Ex. 9, Symbolic vs. Numerical Programming

Stefan Karlsson

May 12, 2014

1 Introduction

In this exercise we will use symbolic programming. To introduce this topic let's consider the following irrational expression, which we can not express exactly using regular numerical representation.

$$\beta = \frac{1+\sqrt{2}}{3} \approx 0.8$$

To get a numerical approximation, we can use matlab directly by simply typing:

```
betaNumeric = (1 + sqrt(2))/3
```

The numeric variable `betaNumeric` is clearly just an approximation. The numerical precision of the computer quickly becomes evident if we try to express the relation:

$$\beta^2 - 2\frac{\beta}{3} - \frac{1}{9} = 0$$

by the following Matlab code:

```
betaNumeric^2 - 2*betaNumeric/3 -1/9
```

To express the irrational β exactly, we need to resort to symbolic programming. This is very different from numerical programming, and can be a very powerful tool to avoid numerical errors, and perform common mathematical operations(such as derivation) in an exact way. This will be familiar to those who have had experience in environments such as Mathematica, Maple and MuPad. To use symbolics, we need to declare our variables specifically as symbolic first:

```
betaSymbolic = sym('(1 + sqrt(2))/3')
```

Notice how `betaSymbolic` is now echoed as a symbolic and its expression is not evaluated numerically. To get a symbolic expression to be evaluated numerically we write:

```
eval(betaSymbolic)          %yields the same value as "betaNumeric"
```

With the symbolic variable, lets again see if we can get Matlab to deduce that $\beta^2 - 2\frac{\beta}{3} - \frac{1}{9} = 0$, by

```
symbolicExpression = betaSymbolic^2 - 2* betaSymbolic/3 -1/9
```

This should get you a long expression of symbolics, seemingly no closer to zero. If you were to simplify this expression with pen and paper (don't do that!) you would find it to actually equal zero. In order to make Matlab do this simplification for you:

```
simplify(symbolicExpression)     %simplifies to zero
```

And there you have it, zero! In the above example, its quite obvious what the simplest form of the expression is (it is 0). It is not always obvious what expression is a simplification however, and may depend on what you are trying to achieve. While `simplify` is the most useful and general-purpose function for this, there exists a range of other symbolic simplifier functions, such as `simple`, `factor`, `collect`, `expand` and `horner`.

1.1 Algebraic Symbols

Even more powerful than previous example is to use algebraic expression with unknown variables. We are used to having variables in Matlab, but we have so far only encountered numerical variables, not algebraic ones. First we need to declare variables as symbols for this to work. A fast way to do that is by:

```
syms t a b c d     %declares symbolic variables t, a, b, c and d
```

These symbols represent any number, and can be used in most Matlab functions. With symbolic expressions of one or more variables we can perform many of the mathematical operations such as differentiation (using `diff`) and integration (using `int`):

```
diff(  t^2    )
int(   cos(t) )         %also known as "anti-derivative"
```

To really see the use of this, consider more complicated functions, such as:

$$\left(t + \sqrt{t} - t^2\right)^2$$

Consider doing differentiation and integration of above function. Tedious by hand, but with symbolic programming:

```
int ((t+sqrt(t)-t^2)^2)        %anti-derivative
diff((t+sqrt(t)-t^2)^2)        %first order derivative
diff((t+sqrt(t)-t^2)^2,  2)    %second order derivative
diff((t+sqrt(t)-t^2)^2,  10)   %tenth order derivative
```

Taking tenth order derivative would take alot of time for a human, also notice how horribly long the expression turned out. Lets simplify it:

```
simplify(diff((t+sqrt(t)-t^2)^2,10))
```

If this is the first time you encounter functions like these, you may feel like you missed out during your calculus studies...

