Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea in Fisica

Ottimizzazione numerica degli impulsi ottici in un interferometro atomico

Relatore:
Prof. Marco Prevedelli

Presentata da:
Gioele Rosini

Anno Accademico 2019/2020

Sommario

Per esperimenti di interferometria atomica d'avanguardia risulta sempre più indispensabile un'accurata descrizione analitica degli impulsi laser nel regime di transizione quasi-Bragg.
Lo scopo di questa tesi è di mettere a confronto due teorie analitiche per la diffrazione di onde di materia tramite luce laser, andandone a paragonare i risultati.
In particolare il recente modello di Siemß et al. [1], che propone un'interpretazione intuitiva basata sul teorema adiabatico, sarà messo a confronto con l'approccio preesistente basato sulle contribuzioni di Müller et al. [2]
Entrambi i modelli sono stati programmati in ambiente Python con libreria per l'analisi quantistica Qutip, per determinare la fedeltà ai risultati numerici delle aree in cui valga la condizione di Bragg secondo i due modelli analitici.

Indice

1 Introduzione **2**
1.1 Interferometria Atomica . . . 2
1.2 Diffrazione di Bragg . . . 2
1.3 Regime quasi-Bragg . . . 3
1.4 Teoria di Müller et al. . . . 4
1.5 Teoria adiabatica di Siemß . . . 4
1.6 Capitoli Successivi . . . 5

2 Fondamenti Teorici **6**
2.1 Atomo a due livelli . . . 6
2.2 Diffrazione di Bragg . . . 7
2.3 Elementi per l'analisi del regime di transizione . . . 11
2.4 Elementi del sistema analitico basato su teorema adiabatico . . . 13

3 Il Lavoro Svolto **16**
3.1 Approccio . . . 16
3.2 Procedimento . . . 17
3.3 Risultati . . . 18

A Ambiente hardware e software **22**

B Sorgenti Python **23**

Capitolo 1

Introduzione

1.1 Interferometria Atomica

L'interferometria atomica è un metodo potente per testare principi fondamentali della fisica come il principio di equivalenza [3] e l'elettrodinamica quantistica. Gli interferometri atomici si sono anche dimostrati sensori di alta qualità con applicazioni come gradiometri gravimetrici, gravimetri ad atomi freddi e giroscopi.
Un interferometro atomico è uno strumento che, sfruttando la struttura ondulatoria della materia, può generare il fenomeno di interferenza tra due o più fronti d'onda. Quindi un interferometro atomico è analogo ad un classico interferometro ottico ma con la vitale distinzione che i ruoli di luce e materia sono scambiati.
Infatti, in un interferometro atomico nuvole atomiche si propagano nei bracci dell'interferometro mentre impulsi laser ne controllano la propagazione, fungendo da specchi e beam splitter per le onde di materia. Un esperimento di interferometria atomica è solitamente composto da tre fasi: una prima in cui si prepara il campione atomico, una seconda in cui esso viene manipolato attraverso i fasci laser ed infine una fase di rilevamento del campione atomico.
Poiché questi esperimenti richiedono (ed ottengono) un'estrema precisione nel controllo dei pacchetti d'onda atomici utilizzando impulsi laser, sarà essenziale una fine comprensione della teoria che ne determina il comportamento.

1.2 Diffrazione di Bragg

Fasci di luce accuratamente calibrati possono assumere comportamenti di diffrazione Bragg del tutto analoghi a beam splitters e specchi. Fissandoli agli equivalenti degli impulsi $\pi/2$ e π, i fasci Raman contro-propaganti imprimono un momento agli atomi, per via della condizione di risonanza per transizioni Raman stimolate.
Si avrà una variazione di energia cinetica tanto maggiore quanto più è alto l'ordine della

diffrazione di Bragg, esso infatti indica il numero di fotoni con cui ciascun atomo interagisce concatenando transizioni Raman stimolate.

