**Лабораторная работа № 1-2**

**Исследование функций**

Решение уравнений, нахождение максимума или минимума функции одной или нескольких переменных осуществляются вызовом специальных функций MATLAB. Число аргументов этих функций может быть различным, в зависимости от требуемого вида результата. Для работы с ними, как правило, необходимо запрограммировать исследуемую функцию, например, в виде файл-функции. При этом можно обращаться к файл-функции либо по имени файла, либо по ссылке на нее.

Если исследуемая функция задается достаточно простой и короткой формулой, то не обязательно составлять файл-функцию. Вместо этого удобно ввести встраиваемую функцию (inline-функцию), воспользовавшись функцией inline, или определить анонимную функцию.

**Встраиваемые и анонимные функции**

Встраиваемая функция определяется при помощи функции inline, обращение к которой выглядит следующим образом:

Имя\_функции = inline('формула', список\_аргументов)

Список аргументов не обязателен, а 'формула' является текстовой строкой и задает выражение для вычисления значения функции.

Следующий пример демонстрирует создание в рабочей среде встраиваемой функции fun:

>> fun = inline('sin(x) - x.^2.\*cos(x)')

fun =

Inline function:

fun(x) = sin(x) - x.^2.\*cos(x)

Inline-фунция fun может быть использована как любая другая функция

MATLAB, например:

>> y=fun(0.5)

y =

0.2600

Если функция зависит от нескольких переменных, то все они являются аргументами введенной inline-функции и располагаются в алфавитном порядке:

>> fun1 = inline('sin(a\*x) - x.^2.\*cos(b\*x)')

fun1 =

Inline function:

fun1(a,b,x) = sin(a\*x) - x.^2.\*cos(b\*x)

Для изменения порядка аргументов их следует перечислить через запятые в списке после выражения, определяющего вид функции:

>> fun2 = inline('sin(a\*x) - x.^2.\*cos(b\*x)', 'x', 'a', 'b')

fun2 =

Inline function:

fun2(x,a,b) = sin(a\*x) - x.^2.\*cos(b\*x)

Если в списке случайно пропущен хотя бы один из аргументов, то inline-функцией воспользоваться не удастся:

>> fun3 = inline('sin(a\*x) - х.^2.\*cos(b\*x)', 'х', 'b')

fun3 =

Inline function:

fun3(х,b) = sin(a\*x) - х.^2.\*cos(b\*x)

Даже при наличии переменной *a* в рабочей среде вызов функции fun3 при ведет к сообщению о том, что аргумент *a* не задан:

>> a=1;

>> fun3(5,0)

??? Error: The expression to the left of the equals sign is not a valid target for an assignment.

Error in ==> inlineeval at 11

eval(INLINE\_INPUTEXPR\_);

Error in ==> inline.subsref at 25

INLINE\_OUT\_ = inlineeval(INLINE\_INPUTS\_, INLINE\_OBJ\_.inputExpr, INLINE\_OBJ\_.expr);

Этот пример демонстрирует, что при вычислении значения встраиваемой функции переменные рабочей среды недоступны. Все аргументы функции inline должны быть символьными строками, заключенными в апострофы, или строковыми переменными. В противном случае получается недопустимая конструкция. Работа со строковыми переменными описана далее в книге, а при чтении этой главы достаточно придерживаться простого правила – ставить апострофы в аргументах функции inline.

Альтернативный способ задания исследуемой функции состоит в объявлении анонимной функции с помощью оператора указателя (@):

Имя\_функции = @(список\_аргументов) формула

В отличие от inline-функции, и аргументы, и формула записываются в обычном виде, а не как текстовые строки в апострофах. Кроме того, анонимной функции доступны переменные рабочей среды, которые входят в формулу. Однако они являются константами, в качестве которых берутся значения этих переменных в момент создания анонимной функции, и последующее изменение их значений не будет учитываться при вычислении функции:

>> a=1;

>> gun3 = @(x, b) (sin(a\*x) - x.^2.\*cos(b\*x))

gun3 =

@(x, b) (sin(a\*x) - x.^2.\*cos(b\*x))

>> gun3(5,0)

ans =

-25.9589

>> a=1000;

>> gun3(5,0)

ans =

-25.9589

По способу использования анонимная функция напоминает inline-функцию, но отличается тем, что создается указатель на функцию, который связан с исполняемым кодом. Это хорошо видно либо в окне Workspace, либо при выводе информации о функциях с помощью whos:

whos gun3

Name Size Bytes Class Attributes

gun3 1x1 16 function\_handle

>> whos fun3

Name Size Bytes Class Attributes

fun3 1x1 914 inline

Информация о выделенной под функции памяти показывает, что для анонимной функции исполняемый код и указатель на нее отделены, а для inline-функции это единый объект.

Вышеописанные способы задания исследуемых математических функций (по имени М-файла, по ссылке, анонимная функция или inline-функция) допустимы во всех вычислительных алгоритмах MATLAB. Для достаточно простых исследуемых функций удобнее всего определить соответствующую inline или анонимную функцию. В то же время, если функция задана громоздким выражением, лучше всего написать соответствующий М-файл. А в случае, когда функция вычисляется по сложному алгоритму, это оказывается просто необходимым.

**Решение уравнений**

Для нахождения корней произвольных уравнений служит функция fzero, a для определения всех корней полиномов применяется roots.