1.2 Symbolic Matrices

Matrices and symbolic variables work well together in Matlab:

```
A = [a b; ...        %defines a 2 by 2 symbolic matrix
     c d;]
```

The regular matrix functions of trace, determinant and inverse will all work on matrix A, yielding symbolic expressions:

```
trace(A)
det(A)
inv(A)                %not recommended for larger symbolic matrices
```

1.3 Assumptions

For a variable $a > 0$ we have equality for:

$$\sqrt{a^2} = a$$

notice that this only holds if we assume the variable is positive. Lets see what happens in Matlab with our symbolic variable `a`:

```
sqrt(a^2)             %will not equal "a", even if we simplify
```

We would like to include the assumptions on `a` in our calculations, and an easy way to do that is in the definition of `a`:

```
syms a positive;      %a is now assumed positive

sqrt(a^2)
sqrt(b^2)             %b is NOT assumed positive
```

2 Differentiation

We saw some simple examples of differentiation and integration symbolically(which we call analytically in math) in section 1.1. Lets see how we could do differentiation numerically. In numerical differentiation, we work with samples of the function, instead of an analytical description. Instead of doing analytical differentiation we do elementwise differences in the numerical case. We use the function `gradient` for this purpose:

```
x = 1:4;
gradient(x)           %linear input, should give constant output of 1
```

We have worked with sampled functions previously (audio/music exercises), so lets take the familiar cosinus function as an example. We know that the derivative of $cos(t)$ is equal to $-sin(t)$, and can remind ourselves by:

```
syms t;
diff(cos(t))
```

lets try to estimate this numerically, by sampling the function over the range $[0, 2\pi]$:

```
Fs = 3;               %sampling frequency
Ts = 1/Fs;            %length between samples
t  = 0: Ts :2*pi;     %sample the range
f  = cos(t);
df = gradient(f,Ts);  %estimate derivative, need "lenght between samples"

plot(t,f); hold on
plot(t,df,'r');
plot(t,-sin(t),'g');

legend('cos(t)', 'numerical derivative', 'exact derivative');
```

As we can see, the numerical derivative comes pretty close to the exact one in this case. The accuracy depends on how richly we describe the function. Try increasing and decreasing the sampling frequency to see the impact on the precision. More samples means better accuracy.

3 Integration (anti-derivatives)

For numeric integration we have to approximate using sums. We can use the `cumsum` function to simply sum the values of the vector together, but a better approximation is through the trapezoidal method which is implemented in `cumtrapz`:

```
c = ones(1,4);
cumtrapz(c)           %constant input, should give linear output
```

We know that the anti-derivative of $cos(t)$ is equal to $sin(t)$, and can remind ourselves by:

```
syms t;
int(cos(t))
```

lets try to estimate this numerically, by sampling the function over the range $[0, 2\pi]$:

```
Fs = 3;               %sampling frequency
Ts = 1/Fs;            %length between samples
t  = 0: Ts :2*pi;     %sample the range
f  = cos(t);

F = cumtrapz(f)*Ts;   %estimate anti-derivative, we need to multiply with Ts
                      %because cumtrapz assumes unit length between samples
plot(t,f); hold on
plot(t,F,'r');
plot(t,sin(t),'g');

legend('cos(t)', 'numerical anti-derivative', 'exact anti-derivative');
```

As with the derivatives, accuracy depends on how richly we describe the function - Higher sampling frequency means better accuracy. However, there is one additional problem with the approach of the above snippet. Recall from elementary calculus that

$$\int_0^t f(x)dx = F(t) - F(0) = F(t) + C \tag{1}$$

where $F(t)$ is the anti-derivative. To use a summation approximation (as we have done in above snippet) means that we have assumed $F(0) = 0$. Since we know before-hand that $F(0) = sin(0) = 0$, our assumption is true. If we change the analyzed function, we run into trouble as below snippet illustrates:

```
Fs = 10;
Ts = 1/Fs;
P  = pi/10;                %phase shift of the function(moving it to the right)

t = 0: Ts :2 * pi;
f = cos(t - P);            %define function with phase shift
C  = 0;                    %the initial value
F = cumtrapz(f)*Ts + C;    %include initial value

plot(t,f); hold on
plot(t,F,'r');
plot(t,sin(t-P),'g');
%legend with the phase shift information:
legend(['cos(t - ' num2str(P,2) ')'], ...
        'numerical anti-derivative' , ...
        'exact anti-derivative');
```

If you change the initial value `C`, you will modify the solution which can be tuned to a reasonable result. In above example, the correct value is `C = -sin(P)`, make sure you understand why! If in doubt, inspect Equation 1.