Prendendo come esempio un impulso π, la transizione di Bragg ideale è tale per cui l'intera popolazione atomica venga trasferita da uno stato iniziale di impulso unico e definito ad uno stato finale anch'esso unico e definito. Però in un processo reale di diffrazione di Bragg la luce interagisce con una nuvola atomica che ha una certa distribuzione di impulso e, di conseguenza, non sarà possibile ottenere perfetta efficienza di trasferimento negli impulsi specchio o beam-splitter. Per questa ragione in un processo reale dovremo aspettarci la graduale perdita di atomi che passino in altri stati di impulso.

1.3 Regime quasi-Bragg

Esistono soluzioni analitiche semplici dell'equazione di Schrödinger esclusivamente per i due casi limite del regime di diffrazione Bragg: impulsi laser brevi ed intensi sono descritti dal regime di Raman-Nath, mentre impulsi lunghi e deboli sono denominati come regime deep-Bragg.

Il regime Raman-Nath purtroppo, non permette di ottenere in modo efficiente grandi trasferimenti di impulso, come si vorrebbe per l'interferometria atomica ultrasensibile.

Nel caso del regime deep-Bragg, poiché se vale l'eliminazione adiabatica dei livelli intermedi solo $|+Nhk\rangle$ e $|-Nhk\rangle$ rimangono accoppiati, ciò è possibile per impulsi laser di lunga durata, anche se essa è condizione indesiderata come noteremo più avanti in questa analisi.
Dato che la sensibilità dei gravimetri è proporzionale all'impulso trasferito effettivo, per massimizzarli si vorrebbe quindi ottenere diffrazione di Bragg di ordine N elevato.

Questi grandi trasferimenti di impulso sono anche possibili in particolari casi del regime di transizione tra i due casi limite, detto quasi-Bragg. Specialmente quando combinati con pacchetti d'onda regolari (e.g. impulsi Gaussiani), gli impulsi quasi-Bragg permettono di avere deboli accoppiamenti agli stati fuori risonanza, comparabili al regime di diffrazione di Bragg [4]. Ciò allenta il requisito di lunghi tempi di interazione migliorando fortemente l'efficienza nei processi di diffrazione con ensemble atomici ultra-freddi.
Altri vincoli sul sistema sono:

- la frequenza di recoil ω_r è fissata dalla specie atomica e dal tipo di transizione utilizzata, quindi non è facilmente regolabile.
- la potenza del laser applicato, proporzionale a Ω^2, è limitata. Le transizioni Bragg a N grande diventano rapidamente irraggiungibili.
- la durata temporale τ degli impulsi laser, infatti essi hanno una larghezza in frequenza di ordine $1/\tau$ e riescono a essere risonanti con diversi valori di impulso nella

distribuzione di p della nuvola atomica. Con impulsi molto lunghi si utilizzano solo gli atomi con $p \sim 0$ e l'ampiezza del segnale si riduce di conseguenza.

Inoltre è importante ottenere un'elevata fedeltà degli impulsi, in caso contrario si avrebbe una rapida riduzione del contrasto delle frange di interferenza.

Nel regime quasi-Bragg non valgono le approssimazioni che hanno portato a soluzioni analitiche per i casi limite Raman-Nath e deep-Bragg. Esistono poche descrizioni analitiche della dinamica di Schrödinger generata da questo tipo di pacchetti d'onda, le teorie di riferimento sono le seguenti.

1.4 Teoria di Müller et al.

In [2] Müller et al. hanno presentato una teoria analitica del regime quasi-Bragg che permette la trattazione di ordini di diffrazione arbitrari e di funzioni inviluppo degli impulsi laser arbitrarie.
Per farlo hanno sviluppato un metodo sistematico per ottenere soluzioni sempre più precise dell'equazione di Schrödinger, a partire dall'approssimazione adiabatica standard. Più in dettaglio l'approccio è in due fasi.

Prima hanno risolto la dinamica efficace di una Hamiltoniana a due livelli usando gli autovalori dell'equazione di Mathieu [6], dopo aver eliminato per approssimazione adiabatica tutti gli accoppiamenti con stati fuori risonanza.

