Pengenalan I: Operasi, Variabel, dan Matriks

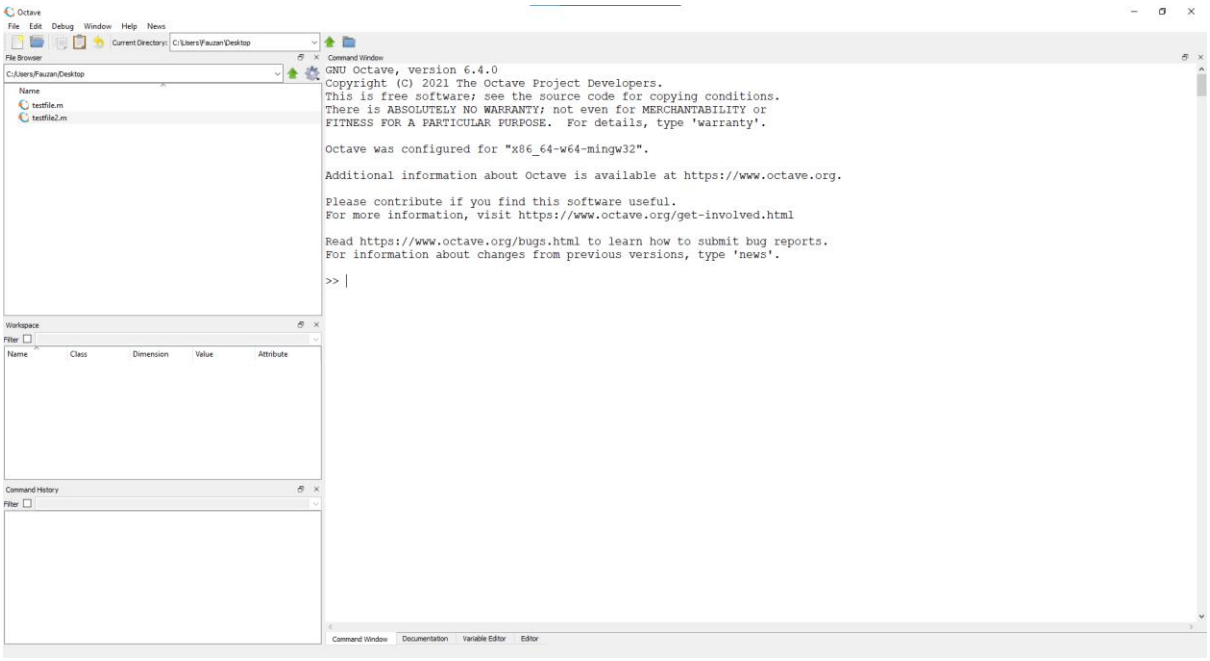
Pengenalan

Octave adalah *software* dan bahasa pemrograman yang umum digunakan dalam analisis numerik. *Syntax* pada Octave kompatibel dengan MATLAB, bahkan *script file* dari Octave juga menggunakan ekstensi yang sama (.m). Pada praktikum ini, akan dijelaskan tentang penggunaan Octave dan pengaplikasiannya dalam Persamaan Diferensial Numerik.

Pastikan Octave telah terinstal pada computer kalian. Kalian dapat mengunduh Octave di https://www.gnu.org/software/octave/download.

Overview

Setelah terinstal, akan ada dua jenis *launcher*, "Octave-6.4.0 (Local) (GUI)" dan "Octave-6.4.0 (Local) (CLI)". Jalankan "Octave-6.4.0 (Local) (GUI)", lalu akan muncul jendela seperti ini:



Secara default, Terdapat 4 sub-jendela, yaitu *File Explorer*, *Workspace*, *Command History*, dan *Command Window* (dll.)

- *File Explorer* berisi *path* yang dituju pada *address bar*. *File Explorer* berguna untuk membuka file .m ataupun file-file lain tanpa harus membuka Windows Explorer.
- *Workspace* berisi data tentang variabel yang didefinisikan saat menjalankan pemrograman nantinya. Data-data yang disajikan berupa identitas dari tiap variabel, seperti nama, kelas, dimensi, *value*, dan atribut.

- *Command History* berisi Riwayat dari semua *command* yang dijalankan pada *Command Window*.
- *Command Window* sebenarnya mempunyai 3 sub-jendela lain yang dapat ditukar ataupun ditumpuk.
    - *Command Window* adalah *command line* utama dari Octave dan digunakan untuk menjalankan semua kode maupun program dari Octave. Untuk menjalankan kode dari *Command Window*, ketikkan kode yang diinginkan, lalu tekan *Enter*.
    - *Documentation* berisi dokumentasi dari Octave, seperti *tutorial*, detail dari fungsi-fungsi *built-in*, penerapan, dll.
    - *Variable Editor* berguna untuk mengubah variabel yang ada di *Workspace*. *Variable Editor* berbentuk baris dan kolom sehingga memudahkan untuk membuat matriks.
    - *Editor* adalah jendela *scripting* utama dari Octave. *Script* yang dibuat di *Editor* dapat disimpan dalam file ekstensi .m dan dijalankan dengan *Command Window* ataupun dengan menekan F5.

Aritmatika Standar

Octave dapat digunakan untuk kalkulasi numerik dasar. Octave mengenal operator aritmatika (`+`, `-`, `*`, `/`), operator pangkat (`^`) (berbeda dengan kebanyakan Bahasa pemrograman), fungsi eksponen dan logaritma (`exp`, `log`), dan fungsi trigonometri (`sin`, `cos`, …). Kalkulasi pada Octave juga bekerja pada bilangan real ataupun imajiner (I, j). Beberapa konstanta, seperti bilangan Euler (`e`) dan bilangan pi (`pi`), sudah *pre-defined* pada Octave. Anda juga dapat menambahkan komentar pada kode dengan menggunakan persen (`%`).

```
>> 1 + 2 %Operasi Aritmatika Standar
ans = 3
```

Terlihat bahwa *value* hasil *running* disimpan dalam suatu variabel yang bernama `ans`. Variabel ini akan menyimpan seluruh hasil dari ekspresi yang di-*input* dan akan ditimpa jika ada ekspresi baru yang menyimpan hasil.

Ekspresi tersebut juga dapat disimpan ke suatu variabel.

```
>> a = 2 + 3
a = 5
```

Terlihat bahwa *output* menunjukkan variabel `a`, bukan `ans`.

Jika tidak ingin *print* hasilnya, beri titik koma (;) pada akhir *line*…

```
>> a = 1 + 3;
```

… dan jika ingin menampilkan lagi outputnya, cukup memanggil variabelnya.

```
>> a
a = 4
```

Untuk *update* variabel, dapat menggunakan cara ini:

```
>> a = a + 6 %Variabel a ditambah 6
a = 10
>> a = a / 5 %Dapat disesuaikan dengan operator lainnya
a = 2
>> a -= 6 %Idem
a = -4
>> a *= -5 %Idem
a = 20
>> a = "ayam" %Update variabel a dengan nilai baru
a = ayam
```

Sebagai tambahan, tidak seperti Mathematica, *value* pada Octave menggunakan presisi mesin (*machine precision*), sehingga beberapa hasil yang ditampilkan mungkin saja tidak bulat. Sebagai contoh, akan ditampilkan hasil dari $e^{i\pi}$ (Identitas Euler)

```
>> e^(i*pi)
ans = -1.0000e+00 + 1.2246e-16i
```

Matriks I: Pendefinisian dan Pemanggilan Indeks

Matriks ataupun vektor akan menjadi hal esensial untuk analisis numerik. Untuk mendefinisikan matriks, gunakan kurung siku (`[]`) untuk membuat matriks. Elemen pada baris dipisahkan dengan koma (`,`), dan elemen pada kolom dipisahkan dengan titik koma (`;`)

Contoh:

```
>> A = [1, 2, 3; 4, 5, 6; 7, 8, 9]
A =

   1   2   3
```

```
   4   5   6
   7   8   9
```

Catatan: Sebenarnya kita dapat mendefinisikan kode untuk matriks tanpa menggunakan koma ataupun titik koma. Cukup dengan menggunakan spasi dan *Enter*…

```
>> B = [1 2 3
4 5 6
7 8 9]
B =

   1   2   3
   4   5   6
   7   8   9
```

… namun hal ini dapat menyebabkan beberapa ambiguitas dalam pendefinisian beberapa elemen dalam matriks. Disarankan tetap menggunakan titik dan titik koma, kecuali untuk matriks yang isinya simpel.

Selanjutnya, untuk memanggil elemen dari matriks, gunakan tanda kurung (`(a, b)`) dengan `a` adalah indeks baris dan `b` adalah indeks kolom.

Catatan: Indeks pada Octave dimulai dari 1.

```
>> A(2, 3)
ans = 6
```

Jika ingin memanggil lebih dari satu elemen, kalian dapat menggunakan titik dua (`a:b`) ataupun kurung siku (`[a,b]`). Titik dua (`a:b`) akan memanggil elemen baris/kolom dari `a` hingga `b`…

Contoh: Memanggil elemen baris ke-2 dan kolom dari 2 hingga 3:

```
>> A(2, 2:3)
ans =

   5   6
```

… dan kurung siku (`[a,b]`) akan memanggil elemen baris/kolom ke-`a` dan ke-`b`.