4 Taylor series

Many functions can be expressed through a Taylor polynomial expansion around a point of interest $t = a$:

$$f_a(t) = f(a) + \frac{\dot{f}(a)}{1!}(x-a)^1 + \frac{\ddot{f}(a)}{2!}(x-a)^2 + \ldots$$

In Matlab we can do this by the function `taylor`. Lets approximate $sin(t)$ around the point of interest $a = 0$, up to order of 6:

```
syms t;
TaylorOrder = 6;
sinApprox = taylor(sin(t),'ExpansionPoint',0,  'Order',TaylorOrder);

%use "ezplot" to plot symbolic expressions:
h1 = ezplot(sin(t));
hold on
```

```
h2 = ezplot(sinApprox);

set(h1, 'Color','r');
ylim([-1.5 2])
legend('sin(t)',['Taylor Order:' num2str(TaylorOrder)])
```

5 Equation solving

We had experience solving equations numerically, lets take a look at the powerful symbolic approach to it. Consider the following equation:

$$ax^2 + bx + 3 = 0$$

with 2 solutions

$$x = -\frac{b}{2a} \pm \frac{\sqrt{b^2 - 12a}}{2a}$$

In Matlab we solve this by:

```
syms a b x
sols = solve(a*x^2+b*x+3)
```

This gives a vector `sols` of 2 symbolic expressions (one for each solution). If we wish to assign some specific values to any symbolic expression, we use `subs`. Say we wish to know solutions for `a=1` and `b=2`:

```
subs(sols,[a b],[1 2])
```

Notice how this does not provide a numberical solution, even though we replaced the symbolic variables `a` and `b`.

6 Differential equations

The differential equation:

$$\frac{dy}{dt} = 1 + y^2$$

has solution $y = \tan(C + t)$, for a constant C. To check that this is true, we write:

```
syms t C
y = tan(C + t);
simplify(  diff(y)==y^2+1  )
```

To find a solution from the start, we use `dsolve`:

```
syms y
dsolve('Dy=1+y^2')
```

This will give us 3 solutions: the real solution $y = \tan(C + t)$ and two imaginary solutions $y = \pm\imath$. To restrict to real numbers:

```
syms y real
dsolve('Dy=1+y^2')
```

The constant in the equation can be resolved if we provide more constraints. The most common constraint is an initial condition (e.g. $y(0) = 1$):

```
syms y real;
dsolve(  'Dy=1+y^2' ,   'y(0)=1')
```

For a third order example, we will solve the following equation:

$$\frac{d^3y}{dx^3} = y$$

To solve a third order differential equation we need 3 initial conditions. We use:

$$\begin{aligned} y(0) &= 1 \\ \frac{dy}{dx} &= -1 \\ \frac{d^2y}{dx^2} &= \pi \end{aligned}$$

Which looks like:

```
syms y real;
dsolve('D3y=y'    , ...       %the equation
       'y(0)=1'   , ...       %first constraint...
       'Dy(0)=-1' , ...       %second...
       'D2y(0)=pi', ...       %third
       'x')                   %indicate the independent variable
```

The solution is easy to verify in theory. I don't recommend you carry out the calculations by yourself however. This is what we have computers for.