Sono poi giunti a descrivere gli stati del sistema nel regime quasi-Bragg per approssimazioni successive dell'equazione di Schrödinger, ottenendo equazioni differenziali disomogenee per gli stati dell'impulso confinanti con lo stato iniziale e quello finale.

Infine hanno derivato le perdite di popolazione per occupazione di stati diversi da quelli permessi dalla condizione di Bragg. Hanno trovato che questa popolazione può essere mantenuta estremamente bassa, assumendo che l'interazione sia applicata con una funzione di inviluppo continua nel tempo.

1.5 Teoria adiabatica di Siemß

In [1] Siemß et al. hanno mostrato che la fisica dei fasci laser Bragg può essere rappresentata con alta precisione tramite un modello basato sul teorema adiabatico.
Usando il teorema adiabatico in combinazione con metodi analitici della teoria di diffrazione hanno determinato una formulazione per la matrice di scattering per singoli impulsi nel regime quasi-Bragg.
La matrice di scattering ottenuta dipende da fasi dinamiche e da correzioni non adiabatiche di primo ordine corrispondenti a fasi e perdite di Landau-Zener [7]. Nei loro calcoli si sono concentrati su impulsi laser di inviluppo Gaussiano basandosi sui risultati

di Müller et al. Sfruttando la simmetria del sistema hanno introdotto una nuova base degli stati dell'impulso simmetrici ed antisimmetrici. Considerando anche l'impossibilità di passaggio da uno stato con ordine di diffrazione Bragg pari ad uno dispari, hanno ricavato quattro espressioni distinte per le Hamiltoniane degli stati.
Nell'esempio degli atomi di ^{87}Rb le transizioni di Bragg possibili sono limitate agli ordini $N \leq 5$ a causa degli effetti indesiderati di emissione spontanea [8]. Limitandosi per questa ragione ai soli ordini di diffrazione $N = 2, .., 5$, hanno prodotto grafici rappresentanti le zone di alta fedeltà dell'impulso rispetto allo stato desiderato al variare della frequenza di Rabi di picco e della durata temporale dell'impulso laser.
Grazie alle dipendenze dalla dinamica e dalle fasi di Landau-Zener, hanno ottenuto un'espressione che, data la fase differenziale desiderata ed il picco di frequenza di Rabi Ω_0 applicato, determini la durata temporale τ dell'impulso laser da applicare.

1.6 Capitoli Successivi

Nel Cap. 2 saranno introdotti gli elementi di teoria quantistica utilizzati, partendo dal modello dell'atomo a due livelli, in cui il processo ideale di diffrazione di Bragg si riduce. Successivamente sarà descritto il fenomeno della transizione Raman stimolata e si arriverà ad una descrizione del fenomeno della diffrazione Bragg come concatenazione di transizioni Raman stimolate. Infine saranno presentati elementi di teoria per i due sistemi analitici nel caso specifico dell'impulso Gaussiano.
Nel Cap. 3 verrà descritto in dettaglio il lavoro svolto incentrato sulla programmazione ed i grafici risultanti.

Capitolo 2

Fondamenti Teorici

2.1 Atomo a due livelli

Il filo del discorso presentato è stato basato sulle note di Steck [9].
Il modello quantistico da cui partire per studiare l'interazione tra ciascun atomo ed il campo elettromagnetico in cui è immerso è quello dell'atomo a due livelli. In esso gli unici due stati interni dell'atomo sono lo stato fondamentale e lo stato eccitato. Mentre gli stati atomici saranno trattati in modo quantistico, una trattazione classica sarà riservata al campo elettromagnetico.
Il termine di interazione con il campo è della forma

$$H_I = -\vec{d} \cdot \vec{E}\,, \tag{2.1}$$

dove $\vec{d}$ è il momento di dipolo elettrico dell'atomo mentre $\vec{E}$ è il campo elettrico dell'onda elettromagnetica. Notiamo anche come in esperimenti con atomi freddi vengano frequentemente utilizzati metalli alcalini, ben rappresentati dall'atomo di idrogeno. Quindi si può supporre che la Hamiltoniana imperturbata H_0 descriva il solo elettrone esterno, mentre si assume che il nucleo e gli elettroni nelle shell interne restino invariati nei loro stati quantistici.

