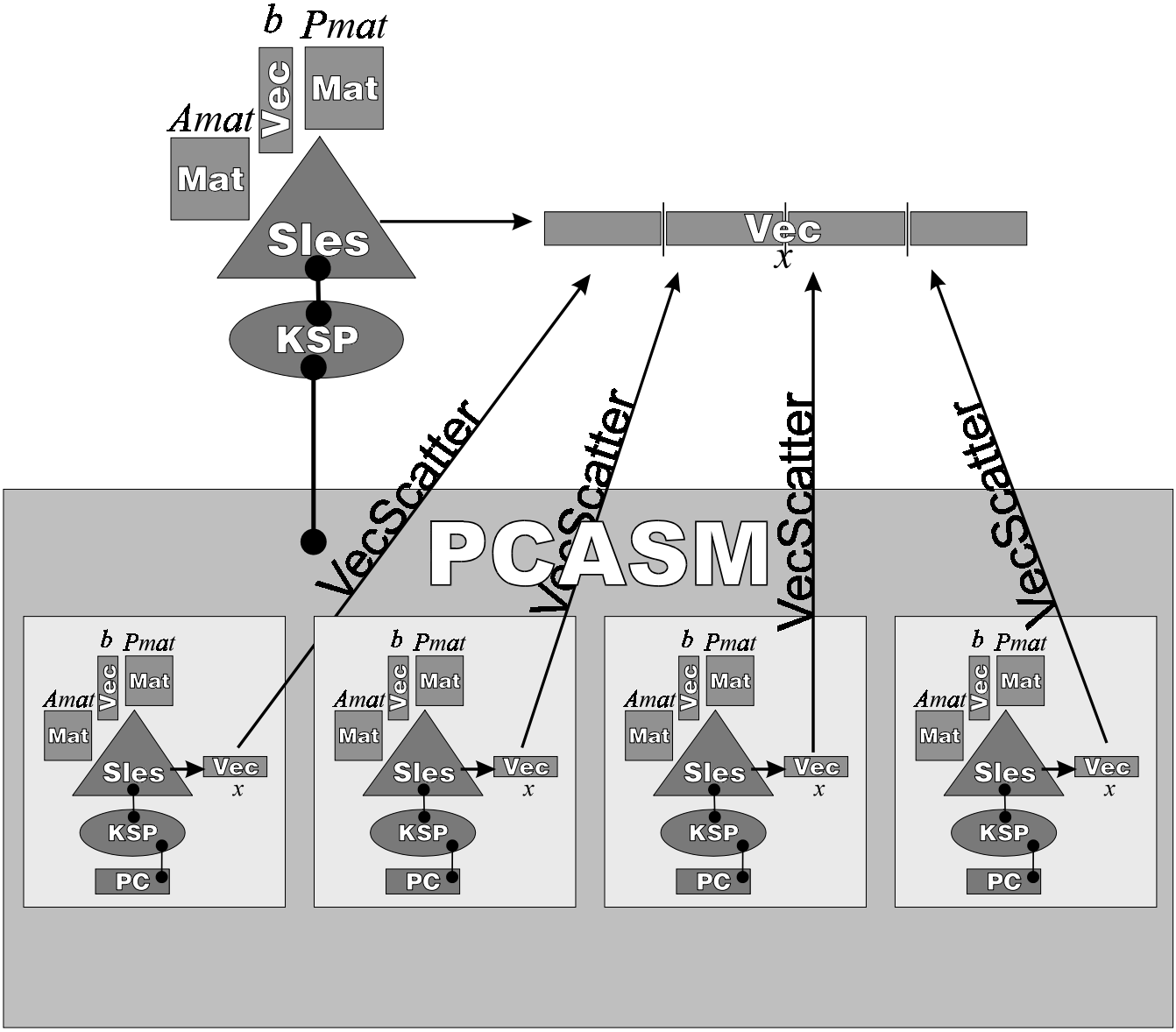
9.6 Szczegóły

Po wywołaniu funkcji `PCSetUp(pc)` gdzie `pc` jest preconditionerem AMS zostanie utworzony szereg komponentów z których składa się nasz obiekt. Obrazuje to Rysunek 10, który pokazuje strukturę preconditionera Schwarza dla czterech podobszarów. Każdy z podobszarów ma własny SLES, wektor i macierze, służące do rozwiązywania zadań lokalnych. Macierze zadań lokalnych tworzone są z macierzy zadania na podstawie zbioru indeksów definiującego podobszar. Wektory zadań lokalnych mają rozmiar wynikający z wielkości macierzy zadania lokalnego. Do przenoszenia danych pomiędzy wektorem globalnym a wektorami lokalnymi służą (automatycznie generowane) obiekty typu **`VecScatter`**. Użytkownik ma możliwość kontrolowania każdego z zadań lokalnych poprzez SLES z nim związany. Aby uzyskać do niego (do nich) dostęp, należy najpierw wydobyć tablicę SLESów dla zadań lokalnych z preconditionera. Robi się to przy pomocy funkcji **`PCASMGetSubSLES(PC pc,int *n_local,int *first_local,SLES **sles)`**. Pierwszy argument tej funkcji to preconditioner z którego wydobywamy SLESy. Drugi argument to liczba SLESów przechowywanych w procesorze, który wywołał tą funkcję, trzeci to numer pierwszego SLESu przechowywanego w procesorze który wywołał tę funkcję a ostatni to wskaźnik do tablicy SLESów, w której znajdą się wyciągane SLESy. Ważne jest to, że funkcja działa lokalnie, to znaczy w przypadku programu równoległego po wywołaniu tej funkcji dostajemy SLESy przechowywane jedynie w procesorze, który wywołał tą funkcję.

9.7 Przykład

Mając już tablicę SLESów możemy kontrolować sposób rozwiązywania zadań. Na przykład możemy zażądać, aby każde zadanie lokalne było rozwiązywane metodą niekompletnego rozkładu Choleskiego ICC.

```
#include "sles.h";
PC pc,pc1;
SLES *slesy;
...
PCASMGetSubSLES(pc,&n,&zero,&slesy);
for(i=zero;i<zero+n;i++)
{
 SLESGetPC(slesy[i],&pc1);
 PCSetType(pc1,PCICC);
}
```



Rysunek 10. Struktura preconditionera Schwarza dla 4 podobszarów.

II. Rozwiązywanie układów równań nieliniowych

W drugiej części podręcznika omówimy wykorzystanie PETSc do rozwiązywania układów nieliniowych równań algebraicznych. W poprzedniej części nauczyliśmy się używać PETSc do rozwiązywania układów równań liniowych, korzystając z funkcji zawartych w komponencie SLES. Typowy schemat postępowania był następujący:

- wygenerować macierz układu i prawą stronę
- utworzyć kontekst SLESu przy pomocy `SLESCreate()`
- przekazać SLESowi parametry zadania, tzn. macierz i wektor prawej strony: `SLESSetOperators()`
- ustawić pożądane parametry SLESu, np. rodzaj metody iteracyjnej, warunki stopu, itp. `SLESSetFromOptions()`
- rozwiązać równanie przy pomocy `SLESSolve()`

Bardzo miłą cechą używania SLESu było m.in. to, że *wszystkie* parametry miały ustawione rozsądne (!) wartości domyślne, dzięki czemu nie musieliśmy martwić się na samym początku o ich dobór. Można więc powiedzieć, że SLES działał trochę na zasadzie czarnej skrzynki, z tą różnicą, że w każdej chwili mieliśmy możliwość zmiany parametrów jej działania. Między innymi, SLES pozwalał nam w łatwy sposób zmienić metodę rozwiązywania układu i warunki stopu.

Podobnie będzie się nam pracowało z kolejnym składnikiem PETSc, służącym tym razem do (równoległego) rozwiązywania nieliniowych układów równań: SNES. Zanim przejdziemy do omówienia sposobów wykorzystania tego komponentu PETSc, zrobimy najpierw krótki wstęp o metodach rozwiązywania równań nieliniowych. Po przypomnieniu metod rozwiązywania takich równań, przedstawimy ogólny schemat użycia SNESu - nieliniowego analogonu SLESu - obiektu służącego do rozwiązywania równań nieliniowych. Następnie zobaczymy na dwóch przykładach, jak w praktyce wygląda stosowanie SNESu. Między innymi, użyjemy SNESu do zaimplementowania prostego programu rozwiązującego układ powstały po dyskretyzacji nieliniowego zagadnienia konwekcji-dyfuzji

$$-\Delta u + (u\cdot\nabla)u = f.$$

W trakcie szczegółowego omawiania przedstawionych przykładów, poznamy techniki efektywnego wykorzystania całej naszej wiedzy o dotychczas poznanych składnikach PETSc (macierzach, SLESie, itp.) oraz omówimy bliżej bardzo pożyteczną ideę kontekstu.

Zakończymy tę część opisem możliwości modyfikacji działania SNESu: wyboru wariantu metody Newtona, kryteriów stopu, itd. oraz podsumowaniem składni wywołań procedur SNESu.

10. Równania nieliniowe

Jak wiadomo z podstawowego kursu analizy numerycznej, zazwyczaj nie istnieje bezpośredni algorytm rozwiązania równania nieliniowego postaci

$$F(x^*) = 0,$$

gdzie $F: R^N \to R^N$. Do rozwiązywania równań powyższej postaci używa się metod iteracyjnych, takich jak metoda iteracji prostej Banacha:

$$x_{n+1} := x_n + F(x_n),$$

czy też metoda Newtona

$$x_{n+1} := x_n - [F'(x_n)]^{-1} F(x_n)$$

gdzie $F'(x_n)$ jest macierzą pochodnej (zwanej także jakobianem) odwzorowania F w punkcie x_n. Czytelnika zainteresowanego, jakie założenia na funkcję F oraz przybliżenie początkowe x_0 gwarantują zbieżność powyższych procesów iteracyjnych, odsyłamy do podręczników analizy numerycznej [6], [18], oraz licznych monografii, np. [15], [11].

Zauważmy, że w przypadku iteracji Newtona nie ma potrzeby (ani sensu) odwracania macierzy $F'(x_n)$ dla obliczenia x_{n+1}. Kolejne przybliżenie rozwiązania x^* uzyskamy najpierw wyznaczając poprawkę na x_n

$$F'(x_n)\cdot\Delta = -F(x_n),$$

rozwiązując względem Δ układ z macierzą jakobianu, a następnie kładąc $x_{n+1} = x_n + \Delta$. Oczywiście, ponieważ macierz pochodnej zmienia się w każdym kroku, rozwiązanie jednego układu liniowego jest dużo tańsze niż najpierw

wyznaczenie macierzy odwrotnej, a dopiero potem pomnożenie jej przez wektor $F(x_n)$.

Reasumując, aby znaleźć nowe przybliżenie, musimy skonstruować macierz pochodnej, a następnie rozwiązać układ liniowy z tą macierzą. Jeśli do rozwiązywania tego układu użyjemy metody iteracyjnej (co może być uzasadnione w przypadku, gdy mamy do czynienia z macierzą rzadką dużego rozmiaru), taka metoda będzie wtedy tzw. niedokładną (ang. *inexact*) metodą Newtona. Możliwe są też inne modyfikacje metody Newtona, o mniejszym koszcie jednego kroku (często właśnie konstrukcja macierzy jakobianu jest zaskakująco kosztownym etapem metody Newtona). W takich przypadkach czasem stosuje się wariant, w którym macierz jakobianu jest uaktualniana dopiero po wykonaniu kilku kroków metody. Innym rozwiązaniem jest aproksymacja macierzy jakobianu przy pomocy różnic skończonych (wielowymiarowy analog metody siecznych). Zainteresowanych przeglądem rozmaitych metod rozwiązywania układów równań nieliniowych odsyłamy do zwięzłej książki [11] oraz do monografii [15].

Solvery nieliniowe dostępne w PETSc dotyczą wyłącznie metod typu Newtona. Nic dziwnego: implementacja metody iteracji (*nomen omen*) prostej jest co prawda nieskomplikowana, ale jej szybkość zbieżności pozostawia dużo do życzenia. Natomiast metoda Newtona jest zbieżna kwadratowo, a ponieważ w każdym kroku algorytmu musimy rozwiązać układ równań liniowych, będziemy mieli doskonałą okazję wykorzystania pełnej mocy i elastyczności SLESu.

Na niekorzyść algorytmu Newtona zdaje się działać fakt, że wymaga on dobrego przybliżenia początkowego. Na szczęście, tę wadę metody Newtona często można obejść w przypadku dyskretyzacji równań różniczkowych cząstkowych, gdy "dobre" przybliżenie początkowe można zgadnąć, lub wyliczyć w mniej lub bardziej wyrafinowany sposób. Ponadto, w PETSc zastosowano heurystyczne techniki "globalizujące" zbieżność metody Newtona. Bazują one na spostrzeżeniu, że w przypadku, gdy startujemy ze złego przybliżenia, poprawka Δ jest co prawda wektorem skierowanym w dobrym kierunku (tzn. w stronę rozwiązania x^*), ale za dużej długości. Taka sytuacja prowadzi do "przestrzelenia" rozwiązania[22] i w efekcie do niezbieżności całego procesu. Dlatego wprowadza się dodatkową operację poszukiwania optymalnej długości wektora poprawki (tzw. *line search methods*). Więcej informacji na ten temat znajduje się w [11], rozdział 8.

