Znajdywanie pierwiastków nieliniowych układów równań z pomocą pakietu SciPy

Bartłomiej Brzozowiec

Spis treści

1 Cel pracy

Celem pracy było poznanie funkcji `fsolve()` z pakietu *SciPy*[1], a także napisanie programu wykorzystującego tą funkcję. Preferowanym interfejsem użytkownika programu był interfejs graficzny.

Funkcja `fsolve()` służy do rozwiązywania układów dowolnej liczby równań (w ogólności) nieliniowych.

2 Wstęp teoretyczny

2.1 Sformułowanie zagadnienia

W przypadku rozwiązywania pojedynczego równania możemy przenieść wszystkie wyrażenia na lewą stronę równania, otrzymując

$$f(x) = 0 \tag{1}$$

i następnie znajdując wszystkie wartości x (*pierwiastki* równania), dla których równanie jest prawdziwe. Kiedy występuje tylko jedna niezależna zmienna, problem można określić jako *jednowymiarowy*.

Kiedy mamy więcej niż jedną zmienną, potrzebujemy więcej niż jednego równania do znalezienia wartości spełniających równocześnie wszystkie równania. W przypadku N zmiennych potrzebujemy N równań. W ogólności układ równań może mieć jedno konkretne rozwiązanie (tzn. N pierwiastków), więcej niż jedno rozwiązanie, nieskończenie wiele rozwiązań lub rozwiązanie może w ogóle nie istnieć.

Układ równań możemy zatem opisać stosując notację wektorową, chcąc znaleźć jeden lub więcej N-wymiarowy wektor rozwiązań $\mathbf{x}$:

$$\mathbf{F}(\mathbf{x}) = \mathbf{0} \tag{2}$$

gdzie $\mathbf{F}$ oznacza N równań, które wszystkie równocześnie mają być spełnione.

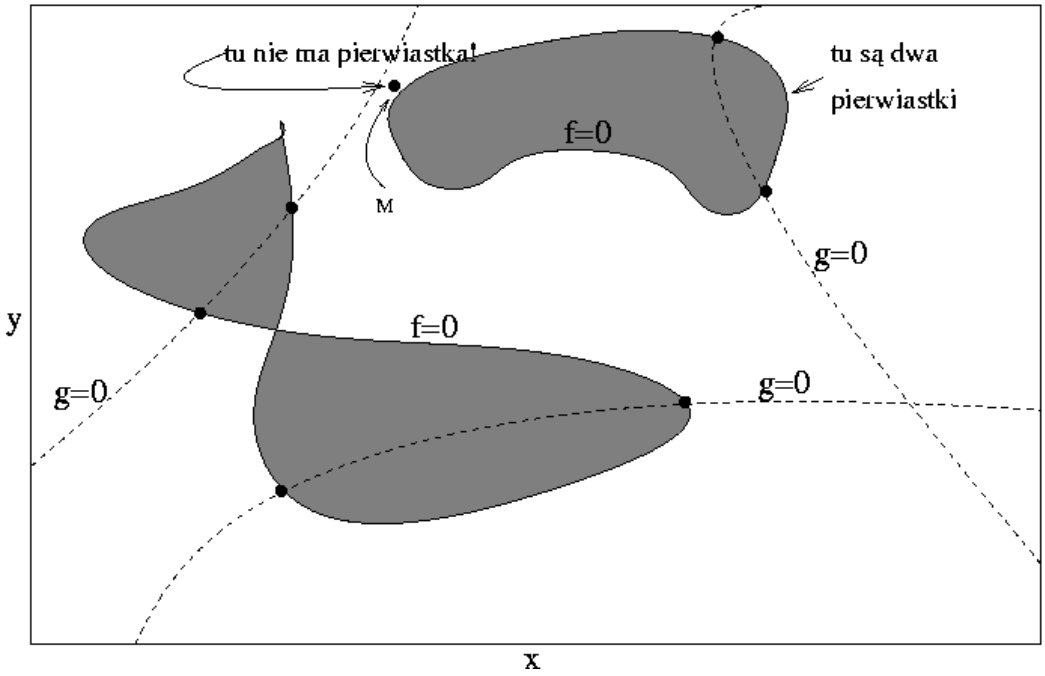
Nie należy się sugerować notacyjnym podobieństwem wzorów (1) i (2). Znalezienie rozwiązań N układów równań jest w ogólności *znacznie* trudniejsze niż rozwiązanie równania w przypadku jednowymiarowym.