Contoh: Memanggil elemen baris ke-1 dan kolom ke-1 dan ke-3:

```
>> A(1, [1,3])
ans =

   1   3
```

Contoh: Memanggil elemen baris ke-1 dan ke-3, dan kolom dari 2 hingga 3:

```
>> A([1,3], 2:3)
ans =

   2   3
   8   9
```

Matriks II: Operasi Matriks

Matriks yang telah dibuat dapat diubah isinya, ditranspos, diinvers, dll. Untuk mengubah isi dari matriks, dapat menggunakan pemanggilan indeks.

```
>> A(2, 3) = 5 %Mengubah satu elemen
A =

   1   2   3
   4   5   5
   7   8   9

>> A(2, [1,3]) = 3 %Mengubah banyak elemen dengan 1 nilai
A =

   1   2   3
   3   5   3
   7   8   9

>> A(1:3, 1) = [2; 4; 8] %Mengubah banyak elemen dengan nilai
                          yang bersesuaian dengan tempat
A =

   2   2   3
   4   5   3
   8   8   9

>> A(1:3, 3) = [2, 4, 8] %Jika posisi salah, akan dicoba
                          dengan transpose nya
```

```
A =

   2   2   2
   4   5   4
   8   8   8

>> A(2, 1:3) = [2, 4] %Jika masih tidak bisa dengan transpose
                        nya, akan ada pesan error
error: =: nonconformant arguments (op1 is 1x3, op2 is 1x2)
```

Operasi matriks pada Octave mengikuti operasi pada aljabar linear biasa, dimana perkalian matriks harus mengikuti baris dan kolom yang sesuai.

```
>> A * 2 %Perkalian scalar
ans =

    4    4    4
    8   10    8
   16   16   16

>> A + B %Penjumlahan matriks
ans =

    3    4    5
    8   10   10
   15   16   17

>> A * B %Perkalian matriks
ans =

    24    30    36
    52    65    78
    96   120   144
```

Untuk memanggil transpos dari matriks, gunakan tanda petik satu (`'`), dan untuk memanggil invers dari matriks, gunakan fungsi `inv`. Jika matriks yang diinvers singular ($\det = 0$), maka akan muncul pesan peringatan dan elemennya akan menjadi `Inf`.

```
>> D = [1, 2, 3; 4, 5, 6]
D =
```

```
   1   2   3
   4   5   6

>> D' &Transpos dari matriks
ans =

   1   4
   2   5
   3   6

>> E = [1, 2; 5, 7]
E =

   1   2
   5   7

>> inv(E) &Invers dari matriks
ans =

  -2.3333   0.6667
   1.6667  -0.3333

>> F = [1, 2; 4, 8]
F =

   1   2
   4   8

>> inv(F) %Contoh invers untuk matriks singular
warning: matrix singular to machine precision
ans =

   Inf   Inf
   Inf   Inf
```

Octave juga mempunyai operator khusus, yaitu (\). Operator ini adalah operator "pembagian" matriks. Pendefinisian untuk `A \ b` ekuivalen dengan `inv(A) * b`. Operasi ini sangat berguna untuk penyelesaian sistem linear. Sebagai contoh, misalkan kita mempunyai sistem linear berikut:

$$4x_1 - 2x_2 = 20$$

$$-5x_1 - 5x_2 = -10$$

Jika kita ubah dalam bentuk perkalian matriks $Ax = b$, diperoleh:

$$A = \begin{bmatrix} 4 & -2 \\ -5 & -5 \end{bmatrix}$$
$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$
$$b = \begin{bmatrix} 20 \\ -10 \end{bmatrix}$$

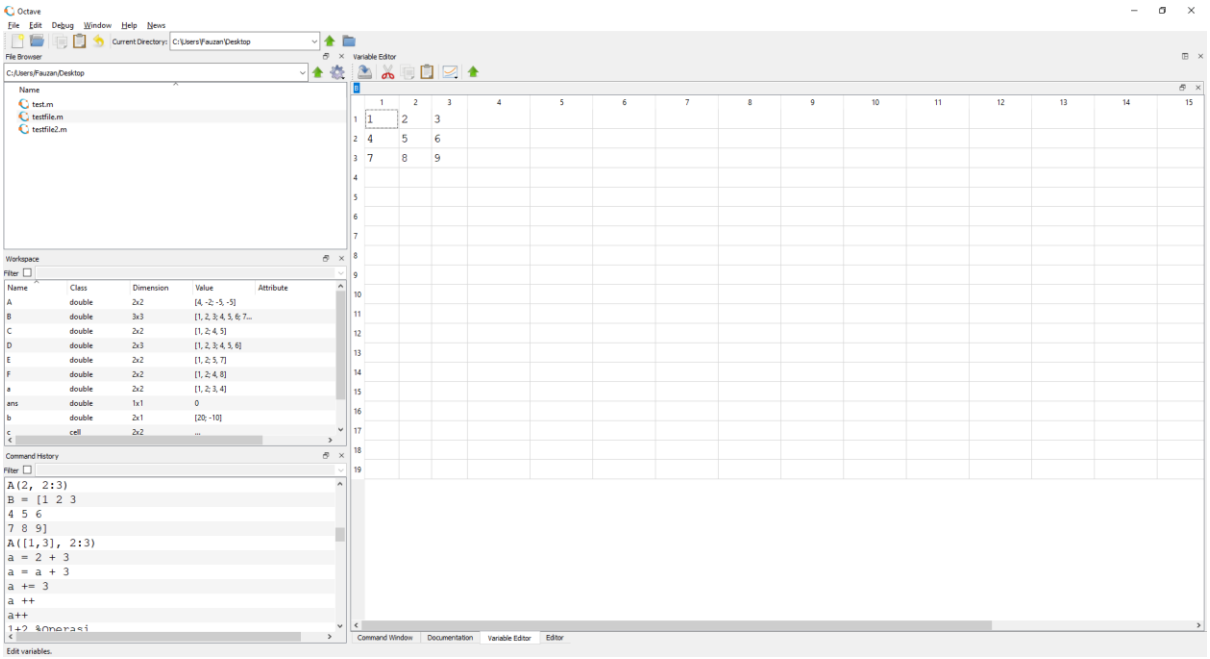
Jika dimasukkan ke Octave, kita akan memperoleh:

```
>> A = [4, -2; -5, -5]
A =

   4  -2
  -5  -5

>> b = [20; -10]
b =

   20
  -10

>> x = A \ b
x =

   4
  -2
```

Diperoleh $x_1 = 4$ dan $x_2 = -2$.

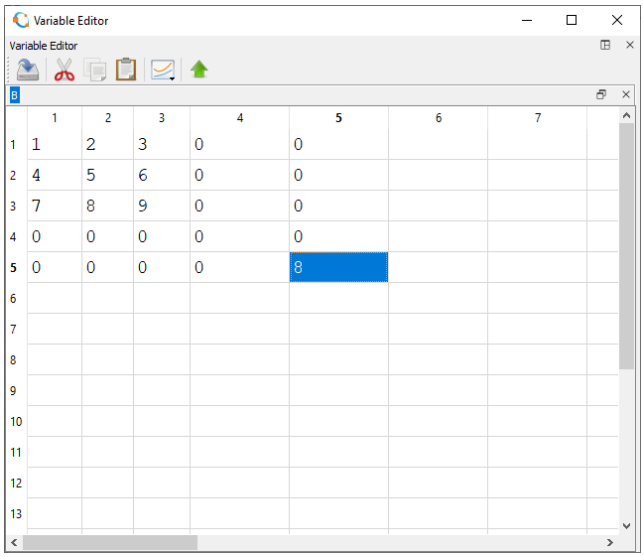
Supplementary A: Variable Editor

Sesuai namanya, *Variable Editor* digunakan untuk mengubah nilai dari suatu variabel. Untuk menggunakannya, *double-click* variabel yang ingin diubah pada *workspace*. Variabel tersebut akan muncul pada tab *Variable Editor* (atau tekan Ctrl+6).



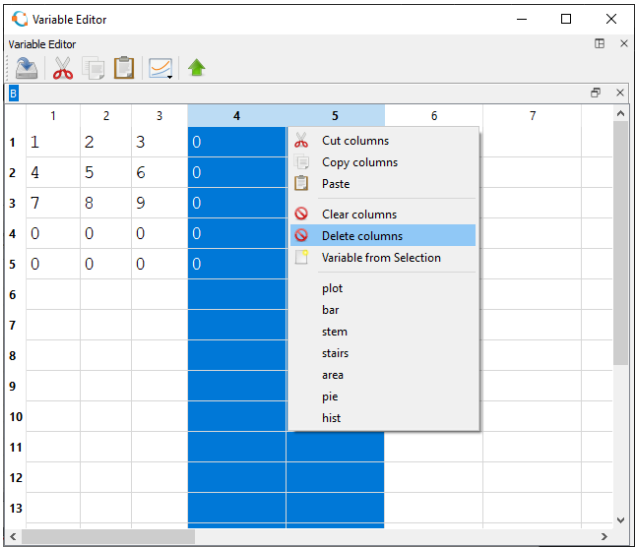
Antarmuka dari *Variable Editor* mirip seperti Excel, sehingga kalian bisa langsung mengubah nilai matriks yang ada dengan memilih kotak yang berisi elemen yang ingin diubah, lalu mengetik nilai yang baru.

Jika kalian memberi nilai baru pada kotak kosong di *Variable Editor*, matriks tersebut akan mengalami perubahan bentuk, menyesuaikan dengan posisi elemen nilai baru tersebut.