11. Schemat użycia SNESu do rozwiązania równania $F(x^*)=0$

Ogólny sposób postępowania przy rozwiązywaniu równań nieliniowych w PETSc będzie bardzo podobny do przypadku liniowego. Jak poprzednio, najpierw trzeba będzie określić obiekty, na których będziemy operować, tzn. funkcję F i jej macierz pochodnej F' pisząc odpowiednie procedury.

Następnie, powołamy do życia maszynkę do rozwiązywania równań nieliniowych (czyli SNES) i przekażemy do niej dane zadania - powiemy jej, co ma być dla niej funkcją F i pochodną F' (po prostu przekażemy jej parametry - analogicznie jak w SLESie, gdzie musieliśmy przekazać SLESowi macierz układu i wektor prawej strony). W końcu, ustawimy dodatkowe parametry SNESu (albo i nie, pamiętając, że wartości domyślne są *zazwyczaj* rozsądnie dobrane), wygenerujemy przybliżenie początkowe i każemy SNESowi rozwiązać nasze równanie.

Tak więc, schemat wykorzystania SNESu do rozwiązania równania nieliniowego jest następujący:

- zdefiniować funkcję F i jej jakobian F'
- wygenerować SNES: `SNESCreate()`
- przekazać SNESowi naszą funkcję i jakobian: `SNESSetFunction()` oraz `SNESSetJacobian()`
- ustawić opcje w SNESie, `SNESSetFromOptions()`
- wyliczyć przybliżenie początkowe
- rozwiązać równanie! `SNESSolve()`

W kolejnych rozdziałach przedstawimy dwa przykładowe problemy nieliniowe (banalny i prawie realny, oba bardzo pouczające!) i przyjrzymy się dokładniej, jak używać SNESu i jego funkcji w tych sytuacjach. Precyzyjna składnia omówionych funkcji zostanie zbiorczo ponownie przedstawiona w rozdziale podsumowującym.

12. Pierwsze zadanie nieliniowe w PETSc

Dla szczegółowego omówienia sposobu używania SNESu wykorzystamy dwa przykładowe problemy. Pierwszy z nich jest bardzo prosty i do jego rozwiązania nie potrzebujemy używać aż PETSc: wystarczyłoby wykorzystać do tego celu - powiedzmy - Matlab'a albo Maple'a. Jednak właśnie ze względu na prostotę zagadnienia, będzie nam łatwiej skoncentrować się na funkcjach PETSc. Bardzo podobny przykład jest także rozważany w oficjalnym *manualu* [2].

[22] Kto nie wierzy, niech spróbuje zastosować algorytm Newtona do funkcji *arctg(x)* z punktem startowym x_0=10.

Problem. Rozwiązać układ dwóch równań nieliniowych

$$\begin{cases} x^2 + y^2 = 1 \\ x + y = 0 \end{cases}$$

Naturalnie, rozwiązaniem są pary *(x,y)* postaci $\left(\pm\frac{\sqrt{2}}{2},\pm\frac{\sqrt{2}}{2}\right)$ Rozwiązanie polega na znalezieniu zera funkcji

$$F\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x^2 + y^2 - 1 \\ x + y \end{pmatrix}$$

Jej pochodna jest oczywiście[23] zadana macierzą następującej postaci:

$$F'\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2x & 2y \\ 1 & 1 \end{pmatrix}$$

Postępując zgodnie ze schematem, w naszym programie najpierw powinniśmy zdefiniować funkcję F oraz macierz pochodnej. Dla większej elegancji, kody źródłowe funkcji i jej pochodnej będziemy trzymać w osobnych plikach[24]. Umówmy się więc, że plik `uklad2x2.c` będzie zawierać kod źródłowy programu głównego, tzn. funkcję `main()`, podczas gdy definicja procedury obliczającej *F(X)* znajdzie się w pliku `fun2x2.c`, a kod dla jakobianu *F´(X)* w pliku `jac2x2.c`. W kolejnych rozdziałach przyjrzymy się dokładnie zawartości tych plików.

12.1 Definicja F

Na początek obejrzyjmy plik `fun2x2.c`, zawierający kod procedury obliczającej wartości funkcji F:

`fun2x2.c`

```
#include "snes.h" /* plik z definicjami funkcji i obiektow SNES;
                     zawiera także dyrektywę #include  "sles.h" */

/*****************************************************
        Procedura obliczajaca wartosci funkcji F
*****************************************************/
int Funkcja(SNES MojSnes,Vec X,Vec F,void *Ctx)
{
  int    ierr;
  Scalar *TablicaX, *TablicaF;

  /* Pobierz wskazniki do tablicy z wspolrzednymi X i F */
  ierr = VecGetArray(X, &TablicaX); CHKERRQ(ierr);
  ierr = VecGetArray(F, &TablicaF); CHKERRQ(ierr);

  /* Oblicz funkcje F */
  TablicaF[0] = TablicaX[0]*TablicaX[0] + TablicaX[1]*TablicaX[1] - 1.0;
  TablicaF[1] = TablicaX[0] + TablicaX[1];

  /* Zamknij dostep do wartosci wektorow */
  ierr = VecRestoreArray(X, &TablicaX); CHKERRQ(ierr);
  ierr = VecRestoreArray(F, &TablicaF); CHKERRQ(ierr);
  return(0);
}
```

Aby SNES mógł z nią współpracować, procedura obliczająca *F(X) musi* być zadeklarowana w ściśle określony sposób: `int Funkcja(SNES MojSnes,Vec X,Vec F,void *Ctx)`. Pierwszy argument, `MojSnes`, to obiekt SNES, do którego później przekażemy tę funkcję. W naszych przykładach nigdy nie będziemy z niego bezpośrednio korzystać, ale warto pamiętać, że jest to możliwe. Wektor `X` jest argumentem funkcji *F*, natomiast do wekto-

[23] Niektórzy twierdzą, że w matematyce wszystko jest albo oczywiste, albo fałszywe...

[24] Jest to typowy - i zalecany - sposób pisania dużych programów w C. Główny program, tzn. funkcja `main()`, ma się składać z wywołań wielu niedługich funkcji. Wygodnie jest pogrupować kody źródłowe tych funkcji w kilku osobno kompilowanych plikach. Więcej na ten temat można dowiedzieć się z [12].

ra F będą wpisywane wartości *F(X)*. Ostatni parametr, Ctx, to kontekst definiowany przez użytkownika. Może on zostać wykorzystany do przekazania z programu dodatkowych parametrów potrzebnych do wyliczenia *F*, podobnie jak to było w przypadku definiowania preconditionera powłokowego (patrz rozdział „Tworzenie własnych preconditionerów" na stronie 34). W naszym przykładzie nie korzystamy z Ctx, niemniej musimy go uwzględnić w liście parametrów procedury Funkcja(), bo takiego formatu procedury będzie oczekiwał SNES.

Sama definicja Funkcji() nie jest skomplikowana. Mając dany wektor X (o dwóch współrzędnych), najpierw przygotujemy się do pobrania z niego kolejnych współrzędnych:

```
ierr = VecGetArray(X, &TablicaX); CHKERRQ(ierr);
```

W tym momencie, wskaźnik TablicaX wskazuje na tablicę elementów wektora X. Od tej pory możemy więc odwoływać się do poszczególnych współrzędnych (powiedzmy, do i-tej) tradycyjnym TablicaX[i]. Podobnie czynimy z wektorem F, aby z kolei łatwo *wpisywać* do niego wartości kolejnych współrzędnych. Dalej, obliczamy *F(X):*

```
TablicaF[0] = TablicaX[0]*TablicaX[0] + TablicaX[1]*TablicaX[1] - 1.0;
TablicaF[1] = TablicaX[0] + TablicaX[1];
```

implementując wzór

$$F\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^{\,2} + x_2^{\,2} - 1 \\ x_1 + x_2 \end{pmatrix}$$

Na zakończenie musimy pamiętać, aby odblokować wektory X i F komendą VecRestoreArray(). Ta funkcja uaktualnia wektor po dokonanym dostępie do tablicy jego wartości [25].

12.2 Definicja pochodnej F

Po zdefiniowaniu funkcji, czas zapoznać się z kodem procedury obliczającej macierz jakobianu. Oto zawartość pliku jac2x2.c, w którym podajemy sposób obliczania macierzy jakobianu:

jac2x2.c

```
/*****************************************************
                  Definicja pochodnej F
*****************************************************/

#include "snes.h"
int Jakobian(SNES MojSnes, Vec X, Mat *Jac,
               Mat *Precond, MatStructure *flag, void *Ctx)
{
  Scalar *x;            /* Wskaznik do wartosci wektora X */
  Scalar A[4];          /* Tu wpiszemy elementy macierzy Jacobiego */
  int ierr;
  int idx[2] = {0,1};

  ierr = VecGetArray(X, &x); CHKERRQ(ierr);

  /* Oblicz elementy jakobianu i wstaw do macierzy Jakobian */
  A[0] = 2.0*x[0]; A[1] = 2.0*x[1]; /* pierwszy wiersz */
  A[2] = 1.0; A[3] = 1.0;           /* drugi wiersz */
  ierr = MatSetValues(*Jac,2,idx,2,idx,A,INSERT_VALUES); CHKERRQ(ierr);
  *flag = SAME_NONZERO_PATTERN;

  /* Odblokuj i uaktualnij X */
  ierr = VecRestoreArray(X,&x); CHKERRQ(ierr);

  /* Zloz macierz  */
  ierr = MatAssemblyBegin(*Jac, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
```

[25] Ta pozornie niepotrzebna para funkcji dostępu VecGetArray()/VecRestoreArray() jest jednak niezbędna w implementacji równoległej, podobnie jak duet VecAssemblyBegin()/VecAssemblyEnd().

```
  ierr = MatAssemblyEnd(*Jac, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
  return(0);
}
```

Podobnie jak poprzednio, procedura wyznaczająca jakobian, *F′(X)*, *musi* być specjalnego typu:

`int Jakobian(SNES MojSnes, Vec X, Mat *Jac, Mat *Precond, MatStructure *flag, void *Ctx);`

gdzie `MojSnes` jest obiektem SNES do którego zostanie później przekazana ta procedura, a `X` jest punktem w którym liczymy jakobian. Do macierzy `Jac` (zwróćmy uwagę, że argumentem jest wskaźnik, a nie sama macierz!) jest wpisywana wyznaczona macierz pochodnej i to jest podstawowe zadanie tej procedury: dostarczenie sposobu obliczania macierzy *F′(X)*.

Macierz `Precond` oznacza macierz preconditionera (zazwyczaj będzie to ta sama macierz `Jac`). Parametr `flag` jest informacją dla PETSc, czy w następnej iteracji Newtona będzie można wykorzystać struktury preconditionera, zbudowane na użytek poprzedniej - podobnie jak w przypadku funkcji `SLESSetOperators()`. Wartości, jakie można przypisywać parametrowi `flag` były opisane w poprzedniej części (patrz Rozdział 4 na stronie 22). Ostatni parametr, `Ctx`, to kontekst definiowany przez użytkownika. Może on zostać wykorzystany do przekazania dodatkowych parametrów z programu potrzebnych do wyliczenia *F′*, podobnie jak w przypadku implementacji funkcji F w procedurze `Funkcja()`. W naszym przykładzie nie korzystamy z `Ctx`.