[1] http://www.scipy.org

2.2 Numeryczne rozwiązywanie układów równań nieliniowych – metoda Newtona-Raphsona

Śmiało można stwierdzić, że *nie istnieją* dobre ogólne metody rozwiązywania układów równań nielinowych [1]. Co więcej, nietrudno jest zauważyć, dlaczego (prawdopodobnie) nie będą *nigdy* istniały dobre ogólne metody:
Rozważmy przypadek dwuwymiarowy, gdzie jednocześnie chcemy rozwiązać

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0. \end{aligned}$$



Rysunek 1: Graficzne przedstawienie rozwiązania dwóch nieliniowych równań z dwoma niewiadomymi. Linią ciągłą jest oznaczona funkcja $f(x,y)$, a przerywaną $g(x,y)$. Szukane rozwiązania są na przecięciu tych, niezależnych od siebie, krzywych. Ich liczba jest *a priori* nieznana. (Ilustracja na podst. [1].)

Funkcje f i g to dwie dowolne funkcje, które dzielą płaszczyznę na obszary, gdzie wartość odpowiednich funkcji jest dodatnia bądź ujemna. Nas interesują granice tych obszarów, gdzie funkcje przyjmują zero, czyli krzywe oznaczone linią przerywaną i ciągłą. Szukanymi przez nas rozwiązaniami (o ile istnieją) są punkty przecięcia się tych krzywych. Niestety, funkcje f i g nie mają, w ogólności, ze sobą nic wspólnego! Mówiąc inaczej, nie ma niczego specjalnego we wspólnym punkcie ani „z punktu widzenia" funkcji

f, ani funkcji g. W przypadku metod numerycznych szczególnie problematyczny może też być przypadek punktu oznaczonego na rysunku (1) jako M. Wykresy funkcji „zbliżają się" w nim do siebie, ale mimo wszystko nie jest to szukane miejsce zerowe.

Z powodu powyższych problemów numeryczne szukanie pierwiastków jest praktycznie niemożliwe bez wcześniejszego przeanalizowania danego zagadnienia do rozwiązania. Prawie zawsze należy użyć jakichś dodatkowych informacji, specyficznych dla danego problemu, i odpowiedzieć na podstawowe pytania w rodzaju „czy spodziewamy się pojedynczego rozwiązania?" albo „gdzie w przybliżeniu znajduje się rozwiązanie?".

Jedną z metod numerycznego rozwiązywania układów równań jest metoda Newtona-Raphsona [1], która bardzo dobrze zbiega do szukanych pierwiastków, jeśli podamy wystarczająco dobre początkowe przybliżenie. Metoda moża zasygnalizować także brak powodzenia, co może oznaczać, że nasz domniemany pierwiastek nie istnieje w pobliżu podanego przybliżenia.

Typowy problem daje N funkcji do wyzerowania, które mają N zmiennych:

$$F_i(x_1, x_2, \ldots, x_N) = 0 \qquad i = 1, 2, \ldots, N. \tag{3}$$

Oznaczmy przez $\mathbf{x}$ cały wektor wartości x_i oraz przez $\mathbf{F}$ cały wektor funkcji F_i. W otoczeniu $\mathbf{x}$ każda z funkcji F_i może być rozwinięta w szereg Taylora:

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2). \tag{4}$$

Macierz pochodnych cząstkowych, występująca w równaniu (4), to *macierz Jacobiego* $\mathbf{J}$ (zwana także czasem w literaturze *jakobianem*):

$$J_{ij} \equiv \frac{\partial F_i}{\partial x_j}. \tag{5}$$

W notacji macierzowej równanie (4) można zapisać jako

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2). \tag{6}$$

Przy zaniedbaniu $O(\delta\mathbf{x}^2)$ i po przyrównaniu $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$ otrzymujemy układ równań liniowych, z którego możemy obliczyć wektor poprawek $\delta\mathbf{x}$, które przybliżają równocześnie każdą funkcję do zera, a mianowicie

$$\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{F} \tag{7}$$

Równanie macierzowe (7) można numerycznie rozwiązać metodą rozkładu LU. Poprawki są następnie dodawane do wektora rozwiązań

$$\mathbf{x}_{nowe} = \mathbf{x}_{stare} + \delta\mathbf{x} \tag{8}$$

i procedura jest powtarzana aż do uzbieżnienia wyniku.

