FockMap Library Cookbook: A Progressive Tutorial for Symbolic Fock-Space Operator Algebra and Fermion-to-Qubit Encodings in F#

John Azariah*[1]

[1]University of Technology Sydney

February 2026

Abstract

This document is the companion walkthrough for the FockMap library Azariah [2026], a composable functional framework for symbolic operator algebra and fermion-to-qubit encodings. Organized as 13 progressive chapters, it covers every public type, function, and workflow in the library: from single-qubit Pauli operators through the three-level algebraic hierarchy (`C`/`P`/`S`), indexed operators, creation and annihilation, fermionic and bosonic normal ordering, five built-in encodings, custom encoding schemes, tree-based encoding via Fenwick and balanced trees, full Hamiltonian construction, and mixed bosonic–fermionic systems. Each chapter introduces concepts through worked F# code examples that the reader can execute directly. The final chapter ties everything together with a capstone script that encodes the H_2 molecule under three encodings and compares Pauli weight and term count.

This document is also available as hosted Markdown documentation at the repository website.

Contents

*ORCID: 0009-0007-9870-1970

1 Hello, Qubit

Everything in quantum computing eventually becomes a **Pauli operator**. These are 2×2 matrices that act on a single qubit:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{1}$$

In FockMap they are a simple discriminated union:

```
#r "nuget:␣FockMap"
open Encodings
open System.Numerics

let identity = I
let bitFlip = X
let combined = Y
let phase = Z
```

Multiplying two Paulis always yields another Pauli *times a phase*. The algebra is exact—no floating point involved:

```
let (result, phase) = X * Y
// result = Z, phase = Pi because XY = iZ

let (result2, _) = Y * X
// result2 = Z, phase = Mi because YX = -iZ (anti-commutation!)

let (result3, _) = X * X
// result3 = I, phase = P1 every Pauli squares to identity
```

Notice that $X \cdot Y \neq Y \cdot X$—they differ by a sign. This is the **anti-commutation** property, fundamental to quantum mechanics.

1.1 Phases without floating point

The four phase values $\{+1, -1, +i, -i\}$ live in their own type:

```
// Phase is a discriminated union: P1 (+1), M1 (-1), Pi (+i), Mi (-i)
Pi * Pi // M1 because i * i = -1
M1 * Mi // Pi because (-1) * (-i) = +i
P1 * M1 // M1 the identity doesn't change anything
```

When you need to fold a phase into a complex number:

```
let c = Complex(2.0, 0.0)
Pi.FoldIntoGlobalPhase c // Complex(0.0, 2.0)
M1.FoldIntoGlobalPhase c // Complex(-2.0, 0.0)
```

Key insight: FockMap tracks phases symbolically using `Phase` and only converts to floating-point `Complex` at the boundaries. This eliminates the rounding errors that plague naïve Pauli algebra implementations.

2 Building Expressions: The C / P / S Hierarchy

Real quantum operators are **sums of products** of operators, each with a coefficient. FockMap represents this with three nested types:

Type	Role	Analogy
`C<'T>`	Coefficient × single operator	A letter with emphasis
`P<'T>`	Product of operators	A word (ordered sequence)
`S<'T>`	Sum of products	A sentence

These types are **generic**—they work with any operator type.

2.1 C — a single weighted operator

```
let one_x = C<Pauli>.Apply X // 1 * X
let half_y = C<Pauli>.Apply(Complex(0.5, 0.0), Y)
```

2.2 P — an ordered product (tensor product)

```
let xy = one_x * half_y
// P<Pauli> with Coeff = 0.5, Units = [X; Y]

let xzy = P<Pauli>.Apply [| X; Z; Y |]
// 1 * (X (x) Z (x) Y)

let scaled = xzy.ScaleCoefficient(Complex(3.0, 0.0))
// 3 * (X (x) Z (x) Y)
```

Reduction normalises internal coefficients into the single overall coefficient:

```
let mixed = P<Pauli>.Apply(Complex(2.0, 0.0), [| half_y; one_x |])
let clean = mixed.Reduce.Value
// Coeff = 1.0, Units = [Y; X]
```

2.3 S — a sum of products (the Hamiltonian shape)

```
let s1 = S<Pauli>.Apply(P<Pauli>.Apply [| X; Z |])
let s2 = S<Pauli>.Apply(P<Pauli>.Apply [| Y; I |])

let hamiltonian = s1 + s2
let doubled = s1 + s1 // 2*(X (x) Z)
```

`S<'T>` stores its terms in a `Map<string, P<'T>>`, keyed by string representation. Like terms combine automatically: this is what makes `s1 + s1` produce $2 \cdot (X \otimes Z)$ instead of two separate entries.

