QuantumClifford.jl API Reference for CAMPS.jl

Essential Functions Only

This document contains only the QuantumClifford.jl functions needed for CAMPS.jl implementation. Each function includes its signature, what it returns, and a concrete example.

1. Pauli Operators

1.1 Creating Pauli Operators

String Macro: P"..."

```julia
using QuantumClifford

# Single-qubit Paulis
P"I"    # Identity
P"X"    # Pauli X
P"Y"    # Pauli Y
P"Z"    # Pauli Z

# Multi-qubit Paulis (tensor product, left-to-right = qubit 1 to n)
P"XYZ"   # X₁ ⊗ Y₂ ⊗ Z₃
P"X_Z"   # X₁ ⊗ I₂ ⊗ Z₃  (underscore = Identity)

# With phases
P"-XYZ"   # -1 × (X ⊗ Y ⊗ Z)
P"iXYZ"   # +i × (X ⊗ Y ⊗ Z)
P"-iXYZ"  # -i × (X ⊗ Y ⊗ Z)
```

Programmatic Creation: single_x, single_y, single_z

```julia
# Create single Pauli on qubit j of n-qubit system
# Signature: single_x(n, j), single_y(n, j), single_z(n, j)

single_z(4, 2)   # Returns: P"_Z__" (Z on qubit 2 of 4 qubits)
single_x(3, 1)   # Returns: P"X__" (X on qubit 1 of 3 qubits)
single_y(5, 3)   # Returns: P"__Y__" (Y on qubit 3 of 5 qubits)
```

Use in CAMPS.jl: Creating the original Pauli for twisted Pauli computation.

```julia
# When T-gate is applied to qubit q:
P_original = single_z(n, q)  # T-gate rotates around Z axis
```

1.2 Pauli Properties

Number of Qubits: nqubits

```julia
P = P"XYZ"
nqubits(P)   # Returns: 3
```

Binary Encoding: xbit and zbit

CRITICAL FOR GF(2) MATRIX CONSTRUCTION

```julia
# xbit(P) returns Bool vector: true where Pauli has X or Y component
# zbit(P) returns Bool vector: true where Pauli has Y or Z component

P = P"IXYZ"

xbit(P)   # Returns: Bool[0, 1, 1, 0]
          #          I→0, X→1, Y→1, Z→0

zbit(P)   # Returns: Bool[0, 0, 1, 1]
```

```
            #            I→0, X→0, Y→1, Z→1
```

Pauli encoding table:

Pauli	xbit	zbit
I	0	0
X	1	0
Y	1	1
Z	0	1

Use in CAMPS.jl: Building the GF(2) matrix for bond dimension prediction.

```
function build_gf2_matrix(twisted_paulis::Vector{PauliOperator})
    t = length(twisted_paulis)
    n = nqubits(twisted_paulis[1])
    M = zeros(Bool, t, n)
    for k in 1:t
        M[k, :] = xbit(twisted_paulis[k])  # This is the key line!
    end
    return M
end
```

Indexing: P[j]

```
# P[j] returns (x_bit, z_bit) tuple for qubit j

P = P"IXYZ"

P[1]   # Returns: (false, false)  → I
P[2]   # Returns: (true, false)   → X
P[3]   # Returns: (true, true)    → Y
P[4]   # Returns: (false, true)   → Z
```

Use in CAMPS.jl: Building OFD disentangler gates.

```
function get_pauli_at(P::PauliOperator, j::Int)::Symbol
    x, z = P[j]
    !x && !z && return :I
    x && !z  && return :X
    x && z   && return :Y
    return :Z
end
```

Phase Access: P.phase[]

```
# P.phase[] returns UInt8 encoding the phase
# 0x00 = +1
# 0x01 = +i
# 0x02 = -1
# 0x03 = -i
```

```julia
P"XYZ".phase[]    # Returns: 0x00 (+1)
P"-XYZ".phase[]   # Returns: 0x02 (-1)
P"iXYZ".phase[]   # Returns: 0x01 (+i)
P"-iXYZ".phase[]  # Returns: 0x03 (-i)
```