12.3 Program główny: rozwiązanie równania przy pomocy SNESu

Poniżej omówimy sposób wykorzystania procedur `Funkcja()` i `Jakobian()` do rozwiązania naszego zagadnienia przy użyciu SNESu. Oto plik `uklad2x2.c`, zawierający odpowiedni kod:

`uklad2x2.c`

```
#include "snes.h"
#include <stdio.h>
static char help[] = "Rozwiazuje nieliniowy uklad 2x2\n\
Opcje: -x0 [Scalar] -y0 [Scalar] wartosci wspolrzednych przybl. poczatk.\n\n";

/* Deklaracje procedur obliczajacych funkcje i jakobian  */
int Jakobian(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
int Funkcja(SNES,Vec,Vec,void*);

/*****************************************************
                  Program glowny
*****************************************************/
int main( int argc, char **argv )
{
  SNES     MojSnes;        /* obiekt SNES, ktorym rozwiazemy zadanie */
  Vec      X, R;           /* wektor rozwiazania i residualny */
  Mat      MatJ;           /* macierz jakobianu */
  int      ierr, flg, its;
  int      idx[2] = {0,1};
  Scalar   X0[2];          /* wspolrz. przyblizenia poczatkowego */

  PetscInitialize( &argc, &argv,PETSC_NULL,help );

/* Tworzenie potrzebnych obiektow: SNESu, wektorow, macierzy jakobianu */
  ierr = SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS,&MojSnes);
  CHKERRA(ierr);
  ierr = VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,2,&X); CHKERRA(ierr);
  ierr = VecDuplicate(X,&R); CHKERRA(ierr);
  ierr = MatCreate(MPI_COMM_WORLD,2,2,&MatJ); CHKERRA(ierr);

/* Przekaz SNESowi funkcje i wektor residualny   */
  ierr = SNESSetFunction(MojSnes, R, Funkcja, PETSC_NULL); CHKERRA(ierr);

/* Przekaz SNESowi macierz jakobianu i funkcje obliczajaca jakobian  */
  ierr = SNESSetJacobian(MojSnes,MatJ,MatJ,Jakobian,PETSC_NULL);CHKERRA(ierr);
```

```
/*  Uwzglednij opcje uruchomienia SNESu typu -snes_view -snes_monitor */
  ierr = SNESSetFromOptions(MojSnes); CHKERRA(ierr);

/* Przyblizenie poczatkowe  */
  ierr = OptionsGetScalar( PETSC_NULL, "-x0", &X0[0], &flg ); CHKERRA(ierr);
  if( !flg ) X0[0] = 1.0;
  ierr = OptionsGetScalar( PETSC_NULL, "-y0", &X0[1], &flg ); CHKERRA(ierr);
  if( !flg ) X0[1] = 2.0;
  ierr = VecSetValues(X, 2, idx, X0, INSERT_VALUES ); CHKERRA(ierr);
  ierr = VecAssemblyBegin(X); CHKERRA(ierr);
  ierr = VecAssemblyEnd(X); CHKERRA(ierr);

/* Rozwiaz!    */
  ierr = SNESSolve(MojSnes,X,&its); CHKERRA(ierr);

  PetscPrintf(MPI_COMM_WORLD, "Liczba iteracji = %d\n", its);
  PetscPrintf(MPI_COMM_WORLD,"Wektor rozwiazania:\n");
  ierr = VecView(X, VIEWER_STDOUT_WORLD); CHKERRA(ierr);

/* Zwolnij pamiec */
  ierr = VecDestroy(X); CHKERRA(ierr);
  ierr = VecDestroy(R); CHKERRA(ierr);
  ierr = MatDestroy(MatJ); CHKERRA(ierr);
  ierr = SNESDestroy(MojSnes); CHKERRA(ierr);

/* Koniec bajki  */
  PetscFinalize();
  return 0;
}
```

Przyjrzyjmy się funkcji `main()`. Przeglądając pobieżnie komentarze, możemy stwierdzić z satysfakcją, że zapowiedź z Rozdziału 11 spełniła się: w kolejnych krokach wykonywania programu

1. uruchamiamy PETSc;
2. tworzone są potrzebne obiekty: SNES, wektory, macierz jakobianu;
3. SNES otrzymuje potrzebne mu obiekty: funkcję i pochodną;
4. wybieramy przybliżenie początkowe;
5. rozwiązujemy równanie przy pomocy `SNESSolve()`;
6. oglądamy wyniki i sprzątamy po sobie.

Tak właśnie myślelibyśmy układając sobie w myśli szkicowy algorytm metody Newtona: przygotować dane, zdefiniować funkcję i jakobian, a potem zaprogramować iterację Newtona. Oczywiście, dla programisty wszystkie etapy, poza ostatnim, są jedynie grą wstępną przed właściwym aktem rozwiązania układu. Dzięki mocy narzędzi PETSc, etap ostatni - iteracja Newtona - już *nie wymaga* programowania! Wystarczy wywołać `SNESSolve()`... Dzięki temu, co wielokrotnie podkreślaliśmy, programista może skupić się na algorytmie, eksperymentowaniu i analizie wyników. Żmudna dłubanina przy implementacji podstawowych narzędzi została mu oszczędzona, dzięki wielkiej pracy twórców PETSc.

Oto przykładowy wynik uruchomienia powyższego programu:

```
11:/home/staff/przykry/Petsc> test
Liczba iteracji = 7
Wektor rozwiazania:
-0.707107
0.707107
```

Jak widać, algorytm rzeczywiście jest zbieżny do rozwiązania ($\frac{\sqrt{2}}{2} = 0.7071...$). Zobaczmy teraz, jakie są szczegóły poszczególnych kroków algorytmu. Na początku pliku `uklad2x2.c` pojawiają się dyrektywy:

```
#include "snes.h"
#include <stdio.h>
```

Dzięki pierwszej możemy korzystać z wszystkich dobrodziejstw SNESu, SLESu i reszty aparatu PETSc, jaki do tej pory opanowaliśmy. Dodatkowo `#include <stdio.h>` umożliwi nam korzystanie ze standardowych procedur wejścia/wyjścia. W zasadzie wstawiamy tę dyrektywę z przyzwyczajenia, bo (prawdę mówiąc) w tym programie wcale z nich nie korzystamy... Dalej, po dobrze już nam znanej deklaracji

```
static char help[] = "Rozwiazuje nieliniowy uklad 2x2\n\
Opcje: -x0 [Scalar] -y0 [Scalar] wartosci wspolrzednych przybl. poczatk.\n\n";
```

następują deklaracje procedur obliczających funkcje i jakobian:

```
int Jakobian(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
int Funkcja(SNES,Vec,Vec,void*);
```

To jest ważna część nagłówka, ponieważ te dwie procedury są zdefiniowane w innych plikach. Taka deklaracja umożliwi kompilatorowi właściwe zinterpretowanie wywołań tych procedur w funkcji `main()`, którą teraz zaczniemy analizować krok po kroku.

Na początku funkcji `main()` deklarujemy kilka zmiennych roboczych:

```
SNES     MojSnes;         /* obiekt SNES, ktorym rozwiazemy zadanie */
Vec      X, R;            /* wektor rozwiazania i residualny */
Mat      MatJ;            /* macierz jakobianu */
int      ierr, flg, its;
int      idx[2] = {0,1};
Scalar   X0[2];           /* wspolrz. przyblizenia poczatkowego */
```

Zmienne `MojSnes, X, R, MatJ` będą nam naturalnie potrzebne do rozwiązania naszego problemu. Zmienna `ierr` będzie przyjmować kod wyniku procedur PETSc, sprawdzany następnie (czy nie jest to kod błędu) przez makro `CHKERRA()`[26]. Ze zmiennej `flg` będzie korzystać funkcja `OptionsGetScalar()`; do zmiennej `its` SNES wpisze liczbę wykonanych iteracji. Sens ostatnich dwóch deklaracji wyjaśni się za chwilę. Na razie, po deklaracjach, inicjalizujemy PETSc:

```
PetscInitialize( &argc, &argv,PETSC_NULL,help );
```

i zaraz tworzymy potrzebne nam obiekty PETSc: przede wszystkim kontekst SNESu dla rozwiązywania układu równań nieliniowych

```
SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS,&MojSnes);
```

Jak możemy się przekonać, funkcja tworzenia SNESu ma trzy parametry, gdzie pierwszy parametr to komunikator PETSc (u nas, jak zwykle, jest to `MPI_COMM_WORLD`), jako drugi zawsze będziemy podawali predefiniowaną stałą `SNES_NONLINEAR_EQUATIONS`, natomiast ostatnim argumentem jest wskaźnik do zmiennej typu SNES. Następnie generujemy dwa wektory: rozwiązania i residualny, każdy o dwóch współrzędnych (oczywiście, wystarczy wygenerować pierwszy, a drugi tylko duplikować)

```
VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,2,&X);
VecDuplicate(X,&R);
```

W końcu generujemy macierz jakobianu o wymiarach 2x2:

```
MatCreate(MPI_COMM_WORLD,2,2,&MatJ);
```

Teraz, mając utworzone potrzebne obiekty oraz funkcje do obliczania F i F', możemy to wszystko przekazać SNESowi. Jak wcześniej wspominaliśmy, służą temu dwie funkcje: `SNESSetFunction()` oraz `SNESSetJacobian()`. Format pierwszej z nich jest następujący: **`int SNESSetFunction(SNES MojSnes,Vec R,int (*Funkcja)(SNES,Vec,Vec,void*),void *Ctx)`**. Pierwszym argumentem jest `MojSnes` - obiekt SNESu, do którego chcemy przekazać funkcję obliczającą wartości *F(X)*. Drugi argument, `R`, to wektor w którym będzie przechowywane residuum (czyli, *de facto*, wartości funkcji *F*). Trzecim argumentem jest *nazwa* procedury - u nas: `Funkcja()` - obliczającej wartości *F(X)*. Jak widać z powyższego formatu, SNES (i kompilator...) oczekuje, że `Funkcja()` jest deklarowana jako **`int Funkcja(SNES MojSnes,Vec X,Vec F,void *Ctx)`** i właśnie dlatego w taki, a nie inny sposób definiowaliśmy naszą `Funkcję()` w pliku `fun2x2.c` powyżej. Ostatni argument dla `SNESSetFunction()` to wskaźnik do kontekstu, wykorzystywanego przez SNES i `Funkcję()`.

Druga funkcja przekazuje do SNESu macierz jakobianu i metodę uaktualniania go:
`int SNESSetJacobian( SNES MojSnes, Mat Jac, Mat Prec, int (*Jakobian)(SNES,`

[26]To bardzo przydatna funkcja, pozwalająca szybko namierzyć błąd w kodzie źródłowym.

Vec, Mat*, Mat*, MatStructure*,void*), void *Ctx)

Podobnie jak poprzednio, pierwszym argumentem jest `MojSnes` - obiekt SNESu, do którego chcemy przekazać funkcję obliczającą *F'(X)*, tzn. macierz jakobianu. Drugim argumentem jest więc macierz jakobianu `Jac`, do której będą wprowadzane wartości elementów, obliczone w procedurze `Jakobian()`. Trzecim argumentem jest macierz preconditionera `Prec` dla macierzy `Jac`, przydatna przy rozwiązywaniu układów z tą macierzą. Zazwyczaj wystarcza podać jako `Prec` po prostu tą samą macierz `Jac`. Kolejny, czwarty argument to *nazwa* procedury - u nas: `Jakobian()` - obliczającej elementy macierzy jakobianu *F'(X)*. Jak widać z powyższego formatu, SNES (i kompilator...) oczekuje, że funkcja `Jakobian()` jest deklarowana jako **`int Jakobian(SNES MojSnes,Mat *Jac,Mat *Prec,int *flg,void *Ctx)`** i właśnie dlatego w taki, a nie inny sposób definiowaliśmy naszą procedurę `Jakobian()` w pliku `jac2x2.c` powyżej. Ostatni argument dla `SNESSetJacobian()` to wskaźnik do kontekstu, wykorzystywanego przez procedurę `Jakobian()` i zawierającego dane używane przy wyznaczaniu macierzy *F'(X)*.

Ponieważ nasze procedury: `Funkcja()` oraz `Jakobian()` nie korzystają z kontekstu użytkownika, jako ostatni argument wywołania `SNESSetFunction()` i `SNESSetJacobian()` podajemy `PETSC_NULL`:

```
 /* Przekaz SNESowi funkcje i wektor residualny */
SNESSetFunction(MojSnes, R, Funkcja, PETSC_NULL);

 /* Przekaz SNESowi macierz jakobianu i funkcje obliczajaca jakobian */
SNESSetJacobian(MojSnes,MatJ,MatJ,Jakobian,PETSC_NULL);

 /* Uwzglednij opcje uruchomienia SNESu typu -snes_view -snes_monitor  */
SNESSetFromOptions(MojSnes);
```

W ostatniej instrukcji, podobnie jak w SLESie, ustawiamy opcje SNESu zgodnie z argumentami podanymi w linii komend podczas wywołania programu, a jeśli nie były one podane - to zgodnie z wartościami domyślnymi.