**Решение произвольных уравнений**

Функция fzero позволяет приближенно вычислить корень уравнения на некотором интервале или ближайший к заданному начальному приближению. В простейшем варианте fzero вызывается с двумя входными и одним выходным аргументом х = fzero('func\_name', х0), где func\_name – имя файл-функции, вычисляющей левую часть уравнения, *х*0 – начальное приближение к корню, *х* – найденное приближенное значение корня. Решите, например, на отрезке [-5, 5] уравнение:

|  |  |  |
| --- | --- | --- |
|  | sin x - x2 cos x = 0. | (6.1) |

Перед нахождением корней полезно построить график функции, входящей в левую часть уравнения. Для получения графика можно прибегнуть к plot, но все равно понадобится запрограммировать функцию, поэтому имеет смысл воспользоваться fplot, которая к тому же позволяет получить более точный график по сравнению с plot.

В листинге 6.1 приведен текст требуемой файл-функции.

function y = myf(x)

y = sin(x) - x.^2.\*cos(x);

Теперь постройте график функции myf, используя fplot, и нанесите сетку.

>> fplot('myf', [-5 5])

>> grid on

>>



Рис. 6.1. График левой части уравнения (6.1)

Из графика функции, изображенного на рис. 6.1, видно, что на этом отрезке имеются четыре корня. Один корень равен нулю, в чем несложно убедиться, подставив *х* = 0 в уравнение (6.1).

Уточните значение корня, расположенного вблизи *х* = -5, при помощи fzero:

>> xl = fzero('myf', -5)

xl =

-4.7566

Итак, приближенное значение корня равно -4.7566. При указании начального приближения к корню алгоритм fzero автоматически отделяет корень, т. е. вблизи заданного начального приближения находится отрезок, содержащий корень. В этом случае fzero может использовать больший интервал определения функции, чем исходный отрезок [-5, 5]. Проверьте ответ, вычислив значение функции myf в точке *x*1

>> myf(xl)

ans =

2.6645e-015

Конечно, то, что значение функции близко к нулю, вообще говоря, не означает достаточную точность найденного корня. Гарантированная точность приближенного значения определяется расстоянием до его истинного значения или (что фактически тоже самое) количеством верных значащих цифр.

Заданию точности вычислений посвящен далее разд. "Управление ходом вычислений". Для того чтобы увидеть больше значащих цифр корня *x*l, следует установить формат long и вывести xl еще раз (точность проводимых пакетом MATLAB расчетов не зависит от формата вывода результата!).

>> format long

>> x1

x1 =

-4.756559405702904

Возникает вопрос, сколько в ответе точных значащих цифр, т. е. *с какой точностью* найдено решение. На компьютере вычисления производятся с числами, имеющими 52 двоичных разряда в мантиссе (без учета порядка числа). Это соответствует относительной погрешности представления чисел, которую возвращает функция eps, вызываемая без входных аргументов:

>> eps

ans =

2.220446049250313e-016

Алгоритм функции fzero по умолчанию находит корень уравнения с точностью eps, т. е. ± 2 в шестнадцатом знаке после десятичной точки – практически с максимально возможной точностью. Чтобы убедиться в этом, измените формат представления данных на long e и выведете *x*1:

>> format long e

>> x1

x1 =

-4.756559405702904e+000

Число значащих цифр увеличилось на одну (по сравнению с форматом long). Это самое точное представление числа в десятичной форме для пакета MATLAB. Измените последнюю цифру 4 на 5 и вычислите значение myf в точке *x*l:

>> x1=-4.756559405702905e+000

x1 =

-4.756559405702905e+000

>> myf(x1)

ans =

-1.765254609153999e-014

Поскольку исследуемая функция сменила знак, то между этими значениями лежит искомый корень уравнения. Следовательно, ошибка в найденном корне в последней значащей цифре.

Численное решение любой задачи требует дополнительных расчетов, подтверждающих (как правило, только косвенно) правильность полученных результатов. Для задачи нахождения корня уравнения вы только что проделали такой численный эксперимент.

Проверьте работу fzero, вычислив корень myf, расположенный вблизи нуля, там, где точное значение корня равно нулю.

>> x4 = fzero('myf', -0.1)

x4 =

-1.242386505963434e-022

Функция fzero действительно гарантирует, что точность решения не меньше eps.

Найдите самостоятельно корни *х*2 и *х*3, расположенные около точек –2 и –5.

Вместо начального приближения вторым параметром fzero можно задать интервал, на котором следует найти корень:

>> x2 = fzero('myf',[-3 -1])

x2 =

-1.853927459696150e+000

>>

На границах указываемого интервала функция должна принимать значения разных знаков, иначе выведется сообщение об ошибке!

В качестве исследуемой функции может выступать и встроенная математическая функция, например

>> fzero('sin', [2 4])

ans =

3.141592653589793e+000

Допустимы другие способы вызова fzero. Во-первых, функцию с исследуемой математической функцией можно задать при помощи указателя на нее:

>> format short

>> x2 = fzero(@myf,[-3 -1])

x2 =

-1.8539

Во-вторых, воспользовавшись функцией inline:

>> fun = inline('sin(x) - x.^2.\*cos(x)')

fun =

Inline function:

fun(x) = sin(x) - x.^2.\*cos(x)

>> x1=fzero(fun,-5)

x1 =

-4.7566

>> fun(x1)

ans =

2.6645e-015

В-третьих, создав анонимную функцию:

>> fun = @(x) sin(x) - x.^2.\*cos(x)

fun =

@(x) sin(x) - x.^2.\*cos(x)

>> x1=fzero(fun,-5)

x1 =

-4.7566

Обращение к fzero с двумя выходными аргументами позволяет не только приближенно найти корень, но и получить значение функции в найденной точке.