Chiamando i due stati dell'atomo $|g\rangle$ ed $|e\rangle$, quando l'elettrone esterno si trova rispettivamente nello stato fondamentale oppure nel primo stato eccitato, il dipolo elettrico dell'atomo corrisponde a

$$\vec{d}_{eg} = \langle e| - e\vec{r}|g\rangle \tag{2.2}$$

e possiamo definire la frequenza di Rabi come

$$\Omega := \frac{|\vec{d}_{eg} \cdot \vec{E}_0|}{\hbar}\,. \tag{2.3}$$

Essa caratterizza l'intensità dell'accoppiamento tra atomo e campo ed in generale può aver dipendenza dal tempo.

2.2 Diffrazione di Bragg

Consideriamo quindi il processo di diffrazione di un atomo a due livelli per interazione con un'onda stazionaria.
Il potenziale è sinusoidale nello spazio quindi questo sistema è equivalente ad un pendolo quantistico, ma in un regime dove gli atomi abbiano sufficiente energia da non essere legati al reticolo ottico delle buche di potenziale. La dinamica del pendolo quantistico mostra un comportamento che è distintamente non-classico: l'impulso trasferito dal potenziale agli atomi è quantizzato.
Per un'onda stazionaria composta da due onde progressive uguali ma contro-propaganti si ha un campo della forma

$$\begin{aligned}\mathbf{E}(x,t) &= \hat{z}E_0[\cos(kx-\omega t)+\cos(kx+\omega t)] \\ &= \hat{z}E_0\cos(kx)(e^{-i\omega t}+e^{i\omega t}) \\ &= \mathbf{E}^{(+)}(x,t)+\mathbf{E}^{(-)}(x,t)\,,\end{aligned} \tag{2.4}$$

dove $\pm\omega$ sono le frequenze associate alle due onde contro-propaganti mentre k è il vettore d'onda dell'onda elettromagnetica.
Passando in un sistema di riferimento rotante alla frequenza del laser ed applicando la RWA (approssimazione di onda rotante) si ottiene una frequenza di Rabi pari a $\Omega(x) = \Omega_0\cos(kx)$.
Da ciò si può ricavare il potenziale di dipolo efficace (ignorando una costante)

$$V_{eff}(x) = V_0\cos(2kx)\,, \tag{2.5}$$

dove l'ampiezza del potenziale è

$$V_0 = \frac{\hbar|\Omega_0|^2}{8\Delta}\,. \tag{2.6}$$

in cui il detuning Δ è la differenza di frequenza tra i due fasci laser contro-propaganti.
L'equazione di Schrödinger del sistema è

$$\begin{aligned}i\hbar\partial_t|\psi\rangle &= \left[\frac{p^2}{2M}+V_0\cos(2kx)\right]|\psi\rangle \\ &= \left[\frac{p^2}{2M}+\frac{V_0}{2}(e^{i2kx}+e^{-i2kx})\right]|\psi\rangle\,,\end{aligned} \tag{2.7}$$

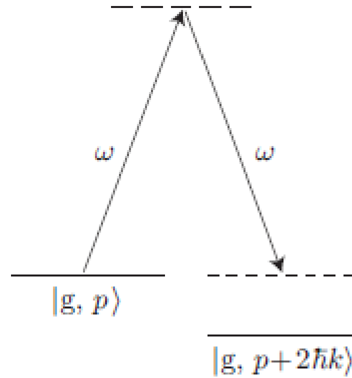

Figura 2.1: Transizione Raman Stimolata

che può essere scritta nella rappresentazione dell'impulso come

$$i\hbar\partial_t\psi(p) = \frac{p^2}{2M}\psi(p) + \frac{V_0}{2}\left[\psi(p+2\hbar k) + \psi(p-2\hbar k)\right], \tag{2.8}$$

dove $\psi(p) := \langle p|\psi\rangle$.

