QAOA VRP Experiments

April 5, 2025

[58]:

```python
import numpy as np
import itertools

# Customers
n_customers = 3

# Location (depot and customer) - 2D example coordinate with no obstacles at all
locations = [(0, 0), (1, 2), (5, 3), (2, 5)]

# use euclidean distance
def distance(loc1, loc2):
    return np.sqrt((loc1[0] - loc2[0])**2 + (loc1[1] - loc2[1])**2)

# Creating distances matrix
n_locations = n_customers + 1
distance_matrix = np.zeros((n_locations, n_locations))
for i in range(n_locations):
    for j in range(n_locations):
        distance_matrix[i, j] = distance(locations[i], locations[j])

print("Distance Matrix:")
print(distance_matrix)
```

```
Distance Matrix:
[[0.         2.23606798 5.83095189 5.38516481]
 [2.23606798 0.         4.12310563 3.16227766]
 [5.83095189 4.12310563 0.         3.60555128]
 [5.38516481 3.16227766 3.60555128 0.        ]]
```

[59]:

```python
import numpy as np
import itertools

# exhaustive search as a base to compare with quantum
def exhaustive_search_vrp_with_operations(distance_matrix):
    n_customers = len(distance_matrix) - 1
    customers = list(range(1, n_customers + 1))
    best_route = None
    best_distance = float('inf')
```

```
    operation_count = 0 # count this for comparation with quantum purpose

    for route in itertools.permutations(customers):
        total_distance = distance_matrix[0, route[0]]  # Depot to first customers
        operation_count += 1  # Increment the count

        for i in range(len(route) - 1):
            total_distance += distance_matrix[route[i], route[i+1]]
            operation_count += 1

        total_distance += distance_matrix[route[-1], 0]  # Last customer to depot
        operation_count += 1

        if total_distance < best_distance:
            best_distance = total_distance
            best_route = route
            operation_count += 1 # also increment the count

    return best_route, best_distance, operation_count

best_route_classic, best_distance_classic, operation_count_classic =␣
 ↪exhaustive_search_vrp_with_operations(distance_matrix)

print("Best Route (Classical):", best_route_classic)
print("Total Distance (Classical):", best_distance_classic)
print("Num of Operation (Classical):", operation_count_classic)
```

```
Best Route (Classical): (1, 3, 2)
Total Distance (Classical): 14.83484880797746
Num of Operation (Classical): 26
```

[63]:

```
import pennylane as qml
import numpy as np
import itertools

# Assuming distance_matrix is already defined from previous blocks

n_customers = len(distance_matrix) - 1
n_locations = n_customers + 1
n_qubits = n_locations * n_locations  # Corrected: n_locations * n_locations

print(f"Number of qubits: {n_qubits}") # debug

# Quantum Representation: Encoding the problem into qubits
print("Quantum Representation: Each qubit represents a possible edge between␣
 ↪locations.")
print(f"Total qubits: {n_qubits}")
```

```python
for i in range(n_locations):
    for j in range(n_locations):
        print(f"Qubit {i * n_locations + j} represents edge from location {i} to {j}")

# Cost Hamiltonian (Objective Function)
def cost_hamiltonian():
    cost = 0
    for i in range(n_locations):
        for j in range(n_locations):
            if i != j:
                cost += distance_matrix[i, j] * (1 - qml.PauliZ(i * n_locations + j)) / 2
    return cost

print("\nCost Hamiltonian (Objective Function):")
print(cost_hamiltonian())

# Constraint Hamiltonian
def constraint_hamiltonian():
    constraint = 0
    for i in range(1, n_locations):  # Each customer visited once
        term = 0
        for j in range(n_locations):
            term += (1 - qml.PauliZ(j * n_locations + i)) / 2
        constraint += (term - 1) ** 2

    for i in range(n_locations):  # Each location left once
        term = 0
        for j in range(n_locations):
            term += (1 - qml.PauliZ(i * n_locations + j)) / 2
        constraint += (term - 1) ** 2
    return constraint * 10  # Penalty for constraints

# QAOA Circuit
def qaoa_circuit(params, wires):
    depth = len(params) // 2
    for w in wires:
        qml.Hadamard(wires=w)

    for d in range(depth):
        gamma = params[d]
        beta = params[depth + d]

        hamiltonian = qml.Hamiltonian([1], [cost_hamiltonian()])
        qml.evolve(hamiltonian, gamma, wires)
        for w in wires:
```

```python