3 Implementacja

3.1 Użyte oprogramowanie i biblioteki

Program będący częścią niniejszej pracy jest napisany w języku skryptowym *python*[2]. Korzysta on z bibliotek *PyQt*[3] (w celu dostarczenia estetycznego i wygodnego graficznego interfejsu użytkownika) oraz *SciPy*[4], która dostarcza programiście szereg funkcji z dziedziny metod numerycznych i obliczeń naukowo-inżynierskich.

W szczególności, w pracy nad pisaniem oraz testowaiem programu zostały użyte następujące wersje oprogramowania (może być to istotna informacja w przypadku dostosowywania programu w przyszłości, kiedy API[5] może ulec zmianie):

- *python* - 2.4.4
- *Qt* - 3.3.7 (*PyQt* to tak naprawdę zbiór dowiązań do biblioteki *Qt*, która jest napisana w C++)
- *PyQt* - 3.17 (*sip* - 4.5.2)
- *SciPy* - 0.5.2 (*NumPy* - 1.0.1)

3.2 Funkcja *fsolve()* pakietu *SciPy*

Do znajdywania pierwiastków wielomianów należy użyć funkcji `root()`. Funkcja `fsolve()`[6] służy do znajdywania pierwiastków układów równań nielinowych [2]. Przykładowa sesja w trybie interaktywnym *pythona* (tutaj z użyciem *ipythona*[7]) może wyglądać następująco:
Najpierw należy wczytać odpowiednie moduły (a właściwie zaimportować odpowiednie funkcje do aktualnej przestrzeni nazw):

```
In [1]: from scipy import *

In [2]: from scipy.optimize import fsolve
```

[2] http://www.python.org
[3] http://www.riverbankcomputing.co.uk/pyqt/
[4] http://www.scipy.org
[5] Application Programming Interface
[6] Jeśli zajrzymy do źródeł *SciPy*, zauważymy, że funkcja *fsolve()* wywołuje tak naprawdę kod napisany w Fortranie. W plikach źródłowych znajdziemy komentarz, że użyta metoda to „zmodyfikowana metoda hybrydowa Powella" (w oryginale: „modification of the powell hybrid method"), a kod jest częścią projektu Minpack i pochodzi z marca 1980 roku.
[7] http://ipython.scipy.org

Funkcja `fsolve()` wymaga (co najmniej) podania funkcji (lub układu funkcji) do rozwiązania oraz punktu startowego (przybliżonego rozwiązania). Przyjmijmy następujące przykłady:

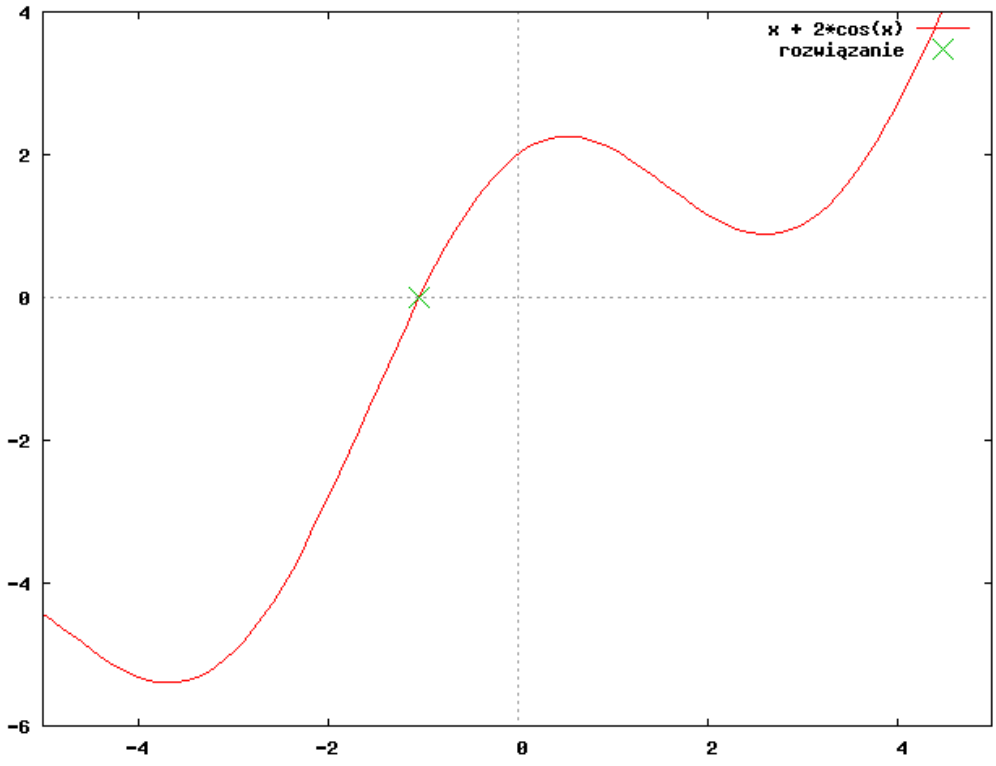
- przykład pojedynczego równania:

