JAX or SymPy

The currently supported, known constraints are defined as bounds on the image of first or second order derivatives of the function. Therefore, they encompass monotonic, or convexity/concavity constraints.

For any trained prediction model, we might want to check if it adheres to the known constraints. For this, we can either symbolically derive the model using SymPy and analyze the image of its derivatives, alternatively, we can use JAX for automatic derivatives of native python functions.

Usage

In [3]:

```
import jax.numpy as jnp

import SCRBenchmark.SRSDFeynman as srsdf
from SCRBenchmark import Benchmark

################ ICh6Eq20 ################
ICh6Eq20 = Benchmark(srsdf.FeynmanICh6Eq20)

def f(x):
  return jnp.exp(-(x[0] / x[1]) ** 2 / 2) / (jnp.sqrt(2 * jnp.pi) * x[1])

print("JAX ICh6Eq20 Test:")
print(ICh6Eq20.check_constraints(f, Library = "JAX"))

print("SymPy ICh6Eq20 Test:")
print(ICh6Eq20.check_constraints("exp(-(x0 / x1) ** 2 / 2) / (sqrt(2 * pi) * x1)"))


################ ICh9Eq18 ################
ICh9Eq18 = Benchmark(srsdf.FeynmanICh9Eq18)

def g(x):
  return srsdf.feynman.GRAVITATIONAL_CONSTANT * \
    x[0] * x[1] / ((x[2] - x[3]) ** 2 + (x[4] - x[5]) ** 2 + (x[6] - x[7]) ** 2)

print("JAX ICh9Eq18 Test:")
print(ICh9Eq18.check_constraints(g, Library = "JAX"))

print("SymPy ICh9Eq18 Test:")
print(ICh9Eq18.check_constraints(" 6.67430e-11 * x0 * x1 / ((x2 - x3) ** 2" + \
            " + (x4 - x5) ** 2 + (x6 - x7) ** 2)"))
```

```
JAX ICh6Eq20 Test:
(True, [])
SymPy ICh6Eq20 Test:
(True, [])
JAX ICh9Eq18 Test:
(True, [])
SymPy ICh9Eq18 Test:
(True, [])
```

JAX provides faster execution but less accurate (at higher precision) gradients.

Interchangeability

The following code proves that, up to a certain precision, **both SymPy and JAX calculate the same gradients. Therefore, JAX serves as a suitable alternative for **(1)** models that cannot be derived symbolically, or **(2)** for quicker model evaluation during training (e.g., to guide the search).