            qml.RX(2 * beta, wires=w)

# Device
dev = qml.device("default.qubit", wires=n_qubits)

# Cost Function to optimize
@qml.qnode(dev)
def cost_function(params):
    qaoa_circuit(params, wires=range(n_qubits))
    cost_expval = qml.expval(cost_hamiltonian())
    constraint_expval = qml.expval(constraint_hamiltonian())
    return cost_expval, constraint_expval

def optimized_cost(params):
    cost, constraint = cost_function(params)
    return cost.item() + constraint.item()

# Optimizer
optimizer = qml.AdamOptimizer(stepsize=0.1)
depth = 2
params = np.random.rand(depth * 2)  # Initial gamma and beta

# QAOA Parameters (Start)
print("\nQAOA Parameters (Start):")
print(f"Depth: {depth}")
print(f"Initial Gamma: {params[:depth]}")
print(f"Initial Beta: {params[depth:]}")

# Optimization Loop
steps = 100
for i in range(steps):
    params = optimizer.step(optimized_cost, params)
    if (i + 1) % 10 == 0:
        print(f"Step {i+1}: Cost = {optimized_cost(params)}")

# QAOA Parameters (End)
print("\nQAOA Parameters (End):")
print(f"Optimal Gamma: {params[:depth]}")
print(f"Optimal Beta: {params[depth:]}")

# Quantum Solution
@qml.qnode(dev)
def get_probabilities(params):
    qaoa_circuit(params, wires=range(n_qubits))
    return [qml.probs(wires=i) for i in range(n_qubits)]

probabilities = get_probabilities(params)
```

```python
def interpret_pennylane_result(probabilities, n_customers):
    routes = []
    for i in range(n_customers + 1):
        for j in range(n_customers + 1):
            if i != j and probabilities[i * (n_customers + 1) + j][1] > 0.5:
                routes.append((i, j))
    return routes

routes_quantum = interpret_pennylane_result(probabilities, n_customers)
print("\nQuantum Solution (Routes):", routes_quantum)

# Calculate the distance for the Quantum Solution
def calculate_quantum_distance(route, distance_matrix):
  total_distance = 0
  start = 0
  for end in route:
    total_distance += distance_matrix[start, end[1]]
    start = end[1]
  total_distance += distance_matrix[start, 0]
  return total_distance

quantum_distance = calculate_quantum_distance(routes_quantum, distance_matrix)

print("Total Distance (Quantum):", quantum_distance)
```

```
Number of qubits: 16
Quantum Representation: Each qubit represents a possible edge between locations.
Total qubits: 16
Qubit 0 represents edge from location 0 to 0
Qubit 1 represents edge from location 0 to 1
Qubit 2 represents edge from location 0 to 2
Qubit 3 represents edge from location 0 to 3
Qubit 4 represents edge from location 1 to 0
Qubit 5 represents edge from location 1 to 1
Qubit 6 represents edge from location 1 to 2
Qubit 7 represents edge from location 1 to 3
Qubit 8 represents edge from location 2 to 0
Qubit 9 represents edge from location 2 to 1
Qubit 10 represents edge from location 2 to 2
Qubit 11 represents edge from location 2 to 3
Qubit 12 represents edge from location 3 to 0
Qubit 13 represents edge from location 3 to 1
Qubit 14 represents edge from location 3 to 2
Qubit 15 represents edge from location 3 to 3

Cost Hamiltonian (Objective Function):
1.118033988749895 * (-1 * Z(1) + 1 * I(1)) + 2.9154759474226504 * (-1 * Z(2) + 1
```

```
* I(2)) + 2.692582403567252 * (-1 * Z(3) + 1 * I(3)) + 1.118033988749895 * (-1 *
Z(4) + 1 * I(4)) + 2.0615528128088303 * (-1 * Z(6) + 1 * I(6)) +
1.5811388300841898 * (-1 * Z(7) + 1 * I(7)) + 2.9154759474226504 * (-1 * Z(8) +
1 * I(8)) + 2.0615528128088303 * (-1 * Z(9) + 1 * I(9)) + 1.8027756377319946 *
(-1 * Z(11) + 1 * I(11)) + 2.692582403567252 * (-1 * Z(12) + 1 * I(12)) +
1.5811388300841898 * (-1 * Z(13) + 1 * I(13)) + 1.8027756377319946 * (-1 * Z(14)
+ 1 * I(14))

QAOA Parameters (Start):
Depth: 2
Initial Gamma: [0.01563641 0.42340148]
Initial Beta: [0.39488152 0.29348817]
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
Cell In[63], line 92
     90 steps = 100
     91 for i in range(steps):
---> 92     params = optimizer.step(optimized_cost, params)
     93     if (i + 1) % 10 == 0:
     94         print(f"Step {i+1}: Cost = {optimized_cost(params)}")

File ␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\optimize gradient_desc
 ↪py:93, in GradientDescentOptimizer.step(self, objective_fn, grad_fn, *args,␣
 ↪**kwargs)
     75 def step(self, objective_fn, *args, grad_fn=None, **kwargs):
     76     """Update trainable arguments with one step of the optimizer.