Questa forma è ricavata riconoscendo $\exp(ikx)$ come un operatore di traslazione per l'impulso o, in alternativa, da una trasformata di Fourier esplicita dell'equazione dalla rappresentazione spaziale a quella nello spazio degli impulsi. Quindi l'evoluzione nell'onda stazionaria impone una struttura a gradini sull'impulso, tale per cui un atomo allo stato iniziale nella funzione d'onda piana $|p\rangle$, successivamente potrà solamente occupare gli stati $|p + 2n\hbar k\rangle$ per n interi.

La quantizzazione dell'impulso ha un'interpretazione chiara nei termini di assorbimento ed emissione stimolata di fotoni dell'onda stazionaria: se l'atomo assorbe un fotone che stava viaggiando in una direzione e poi lo riemette nell'onda progressiva contro-propagante, l'atomo subirà un rinculo, cambiando il suo impulso di un valore pari al doppio dell'impulso del fotone, ovvero di $2\hbar k$.
Naturalmente, l'argomento appena considerato era basato su un trattamento classico del campo, quindi è la periodicità spaziale del potenziale ad imporre la struttura a gradini in questo modello. Ciononostante, il trasferimento di impulso agli atomi può essere visto come una transizione Raman stimolata, ovvero una transizione a due fotoni da uno stato fondamentale ad uno stato eccitato e poi di nuovo su un altro stato fondamentale.

Possiamo usare l'Eq. (2.8) per trovare le equazioni accoppiate tra i due stati, assumendo

che gli abbinamenti agli altri stati siano trascurabili.

$$
\begin{aligned}
i\hbar\partial_t\psi(p) &= \frac{p^2}{2M}\psi(p) + \frac{V_0}{2}\psi(p+2\hbar k) \\
i\hbar\partial_t\psi(p+2\hbar k) &= \frac{(p+2\hbar k)^2}{2M}\psi(p) + \frac{V_0}{2}\psi(p)
\end{aligned}
\tag{2.9}
$$

Queste sono le equazioni del moto che descrivono un sistema a due livelli, con frequenza di Rabi

$$
\Omega_R = \frac{V_0}{\hbar} = \frac{|\Omega_0|^2}{8\Delta}\,, \tag{2.10}
$$

ed un salto di energia tra gli stati interagenti pari a

$$
\Delta E = \frac{(p+2\hbar k)^2}{2M} - \frac{p^2}{2M} = \frac{2\hbar k p}{M} + 4\hbar\omega_r\,, \tag{2.11}
$$

dove l'energia di rinculo

$$
\hbar\omega_r := \frac{\hbar^2 k^2}{2M} \tag{2.12}
$$

(recoil energy)

è l'energia cinetica atomica associata con il rinculo dovuto all'emissione di un singolo fotone.

Quindi la popolazione atomica oscilla tra i due stati dell'impulso alla frequenza di Rabi effettiva Ω_R: poiché abbiamo eliminato lo stato intermedio (eccitato), il sistema a tre livelli si comporta approssimativamente come un sistema a due livelli efficace.
La transizione Raman stimolata a due fotoni è un esempio di un processo di diffrazione di Bragg. In effetti, è la più semplice forma di diffrazione di Bragg (detta di "primo ordine"); in generale la diffrazione di Bragg di n-esimo ordine è una transizione a $2n$ fotoni che ricopre un intervallo di $2n\hbar k$ impulsi compresi tra gli stati $|\pm n\hbar k\rangle$.
Il termine "diffrazione di Bragg" si applica al regime debolmente accoppiato dove gli stati intermedi non sono apprezzabilmente popolati, e quindi la transizione tra i due stati dell'impulso distanti può essere trattata come un problema a due livelli. In questo regime, il trasferimento classico tra queste regioni di impulso distinte è proibito, poiché il potenziale classico non è abbastanza grande da causare un tale cambiamento nell'impulso classico.

