Exercises – Debugging

1 Debugging expression graphs

1. Create a `Function` mapping the following `x` to `z`:

   ```
   x = SX.sym('x')
   y = sin(x)
   z = y/x+y
   ```

   Verify that `f(0.01)` yields `1.00998`

2. Use printme to print out the value of `y` during function evaluation. Verify that you get `9.9998333341666645e-03`.

3. The following `Function` only makes sense when $x >= 0$:

   ```
   x = MX.sym("x")
   y = sqrt(x)
   f = Function("f",[x],[y])
   ```

   Add an `attachAssert` in here such that computing `f(-3)` results in the error message `That's not allowed`.

4. What happens if you make `f` output `jacobian(y,x)` instead?

5. Create a `Function` mapping `x` to `y`:

   ```
   x = MX.sym('x')
   y = if_else(x<=1,x*x,-0.5*x*x+3*x-1.5)
   ```

   Make a plot using a linspace from $x = -4$ to $x = 5$.

6. Apply a `printme` on both the true-clause (`x*x`) and the false-clause. Evaluate at $x = 3$. Which clause is evaluated?

7. Read the help-string of `if_else`. Repeat the above exercise with short-circuiting activated. What difference do you see?

2 Debugging options for Functions

The experiments of the first section are rather straightforward. In reality, you'll often run into a need to debug when you are solving a complex problem, e.g. nonlinear optimization using the `Opti` wrapper. Let's try some experiments closer to that case.

1. Run the following snippet:

   ```
   opti = Opti()
   x = opti.variable()
   opti.subject_to(sqrt(x)==0.3)
   opti.set_initial(x, 3)
   opts = {}
   opti.solver('ipopt',opts)
   opti.solve()
   ```

   You get `WARNING("solver:nlp_g failed: NaN detected for output g, at (row 0, col 0).")`

   Let's say you have a hunch that the square root is causing issues.

   Use the `monitor` option to make print the inputs and outputs of the internally generated constraint `Function` `nlp_g`.

   Verify that you get negative `x`'es.

   Why does the solver still converge?

2. Repeat the same after using `DM.set_precision(digits)` to get higher precision readings on the screen.

3. The output of the above exercise contains a time-stamped CasADi warning. From `https://github.com/casadi/casadi/blob/3.5.1/casadi/core/oracle_function.cpp#L44-L63` find the option to disable this warning.

4. Start again from the following optimization problem:

   ```
   opti = Opti()
   x = opti.variable(2, 2)
   opti.subject_to(sqrt(x-1)==0.3)
   opti.set_initial(x, 3)
   opti.solver("ipopt",{"show_eval_warnings":False})
   opti.solve()
   ```

   Compare the effect of putting `printme` and `monitor` directly under the square root.

5. With `specific_options`, we can apply options to specific helper functions automatically generated by CasADi on demand of the concrete nonlinear programming interface. Activate `print_in` and `print_out` for the Langrange Hessian of the following problem:

```
opti = Opti()

x = opti.variable()
y = opti.variable()
z = opti.variable()

opti.minimize(x**2 + 100*z**2)

opti.subject_to(z ==y- (1-x)**2)

opti.set_initial(x, 2.5)
opti.set_initial(y, 3.0)
opti.set_initial(z, 0.75)
```

From inspection, you can deduce that the solution is given by $x = 0, y = 1, z = 0$.

What Lagrange Hessian do you expect at the solution? Why is it not visible in the output? As a hint, divide the reported scaled objective to the unscaled objective. Does the result look familiar?

6. Use `get_function` on `opti.debug.casadi_solver` to get access to the `nlp_hess_l` Function.

   Evaluate it at the solution and for `lam_f`$= 1$.

7. (extra) Create SX symbols for all inputs of `nlp_hess_l` using `*_in` methods. Calling `nlp_hess_l` symbolically with these, can you interpret the result?
   Compare its simplicity with `nlp_hess_l.disp(True)`.

8. Change the solver to `sqpmethod` with `qrqp` as QP solver. Instead of applying `print_in`/`print_out` on just the Hessian, apply it on the QP solver as a whole via `qpsol_options`.

   Can you find the meaning of all inputs/outputs in `https://web.casadi.org/python-api/#qp`?

9. The QP solver has another option available as shown in the slides. Verify that you get the same numbers.

10. Printing on the console is one thing, but suppose that you want to log and analyze a bunch of QPs produced by an model predictive control scheme. A save-to-disk feature would be handy, right? Find an option in the slides to do this.

    Your current directory will suddenly grow with a bunch of files. Open the one related to the Hessian with `DM.from_file`.

11. Specify a `txt` format for saving to the disk. Compare the two formats. Which is easier to read by a human? Which scales better for large systems?

12. Activate the `dump` option and inspect `Function.load('qpsol.casadi')`. What object is returned? Call it with some saved numeric inputs.
    You may use `qpsol.generate_in('qpsol.000000.in.txt')` to slurp in all at once.