     77 
     78     Args:
   (...)
     90         If single arg is provided, list [array] is replaced by array.
     91     """
---> 93     g, _ = self.compute_grad(objective_fn, args, kwargs, grad_fn=grad_fn)
     94     new_args = self.apply_grad(g, args)
     96     # unwrap from list if one argument, cleaner return

File ␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\optimize gradient_desc
 ↪py:122, in GradientDescentOptimizer.compute_grad(objective_fn, args, kwargs,␣
 ↪grad_fn)
    104 r"""Compute gradient of the objective function at the given point and␣
 ↪return it along with
    105 the objective function forward pass (if available).
    106 
   (...)
    119     will not be evaluted and instead ``None`` will be returned.
    120 """
```

```
    121 g = get_gradient(objective_fn) if grad_fn is None else grad_fn
--> 122 grad = g(*args, **kwargs)
    123 forward = getattr(g, "forward", None)
    125 num_trainable_args = sum(getattr(arg, "requires_grad", False) for arg in␣
 ↪args)

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\_grad.
 ↪py:163, in grad.__call__(self, *args, **kwargs)
    157 if not isinstance(argnum, int) and not argnum:
    158     warnings.warn(
    159         "Attempted to differentiate a function with no trainable␣
 ↪parameters. "
    160         "If this is unintended, please add trainable parameters via the '
    161         "'requires_grad' attribute or 'argnum' keyword."
    162     )
--> 163     self._forward = self._fun(*args, **kwargs)
    164     return ()
    166 grad_value, ans = grad_fn(*args, **kwargs)  # pylint: disable=not-callable

Cell In[63], line 75, in optimized_cost(params)
     74 def optimized_cost(params):
---> 75     cost, constraint = cost_function(params)
     76     return cost.item() + constraint.item()

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\workflow qnode.
 ↪py:1164, in QNode.__call__(self, *args, **kwargs)
   1162 if qml.capture.enabled():
   1163     return qml.capture.qnode_call(self, *args, **kwargs)
-> 1164 return self._impl_call(*args, **kwargs)

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\workflow qnode.
 ↪py:1150, in _impl_call(self, *args, **kwargs)
      0 <Error retrieving source code with stack_data see ipython/ipython#13598>

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\workflow qnode.
 ↪py:1103, in _execution_component(self, args, kwargs, override_shots)
      0 <Error retrieving source code with stack_data see ipython/ipython#13598>

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\workflow execution.
 ↪py:666, in execute(tapes, device, gradient_fn, interface, transform_program,␣
 ↪inner_transform, config, grad_on_execution, gradient_kwargs, cache, cachesize,
 ↪max_diff, override_shots, expand_fn, max_expansion, device_batch_transform,␣
 ↪device_vjp, mcm_config)
      0 <Error retrieving source code with stack_data see ipython/ipython#13598>
```

```
File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\workflow execution.
↪py:316, in inner_execute(tapes, **_)
      0 <Error retrieving source code with stack_data see ipython/ipython#13598>

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ odifiers\simu
↪py:30, in _track_execute.<locals>.execute(self, circuits, execution_config)
     28 @wraps(untracked_execute)
     29 def execute(self, circuits, execution_config=DefaultExecutionConfig):
---> 30     results = untracked_execute(self, circuits, execution_config)
     31     if isinstance(circuits, QuantumScript):
     32         batch = (circuits,)

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ odifiers\sing
↪py:32, in _make_execute.<locals>.execute(self, circuits, execution_config)
     30     is_single_circuit = True
     31     circuits = (circuits,)
---> 32 results = batch_execute(self, circuits, execution_config)
     33 return results[0] if is_single_circuit else results

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\logging\ ecorators.