$$x + 2cos(x) = 0,$$

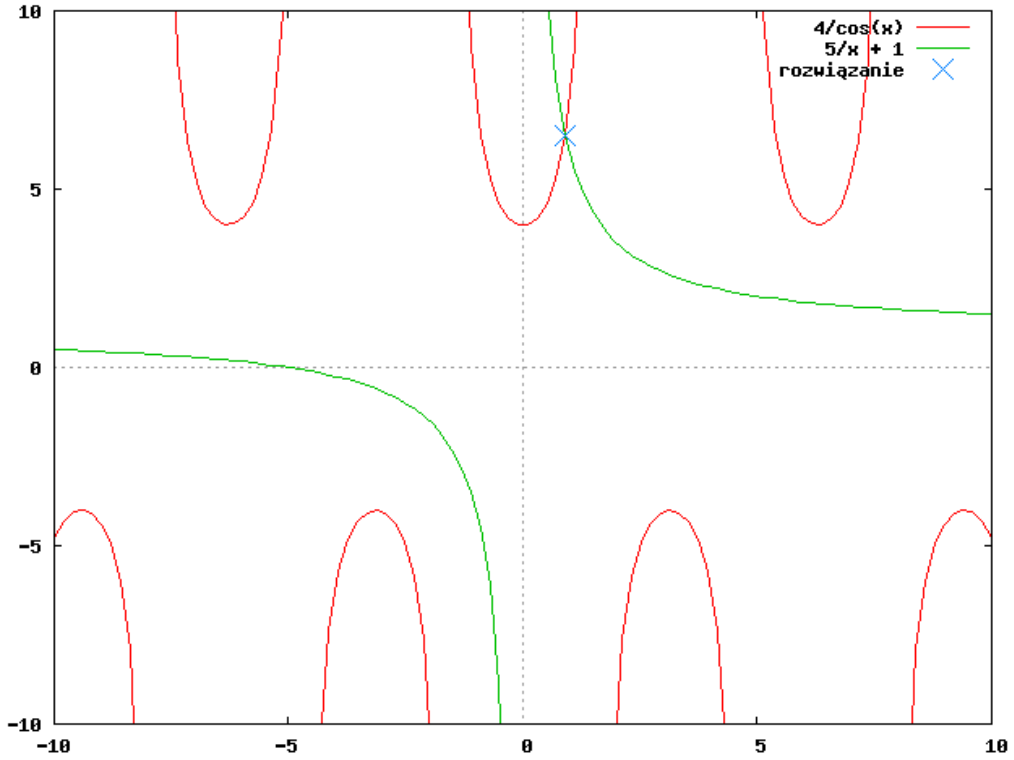
- przykład układu dwóch równań:

$$\begin{aligned} x_1 cos(x_2) &= 4 \\ x_1 x_2 - x_2 &= 5. \end{aligned}$$

Rozwiązanie graficzne obu przykładów przedstawiają, odpowiednio, wykresy (2) i (3).



Rysunek 2: Graficzne rozwiązanie pojedynczego równania $x + 2cos(x) = 0$



Rysunek 3: Graficzne rozwiązanie układu równań: $x_1 cos(x_2) = 4$ oraz $x_1 x_2 - x_2 = 5$.

Za punkty startowe przyjmijmy, odpowiednio, 0.3 oraz $(1; 1)$.