In [1]:
```
import jax
import numpy as np
import sympy
import SCRBenchmark.Constants.StringKeys as sk
import SCRBenchmark.base as base



import SCRBenchmark.SRSDFeynman as srsdf
from SCRBenchmark import Benchmark
ICh9Eq18 = Benchmark(srsdf.FeynmanICh9Eq18)

####### the actual equations once as string, once as JAX function #######
expression = " 6.67430e-11 * x0 * x1 / ((x2 - x3) ** 2" + \
            " + (x4 - x5) ** 2 + (x6 - x7) ** 2)"
def f(x):
  return srsdf.feynman.GRAVITATIONAL_CONSTANT * \
    x[0] * x[1] / ((x[2] - x[3]) ** 2 + (x[4] - x[5]) ** 2 + (x[6] - x[7]) ** 2)


########################## check constraints ##########################
constraints = ICh9Eq18.get_constraints()

constraints = [c for c in constraints
               if c[sk.EQUATION_CONSTRAINTS_DESCRIPTOR_KEY]
               !=sk.EQUATION_CONSTRAINTS_DESCRIPTOR_NO_CONSTRAINT]
if(len(constraints) == 0):
    print("no constraints")

if(ICh9Eq18.datasets is None):
    ICh9Eq18.read_datasets_for_constraint_checking()

########################## symbolic derivatives ##########################

# replace the sympy local dictionary with the display names of variables
```

```
local_dict = ICh9Eq18.equation.get_sympy_eq_local_dict()

# parse the provided candidate expression
# will use display names if specified
expr = sympy.parse_expr(expression, evaluate=False, local_dict= local_dict)

#calculate all first order partial derivatives of the expression
f_primes = [(sympy.Derivative(expr, var).doit(),var.name, 1)
            for var
            in local_dict.values()]

#calculate all second order partial derivatives of the expression
# (every possible combination [Hessian])
f_prime_mat = [[ (sympy.Derivative(f_prime, var).doit(),
                  [prime_var_name,var.name],
                  2 )

                  for var
                  in local_dict.values()]
                for (f_prime, prime_var_name, _)
                in f_primes]

#flatten 2d Hessian to 1d list and combine them
f_prime_mat_flattened = [item for sublist in f_prime_mat for item in sublist]
derviatives = f_primes+f_prime_mat_flattened


########################## JAX derivatives #########################

# replace the sympy local dictionary with the display names of variables
var_names = [v.name for v in ICh9Eq18.equation.get_vars()]

g = jax.jit(jax.grad(f))
hessian = jax.jit(jax.hessian(f))

########################## check constraints #########################
sympy_violated_constraints = []
JAX_violated_constraints = []
#check for all existing constraints if they are met
for constraint in constraints:
  #every constraint has a specific input range in which they apply
  xs = ICh9Eq18.datasets[constraint[sk.EQUATION_CONSTRAINTS_ID_KEY]]

  ########################## SymPy ##########################
  matches = [ derivative for (derivative, var, _) in derviatives
             if var == constraint[sk.EQUATION_CONSTRAINTS_VAR_NAME_KEY]]
  derivative = matches[0]
  f = sympy.lambdify(local_dict.keys(), derivative, "numpy")
  #calculate gradient per data point
  # gradients = np.array([ f(*row) for row in xs ])
  # speedup of 5:
  f_v = np.vectorize(f)
  gradients_sympy = f_v(*(xs.T))
  descriptor_sympy = base.get_constraint_descriptor_for_gradients(gradients_sympy)

  ########################## JAX ##########################
```

```python
    var_name_constraint = constraint[sk.EQUATION_CONSTRAINTS_VAR_NAME_KEY]
    descriptor_JAX = sk.EQUATION_CONSTRAINTS_DESCRIPTOR_UNKOWN_CONSTRAINT

    # checking the different types of constraints supported
    if(constraint[sk.EQUATION_CONSTRAINTS_ORDER_DERIVATIVE_KEY] == 1):
      #constraint is defined for the first order derivative
      # the signs of the functions gradient are to be checked for the input domain
      var_index = var_names.index(var_name_constraint)
      gradients = y = jax.vmap(g)(xs)
      var_gradients = gradients[:,var_index]
      descriptor_JAX = base.get_constraint_descriptor_for_gradients(var_gradients)

    elif(constraint[sk.EQUATION_CONSTRAINTS_ORDER_DERIVATIVE_KEY] == 2):
      var1_index = var_names.index(var_name_constraint[0])
      var2_index = var_names.index(var_name_constraint[1])
      hessian_gradients = jax.vmap(hessian)(xs)
      var_gradients = hessian_gradients[:,var1_index,var2_index]
      descriptor_JAX = base.get_constraint_descriptor_for_gradients(var_gradients)


    else:
      raise "constraint was available but it was not handled/checked"

    ############ gradients are almost equal ############
    if( not np.allclose(gradients_sympy,var_gradients)):
       print("Gradients do not match!")

    if(descriptor_sympy != constraint[sk.EQUATION_CONSTRAINTS_DESCRIPTOR_KEY]):
        sympy_violated_constraints.append(constraint)

    if(descriptor_JAX != constraint[sk.EQUATION_CONSTRAINTS_DESCRIPTOR_KEY]):
        JAX_violated_constraints.append(constraint)


print('SymPy Result:')
print((len(sympy_violated_constraints) == 0, sympy_violated_constraints))

print('JAX Result:')
print((len(JAX_violated_constraints) == 0, JAX_violated_constraints))
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for mor
e info.)
SymPy Result:
(True, [])
JAX Result:
(True, [])
```