>> [x2, f] = fzero(@myf, -2)

x2 =

-1.8539

f =

-2.2204e-016

Важной особенностью fzero является то, что она вычисляет только те корни, в которых функция меняет знак, а не касается оси абсцисс. Найти корень уравнения *х*2 = 0 при помощи fzero не удается:

>> fun=inline('х.^2');

>> x = fzero(fun, -0.1)

??? Error using ==> fzero

FZERO cannot continue because user supplied inline object ==> fun failed with the error below.

Error: The expression to the left of the equals sign is not a valid target for an assignment.

В данном примере fzero пыталась найти промежуток, на границах которого значения функции myf имеют различные знаки, что гарантировало бы существование корня на этом промежутке. Такой промежуток, естественно, определить не удалось, и fzero вывела сообщение об ошибке в командное окно.

Программирование собственных приложений с использованием вычислительных функций требует получения сведений о завершении вычислительного процесса для перехода к соответствующему блоку алгоритма. Обращение к fzero с тремя выходными аргументами

>> [xl, fl, flag] = fzero(fun, 0.1);

позволяет выбрать дальнейшие действия в зависимости от содержимого переменной flag. Положительное значение flag свидетельствует об успешном завершении вычислительного процесса. Отрицательное значение говорит о том, что либо не удалось отделить интервал со сменой знака функции на границах, либо в процессе вычислений получилось комплексное значение, бесконечность, или выполнена операция с неопределенным результатом, например, деление нуля на ноль.

Функция fzero, так же как и большинство вычислительных функций MATLAB, допускает управление многими параметрами заложенных в ней алгоритмов. Кроме того, для контроля выполнения алгоритма и изменения его параметров следует получить информацию о ходе вычислений. Управление алгоритмом и контроль за ним требуют специальных способов вызова функций MATLAB (см. разд. "Управление ходом вычислений").

**Самостоятельная работа**

Найти корни уравнения:

1. *y* = 2*x*4 – 8*x*3 + 8*x*2 – 1,
2. y = *x*3 + 0,4*x*2 – 0,6*x* – 1,6,
3. *y* = *x*5 – *x*3 + 1,
4. 
5. *у* = *хе~х* + sin *x*,
6. *ex/*5 *–* 2(*x –* 1)2 *=* 0*.*

**Решение систем нелинейных уравнений**

Функция fsolve, вообще говоря, предназначена для решения систем нелинейных уравнений вида *F*(*x*) = 0, где *х* – вектор или матрица неизвестных, а *F* – функция, значением которой является вектор или матрица. Алгоритм ее работы использует начальное значение *х*0 и базируется на минимизации суммы квадратов составляющих функции *F* методами Гаусса-Ньютона и Левенберга–Марквардта. В простейшем случае обращение к fsolve имеет вид:

х = fsolve(F,X0).

В частности, функцию fsolve можно использовать и как альтернативу функции fzero для нахождения корня единственного нелинейного уравнения (пример 15.7), например, sin(x) = 0.

Пример 15.7. Поиск корня с помощью функции fsolve

>> x=fsolve(@sin,1)

Optimization terminated: first-order optimality is less than options.TolFun.

x =

0

>>

Интересно, что второй аргумент функции fsolve может быть задан в виде вектора начальных значений, и для каждого компонента этого вектора будет найдено близлежащее решение (пример 15.8).

Пример 15.8, Поиск решений для вектора начальных значений

>> x=fsolve(@sin,[1 2 3 4 5 6],optimset('fsolve'))

Optimization terminated: first-order optimality is less than options.TolFun.

x =

0 3.1416 3.1416 3.1416 6.2832 6.2832

>>

В отличие от fzero функция fsolve может найти приближенное решение для упоминавшегося выше уравнения *х*2 = 0 точки разрыва функции tg(*x*) (пример 15.9).

Пример 15.9. Поиск решения *х*2 = 0 и tg (*x*) = 0

>> x=fsolve('x^2',1,optimset('Display','off'))

x =

0.0078

>>

>> x=fsolve(@tan,1,optimset('Display','off'))

x =

2.3205e-010

>>

Смысл третьего параметра в двух предыдущих обращениях заключается в подавлении "лишних" сообщений. Более подробно о дополнительных входных параметрах функции fsolve см. в разд. 15.6.

Главным назначением функции fsolve является решение систем нелинейных уравнений, что мы продемонстрируем на примере 15.10.

Пример 15.10. Решение системы нелинейных уравнений

*у*1 = *x*1 + *х*2 – sin *x*1,

*у*2 = *x*1 - *х*2 - cos *х*2.

Объявим функцию funsc, значения которой сформируем в виде вектора-столбца:

function y=funsc(x)

y=[x(1)+x(2)-sin(pi\*x(1));

x(1)-x(2)-cos(pi\*x(2))];

А теперь обратимся к функции fsolve, задавая каждый раз разные начальные значения (*x*l, *x*2). Так как fsolve подобно fzero умеет возвращать и вектор-столбец функций в найденной точке, то мы воспользуемся двумя выходными параметрами, чтобы оценить точность решения. Обратите внимание на то, что координаты начальной точки тоже представлены в виде вектора-столбца:

>> [x,f]=fsolve(@funsc,[0.2;1],optimset('Display','off'))

x =

0.3915

0.5510

f =

1.0e-010 \*

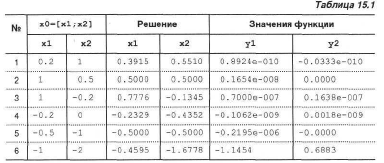
0.8924

-0.0330

>>

Результаты последующих обращений к функции fsolve из разных стартовых точек приведены в табл. 15.1.