↪py:61, in log_string_debug_func.<locals>.wrapper_entry(*args, **kwargs)
     54     s_caller = "::L".join(
     55         [str(i) for i in inspect.getouterframes(inspect.currentframe(),␣
↪2)[1][1:3]]
     56     )
     57     lgr.debug(
     58         f"Calling {f_string} from {s_caller}",
     59         **_debug_log_kwargs,
     60     )
---> 61 return func(*args, **kwargs)

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ efault_qubit.
↪py:597, in execute(self, circuits, execution_config)
    594     updated_values["gradient_method"] = gradient_method
    596 if execution_config.use_device_gradient is None:
--> 597     updated_values["use_device_gradient"] = gradient_method in {
    598         "adjoint",
    599         "backprop",
    600     }
    601 if execution_config.use_device_jacobian_product is None:
    602     updated_values["use_device_jacobian_product"] = gradient_method ==␣
↪"adjoint"
```

```
File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ efault_qubit.
↪py:598, in <genexpr>(.0)
    594     updated_values["gradient_method"] = gradient_method
    596 if execution_config.use_device_gradient is None:
    597     updated_values["use_device_gradient"] = gradient_method in {
--> 598         "adjoint",
    599         "backprop",
    600     }
    601 if execution_config.use_device_jacobian_product is None:
    602     updated_values["use_device_jacobian_product"] = gradient_method ==␣
↪"adjoint"

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ efault_qubit.
↪py:863, in _simulate_wrapper(circuit, kwargs)
      0 <Error retrieving source code with stack_data see ipython/ipython#13598>

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\logging\ ecorators.
↪py:61, in log_string_debug_func.<locals>.wrapper_entry(*args, **kwargs)
     54     s_caller = "::L".join(
     55         [str(i) for i in inspect.getouterframes(inspect.currentframe(),␣
↪2)[1][1:3]]
     56     )
     57     lgr.debug(
     58         f"Calling {f_string} from {s_caller}",
     59         **_debug_log_kwargs,
     60     )
---> 61 return func(*args, **kwargs)

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ ubit\simulate
↪py:359, in simulate(circuit, debugger, state_cache, **execution_kwargs)
    354     return tuple(results)
    356 ops_key, meas_key = jax_random_split(prng_key)
    357 state, is_state_batched = get_final_state(
    358     circuit, debugger=debugger, prng_key=ops_key, **execution_kwargs
--> 359 )
    360 if state_cache is not None:
    361     state_cache[circuit.hash] = state

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\logging\ ecorators.
↪py:61, in log_string_debug_func.<locals>.wrapper_entry(*args, **kwargs)
     54     s_caller = "::L".join(
     55         [str(i) for i in inspect.getouterframes(inspect.currentframe(),␣
↪2)[1][1:3]]
     56     )
```

```
     57     lgr.debug(
     58         f"Calling {f_string} from {s_caller}",
     59         **_debug_log_kwargs,
     60     )
---> 61 return func(*args, **kwargs)

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ ubit\simulate.py:241, in measure_final_state(circuit, state, is_state_batched, **execution_kwargs)
    219 @debug_logger
    220 def measure_final_state(circuit, state, is_state_batched, **execution_kwargs) -> Result:
    221     """
    222     Perform the measurements required by the circuit on the provided stat .
    223 
    224     This is an internal function that will be called by the successor to ``default.qubit``.
    225 
    226     Args:
    227         circuit (.QuantumScript): The single circuit to simulate. This circuit is assumed to have
    228             non-negative integer wire labels
    229         state (TensorLike): The state to perform measurement on
    230         is_state_batched (bool): Whether the state has a batch dimension or not.
    231         rng (Union[None, int, array_like[int], SeedSequence, BitGenerator, Generator]): A
    232             seed-like parameter matching that of ``seed`` for ``numpy.random.default_rng``.
    233             If no value is provided, a default RNG will be used.
    234         prng_key (Optional[jax.random.PRNGKey]): An optional ``jax.random.PRNGKey``. This is
    235             the key to the JAX pseudo random number generator. Only for simulation using JAX.
    236             If None, the default ``sample_state`` function and a ``numpy.random.default_rng``
    237             will be used for sampling.