2.4 Coefficient hygiene and zero propagation

Every level has a `Reduce` method that replaces `NaN` and infinity with zero, preventing numerical corruption from propagating. A product containing any zero-coefficient unit becomes the zero product eagerly, so downstream code never wastes time on trivial terms.

3 Operators on Specific Qubits

The `IxOp` type tags each operator with a mode index:

```
let x0 = IxOp<uint32, Pauli>.Apply(0u, X) // "X on qubit 0"
let z3 = IxOp<uint32, Pauli>.Apply(3u, Z) // "Z on qubit 3"
```

3.1 Parsing from strings

```
let parsed = Pauli.FromString "(X,␣2)"

let term = PIxOp<uint32, Pauli>.TryCreateFromString
             Pauli.Apply "[(X,␣0)|(Z,␣3)]"

let expr = SIxOp<uint32, Pauli>.TryCreateFromString
```

```
            Pauli.Apply "{[(X,␣0)|(Z,␣1)];␣[(Y,␣0)|(I,␣1)]}"
```

The format uses `[...|...]` for products and `{...; ...}` for sums. You pass a parser function for the underlying operator type.

4 Creation and Annihilation

Quantum chemistry works with **ladder operators**: creation ($a^\dagger$) and annihilation (a):

```
let create = Raise // a+
let destroy = Lower // a
let nothing = Identity // I
```

4.1 Indexed ladder operators

```
let adag2 = LadderOperatorUnit.FromUnit(true, 2u) // a+_2
let a1 = LadderOperatorUnit.FromUnit(false, 1u) // a_1
```

4.2 Product terms

A typical quantum chemistry term: $a_0^\dagger a_1^\dagger a_1 a_0$:

```
let twoBody = LadderOperatorProductTerm.FromUnits [|
    (true, 0u); (true, 1u); (false, 1u); (false, 0u)
|]

twoBody.IsInNormalOrder // true
twoBody.IsInIndexOrder // true
```

5 Normal Ordering: Making Physics Legal

The central problem: we must put operator expressions in **normal order**—all creation operators before all annihilation operators.

Fermions obey the canonical anti-commutation relations (CAR):

$$\{a_i, a_j^\dagger\} = a_i a_j^\dagger + a_j^\dagger a_i = \delta_{ij} \tag{2}$$

```
let disordered =
    P<IxOp<uint32, LadderOperatorUnit>>.Apply [|
        LadderOperatorUnit.FromUnit(false, 0u) // a_0
        LadderOperatorUnit.FromUnit(true, 1u) // a+_1
    |]
    |> S<IxOp<uint32, LadderOperatorUnit>>.Apply

let ordered =
    LadderOperatorSumExpr<FermionicAlgebra>.ConstructNormalOrdered
        disordered
// Result: -1 * a+_1 a_0 (sign flipped!)
```

Same-index operators generate an identity term:

$$a_0 a_0^\dagger = \delta_{00} - a_0^\dagger a_0 = 1 - a_0^\dagger a_0 \tag{3}$$

5.1 The algebra is pluggable

Commutation relations live behind an interface:

```
type ICombiningAlgebra<'op> =
    abstract Combine :
        P<IxOp<uint32,'op>>
            -> C<IxOp<uint32,'op>>
            -> P<IxOp<uint32,'op>>[]
```

Algebra	Class	Physics	Key behaviour
Fermionic	`FermionicAlgebra`	Electrons	Swap ⇒ sign flip
Bosonic	`BosonicAlgebra`	Photons, phonons	Swap ⇒ no sign

5.2 Bosonic normal ordering

Compare $b_0 b_0^\dagger$ under bosonic algebra:

```
let bosonicResult = constructBosonicNormalOrdered bosonicExpr
// Result: 1 + b+_0 b_0 (PLUS sign --- bosons commute)
```

Fermionic: $a_0 a_0^\dagger = 1 - a_0^\dagger a_0$. Bosonic: $b_0 b_0^\dagger = 1 + b_0^\dagger b_0$. That single sign difference distinguishes matter from light.