Wszystkie dane o równaniu są już przekazane do SNESu, opcje ustawione, moglibyśmy więc już rozwiązać równanie... jeśli tylko określilibyśmy przybliżenie początkowe. Przybliżenie początkowe (analogicznie jak w SLESie) wpiszemy wprost do wektora rozwiązania. Właśnie do tego przydadzą się nam tablice `X0` i `idx`, zadeklarowane na początku programu.

```
/* Przyblizenie poczatkowe */
OptionsGetScalar( PETSC_NULL, "-x0", &X0[0], &flg );
if( !flg ) X0[0] = 1.0;
OptionsGetScalar( PETSC_NULL, "-y0", &X0[1], &flg );
if( !flg ) X0[1] = 2.0;
VecSetValues(X, 2, idx, X0, INSERT_VALUES );
ierr = VecAssemblyBegin(X); CHKERRA(ierr);
ierr = VecAssemblyEnd(X); CHKERRA(ierr);
```

Ponieważ dopuszczamy możliwość podawania tego przybliżenia przez użytkownika z linii komend, stąd użycie funkcji `OptionsGetScalar()`. Jeśli użytkownik nie podał którejś z opcji, wtedy przyjmowana jest wartość domyślna. Warto przyjrzeć się jeszcze raz, jak to się dzieje. Powiedzmy, że użytkownik uruchomił nasz program w następujący sposób:

```
18:/home/staff/przykry/Petsc> test -y0 5.0
```

Wtedy pierwsze wywołanie funkcji `OptionsGetScalar()` wpisze do zmiennej `flg` wartość `FALSE`, bo opcji `-x0` nie podano. Wobec tego wykona się instrukcja `if( !flg ) X0[0] = 1.0;` i na `X0[0]` przyjęta będzie wartość domyślna. Ponieważ druga opcja została podana, `OptionsGetScalar()` przypisze na `X0[1]` wartość `5.0`, a program nie wejdzie do drugiego `if()`. Na zakończenie, funkcją `VecSetValues()` wpisujemy[27] do wektora rozwiązania `X` liczby z tablicy `X0`.

Po określeniu przybliżenia początkowego nie pozostaje nam już nic innego, jak nakazać SNESowi rozwiązanie naszego zadania:

```
SNESSolve(MojSnes,X,&its);
```

które możemy dodatkowo sobie obejrzeć przy pomocy `VecView()`:

[27]Uważny czytelnik z pewnością zauważył, że można było postąpić analogicznie jak w definicji funkcji F i użyć procedur `VecGetArray()/VecRestoreArray()`. Zgoda, ale wtedy nie zobaczylibyśmy, jak robić to inaczej... Faktem jest, że często (zwłaszcza w przypadku programu sekwencyjnego) użycie tandemu `VecGetArray()/VecRestoreArray()` jest sensowniejsze.

```
PetscPrintf(MPI_COMM_WORLD, "Liczba iteracji = %d\n", its);
PetscPrintf(MPI_COMM_WORLD,"Wektor rozwiazania:\n");
VecView(X, VIEWER_STDOUT_WORLD);
```

Format funkcji `SNESSolve()` jest analogiczny jak funkcji `SLESSolve()`: **`int SNESSolve(SNES MojSnes,Vec X,int *its)`** gdzie `MojSnes` zawiera już wprowadzone parametry naszego problemu, to znaczy `Funkcję()` i `Jakobian()`. Wektor `X` na wejściu do `SLESSolve()` musi zawierać przybliżenie początkowe rozwiązania; po wyjściu zawiera ostatnio uzyskane przybliżenie, uzyskane po `its` iteracjach (trzeci argument, wskaźnik `its` do zmiennej typu integer, pozwoli SNESowi wpisać do niej liczbę wykonanych iteracji). Jak widać, najpoważniejsza część pracy - algorytm i rozwiązanie równania nieliniowego - jest całkowicie ukryta w niewinnym wywołaniu `SNESSolve()`. Na zakończenie można zwolnić używaną pamięć (warto wyrobić sobie taki nawyk) i to wszystko!

```
VecDestroy(X);
VecDestroy(R);
MatDestroy(MatJ);
SNESDestroy(MojSnes);
PetscFinalize();
```

Na zakończenie podajemy postać pliku `Makefile`, którego można użyć do skompilowania omówionego powyżej programu:

`Makefile`

```
include $(PETSC_DIR)/bmake/$(PETSC_ARCH)/base
CC = gcc -DPETSC_LOG
CLINKER = gcc
CFLAGS = $(PCONF) $(PETSC_INCLUDE) $(BS_INCLUDE)
LIBBASE = libpetscsles libpetscsnes
OPTS = -O
BOPT = O

OBJECTS_EX1 = uklad2x2.o fun2x2.o jac2x2.o

EXECUTABLE = test

ex1: $(OBJECTS_EX1)
        -$(CLINKER) -o $(EXECUTABLE) $(OBJECTS_EX1) \
        -lm $(PETSC_LIB)

#
# ex1 source files
#

uklad2x2.o: uklad2x2.c
        $(CC) $(CFLAGS) $(OPTS) -c uklad2x2.c

fun2x2.o: fun2x2.c
        $(CC) $(CFLAGS) $(OPTS) -c fun2x2.c

jac2x2.o: jac2x2.c
        $(CC) $(CFLAGS) $(OPTS) -c jac2x2.c
```

12.4 Ćwiczenia

A) Zaimplementuj metodę iteracji prostej dla rozwiązywania równań nieliniowych, działającą analogicznie jak SNES. Testuj ją na powyższym przykładzie.

B) Naucz się obsługiwać kilka opcji uruchomienia SNESu (patrz rozdział „Co jeszcze warto wiedzieć o SNESie?" na stronie 65). Zbadaj, który wariant metody Newtona jest najbardziej efektywny na omawianym przykładzie.

C) Zmień funkcję na bardziej skomplikowaną[28] i zobacz, jak przybliżenie początkowe wpływa na przebieg procesu

[28] Nie zapomnij zmienić też jakobianu!

zbieżności.

13. Nieliniowe zagadnienie różniczkowe w PETSc

W tym rozdziale zajmiemy się (wciąż bardzo elementarnym) przykładem nieliniowego równania pochodzącego z dyskretyzacji równania konwekcji-dyfuzji

$$-u''(x)+u(x)\cdot u'(x)=f(x)\ dla\ x\in(0,1),$$
$$u(0)=u(1)=0.$$

Dla prostoty, to równanie będziemy aproksymować metodą różnic skończonych, dostając problem dyskretny postaci:

Problem. Dla danego wektora $f_h=(f_1,...,f_n)^T$ znaleźć $u_h=(u_1,...,u_N)^T$ spełniający

$$-\frac{u_{n-1}-2u_n+u_{n+1}}{h^2}+u_n\cdot\frac{u_{n+1}-u_{n-1}}{2h}=f_n \text{ dla } n=1,...,N,$$
$$u_0=u_{N+1}=0.$$

Jak widzimy, mamy tym razem do czynienia z układem N równań nieliniowych, gdzie N oznacza liczbę wewnętrznych węzłów siatki (warunki brzegowe Dirichleta wyznaczają od razu u_0 i u_{N+1}, więc możemy ograniczyć się jedynie do obliczenia u w pozostałych węzłach). Jeśli więc siatka jest gęsta, to rozmiar zadania nieliniowego jest bardzo duży, a macierz pochodnej rozrzedzona. Właśnie to powoduje, że stosowanie PETSc do problemów pochodzących z dyskretyzacji nieliniowych równań różniczkowych cząstkowych zaczyna być sensowne. Narzędzia dostępne w PETSc (a więc także w SNESie) były tworzone właśnie z myślą o efektywnych metodach rozwiązywania problemów powstałych z dyskretyzacji równań różniczkowych cząstkowych.

13.1 Wstępna analiza problemu: wybór strategii

Zobaczymy (nareszcie!) jak, używając PETSc, napisać zwarty, prosty, a do tego elegancki kod, rozwiązujący nieliniowe równanie różniczkowe. Na naszym przykładzie nauczymy się pewnych technik, skutecznych również w bardziej skomplikowanych przypadkach.

Zacznijmy od przeanalizowania problemu. To jest bardzo ważne, albowiem od tego, w jaki sposób podejdziemy do zagadnienia, będzie zależało wiele naszych następnych posunięć. Mamy do czynienia z nieliniowym zagadnieniem dyskretnym, które zamierzamy rozwiązywać metodą Newtona. Musimy więc najpierw przeformułować to zagadnienie jako zero pewnego odwzorowania $F:R^N\to R^N$. Oznaczając przez u_h wektor rozwiązania:

$$u_h=\begin{pmatrix}u_1\\ \vdots\\ u_N\end{pmatrix},$$

możemy zapisać zagadnienie dyskretne w postaci wektorowej[29]:

$$A_hu_h+u_h\cdot B_hu_h=f_h,$$

gdzie A_h oznacza macierz (minus) laplasjanu, a B_h macierz dyskretyzacji pierwszej pochodnej różnicą centralną, tzn.:

$$A_h=\frac{1}{h^2}\begin{bmatrix}2&-1&&\\-1&2&\ddots&\\&\ddots&\ddots&-1\\&&-1&2\end{bmatrix},\quad B_h=\frac{1}{2h}\begin{bmatrix}0&1&&\\-1&0&\ddots&\\&\ddots&&1\\&&-1&0\end{bmatrix}$$

Zwróćmy uwagę, że tym razem operację mnożenia wektorów $u_h\cdot B_h\,u_h$ rozumiemy nie w sensie iloczynu skalarnego, tylko jako operację mnożenia odpowiadających współrzędnych.

Przy takim zapisie łatwo już określić funkcję, której zerem będzie rozwiązanie naszego zagadnienia:

[29]Należy zdawać sobie sprawę z tego, że dobry zapis wektorowy algorytmów numerycznych ma istotne znaczenie, nie tylko dla przejrzystości kodu, ale także - jego efektywności na nowoczesnych architekturach, zob. [17], [5].

$$F(u) := A_h u + u \cdot B_h u - f_h.$$

Jej pochodną będzie naturalnie taki operator liniowy $F'(u)$, który w działaniu na wektor w da wektor $F'(u)w$, określony przez $F'(u)w = A_h w + u \cdot B_h w + w \cdot B_h u$. Określiliśmy więc F oraz wyprowadziliśmy wzór na F'. Jak pamiętamy, SNES będzie potrzebował procedur obliczających F i F'. Musimy teraz zastanowić się, jak zaimplementować te funkcje w PETSc. Tutaj pojawia się następna zaleta zapisu wektorowego: daje się on bardzo łatwo „przetłumaczyć" na kod źródłowy PETSc!

13.2 Definicja funkcji F

Popatrzmy, z jaką łatwością i elegancją[30] możemy zaimplementować procedurę obliczającą wartości funkcji F. Otóż przypuśćmy, że już zdefiniowaliśmy gdzieś w programie macierze A_h i B_h dyskretyzacji laplasjanu i pierwszej pochodnej (skądinąd doskonale wiemy, jak to zrobić!). Wtedy reszta to fraszka: wystarczy wygenerować wektor prawej strony i w procedurze zapisać, że wartość $F(X)$ oblicza się w/g następującego algorytmu:

$F(X)$	=	$A_h X$	←	macierz laplasjanu pomnożona przez X funkcją `MatMult()`
		$+ X \cdot B_h X$	←	wynik mnożenia macierzy pierwszej pochodnej przez X, razy (po współrzędnych) X - funkcje `MatMult()` oraz `VecPointwiseMult()`,
		$- f_h$	←	prawa strona

Co prawda widać, że w tym celu będzie nam trzeba kilku wektorów pomocniczych, tak, abyśmy mieli gdzie przechowywać wyniki pośrednie, ale prostota definicji F jest uderzająca. Części liniowe F określają nam odpowiednie macierze, nieliniowości implementujemy wykonując odpowiednie operacje na wektorach.

Czy z pochodną $F'(u)$ pójdzie nam równie łatwo?...

13.3 Pochodna F

Jak wiadomo, pochodną odwzorowania $F: R^N \to R^N$ możemy reprezentować jako macierz $N \times N$. Zwróćmy jednak uwagę na pewien detal: było nam wygodniej zapisać pochodną F nie jako macierz pochodnych cząstkowych, lecz jako operator, którego wartości na zadanym wektorze w są zdefiniowane pewnym wzorem. W taki właśnie sposób definiuje się w PETSc macierze powłokowe (*shell matrices*), o których była mowa na stronie 18. Idea macierzy powłokowych wyjątkowo dobrze sprawdzi się w naszym zagadnieniu:

- Pochodną było nam po prostu *łatwiej* określić operatorowo, niż macierzowo. Niby mała rzecz, bo i tak umiemy różniczkować, ale każdy, kto doświadczył drobnej pomyłki w trakcie obliczeń na papierze, doceni fakt, że nie musimy tego robić, o ile używamy macierzy powłokowej.
- Przypomnijmy sobie, że w każdym kroku metody Newtona, macierz jakobianu $F'(X)$ zmienia się, ponieważ zmienia się punkt, w którym jest wyznaczana. Gdybyśmy używali postaci macierzowej, za każdym razem musielibyśmy uaktualniać elementy tej macierzy. Mając macierz powłokową, nie musimy robić nic, ponieważ zmiany są uwzględniane na bieżąco.
- Do rozwiązywania układów z macierzą pochodnej $F'(X)$ w PETSc będziemy zapewne stosować jakąś metodę iteracyjną, a te wymagają jedynie operacji mnożenia przez macierz. Ponieważ i tak na początku procesu Newtona nie mamy dobrego przybliżenia, będziemy mogli pozwolić sobie na mniej dokładne rozwiązywanie układów z macierzą jakobianu w pierwszych kilku krokach, co dramatycznie obniży koszt całego procesu. Mówimy o tym więcej w rozdziale „Inne opcje" na stronie 66.
- Wreszcie, co też jest ważne, wykorzystamy już *istniejące* obiekty. W definicji funkcji F używaliśmy macierzy laplasjanu i pochodnej, i teraz te same macierze posłużą nam do zdefiniowania F'. Nie będziemy w bólach definiować element po elemencie macierzy jakobianu!