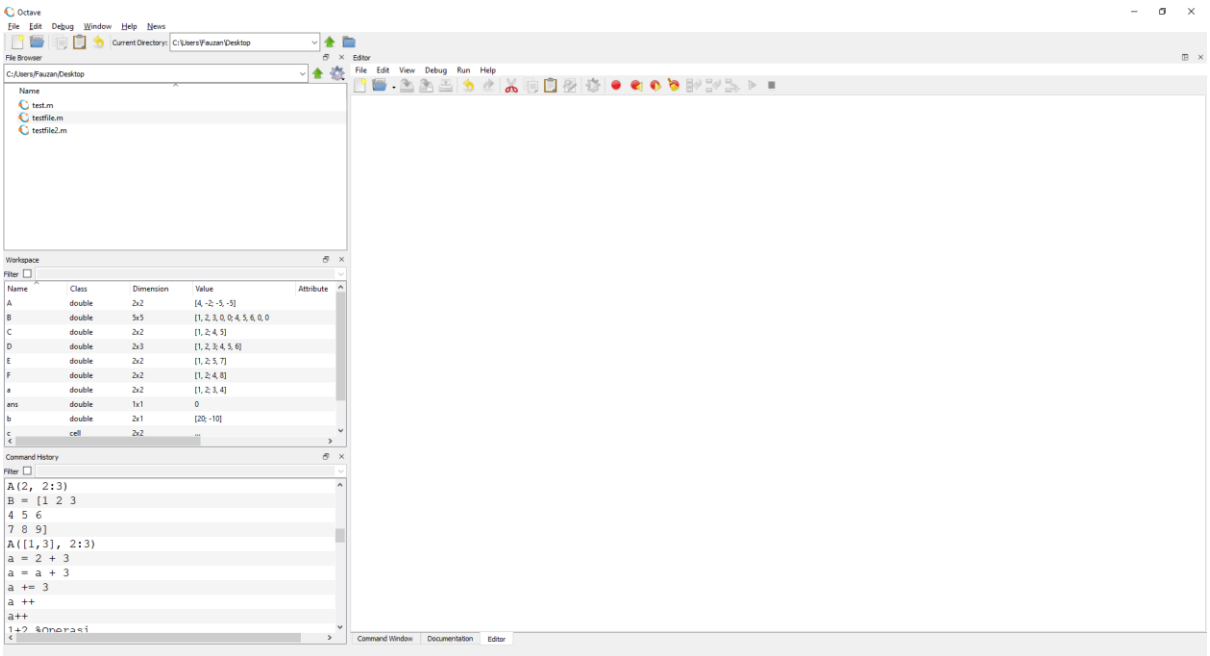
Pada gambar di atas, nilai baru dimasukkan pada baris 5 kolom 5, sehingga matriks tersebut berubah menjadi 5x5.

Untuk menghapus suatu baris/kolom, klik kanan angka baris/kolom (atau pilih lebih dari satu baris/kolom dengan Ctrl/Shift), lalu klik *Delete rows/Delete columns*.



Supplementary B: Editor

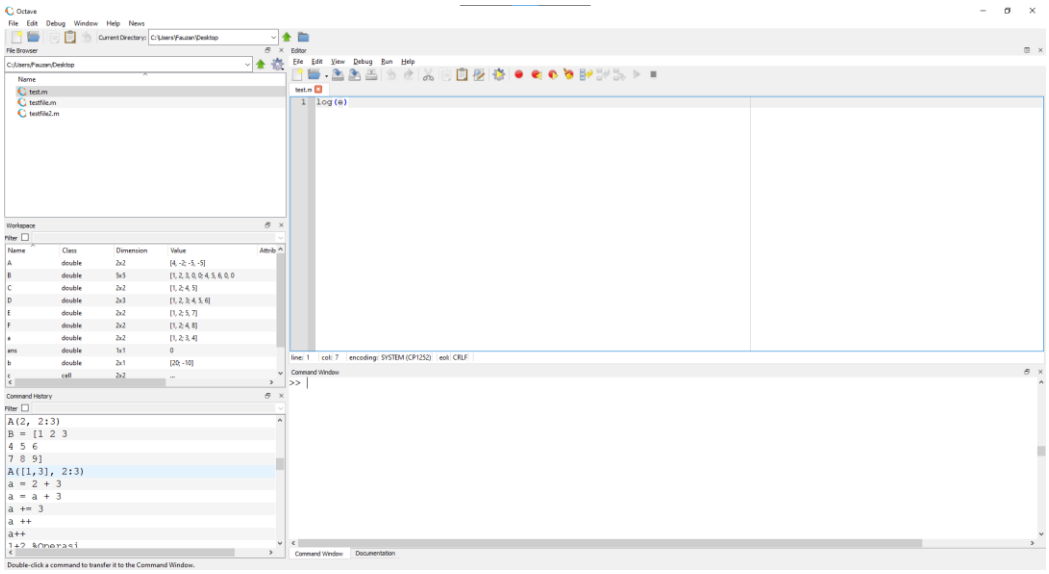
Sejauh ini, kita menggunakan *command line* untuk menjalankan kode. Namun, kalian juga bisa melakukan *scripting* seperti pemrograman pada umumnya dengan menggunakan *Editor*. Untuk memunculkan *Editor*, klik Tab *Editor* (atau tekan Ctrl+4).



Untuk membuat file baru, klik *New Script* (ikon kertas). Kalian juga dapat membuka file .m (file MATLAB) yang tersimpan dengan *double-click* file tersebut di File Explorer pada Octave ataupun di Windows Explorer (atau /root atau lainnya).

Script bekerja layaknya Bahasa pemrograman biasa. Kalian dapat menjalankan *script* yang telah dibuat dengan menekan F5 (jika file belum di-*save*, akan muncul kotak dialog untuk *save*). *Output* dari *script* tersebut akan muncul pada *Command Window*.

Catatan: Jika kalian menggunakan E*ditor*, disarankan mengubah *layout* dari *Editor* dan *Command Window* untuk mempermudah melihat *output* (tidak perlu ganti tab). Untuk memindahkan *layout*, klik dan *drag* tulisan *Editor* pada bagian atas jendela *Editor* ke tempat yang diinginkan. Contohnya seperti gambar berikut:



Pengenalan II: *Programming* dan Fungsi

Programming I: Input

Dalam pemrograman, seringkali pengguna diminta memberi suatu *input*, entah suatu nilai, *string*, dll., ke program, lalu program tersebut akan menggunakan *input* tersebut sebagai nilai dari suatu variabel. Hal ini juga dapat dilakukan pada Octave. Untuk membuat Octave meminta *input* dari *user*, gunakan *syntax* `input(prompt)`, dengan `prompt` adalah *string* yang berisi pesan dalam *input*.

```
>> A = input("Masukkan suatu angka: ")
Masukkan suatu angka: 135
A = 135
```

Jika tidak ingin membuat pesan input, cukup isi "" sebagai `prompt`.

```
>> A = input("")
100
A = 100
```

Perlu diketahui bahwa *input* yang diberikan pengguna akan dievaluasi sebagai ekspresi. Jadi, bisa saja *input* yang diberikan akan dievaluasi sebagai kode Octave. Sebagai contoh, jika kita memasukkan operasi bilangan pada *input*…

```
>> B = input("Operasi bilangan: ")
Operasi bilangan: 2 + 3
B = 5
```

…, maka operasi tersebut akan dievaluasi dan memberikan hasil operasinya. Jika kita memasukkan kode Octave, seperti meng-*assign* suatu variabel…

```
>> C = input("Assign variabel: ")
Assign variabel: x = 25
C = 25
```

…, maka nilai dari variabel yagn di-*assign* akan masuk ke variabel *input*…

```
>> x
x = 25
```

… sekaligus variabel yang di-*assign* di dalam *input*. Jika kalian ingin agar *input* yang dimasukkan tidak dievaluasi, *input* tersebut dapat diubah terlebih dahulu menjadi *string*.

```
>> D = input("Masukkan string: ")
Masukkan string: "x + 25"
```

```
D = x + 25
>> typeinfo(D) % untuk menentukan tipe data variabel
ans = sq_string % menunjukkan tipe data string
```

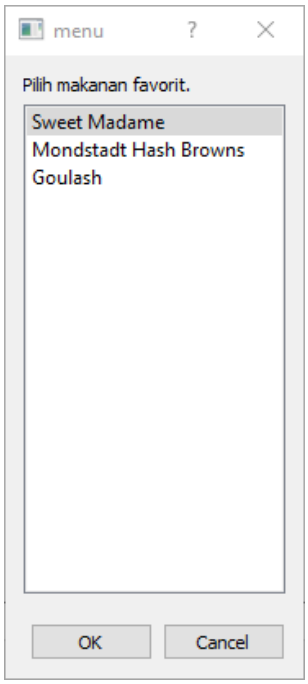
Bisa juga dengan menambah argumen pada `input()` menjadi `input(prompt, "s")`. Jika menambahkan argumen, maka apapun *input* yang kalian masukkan akan menjadi *string* tanpa perlu menggunakan tanda petik.

```
>> E = input("Masukkan string: ", "s")
Masukkan string: x + 25
E = x + 25
>> typeinfo(E)
ans = sq_string
```

Note: *Syntax* `input()` sebaiknya digunakan sebagai nilai dari suatu variabel.

Selain menggunakan `input()`, kita juga bisa menggunakan *syntax* `menu(title, op1, op2, ...)`. *Syntax* tersebut akan memunculkan kotak dialog dengan judul `title` dan pilihan `op1`, `op2`, dst. (sesuai yang dimasukkan). *Syntax* ini sangat berguna untuk program-program interaktif karena mempunyai GUI sendiri.

```
>> F = menu("Pilih makanan favorit.", "Sweet Madame",
"Mondstadt Hash Browns", "Goulash")
```




```
F = 1
```

Tergantung pilihan kalian, variabel yang mengandung `menu()` akan diisi bilangan dari 1 hingga n tergantung banyaknya pilihan. Dalam contoh di atas, pilihan "`Sweet Madame`"

akan memberi nilai 1, "Mondstadt Hash Browns" akan memberi nilai 2, dan "Goulash" akan memberi nilai 3. Jika kalian memilih *Cancel* atau membuat pilihan yang tidak valid, maka variabelnya akan diberi nilai 0.

