Пакеты научных вычислений

Лекция 7

Программирование в Maple

Создание процедур. Создание модулей, пакетов, библиотек. Работа с файлами.

Создание графических приложений

Maplets

Наседкина А. А.

Процедуры в Maple

- Общие сведения о процедуре Maple
- Полный синтаксис процедуры
- Вывод кода процедуры на экран
- Локальные и глобальные переменные
- Работа с аргументами процедур
- Рекурсивные процедуры
- Вывод сообщений об ошибках, выход из процедуры

Общие сведения о процедуре, краткий синтаксис процедуры

• Процедура – это пользовательская команда Maple.
• Процедура может не иметь аргументов, иметь один аргумент или несколько аргументов (тогда они перечисляются через запятую).
• Все описание процедуры должно находиться в одной выполнимой группе (execution group). Переход на новую строку осуществляется с помощью **Shift+Enter**

Краткий синтаксис процедуры

- *proc_name*:=**proc (***parameterSequence***)**

  statementSequence;

  end proc;

Слова, выделенные синим могут отсутствовать.

Пример простейшей процедуры без аргументов

```
> ex := proc( )
  √2;
  end proc;
```

ex := **proc( ) sqrt(2) end proc**

Вызов и выполнение процедуры:

```
> ex( );
```

$\sqrt{2}$

- *proc_name* – имя процедуры
- **proc … end proc** – служебные слова, начало и конец процедуры
- *statementSequence* – последовательность выражений, реализующих *тело* процедуры. Возвращается значение последнего выражения (команды) из этой последовательности или значение выражений, указанных после команды **return**.

Полный синтаксис процедуры

- *proc_name:=***proc** (*parameterSequence* **::** *type*) **::** *returnType*;
  local *localSequence*;
  global *globalSequence*;
  option *optionSequence*;
  description *descriptionSequence*;
  uses *usesSequence*;
  statementSequence;
  end proc;

Слова, выделенные синим, могут отсутствовать.

- *parameterSequence* – последовательность формальных параметров (*аргументов*) процедуры. Каждому формальному параметру можно предписать (*декларировать*) определенный тип данных с помощью оператора двойного двоеточия **::** и следующего за ним названия типа данных *type*. При вызове процедуры в случае несоответствия какого-либо параметра его заявленному типу будет выдаваться системное сообщение об ошибке. Для параметров можно также задать значения по умолчанию.
- *returnType* – необязательный предполагаемый тип возвращаемого значения процедуры. По умолчанию, если тип возвращаемого значения не соответствует предполагаемому, ошибки не происходит.

Полный синтаксис процедуры (продолжение)

- **local** – служебное слово для описания последовательности локальных переменных *localSequence*. *Локальными* называются переменные, которые используются только внутри данной процедуры. Для локальных переменных можно задавать тип в виде **::** *returnType*.
- **global** – служебное слово для описания последовательности глобальных переменных *globalSequence*. *Глобальными* называются переменные, которые не являются локальными, но также используются данной процедурой. Описание глобальных переменных используется в том случае, если этим переменным внутри процедуры будут присвоены какие-то значения. Для глобальных переменных нельзя задать тип внутри процедуры.

- **option** – служебное слово для описания последовательности опций процедуры *optionSequence*. В качестве опций используются специальные слова, например, **arrow** (стрелка), **builtin** (встроенная процедура), **operator** (оператор), **remember** (опция для эффективной работы рекурсивных процедур) , **`Copyright...`** и некоторые другие.

- **description** – служебное слово, за которым следуют комментарии *descriptionSequence* о назначении процедуры и ее работе (одна или несколько строк). В отличие от комментариев, задаваемых символом #, данная информация выводится на экран при печати кода процедуры.

- **uses** – служебное слово для описания последовательности *usesSequence* связанных имен и модулей, которые будут использованы в теле процедуры. Может быть использовано для подключения пакетов, например: **`uses StringTools`**;

Примеры процедур: обязательные аргументы, количество аргументов при вызове процедуры, декларирование типов аргументов

Пример процедуры с аргументами

```
> p := proc(a, b) a + b; a - b : end proc:
```

Возвращается значение последней команды в теле процедуры

```
> p(1, 2);
                                  -1
> p(2, 1);
                                   1
```

Избыточное количество аргументов при вызове процедуры – ошибки не происходит:

```
> p(1, 2, 3);
 -1
```

Недостаточное количество аргументов при вызове процедуры – сообщение об ошибке:

```
> p(2);
Error, invalid input: p uses a 2nd argument, b, which is missing
```

Пример процедуры с декларированием типов аргументов

```
> f := proc (a :: integer, b) a + b end proc:
> f(2, 3);
                                   5
> f(2.5, 3);
Error, invalid input: f expects its 1st argument, a, to be of
type integer, but received 2.5
```

Примеры процедур: декларирование типов аргументов, несколько типов данных, значения по умолчанию

Пример процедуры с декларированием нескольких типов данных для аргументов

```
f := proc(a :: {integer, float}, b :: integer) a^b end proc:
> f(2, 3);
                                  8
> f(2, 2.5);
Error, invalid input: f expects its 2nd argument, b, to be of
type integer, but received 2.5
> f(2.5, 2);
                                 6.25
```

Пример процедуры с декларированием типов аргументов и их значений по умолчанию

```
> f := proc(a :: integer := 10, b :: integer := 100.1) a + b end proc:

> f(3);
                                103.1
> f(3, 4);
                                  7
> f(3.5, 4.5)
                                110.1
```

Примеры процедур: описатели option, description, uses

*Пример процедуры с опциями функционального оператора (использование **option**)*

```
> f := proc(x) option operator, arrow; x^2-1 end proc;
```

Процедура задает функциональный оператор, ее запись эквивалента команде:

```
> f := x -> x^2-1;
```

*Пример процедуры с комментарием о ее назначении (использование **description**)*

```
> lc := proc( s, u, t, v )
        description "forms a linear combination of the
arguments";
        s * u + t * v
end proc:
```

Вывод на экран комментариев к процедуре

```
> Describe(lc);
# forms a linear combination of the arguments lc( s, u, t, v )
```

*Пример процедуры с подключением пакета (использование **uses**)*

```
> LastWord:=proc(s::string)
uses StringTools;
Split(s);%[-1];
end proc:
> LastWord("Hello world!");
                                    "world!"

> LastWord(a);
Error, invalid input: LastWord expects its 1st argument, s, to
be of type string, but received a
```

Возврат нескольких значений из процедуры

Процедура находит все простые числа на заданном интервале и выводит их количество и сами числа в виде списка.

```
> PrimesAtInterval := proc(a, b :: integer)
  local COUNT, PRIMES, n;
  COUNT := 0 : PRIMES := [ ];
  for n from a to b do
  if isprime(n) then COUNT := COUNT + 1; PRIMES := [op(PRIMES), n]
      end if
  end do:
  COUNT, PRIMES;
  end proc:

> PrimesAtInterval(100, 200);
21, [101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
    173, 179, 181, 191, 193, 197, 199]

> (n, s) := PrimesAtInterval(100, 200) :
> n;
                                  21

> s;
[101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199]
```

Вывод кода процедуры на экран

Вывод кода пользовательской процедуры на экран

- **print(***proc_name***);**
- **eval(***proc_name***);**

Пример пользовательской процедуры

```
> lc := proc( s, u, t, v )
        description "forms a linear combination
                     of the arguments";
        s * u + t * v
end proc:
> eval(lc);
```

proc(*s*, *u*, *t*, *v*)

description

"form a linear combination of the arguments;"

s * *u* + *t* * *v*

end proc

Вывод кода процедуры на экран (продолжение)

Вывод кода процедуры из библиотеки Maple на экран
(кроме встроенных процедур, с опцией builtin)

- **interface('verboseproc'=2): print(*proc_name*);**
- **interface('verboseproc'=2): eval(*proc_name*);**

Пример процедуры из библиотеки

```
> print(issqr);
            proc(n) ... end proc
> interface('verboseproc' = 2):
> print(issqr);
      proc(n)
          option
          Copyright (c) 1990 by the University of
          Waterloo. All rights reserved. ;
          if type(n, integer) then
              evalb(isqrt(n)^2 = n)
          elif type(n, numeric) then
              false
          else
              'issqr(n)'
          end if
      end proc
```

