Part III

Tutorial: NLOPT and canonical coordinates of quantum logic gates

Talk is cheap. Show me the code.

Linus Torvalds

NLOPT

NLopt is a free/open-source library for nonlinear optimization, providing a common interface for a number of different free optimization routines available online as well as original implementations of various other algorithms.

- Callable from C, C++, Fortran, Matlab or GNU Octave, Python, GNU Guile, Julia, GNU R, Lua, OCaml, Rust and Crystal.
- Maintained by https://github.com/stevengj/nlopt.

canonical coordinates of $\mathrm{U}(2)$

Sigma matrices

$$\sigma_0 = -\frac{i}{2}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \sigma_1 = \frac{1}{2}\begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix},$$
$$\sigma_2 = \frac{1}{2}\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix},\ \sigma_3 = \frac{1}{2}\begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}.$$

Unitary algebra of the rank one:

$$\mathfrak{u}(2) = \left\{ x^a \sigma_a : x^a \in \mathbb{R}^4 \right\}.$$

canonical coordinates of $\mathrm{U}(2)$

Canonical coordinates of the first kind:

$$\mathbb{R}^4 \to \mathrm{U}(2) : x^a \mapsto \exp(x^a \sigma_a)$$

Canonical coordinates of the second kind:

$$\mathbb{R}^4 \to \mathrm{u}(2) : x^a \mapsto \exp(x^0 \sigma_0) \exp(x^1 \sigma_1) \exp(x^2 \sigma_2) \exp(x^3 \sigma_3)$$

canonical coordinates of $\mathrm{U}(4)$

Sigma strings

$$\sigma_{ab} = 2i\sigma_a \otimes \sigma_b, \quad a, b = 0, 1, 2, 3.$$

Unitary algebra of the rank one:

$$\mathfrak{u}(4) = \left\{ x^{ab}\sigma_{ab} : x^{ab} \in \mathbb{R}^{16} \right\}.$$

canonical coordinates of $\mathrm{U}(4)$

Canonical coordinates of the first kind:

$$\mathbb{R}^{16} \to \mathrm{U}(4) : x^{ab} \mapsto \mathtt{exp}(x^{ab}\sigma_{ab}).$$

Canonical coordinates of the second kind:

$$\mathbb{R}^{16} \to \mathrm{U}(4) : x^{ab} \mapsto \prod_{a=0}^{3}\prod_{b=0}^{3} \mathtt{exp}(x^{ab}\sigma_{ab}).$$

implementations: class Sigma

```
1 class Sigma:
2     '''
3     Sigma class generates algebraic primitives required
      to compute the system of implicit ODEs for Euler
      angles.
4     '''
5   ...
```

usage: `Sigma(1,2)` $= \sigma_{12}$.

implementations: class Euler

```
1 class Euler():
2   '''
3   Euler class genetates various list of indices of sigma
       strings of the rank n.
4   '''
5   ...
```

usage: `Euler(1).basis = [[0],[1],[2],[3]]`.

implementations: function first_kind and second_kind

```
...
def first_kind(theta):
  '''
  returns an element of SU(2^n) from given canonical
    coordinates of the first kind whose generators are
    sigma strings.
    '''
...
def second_kind(theta):
'''
returns an element of SU(2^n) from given canonical
    coordinates of the second kind whose generators are
    sigma strings.
'''
...
```

usage:

`first_kind(theta1,theta2,theta3,theta4)`$=\exp(\theta^a \sigma_a)$

implementations: objective function

```
...
def diff_sum_grad_fk(theta):
  '''
  returns the diff_sum of a first kind matrix and a
    complex matrix with its gradient.
  The first element of the return is the gradient and
    the last element is the value of diff_sum.
  '''
...
def diff_sum_grad_sk(theta):
    '''
    returns the diff_sum of a second kind matrix and a
    complex matrix with its gradient.
    The first element of the return is the gradient and
    the last element is the value of diff_sum.
    '''
...
```

returns for the first kind

$$\partial_\theta \underset{i,j}{\text{mean}} \left(\| \exp(\theta^a \sigma_x)_{ij} - M_{ij} \|^2 \right), \; \underset{i,j}{\text{mean}} \left(\| \exp(\theta^a \sigma_x)_{ij} - M_{ij} \|^2 \right)$$

implementations: objective function

NLOPT objective function template:

```
def f(x,grad):
    if grad.size > 0:
       ...set grad to gradient, in-place...
    return ...value of f(x)...
```

Objective function of the first kind:

```
def obj_fk(x,grad):
    tf_x = tf.constant(x,dtype=tf.float64)
    if grad.size > 0:
      obj = fk_grad(tf_x)
      for i in range(grad.size):
        grad[i] = obj[0].numpy()[i]

    return obj[1].numpy()

```

implementations: quantum logic gates

```
1 h_gate = tf.cast(tf.constant([[1.,1.],[1.,-1.]])/tf.sqrt
    (2.),dtype=tf.complex128); # Hadarmard gate
2 s_gate = tf.constant([[1,0],[0,-1.0j]],dtype=tf.
    complex128); # S gate
3 t_gate = tf.constant(np.array([[1,0],[0,np.exp(1.0j*np.
    pi/8)]],dtype=np.complex128)); # T gate
4 cnot_gate = tf.constant(np.array
    ([[1,0,0,0],[0,1,0,0],[0,0,0,1],[0,0,1,0]],dtype=np.
    complex128)); #CNOT gate
```

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \; S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix},$$

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}, \; \mathrm{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

implementations: optimization algorithm

```
1 opt = nlopt.opt(nlopt.LD_LBFGS,16) # chooses an
    optimization algorithm.
2 opt.set_min_objective(obj_sk) # sets the objective
    function for the chosen optimization algorithm.
3 opt.set_xtol_rel(1e-5) # sets the tolerance for the
    convergence.
4 theta_opt_sk = opt.optimize(np.ones(16)) # sets the
    initial varaibles.
5
```