Untuk *output*, mungkin cukup untuk memanggil variabel itu sendiri, seperti...

```
>> C
C = 25
```

..., namun kalian juga bisa hanya memunculkan nilai dari variabelnya tanpa sekaligus memunculkan variabel tersebut dengan menggunakan *syntax* `disp()`. *Syntax* ini digunakan jika yang di-*output* hanya suatu variabel atau *string* simpel, dll.

```
>> disp(C)
25
>> disp("Simple string")
Simple string
```

Jika yang ingin dimunculkan adalah pesan yang membutuhkan banyak *formatting*, kalian bisa menggunakan *syntax* `printf()`. *Syntax* tersebut dapat melakukan *formatting* pesan agar dapat menerima variabel selain *string*.

Note: Gunakan `\n` pada akhir *string* di `printf()` agar program memasukkan "Enter".

```
>> x = input("Masukkan angka: ");printf("Anda memasukkan
angka %d.\n", x)
Masukkan angka: 25
Anda memasukkan angka 25.
```

Pada contoh di atas, kita ingin agar variabel `x` dapat di-*output* bersama dengan pesan *string*. Kita menggunakan `%d` agar nilai `x` dapat di-*print* sebagai bilangan desimal. Jika variabelnya berisi *string*, maka gunakan `%s`. Jika variabelnya berisi *float*, gunakan `%f` untuk *print* dalam bentuk desimal, atau `%.nf` untuk sekaligus mengatur angka di belakang koma sebanyak `n`.

```
>> printf("pi = %f\n", pi);printf("pi = %.12f\n", pi)
pi = 3.141593
pi = 3.141592653590
```

Jika *float* tersebut ingin di-*print* dalam notasi saintifik, gunakan `%e` atau `%E`. Keduanya hanya berbeda di hasil *output* yang berupa E (besar) ataupun e (kecil).

```
>> printf("euler = %e\n", e);printf("euler = %E\n", e)
euler = 2.718282e+00
euler = 2.718282E+00
```

Jika ingin *print* karakter persen itu sendiri (%), gunakan `%%`.

Jika ada lebih dari satu *formatting* di satu `printf()`, maka variabelnya juga harus dimasukkan secara berurutan.

```
>> name = input("Masukkan nama: ", "s");
Masukkan nama: Ayato
>> fave = input("Minuman favorit: ", "s");
Minuman favorit: boba
>> price = input(["Harga " fave ": "]);
Harga boba: 30000
>> printf("%s suka %s seharga %d Mora.\n", name, fave,
price);
Ayato suka boba seharga 30000 Mora.
```

Programming II: Conditionals

Seperti halnya bahas pemrograman, Octave pun juga memiliki *conditional statements*. Secara umum, *conditional statement* pada Octave berbentuk:

```
cond
  body
endcond
```

Pada potongan kode di atas, `cond` adalah jenis *conditional statement* yagn digunakan, bisa berupa `if`, `for`, dan lainnya, `body` berisi kode yang dijalankan ketika `cond` terpenuhi, dan `endcond` adalah bagian penutup dari *conditional statement*, bisa berupa `endif`, `endfor`, dan lainnya tergantung `cond` apa yang digunakan.

Operasi dasar yang digunakan pada *conditional statements* adalah operasi perbandingan, dimana pada dasarnya, dua atau lebih nilai dibandingkan dengan operator dan dicek apakah memenuhi atau tidak. Jika memenuhi, maka nilainya 1, dan jika tidak, maka nilainya 0. Ada 6 operator dasar untuk perbandingan:

- sama dengan (==)
- lebih dari (>)
- kurang dari (<)
- lebih dari atau sama dengan (>=)

- kurang dari atau sama dengan (<=)
- tidak sama dengan (!= atau ~=)

```
>> 2 < 3
ans = 1
>> 4 == 5
ans = 0
```

Selain operator di atas, ada juga *syntax* untuk perbandingan:

- `isequal(a, b, c, ...)` mengecek apakah `a`, `b`, dan `c` semuanya sama.
- `strcmp(s1, s2)` mengecek apakah `s1` dan `s2` adalah *string* yang sama.
- `strncmp(s1, s2, n)` mengecek apakah `n` karakter pertama pada `s1` dan `s2` sama.
- `strcmpi(s1, s2)` mirip `strcmp()`, namun tidak *case-sensitive*.
- `strncmpi(s1, s2, n)` mirip `strncmp()`, namun tidak *case-sensitive*.

```
>> isequal(1, 3, 5)
ans = 0
>> strcmp("ayam", "Ayam")
ans = 0
>> strcmpi("ayam, "Ayam")
ans = 1
>> strncmp("sayamakan", "saya makan", 4)
ans = 1
```

Berikut beberapa jenis *conditional statement* pada Octave. Kode-kode ini akan ditulis di *editor* dan *output* akan dipisahkan oleh >>

Conditionals: If

`If` adalah *conditional statement* dasar dalam *decision-making* melalui perbandingan nilai. `If` memiliki 3 bentuk. Bentuk pertama:

```
if (cond)
  body;
endif
```

Bentuk ini adalah bentuk paling simpel dalam menggunakan `if`. Jika `cond` bernilai 1, maka `body` dieksekusi, dan sebaliknya. Contoh:

```
x = input("Masukkan nilai x: ");
if x > 0
  printf("%d adalah bilangan positif.\n", x);
endif
>>
```

```
Masukkan nilai x: 25
25 adalah bilangan positif.
```

Bukanlah `if` jika tidak ada `else`. Untuk menggunakannya, cukup menyelipkan bagian `else` layaknya `if` sehingga menjadi:

```
if (cond)
  body1;
else
  body2;
endif
```

Contoh:

```
x = input("Masukkan x: ");
if mod(x, 2) == 0
  printf("x genap.\n");
else
  printf("x ganjil.\n");
endif
>>
Masukkan x: 4
x genap.
>>
Masukkan x: 5
x ganjil.
```

Kita pun juga dapat membuat lebih dari 2 *condition* selain `if` dan `else`. Cukup tambahkan bagian `elseif`. Kita dapat menambahkan berapapun banyaknya `elseif` sesuka hati (dan komputer), selama bagian akhirnya adalah `else`.

```
if (cond1)
  body1;
elseif (cond2)
  body2;
else
  body3;
endif
```

Conditionals: Switch

Untuk beberapa kasus, lebih jelas jika kita menggunakan model kode seperti di atas. Namun, terkadang kita ingin membuat program berjalan sesuai *input*, dan jika menggunakan `if-else`, kodenya akan terlihat jelek. Maka, kita juga bisa menggantinya dengan kode `switch`. Bentuk umum dari `switch` adalah:

```
switch (var)
  case lab1
    body1;
  case lab2
    body2;
  otherwise
    bodyn;
endswitch
```

Pada kode di atas, `var` akan dicocokkan dengan `lab1`, `lab2`, dst. yang sesuai. Jika tidak ada yang sesuai, kode akan masuk ke bagian `otherwise`. Layaknya `elseif`, kita juga dapat menambahkan berapapun banyaknya `case` sesuka hati, selama terdapat paling tidak satu `case` (bahkan bagian `otherwise` opsional).

Contoh:

```
mnu = input("Masukkan metode: ");
switch (mnu)
  case 1
    printf("Bisection.\n")
  case 2
    printf("Regula Falsi.\n")
  otherwise
    printf("Input tidak valid.\n")
endswitch
>>
Masukkan metode: 1
Bisection.
>>
Masukkan metode: 3
Input tidak valid.
```

Jika `case` berisi *array*, kode akan masuk `case` tersebut jika `var` sesuai dengan salah satu elemen di *array* tersebut.

```
A = 7;
switch (A)
  case {6, 7}
    printf("A adalah 6 atau 7");
  otherwise
    printf("A bukanlah 6 ataupun 7");
endswitch
>>
A adalah 6 atau 7
```

Loops: For

Bentuk umum dari `for` adalah:

```
for var = expr
  body;
endfor
```

Biasanya isi dari `expr` adalah `a:b`, yang menyebabkan `var` diiterasi dari `a` hingga `b`. Secara umum, `for` akan meng-*assign* tiap kolom pada `expr` ke `var` (bentuk *range* `a:b` secara umum adalah vektor baris, sehingga iterasi kolom pada `a:b` adalah dari `a` hingga `b`). Contoh:

```
fib = ones(1, 10); % ones(1, 10) = matriks 1x10 berisi 1.
for i = 3: 10
  fib(i) = fib(i - 1) + fib(i - 2);
endfor
disp(fib)
>>

    1    1    2    3    5    8   13   21   34   55
```

Karena iterasinya antar kolom, maka jika `expr` adalah suatu matriks, maka var akan diiterasi sebagai vektor kolom.

```
for i = [1, 2, 3; 4, 5, 6; 7, 8, 9]
  i
endfor
>>

i =

   1
   4
   7

i =

   2
   5
   8

i =

   3
   6
   9
```

Loops: While

Bentuk umum dari `while` adalah:

```
while (cond)
  body;
endwhile
```

Serupa dengan `if`, `while` akan menjalankan `body` jika `cond` bernilai taknol. Namun, akan diulang terus hingga `cond` bernilai nol, baru berhenti.