Use in CAMPS.jl: Applying Pauli string phase to MPS.

```julia
function phase_to_complex(phase_byte::UInt8)::ComplexF64
    phases = (1.0+0.0im, 0.0+1.0im, -1.0+0.0im, 0.0-1.0im)
    return phases[phase_byte + 1]
end

# Usage:
phase = phase_to_complex(P.phase[])
mps = phase * mps
```

2. Stabilizer States

Creating Stabilizers: Stabilizer

```julia
# Stabilizer wraps a list of commuting Pauli operators
# Used to represent quantum states via their stabilizer generators

# From Pauli operators:
s = Stabilizer([P"ZZ", P"XX"])   # Bell state |00⟩ + |11⟩

# Identity stabilizer (|0⟩^⊗n state):
one(Stabilizer, n)   # Has stabilizers Z₁, Z₂, ..., Zₙ

# Example:
one(Stabilizer, 3)   # Stabilizers: Z__, _Z_, __Z (i.e., |000⟩)
```

Accessing Stabilizers

```julia
s = Stabilizer([P"ZZ", P"XX"])

s[1]         # Returns: P"ZZ" (first stabilizer, a PauliOperator)
s[2]         # Returns: P"XX" (second stabilizer)
length(s)    # Returns: 2
```

Use in CAMPS.jl: Wrapping a Pauli to transform it through a Clifford.

```julia
function commute_pauli_through_clifford(P::PauliOperator, C)
    stab = Stabilizer([P])   # Wrap single Pauli in Stabilizer
    # ... apply Clifford ...
    return stab[1]           # Extract transformed Pauli
end
```

3. Clifford Operators

3.1 MixedDestabilizer (For Tracking Accumulated Clifford)

```julia
# MixedDestabilizer tracks both stabilizers AND destabilizers
# This is needed for full Clifford operator representation

# Create from stabilizer state:
md = MixedDestabilizer(one(Stabilizer, n))

# This represents the identity Clifford on n qubits
```

```
Use in CAMPS.jl: Storing the accumulated Clifford C in CAMPSState.
mutable struct CAMPSState
    clifford::MixedDestabilizer   # Accumulated Clifford
    # ...
end

# Initialize:
state.clifford = MixedDestabilizer(one(Stabilizer, n))

3.2 CliffordOperator (Dense Tableau)
# CliffordOperator is a dense representation of a Clifford
# Can be created from MixedDestabilizer or symbolic gates

# From MixedDestabilizer:
cliff = CliffordOperator(md)

# Identity:
one(CliffordOperator, n)

# Number of qubits:
nqubits(cliff)   # Returns: n

3.3 Inverting Cliffords: inv
CRITICAL FOR TWISTED PAULI COMPUTATION
# inv(cliff) returns the inverse Clifford operator

cliff = CliffordOperator(sHadamard(1))
cliff_inv = inv(cliff)

# cliff_inv represents H† = H (Hadamard is self-inverse)

Use in CAMPS.jl: Computing twisted Paulis P_twisted = C† · P · C.
function commute_pauli_through_clifford(P::PauliOperator, C::MixedDestabilizer)
    stab = Stabilizer([P])
    C_inv = inv(CliffordOperator(C))   # Get C†
    apply!(stab, C_inv)                  # Compute C† · P · C
    return stab[1]
end

3.4 Applying Cliffords: apply!
CRITICAL FUNCTION - UNDERSTAND THIS CAREFULLY
# apply!(stabilizer, clifford) modifies stabilizer IN-PLACE
# It computes: clifford · stabilizer · clifford†

# Signature:
apply!(stab::Stabilizer, cliff::CliffordOperator)  # Returns modified stab
apply!(md::MixedDestabilizer, gate::SymbolicGate)  # Returns modified md

# Example 1: Hadamard transforms Z → X
stab = Stabilizer([P"Z"])
apply!(stab, CliffordOperator(sHadamard(1)))
stab[1]   # Returns: P"X" (because H·Z·H† = X)

# Example 2: Hadamard transforms X → Z
```

```
stab = Stabilizer([P"X"])
apply!(stab, CliffordOperator(sHadamard(1)))
stab[1]   # Returns: P"Z" (because H·X·H† = Z)

# Example 3: CNOT propagation
stab = Stabilizer([P"_Z"])   # Z on qubit 2
apply!(stab, CliffordOperator(sCNOT(1, 2)))
stab[1]   # Returns: P"ZZ" (because CNOT·Z₂·CNOT† = Z₁Z₂)
```

Key insight for twisted Paulis:
We want: C† · P · C
apply!(stab, cliff) computes: cliff · stab · cliff†
So we use: apply!(stab, inv(C)) which computes: C⁻¹ · P · (C⁻¹)† = C† · P · C ✓
Use in CAMPS.jl:

```
# For twisted Pauli computation:
function commute_pauli_through_clifford(P::PauliOperator, C::MixedDestabilizer)
    stab = Stabilizer([P])
    apply!(stab, inv(CliffordOperator(C)))  # Computes C† · P · C
    return stab[1]
end

# For accumulating Clifford gates:
function apply_clifford_gate!(state::CAMPSState, gate)
    apply!(state.clifford, gate)  # Composes gate into accumulated Clifford
end
```

4. Symbolic Gates

These are efficient representations of common Clifford gates. Use these when building circuits and applying gates.

4.1 Single-Qubit Gates

```
# All take qubit index (1-indexed)

sHadamard(q)    # Hadamard on qubit q
sPhase(q)       # S gate (√Z) on qubit q
sInvPhase(q)    # S† gate on qubit q
sX(q)           # Pauli X on qubit q
sY(q)           # Pauli Y on qubit q
sZ(q)           # Pauli Z on qubit q

# Examples:
sHadamard(1)    # H on qubit 1
sPhase(3)       # S on qubit 3
sInvPhase(2)    # S† on qubit 2
```

4.2 Two-Qubit Gates

```
sCNOT(control, target)   # CNOT: control controls, target flips
sCPHASE(q1, q2)          # CZ gate (symmetric)
sSWAP(q1, q2)            # SWAP gate

# Examples:
sCNOT(1, 2)     # CNOT with control=1, target=2
sCPHASE(2, 3)   # CZ on qubits 2 and 3
sSWAP(1, 4)     # SWAP qubits 1 and 4
```

4.3 Applying Symbolic Gates to MixedDestabilizer

```
md = MixedDestabilizer(one(Stabilizer, 3))

# Apply gates directly:
apply!(md, sHadamard(1))
apply!(md, sCNOT(1, 2))
apply!(md, sPhase(3))

# Gates compose into the accumulated Clifford
```

4.4 Inverting Symbolic Gates

```
# inv() works on symbolic gates too

inv(sHadamard(1))    # Returns sHadamard(1) (self-inverse)
inv(sPhase(1))       # Returns sInvPhase(1)
inv(sInvPhase(1))    # Returns sPhase(1)
inv(sCNOT(1, 2))     # Returns sCNOT(1, 2) (self-inverse)
inv(sCPHASE(1, 2))   # Returns sCPHASE(1, 2) (self-inverse)
```

Use in CAMPS.jl: Building OFD disentangler and applying its inverse.

```
function build_ofd_gates(P::PauliOperator, control::Int, n::Int)
    gates = []
    for j in 1:n
        j == control && continue
        x, z = P[j]

        if x && !z       # X
            push!(gates, sCNOT(control, j))
        elseif x && z    # Y: CY = S†·CNOT·S
            push!(gates, sInvPhase(j))
            push!(gates, sCNOT(control, j))
            push!(gates, sPhase(j))
        elseif !x && z   # Z
            push!(gates, sCPHASE(control, j))
        end
    end
    return gates
end

# Apply inverse to accumulated Clifford:
function apply_disentangler_inverse!(clifford, gates)
    for g in reverse(gates)
        apply!(clifford, inv(g))
    end
end
```

5. Clifford Enumeration

5.1 All n-Qubit Cliffords: enumerate_cliffords

CRITICAL FOR OBD ALGORITHM

```
# enumerate_cliffords(n) returns an iterator over all n-qubit Cliffords

# Single-qubit: 24 Cliffords
length(collect(enumerate_cliffords(1)))    # Returns: 24
```

```
# Two-qubit: 11,520 Cliffords (but 720 up to single-qubit gates)
all_2q = collect(enumerate_cliffords(2))
length(all_2q)   # Returns: 11520

# Usage - iterate directly:
for cliff in enumerate_cliffords(2)
    # cliff is a CliffordOperator
    # Do something with it
end

# Or collect into array:
two_qubit_cliffords = collect(enumerate_cliffords(2))
```

Use in CAMPS.jl: OBD optimization loop.

```
function disentangle_obd!(state, strategy)
    all_cliffs = collect(enumerate_cliffords(2))

    for bond in 1:(n-1)
        best_cliff = nothing
        best_entropy = current_entropy

        for cliff in all_cliffs
            # Try this Clifford, measure entropy
            # Keep track of best
        end

        if best_cliff !== nothing
            # Apply best Clifford
        end
    end
end
```

5.2 Random Clifford: random_clifford

```
# random_clifford(n) returns a random n-qubit Clifford
# random_clifford(rng, n) uses specified RNG

cliff = random_clifford(3)           # Random 3-qubit Clifford
cliff = random_clifford(rng, 2)      # With specific RNG

# Returns: CliffordOperator
```

Use in CAMPS.jl: Generating random circuits for benchmarks.

```
function random_clifford_t_circuit(n, depth, t_density; rng=Random.GLOBAL_RNG)
    gates = []
    for layer in 1:depth
        for q in 1:n
            if rand(rng) < t_density
                push!(gates, TGate(q))
            else
                cliff = random_clifford(rng, 1)  # Random 1-qubit Clifford
                push!(gates, CliffordGate(cliff, q))
            end
        end
    end
    return gates
```

```
end
```

6. GF(2) Linear Algebra

6.1 Gaussian Elimination: gf2_gausselim!

```
# gf2_gausselim!(M) performs Gaussian elimination over GF(2) IN-PLACE
# GF(2) means binary field: addition = XOR, multiplication = AND

M = Bool[1 1 0;
         0 1 1;
         1 0 1]

gf2_gausselim!(M)   # Modifies M in-place

# After elimination, M is in row echelon form
# Non-zero rows indicate linearly independent rows
```

Use in CAMPS.jl: Computing GF(2) rank for bond dimension prediction.

```
function gf2_rank(M::Matrix{Bool})::Int
    isempty(M) && return 0

    M_copy = copy(M)           # Don't modify original
    gf2_gausselim!(M_copy)     # QuantumClifford function

    # Count non-zero rows = rank
    rank = 0
    for row in 1:size(M_copy, 1)
        if any(M_copy[row, :])
            rank += 1
        end
    end
    return rank
end

# Then use for prediction:
function predict_bond_dimension(twisted_paulis)
    M = build_gf2_matrix(twisted_paulis)  # Uses xbit()
    t = size(M, 1)
    r = gf2_rank(M)                        # Uses gf2_gausselim!()
    return 2^(t - r)
end
```

7. Complete Usage Examples

Example 1: Twisted Pauli Computation

```
using QuantumClifford

# Setup: 3-qubit system with some Clifford gates applied
n = 3
C = MixedDestabilizer(one(Stabilizer, n))

# Apply H on qubit 1, CNOT(1,2)
apply!(C, sHadamard(1))
apply!(C, sCNOT(1, 2))
```

```
# Now compute twisted Pauli for T-gate on qubit 2
P_original = single_z(n, 2)   # Z on qubit 2

# Compute C† · Z₂ · C
stab = Stabilizer([P_original])
apply!(stab, inv(CliffordOperator(C)))
P_twisted = stab[1]

println(P_twisted)  # Will show the transformed Pauli
```

Example 2: Check Disentanglability

```
using QuantumClifford

P = P"XZY"  # Twisted Pauli
free_qubits = BitVector([true, false, true])  # Qubits 1 and 3 are free

# Check which free qubits have X or Y
x = xbit(P)   # [true, false, true] for X, Z, Y

for j in 1:3
    if free_qubits[j] && x[j]
        println("Can disentangle using qubit $j as control")
        break
    end
end
# Output: "Can disentangle using qubit 1 as control"
```