6 Your First Encoding

The problem: quantum computers have qubits, but chemistry uses fermions. We need a mapping.

6.1 Jordan–Wigner

The simplest encoding Jordan and Wigner [1928] inserts a chain of Z operators on all preceding qubits:

```
let result = jordanWignerTerms Raise 2u 4u

for term in result.DistributeCoefficient.SummandTerms do
    printfn "%s␣%s" term.PhasePrefix term.Signature
// 0.5 ZZXI
// -0.5i ZZYI
```

The result is $a_2^\dagger = \frac{1}{2}(ZZXI) - \frac{i}{2}(ZZYI)$.

6.2 The Z-chain problem

The Pauli weight grows **linearly**—$O(n)$. For 100 qubits the last operator touches all 100 qubits.

7 Five Encodings, One Interface

Every encoding function has the same type signature:

```
type EncoderFn =
    LadderOperatorUnit -> uint32 -> uint32
        -> PauliRegisterSequence
```

This makes them drop-in replacements:

```
let jw = jordanWignerTerms Raise mode n // O(n)
let bk = bravyiKitaevTerms Raise mode n // O(log n)
let par = parityTerms Raise mode n // O(n)
let bt = balancedBinaryTreeTerms Raise mode n // O(log n)
let tt = ternaryTreeTerms Raise mode n // O(log_3 n)
```

7.1 PauliRegister and PauliRegisterSequence

Every encoding returns a `PauliRegisterSequence`—a sum of `PauliRegister` terms:

```
let reg = PauliRegister("ZZXI", Complex.One)
reg.Signature // "ZZXI"
reg.Coefficient // Complex(1.0, 0.0)
reg.[0] // Some Z
```

8 How Encodings Work Under the Hood

8.1 Majorana decomposition

A ladder operator is split into two Majorana operators:

$$a_j^\dagger = \tfrac{1}{2}(c_j - i\, d_j), \qquad a_j = \tfrac{1}{2}(c_j + i\, d_j) \tag{4}$$

The Majorana operators are built from three index sets:

$$c_j = X_{U(j)\cup\{j\}} \cdot Z_{P(j)} \tag{5}$$

$$d_j = Y_j \cdot X_{U(j)} \cdot Z_{(P(j)\oplus \mathrm{Occ}(j))\setminus\{j\}} \tag{6}$$

8.2 The EncodingScheme record

```
type EncodingScheme =
    { Update : int -> int -> Set<int> // U(j, n)
      Parity : int -> Set<int> // P(j)
      Occupation : int -> Set<int> } // Occ(j)
```

8.3 Custom encoding in 5 lines

```
let myJW : EncodingScheme =
    { Update = fun _ _ -> Set.empty
      Parity = fun j -> set [ for k in 0 .. j-1 -> k ]
      Occupation = fun j -> set [j] }

let myResult = encodeOperator myJW Raise 2u 4u
// Identical to jordanWignerTerms Raise 2u 4u!
```

Compare that to the ∼200 lines needed in other frameworks McClean et al. [2020].

8.4 Inspecting Majorana assignments

```
let cAssign = cMajorana jordanWignerScheme 2 4
// [(0, Z); (1, Z); (2, X)]

let dAssign = dMajorana jordanWignerScheme 2 4
// [(0, Z); (1, Z); (2, Y)]

let cReg = pauliOfAssignments 4 cAssign Complex.One
// PauliRegister("ZZXI", 1.0)
```

9 Trees and Fenwick Trees

9.1 Why trees?

The Z-chain in Jordan–Wigner grows linearly because it uses a linear data structure. A **tree** shares parity information, cutting depth to $O(\log n)$.

