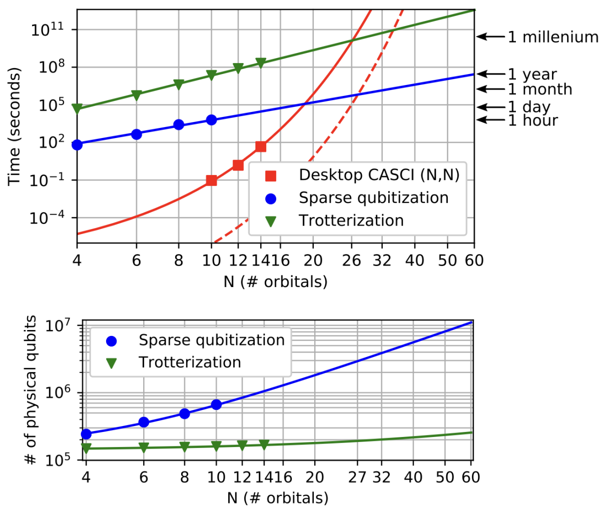
Appendix: Estimation of Compute Times as Functions of Problem Sizes

In [1], Fig. 3 shows estimates of compute resources, in time and number of physical qubits, for the CAS(N,N) problems solved by the three different methods: CASCI as the classical method, and the two quantum computational methods based on Trotterization and with further sparse qubitization. The authors then extrapolated these estimates to curves that approximate the scaling of compute resources of the three different methods.

Here, we recall Fig. 3 in [1],



Now, we wish to estimate the approximated functions that describe those curves (particularly, Trotterization and CASCI).

Let T be the compute time, and N be the number of orbitals, we then wish to find $f(\cdot)$ such that $T \approx f(N)$.

Since the x- and y-axis are log-scaled, we have

$$x = \log_2 N, y = \log_{10} T \tag{1}$$

1 Trotterization

For Trotterization, y is a linear function of x, passing through roughly $(2,5)$, $(3,6.5)$ and $(5.1,11)$. Hence, solving for the linear function with these points numerically, we have

$$y = 1.97x + 0.8. \tag{2}$$

This implies

$$\log_{10} T = 1.97 \log_2 N + 0.8, \tag{3}$$
$$\implies \quad T = (10^{0.8}) 10^{1.97 \log_2 N}. \tag{4}$$

Since $10^{\log_2 N} = N^{\log_2 10}$,

$$T = (10^{0.8}) N^{1.97 \log_2 10}, \tag{5}$$
$$\approx 6.31 \times N^{6.54}, \tag{6}$$

which is a polynomial growth of order 6.54. When we calculate the R-Squared for this fitting, we have about 0.97. The Python code for this estimation is attached at the end of this section.

2 Desktop CASCI

We can show that a linear function in the log-scaled plot is equivalent to a polynomial in the regular plot, in which the quantum computational methods all correspond to as seen in the previous sections.

For Desktop CASCI, the three estimated points see a growth in increment with problem size, suggesting T growing super-polynomially with N. Considering this pattern, we assume it to be an exponential function. As the first step, we observe that the curve passes through roughly $(2,-5.5)$, $(2.6,-4)$, and $(4.75,11)$ as the x- and y-axis are log-scaled for T and N, respectively.

2.1 Desktop CASCI: exponential function

Let y be an exponent of two of x, that is,

$$y = a * 2^{bx} + c. \tag{7}$$

Note that we use $y = 2^{bx}$ for a simpler interpretation of N. The results remain consistent when considering y as an exponential function of x.

Solving for a, b, and c with the observed points, we numerically have

$$y = 1.03 * 2^{0.901x} - 9.13. \tag{8}$$

Then, we have

$$\log_{10} T = 1.03 * 2^{0.901(\log_2 N)} - 9.13, \tag{9}$$

$$\implies \quad T = 10^{1.03N^{0.901} - 9.13}. \tag{10}$$

For the R-Squared values to the data and fitting curve, we have 0.9995. The Python code for this estimation is attached at the end of this section.

3 Intersection between Trotterizatio and Desktop CASCI

Utilizing the given estimates to approximate the underlying graphs depicted in Fig. 3 in [1], we compute the point of intersection between the Trotterization and CASCI curves.

$$(10^{0.8})10^{1.97 \log_2 N} = 10^{1.03N^{(0.901)} - 9.13} \tag{11}$$

We found $N \approx 25.76$ and $T \approx 1.13 * 10^{10}$. The Python code for this estimation is attached at the end of this section.