    238         mid_measurements (None, dict): Dictionary of mid-circuit measurements
    239 
    240     Returns:
--> 241         Tuple[TensorLike]: The measurement results
    242     """
    244     rng = execution_kwargs.get("rng", None)
    245     prng_key = execution_kwargs.get("prng_key", None)
```

```
File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ ubit\simulate
↪py:242, in <genexpr>(.0)
    219 @debug_logger
    220 def measure_final_state(circuit, state, is_state_batched,␣
↪**execution_kwargs) -> Result:
    221     """
    222     Perform the measurements required by the circuit on the provided stat .
    223 
    224     This is an internal function that will be called by the successor to␣
↪``default.qubit``.
    225 
    226     Args:
    227         circuit (.QuantumScript): The single circuit to simulate. This␣
↪circuit is assumed to have
    228             non-negative integer wire labels
    229         state (TensorLike): The state to perform measurement on
    230         is_state_batched (bool): Whether the state has a batch dimension␣
↪or not.
    231         rng (Union[None, int, array_like[int], SeedSequence, BitGenerator,␣
↪Generator]): A
    232             seed-like parameter matching that of ``seed`` for ``numpy.
↪random.default_rng``.
    233             If no value is provided, a default RNG will be used.
    234         prng_key (Optional[jax.random.PRNGKey]): An optional ``jax.random.
↪PRNGKey``. This is
    235             the key to the JAX pseudo random number generator. Only for␣
↪simulation using JAX.
    236             If None, the default ``sample_state`` function and a ``numpy.
↪random.default_rng``
    237             will be used for sampling.
    238         mid_measurements (None, dict): Dictionary of mid-circuit␣
↪measurements
    239 
    240     Returns:
    241         Tuple[TensorLike]: The measurement results
--> 242     """
    244     rng = execution_kwargs.get("rng", None)
    245     prng_key = execution_kwargs.get("prng_key", None)

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ ubit\measure.
↪py:233, in measure(measurementprocess, state, is_state_batched)
    225 def measure(
    226     measurementprocess: MeasurementProcess, state: TensorLike,␣
↪is_state_batched: bool = False
    227 ) -> TensorLike:
    228     """Apply a measurement process to a state.
```

```
    229
    230     Args:
    231         measurementprocess (MeasurementProcess): measurement process to␣
 ↪apply to the state
    232         state (TensorLike): the state to measure
--> 233         is_state_batched (bool): whether the state is batched or not
    234
    235     Returns:
    236         Tensorlike: the result of the measurement
    237     """
    238     return get_measurement_function(measurementprocess, state)(
    239         measurementprocess, state, is_state_batched
    240     )

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\devices\ ubit\measure.
 ↪py:66, in state_diagonalizing_gates(measurementprocess, state, is_state_batche )
     53 def state_diagonalizing_gates(
     54     measurementprocess: StateMeasurement, state: TensorLike,␣
 ↪is_state_batched: bool = False
     55 ) -> TensorLike:
     56     """Apply a measurement to state when the measurement process has an␣
 ↪observable with diagonalizing gates.
     57
     58     Args:
   (...)
     64         TensorLike: the result of the measurement
     65     """
---> 66     for op in measurementprocess.diagonalizing_gates():
     67         state = apply_operation(op, state,␣
 ↪is_state_batched=is_state_batched)
     69     total_indices = len(state.shape) - is_state_batched

File ~\AppData\Local\Programs\Python\Python311\Lib\contextlib.py:81, in␣
 ↪ContextDecorator.__call__.<locals>.inner(*args, **kwds)
     78 @wraps(func)
     79 def inner(*args, **kwds):
     80     with self._recreate_cm():
---> 81         return func(*args, **kwds)

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\measurem nts\measureme
 ↪py:351, in diagonalizing_gates(self)
    348 cls = self.__class__
    349 copied_m = cls.__new__(cls)
--> 351 for attr, value in vars(self).items():
    352     setattr(copied_m, attr, value)
    354 if self.obs is not None:
```

```
File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\ops\op_math\sprod.
↪py:226, in diagonalizing_gates(self)
    220 @handle_recursion_error
    221 def diagonalizing_gates(self):
    222     r"""Sequence of gates that diagonalize the operator in the␣
↪computational basis.
    223 
    224     Given the eigendecomposition :math:`O = U \Sigma U^{\dagger}` where
    225     :math:`\Sigma` is a diagonal matrix containing the eigenvalues,
--> 226     the sequence of diagonalizing gates implements the unitary :math:
↪`U^{\dagger}`.
    227 
    228     The diagonalizing gates rotate the state into the eigenbasis
    229     of the operator.