Contoh:

```
fib = ones(1, 10);
i = 3;
while i <= 10
  fib(i) = fib(i - 1) + fib(i - 2);
  i++;
endwhile
disp(fib)
>>

    1    1    2    3    5    8   13   21   34   55
```

Pada contoh di atas, penting untuk memasukkan bagian `i++` agar suatu saat nilai `i` akan lebih dari 10. Hati-hati menggunakan `while`, karena dapat mengakibatkan *infinite loop*.

Loops: Do

Bentuk umum dari do adalah:

```
do
  body
until (cond)
```

Sekilas, `do` terlihat serupa dengan `while`. Yang membedakannya adalah `do` akan terus menjalankan `body` ketika `cond` bernilai 0 dan berhenti ketika `cond` bernilai taknol. Kondisi `cond` pada `do` juga berada di akhir, sehingga `body` pasti akan dijalankan paling tidak sekali. Perbedaan kecil selanjutnya adalah `do` tidak memakai `enddo` seperti layaknya `endif`, `endwhile`, dan sejenisnya.

Contoh:

```
fib = ones(1, 10);
i = 2;
do
  i++;
  fib(i) = fib(i - 1) + fib(i - 2);
until i == 10
disp(fib)
```

```
>>

    1     1     2     3     5     8    13    21    34    55
```

Break dan Continue

`break` dan `continue` adalah dua *statement* yang digunakan dan hanya digunakan dalam *loop*. *Statement* `break` akan langsung mengeluarkan program dari *loop*, sedangkan `continue` akan langsung menuju iterasi selanjutnya tanpa menyelesaikan sisa kode pada badan *loop*.

Contoh perbedaan `break` dan `continue`:

```
a = [];
for i = 1:10
  if mod(i, 5) == 0
    break;
  endif
  a = [a, i];
endfor
disp(a)
>>

   1   2   3   4
```

```
a = [];
for i = 1:10
  if mod(i, 5) == 0
    continue;
  endif
  a = [a, i];
endfor
disp(a)
>>

   1   2   3   4   6   7   8   9
```

Programming III: *Function File* dan *Script File*

Sebelum kita lanjutkan, kita harus terlebih dahulu mengetahui tentang *function file* dan *script file*.

Function file adalah *file* yang dapat digunakan oleh Octave untuk memanggil fungsi yang telah didefinisikan di dalamnya. *Function file* ini berguna jika kalian ingin menggunakan fungsi tersebut secara berkala.

Script file adalah *file* yang berisi kumpulan perintah Octave, layaknya *script* pemrograman. *Script file* berguna untuk pemrograman dan menjalankan/menyimpan suatu urutan perintah, sehingga bisa dijalankan kembali nantinya. Untuk selanjutnya, *script file* akan disebut "program".

Permasalahannya, kedua jenis file tersebut mempunyai ekstensi yang serupa (.m), namun *function file* tidak dapat dijalankan layaknya program.

Misal kita mempunyai fungsi yang ingin disimpan dalam program bernama `testfile.m` (untuk sekarang kita akan abaikan dulu maksud dari tiap bagian dari fungsi ini. Intinya fungsi ini akan menampilkan variabel `message` yang kita masukkan.

```
function test(message)
  printf("%s\n", message);
endfunction

test("AyatoBoba");
```

Jika program tersebut dijalankan, akan muncul pesan peringatan…

```
warning: function name 'test' does not agree with function
filename...
```

…dan mungkin saja akan diikuti *error* lain. Jika kalian ingin membuat program, jangan gunakan `function` di *line* pertama yang dieksekusi.

Sekarang kita modifikasi `testfile.m` di atas.

```
1;
function test(message)
  printf("%s\n", message);
endfunction

test("AyatoBoba");
```

Di sini, kita menambahkan *line* yang tidak berpengaruh apa-apa dalam program kita sebelum *line* pendefinisian fungsi. Untuk membedakan *function file* dengan program, Octave mengecek perintah pertama yang dieksekusi. Jika perintah tersebut adalah pendefinisian fungsi, maka *file* tersebut akan dianggap sebagai *function file*, dan jika bukan, maka *file* tersebut akan dianggap sebagai program.

Sekarang kita masuk ke fungsi, pendefinisian, dan embel-embelnya. Fungsi adalah suatu bagian dari program yang nantinya akan dipanggil. Fungsi sangat berguna jika bagian program

tersebut nantinya akan digunakan berkali-kali. Fungsi juga berguna agar pengorganisasian kode program lebih bagus. *Syntax* untuk pendefinisian fungsi adalah:

```
function name
  body
endfunction
```

Potongan kode di atas akan membuat fungsi `name` dengan `body` adalah isi dari fungsi tersebut. Untuk memanggil fungsi tersebut, cukup dengan memanggil `name`. Contoh:

```
function bangun
  printf("BANGUN!!!!!\n");
endfunction

bangun;
>>
BANGUN!!!!!
```

Kalian juga bisa menambahkan argumen (biasanya berupa variabel), ke fungsinya.

```
function bangun(message)
  printf("\a");
  printf("%s\n", message);
endfunction

bangun("BANGUN WOY!!");
>>
BANGUN WOY!!
```

Pada kedua contoh di atas, fungsinya tidak benar-benar memberikan suatu *value*, melainkan hanya sekedar *output*. Dalam kebanyakan kasus, kita menggunakan fungsi agar bisa mendapatkan suatu nilai yang dapat di-*assign* ke suatu variabel. Agar kita bisa mendapatkan value, maka kita harus meng-*assign* variabel untuk *return*. Strukturnya menjadi:

```
function retval = name (args)
  body
endfunction
```

retval adalah variabel lokal (namanya tidak harus retval) yang akan digunakan sebagai return value sehingga dapat di-*assign*. retval bisa berupa variabel, jika kita ingin me-*return* satu *value*, ataupun bisa berupa *list* dari variabel jika ingin me-*return* lebih dari satu *value*.

Contoh *return* satu nilai:

```
function x = quadratic(a)
  x = a^2;
endfunction
```

```
y = quadratic(2);
disp(y);
>>
4
```

Contoh *return* lebih dari satu nilai:

```
function [am, gm] = AMGM(v)
  am = sum(v) / length(v);
  gm = nthroot(prod(v), length(v));
endfunction

V = [1, 2, 3, 4, 5, 6, 7, 8, 9];
[amean, gmean] = AMGM(V);
printf("Arithmetic mean of %s is %g\n", mat2str(V), amean);
printf("Geometric mean of %s is %g\n", mat2str(V), gmean);
>>
Arithmetic mean of [1 2 3 4 5 6 7 8 9] is 5
Geometric mean of [1 2 3 4 5 6 7 8 9] is 4.14717
```

Octave juga mempunyai *syntax* `return` sendiri. Namun, `return` pada Octave tidak digunakan untuk me-*return* suatu *value*, melainkan untuk keluar dari fungsi (serupa dengan `break` pada *loop*).

Metode Euler dan Runge-Kutta

Metode Euler

Metode Euler metode paling dasar dalam mencari solusi dari permasalahan nilai awal dari suatu PD. Metode ini dikembangkan dari Teorema Taylor:

$$y(t_{i+1}) = y(t_i) + (t_{i+1} - t_l)y'(t_i) + \cdots$$

Misalkan kita mempunyai suatu persamaan diferensial dengan nilai awal:

$$y' = f(t, y), a \leq t \leq b$$
$$y(a) = \alpha$$

maka solusi secara numeriknya adalah $w_i = y(t_i)$, dengan:

$$w_1 = \alpha$$
$$w_{i+1} = w_i + hf(t_i, w_i), \qquad i = 1, 2, \dots, n$$

dengan $n + 1, n \in \mathbb{N}$ menyatakan banyaknya titik nantinya. Solusi kita akan berupa titik yang nantinya dapat menggunakan interpolasi untuk nilai yang tidak dimuat di w_i.

Algoritma untuk metode Euler adalah sebagai berikut:

```
function [t, w] = euler(f, a, b, n, alpha)
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;
  for i = 1: n
    t(i + 1) = t(i) + h;
    m1 = f(t(i), w(i));
    w(i + 1) = w(i) + h * m1;
  endfor
endfunction
```

Disini, inputnya adalah:

- `f = f(t, y)` merupakan suatu fungsi,
- `a` dan `b` berturut-turut batas bawah dan batas atas dari t,
- `n` merupakan pembagi untuk *step size* dan `n + 1` yang digunakan sebagai banyaknya titik, dan
- `alpha` merupakan nilai awal