Таблица 15.1



Из табл. 15.1 видно, что для первых пяти начальных точек было найдено пять разных решений, тогда как шестой эксперимент вместо корня обнаружил нечто похожее на точку локального минимума.

Разобраться со всеми ситуациям помогут графики наших функций. На рис. 15.2 представлены:

• графики двух функций: *x*2 = sin(*x*1) – *x*1 (сплошная линия) и *x*1 = cos (*x*2)+*x*2 (штриховая линия). Эти функции получены путем разрешения уравнений системы относительно соответствующих переменных;

• решения системы – крупные точки пересечения этих графиков и соответствующие стартовые точки (мелкие) с соединяющими их отрезками.

Для их построения была использована программа prog15\_l.m (пример 15.11).

function progl5\_l

axes('Xlim', [-1 1.5],'Ylim', [-1 1.5])

axis equal; grid off; hold on

x1=-1:0.1:1.5; y=sin(pi\*x1)-x1; plot(x1,y,'k-');

x2=-1:0.1:1.5; y=cos(pi\*x2)+x2; plot(y,x2,'k:');

xlabel('xl'); ylabel('x2');

title('Решение нелинейной системы','FontName','Courier');

x1x2=[0.2,1,1,0.5,1,-0.2,-0.2,0,-0.5,-1,-1,-2];

for j=1:6

x1=x1x2(2\*j-1);

x2=x1x2(2\*j);

line(x1,x2, 'Marker', '.', 'MarkerSize', 10);

x = fsolve(@funsc,[x1,x2],optimset('Display','off'));

line(x(1),x(2),'Marker','.','MarkerSize',20);

plot([x1 x(1)],[x2 x(2)],'k-');

end



Рис. 15.2. Графики функций

В выделенные пределы неудачная стартовая точка не попала, но если бы на полученные графики были нанесены линии уровня минимизируемой функции *y*12 + *y*22, то можно было бы обнаружить, что точка 6 действительно соответствует локальному минимуму.

Среди выходных аргументов функций fsolve могут присутствовать такие же по смыслу параметры, которые допускала функция fzero:

[х,fval,exitflag,output] = fsolve(fun,x0).

Значение признака exitflag = 1 свидетельствует о том, что решение системы найдено. При exitflag = 0 решение не найдено, т. к. итерационный процесс был прерван в связи с достижением максимального количества итераций или обращений к функции fun. При exitflag = -1 достигнутый минимум не является решением системы.

В структуре output появились дополнительные поля с именами:

• stepsize, в котором выдается величина шага при завершении поиска;

• firstorderopt, в котором выдается точность, достигнутая по значению функции.

Например, на одном из предыдущих запусков мы могли бы получить следующую информацию:

>> [x,f,e\_flag,inform] = ...

fsolve(@funsc,[1;0.5],optimset('Display','off'))

x =

0.5000

0.5000

f =

1.0e-008 \*

0.1654

-0.0000

e\_flag =

1

inform =

iterations: 6

funcCount: 21

algorithm: 'trust-region dogleg'

firstorderopt: 1.6538e-009

message: [1x76 char]

>>

Обратите внимание на то, что в отличие от функции fzero здесь число итераций не совпадает с количеством обращений к функции fun. Дело в том, что алгоритм поиска решения в функции fsolve активно использует якобиан системы уравнений, который в данном примере находится численным методом (аналог замены производных разностями). В функции fun может быть предусмотрена возможность задания операторов, вычисляющих якобиан по аналитическим формулам (пример 15.12). Это может существенно уменьшить количество обращений к функции fun, и, тем самым, ускорить итерационный процесс. Подключение якобиана предусматривает два момента. Во-первых, среди входных параметров при обращении к функции fsolve должно быть соответствующее указание типа optimset (' jacobian', 'on'). Во-вторых, операторы для вычисления якобиана должны входить в определение функции fun, которая в данном случае должна возвращать два выходных аргумента.

Пример 15.12. Вычисление якобиана по аналитическим формулам

function [F, J] = fun(x)

F = ... % вычисление вектора или матрицы системы

if nargout > 1 % два выходных значения

J = ... % вычисление якобиана как функции от х

end

В рассматриваемом примере якобиан представляет собой матрицу 2x2, образованную частными производными от *y*1 и *у*2 по *x*1 и *х*2:



Добавим к функции funsс формулы для вычисления якобиана:

function [y,j] = funscj(x)

y=[x(1)+x(2)-sin(pi\*x(1));

x(1)-x(2)-cos(pi\*x (2))];

if nargout>1

j=[1-pi\*cos(pi\*x(1)) 1;

1 -1+pi\*sin(pi\*x(2))];

end

Зададим *х*0=[1;0.5] и выясним, насколько помогает наличие формул вычисления якобиана:

>> [x,f,e\_flag,inform] = ...

fsolve(@funscj,[1;0.5],optimset('Jacobian','on'))

Optimization terminated: first-order optimality is less than options.TolFun.

x =

0.5000

0.5000

f =

1.0e-008 \*

0.1653

0.0000

e\_flag =

1

inform =

iterations: 6

funcCount: 7

algorithm: 'trust-region dogleg'

firstorderopt: 1.6525e-009

message: [1x76 char]

>>

По сравнению с предыдущим запуском для нахождения корня с такой же точностью потребовались те же 7 итераций, но количество обращений к функции fun уменьшилось в три раза. Справедливости ради следует отметить, что время работы функции за счет вычисления якобиана увеличилось примерно в два раза. Так что общий выигрыш по скорости нахождения корня составил примерно 50%.