    230 
    231     A ``DiagGatesUndefinedError`` is raised if no representation by␣
↪decomposition is defined.
    232 
    233     .. seealso:: :meth:`~.Operator.compute_diagonalizing_gates`.
    234 
    235     Returns:
    236         list[.Operator] or None: a list of operators
    237     """
    238     return self.base.diagonalizing_gates()

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\ops\op_math\composite.
↪py:287, in diagonalizing_gates(self)
    284     # the lists of ops with multiple operators can be handled if there is␣
↪a matrix
    285     return self.has_matrix
--> 287 return all(op.has_diagonalizing_gates for op in self)

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\ops\op_math\composite.
↪py:246, in eigendecomposition(self)
    244     else:
    245         i += 1
--> 246 if first_group_idx is not None:
    247     groups[first_group_idx][0].append(op)
    248 else:
    249     # Create new group

File␣
↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\ops\op_math\sum.
↪py:330, in matrix(self, wire_order)
    316 @handle_recursion_error
```

```
    317 def matrix(self, wire_order=None):
    318     r"""Representation of the operator as a matrix in the computational basis.
    319 
    320     If ``wire_order`` is provided, the numerical representation considers the position of the
    321     operator's wires in the global wire order. Otherwise, the wire order defaults to the
    322     operator's wires.
    323 
    324     If the matrix depends on trainable parameters, the result
    325     will be cast in the same autodifferentiation framework as the parameters.
    326 
    327     A ``MatrixUndefinedError`` is raised if the matrix representation has not been defined.
    328 
    329     .. seealso:: :meth:`~.Operator.compute_matrix`
--> 330 
    331     Args:
    332         wire_order (Iterable): global wire order, must contain all wire labels from the
    333         operator's wires
    334 
    335     Returns:
    336         tensor_like: matrix representation
    337     """
    338     if self.pauli_rep:
    339         return self.pauli_rep.to_mat(wire_order=wire_order or self.wires)

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\pauli\pauli_arithmetic.py:850, in to_mat(self, wire_order, format, buffer_size)
    848 n_wires = len(wire_order)
    849 matrix_size = 2**n_wires
--> 850 matrix = sparse.csr_matrix((matrix_size, matrix_size), dtype="complex128")
    851 op_sparse_idx = _ps_to_sparse_index(pauli_words, wire_order)
    852 _, unique_sparse_structures, unique_invs = np.unique(
    853     op_sparse_idx, axis=0, return_index=True, return_inverse=True
    854 )

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\pauli\pauli_arithmetic.py:911, in _to_dense_mat(self, wire_order)
    908 """Computes the matrix-vector product of the Pauli sentence with a state vector.
    909 See pauli_sparse_matrices.md for the technical details."""
    910 wire_order = self.wires if wire_order is None else Wires(wire_order)
```

```
--> 911 if not wire_order.contains_wires(self.wires):
    912     raise ValueError(
    913         "Can't get the matrix for the specified wire order because it "
    914         f"does not contain all the Pauli sentence's wires {self.wires}"
    915     )
    916 pauli_words = list(self)  # Ensure consistent ordering

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\pennylane\pauli\pa li_arithmetic
 ↪py:968, in _sum_same_structure_pws_dense(self, pauli_words, wire_order)
    966 coeff = self[pw]
    967 csr_data = pw._get_csr_data(wire_order, 1)
--> 968 ml_interface = qml.math.get_interface(coeff)
    969 if ml_interface == "torch":
    970     csr_data = qml.math.convert_like(csr_data, coeff)

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\scipy\sparse\_comp essed.
 ↪py:1050, in _cs_matrix.toarray(self, order, out)
   1048 if out is None and order is None:
   1049     order = self._swap('cf')[0]
-> 1050 out = self._process_toarray_args(order, out)
   1051 if not (out.flags.c_contiguous or out.flags.f_contiguous):
   1052     raise ValueError('Output array must be C or F contiguous')

File␣
 ↪~\AppData\Local\Programs\Python\Python311\Lib\site-packages\scipy\sparse\_base
 ↪py:1267, in _spbase._process_toarray_args(self, order, out)
   1265     return out
   1266 else:
-> 1267     return np.zeros(self.shape, dtype=self.dtype, order=order)

MemoryError: Unable to allocate 64.0 GiB for an array with shape (65536, 65536)␣
 ↪and data type complex128
```

[ ]: