Pracownia nr 2

24.10.2009

1 Zadanie : Ciąg Fibonacciego

Zaprogramujemy ciąg zdefiniowany w sposób rekurencyjny. Jest to taki opis, który definiuje kolejny wyraz ciągu jako funkcje wyrazów poprzednich. Rozważmy ciąg liczbowy, którego n-ty wyraz jest równy sumie dwóch poprzedzających wyrazów, tzn.:

$$a_n = a_{n-1} + a_{n-2}$$

$$a_1 = 1 \quad a_2 = 1$$

Jest to tak zwany *ciąg Fibonacciego* (r. 1202).

1.1 Definicja ciągu

1. Napiszemy funkcję obliczającą n-ty wyraz ciągu Fibonacciego. Nazwiemy ją **fibo**:

```
function f = fibo(n)
 if ( n == 1 || n == 2)
  f = 1;
 else
  f = fibo(n-1) + fibo(n-2);
 end
end
```

   Zastosowaliśmy instrukcję warunkową **if** dla sprawdzenia warunku 'czy **n** jest równe 1 lub **n** równe 2?'. Zwróćmy uwagę na operator porównania '==' - czy równe? - oraz operator logiczny lub '||'.

   Jeżeli warunek jest spełniony wtedy a_n jest równe 1. W przeciwnym wypadku - **else** - wywołujemy ponownie funkcję **fibo** dla poprzednich wartości **n**. Instrukcję warunkową kończymy słowem **end**.

   Konstrukcję funkcji w której, odwołujemy się do niej samej, do momentu aż nie dojdziemy do znanej wartości, nazywamy również *rekurencyjną*.

2. Co możemy jeszcze zauważyć w funkcji **fibo**? Nie przejdzie sztuczka z poprzedniego zadania: nie da się jej zastosować podając jako argumentu kilku kolejnych liczb naturalnych w tablicy. Dlaczego?

3. Aby otrzymać kolejne wyrazy ciągu Fibonacciego dla $n = 1, 2, 3, ...$ musimy posłużyć się tak zwaną *pętlą*. Pętla polega na powtarzaniu w kółko pewnych instrukcji zgodnie z góry zadaną regułą. Może to wyglądać następująco:

```
>> n = 12;
>> for i = 1 : n
>>   a(i) = fibo(i);
>> end
```

   Tutaj reguła jest taka: dla (**for**) **i** od 1 do **n** wykonuj przypisanie do kolejnego elementu tablicy **a** kolejnego wyrazu ciągu Fibonacciego.

4. Tablica **a** zawiera teraz pierwsze **n** wyrazów ciągu. Spróbujmy je narysować na wykresie:

```
plot(a,'+')
```

5. Zapiszmy kolejne komendy, pętle i rysowanie, jako nowy skrypt. Nazwijmy go **ciag_fibo.m**; spróbujmy zwiększyć wartość **n** i zobaczmy na wykresie jak szybko rosną kolejne wyrazy ciągu.

```
>> clear all
>> ciag_fibo
```

1.2 Ilustracja graficzna

Leonardo z Pizy (czyli Fibonacci) rozważał taki ciąg w celu opisu przyrostu królików w populacji. Wyobraźmy sobie klatkę w której żyją króliki, które rozmnażają się zgodnie z kolejnymi wyrazami ciągu. Spróbujemy to narysować, żeby zobaczyć jak szybko klatka zapełnia się królikami.

1. Niech klatka będzie kwadratem o boku równym 1, a króliki oznaczmy kółeczkami. Kolejny wyraz ciągu Fibonacciego oznacza liczbę królików aktualnie znajdujących się w klatce. Skorzystamy ze skryptu **ciag_fibo** dopisując kilka nowych linii:

```
>> clear all
>> hold on
>> n = 100;
>> for i = 1 : n
>>   a(i) = fibo(i);
>>   if ( i > 1 )
>>       x = rand(1, a(i) - a(i-1));
>>       y = rand(1, a(i) - a(i-1));
>>       plot(x,y,'o')
>>     drawnow
>>   end
>> end
```

2. Do pętli dopisaliśmy instrukcję warunkową wewnątrz której rysujemy króliki. Funkcja **rand** tworzy losowe tablice współrzędnych **x,y** w których umieszczone będą środki kolejnych rysowanych kółeczek, oznaczających nowe króliki.

3. Zapiszmy nowy skrypt jako **kroliki_fibo.m** i zobaczmy jak to działa:

```
>> clear all
>> kroliki_fibo
```

2 Zadanie : Przybliżanie liczby π

Obliczymy przybliżoną wartość liczby π. Skorzystamy w tym celu z następującego wzoru na pole koła

$$p = \pi r^2$$

Biorąc zbiór punktów o współrzędnych całkowitych z kwadratu $[-r, r] \times [-r, r]$, dla każdego punktu możemy sprawdzić, czy leży on w kole o środku w punkcie $(0, 0)$ i promieniu r. Liczba takich punktów przybliża pole p koła wpisanego w kwadrat. Zatem przybliżoną wartość liczby π możemy uzyskać ze wzoru

$$\pi = \frac{p}{r^2}$$

2.1 Pierwsze przybliżenie $r = 2$

1. Tworzymy zbiór punktów w kwadracie $[-r, r] \times [-r, r]$. W tym celu utworzymy dwie tablice **x** i **y** zawierające wartości odpowiednich współrzędnych punktów w kwadracie. Proszę zwrócić uwagę na zastosowanie operatora '**:**' oraz '''

   ```
   >> r = 2;
   >> x = [-r : r]
   >> y = -x'
   ```

   Operator ''' to tak zwana *transpozycja* - ustawia tablicę **x** "na głowie".

2. Następnie tworzymy macierze - czyli tablice tablic - **X** i **Y** odpowiadające zbiorom współrzędnych x i y punktów kwadratu. Współrzędnym każdego punktu kwadratu odpowiada para elementów znajdujących się na tym samych miejscach w macierzach.

   ```
   >> X = [x;x;x;x;x]
   >> Y = [y,y,y,y,y]
   ```

3. Zliczamy punkty leżące w kole o promieniu r. Liczbę tych punktów zapiszmy w **p**. W tym celu musimy sprawdzić dla każdego punktu o współrzędnych (x, y), czy $\sqrt{x^2 + y^2} \leq r$. Możemy to zrobić w następujący sposób:

   ```
   >> sqrt( X.^2 + Y.^2) <= r
   ```

4. O co chodzi? Dla każdego elementu macierzy **X** i odpowiadającemu mu elementowi macierzy **Y** sprawdzamy warunek napisany w poprzednim punkcie. Używamy funkcji **sqrt** (od angielskiego *square root*) obliczającej pierwiastek kwadratowy. Potrafi ona działać na całych tablicach, a nawet macierzach. Aby wykonać operację podniesienia do potęgi '^', dla każdego elementu macierzy po kolei musimy użyć operatora '**.**'.

5. Napisana przez nas instrukcja jest zdaniem logicznym. W wyniku otrzymujemy macierz zer i jedynek - wartości zdania logicznego dla każdego punktu o współrzędnych (x, y). Zero oznacza zdanie fałszywe - punkt nie leży w kole, jedynka prawdziwe.

6. Liczymy punkty leżące w kole. Wystraczy tylko policzyć ile jest jedynek w wynikowej macierzy. Służy do tego funkcja **nnz** (od angielskiego *number of non zeros*):

   ```
   >> p = nnz(sqrt( X.^2 + Y.^2) <= r);
   ```

7. Sprawdźmy, ile wynosi przybliżone pole koła.

```
>> p
```

8. Obliczamy przybliżoną wartość π zgodnie ze wzorem $\pi \approx \frac{p}{r^2}$.

```
>> moje_pi = p / (r^2)
```

2.2 Kolejne przybliżenia

1. Bazując na wykonanych już operacjach spróbujmy poprawić przybliżenie naszych obliczeń. Zapiszmy je w pliku **przybliz_pi.m**. Stworzymy teraz funkcję, za pomocą której obliczymy kolejne przybliżenia ($r = 3, 4, ...$) π według naszego schematu. Funkcja będzie nazywać się **przybliz_pi**, przyjmować jeden argument r i zwracać liczbę zapisaną w zmiennej **moje_pi**.

```
function moje_pi = przybliz_pi(r)
```

2. Musimy teraz zmodyfikować nasz algorytm aby działał dla dowolnych wartości **r**. W tym celu usuńmy linię:

```
r = 2;
```

oraz zmodyfikujmy definicje macierzy **X** i **Y**:

```
X = ones(2*r + 1);
Y = ones(2*r + 1);
for i = 1 : 2*r + 1
   X(i, :) = x;
   Y(:, i) = y;
end
```

Zakończmy wszystkie linie, poza pętlą, średnikiem aby funkcja nie wypisywała niepotrzebnie wartości zmiennych oraz zakończmy kod funkcji słowem kluczowym **end**.

2.3 Jak dobrze umiemy przybliżać?

1. Sprawdźmy teraz jak dobrze przybliża nasza funkcja. Wypiszemy wartości kolejnych przybliżeń policzonych za pomocą naszej funkcji oraz błąd przybliżenia równy:

$$|\mathbf{moje\_pi} - \pi|$$

2. Utwórzmy tablice **r**, w którym zapiszemy wartości r, dla których chcemy wykonać obliczenia.

```
r = [2 3 4 5 10 20 100 1000];
```

3. Wykonajmy teraz w pętli naszą funkcję dla kolejnych wartości r.

```
for k = 1 : size(r,2)
    disp( [k, przybliz_pi(k), abs(przybliz_pi(k)-pi)] )
end
```

4. Zastanów się jak narysować dokładność (czyli wielkość błędu) kolejnych przybliżeń na wykresie.