Пример встроенной процедуры

```
> interface('verboseproc' = 2):
> print(conjugate);
      proc( )
          option builtin = conjugate;

      end proc
```

Локальные и глобальные переменные: декларирование локальных переменных

Пример процедуры с локальными переменными

Процедура *maximum*, находит максимум из заданного списка целых чисел.

```
> maximum := proc (s::(list(integer)))
local max, i;
 max := s[1];
  for i to nops(s) do
     if s[i]>max then
        max := s[i]
     end if;
  end do;
 max;
end proc;

> maximum([4,1,8,-100]);
```

8

```
> maximum(4,1,8,-100);
Error, invalid input: maximum expects its 1st argument, s, to be
of type list(integer), but received 4
> maximum([4,1,8,z]);
Error, invalid input: maximum expects its 1st argument, s, to be
of type list(integer), but received [4, 1, 8, z]
```

Локальные и глобальные переменные: декларирование локальных переменных

Если удалить строку описания локальных переменных, то будут выведены предупреждения о том, что в процедуре используются переменные max и i, которые будут декларироваться локальными:

```
> maximum := proc (s :: list (integer))
    max := s[1];
    for i from 1 to nops(s) do
      if s[i] > max then
        max := s[i]
      end if
    end do;
    max;
  end proc:
Warning, `max` is implicitly declared local to procedure `maximum`
Warning, `i` is implicitly declared local to procedure `maximum`
```

Различие между локальными и глобальными переменными

1) Декларирование локальной переменной

```
> my_pi:=3.14:
> CircleArea1:=proc(r)
local my_pi;
my_pi:=evalf(Pi,10);
my_pi*r^2;
end proc:
> CircleArea1(5);
                    78.53981635
> my_pi; r:=5: my_pi*r^2;
                    3.14
                    78.50
```

2) Декларирование глобальной переменной

```
> my_pi:=3.14:
> CircleArea2:=proc(r)
global my_pi;
my_pi:=evalf(Pi,10);
my_pi*r^2;
end proc:
> CircleArea2(5);
                    78.53981635
> my_pi; r:=5: my_pi*r^2;
                    3.141592654
                    78.53981635
```

Работа с аргументами процедуры: параметры _passed и _npassed для всех переданных аргументов

- **_passed** – последовательность всех аргументов, переданных процедуре при ее вызове (устаревший вариант: **args**), имеет тип *exprseq*
- **_npassed** – число всех аргументов, переданных процедуре при ее вызове (устаревший вариант: **nargs**)

Пример процедуры с использованием имен _passed и _npassed

Процедура находит максимум из произвольной последовательности чисел.

```
> maximum := proc () local max, i;
    max := _passed[1];
    for i from 2 to _npassed do
        if _passed[i] > max then
            max := _passed[i]
        end if
    end do;
    max;
end proc:
> maximum(2, 5, 77, -10, 100.2);
```

100.2

Работа с аргументами процедуры: параметры _rest и _nrest для «лишних» аргументов

Если при вызове процедуры число переданных аргументов больше числа обязательных, то можно получить доступ к оставшимся «лишним» аргументам:

- **_rest** – последовательность «лишних» аргументов, переданных процедуре при ее вызове, имеет тип *exprseq*
- **_nrest** – число «лишних» аргументов, переданных процедуре при ее вызове

Пример процедуры с использованием имени _rest

```
> f := proc(a, b) local xarg, sumarg, x;
  xarg := a, b, _rest; sumarg := 0;
  for x in xarg do sumarg := sumarg + x; end do;
  sumarg;
  end proc:
> f(3, 2)
                                    5
> f(1, 2, 3, 4, 5)
                                   15
> f(1)
Error, invalid input: f uses a 2nd argument, b, which is missing
```

Рекурсивные процедуры: опция remember

Опция **remember** сокращает время работы рекурсивной процедуры

Пример. Вычисление n-го числа Фибоначчи. Числа Фибоначчи задаются формулой $f_n = f_{n-1} + f_{n-2}$ для $n \geq 2$, $f_0=0$, $f_1=1$.

```
> Fib1 := proc(n)
  if n < 2 then n
  else Fib1(n-1) + Fib1(n-2)
  end if end proc:

> Fib2 := proc(n)
  option remember
  if n < 2 then n
  else Fib2(n-1) + Fib2(n-2)
  end if end proc:

> Fib1(30);
                          832040

> Fib2(30);
                          832040
```

Проверим с помощью команды `fibonacci(n)` из пакета `combinat`.

```
> with(combinat) :fibonacci(30);
                          832040
```

Вычислим время работы процедур:

```
> time(Fib1(30)); time(Fib2(30));
                          2.324
                          0.
```

Вывод пользовательского сообщения об ошибке и аварийный выход из процедуры

- **error** "*Message %1….String…%2….*",*par1*,*par2*,…

В строке сообщения об ошибке вместо %1 подставляется значение *par1*, вместо %2 подставляется значение *par2* и т. д.

```
sq := proc(x :: numeric)
 if x < 0 then error "Неверный аргумент: %1", x;
    end if;
 sqrt(x);
 end proc:
```

При выполнении команды **error** все оставшиеся команды в теле процедуры игнорируются.

Вызов и выполнение процедуры.

```
> sq(2);
```

$$\sqrt{2}$$

```
> sq(2.5);
```

$$1.581138830$$

```
> sq(-2);
Error, (in sq) Неверный аргумент: -2
```

(выдается пользовательское сообщение об ошибке)

```
> sq(`b`);
Error, invalid input: sq expects its 1st argument, x, to be of
type numeric, but received b
```

(выдается системное сообщение об ошибке при проверке типа аргумента)

Выход из процедуры в любом месте ее тела и возврат значений

- **return** *exp1,expr2,...*

```
> sq1 := proc(x :: numeric)
    if x < 0 then return abs(x) end if;
    sqrt(x);
  end proc:
> sq1(4)
                              2
> sq1(a)
Error, invalid input: sq1 expects its 1st argument, x, to be of
type numeric, but received a
> sq1(-4.5)
                             4.5
```

При выполнении команды **return** все оставшиеся команды в теле процедуры игнорируются.

Обобщение процедур

- Модули
- Макроопределения alias и macro
- Создание пакетов и библиотек

Модуль

- **Модуль** – это обобщение процедуры. Процедура позволяет создать команду из последовательности Maple-команд. С помощью модуля можно создать более сложную структуру, включающую набор процедур и данных. Модуль позволяет *экспортировать переменные*

Полный синтаксис модуля

- **module**()
  export eseq;
  local lseq;
  global gseq;
  option optseq;
  description dseq;
  uses usesSequence;
  statementSequence
  end module
- Описатель **export** задает имена процедур и переменных, доступных для вызова
- Остальные описатели аналогичны соответствующим описателям в процедуре

Сравнение модуля и процедуры

- **Процедура**

```
> makezp := proc(a, b) local plus, times; plus := (a, b) -> a + b; times := (a, b) -> a*b; plus, times end proc:
> makezp(5, 10)
                                    plus, times
> z := makezp( )
                                  z := plus, times
> z[1](5, 10)
                                        15
> z[2](5, 10)
                                        50
> z(5)
Error, invalid input: plus uses a 2nd argument, b, which is missing
> z(5, 10, 20)
                                      15, 50
```

- **Модуль**

```
> z := module( ) export plus, times; plus := (a, b) -> a + b; times := (a, b) -> a*b end module:
> z:-plus(5, 10)
                                                15
> z:-times(5, 10)
                                                50
```

- Обращение к экспортируемым объектам модуля – с помощью команды **:-**

Макроопределения: alias

Новые имена для уже существующих функций

- **alias(e1, e2, ..., eN)** – аббревиатуры для имен и функций, где **ei** имеют вид **new=old**
- Нельзя определить одну аббревиатуру alias через другую

```
> binomial(4, 2);
```

$$6$$

```
> alias(C = binomial) :
> C(4, 2)
```

$$6$$

```
> alias(F = F(x), Fx = d/dx F(x))
```

$$C, F, Fx$$

```
> ∂/∂x F
```

$$Fx$$

```
> ∂/∂x Fx
```

$$\frac{\partial^2}{\partial x^2} F$$

```
> restart,
> alias(v = LinearAlgebra[VandermondeMatrix])
```

$$v$$

```
> v([3, 2, 1])
```

$$\begin{bmatrix} 1 & 3 & 9 \\ 1 & 2 & 4 \\ 1 & 1 & 1 \end{bmatrix}$$

Макроопределения: macro

Макросы

- **macro(e1, e2, …, eN)** – макро-абрревиатуры для имен и функций, где **ei** имеют вид **new=old**
- Можно использовать для переопределения имеющихся команд

```
> macro(F = combinat['fibonacci'])
                                F
> F(30);
                              832040
> subs(x = 2, sin(x));
                              sin(2)
> macro(subs = evalf@subs) :
> subs(x = 2, sin(x));
                           0.9092974268
```

Создание пакетов

- **Пакет** – это набор пользовательских процедур для решения задач определенного класса.