References

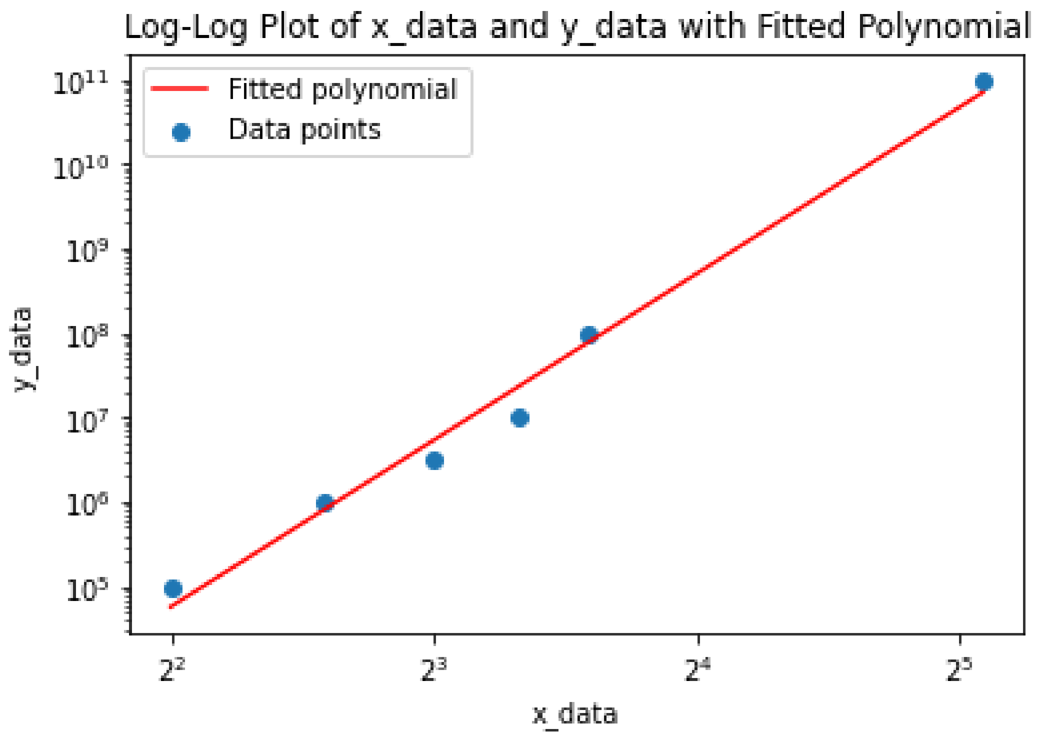
[1] V. E. Elfving et al. *How will quantum computers provide an industrially relevant computational advantage in quantum chemistry?* 2020. arXiv: `2009.12472 [quant-ph]`.

Appendix-Codes

April 11, 2023

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score

x_data = np.array([4, 6, 8, 10, 12, 34])
y_data = np.array([10**5, 10**6, 10**6.5, 10**7, 10**8, 10**11])

degree = 1

coefficients = np.polyfit(np.log2(x_data), np.log10(y_data), degree)
polynomial = np.poly1d(coefficients)

print(f"Fitted polynomial: {polynomial}")

# Calculate R-squared
y_pred = polynomial(np.log2(x_data))
r_squared = r2_score(np.log10(y_data), y_pred)
print("R-squared value:", r_squared)

fig, ax = plt.subplots()

# Plotting the data points
ax.scatter(x_data, y_data, label="Data points")

# Plotting the fitted polynomial
x_range = np.linspace(min(x_data), max(x_data), 1000)
y_fitted = 10**polynomial(np.log2(x_range))
ax.plot(x_range, y_fitted, label="Fitted polynomial", color="red")

# Set the x-axis to log base 2 scale
ax.set_xscale('log', base=2)

# Set the y-axis to log scale (10)
ax.set_yscale('log')

# Add labels for the axes
ax.set_xlabel('x_data')
ax.set_ylabel('y_data')

# Set the title of the figure
ax.set_title('Log-Log Plot of x_data and y_data with Fitted Polynomial')

# Display the legend
ax.legend()

# Display the figure
plt.show()
```

```
Fitted polynomial:
1.97 x + 0.8202
R-squared value: 0.9867653052858756
```

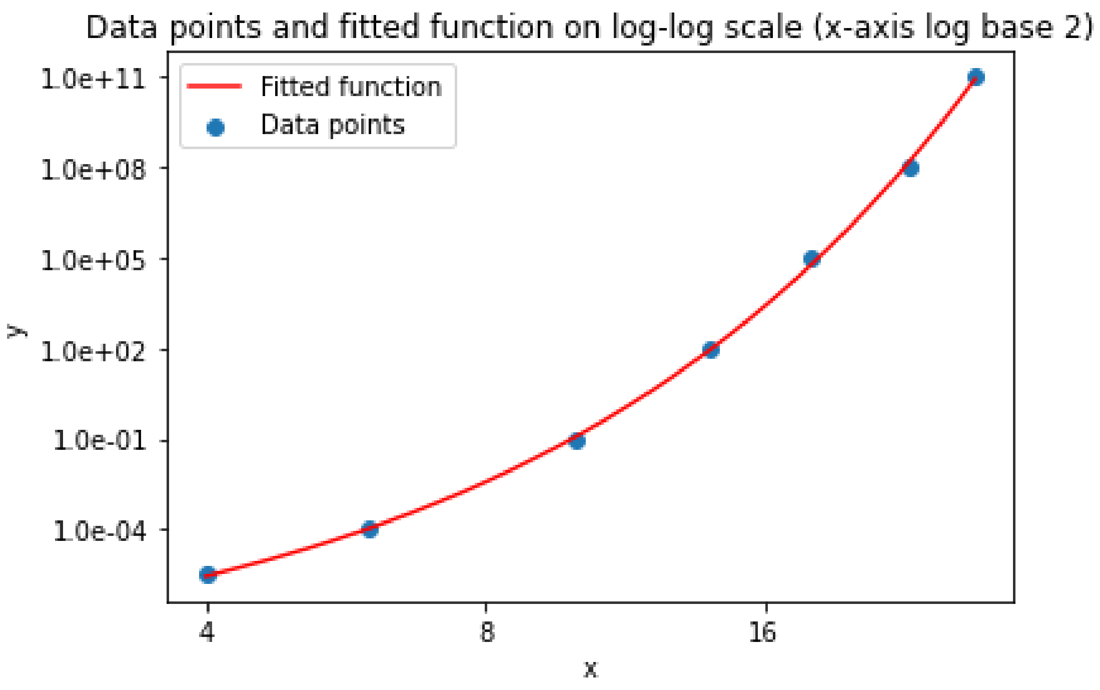


[2]:
```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