Wobec powyższego, wystarczy odpowiednio zdefiniować macierz powłokową.

13.4 Koncept kontekstu

Pomyślmy chwilkę o tym, jak w praktyce implementowalibyśmy funkcję F. Jak wiadomo, SNES wymaga od nas

[30] Wbrew twierdzeniom rozmaitych domorosłych hackerów i innym pozorom twierdzimy, że czytelny i zrozumiały, choć być może nieoptymalny kod jest bardziej wartościowy od pełnego sztuczek superniezrozumiałego superkodu. Z naszego punktu widzenia, przy pisaniu dużej aplikacji gigantyczną porcję czasu zajmuje odpluskwianie programu. Szczególnie niewdzięcznie wygląda to zadanie w przypadku aplikacji numerycznych, takich, jakimi się tu zajmujemy (cóż, przykre doświadczenia życiowe...) Elegancki, zwarty, modułowy kod pozwala na lepsze śledzenie działania programu i szybszą lokalizację błędów. **Efektywność programu to nie tylko szybkość jego działania, ale także szybkość doprowadzenia go do działania.** Z drugiej strony, nie należy wpadać w skrajności i np. poświęcać całą szybkość programu dla piękna kodu źródłowego - zob. też „Koncept kontekstu" na stronie 57.

funkcji formatu **`int Funkcja(SNES MojSnes,Vec X,Vec F,void * Ctx)`**. My zaś dla wyliczenia *F* potrzebujemy odwołać się do mnożenia przez macierze A_h i B_h. Jak to zrealizować w praktyce?

Ktoś bardzo początkujący mógłby pomyśleć o takiej realizacji[31]:

```
int Funkcja( SNES MojSnes, Vec X, Vec F, void * Ctx )
{
  Mat A, B;

  /* generuj A i B, a nastepnie przypisz im wartosci */
  MatCreate(A); MatCreate(B);
  MatSetValues(A); ...itd....

  /* uzyj A i B do obliczenia F */
  MatMult(A); VecAXPY(); MatMult(B); itd...

  /* zwolnij macierze A i B */
  MatDestroy(A); MatDestroy(B);
}
```

Zaproponowana funkcja w zasadzie nie wygląda źle: macierze `A` i `B` są używane lokalnie przez *F*, więc są też zadeklarowane lokalnie. Po wykorzystaniu są niszczone, bo przy następnym wywołaniu funkcja zainicjuje inne zmienne wewnętrzne. Niestety, ta implementacja jest nie do przyjęcia, bo jest kompletnie nieefektywna. Zdajmy sobie sprawę z tego, że przy *każdym* obliczeniu wartości *F(X)*, będą generowane dwie (całkiem duże) macierze, wpisywane do nich wartości - i za każdym razem dokładnie te same! Jest to szalona rozrzutność, dająca w konsekwencji owszem, działający kod, ale potwornie wolny...

Podstawową wadą zaproponowanej procedury jest wielokrotne robienie tego samego: generowanie wciąż tych samych macierzy. Najprostszym sposobem usunięcia tego problemu byłoby jednokrotne utworzenie i wygenerowanie elementów macierzy `A` i `B` (raz, a dobrze!) gdzieś *na zewnątrz* procedury `Funkcja()`. Świetnie, ale jak skorzystać z tych macierzy *wewnątrz* procedury? Znów, najszybciej przychodzi nam do głowy rozwiązanie najprostsze, ale jednocześnie najbardziej bałaganiarskie: po prostu, trzeba zadeklarować te macierze jako zmienne globalne!

```
Mat A, B; /* nasze macierze jako zmienne globalne:
             deklarowane na zewnatrz wszystkich funkcji */

/*****************************************************
                  Program glowny
*****************************************************/

int main(int argc, char **argv)
{

  /* generuj A i B, a nastepnie przypisz im wartosci */
  MatCreate(A); MatCreate(B);
  MatSetValues(A); ...itd...

  /* ......... */
  SNESSetFunction( Funkcja );

  /* ......... */
  /* zwolnij macierze A i B */
  MatDestroy(A); MatDestroy(B);
  return(0);
}

/*****************************************************
              Procedura obliczajaca F(X)
*****************************************************/
int Funkcja( SNES MojSnes, Vec X, Vec F, void * Ctx )
{
```

[31]Tak jak podczas rzeczywistego procesu zastanawiania się nad implementacją, nie dbamy o szczegóły syntaktyczne wywołań poszczególnych funkcji, skupiając się na ich istotnych argumentach.

```
  /* uzyj A i B do obliczenia F */
  MatMult(A); VecAXPY(); MatMult(B); itd...
  return(0);
}
```

To już jest całkiem dobre, a zwłaszcza - znacznie bardziej efektywne, niż poprzednia propozycja. Niemniej, kiedy wiemy, jak to zrobić efektywnie, powinniśmy spróbować zrobić to elegancko i temu właśnie będzie służył tytułowy **koncept kontekstu użytkownika** (pomysł na grę słów zaczerpnęliśmy od B.Smitha). Cóż, eleganckim rozwiązaniem byłoby przekazanie do `Funkcji()` macierzy A_h i B_h (uprzednio wygenerowanych na zewnątrz procedury mnożenia!) w postaci *parametrów*. Z tym jednak pozornie jest kłopot: format `Funkcji()` jest sztywny i nie możemy dowolnie modyfikować listy jej argumentów. Na szczęście, ostatnim dopuszczalnym argumentem funkcji jest wskaźnik do kontekstu użytkownika, zawierającego pomocnicze dla procedury obiekty i właśnie w ten sposób przekażemy nasze macierze!

Jak taki kontekst powinien wyglądać? Zasadniczo, będzie to struktura w C, zawierająca parametry, które zechcemy przekazać do `Funkcji()`. Tymi parametrami mogą być macierze, wektory robocze (o tak, przydadzą się!), a nawet wskaźniki do funkcji udostępniających zmienne lokalne funkcjom używającym naszego kontekstu. Taka struktura, ze zmiennymi lokalnymi, funkcjami dostępu itp. mocno pachnie porządną klasą w C++ i tak jest w rzeczywistości: programowanie z użyciem kontekstów ma wiele cech programowania obiektowego[32].

Wracając do naszej konkretnej sytuacji, kontekst, jaki chcielibyśmy przekazać do `Funkcji()`, powinien zawierać dwie macierze: dyskretyzacji laplasjanu i pierwszej pochodnej. Kontekst ma być strukturą, więc wygodnie nam będzie najpierw zdefiniować odpowiedni typ strukturalny:

```
typedef struct {
  Mat A;
  Mat B;
} USERCTX;
```

a następnie zadeklarować zmienną tego typu, zaincjalizować kontekst, wykorzystać i porzucić:

```
int main(int argc, char **argv)
{
  USERCTX UserCtx;

  /* inicjalizacja kontekstu */
  USERCTXCreate( &UserCtx );

  /* ....wykorzystanie kontekstu.... */
  SNESSetFunction( MojSnes, X, R, Funkcja, (void *) &UserCtx );

  /* zwolnienie kontekstu */
  USERCTXDestroy( &UserCtx );
}
```

Funkcje `USERCTXCreate()` oraz `USERCTXDestroy()`, także zdefiniowane przez nas[33], ładnie oddzielą od reszty kodu etap "tworzenia" kontekstu (u nas: wygenerowanie macierzy i ich elementów) i jego usunięcia:

```
int USERCTXCreate( USERCTX *UserCtx )
{
  MatCreate( &UserCtx->A ); MatCreate( &UserCtx->B );
  MatSetValues( UserCtx->A );
  ...itd....
}
int USERCTXDestroy( USERCTX *UserCtx )
{
  MatDestroy( UserCtx->A ); ...itd...
}
```

Teraz użycie naszego kontekstu w `Funkcji()` wyglądałoby mniej więcej tak:

```
int Funkcja( SNES MojSnes, Vec X, Vec F, void * Ctx )
{
  USERCTX *UserCtx;
```

[32]Osoby dociekliwe z pewnością zauważyły, że typy (konteksty) `Vec`, `Mat`,`SLES` itp. maja bardzo wiele cech regularnych klas w C++.

[33]Podobieństwo nazw do, na przykład, `SLESCreate()` i `SLESDestroy()` jest całkowicie świadome.

```
  /* najpierw rzutujemy wskaznik na odpowiedni typ */
  UserCtx = (USERCTX *) Ctx;

  /* uzyj A i B do obliczenia F */
  MatMult( UserCtx->A, X, F ); VecAXPY(); MatMult(); itd...
  return(0);
}
```

13.5 Implementacja

Podobnie jak w przypadku układu 2x2, funkcję F i pochodną F' zdefiniujemy w osobnych plikach. Kod źródłowy funkcji `main()`, zapisany w kolejnym pliku, będzie zawierał zwarty kod realizujący ogólny schemat użycia SNESu.

Zaczynamy od kontekstu użytkownika

W kilku miejscach (w pliku definiującym funkcję F, w pliku definiującym pochodną F', w funkcji `main()`) będziemy się odwoływać do naszego kontekstu. Dobrze więc byłoby, gdybyśmy dołączali definicję tego typu z jednego, wspólnego pliku nagłówkowego, powiedzmy, `user.h`. Oto jego zawartość:

`user.h`

```
static Scalar zero = 0.0;
static Scalar one  = 1.0;
static Scalar mone = -1.0;
typedef struct
{
      Mat Lap;    /* macierz laplasjanu          */
      Mat Poch;   /* macierz pierwszej pochodnej */
      Vec X;      /* wektor rozwiazania          */
      Vec F;      /*        prawej strony        */
      Vec R;      /*        residuum             */
      Vec y;      /*        pomocniczy           */
      Vec z;      /*        pomocniczy           */
} USERCTX;
```

Oprócz macierzy A_h i B_h, które tutaj nazwaliśmy bardziej opisowo `Lap` i `Poch`, w kontekście umieściliśmy jeszcze kilka wektorów roboczych, które wykorzystamy potem w procedurach `Funkcja()` i `Jakobian()`.

Dodatkowo, zdefiniowaliśmy parę stałych typu `Scalar`, bardzo przydatnych do wykonania operacji obliczenia sumy lub różnicy dwóch wektorów przy pomocy funkcji `VecAXPY()`. Będą one dostępne w każdym pliku zawierającym dyrektywę `#include "user.h"`.

Implementacja funkcji *F(X)*

Procedura `Funkcja()` definiuje sposób wyliczania wartości *F(X)*. Oto plik zawierający jej kod źródłowy:

`cdfun.c`

```
#include "snes.h" /* plik z definicjami funkcji i obiektow SNES; zawiera takze
                     dyrektywe #include "sles.h" */
#include "user.h" /* kontekst uzytkownika i przydatne stale */

/*****************************************************
       Procedura obliczajaca wartosci funkcji F
*****************************************************/

int Funkcja(SNES MojSnes,Vec X,Vec F,void *Ctx)
{
int    ierr;
USERCTX *UserCtx = (USERCTX *)Ctx;

  /* Pomnoz przez pochodna centralna... */
```

```
  ierr = MatMult( UserCtx->Poch, X, UserCtx->y ); CHKERRQ(ierr);

  /* ...i domnoz przez u */
  ierr = VecPointwiseMult( X, UserCtx->y, F ); CHKERRQ(ierr);

  /* Pomnoz przez laplasjan */
  ierr = MatMult( UserCtx->Lap, X, UserCtx->y ); CHKERRQ(ierr);

  /* Zsumuj! */
  ierr = VecAXPY( &one, UserCtx->y, F ); CHKERRQ(ierr);
  ierr = VecAXPY( &mone, UserCtx->F, F ); CHKERRQ(ierr);
  return(ierr);
}
```

Wykorzystujemy wektory robocze kontekstu `UserCtx` do przechowywania wyników pośrednich, obliczanych według algorytmu z rozdziału 13.2 na stronie 57. Zwróćmy ponadto uwagę na użycie funkcji `VecAXPY()` ze stałymi `one` i `mone` (definiowanymi w pliku "`user.h`") dla obliczenia sumy i różnicy dwóch wektorów.

Implementacja macierzy pochodnej *F′(X)*

Jak pamiętamy, zdecydowaliśmy się implementować *F′(X)* w postaci macierzy powłokowej. Definiowanie macierzy powłokowej odbywa się w dwóch etapach: najpierw tworzymy powłokę komendą `MatShellCreate()`, a następnie przekazujemy do niej funkcje, realizujące operacje dopuszczalne na naszej macierzy, np. operację mnożenia: `MatShellSetOperation()` z odpowiednimi parametrami (patrz Rozdział „Macierze powłokowe" na stronie 18). Zakładając, że powłokowa macierz jakobianu jest już dobrze zdefiniowana - robimy to w programie głównym -możemy przystąpić do implementacji funkcji `Jakobian()`, przekazującą SNESowi macierz pochodnej: i tu czeka nas bardzo miła niespodzianka!

`cdjac.c`

```
/*****************************************************
                  Definicja pochodnej F
*****************************************************/
#include "snes.h"
#include "user.h"

/* Implementacja funkcji Jakobian(), przekazywanej do SNESu */

int Jakobian(SNES MojSnes, Vec X, Mat *Jac,
               Mat *Precond, MatStructure *flag, void *Ctx)
{
  *flag = SAME_NONZERO_PATTERN;
  return(0);
}

/* Implementacja mnozenia przez macierz jakobianu */

int MultJakobian( Mat Jac, Vec VecIn, Vec VecOut )
{
int ierr;
USERCTX *UserCtx;

  /* Najpierw musimy wydobyc kontekst */
  ierr = MatShellGetContext( Jac, (void **)&UserCtx );CHKERRQ(ierr);

  /* Pomnoz przez pochodna centralna... */
  ierr = MatMult( UserCtx->Poch, VecIn, UserCtx->y ); CHKERRQ(ierr);

  /* ...i domnoz przez u */
  ierr = VecPointwiseMult( UserCtx->X, UserCtx->y, VecOut ); CHKERRQ(ierr);
  /* Pomnoz przez laplasjan */
  ierr = MatMult( UserCtx->Lap, VecIn, UserCtx->y ); CHKERRQ(ierr);
  /* Zsumuj! */
  ierr = VecAXPY( &one, UserCtx->y, VecOut ); CHKERRQ(ierr);
```

```
  ierr = MatMult( UserCtx->Poch, UserCtx->X, UserCtx->y ); CHKERRQ(ierr);
  ierr = VecPointwiseMult( VecIn, UserCtx->y, UserCtx->z ); CHKERRQ(ierr);
  ierr = VecAXPY( &one, UserCtx->z, VecOut ); CHKERRQ(ierr);
  return(ierr);
}
```

Tak jest, funkcja `Jakobian()` właściwie *nie robi nic*. Rzeczywiście, skoro macierz jakobianu zadana jest przez operację mnożenia przez wektor - a to jest określone wcześniej przy definiowaniu macierzy powłokowej - wyznaczanie macierzy jakobianu w specjalnej procedurze jest zbędne. Niemniej, *musimy* przekazać jakąś sensowną funkcję `Jakobian()` do SNESu, więc stąd nasza definicja.

Ponieważ wszystko załatwia operacja mnożenia, jest to dobre miejsce na zdefiniowanie procedury mnożenia wektora przez macierz jakobianu. Ta procedura zostanie potem przekazana jako argument `MatShellSetOperation()`, dopełniając definiowanie powłokowej macierzy $F'(X)$. Procedurę `MultJakobian()`, definiującą *de facto* macierz $F'(X)$, wykorzystamy przy okazji tworzenia powłokowej macierzy jakobianu w programie głównym.

Program główny: wykorzystanie SNESu

Główny plik, poza funkcją `main()`, zawiera także funkcje tworzenia i niszczenia kontekstu użytkownika, `USERCTXCreate()` i `USERCTXDestroy()`.

`cd.c`

```
#include <stdio.h>
#include <math.h>   /* zawiera funkcje sin(x) */
#include "snes.h"
#include "user.h"

static char help[] = "Rozwiazuje nieliniowe jednowymiarowe rownanie\n\
konwekcji-dyfuzji na odcinku [0,L]\n\
Opcje: -N [int] liczba wewnetrznych wezlow siatki\n\
       -L [Scalar]  dlugosc odcinka\n\n";

/* Deklaracje procedur obliczajacych funkcje i jakobian */
extern int MultJakobian( Mat, Vec, Vec);
int Jakobian(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
int Funkcja(SNES,Vec,Vec,void*);
int USERCTXCreate(USERCTX*,int,Scalar);
int USERCTXDestroy(USERCTX*);

/*************************************************************
                     Program glowny
*************************************************************/

int main( int argc, char **argv )
{
  SNES     MojSnes;        /* obiekt SNES, ktorym rozwiazemy zadanie */
  Mat      MatJ;           /* macierz jakobianu */
  USERCTX  UserCtx;        /* kontekst uzytkownika */
  int          n;          /* liczba wezlow siatki */
  static Scalar L = M_PI;  /* dlugosc przedzialu calkowania,
                              zatem krok siatki h = L/n */
  int      ierr, flg, its;

  PetscInitialize( &argc, &argv,PETSC_NULL,help );

  /* Interpretacja opcji uzytkownika */
  ierr = OptionsGetInt( PETSC_NULL, "-N", &n, &flg ); CHKERRA(ierr);
  if( !flg ) n = 3;
  ierr = OptionsGetScalar( PETSC_NULL, "-L", &L, &flg );
  CHKERRA(ierr);

  /* Tworzenie potrzebnych obiektow: SNESu, wektorow, macierzy pomocniczych */
  ierr = SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS,&MojSnes);
```

```
CHKERRA(ierr);
  ierr = USERCTXCreate( &UserCtx, n, L ); CHKERRA(ierr);
  VecView( UserCtx.F, VIEWER_DRAWX_WORLD  ); PetscSleep(5);

  /* Przekaz SNESowi funkcje i wektor residualny */
  ierr = SNESSetFunction(MojSnes, UserCtx.R, Funkcja, &UserCtx);
CHKERRA(ierr);

  /* Przekaz SNESowi macierz jakobianu i funkcje obliczajaca jakobian */
  ierr = MatCreateShell(MPI_COMM_WORLD,n,n,n,n,&UserCtx,&MatJ); CHKERRA(ierr);
  ierr = MatShellSetOperation(MatJ, MATOP_MULT, MultJakobian ); CHKERRA(ierr);
  ierr = SNESSetJacobian(MojSnes,MatJ,MatJ,Jakobian,&UserCtx); CHKERRA(ierr);

  /* Uwzglednij opcje uruchomienia SNESu typu -snes_view -snes_monitor  */
  ierr = SNESSetFromOptions(MojSnes); CHKERRA(ierr);

  /* Rozwiaz! */
  ierr = SNESSolve(MojSnes,UserCtx.X,&its); CHKERRA(ierr);

  PetscPrintf(MPI_COMM_WORLD, "Liczba iteracji = %d\n", its);
  PetscPrintf(MPI_COMM_WORLD,"Wektor rozwiazania:\n");
  ierr = VecView(UserCtx.X, VIEWER_DRAWX_WORLD); CHKERRA(ierr);
  PetscSleep( 10 );

  /* Zwolnij pamiec */
  ierr = USERCTXDestroy(&UserCtx); CHKERRA(ierr);
  ierr = MatDestroy(MatJ); CHKERRA(ierr);
  ierr = SNESDestroy(MojSnes); CHKERRA(ierr);
  PetscFinalize();
  return 0;
}

/*************************************************************
           Funkcje zwiazane z kontekstem uzytkownika
*************************************************************/

int USERCTXCreate( USERCTX * UserCtx, int n, Scalar L )
{
extern int GenerujLaplasjan1D( Mat, Scalar );
extern int GenerujPochodnaCentralna1D( Mat, Scalar );
int ierr=0;
int i;
double val;
double h = L/(n+1); /* krok siatki */

  /* tworzymy wektory uzywane w kontekscie naszego zagadnienia */
  ierr = VecCreate(MPI_COMM_WORLD,PETSC_DECIDE,n,&UserCtx->X); CHKERRQ(ierr);
  ierr = VecDuplicate(UserCtx->X,&UserCtx->R); CHKERRQ(ierr);
  ierr = VecDuplicate(UserCtx->X,&UserCtx->F); CHKERRQ(ierr);
  ierr = VecDuplicate(UserCtx->X,&UserCtx->y); CHKERRQ(ierr);
  ierr = VecDuplicate(UserCtx->X,&UserCtx->z); CHKERRQ(ierr);

  /* tworzymy macierze uzywane w kontekscie naszego zagadnienia */
  ierr = MatCreate(MPI_COMM_WORLD,n,n,&UserCtx->Lap); CHKERRQ(ierr);
  ierr = MatCreate(MPI_COMM_WORLD,n,n,&UserCtx->Poch); CHKERRQ(ierr);

  /* generujemy elementy macierzy pomocniczych */
  ierr = GenerujLaplasjan1D( UserCtx->Lap, (Scalar) h ); CHKERRQ(ierr);
  ierr = MatView( UserCtx->Lap, VIEWER_DRAWX_WORLD ); CHKERRQ(ierr);
  PetscSleep( 3 );
  ierr = GenerujPochodnaCentralna1D( UserCtx->Poch, (Scalar) h
);CHKERRQ(ierr);
  ierr = MatView( UserCtx->Poch, VIEWER_DRAWX_WORLD ); CHKERRQ(ierr);
  PetscSleep( 3 );
```

```
  /* generujemy prawa strone */
  for( i = 1 ; i <= n ; i++ )
  {
      val = sin(i*h) + sin(i*h)*cos(i*h); /* rozwiazanie u(x) = sin(x) */
      VecSetValue( UserCtx->F, i-1, (Scalar)val, INSERT_VALUES );
  }

  /* Przyblizenie poczatkowe */
  /* Zakladamy, ze zerowe... Od razu wpisujemy do wektora rozwiazania X */
  ierr = VecSet(&zero, UserCtx->X); CHKERRQ(ierr);
  return(ierr);
}

int USERCTXDestroy( USERCTX * UserCtx )
{
int ierr;
  ierr = VecDestroy(UserCtx->X); CHKERRQ(ierr);
  ierr = VecDestroy(UserCtx->R); CHKERRQ(ierr);
  ierr = VecDestroy(UserCtx->F); CHKERRQ(ierr);
  ierr = VecDestroy(UserCtx->z); CHKERRQ(ierr);
  ierr = VecDestroy(UserCtx->y); CHKERRQ(ierr);
  ierr = MatDestroy(UserCtx->Lap); CHKERRQ(ierr);
  ierr = MatDestroy(UserCtx->Poch); CHKERRQ(ierr);
  return(ierr);
}
```

Program główny - funkcja `main()` - jest znów prościutki, co więcej, jest on prawie identyczny z programem rozwiązującym problem 2x2 z poprzedniego przykładu. To bardzo ważne spostrzeżenie, albowiem oznacza, że raz nauczywszy się posługiwać SNESem, będziemy mogli używać go bez kłopotu w innych sytuacjach[34].

Po wstępnych `#include` następuje lista prototypów procedur - zdefiniowanych gdzie indziej, np. w innych plikach - a wykorzystywanych poniżej. Przydaje się to kompilatorowi dla pełniejszego sprawdzania poprawności wywołań tych funkcji. W samej funkcji `main()` deklarujemy na początku kilka zmiennych lokalnych, w tym obiekt SNESu `MojSnes`, macierz (za chwilę okaże się, że powłokową) jakobianu `MatJ` oraz kontekst `UserCtx`, zawierający m.in. macierze laplasjanu i pierwszej pochodnej, a także wektory rozwiązania i residuum. Zaczynamy klasycznie, od inicjalizacji PETSc oraz ściągnięcia opcji użytkownika:

```
PetscInitialize( &argc, &argv,PETSC_NULL,help );

/* Interpretacja opcji uzytkownika */
OptionsGetInt( PETSC_NULL, "-N", &n, &flg );
if( !flg ) n = 3;
OptionsGetScalar( PETSC_NULL, "-L", &L, &flg );
```

Dalej, tworzymy SNES i kontekst użytkownika

```
/* Tworzenie potrzebnych obiektow: SNESu, wektorow, macierzy pomocniczych */
SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS,&MojSnes);
USERCTXCreate( &UserCtx, n, L );
```

Następnie, jak zwykle, „nakarmimy” SNES danymi zadania. Najpierw przekażemy mu funkcję:

```
/* Przekaz SNESowi funkcje i wektor residualny */
SNESSetFunction(MojSnes, UserCtx.R, Funkcja, &UserCtx);
```

Zanim przekażemy SNESowi jakobian, musimy najpierw utworzyć macierz (powłokową) jakobianu:

```
MatCreateShell(MPI_COMM_SELF,n,n,n,n,&UserCtx,&MatJ);
MatShellSetOperation(MatJ, MATOP_MULT, MultJakobian );
```

Kontekstem wykorzystywanym przez macierz jakobianu będzie oczywiście nasz `UserCtx`. Jako procedurę mnożenia przekazujemy procedurę `MultJakobian()`. Działa ona zgodnie z tym, co wcześniej wymyśliliśmy w Rozdziale „Implementacja macierzy pochodnej *F'(X)*” na stronie 61.W końcu, możemy przekazać SNESowi (teraz już dobrze określoną) macierz jakobianu `MatJ` i (prawie pustą) funkcję `Jakobian()`:

[34]Kłopoty pojawiają się, a jakże, ale gdzie indziej: na przykład w dopasowaniu opcji SNESu do danego zagadnienia, albo w definicji pochodnej, wyborze lub implementacji preconditionera, itp.

```
SNESSetJacobian(MojSnes,MatJ,MatJ,Jakobian,&UserCtx);
```

Dalszy ciąg programu to pełny standard: uwzględniamy w SNESie opcje linii komend, rozwiązujemy, oglądamy i na koniec sprzątamy po sobie. Dlatego na tym zakończymy analizę funkcji `main()`. Podkreślmy raz jeszcze, że praktycznie nie różni się ona od funkcji `main()` z poprzedniego przykładu!

Funkcje związane z kontekstem `USERCTX`

Są to `USERCTXCreate()` i `USERCTXDestroy()` - mało skomplikowane funkcje, więc komentarze w pliku źródłowym w wystarczającym stopniu opisują ich działanie. Zwróćmy jedynie uwagę na to, że generowaniem elementów macierzy laplasjanu i pierwszej pochodnej zajmują się *zewnętrzne* funkcje `GenerujLaplasjan1D()` i `GenerujPochodnaCentralna1D()`, zdefiniowane gdzie indziej zapewne w jeszcze jednym pliku, którego tu wcale nie zamieszczamy, zostawiając opracowanie tych procedur Czytelnikowi jako ćwiczenie.

13.6 Ćwiczenia

A) Uzupełnij powyższy program o funkcję `GenerujLaplasjan1D()` i `GenerujPochodnaCentralna1D()` i przetestuj go dla rozmaitych prawych stron zagadnienia konwekcji-dyfuzji.

B) Rozwiąż zagadnienie konwekcji-dyfuzji w obszarze dwuwymiarowym.

C) Rozwiąż zagadnienie $-\Delta u + u^3 = f$, z jednorodnymi warunkami brzegowymi Dirichleta.

D) Zastanów się, jak wykorzystać moc SNESu do implementacji metody typu Predyktor-Korektor rozwiązywania układu równań różniczkowych zwyczajnych. (Na każdym kroku schematu PC będziemy musieli rozwiązać równanie nieliniowe!)

E) Trudne, ale warto... Napisz program, rozwiązujący równanie Naviera-Stokesa w obszarze L-kształtnym:

$$\begin{cases} -\Delta u + (u \cdot \nabla)u + \nabla p = f, \\ \nabla \cdot u = 0. \end{cases}$$

Wybór schematu numerycznego dla aproksymacji tego równania nie jest trywialny.

14. Co jeszcze warto wiedzieć o SNESie?

Nie ma czegoś takiego jak idealny solver równań nieliniowych. Często bywa, że dla uzyskania satysfakcjonującej zbieżności metody typu Newtona należy dobrać odpowiednie parametry "relaksacyjne", kryteria stopu, czy opcje pracy solvera liniowego. Przyjrzyjmy się zatem rozmaitym opcjom SNESu.

14.1 Warianty metody Newtona

Jak wspominaliśmy na samym początku części dotyczącej równań nieliniowych, w PETSc zaimplementowano pewne warianty metody Newtona, globalizujące zbieżność procesu iteracyjnego. Zasadniczo, mamy do wyboru dwa warianty: *line search methods* oraz *trust region methods*, różniące się sposobem "uściślania" przybliżenia generowanego przez iterację Newtona. W przypadku metod poszukiwania kierunkowego (*line search*) z grubsza chodzi o to, że przyjmuje się, iż wyliczona poprawka na x_n jest co prawda wektorem o dobrym kierunku, ale nie zawsze dobrej długości. W najprostszym wariancie poszukiwania kierunkowego stopniowo zmniejsza się długość wektora poprawki, tak, aby ostatecznie uzyskać *zmniejszenie* residuum na danym kroku metody Newtona. To nam przynajmniej będzie gwarantowało, że kolejne iteracje metody będą dawały coraz lepsze przybliżenia (w sensie residuum). Więcej na ten temat można dowiedzieć się z Rozdz. 8 interesującej monografii Kelley'a [11]. Metoda *trust region* jest trochę bardziej skomplikowana i nie będziemy jej tu omawiać, odsyłając zainteresowanych do [13]. Standardowo, w SNESie używany jest wariant *qubic line search*.

14.2 Metody poszukiwania kierunkowego

Ustawiając opcję `-snes_type ls` decydujemy się na wariant *line search*, czyli poszukiwania kierunkowego. Wybór czynnika skalującego może być oparty na pewnej interpolacji zachowania się metody w poprzednich iteracjach.

Możemy teraz wybierać stopień wielomianu interpolacyjnego używanego do wyznaczenia lepszego czynnika skalującego: `-snes_eq_ls [quadratic,cubic]`, to znaczy, odpowiednio, funkcją kwadratową lub sześcienną. Więcej szczegółów Czytelnik znajdzie u Kelley'a [11] oraz w [14]. Dodatkowo, możliwy jest także wariant

`-snes_eq_ls basic`. Taki wybór oznacza rezygnację z techniki line search i pozostanie przy klasycznej metodzie Newtona.

Metody wyszukiwania kierunkowego wymagają ustawienia pewnych parametrów relaksacyjnych, `-snes_eq_ls_alpha <alpha>`, `-snes_eq_ls_maxstep <max>`, `-snes_eq_ls_steptol <tol>`, gdzie oznaczenia są zgodne z [4]. Na szczęście, predefiniowane wartości tych parametrów są zazwyczaj rozsądne dla wielu zagadnień.

14.3 Parametry stopu

Przy rozwiązywaniu zagadnienia nieliniowego za pomocą SNESu mamy możliwość ustawienia wartości krytycznych dla kilku warunków stopu, a także zdefiniowania własnego kryterium. Standardowo, możemy określić wartości `atol`, `rtol` i `stol`, odpowiedzialnych za stwierdzenie, odpowiednio, spełnienia warunku tolerancji błędu bezwzględnego lub względnego, lub stagnacji:

- $\|F(x_n)\|_2 <$ `atol`,
- $\frac{\|F(x_n)\|_2}{\|F(x_0)\|_2} <$ `rtol`,
- $\|F(x_n) - F(x_{n-1})\|_2 <$ `stol`.

Te wszystkie parametry, oraz dodatkowo, maksymalną dopuszczalną liczbę iteracji `maxits` i wywołań *F(X)*, `maxfcts`, można ustawić poleceniem:

```
ierr = SNESSetTolerances(SNES snes,double rtol,double atol,double stol,
int maxits,int maxfcts);
```

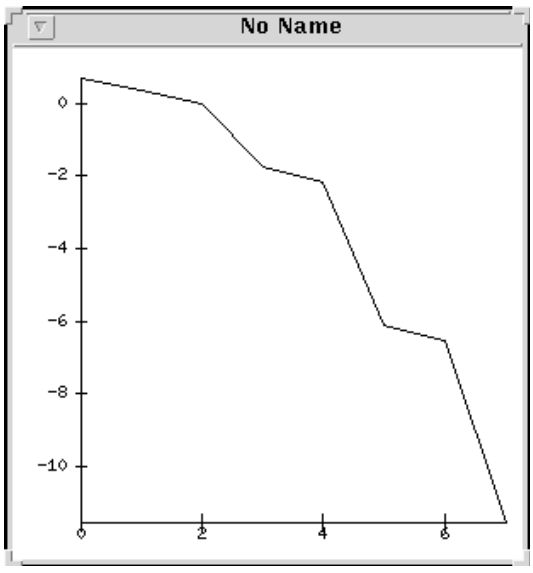
lub z linii komend: `-snes_atol <atol>`, `-snes_rtol <rtol>`, `-snes_stol <stol>`, `-snes_max_it <maxits>` oraz `-snes_max_funcs <maxfcts>`.

14.4 Inne opcje

Oczywiście, możliwe jest użycie *każdej* opcji dotyczącej SLESu (solverów liniowych). Będą one uwzględnione w procesie rozwiązywania układów z macierzą jakobianu.

Zmianę opcji SLESu wykorzystywanego przez SNES możemy uzyskać z programu, wyłuskując kontekst SLESu funkcją **`int SNESGetSLES( SNES MojSnes, SLES * MojSnesSles )`** a następnie używając procedur typu `SLESGetKSP()` lub `SLESGetPC()`. Naturalnie, to samo uzyskamy podając odpowiednie opcje w linii komend.

Zazwyczaj programy (w początkowym stadium) nie są wolne od błędów. Można się w szczególności obawiać, że zaprogramowana przez nas macierz jakobianu nie jest idealna. Na szczęście, możemy porównać działanie programu z naszą macierzą jakobianu i programu, w którym zostanie użyta jej aproksymacja metodą różnic skończonych. W wypadku, gdy używamy macierzy typu powłokowego, możemy naszą funkcję `Jakobian()`, realizującą działanie jakobianu, *zastąpić* jej aproksymacją generowaną wewnętrznie przez PETSc metodą różnic skończonych: wystarczy uruchomić program z opcją `-snes_mf`. Podobny skutek można uzyskać (ale już tylko dla niepowłokowej macierzy jakobianu) stosując opcję `-snes_fd`. Nie znamy bliżej różnic między tymi opcjami. Po co stosować te opcje? Na pewno przydadzą się one tym, którzy nie potrafią (lub nie mają czasu) implementować funkcji `Jakobian()`. Czasem macierz jakobianu może być skomplikowana! Autorzy oficjalnego *manuala* twierdzą ponadto, że możemy w ten sposób łatwo sprawdzić, czy poprawnie zaimplementowaliśmy jakobian: bo jeśli nie, to, rozwiązując zagadnienie raz z własnym jakobianem, a drugi raz z opcją `-snes_mf`, powinniśmy dostać istotnie różniące



Rysunek 11. Przebieg zbieżności SNESu, z wykorzystaniem opcji `-snes_xmonitor`.

się wyniki. To prawda, ale od razu chcielibyśmy ostrzec wszystkich pesymistów: jeśli postępując w wyżej opisany sposób, *rzeczywiście* dostaniecie różne wyniki (rozwiązania, liczbę iteracji, itp.) nie wpadajcie[35] w desperację. Jest bardzo prawdopodobne, że po prostu rozwiązujecie swoje zadanie z niedobrymi parametrami! Pamiętajcie, że wyznaczenie aproksymacji jakobianu jest numerycznie bardzo kosztowne (wymaga za każdym razem obliczania wielu wartości `Funkcji()`) a jednym z ograniczeń ustawianym w SNESie jest właśnie liczba wywołań funkcji!

14.5 Śledzenie pracy SNESu

Przebieg procesu iteracyjnego możemy śledzić używając opcji linii komend `-snes_xmonitor` lub `-snes_monitor`.

Ponadto, każdy SNES posiada (oczywiście!...) swoją przeglądarkę, wyświetlającą parametry wywołania SNESu: **`int SNESView(SNES snes,Viewer viewer)`**, gdzie, jak zwykle, jako `viewer` możemy podać `VIEWER_STDOUT_WORLD`. Z linii komend uzyskamy ten sam efekt podając opcję `-snes_view`. Oto przykładowy wynik obejrzenia SNESu:

```
SNES Object:
  method: ls
    line search variant: SNESCubicLineSearch
    alpha=0.0001, maxstep=1e+08, steptol=1e-12
  maximum iterations=50, maximum function evaluations=1000
  tolerances: relative=1e-08, absolute=1e-50, truncation=1e-12, solution=1e-08
  total number of linear solver iterations=13
  total number of function evaluations=6
KSP Object:
  method: gmres
    GMRES: restart=30, using Modified GramSchmidt Orthogonalization
  maximum iterations=10000, initial guess is zero
  tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
  left preconditioning
PC Object:
  method: none
  linear system matrix = precond matrix:
Matrix Object:
  type=MATSHELL, rows=343, cols=343
```

15. Podsumowanie SNESu

Składnia podstawowych procedur SNESu:

Składnia	opis
`int` **`SNESCreate(`**`MPI_Comm comm,SNESProblemType mode,SNES *MojSnes`**`)`**	Tworzy obiekt typu SNES;
`int` **`SNESSetFunction(`**`SNES MojSnes, Vec R, int (*Funkcja)(SNES,Vec,Vec,void*),void *FunCtx`**`)`**	Przekazuje SNESowi procedurę obliczającą *F(X)*;
`int` **`SNESSetJacobian(`**`SNES MojSnes,Mat Jac,Mat Prec, int(*Jakobian)(SNES,Vec,Mat*,Mat*,MatStructure*,void*), void *JacCtx`**`)`**	Przekazuje SNESowi procedurę obliczającą jakobian *F'(X)*;
`int` **`SNESSetFromOptions(`**`SNES MojSnes`**`)`**	Ustawia opcje SNESu zgodnie z argumentami przekazanymi w linii komend przy wywołaniu programu;
`int` **`SNESSolve(`**`SNES MojSnes,Vec X,int *its`**`)`**	Rozwiązuje równanie *F(X)*=0;
`int` **`SNESView(`**`SNES MojSnes,Viewer viewer`**`)`**	Wypisuje parametry SNESu.

Ważne argumenty tych procedur:

- obiekt typu SNES: `MojSnes` - służący do rozwiązywania równań nieliniowych;

[35]...przynajmniej nie od razu...

- wektory: `R` - wektor residuum, `X` - wektor rozwiązania (przed uruchomieniem `SNESSolve()` wpisujemy do niego przybliżenie początkowe!);
- macierze: `Jac` - macierz jakobianu, `Prec` - macierz, na podstawie której będzie budowany preconditioner dla `Jac` (zazwyczaj wystarcza wziąć `Jac` jako `Prec`);
- funkcje:

  `int Funkcja(SNES MojSnes,Vec X,Vec R,void *FunCtx);` procedura wyznaczająca wektor wartości funkcji `R` na danym wektorze `X`;

  `int Jakobian(SNES MojSnes,Vec X,Mat *Jac,Mat *Prec,int *flag, void *JacCtx);` funkcja wyznaczająca macierz jakobianu `Jac` i preconditionera `Prec` na podstawie wartości rozwiązania na kolejnym kroku `X`; w wyniku musi ustawiać wartość paramteru `flag` identyfikującego zmianę struktury macierzy `Prec`, analogicznie jak w `SLESSetOperators()`; opis użycia tej funkcji w przypadku, gdy te macierze są macierzami powłokowymi, znajduje się w rozdziale „Definicja funkcji F” na stronie 57.

- konteksty użytkownika: `FunCtx` - struktura w C, czyli kontekst, zawierający parametry i prywatne zmienne potrzebne do wyliczenia *F(X)*, wykorzystywany przez `Funkcja()`; `JacCtx` - kontekst zawierający parametry i prywatne zmienne potrzebne do wyliczenia *F′(X)*, wykorzystywany przez `Jakobian()`; najczęściej, tworzymy jeden wspólny dla obu funkcji kontekst; jeśli nie korzystamy z kontekstów, przekazujemy do odpowiednich funkcji `PETSC_NULL`.

Mniej ważne argumenty:

- komunikator MPI: `comm`, u nas zawsze podajemy `MPI_COMM_WORLD`;
- tryb uruchomienia SNESu: `mode` - u nas zawsze będzie to `SNES_NONLINEAR_EQUATIONS` - sygnalizujemy, że SNES będzie wykorzystany do rozwiązania układu równań nieliniowych[36];
- przeglądarka dla SNESu: `viewer` - musi być tekstowa, u nas zawsze `VIEWER_STDOUT_WORLD`;
- `its` - liczba iteracji wykonanych przez SNES; w przypadku stwierdzenia niezbieżności, liczba podawana jest ze znakiem minus.

Schemat wykorzystania SNESu do rozwiązania równania nieliniowego:

- określić kontekst `USERCTX` używany przez SNES;
- zdefiniować funkcję *F* i jej jakobian *F′*, tworząc odpowiednie procedury `Funkcja()` i `Jakobian()`; korzystające z kontekstu typu `USERCTX`;
- w programie głównym zadeklarować potrzebne obiekty: `USERCTX UserCtx; SNES MojSnes; Vec X,R; Mat Jac; int flag,its;`, a następnie wygenerować wektory, macierze i (zapewne) kontekst użytkownika;
- wygenerować SNES:
  `SNESCreate(MPI_COMM_WORLD, SNES_NONLINEAR_EQUATIONS, &MojSnes);`
- przekazać SNESowi naszą funkcję i jakobian:
  `SNESSetFunction(MojSnes, X, Funkcja, (void *) &UserCtx);` oraz
  `SNESSetJacobian(MojSnes, Jac, Jac, Jakobian, (void *) &UserCtx);`
- ustawić opcje w SNESie, `SNESSetFromOptions(MojSnes);`
- wpisać przybliżenie początkowe do wektora `X`;
- rozwiązać równanie! `SNESSolve(MojSnes, X, &its);`

Na zakończenie, podkreślmy raz jeszcze kilka istotnych kwestii odnoszących się do *implementacji* ogólnego schematu użycia SNESu.

Zarówno procedura obliczająca funkcję *F(X)*, jak procedura obliczająca jakobian *F'(X)*, muszą być ściśle określonego typu. Przed użyciem SNESu, musimy wygenerować *niezłe* przybliżenie początkowe. Efektywność całego pro-

[36] Inna możliwa wartość tego parametru to `SNES_UNCONSTRAINED_MINIMIZATION`, dzięki której wygenerujemy SNES w kontekście rozwiązywania zagadnień optymalizacyjnych (niestety, zmienia się wtedy także cały schemat użycia SNESu).

cesu rozwiązywania równania będzie zależała przede wszystkim od sposobu implementacji *F(X)* oraz *F'(X)*. Warto także zastanowić się, czy dobrze są dobrane parametry SNESu.

O czym nie napisaliśmy

Jest jeszcze kilka modułów w PETSc, które można wykorzystać w programowaniu solverów równań cząstkowych.

Oprócz SLESu i SNESu, mamy także moduł TS (ang. *time stepping*), z implementacją metod różnicowych rozwiązywania wielkich układów równań różniczkowych zwyczajnych powstałych na przykład po dyskretyzacji (względem zmiennej przestrzennej) równania parabolicznego. Dodajmy, że PETSc ma możliwość wykorzystywania w tym celu także biblioteki zewnętrznej PVODE autorstwa A.Hindmarsha i in., będącej równoległą wersją biblioteki CVODE (dość wyrafinowanych metod wielokrokowych).

PETSc posiada wiele opcji profilowania kodu. Jedną z nich można uruchomić[37] dodając do linii komend `-petsc_log`. Użytkownik ma wtedy możliwość sprawdzenia, ile czasu CPU (lub jaki procent obliczeń) pochłania wybrany fragment kodu. Można też śledzić sposób wykorzystywania pamięci, obciążenie poszczególnych procesorów, itp. Poza wydrukiem tekstowym, jest także możliwość obejrzenia przebiegu pracy programu w okienku graficznym. Przyznajemy, że trochę żałujemy, iż do tej pory nie zaczęliśmy w pełni wykorzystywać tych opcji.

Dodatkowo, dla *prawdziwych i porządnych* programistów-profesjonalistów, PETSc daje możliwość podłączenia debuggera - zarówno dopiero po wystąpieniu błędu, jak też od początku pracy programu. Debugger może się podłączać na każdym procesorze w osobnym okienku - każdy, kto doświadczył koszmaru programowania równoległego, wie, jakie to może być ważne. I znów, jako nieprofesjonaliści, nie korzystamy z tej sposobności, gorzej, nawet nie bardzo wiemy jak obsługiwać `dbx`[38]. Powód jest prozaiczny: po prostu do tego stopnia zepsuci jesteśmy pięknymi, przyjaznymi i naprawdę porządnymi debuggerami ze środowiska MS Windows (a nawet DOS!), że uczenie się topornego, tekstowego `dbx`'a wydaje się nam niepotrzebną męką: porównanie `dbx`'a i debuggera dla (powiedzmy) Visual C++ wypada tak, jak porównanie edytora `vi` z (powiedzmy) MS Word. Dlatego wolimy poczekać, aż pojawią się porządne debuggery także w Unix'ie (albo, o zgrozo, Windows wyprze i ten system...). Niemniej, osoby chętne, a tym bardziej pragnące pisać *duże* aplikacje, zachęcamy do wykorzystywania dostępnych debuggerów: jest to inwestycja, która prędzej, czy później, sowicie się opłaci!

Dalej, PETSc dysponuje pewnymi możliwościami graficznymi, ale szczerze powiedziawszy, lepiej nie używać procedur PETSc do wizualizacji: sensowniej jest wykorzystać zewnętrzne programy wizualizacji danych, na przykład AVS, Vigie, czy nawet Matlab. Warto natomiast skorzystać z grafiki PETSc dla obserwacji zachowania własnych obiektów lub procedur. Można w szczególności napisać własne - lepsze! - przeglądarki obiektów itp.

PETSc jest tworzone jako biblioteka otwarta. Jak już wspominaliśmy, PETSc może korzystać z zewnętrznej biblioteki PVODE; jest też port do kilku innych. Między innymi, do bibliotek równoległych BlockSolve95 i SPAI, z dodatkowymi solverami równań liniowych oraz do biblioteki matematycznej IBM ESSL; do Matlaba - dla wizualizacji i ewentualnego postprocessingu numerycznego; do narzędzia wizualizacji trójwymiarowych obiektów przy użyciu VRML (*virtual reality*), itd.

Najistotniejszą cechą PETSc, która czyni z niej naprawdę poważne narzędzie, jest przejście na programowanie równoległe na wyższym poziomie. To właśnie, w połączeniu z dostępnością podstawowych solverów i narzędzi numerycznych, stanowi *prawdziwą smakowitość* PETSc! Dlatego zamierzamy w ewentualnej drugiej części tego podręcznika przystępnie wyłożyć generalne zasady programowania równoległego oraz wykorzystania PETSc do (w miarę bezbolesnego) pisania efektywnych aplikacji równoległych, służących rozwiązywaniu równań różniczkowych cząstkowych. Na razie zachęcamy do własnego eksperymentowania z programami równoległymi, wykorzystującymi MPI do uruchamiania jednej aplikacji na kilku stacjach roboczych, co może pozwolić przede wszystkim na rozwiązywanie dużo większych zadań dyskretnych.

Zachęcamy do uczenia się PETSc. To naprawdę szybsze, niż kodowanie wszystkiego od zera...

[37] Pod warunkiem, że program został skompilowany z opcją `-DPETSC_LOG`.

[38] Popularny debugger Unix'owy.

Bibliografia

.[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen: *LAPACK Users Guide - Release 2.0*, SIAM, February 1995.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes and Barry F. Smith: *PETSc 2.0 Users Manual*, ANL-95/11 – Revision 2.0.22, Argonne National Laboratory, 1998. Można go znaleźć w katalogu `$(PETSC_DIR)/docs/` lub na stronie domowej PETSc: `http:/www.mcs.anl.gov/petsc`. Znajdują się tam dwie wersje: `manual.html` (do oglądania przeglądarką WWW) i `manual.ps` (plik w Postscripcie).

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes and Barry F. Smith: *PETSc man pages.* Zawierają opis wszystkich funkcji PETSc. Można je znaleźć w katalogu `$(PETSC_DIR)/docs`, lub na stronie domowej PETSc.

[4] Jr. Dennis, John E. and Robert B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations,* Prentice-Hall Series in Computational Mathematics, Prentice-Hall Inc., Englewood Cliffs, N.J., 1983.

[5] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen and Henk A. van der Vorst. *Solving linear systems on vector and shared memory computers,* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1991.

[6] Maksymilian Dryja, Janina i Michał Jankowscy: *Przegląd metod i algorytmów numerycznych*, Wydawnictwa Naukowo-Techniczne, Warszawa 1988.

[7] Stanley C. Eisenstat and Homer F. Walker*: Choosing the forcing terms in an inexact Newton method,* SIAM J. Sci. Comput., 17(1):16--32, 1996. Special issue on iterative methods in numerical linear algebra (Breckenridge, CO, 1994).

[8] G.H. Golub and Ch.F. Van Loan: *Matrix computations,* Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, Baltimore, MD 1996.

[9] W. D. Gropp: *The Message Passing Interface (MPI) standard*, `http://www.mcs.anl.gov/mpi/`

[10] W. Hackbusch: *Iterative solution of large sparse systems of equations*, Applied Mathematical Sciences, vol. 95, Springer-Verlag, New York 1994.

[11] C. T. Kelley: *Iterative methods for linear and nonlinear equations*, volume 16 of Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995. With separately available software.

[12] Brian W. Kernighan, Dennis M. Ritchie: *Język C*, Wydawnictwa Naukowo Techniczne, Warszawa 1988.

[13] Jorge J. Mor'e, Danny C. Sorensen, Burton G. Garbow and Kenneth E. Hillstrom: *The MINPACK project,* in: Wayne R. Cowell, editor, Sources and Development of Mathematical Software, pages 88--111. 1984.

[14] Jorge J. Mor'e and David J. Thuente: *Line search algorithms with guaranteed sufficient decrease,* ACM Trans. Math. Software, 20(3):286--307, 1994.

[15] J. M. Ortega and W. C. Rheinboldt. *Iterative solutions of nonlinear equations in several variables,* Academic Press, New York, 1970.

[16] Barry Smith, Petter Bjørstad and Willam Gropp: *Domain Decomposition - Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press 1996.

[17] G. W. Stewart: *Afternotes on numerical analysis,* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1996.

[18] J. Stoer and R. Bulirsch: *Wstęp do analizy numerycznej*, Biblioteka matematyczna, PWN, Warszawa, 1995.