Sekarang akan kita coba gunakan untuk menyelesaikan suatu PD. Misal diberikan PD sebagai berikut:

$$y' = y - t^2 + 1, 0 \leq t \leq 2$$
$$y(0) = 0.5$$

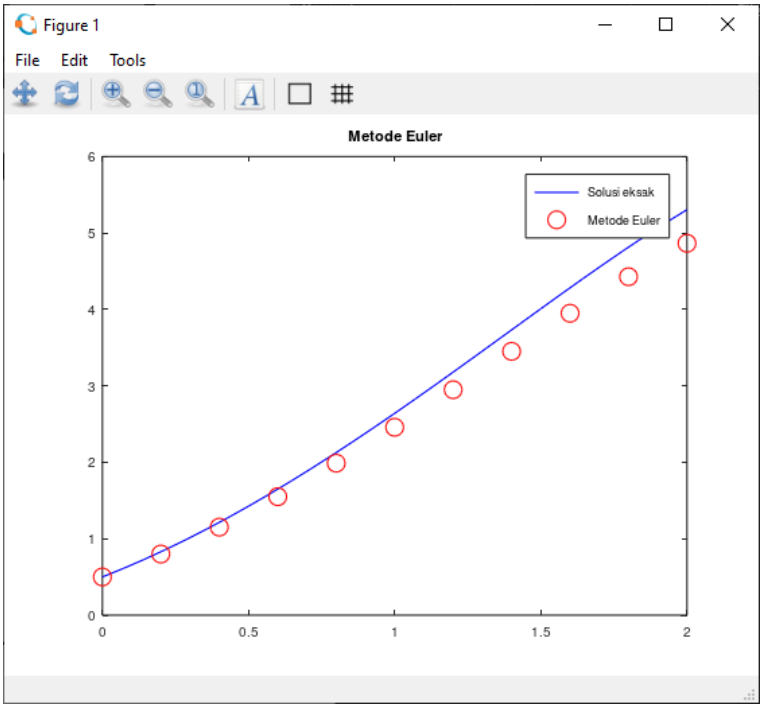
maka kita dapat mendefinisikan `f = @(t, y) (y - t^2 + 1)`, `a = 0`, `b = 2`, dan `alpha = 0.5` (`@` disini menyatakan fungsi anonim yang cara kerjanya mirip dengan fungsi lambda pada Python), sehingga untuk `n = 10`, diperoleh kode sebagai berikut:

```
f = @(t, y) (y - t^2 + 1);
a = 0;
b = 2;
n = 10;
[t_euler, w_euler] = euler(f, a, b, n, alpha)
>>
t_euler =

         0
    0.2000
    0.4000
    0.6000
    0.8000
    1.0000
    1.2000
    1.4000
    1.6000
    1.8000
    2.0000
w_euler =

    0.5000
    0.8000
    1.1520
    1.5504
    1.9885
    2.4582
    2.9498
    3.4518
    3.9501
    4.4282
    4.8658
```

Untuk visualisasinya, kita akan membuat *plot* dari hasil yang kita peroleh. Sebagai referensi, solusi eksak dari PD tersebut adalah $y(t) = (t + 1)^2 - 0.5e^t$. Kita tambahkan kode berikut pada *file*:

```
sln = @(t) (t + 1)^2 - 0.5 * exp(t);
fplot(sln, [0, 2], 'b');
hold on;
scatter(t_euler, w_euler, 'r');
legend('Solusi eksak', 'Metode Euler');
title('Metode Euler')
```

Saat dijalankan, akan muncul jendela *pop-up* yang berisi *plot* yang telah dibuat.



Penjelasan:

- sln berisi fungsi referensi kita untuk di-*plot* dan dibandingkan.
- fplot(f, [a, b]) akan menampilkan *plot* dari suatu fungsi f dengan domain [a, b]. Argumen tambahan 'b' memberi warna biru pada *plot*.
- hold on akan menahan *plot* yang ada agar kita bisa menampilkan banyak plot sekaligus.
- scatter(x, y) akan menampilkan x-y *scatter plot*.
- legend memberi legenda pada *plot* yang telah dibuat. Legenda tersebut dimasukkan berurutan mulai dari *plot* yang didefinsikan terlebih dahulu
- title memberi judul pada *plot*

Metode Runge-Kutta dan Variasinya

Ada beberapa variasi untuk metode Runge-Kutta ($i = 1, 2, \dots, n$):

1. Metode *midpoint*

$$w_1 = \alpha$$
$$w_{i+1} = w_i + hf\left(t_i + \frac{h}{2}, w_i + \frac{h}{2}f(t_i, w_i)\right)$$

2. Metode Euler modifikasi

$$w_1 = \alpha$$
$$w_{i+1} = w_i + \frac{h}{2}\Big(f(t_i, w_i) + f\big(t_{i+1}, w_i + hf(t_i, w_i)\big)\Big)$$

3. Metode Heun (tidak umum digunakan)

$$w_1 = \alpha$$
$$w_{i+1} = w_i + \frac{h}{4}\left(f(t_i, w_i) + 3f\left(t_i + \frac{2h}{3}, w_i + \frac{2h}{3}f\left(t_i + \frac{h}{3}, w_i + \frac{h}{3}f(t_i, w_i)\right)\right)\right)$$

4. Metode Runge-Kutta orde 4

$$w_1 = \alpha$$
$$m_1 = hf(t_i, w_i)$$
$$m_2 = hf\left(t_i + \frac{h}{2}, w_i + \frac{m_1}{2}\right)$$
$$m_3 = hf\left(t_i + \frac{h}{2}, w_i + \frac{m_2}{2}\right)$$
$$m_4 = hf(t_{i+1}, w_i + m_3)$$
$$w_{i+1} = w_i + \frac{m_1 + 2m_2 + 2m_3 + m_4}{6}$$

Algoritma untuk metode *midpoint*:

```
function [t, w] = midpoint(f, a, b, n, alpha)
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;
  for i = 1: n
    t(i + 1) = t(i) + h;
    m1 = f(t(i), w(i));
    m2 = f(t(i) + (h / 2), w(i) + (h / 2) * m1);
    w(i + 1) = w(i) + h * m2;
  endfor
endfunction
```

Algoritma untuk metode Euler modifikasi:

```
function [t, w] = euler(f, a, b, n, alpha)
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;
  for i = 1: n
    t(i + 1) = t(i) + h;
    m1 = f(t(i), w(i));
    m2 = f(t(i + 1), w(i) + h * m1);
    w(i + 1) = w(i) + h * (m1 + m2) / 2;
  endfor
endfunction
```

Algoritma untuk metode Heun:

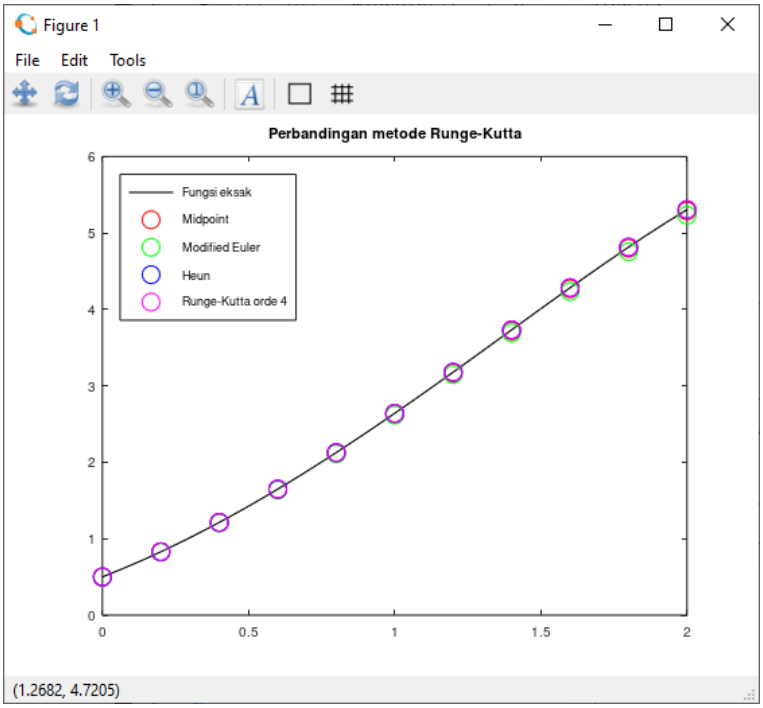
```
function [t, w] = heun(f, a, b, n, alpha)
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;
  for i = 1: n
    t(i + 1) = t(i) + h;
    m1 = f(t(i), w(i));
    m2 = f(t(i) + (h / 3), w(i) + (h / 3) * m1);
    m3 = f(t(i) + (2 * h / 3), w(i) + (2 * h / 3) * m2);
    m4 = m1 + 3 * m3;
    w(i + 1) = w(i) + (h / 4) * m4;
  endfor
endfunction
```

Algoritma untuk metode Runge-Kutta orde 4:

```
function [t, w] = rko4(f, a, b, n, alpha)
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;
  for i = 1: n
    t(i + 1) = t(i) + h;
    k1 = h * f(t(i), w(i));
    k2 = h * f(t(i) + (h / 2), w(i) + (k1 / 2));
    k3 = h * f(t(i) + (h / 2), w(i) + (k2 / 2));
    k4 = h * f(t(i + 1), w(i) + k3);
    w(i + 1) = w(i) + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
  endfor
endfunction
```

Seperti biasa, selanjutnya akan disajikan perbandingannya dalam bentuk *plot*.

```
f = @(t, y) (y - t ^ 2 + 1);
a = 0;
b = 2;
alpha = 0.5;
[t1, w1] = midpoint(f, a, b, 10, alpha);
[t2, w2] = modeuler(f, a, b, 10, alpha);
[t3, w3] = heun(f, a, b, 10, alpha);
[t4, w4] = rko4(f, a, b, 10, alpha);

sln = @(t) (t + 1) ^ 2 - 0.5 * exp(t);

fplot(sln, [0, 2], 'k');
hold on;
scatter(t1, w1, 'r');
scatter(t2, w2, 'g');
scatter(t3, w3, 'b');
scatter(t4, w4, 'm');
legend('Fungsi eksak', 'Midpoint', 'Modified Euler', 'Heun',
'Runge-Kutta orde 4');
legend("location", "northwest");
title('Perbandingan metode Runge-Kutta');
```



Metode Multistep

Metode-metode sebelumnya, seperi Euler, Runge-Kutta, dan kawan-kawannya adalah metode jenis *one-step*, karena kita hanya menggunakan informasi dari satu nilai t_i. Pada modul berikut akan dijelaskan mengenai metode *multistep*, dimana kita menggunakan lebih dari satu nilai t_i untuk membuat aproksimasi.

Terdapat dua jenis metode *multistep*, yaitu:

- *Multistep* eksplisit, dimana kita mengaproksimasi nilai pada t_{i+1} menggunakan nilai t sebelumnya.
- *Multistep* implisit, dimana kita mengaproksimasi nilai pada t_{i+1} menggunakan nilai pada t sebelumnya, sekaligus nilai pada t_{i+1} itu sendiri.

Untuk bagian awal, kita hanya akan menggunakan *multistep* eksplisit, dan *multistep* implisit akan dijelaskan kemudian menggunakan cara lain.

Multistep Eksplisit: Metode *n-step* Adams-Bashforth

Metode n-*step* Adams-Bashforth menggunakan n titik sebelumnya untuk mengaproksimasi nilai. Karena metode ini adalah metode *multistep*, maka n nilai awalnya pun harus diperoleh terlebih dahulu. Misal kita ingin menggunakan metode Adams-Bashforth orde 3, maka w_1, w_2, dan w_3 harus ada terlebih dahulu sebelum dilanjutkan ke metode Adams-Bashforth. Nilai-nilai awal tersebut dapat diperoleh dari metode-metode *one-step* sebelumnya, seperti metode Runge-Kutta, yang akan kita gunakan.

Berikut rumus untuk metode *n-step* Adams-Bashforth, masing-masing sesuai dengan jumlah *step* nya.

1. *Two-step* Adams Bashforth

$$w_0 = \alpha, \qquad w_1 = \alpha_1,$$
$$w_{i+1} = w_i + \frac{h}{2}[3f(t_i, w_i) - f(t_{i-1}, w_{i-1})]$$

2. *Three-step* Adams-Bashforth

$$w_0 = \alpha, \qquad w_1 = \alpha_1, \qquad w_2 = \alpha_2,$$
$$w_{i+1} = w_i + \frac{h}{12}[23f(t_i, w_i) - 16f(t_{i-1}, w_{i-1}) + 5f(t_{i-2}, w_{i-2})]$$

3. *Four-step* Adams-Bashforth

$$w_0 = \alpha, \quad w_1 = \alpha_1, \quad w_2 = \alpha_2, \quad w_3 = \alpha_3,$$
$$w_{i+1} = w_i + \frac{h}{24}[55f(t_i, w_i) - 59f(t_{i-1}, w_{i-1}) + 37f(t_{i-2}, w_{i-2}) - 9f(t_{i-3}, w_{i-3})]$$

4. *Five-step* Adams-Bashforth

$$w_0 = \alpha, \quad w_1 = \alpha_1, \quad w_2 = \alpha_2, \quad w_3 = \alpha_3, \quad w_4 = \alpha_4,$$
$$w_{i+1} = w_i + \frac{h}{720}[1901f(t_i, w_i) - 2774f(t_{i-1}, w_{i-1}) + 2616f(t_{i-2}, w_{i-2}) - 1274f(t_{i-3}, w_{i-3}) + 251f(t_{i-4}, w_{i-4})]$$

Algoritma untuk *two-step* Adams-Bashforth:

```
function [t, w] = adams2(f, a, b, n, alpha)
  % Inisiasi variabel awal
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;

  % Mencari t(2) dan w(2) menggunakan Runge-Kutta orde 4
  i = 1;
  t(i + 1) = t(i) + h;
  m1 = h * f(t(i), w(i));
  m2 = h * f(t(i) + (h/2), w(i) + (m1/2));
  m3 = h * f(t(i) + (h/2), w(i) + (m2/2));
  m4 = h * f(t(i + 1), w(i) + m3);
  w(i+1) = w(i) + (m1 + 2*m2 + 2*m3 + m4) / 6;

  % Algoritma utama Adams-Bashforth
  for i = 2:n
    t(i + 1) = t(i) + h;
    k1 = f(t(i), w(i));
    k2 = f(t(i-1), w(i-1));
    w(i+1) = w(i) + (h/2) * (3*k1 - k2);
  endfor
endfunction
```

Algoritma untuk *three-step* Adams-Bashforth:

```
function [t, w] = adams3(f, a, b, n, alpha)
  % Inisiasi variabel awal
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;

  % Mencari w(2) dan w(3) menggunakan Runge-Kutta orde 4
  for i = 1:2
    t(i + 1) = t(i) + h;
    m1 = h * f(t(i), w(i));
    m2 = h * f(t(i) + (h/2), w(i) + (m1/2));
    m3 = h * f(t(i) + (h/2), w(i) + (m2/2));
    m4 = h * f(t(i + 1), w(i) + m3);
    w(i+1) = w(i) + (m1 + 2*m2 + 2*m3 + m4) / 6;
  endfor

  % Algoritma utama Adams-Bashforth
  for i = 3:n
    t(i + 1) = t(i) + h;
    k1 = f(t(i), w(i));
    k2 = f(t(i-1), w(i-1));
    k3 = f(t(i-2), w(i-2));
    w(i+1) = w(i) + (h/12) * (23*k1 - 16*k2 + 5*k3);
  endfor
endfunction
```

Algoritma untuk *four-step* Adams-Bashforth:

```
function [t, w] = adams4(f, a, b, n, alpha)
  % Inisiasi variabel awal
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;

  % Mencari w(2), w(3), dan w(4) dengan menggunakan
  % Runge-Kutta orde 4
  for i = 1:3
    t(i + 1) = t(i) + h;
    m1 = h * f(t(i), w(i));
    m2 = h * f(t(i) + (h/2), w(i) + (m1/2));
    m3 = h * f(t(i) + (h/2), w(i) + (m2/2));
    m4 = h * f(t(i + 1), w(i) + m3);
    w(i+1) = w(i) + (m1 + 2*m2 + 2*m3 + m4) / 6;
  endfor

  % Algoritma utama Adams-Bashforth
  for i = 4:n
    t(i + 1) = t(i) + h;
    k1 = f(t(i), w(i));
    k2 = f(t(i-1), w(i-1));
    k3 = f(t(i-2), w(i-2));
    k4 = f(t(i-3), w(i-3));
    w(i+1) = w(i) + (h/24) * (55*k1 - 59*k2 + 37*k3 - 9*k4);
  endfor
endfunction
```

Algoritma untuk *five-step* Adams-Bashforth:

```
function [t, w] = adams5(f, a, b, n, alpha)
  % Inisiasi variabel awal
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;

  % Mencari w(2) hingga w(5) menggunakan Runge-Kutta orde 4
  for i = 1:4
    t(i + 1) = t(i) + h;
    m1 = h * f(t(i), w(i));
    m2 = h * f(t(i) + (h/2), w(i) + (m1/2));
    m3 = h * f(t(i) + (h/2), w(i) + (m2/2));
    m4 = h * f(t(i + 1), w(i) + m3);
    w(i+1) = w(i) + (m1 + 2*m2 + 2*m3 + m4) / 6;
  endfor

  % Algoritma utama Adams-Bashforth
  for i = 5:n
    t(i + 1) = t(i) + h;
    k1 = f(t(i), w(i));
    k2 = f(t(i-1), w(i-1));
    k3 = f(t(i-2), w(i-2));
    k4 = f(t(i-3), w(i-3));
    k5 = f(t(i-4), w(i-4));
    w(i+1) = w(i) + (h/720) * (1901*k1 - 2774*k2 + 2616*k3 -
             1274*k4 + 251*k5);
  endfor
endfunction
```

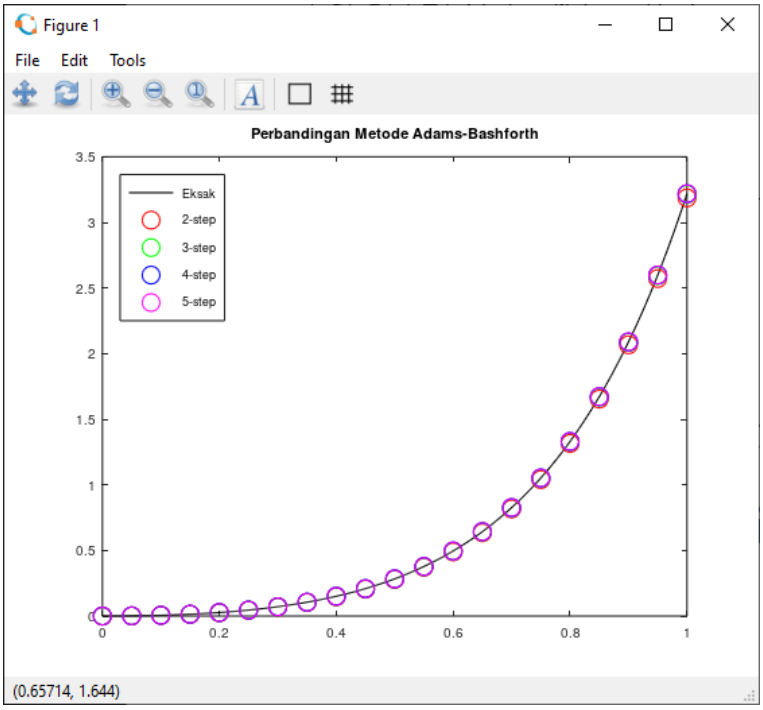
Sekarang akan kita gunakan metode-metode tersebut untuk menyelesaikan suatu permasalahan nilai awal. Misal diberikan permasalahan nilai awal:

$$y' = te^{3t} - 2y, \qquad 0 \le t \le 1, \qquad y(0) = 0,$$

Solusi eksaknya adalah $y(t) = \frac{1}{5}te^{3t} - \frac{1}{25}e^{3t} + \frac{1}{25}e^{-2t}$. Dengan *stepsize* 0.05, kita akan membentuk 20 titik tambahan.

```
% Inisiasi variabel
f = @(t, y) (t*exp(3*t) - 2*y);
a = 0;
b = 1;
alpha = 0;
n = 20;
```

```
% Solusi dengan Adams-Bashforth
[t2, w2] = adams2(f, a, b, n, alpha);
[t3, w3] = adams3(f, a, b, n, alpha);
[t4, w4] = adams4(f, a, b, n, alpha);
[t5, w5] = adams5(f, a, b, n, alpha);

% Solusi eksak
sln = @(t) ((t*exp(3*t)/5) - (exp(3*t)/25) + (exp(-2*t)/25));
w = [];
for i = 1:length(t2)
  w(i) = sln(t2(i));
endfor

% Menampilkan tabel
[w', w2, w3, w4, w5]

% Membuat plot
fplot(sln, [0, 1], 'k');
hold on;
scatter(t2, w2, 'r');
scatter(t3, w3, 'g');
scatter(t4, w4, 'b');
scatter(t5, w5, 'm');
legend('Eksak', '2-step', '3-step', '4-step', '5-step');
legend("location", "northwest");
title('Perbandingan Metode Adams-Bashforth');
```



Terlihat bahwa makin besar *n-step*-nya, maka aproksimasi akan makin akurat.