В теории линейных управляемых систем часто возникают матричные уравнения (пример 15.13). Они могут быть как линейными (уравнения Ляпунова и Сильвестра), так и нелинейными (уравнение Риккати – не путать с дифференциальным уравнением Риккати).

Примет 15.13. Матричные уравнения

Рассмотрим квадратное матричное уравнение вида АХ2 + ВХ + С = 0, отвлекаясь от его происхождения. Определим функцию, аргументами которой кроме неизвестной матрицы х являются матрицы-коэффициенты А, В, С:

function F = quadratic(X,А,В,С)

F = А\*Х\*Х+В\*Х+С;

Зададим конкретные матрицы-коэффициенты А, B, C и стартовую матрицу *x*0:

>> A=eye(2); % единичная матрица 2x2

>> B=eye(2); % единичная матрица 2x2

>> C=[-6 -5;0 -6];

>> X0=ones(2); % матрица 2x2, заполненная единицами

>>

и обратимся к функции fsolve:

>> opt = optimset('Display','off');

>> [X1,F,e\_flag]=fsolve(@quadratic,X0,opt,A,B,C)

Обратите внимание на тот факт, что обращение к функции quadratic требует передачи ей не только аргумента х, но и еще трех дополнительных параметров – матриц А, B, **C.** В приведенном выше вызове функции fsolve показано, как эти параметры должны быть переданы.

В результате решения матричной системы уравнений получим следующие результаты:

X1 =

2.0000 1.0000

-0.0000 2.0000

F =

1.0e-009 \*

0.3957 -0.6771

-0.2319 0.3957

e\_flag =

1

>>

Вообще говоря, матрица X1 = [2 1;0 2] является точным решением нашего матричного уравнения. Однако fsolve нашла приближенное решение, в чем можно убедиться, выдав большее количество значащих цифр:

>> format long

>> X1

X1 =

2.000000000088407 0.999999999829213

-0.000000000046373 2.000000000088407

>>

Именно этим и объясняется ненулевая невязка *F* в найденной точке. Задав другую стартовую матрицу X0 = -2\*еуе(2), найдем второе решение квадратного матричного уравнения:

>> X0 = -2\*eye(2);

>> [X1,F,e\_flag]=fsolve(@quadratic,X0,opt,A,B,C)

X1 =

-3.0000 -1.0000

0 -3.0000

F =

1.0e-013 \*

0 0.3730

0 0

e\_flag =

1

>>

Как и в примере 15.10, не всякая стартовая матрица приводит к настоящему решению: для X0 = -ones (2) получим:

>> X0 = -2\*ones(2);

>> [X1,F,e\_flag]=fsolve(@quadratic,X0,opt,A,B,C)

X1 =

-0.5127 -196.9032

-0.0317 -0.5127

F =

-0.0010 -0.0008

0.0008 -0.0010

e\_flag =

0

>>

И довольно большая невязка F, и значение признака e\_f lag = 0 свидетельствуют о том, что найденное "решение" не является корнем нашего уравнения. Итерационный процесс был прерван в связи с превышением максимального количества итераций или обращений к функции quadratic. Установить более точную причину позволяет анализ одного из полей структуры, которую можно получить в качестве выходного параметра:

>> [X1,F,e\_flag,out]=fsolve(@quadratic,X0,opt,A,B,C)

X1 =

-0.5127 -196.9032

-0.0317 -0.5127

F =

-0.0010 -0.0008

0.0008 -0.0010

e\_flag =

0

out =

iterations: 79

funcCount: 400

algorithm: 'trust-region dogleg'

firstorderopt: 0.4016

message: [1x78 char]

>>

Значение поля out.funcCount свидетельствует о том, что превышено максимально допустимое количество обращений к функции.

Рассмотрим пример 15.14.

Пример 15.14. Поиск корней модуля полинома

В теории полиномов рассматривается вещественная функция комплексного аргумента – модуль полинома. Она, естественно, равна о в тех точках комплексной плоскости, которые являются корнями полинома. Рассмотрим комплексный полином *W* = *z*3 + *z*, корнями которого являются Z1,3 = ±i и Z2 = 0.

Построим функцию – модуль этого полинома, в которой комплексный аргумент z представлен как вещественный вектор *х*[ 1:2]:

function y = abspoly(x)

z = complex(x(1), x(2));

w = z.^3+z;

y = abs(w);

Попробуем с помощью функции fsolve найти решение нелинейной системы из одного уравнения с двумя неизвестными, задавая разные начальные

точки:

>> xl=-0.5; x2=1.5; x0=[xl;x2];

>> options=optimset('Display','off') ;

>> [x,f,e\_flag] = fsolve(@abspoly,x0,options)

Warning: Default trust-region dogleg method of FSOLVE cannot

handle non-square systems; using Gauss-Newton method instead.

> In fsolve at 248

x =

0.0000

1.0000

f =

3.5725e-006

e\_flag =

1

>> warning.)