Example 3: Build OFD Disentangler

```
using QuantumClifford

P = P"XZY"    # Control on qubit 1 (has X)
control = 1
n = 3

gates = []
for j in 1:n
    j == control && continue

    x, z = P[j]

    if x && !z      # X → CNOT
        push!(gates, sCNOT(control, j))
    elseif !x && z  # Z → CZ
        push!(gates, sCPHASE(control, j))
    elseif x && z   # Y → CY = S†·CNOT·S
        push!(gates, sInvPhase(j))
        push!(gates, sCNOT(control, j))
        push!(gates, sPhase(j))
    end
end

# gates now contains: [sCPHASE(1,2), sInvPhase(3), sCNOT(1,3), sPhase(3)]
# (CZ for Z on qubit 2, CY for Y on qubit 3)
```

Example 4: GF(2) Rank Computation

```julia
using QuantumClifford

# Three twisted Paulis
paulis = [P"XZI", P"ZXI", P"YYI"]

# Build GF(2) matrix using xbit
t = length(paulis)
n = nqubits(paulis[1])
M = zeros(Bool, t, n)

for k in 1:t
    M[k, :] = xbit(paulis[k])
end

# M = [1 0 0;    # XZI → xbit = [1,0,0]
#      0 1 0;    # ZXI → xbit = [0,1,0]
#      1 1 0]    # YYI → xbit = [1,1,0]

# Compute rank
M_copy = copy(M)
gf2_gausselim!(M_copy)
rank = count(row -> any(M_copy[row, :]), 1:t)

# rank = 2 (row 3 = row 1 XOR row 2 in GF(2))
# Predicted χ = 2^(3-2) = 2
```

Example 5: OBD Loop Structure

```julia
using QuantumClifford

# Get all 2-qubit Cliffords
all_cliffs = collect(enumerate_cliffords(2))
println("Number of 2-qubit Cliffords: ", length(all_cliffs))

# In OBD, iterate through them:
for cliff in all_cliffs
    # cliff is a CliffordOperator
    # Convert to matrix, apply to MPS, measure entropy
    # (ITensor conversion handled separately)
end
```

8. Summary: Functions Used in CAMPS.jl

Function	Purpose in CAMPS.jl
`P"..."`	Create Pauli strings for testing
`single_x/y/z(n, j)`	Create original Pauli for twisted Pauli computation
`nqubits(P)`	Get system size from Pauli
`xbit(P)`	Build GF(2) matrix rows
`P[j]`	Get Pauli at position j for OFD gate construction
`P.phase[]`	

Get phase for MPS application
Stabilizer([P])
Wrap Pauli for Clifford transformation
one(Stabilizer, n)
Initialize identity Clifford
MixedDestabilizer(stab)
Store accumulated Clifford
CliffordOperator(md)
Convert for inversion
inv(cliff)
Invert Clifford for twisted Pauli computation
apply!(stab, cliff)
Compute cliff · stab · cliff†
apply!(md, gate)
Accumulate Clifford gates
sHadamard/sPhase/sInvPhase(q)
Single-qubit Clifford gates
sCNOT(c,t)/sCPHASE(q1,q2)
Two-qubit Clifford gates
enumerate_cliffords(2)
All two-qubit Cliffords for OBD
random_clifford(rng, n)
Random Cliffords for benchmarks
gf2_gausselim!(M)
GF(2) rank computation

9. Import Statement for CAMPS.jl

```
using QuantumClifford:
    # Types
    PauliOperator, Stabilizer, MixedDestabilizer, CliffordOperator,

    # Pauli creation
    single_x, single_y, single_z,

    # Pauli properties
    nqubits, xbit, zbit,

    # Stabilizer/Clifford operations
    apply!, inv,

    # Symbolic gates
    sHadamard, sPhase, sInvPhase, sX, sY, sZ,
    sCNOT, sCPHASE, sSWAP,

    # Enumeration
    enumerate_cliffords, random_clifford,

    # GF(2)
    gf2_gausselim!
```