9.2 Fenwick Trees

The Bravyi–Kitaev encoding Bravyi and Kitaev [2002], Seeley et al. [2012] is built on a Fenwick tree. FockMap provides a purely functional implementation:

```
let tree = FenwickTree.ofArray (^^^) 0 occupations

FenwickTree.prefixQuery tree 3
FenwickTree.pointQuery tree 5
let tree' = FenwickTree.update tree 2 0
```

9.3 Encoding trees

```
let linear = linearTree 8 // Jordan-Wigner
let binary = balancedBinaryTree 8 // O(log_2 n)
let ternary = balancedTernaryTree 8 // O(log_3 n)
```

9.4 Two frameworks

Framework 1 (index sets, Fenwick-compatible):

```
let scheme = treeEncodingScheme (balancedBinaryTree 8)
encodeOperator scheme Raise 2u 8u
```

Framework 2 (path-based, any ternary tree) Jiang et al. [2020]:

```
let result = encodeWithTernaryTree tree Raise 2u 8u
```

10 Building a Real Hamiltonian

The electronic Hamiltonian in second quantization:

$$H = \sum_{pq} h_{pq}\, a_p^\dagger a_q + \frac{1}{2} \sum_{pqrs} \langle pq|rs\rangle\, a_p^\dagger a_q^\dagger a_s a_r \tag{7}$$

10.1 Step 1 — Define integrals

For H_2 in STO-3G (4 spin-orbitals):

```
let nModes = 4u
let oneBody = Map [
    ("00", Complex(-1.2563, 0.0))
    ("11", Complex(-1.2563, 0.0))
    ("22", Complex(-0.4719, 0.0))
    ("33", Complex(-0.4719, 0.0))
]
```

10.2 Step 2 — Coefficient lookup

```
let lookup (key : string) =
    match key.Length with
    | 2 -> oneBody |> Map.tryFind key
    | 4 -> twoBody |> Map.tryFind key
    | _ -> None
```

10.3 Step 3 — Compute

```
let hamiltonian = computeHamiltonian lookup nModes
```

10.4 Step 4 — Swap the encoding

```
let hBK = computeHamiltonianWith bravyiKitaevTerms lookup nModes
let hTT = computeHamiltonianWith ternaryTreeTerms lookup nModes
```

All three Hamiltonians have the same eigenvalues.

11 Mixed Bosonic–Fermionic Systems

Some models combine fermions and bosons (e.g. electron–phonon coupling).

11.1 Sector tagging

```
let f0_up = fermion Raise 0u // f+_0
let b1_down = boson Lower 1u // b_1
```

11.2 Canonical block order

Cross-sector commutators are zero:

$$[a_i, b_j] = [a_i, b_j^\dagger] = [a_i^\dagger, b_j] = [a_i^\dagger, b_j^\dagger] = 0 \tag{8}$$

The canonical form places all fermionic operators left and bosonic operators right.

11.3 Full mixed normal ordering

`constructMixedNormalOrdered` performs three steps:

1. Sector ordering (fermions left, bosons right; no sign change).
2. Fermionic normal ordering (CAR; sign flips).
3. Bosonic normal ordering (CCR; no sign flips).

```
let result = constructMixedNormalOrdered messyExpr
```

11.4 Decision guide

1. No bosons? Standard fermionic path.
2. Bosons but no qubit mapping yet? Hybrid pipeline.
3. Choosing an encoding? Compare on extracted fermion blocks.
4. Cutoff-sensitive? Convergence checks first.

11.5 Common failure modes

Symptom	Cause	Fix
Unexpected sign flips	Cross-sector/fermionic confusion	Canonicalise first
Non-deterministic shape	No block rule	`constructMixedNormalOrdered`
Bloated encoding	Identity placeholders	Drop identities first
Hard-to-debug modes	Implicit sectoring	Explicit constructors
Unstable bosonic results	Cutoff too low	Sweep and compare

12 The Utility Belt

12.1 Complex number extensions

```
let c = Complex(1.0, 2.0)
c.IsFinite; c.IsNonZero; c.TimesI; c.Reduce

Complex.SwapSignMultiple 3 Complex.One // -1
Complex.SwapSignMultiple 4 Complex.One // +1
```

12.2 Map extensions

```
let m = Map [ ("a", 1); ("b", 2) ]
m.Keys; m.Values
```

12.3 Currying utilities

```
let addTupled = uncurry add // (int * int) -> int
let addCurried = curry addTupled // int -> int -> int
```

13 Grand Finale: Three Encodings, One Molecule

This script ties every chapter together, encoding H_2 with three different encodings and comparing the results:

```
open Encodings
open System.Numerics

let nModes = 4u

let integrals = Map [
    ("00", Complex(-1.2563, 0.0))
    ("11", Complex(-1.2563, 0.0))
    ("22", Complex(-0.4719, 0.0))
    ("33", Complex(-0.4719, 0.0))
    (* two-body integrals omitted for brevity *)
]

let lookup key =
    match (key : string).Length with
    | 2 | 4 -> integrals |> Map.tryFind key
    | _ -> None

let encoders = [
    ("Jordan-Wigner", jordanWignerTerms)
    ("Bravyi-Kitaev", bravyiKitaevTerms)
    ("Ternary␣Tree", ternaryTreeTerms)
]

for (name, encoder) in encoders do
    let ham = computeHamiltonianWith encoder lookup nModes
    let terms = ham.DistributeCoefficient.SummandTerms
    let avgWeight =
        terms
        |> Array.averageBy (fun t ->
            t.Signature
            |> Seq.filter (fun c -> c <> 'I')
            |> Seq.length |> float)
```

```
    printfn "%s: %d terms, avg weight %.2f"
        name terms.Length avgWeight
```

All three Hamiltonians have the same eigenvalues—they represent identical physics. The differences in term count and Pauli weight affect measurement cost on real quantum hardware.

Quick Reference

Encoding functions

Function	Scaling	Best for
`jordanWignerTerms`	$O(n)$	Small systems
`bravyiKitaevTerms`	$O(\log_2 n)$	General purpose
`parityTerms`	$O(n)$	Parity-natural basis
`balancedBinaryTreeTerms`	$O(\log_2 n)$	Binary tree
`ternaryTreeTerms`	$O(\log_3 n)$	Best asymptotic
`encodeOperator`	Varies	Custom scheme
`encodeWithTernaryTree`	Varies	Custom tree

Type cheat sheet

Type	Represents
`Pauli`	I, X, Y, Z
`Phase`	Exact phase: $+1$, -1, $+i$, $-i$
`C<'T>`	Coefficient $\times$ operator
`P<'T>`	Ordered product
`S<'T>`	Sum of products
`IxOp<'idx,'op>`	Indexed operator
`LadderOperatorUnit`	$a^\dagger$ / a / I
`PauliRegister`	Pauli string + coefficient
`PauliRegisterSequence`	Sum of Pauli strings
`EncodingScheme`	Three index-set functions
`EncodingTree`	Tree shape
`FenwickTree<'a>`	Immutable binary indexed tree
`SectorLadderOperatorUnit`	Sector-tagged ladder op

Acknowledgements

This work is dedicated to Dr. Guang Hao Low, whose encouragement to study Bravyi–Kitaev encodings motivated the development of this framework.

References

John Azariah. FockMap: A composable framework for quantum operator encodings, 2026. URL https://github.com/johnazariah/encodings.

Pascual Jordan and Eugene Wigner. Über das Paulische äquivalenzverbot. *Zeitschrift für Physik*, 47: 631–651, 1928. doi: 10.1007/BF01331938.

Jarrod R. McClean, Nicholas C. Rubin, Kevin J. Sung, et al. OpenFermion: the electronic structure package for quantum computers. *Quantum Science and Technology*, 5(3):034014, 2020. doi: 10.1088/2058-9565/ab8ebc.

Sergey B. Bravyi and Alexei Yu. Kitaev. Fermionic quantum computation. *Annals of Physics*, 298: 210–226, 2002. doi: 10.1006/aphy.2002.6254.

Jacob T. Seeley, Martin J. Richard, and Peter J. Love. The Bravyi-Kitaev transformation for quantum computation of electronic structure. *The Journal of Chemical Physics*, 137:224109, 2012. doi: 10.1063/1.4768229.

Zhang Jiang, Amir Kalev, Wojciech Mruczkiewicz, and Hartmut Neven. Optimal fermion-to-qubit mapping via ternary trees with applications to reduced quantum states of chemistry. *PRX Quantum*, 1: 010306, 2020. doi: 10.1103/PRXQuantum.1.010306.