Вывод текста предупреждения можно подавить, если предварительно вы-

полнить команду:

» warning off Optimization:fsoive:NonSquareSystem

**Самостоятельная работа**

Найти решение систем уравнений

1. cos *x* + 2*y* = 2,

*x*2/3 – *y*2/3 = 1/

1. *y*2 –*x*2/2 = 1,

2,1 sin(*x* – 1)+ y =0,1/

1. *x*2+ 2*y*2 = 1,

*x*2 – *y*2 =1.

**Нахождение экстремумов функций**

Вычислительные функции MATLAB позволяют найти точки минимума функции одной или нескольких переменных. Результатом является локальный минимум, т. е. точка, в окрестности которой значения исследуемой функции превышают ее значение в точке локального минимума. Для определения точек локального максимума нет специальной функции, поскольку достаточно искать минимум функции с обратным знаком.

**Минимизация функции одной переменной**

Поиск локального минимума функции одной переменной на некотором отрезке осуществляется при помощи fminbnd, использование которой схоже с fzero. Найдите локальные минимумы функции

|  |  |  |
| --- | --- | --- |
|  |  | (6.2) |

на отрезке [-1.5,1.5]. Требуется предварительно создать соответствующую файл-функцию, назвав ее, к примеру ftest, или ввести inline или анонимную функцию.

>> ftest=@(x) (exp(x.^2)+sin(3\*pi\*x))

ftest =

@(x) (exp(x.^2)+sin(3\*pi\*x))

>>

>> fplot(ftest,[-1.5 1.5])

>> grid on

Перед нахождением локальных минимумов постройте график исследуемой функции командой fplot. Из графика, приведенного на рис. 6.2, видно, что исследуемая функция имеет четыре локальных минимума. Вычислите значение *x*, при котором достигается второй локальный минимум, задав первым аргументом fminbnd имя файл-функции или указатель на нее, а вторым и третьим – границы отрезка, на котором ищется локальный минимум (установка точности и дополнительных параметров минимизации описана в разд. "Управление ходом вычислений"):

>> x2 = fminbnd(ftest,-0.5, 0)

x2 =

-0.1629



Рис. 6.2. Расположение локальных минимумов функции (6.2)

Для лучшего понимания работы fminbnd задайте интервал поиска, содержащий все точки локальных минимумов.

>> xx= fminbnd('ftest', -1.5, 1.5)

xx =

0.4888

Для одновременного вычисления значения функции в точке минимума следует вызвать fminbnd с двумя аргументами:

>> [x2, f] = fminbnd(ftest, -0.5, 0)

x2 =

-0.1629

f =

0.0275

Так же, как и fzero, функция fminbnd может быть вызвана с тремя выходными аргументами:

>> [x2,f,flag]=fminbnd(ftest,-0.5, 0);

Аргумент flag может принимать три значения: положительное – при нахождении локального минимума, нулевое – при достижении максимального количества вызовов исследуемой функции и отрицательное – в случае расходимости вычислительного процесса.

Найдите самостоятельно остальные локальные минимумы и максимумы функции (6.2).

Искать минимум функции можно также с применением fminsearch, рассматриваемой далее, которая предназначена для минимизации функции одной и нескольких переменных.

**Минимизация функции нескольких переменных**

Минимизация функции нескольких переменных является более сложной задачей по сравнению с минимизацией функции одной переменной, однако комбинированный адаптивный алгоритм функции fminsearch позволяет во многих случаях успешно решить эту задачу. Функция fminsearch требует указания начального приближения для искомой точки минимума, которое в случае функции одной переменной должно быть числом, а для функции нескольких переменных – вектором.

Задайте начальное приближение равное -0.5 в примере предыдущего раздела и найдите точку локального минимума:

>> x2 = fminsearch(ftest, -0.5)

x2 =

-0.1629

Выбирая последовательно подходящие начальные приближения, найдите все точки минимумов и максимумов функции.

Рассмотрим теперь минимизацию функций нескольких переменных на примере функции двух переменных

|  |  |  |
| --- | --- | --- |
|  | *f*(*x*, *y*) = sin π*x* sin π*y*. | (6.3) |

Сначала получите представление о поведении функции, построив ее линии уровня при помощи следующих команд:

>> [X, Y] = meshgrid(0:0.01:2);

>> Z = sin(pi\*X).\*sin(pi\*Y);

>> [CMatr, h] = contour(X, Y, Z,[-0.96, -0.9, -0.8, -0.5, -0.1, 0.1, ...,

0.5, 0.8, 0.9, 0.96]);

>> clabel(CMatr,h)

>> colormap (gray)

>>

На получившемся графике, приведенном на рис. 6.3, видно расположение локальных минимумов и максимумов.

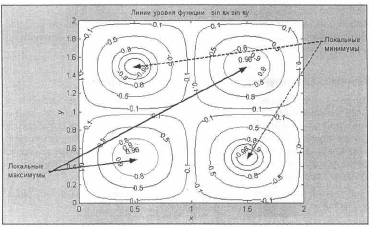




Рис. 6.3. Расположение локальных максимумов и минимумов функции (6.3)

Перед применением fminsearch необходимо создать файл-функцию, вычисляющую значения искомой функции, причем аргументом файл-функции должен быть вектор, первый элемент которого соответствует переменной х, а второй – у. Текст требуемой файл-функции ftest2 приведен в листинге 6.2.

Листинг 6.2. Файл-функция ftest2

function f = ftest2(v)

x = v(1);

y = v(2);

f = sin(pi\*x).\*sin(pi\*y);

Теперь для нахождения локального минимума вызовите fminsearch с двумя входными аргументами – именем файл-функции и начальным приближением и выходным аргументом – вектором с координатами искомой точки минимума:

>> M = fminsearch('ftest2', [1.4, 0.6])

M =

1.5000 0.5000

Решение найдено с точностью 10-4, как по значениям х и у, так и по значению функции. Для получения не только вектора с координатами точки минимума, но и значения функции следует вызвать fminsearch с двумя выходными аргументами:

>> [M, f] = fminsearch(@ftest2, [1.4, 0.6])

M =

1.5000 0.5000

f =

-1.0000

Обращение к fminsearch с третьим дополнительным выходным аргументом flag

>> [M, f, flag] = fminsearch(@ftest2, [1.4, 0.6])

M =

1.5000 0.5000

f =

-1.0000

flag =

1

позволяет записать в него информацию о причине останова вычислительного алгоритма. Смысл его значений тот же, что и для функции fminbnd, рассмотренной в предыдущем разделе.

Исследуемая функция может зависеть от произвольного числа переменных *n*. В этом случае входной аргумент argvect соответствующей файл-функции и начальное приближение должны быть векторами длины *n*.

В нашей задаче не обязательно было программировать файл-функцию в М-файле. Поскольку входным аргументом анонимной функции может быть вектор, то следующие команды приведут к поиску минимума:

>> fun = @(v)sin(pi\*v(1)).\*sin(pi\*v(2));

>> [M, f] = fminsearch(fun, [1.4, 0.6])

M =

1.5000 0.5000

f =

-1.0000

Примечание

Optimization Toolbox содержит дополнительные средства минимизации функций. Например, fminunc можно использовать для поиска минимума функции нескольких переменных.

**Управление ходом вычислений**

Функции fzero, fminbnd и fminsearch допускают определение дополнительных параметров для управления вычислительным процессом и контроля за ним. Параметры задаются в управляющей структуре, которую мы будем называть options, как в справочной системе MATLAB, хотя имя может быть произвольным. Перед вызовом вычислительных функций следует предварительно сформировать переменную options в соответствии с характером требуемого контроля, воспользовавшись функцией optimset. Переменная options на самом деле является структурой. Это новый тип данных – до сих пор вы имели дело только с числовыми массивами.

Приступим к формированию структуры options на примере минимизации функции одной переменной (6.2) при помощи fminbnd. Получим сначала информацию о работе алгоритма минимизации на его последнем шаге. Для этого вызовем функцию optimset с одним выходным аргументом options и двумя входными 'Display' и 'final', а затем укажем options в дополнительном четвертом входном аргументе fminbnd. Всюду дальше мы будем использовать термины свойство, вид контроля, опция или параметр (в данном случае Display) и значение свойства или параметра (в данном случае 'final'). Установите формат вывода long, поскольку нам понадобятся все значащие цифры для анализа результата.

>> format long

>> options = optimset('Display', 'final');

>> x2 = fminbnd(ftest, -0.5, 0, options)

Optimization terminated:

the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004

x2 =

-0.162898455095188

>>

Кроме координаты точки локального минимума выводится информация об успешном завершении вычислительного процесса и точности, с которой (по умолчанию) найдено решение. Для изменения точности следует заново сформировать структуру options, указав еще одну пару входных аргументов. В следующем примере мы задаем не только опцию Display, но и точность 10-9 по аргументу при помощи параметра TolX.

>> options = optimset('Display', 'final', 'TolX', 1.0e-09);

>> x2 = fminbnd(ftest, -0.5, 0, options)

Optimization terminated:

the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-009

x2 =

-0.162899428412680

>>

Сравните полученный результат со значением, вычисленным fminbnd с используемой по умолчанию точностью 10-4. Различие получено в шестом знаке после десятичной точки, следовательно, при первом вычислении точность была 10-5, а не 10-4 как выведено в сообщении. Информация, возвращаемая функцией fminbnd, содержит гарантированную точность, в то время как реальная точность может быть больше.

Аналогичным образом точность задается при нахождении корней и минимизации функции нескольких переменных.

В общем случае входные аргументы optimset задаются попарно options = optimset (...., 'вид\_контроля', значение, …)

В двух предыдущих примерах Display и TolX – виды контроля, а 'final' и 1.0е-09, соответственно – их значения. Возможные сочетания параметров вид\_контроля и значение приведены в табл. 6.1.

Таблица 6.1. Параметры optimset

|  |  |  |  |
| --- | --- | --- | --- |
| Вид контроля | Значение | Результат | Примечание |
| Display | 'off | Информация о вычислительном процессе не выводится |  |
| 'iter' | Выводится информация о каждом шаге вычислительного процесса |  |
| 'final' | Выводится информация только о завершении вычислительного процесса |  |
| 'notify' | Выводится предупреждение, если процесс не сходится (используется по умолчанию) |  |
|  |  |  |  |
| Таблица 6.1. (продолжение) | | | |
| Вид контроля | Значение | Результат | Примечание |
| MaxFunEvals | Целое число | Максимальное количество вызовов исследуемой функции | Только для fminbnd fminsearch |
| Maxlter | Целое число | Максимальное количество итераций вычислительного процесса | Только для fminbnd fminsearch |
| TolFun | Положительное вещественное число | Точность по функции для останова вычислений | Только для fminbnd fminsearch |
| TolX | Положительное вещественное число | Точность по аргументу функции для останова вычислений |  |

Ограничивать количество вызовов функций и число итераций имеет смысл, если есть опасение, что получить решение не удастся из-за расхождения вычислительного процесса. В некоторых случаях приходится уменьшать точность, например, если вычисление исследуемой функции занимает много времени, а требуется получить только несколько первых значащих цифр ответа.