7 Systems of differential equations

2 or more simultaneous differential equations form a system, and is extremely important in engineering and science. An example system of 2 linear equations looks like:

$$\begin{aligned} \dot{x}_1(t) &= -3x_1(t) + 2x_2(t) \\ \dot{x}_2(t) &= -4x_1(t) + x_2(t) \end{aligned}$$

with initial conditions:

$$x_1(0) = 0$$
$$x_2(0) = 1$$

In Matlab we use the same versatile `dsolve` for systems of equations:

```
syms t reals
[x1,x2] = dsolve(...
          'Dx1=-3*x1+2*x2', ...  % the system, first line ...
          'Dx2=-4*x1+x2'  , ...  % second line.
          'x1(0)=0'       , ...  % initial conditions ...
          'x2(0)=1'       , ...
          't');                  % the independent variable
x1 = simplify(x1)                % simplify solutions if possible
x2 = simplify(x2)

%Plot the solutions using ezplot:
h1 = ezplot(x1,[0,2*pi]); hold on;
h2 = ezplot(x2,[0,2*pi]);
set(h2, 'Color','r');
legend('x1','x2');
ylim([-0.4,1.3])
```

Tasks

Solutions that do not fill the following requirements **EXACTLY, PRECISELY AND TO THE LETTER** will not be considered:

- Send your solutions by email to:

  stefan.karlsson@hh.se

  subject: Matlab, Exercise `X`, `YourNames`

- Send all the files that are requested, no more and no less, in one single zip file per exercise, with NO sub-folders in the zip file. All the files, for all the tasks should be bundled into one zip file.

- Put the Names of the authors, in remarks, at the top of every m-file.

- Send the solutions within 2 weeks of every exercise session. That is, you have a two week deadline to hand it in.

- You will get 2 chances to send it in to me correctly.

Task 1

Write an m-file `e9_1.m` that proves that rotation matrices (for 2D rotations in the plane) are invertible by their transpose. Use symbols of the symbolic toolbox in Matlab. The m-file should output in a nice format:

1. The definition of a rotation matrix: R
2. The transpose of a rotation matrix: R^T
3. The inverse of a rotation matrix: R^{-1}
4. Proof that: $R^T R = I$ (identity matrix), $R^{-1}R = I$ and $R^T - R^{-1} = 0$ (the 2-by-2 zero matrix)

Hint: use `simplify`

Task 2

Write an m-file `e9_2.m` that solves the following equation system consisting of 3 differential equations:

$$\begin{aligned}\dot{x}_1(t) &= x_1(t) + x_2(t) - 2x_3(t)\\ \dot{x}_2(t) &= 2x_1(t) - 2x_3(t)\\ \dot{x}_3(t) &= -2x_1(t) + 2x_2(t) + x_3(t)\end{aligned}$$

Make a plot of the solutions in the interval 0-2 sec. Assume the following initial conditions: $x_1(0) = 2$, $x_2(0) = 1$ and $x_3(0) = 1$

Task 3(optional for grade 4)

Create an m-file `e9_3.m` that has a user interface (create with GUIDE). The GUI should have 3 buttons performing integration(anti-derivative using `int`), differentiation and Taylor polynomial when pressed. When pressing any of the buttons, the resulting function should be plotted in an axes object in the GUI, together with the original function. Thus, always there should be two graphs plotted, the original function in blue, and the new function (through integration, differentiation or Taylor) in red.

The function is given in a text box in the GUI, and is a function of `x`.

For the Taylor approximation, its expansion point and its order should be given in two text boxes in the GUI.

The GUI should look like the figure below, with appropriate default values. The default function is `cos(x)`:

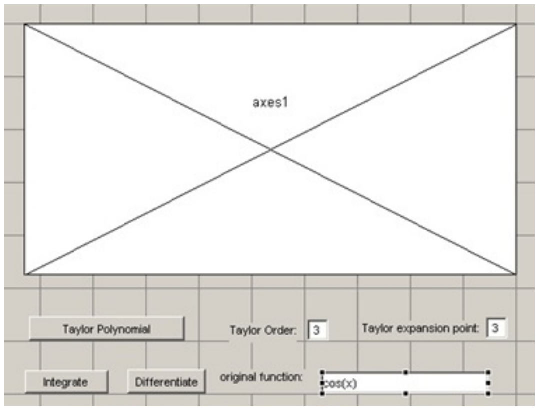

Figure 1: Appearance of the GUI for Task 3