  Сначала создаются и тестируются процедуры для пакета. Созданный пакет нужно сохранить в виде .m-файла
- **save** pack_name, “filename.m” – сохранение пакета pack_name в файл
- **read** “filename.m” – чтение файла с пакетом с диска
- **with**(pack_name) – подключение пакета

```
> Circle[Area] := proc(r) Pi·r^2; end proc:
> Circle[Length] := proc(r)2·Pi·r; end proc:
> save Circle, "circle.m";
> restart;
> with(Circle) :
Error, invalid input: with expects its 1st argument,
pname, to be of type {`module`, package}, but received
Circle
> read "circle.m";
> with(Circle);
                        [Area, Length]
> Area(R)
                            π R^2
```

Создание библиотек

- **Библиотека** – это набор пользовательских процедур и пакетов.

  Сначала нужно создать файл для библиотеки и сохранить путь к нему. Затем задать (подключить) пакеты, процедуры, модули.
- **march** – команда для работы с файлами библиотек
- **savelibname** – переменная с сохраненным путем к библиотеке
- **savelib(name1,name2,..)** – сохранение пакетов и процедур в библиотеку
- **libname** – переменная с путем к библиотеке

```
> restart;
```

Создание библиотеки в каталоге c:\temp (должен существовать)

```
> march('create', "c:\\temp\\my.lib");
> savelibname := "c:\\temp\\my.lib" :
> read "circle.m"; # чтение файла с предварительно созданным пакетом
> with(Circle);# подключение пакета
```

[Area, Length]

```
> sqr := proc(x) x^2; end proc: # определение процедуры
> savelib(sqr, Circle) :
> restart;
```

Подключение дополнительно созданной библиотеки

```
> libname := libname, "c:\\temp\\my.lib";
```

libname := "C:\Program Files (x86)\Maple 11 /lib", "c:\temp\my.lib"

```
> with(Circle) :
> Length(a); sqr(a);
```

$$2\,\pi\,a$$

$$a^2$$

Работа с файлами. Команды ввода и вывода

- Команды для работы с файлами
- Сохранение в файл
- Запись в файл
- Чтение из файла

Работа с файлами

- **save name1, name2, ..., namek, fileName.ext** – сохранение переменных в файл **fileName** (расширение **.m** – внутренний формат Maple, можно указать другое)

Открытие файла

- **open(“fileName.ext”,mode)** – открытие файла (без буферизации), в качестве **mode** можно указать **READ** или **WRITE**
- **fopen(“fileName.ext”,mode,type)** – открытие файла (с буферизацией) , в качестве **mode** можно указать **READ, WRITE** или **APPEND**, в качестве **type – BINARY** или **TEXT**

Чтение из файла

- **read fileName.ext** – чтение информации из файла **filename**
- **readdata(fileID, format, n)** – чтение числовой информации в заданном формате из файла
- **readline(“fileName.ext”)** – чтение следующей строки из заданного файла

Работа с файлами: продолжение

Запись в файл

- **writeto(“fileName.ext”)** – запись всех последующих команд в файл,
- **appendto (“fileName.ext”)** – добавление команд без перезаписи содержимого;
- **writeto(terminal)** – возврат к записи на экран
- **writedata(fileID, data, format)** – запись числовой информации в файл в заданном формате
- **writeline(“fileName.ext”, str ...)** – запись строк в файл
- **fprintf(file, fmt, x1, ..., xn)** – форматированная запись в файл

Закрытие файла

- **fclose(file ...)** – закрытие файла без буферизации
- **close(file ...)** – закрытие файла с буферизацией

Сохранение переменной в файл: использование команды save

- **file.m –** файл внутреннего формата Maple

```
> a := sqrt(2) :
> save a, "d:\\my_file.m" # файл будет создан
```

Содержимое файла "myfile.m" :

```
M7R0
I"a*$""##"""F$6"
```

```
> save a, "d:\\my_file1.txt"
```

Содержимое файла "my_file1.txt" :

```
a := 2^(1/2);
> restart;
> read "d:\\my_file.m"
> a;
```

$$\sqrt{2}$$

Запись в файл и на экран: использование команды writeto

Установка рабочей директории

- **currentdir()** – вывод на экран пути к каталогу с файлом (текущая рабочая директори)
- **currentdir(dirName) –** задание нового пути к каталогу с файлом

```
> currentdir("d:\\"); writeto("myfile.mw")
> d/dx (x^3 - x^2)
> writeto(terminal)
> a := x + y
                    a := x + y
> writeto("myfile.mw")
> x := solve(x + 5)
```

Содержимое файла myfile.mw будет перезаписано

Чтение числовой информации из текстового файла

- **readdata(fileID, format, n)** – чтение числовой информации
  fileID – описатель файла (переменная)
  n - число колонок в файле с данными, колонки должны быть разделены пробелами
  format – формат данных **integer, float, string** или список этих типов

Файл data.txt содержит данные:

```
1  1  50
1  2  55
2  1  55
2  2  70
```

```
> restart,
> currentdir("d:\\");# файл находится там
                                                    "d:\"
> readdata("data.txt", 3)#чтение как вещественных чисел по умолчанию
                                  [[1., 1., 50.], [1., 2., 55.], [2., 1., 55.], [2., 2., 70.]]
> readdata("data.txt", integer, 3) # чтение целых чисел
                                    [[1, 1, 50], [1, 2, 55], [2, 1, 55], [2, 2, 70]]
> readdata("data.txt", integer) # чтение целых первой колонки
                                              [1, 1, 2, 2]
> readdata("data.txt", [integer, integer, float]) # чтение колонок в разных форматах
                                  [[1, 1, 50.], [1, 2, 55.], [2, 1, 55.], [2, 2, 70.]]
```

Запись числовой информации в текстовый файл

- **writedata(fileID, data)**
- **writedata(fileID, data, format)**
  fileID – описатель файла (переменная)
  data - список, вектор или матрица
  format – формат данных **integer, float, string** или список этих типов

```
> currentdir("d:\\");# файл будет там
> A := array([[1.5, 2.2, 3.4], [2.7, 3.4, 5.6], [1.8, 3.1, 6.7]])
```

$$A := \begin{bmatrix} 1.5 & 2.2 & 3.4 \\ 2.7 & 3.4 & 5.6 \\ 1.8 & 3.1 & 6.7 \end{bmatrix}$$

```
> writedata(terminal, A, integer)
1       2       3
2       3       5
1       3       6

> writedata("data.txt", A, integer)# перезапись имеющегося файла
```

Теперь содержимое файла data.txt:

1 2 3
2 3 5
1 3 6

```
> writedata("data1.txt", A, float)
```

Файл data1.txt содержит данные :

1.5 2.2 3.4

2.7 3.4 5.6

1.8 3.1 6.7

Чтение строк из текстового файла: использование команды readline

```
> restart, currentdir("d:\\") :# файл будет там
Содержимое файла data.txt:
1 2 3
2 3 5
1 3 6
> f := "data.txt" :
> first := readline(f); #чтение первой строки
                                        first := "1  2  3"
> second := readline(f); #чтение следующей строки
                                        second := "2  3  5"
> third := readline(f); #чтение следующей строки
                                        third := "1  3  6"
> restart, currentdir("d:\\") :f := "data.txt" : line := readline(f);
  while line ≠ 0 do line := readline(f) :end do; # чтение строк до конца
                                        line := "1  2  3"
                                        line := "2  3  5"
                                        line := "1  3  6"
                                        line := 0
```

Запись строк в текстовый файл: использование команд writeline и fprintf

```
> f1 := fopen("d:\\data2.txt", WRITE, TEXT);
  writeline(f1, "Строка", "Еще строка") # создание нового файла и запись в него
  fclose(f1);
```

data2 — Блокнот

```
Файл  Правка  Формат  Вид  Справка
Строка
Еще строка
```

fprintf(file, fmt, x1, ..., xn) – запись выражений в файл на основе заданного формата строки

Некоторые форматы:

d – целое, f – с плавающей точкой

```
> restart, currentdir("d:\\") :
> x := 3 : y := 12356 :
> fd := fopen("temp_file", WRITE, TEXT); fprintf(fd, "x = %d, y = %f", x, y); fclose(fd)
```

temp_file — Блокнот

```
Файл  Правка  Формат  Вид  Справка
x = 3, y = 12356.000000
```

Maplets

- Maplets – графические приложения пользователя
- Создание Maplets с помощью Maplet Builder
- Создание Maplets с помощью команд пакета Maplets

Общие сведения о Maplet

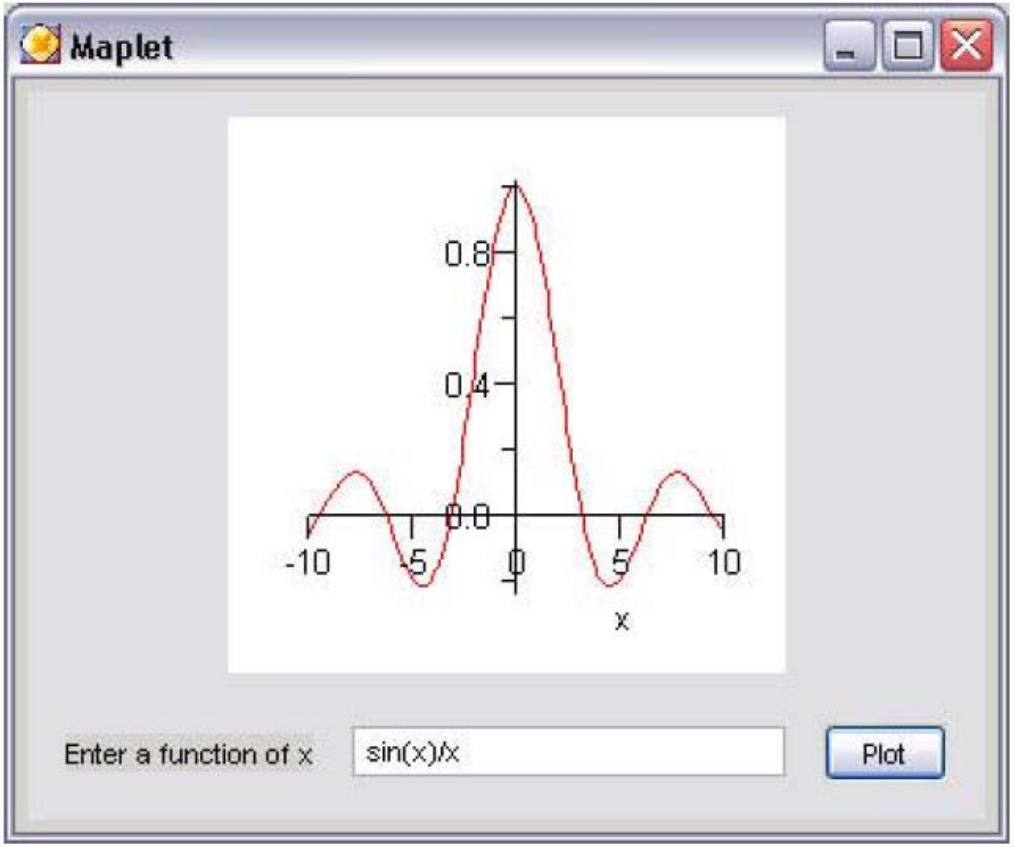
- Maplet – графическое приложение пользователя, созданное с помощью команд Maple
- Maplet можно создать
  - С помощью ассистента Maplet Builder (Tools->Assistants->Maplet Builder)
  - С помощью команд пакета Maplets
- Maplet можно запустить в командной строке Maple или с помощью отдельного приложения Maple MapletLauncher

Пример простого maplet, который выводит сообщение “Hello World”

```
> with(Maplets[Elements]):
> MySimpleMaplet := Maplet([["Hello World"]]):
> Maplets[Display](MySimpleMaplet);
>
```

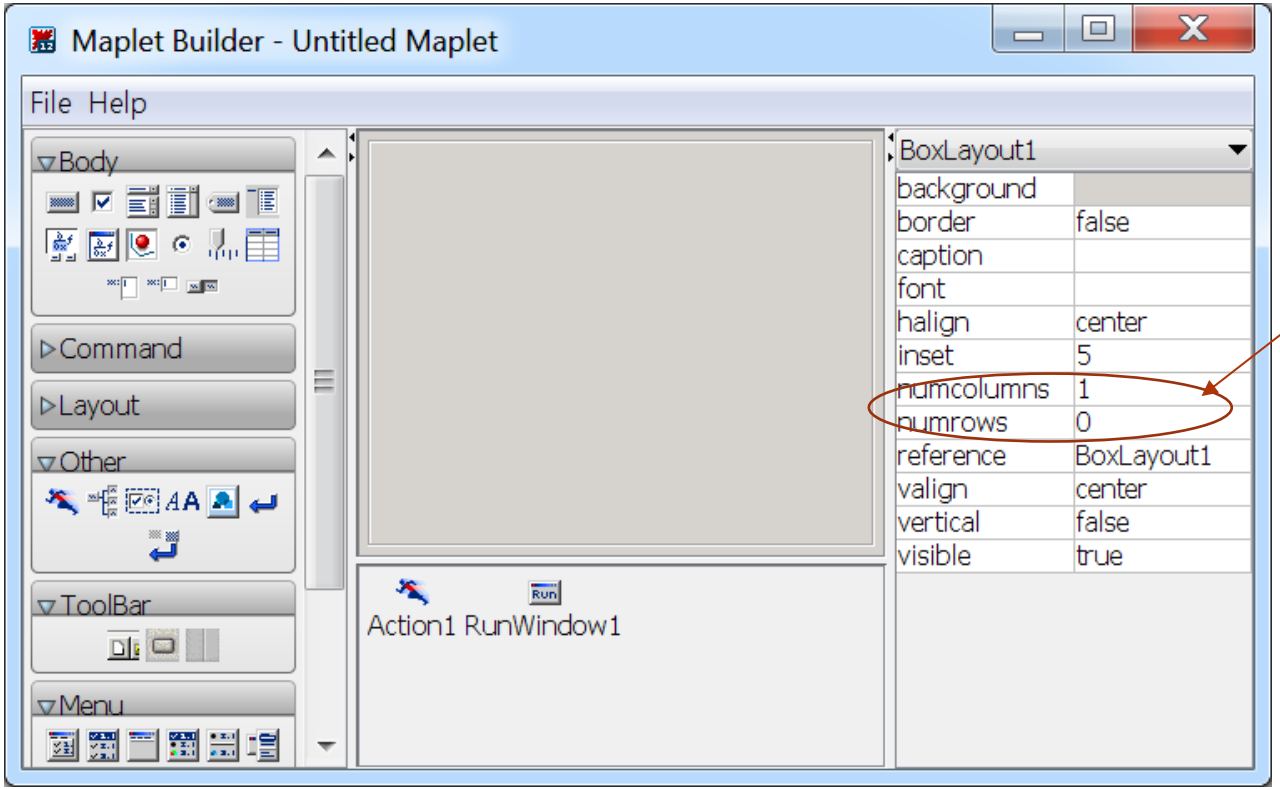
Пример создания maplet с помощью Maplet Builder: функция и ее график

- Готовый maplet: в поле ввода пользователь вводит функцию одной переменной, при нажатии на кнопку Plot в поле вывода появляется график функции



Создание maplet с помощью Maplet Builder

- Tools->Assistants->Maplet Builder

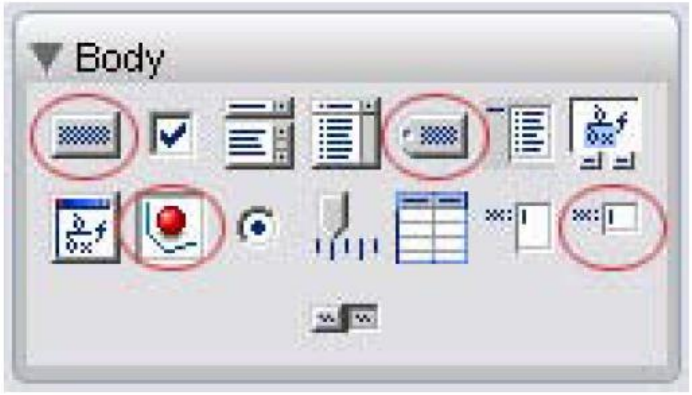


В пустом maplet 1 колонка и 0 строк

- Создание maplet начинается с задания числа полей (элементов) и их графического расположения

Элементы (поля) maplet

- Для создания этого maplet было использовано 4 элемента:

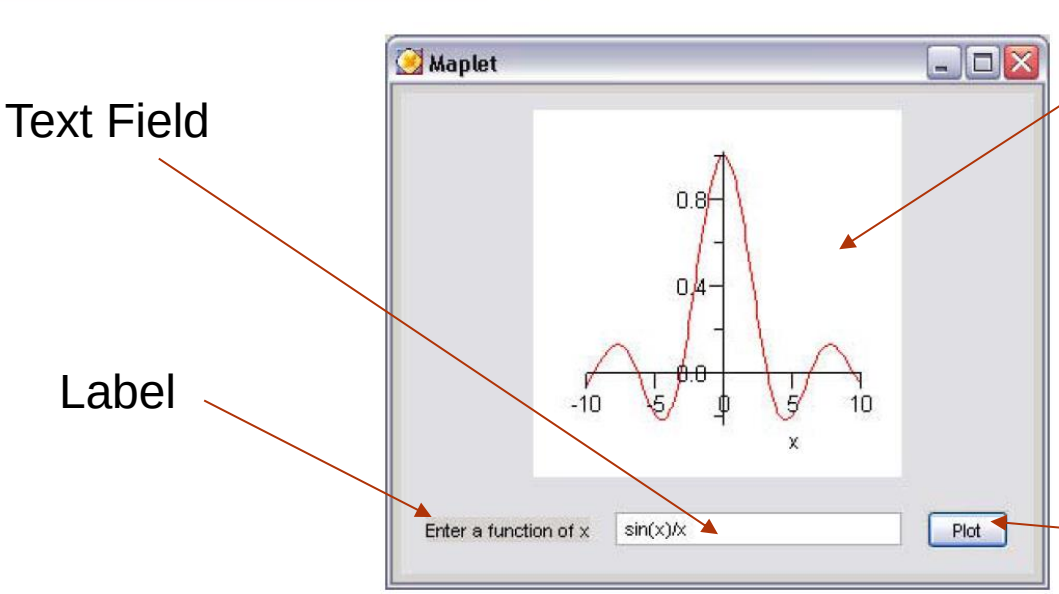


Button element

Label element

Plotter element

TextField element



Plotter

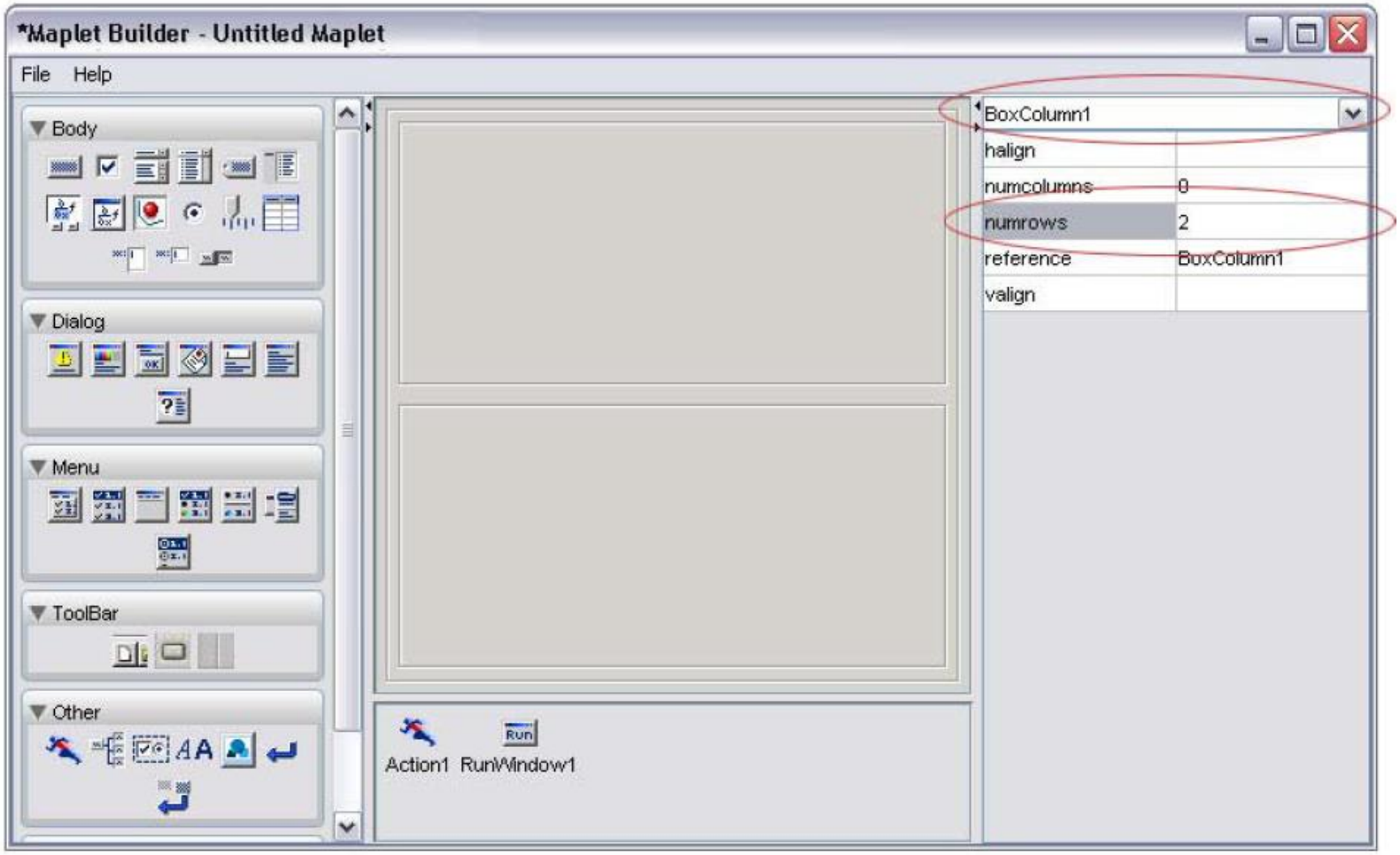
Text Field

Label

Button

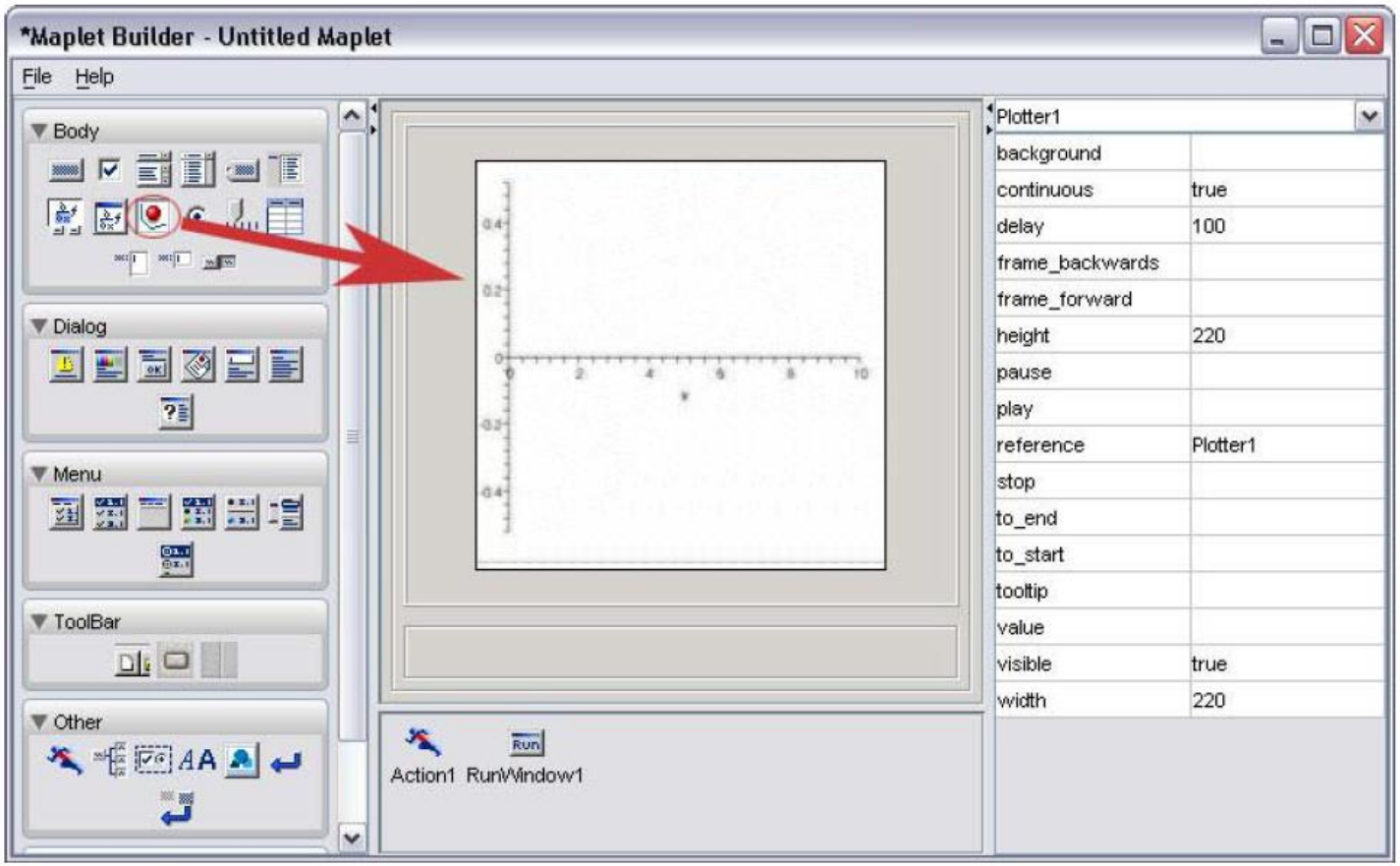
Пример создания maplet с помощью Maplet Builder: шаг 1

- Шаг 1: задать число строк в maplet: в имеющейся первой колонке – 2 строки



Пример создания maplet с помощью Maplet Builder: шаг 2

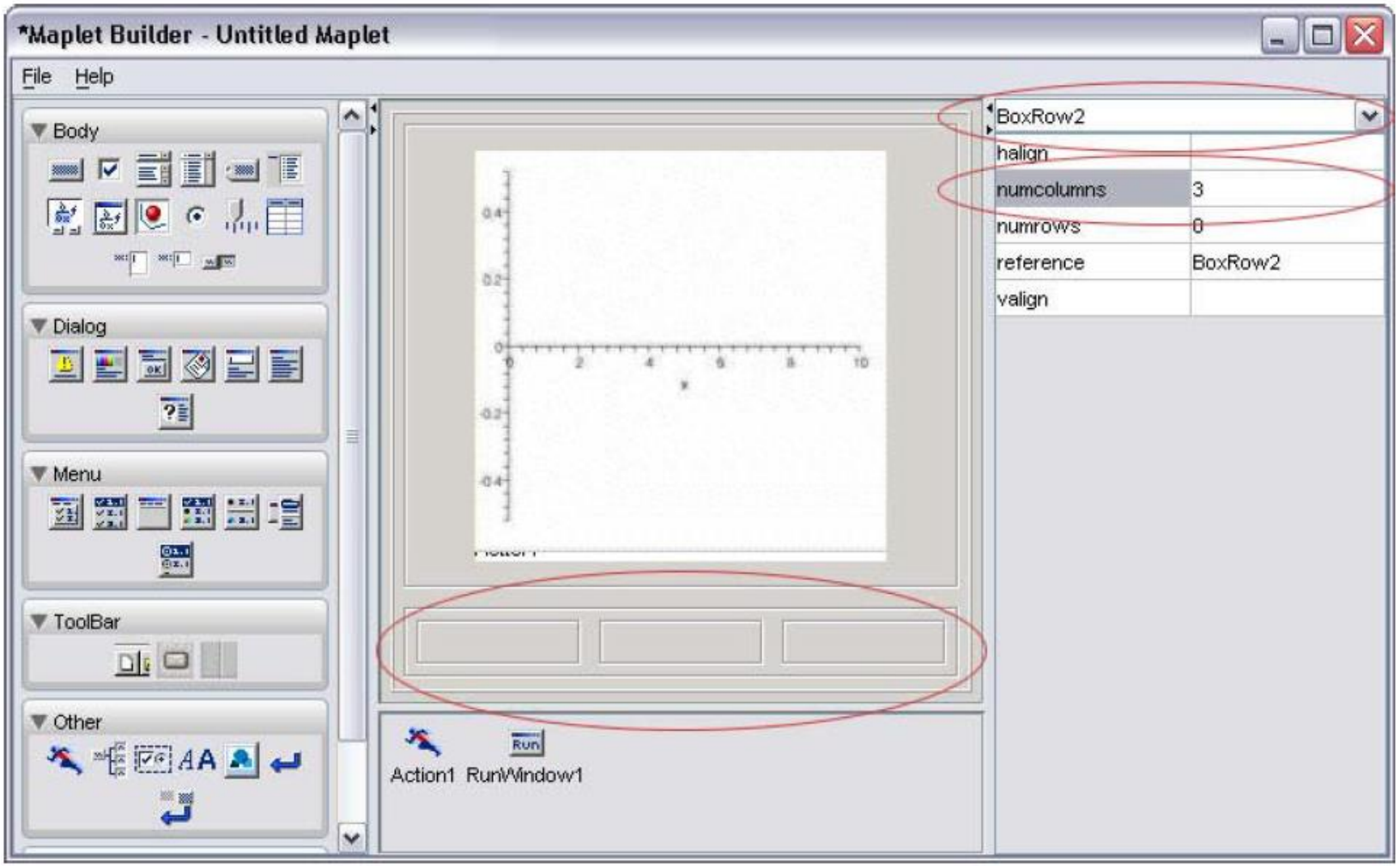
- Шаг 2: добавить элемент plotter в первую строку



- Как удалить элемент: выделить и нажать Del на клавиатуре

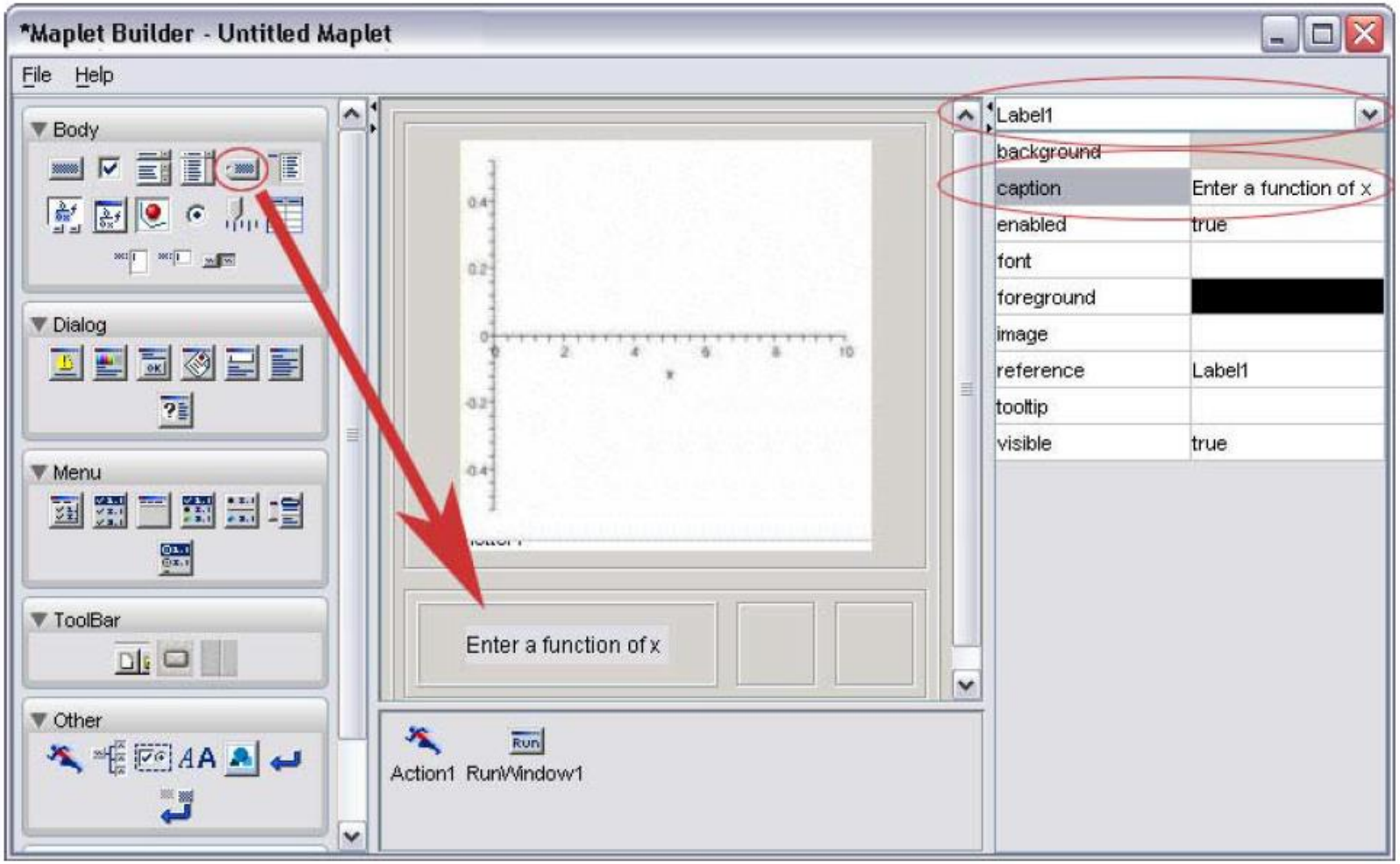
Пример создания maplet с помощью Maplet Builder: шаг 3

- Шаг 3: добавить 3 колонки во вторую строку



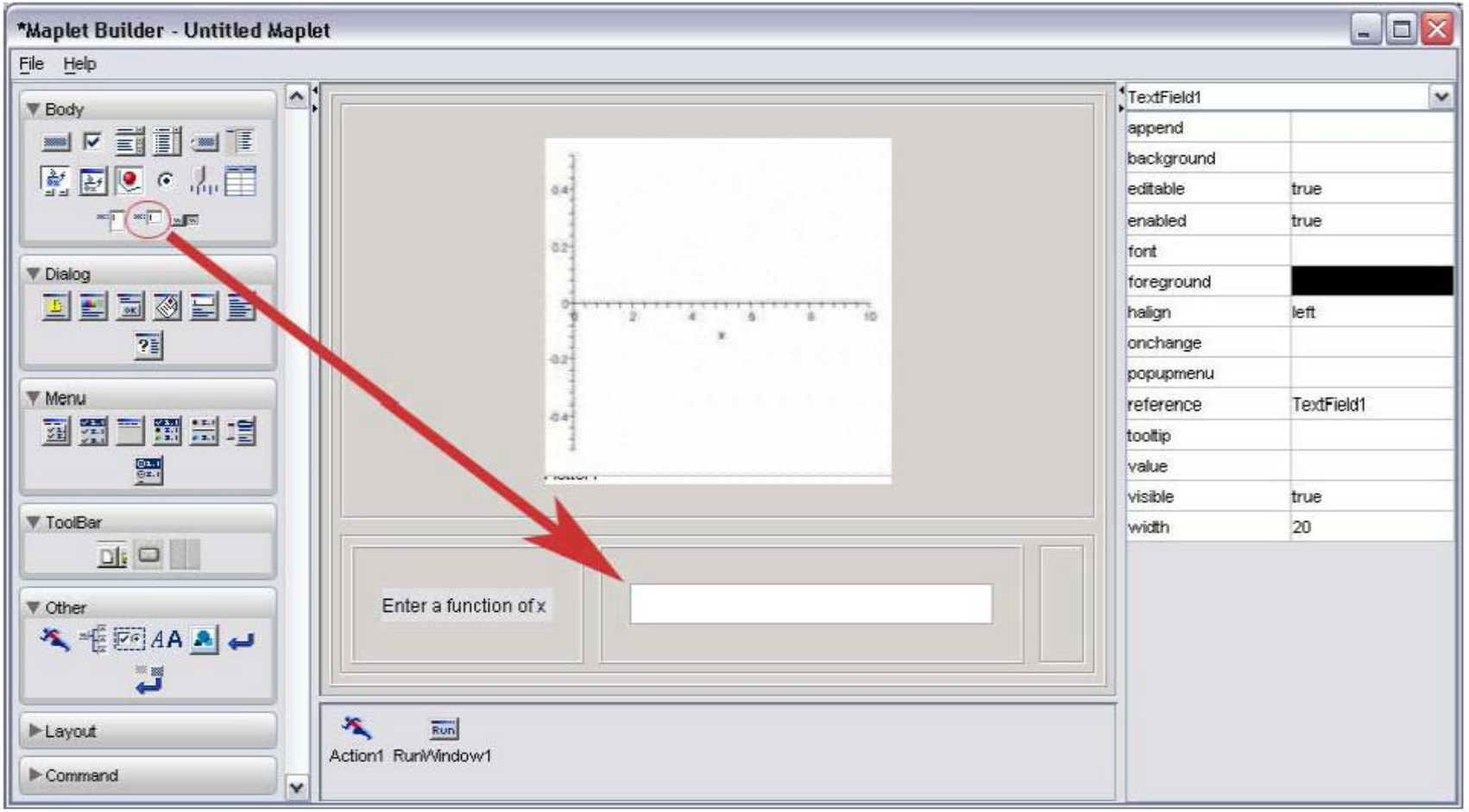
Пример создания maplet с помощью Maplet Builder: шаг 4

- Шаг 4: добавить элемент label во вторую строку



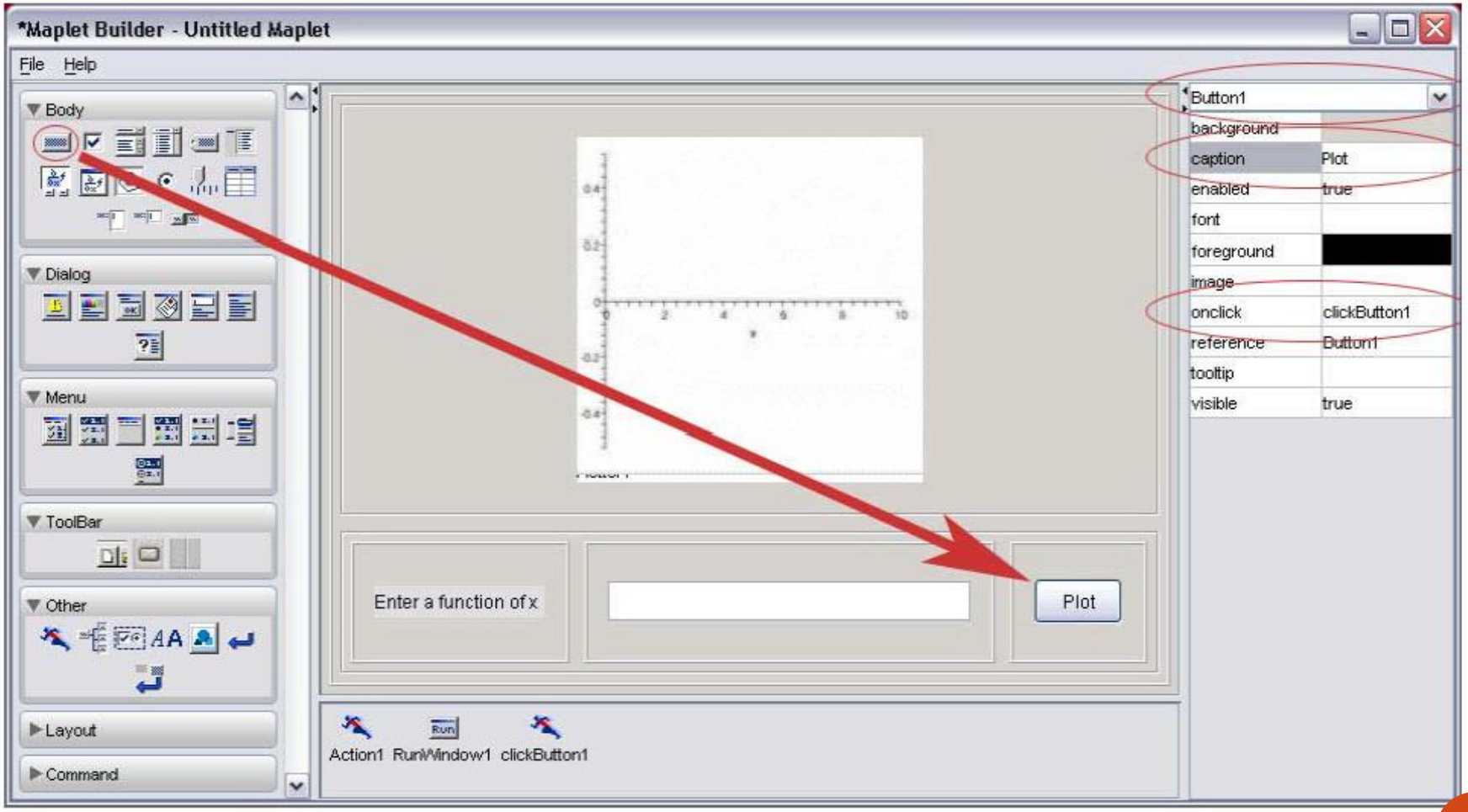
Пример создания maplet с помощью Maplet Builder: шаг 5

- Шаг 5: добавить элемент текстового поля (Text Field) во вторую строку



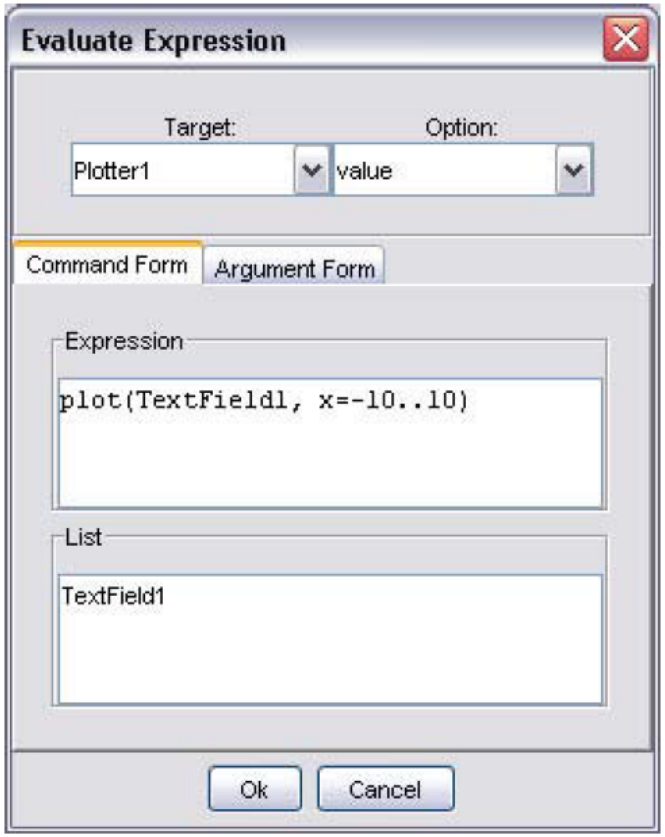
Пример создания maplet с помощью Maplet Builder: шаг 6

- Шаг 6: добавить элемент button во вторую строку



Пример создания maplet с помощью Maplet Builder: шаги 7 и 8

- Шаг 7: в открывающемся окне Evaluate Expression задать



- Шаг 8: Запуск приложения: File->Run (будет предложено сохранить Maplet)

Пример создания maplet с помощью команд пакета Maplets: функция и ее график

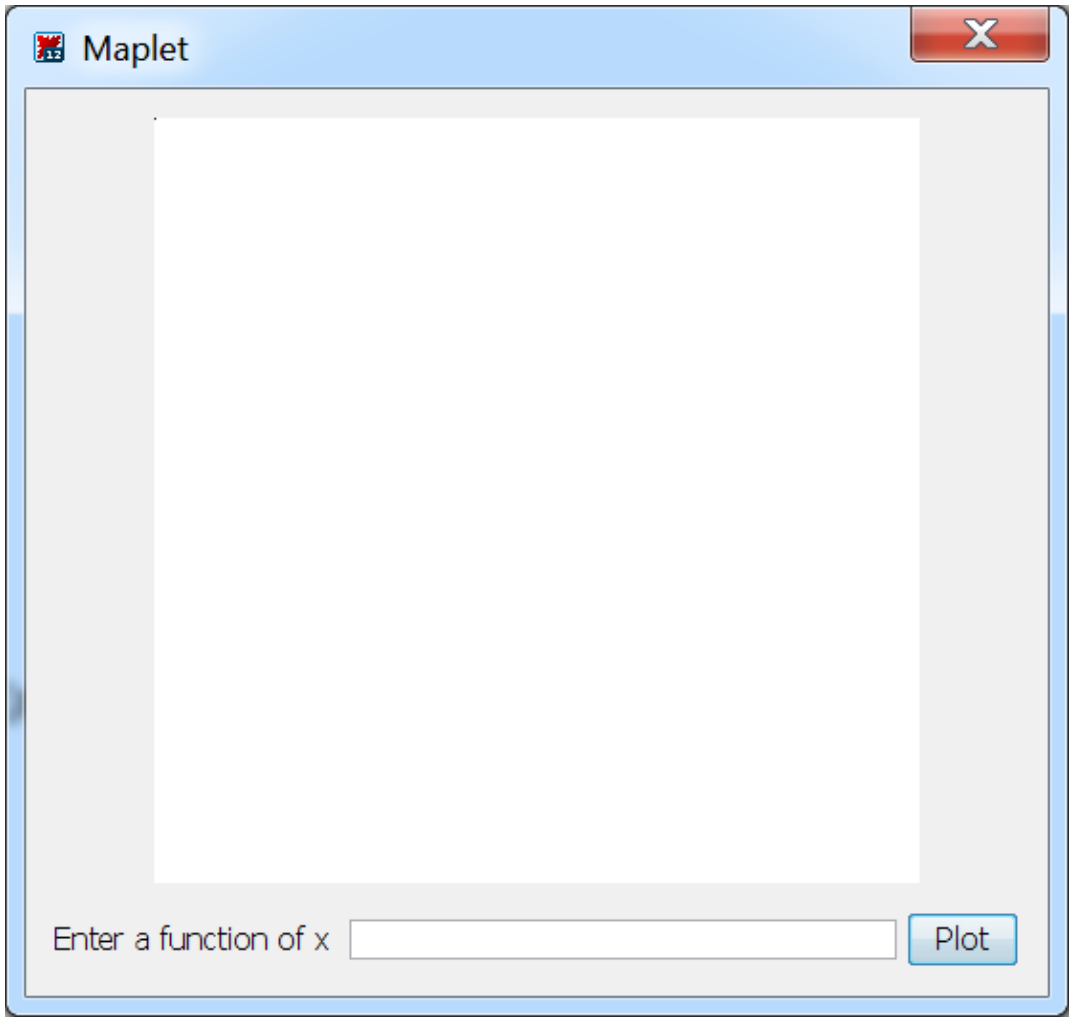
```
> with(Maplets[Elements]) :
> PlottingMaplet := Maplet(
  BoxLayout(
  BoxColumn(
  # First Box Row
  BoxRow(
  # Define a Plot region
  Plotter('reference' = Plotter1)
  # End of first Box Row
   ),
  # Second Box Row
  BoxRow(
  # Define a Label
  Label("Enter a function of x "),
  # Define a Text Field
  TextField('reference' = TextField1),
  # Define a Button
  Button(caption = "Plot", Evaluate(value = 'plot(TextField1, x = -10 ..10)', 'target' = Plotter1))
  # End of second Box Row
   )
  # End of BoxColumn
   )
  # End of BoxLayout
   )
   # End of Maplet
   ) :
```

Подпакеты пакета Maplets:

- Elements
- Tools
- Utilities

Запуск созданного maplet с помощью команды Display пакета Maplets

```
> Maplets[Display](PlottingMaplet);
```



Результат