Perciò, la diffrazione di Bragg è un esempio di tunneling dinamico, ovvero un effetto di tunneling quantistico tra regioni nello spazio delle fasi tra cui le transizioni classiche non sono possibili, ma per ragioni di dinamica (in questo caso la natura asintotica del movimento di particella libera) invece che per barriere di potenziale.

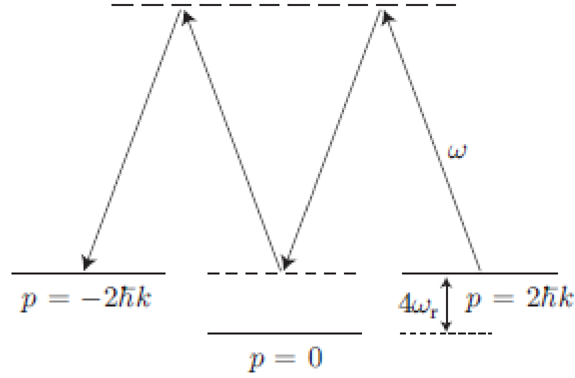

Figura 2.2: Diagramma dei livelli di energia nel caso in cui l'onda sia stazionaria. Si può notare che il detuning per lo stato di moto $|p = 0\rangle$ è pari allo sfasamento dell'energia cinetica.

Nonostante il potenziale abbia una piccola ampiezza, la coerenza quantistica si può accumulare mano a mano che gli atomi sono immersi nel potenziale, che porta gli atomi a cambiare significativamente il loro movimento.
Illustreremo questo processo considerando il caso relativamente semplice della diffrazione di Bragg al secondo ordine. Poi generalizzeremo i risultati al caso di n-esimo ordine.
Nel caso dove l'onda è stazionaria si avranno solamente gli stati $|-2\hbar k\rangle$ e $|2\hbar k\rangle$ accoppiati alla risonanza nel limite di Ω_R piccola. Nessun altro stato sarà sostanzialmente accoppiato a questi campi, a meno che la frequenza di Rabi effettiva non sia abbastanza grande da spingere atomi anche nelle transizioni fuori risonanza, fuoriuscendo dalle restrizioni del regime Bragg.
Ignorando gli accoppiamenti ad altri stati (ancor più fuori risonanza dello stato $|p = 0\rangle$), l'equazione di Schrödinger per i tre stati dell'impulso accoppiati diventa

$$\begin{aligned} i\hbar\partial_t\psi(-2\hbar k,t) &= \frac{(-2\hbar k)^2}{2M}\psi(-2\hbar k,t) + \frac{\hbar\Omega_R}{2}\psi(0,t) \\ i\hbar\partial_t\psi(0,t) &= \frac{\hbar\Omega_R}{2}\left[\psi(-2\hbar k,t) + \psi(2\hbar k,t)\right] \\ i\hbar\partial_t\psi(2\hbar k,t) &= \frac{(2\hbar k)^2}{2M}\psi(2\hbar k,t) + \frac{\hbar\Omega_R}{2}\psi(0,t)\,. \end{aligned} \tag{2.13}$$

Aggiungendo una differenza di energia di $-4\hbar\omega_r$ pari al detuning dallo stato di moto $|p = 0\rangle$, le equazioni diventano

$$\begin{aligned} i\hbar\partial_t\psi(\pm 2\hbar k,t) &= \frac{\hbar\Omega_R}{2}\psi(0,t) \\ i\hbar\partial_t\psi(0,t) &= \frac{\hbar\Omega_R}{2}\left[\psi(-2\hbar k,t) + \psi(2\hbar k,t)\right] - 4\hbar\omega_r\psi(0,t)\,. \end{aligned} \tag{2.14}$$