Metode Predictor-Corrector

Sebelumnya belum dijelaskan tentang metode *multistep* implisit. Adams juga mempunyai metode untuk *multistep* implisit, yang bernama metode *n-step* Adams-Moulton. Secara umum, metode ini lebih akurat daripada metode Adams-Bashford, namun perhitungannya harus menggunakan persamaan implisit. Sebagai contoh, berikut persamaan untuk metode *three-step* Adams-Moulton:

$$w_0 = \alpha, \qquad w_i = \alpha_1, \qquad w_2 = \alpha_2,$$
$$w_{i+1} = w_i + \frac{h}{24}[9f(t_{i+1}, w_{i+1} + 19f(t_i, w_i) - 5f(t_{i-1}, w_{i-1}) + f(t_{i-2}, w_{i-2})]$$

Terlihat bahwa untuk mengaproksimasi w_{i+1}, kita membutuhkan nilai dari w_{i+1} itu sendiri, sehingga kita harus menyelesaikan persamaan implisit terlebih dahulu, dimana dalam beberapa kasus bisa saja tidak ada solusi implisitnya (contoh: $y' = e^y$).

Ada beberapa cara agar kita tetap dapat menggunakan metode implisit, salah satunya adalah dengan metode *predictor-corrector*. Metode ini mengaproksimasi w_{i+1} menggunakan metode eksplisit (*predict*), dan hasilnya akan digunakan sebagai nilai w_{i+1} pada ruas kanan persamaan implisit dari metode implisit, sehingga diperoleh aproksimasi yang lebih baik (*correct*).

Salah satu implementasi dalam metode *framkenstein* ini adalah algoritma Adams *predictor-corrector* orde 4. Algoritma ini menggunakan metode Runge-Kutta untuk mencari nilai awal, menggunakan *four-step* Adams-Bashforth untuk memprediksi nilai aproksimasi, dan menggunakan *three-step* Adams-Moulton untuk "mengoreksi" nilai aproksimasi dari metode Adams-Bashforth.

Algoritma untuk Adams *predictor-corrector* orde 4:

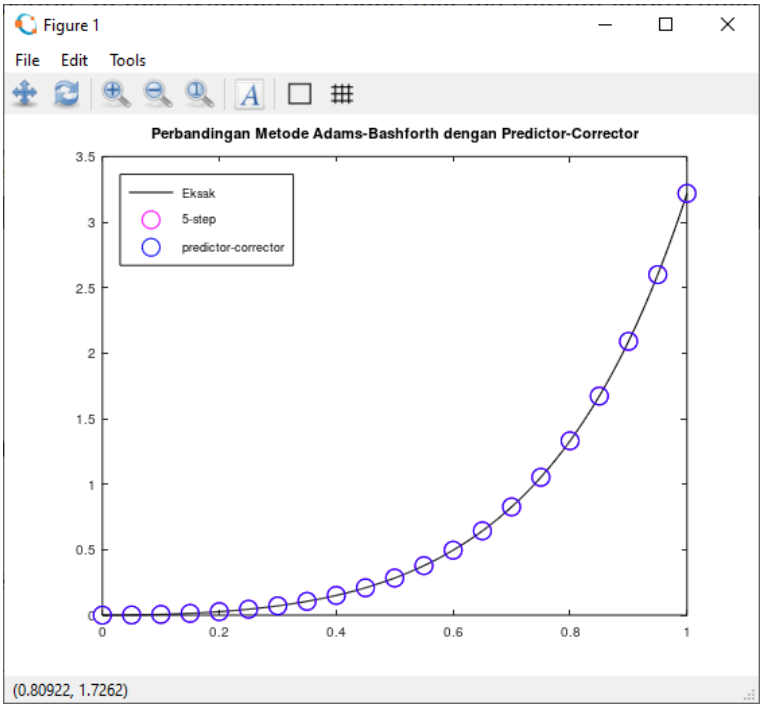
```
function [t, w] = adamspc(f, a, b, n, alpha)
  % Inisiasi variabel awal
  h = (b - a) / n;
  t = zeros(n + 1, 1);
  w = zeros(n + 1, 1);
  t(1) = a;
  w(1) = alpha;

  % Metode Runge-Kutta untuk mencari nilai awal
  for i = 1:3
    t(i + 1) = t(i) + h;
    m1 = h * f(t(i), w(i));
    m2 = h * f(t(i) + (h/2), w(i) + (m1/2));
    m3 = h * f(t(i) + (h/2), w(i) + (m2/2));
    m4 = h * f(t(i + 1), w(i) + m3);
    w(i+1) = w(i) + (m1 + 2*m2 + 2*m3 + m4) / 6;
```

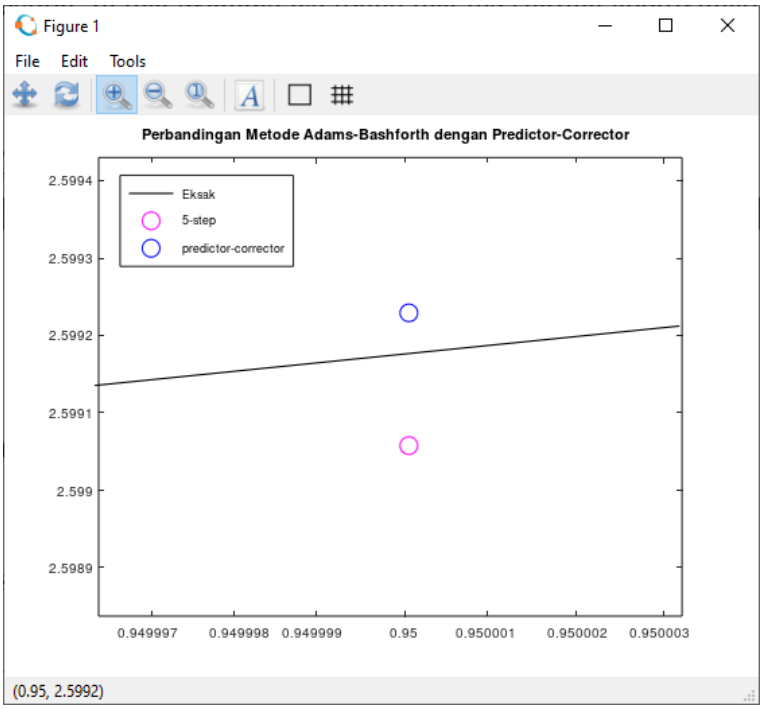
```
  endfor

  % Algoritma utama
  for i = 4:n
    t(i + 1) = t(i) + h;

    % Metode Adams-Bashforth sebagai predictor w(i+1)
    k1 = f(t(i), w(i));
    k2 = f(t(i-1), w(i-1));
    k3 = f(t(i-2), w(i-2));
    k4 = f(t(i-3), w(i-3));
    w(i+1) = w(i) + (h/24) * (55*k1 - 59*k2 + 37*k3 - 9*k4);

    % Metode Adams-Moulton sebagai corrector w(i+1)
    k0 = f(t(i+1), w(i+1));
    w(i+1) = w(i) + (h/24) * (9*k0 + 19*k1 - 5*k2 + k3);
  endfor
endfunction
```

Kita bandingkan metode ini dengan *five-step* Adams Bashforth.

```
% Inisiasi variabel
f = @(t, y) (t*exp(3*t) - 2*y);
a = 0;
b = 1;
alpha = 0;
n = 20;

% Solusi dengan five-step Adams-Bashforth
[t5, w5] = adams5(f, a, b, n, alpha);

% Solusi dengan Adams predictor-corrector orde 4
[tpc, wpc] = adamspc(f, a, b, n, alpha);

% Solusi eksak
sln = @(t) ((t*exp(3*t)/5) - (exp(3*t)/25) + (exp(-2*t)/25));
w = [];
for i = 1:length(t2)
  w(i) = sln(t2(i));
endfor
[w', w5, wpc]

% Membuat plot
fplot(sln, [0, 1], 'k');
hold on;
scatter(t5, w5, 'm');
scatter(tpc, wpc, 'b');
legend('Eksak', '5-step', 'predictor-corrector');
legend("location", "northwest");
```

```
title('Perbandingan Metode Adams-Bashforth dengan Predictor-Corrector');
```





Shooting Method

Finite Difference