x_data = np.array([4, 6, 10, 14, 18, 23, 27])
y_data = np.array([10**-5.5, 10**-4, 10**-1, 90, 10**5, 10**8, 10**11])
```

```
x_log2 = np.log2(x_data)
y_log10 = np.log10(y_data)

def model_func(x, c1, c2, c3):
    return c1 * 2**(c2 * (x)) + c3

initial_params = (1, 1, 0)
params, _ = curve_fit(model_func, x_log2, y_log10, p0=initial_params,␣
 ↪maxfev=10000)

c1, c2, c3 = params
print(f"c1: {c1}, c2: {c2}, c3: {c3}")

fig, ax = plt.subplots()

# Plotting the data points
ax.scatter(x_data, y_data, label="Data points")

# Plotting the fitted function
x_range = np.linspace(min(x_data), max(x_data), 1000)
y_fitted = model_func(np.log2(x_range), c1, c2, c3)
ax.plot(x_range, 10**y_fitted, label="Fitted function", color="red")

# Set the x-axis and y-axis to log scale
ax.set_xscale('log', base=2)
ax.set_yscale('log')

# Formatter functions for x-axis and y-axis
x_formatter = FuncFormatter(lambda x, pos: f"{x:.0f}")
y_formatter = FuncFormatter(lambda y, pos: f"{y:.1e}")

# Set formatters for x-axis and y-axis labels
ax.xaxis.set_major_formatter(x_formatter)
ax.yaxis.set_major_formatter(y_formatter)

ax.set_xlabel("x")
ax.set_ylabel("y")
ax.legend()
ax.set_title("Data points and fitted function on log-log scale (x-axis log base␣
 ↪2)")
plt.show()

# Step 1: Calculate the predicted values using the fitted parameters and the␣
 ↪model function
y_pred = model_func(x_log2, c1, c2, c3)

# Step 2: Compute the residual sum of squares (RSS)
```

```
rss = np.sum((y_log10 - y_pred)**2)

# Step 3: Compute the total sum of squares (TSS)
tss = np.sum((y_log10 - np.mean(y_log10))**2)

# Step 4: Calculate the R-squared value using the formula R^2 = 1 - (RSS/TSS)
r_squared = 1 - (rss / tss)

print("R-squared value:", r_squared)
```

```
c1: 1.0272065192143498, c2: 0.9015460183549963, c3: -9.126604362730038
```



```
R-squared value: 0.9995144955416537
```

[3]:
```
import numpy as np
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

def CASCI_log(x):
    return 1.027 * 2**(0.902 * x) - 9.127

def Trotterization_log(x):
    return 1.97*x + 0.8202
```

```python
def intercept(x):
    return CASCI_log(x) - Trotterization_log(x)

# Initial guess for the x-value at which the functions intersect
x0 = 20

x_intercept = fsolve(intercept, x0)

y_intercept = CASCI_log(x_intercept)

intersection_point = (x_intercept[0], y_intercept[0])
print(f"x-intercept: {x_intercept[0]}, y-intercept: {y_intercept[0]}")
```

```
x-intercept: 4.680983654645295, y-intercept: 10.041737799651228
```

[4]:
```python
import numpy as np
from scipy.optimize import fsolve
import matplotlib.pyplot as plt

def CASCI(n):
    return 10**(1.027 * (n**0.901) - 9.126)

def Trotterization(n):
    return 10**(1.97*np.log2(n) +0.8202)

def intercept(n):
    return CASCI(n) - Trotterization(n)

# Initial guess for the x-value at which the functions intersect
x0 = 50

N_intercept = fsolve(intercept, x0)

T_intercept = CASCI(N_intercept)

intersection_point = (N_intercept[0], T_intercept[0])
print(f"N-intercept: {N_intercept[0]}, T-intercept: {T_intercept[0]}")
```

```
N-intercept: 25.760737798596633, T-intercept: 11318554241.165968
```