Najpierw musimy zdefiniować funkcje, które podamy jako argument dla funkcji `fsolve()`:

```
In [3]: def f(x):
   ...:     return x + 2*cos(x)
   ...:

In [4]: def f2(x):
   ...:     out = [x[0]*cos(x[1]) - 4]
   ...:     out.append(x[1]*x[0] - x[1] - 5)
   ...:     return out
   ...:
```

a następnie możemy poszukać rozwiązania i je wypisać:

```
In [5]: x0 = fsolve(f, 0.3)
```

```
In [6]: print x0
-1.02986652932

In [7]: x02 = fsolve(f2, [1, 1])

In [8]: print x02
[ 6.50409711  0.90841421]
```

Jak widać na przykładzie, drugim argumentem funkcji `fsolve()` jest punkt startowy.

Oprócz funkcji do rozwiązania oraz punktu startowego można podać (opcjonalnie) także szerego innych parametrów. W celu uzyskania szczegółowej pomocy można wpisać

```
In [9]: help(fsolve)

Help on function fsolve in module scipy.optimize.minpack:

fsolve(func, x0, args=(), fprime=None, full_output=0, col_deriv=0,
           xtol=1.49012e-08, maxfev=0, band=None, epsfcn=0.0,
           factor=100, diag=None)
    Find the roots of a function.

    Description:

      Return the roots of the (non-linear) equations defined by
      func(x)=0 given a starting estimate.

    Inputs:
    (...)
```

(wyjście zostało przycięte)

W szczególności, argument `fprime` może zawierać odwołania do funkcji zwracających wartość jakobianu. Jeśli argument ten jest pominięty, wartość jakobianu jest szacowana numerycznie.

Funkcja `fsolve()` może także *zwracać* więcej informacji niż tylko wynik. Żeby uzyskać takie zachowanie, należy dodać argument `full_output` ustawiony na niezerową wartość, np.:

```
In [10]: x0, infodict, ier, mesg = fsolve(f2, [1, 1], full_output=1)
```

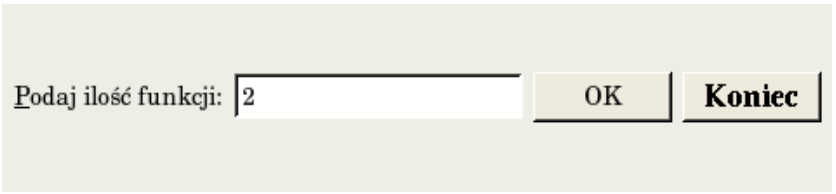
W przypadku niepowodzenia wartość `ier` jest ustawiona na wartość różną od 1, a zmienna `mesg` zawiera czytelny komunikat (po angielsku) mówiący o przyczynie porażki.

3.3 Aplikacja GUI

Stworzenie aplikacji z graficznym interfejsem użytkownika (GUI), wykorzystującej *pythona* i bibliotekę *SciPy* było istotnym elementem pracy. Poniżej została przedstawiona dokumentacja programu (*praca.py*).

3.3.1 Interfejs użytkownika

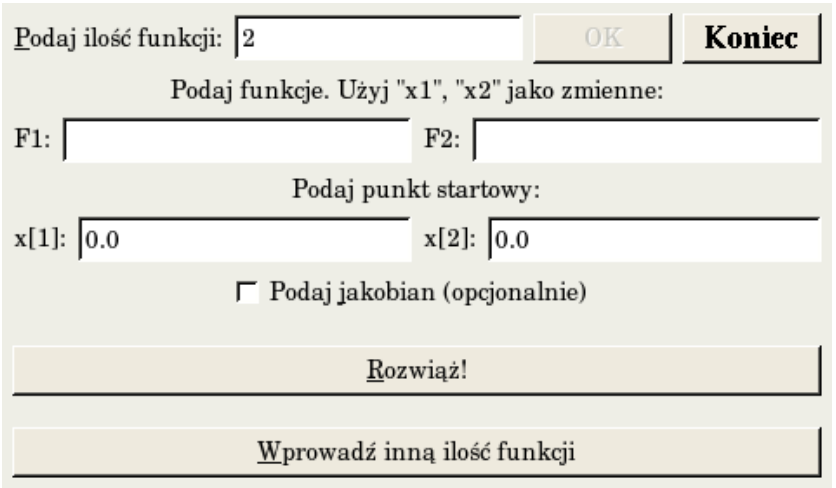
Po uruchomieniu programu na ekranie pojawi się następujące okienko:



Rysunek 4: Początkowy ekran programu

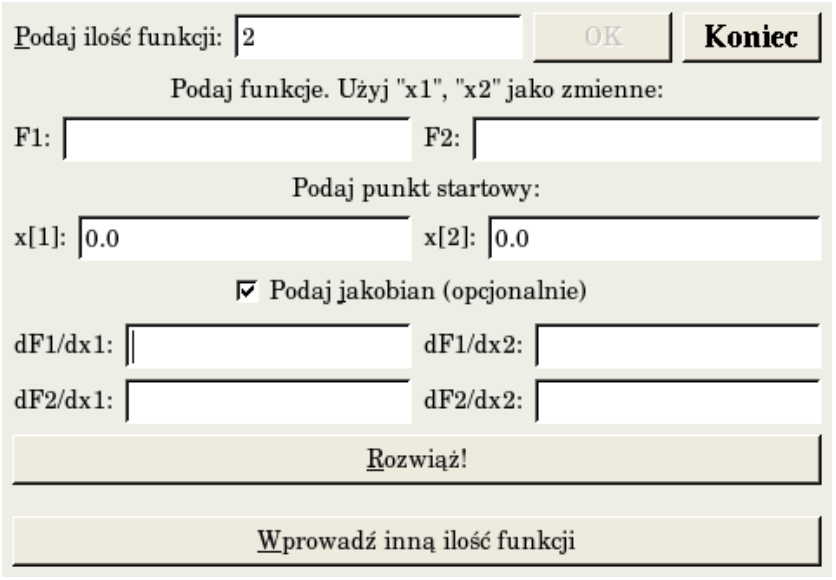
Program obsługuje układ równań o dowolnej ilości funkcji. Należy podać, ile będzie równań w układzie.

Po wybraniu *OK* ujrzymy:



Rysunek 5: Ekran programu - wprowadzanie danych

Należy teraz podać dane wejściowe, tzn. funkcje w układzie równań (przyrównane do zera) oraz punkt startowy. Po zaznaczeniu checkboxa „Podaj jakobian (opcjonalnie)” można podać analitycznie znalezione wartości jakobianu, tzn. pochodne cząstkowe po wszystkich zmiennych:



Rysunek 6: Ekran programu - wprowadzanie danych z jakobianem

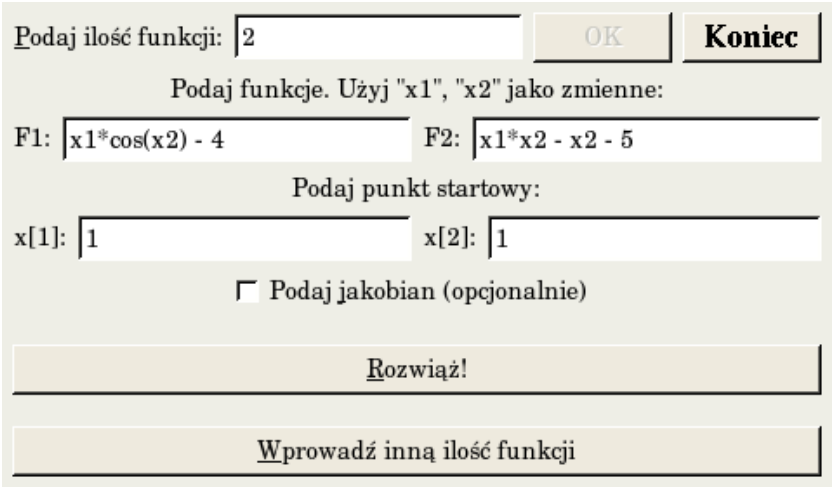
Interesującym (z programistycznego punktu widzenia) aspektem programu jest fakt, że większość widgetów (elementów interfejsu graficznego użytkownika, takich jak etykiety i pola tekstowe) jest tworzona dynamicznie, gdyż ich ilość nie jest z góry znana (użytkownik sam określa ilość równań).