Пользователям, имеющим представление о численных методах, полезны сведения о ходе вычислений, выводимые на экран при значении параметра Display, равном 'iter'. Последовательность команд

>> options = optimset('Display', 'iter', 'TolX', 1.0e-09);

>> x2 = fminbnd(ftest, -0.5, 0, options)

приводит к появлению в командном окне кроме результата еще и таблицы, каждая строка которой соответствует одной итерации и содержит информацию о том, какой раз вызывалась исследуемая функция, текущее приближение и значение функции от него и метод, применяемый на данной итерации.

Func-count x f(x) Procedure

1 -0.309017 0.873024 initial

2 -0.190983 0.063294 golden

3 -0.118034 0.117247 golden

4 -0.163802 0.0275587 parabolic

5 -0.162747 0.0275227 parabolic

6 -0.162898 0.0275217 parabolic

7 -0.162899 0.0275217 parabolic

8 -0.162899 0.0275217 parabolic

9 -0.162899 0.0275217 parabolic

10 -0.162899 0.0275217 parabolic

Optimization terminated:

the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-009

x2 =

-0.162899428412680

>>

Функция optimset служит также для модификации существующей структуры. Например, если сначала была задана только точность по аргументу

>> options = optimset('TolX', 1.0е-09);

а в дальнейшем потребовалось дополнительно указать точность по функции и получить информацию о каждом шаге вычислительного процесса, то достаточно обратиться к функции optimset следующим образом:

>> options = optimset(options, 'TolFun', 1.0e-07, 'Display', 'iter');

Примите во внимание, что можно создать несколько структур с различными именами, к примеру: options1 и options2, для различных вариантов управления вычислительным алгоритмом.

Для просмотра всех текущих опций можно вывести структуру options в командное окно или открыть ее в редакторе массивов двойным щелчком по строке с options в окне браузера переменных рабочей среды. Получение значения отдельного параметра производится при помощи функции optimget, входными аргументами которой являются имя структуры и название требуемого параметра:

>> err = optimget(options, 'TolX')

err =

1.0000e-009

Если данному параметру не было присвоено значение при генерации структуры, то возвращается пустой массив.

Вызов функции optimset без входных аргументов

>> optimset

приводит к отображению в командном окне названия всего множества параметров вместе с их допустимыми значениями. В фигурных скобках указаны значения, используемые в вычислительном алгоритме по умолчанию.

В разделе справочной системы MATLAB Mathematics: Function Functions: Minimizing Functions and Finding Zeros размещена подробная информация о минимизации функций и решении уравнений и, кроме того, приведены ссылки на литературу с описанием вычислительных методов, реализованных в MATLAB. Некоторые дополнительные возможности функций fplot, fzero, fminbnd и fminsearch обсуждаются в следующем разделе. В заключение этого раздела отметим, что большинство вычислительных функций, в том числе fzero, fminbnd и fminsearch, написаны на языке программирования MATLAB. Они имеют открытый код и расположены в подкаталоге \toolbox\matlab\funfun основного каталога MATLAB.

**Исследование функций, зависящих от параметров**

Функции fzero, fminbnd и fminsearch поддерживают работу с математическими функциями, зависящими от параметров. Заметим, что версия MATLAB 7.0 унаследовала от предыдущей тот же подход, что остался в fplot, хотя его описание исключено из справочной системы. Значения параметров последовательно задаются в списке входных аргументов после управляющей структуры options. Заполнение самой структуры не обязательно, если при решении уравнения или минимизации функции достаточно определенных по умолчанию значений 'Display', 'MaxFunEvals', 'Maxlter', 'TolFun', 'TolX'. Как и в случае с fplot допускается указание пустого массива вместо неинтересующей нас управляющей структуры, например

function f = pfun(x, pi, p2)

f = exp(pl\*x) - p2\*sin(x);

>> [xl, fl] = fzero(@pfun, 0, [ ], 0.1, 2)

xl =

0.5570

fl =

0

Аналогичным образом решается задача о поиске локального минимума функции, зависящей от параметров.

В версии 7.0 рекомендуется применять подходы, основанные на использовании анонимных или вложенных функций. В первом случае достаточно задать значения параметров в переменных рабочей среды и обратиться к функции, определив в качестве аргумента анонимную функцию:

>> p1=0.1;

>> p2=2;

>> [xl, fl] = fzero(@(x) exp(p1\*x) - p2\*sin(x), 0)

xl =

0.5570

fl =

0

Поскольку при создании анонимной функции переменные среды фиксируются, то нет смысла определять отдельно анонимную функцию и передавать ее имя в качестве аргумента, ибо для новых параметров все команды надо повторить. В листинге 6.4 приведен пример использования вложенных функций для решения той же задачи.

function [x,y] = froot(p1, p2, x0)

[x, y] = fzero(@pfun, x0);

function f = pfun(x)

f = exp(p1\*x) - p2\*sin(x);

end

end

Теперь вместо вызова функции fzero следует обращаться к собственной функции froot:

>> [x, y] = froot(0.1, 2,0)

x =

0.5570

y =

0

При необходимости можно добавить управляющую структуру options в список входных аргументов для передачи ее fzero. Пользователь пакета может выбрать наиболее приемлемый для него вариант.

Ануфриев И. Е., Смирнов А. Б., Смирнова Е. Н. MATLAB 7. – СПб.: БХВ-Петербург, 2005. – 1104 с: ил.