3.3.2 Przykład działania

Jako przykład możemy wprowadzić układ równań rozważany w rodziale „Funkcja *fsolve()* pakietu *SciPy*", czyli:

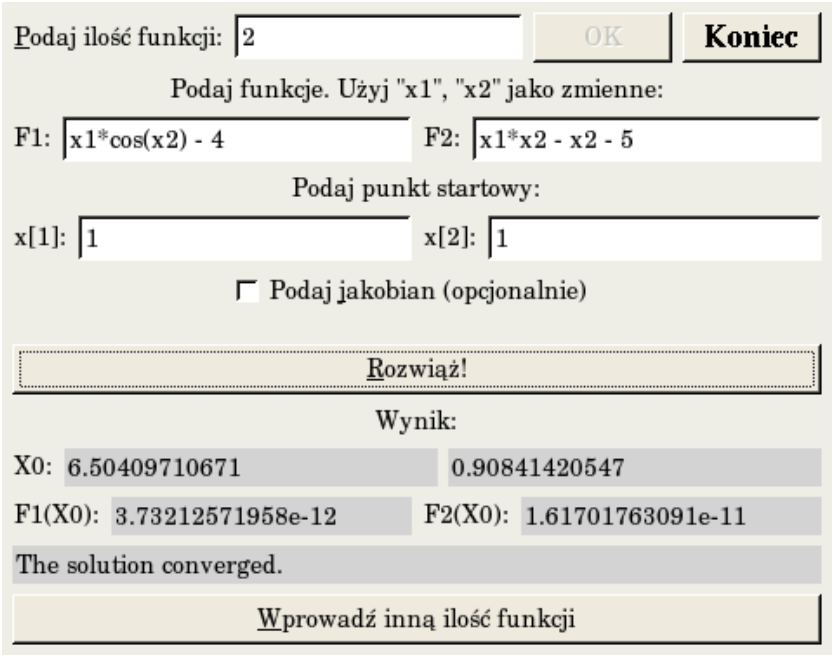
$$\begin{aligned} x_1 cos(x_2) - 4 &= 0 \\ x_1 x_2 - x_2 - 5 &= 0. \end{aligned}$$

oraz punkt startowy $(1; 1)$:



Rysunek 7: Ekran programu - przykładowe wprowadzanie danych

Pomiędzy polami tekstowymi możemy się szybko przemieszczać z użyciem klawiatury za pomocą klawisza *Tab* oraz *Shift + Tab*. Po wprowadzeniu danych możemy wybrać przycisk „Rozwiąż!" w celu wywołania funkcji `fsolve()` i uzyskania wyniku:



Rysunek 8: Ekran programu - przykładowy wynik

W polach na szarym tle znajduje się wynik. W linii *X0* znajduje się znaleziony pierwiastek, a niżej wartości funkcji w znalezionym punkcie (idealnie powinny być równe zero). Na samym dole widzimy komunikat metody, który może być pomocny w przypadku niepowodzenia funkcji `fsolve()`.

Jeżeli chcemy wprowadzić inną ilość funkcji (np. sprawdzić teraz przypadek jednowymiarowy), możemy wybrać przycisk „Wprowadź inną ilość funkcji". W przeciwnym wypadku możemy zmienić dane wejściowe i ponownie wybrać „Rozwiąż!". Przycisk „Koniec" kończy w dowolnym momencie działanie programu (można także nacisnąć klawisz *Escape* na klawiaturze w celu uzyskania tego samego rezultatu).

4 Wnioski

Python wraz z biblioteką *SciPy* to potężne narzędzie, mogące mieć zastosowanie w pracy naukowców i inżynierów, porównywalne[8] nawet z komercyjnymi pakietami w rodzaju Matlaba. Możliwość korzystania z szerokiej gamy bibliotek dostępnych z poziomu *pythona* pozwala na tworzenie bardzo zróżnicowanych aplikacji. Na szczególną uwagę zasługuje biblioteka *PyQt* umożliwiająca szybkie tworzenie zaawansowanych, estetycznych i łatwych w obsłudze aplikacji graficznych.

Stworzony program z graficznym interfejsem użytkownika pozwala znajdować pierwiastki układów równań nieliniowych, a zatem spełnia założenia postawione na początku.

[8] Szczególnie w połączeniu z pakietem Matplotlib (http://matplotlib.sourceforge.net), którego omówienie wykracza poza zakres tej pracy.

5 Bibliografia

Literatura

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: *Numerical Recipes in C. The Art of Scientific Computing. Second Edition*, Cambridge University Press, 1992 (w szczególności rozdział 9.: *Root Finding and Nonlinear Sets of Equations*)

[2] Travis E. Oliphant: *SciPy Tutorial*[9]

[3] Boudewijn Rempt: *GUI Programming with Python: QT Edition*, 2001[10]

6 Lista ilustracji

Spis rysunków

[9] http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf

[10] http://www.commandprompt.com/community/pyqt/