Se si assume che $\Omega_R \ll 4\omega_r$, la popolazione nello stato $|p=0\rangle$ è un $O(\Omega_R^2/\omega_r^2)$ e quindi trascurabile.
Inoltre è possibile fare un'approssimazione adiabatica per l'evoluzione dello stato $|p=0\rangle$, fissando formalmente $\partial_t \psi(0,t) = 0$. Questa è una scorciatoia analoga a considerare la rappresentazione a matrici di densità e sostituire le coerenze rapidamente oscillanti con la loro media temporale. Fare ciò porterebbe alla seguente relazione adiabatica

$$4\omega_r \psi(0,t) = \frac{\Omega_R}{2} \left[\psi(-2\hbar k, t) + \psi(2\hbar k, t)\right] \ , \tag{2.15}$$

che può essere usata per eliminare lo stato intermedio, ottenendo l'evoluzione a due livelli:

$$i\hbar\partial_t \psi(\pm 2\hbar k, t) = \frac{\hbar\Omega_R^2}{16\omega_r} \left[\psi(\pm 2\hbar k, t) + \psi(\mp 2\hbar k, t)\right] \ . \tag{2.16}$$

Quindi vediamo esplicitamente le oscillazioni di Raman-Rabi tra i due stati di moto (un fenomeno ben distinto dall'oscillazione dell'impulso del pendolo classico), e la frequenza di Rabi per Bragg di secondo ordine è $\Omega_{B,2} = \Omega_R^2/8\omega_r$. Il primo termine rappresenta uno Stark shift di $\Omega_{B,2}/2$, dovuto ai processi di diffrazione in cui i fotoni assorbiti e quelli emessi abbiano lo stesso k, mentre il secondo termine rappresenta l'accoppiamento simil-Rabi, dove i fotoni assorbiti ed emessi abbiano k opposti.
Per processi di diffrazione Bragg di ordini $n > 2$, gli stati intermedi sono comunque eliminabili in uno sviluppo analogo a quello visto finora. La frequenza di Rabi per diffrazione Bragg di n-esimo ordine da $n\hbar k$ a $-n\hbar k$ è data da

$$\Omega_{B,n} = \frac{\Omega_R^n}{2^{n-1}\prod_{k=1}^{n-1}\Delta_k} \ , \tag{2.17}$$

dove Δ_k è il detuning del k-esimo stato di moto intermedio. Notando che questi detuning intermedi sono dati da $[n^2-(n-2)^2]\omega_r, [n^2-(n-4)^2]\omega_r, ..., [n^2-(2-n)^2]\omega_r$, si può riscrivere la frequenza come

$$\Omega_{B,n} = \frac{\Omega_R^n}{(8\omega_r)^{n-1}[(n-1)!]^2} \tag{2.18}$$

(Bragg transition rate)

La frequenza di transizione diminuisce per processi Bragg di alto ordine, poiché le frequenze di Rabi calano esponenzialmente con n. Anche per questa ragione nella nostra analisi ci limiteremo ai processi sperimentalmente più comuni, di ordine $n = 2, ..., 5$.

2.3 Elementi per l'analisi del regime di transizione

Il sistema fisico studiato è composto da un atomo esposto a due fasci laser contro-propaganti e fortemente fuori fase rispetto alle transizioni atomiche. Inoltre avranno un

detuning relativo l'uno rispetto all'altro di $\Delta = k(v_0 + N\hbar k/M)$.

Si consideri la diffrazione di un atomo di massa M dovuto ad un'onda stazionaria di luce diretta lungo l'asse z. Ignorando effetti di emissione spontanea, l'Hamiltoniana che descriva questa interazione tra gli atomi ed un onda con numero d'onda k è (in un sistema di riferimento rotante alla frequenza ω del laser)

$$H = \frac{p^2}{2M} - \hbar\Delta|e\rangle\langle e| + \hbar\Omega(t)\cos(kz)(|e\rangle\langle g| + H.c.)\,, \tag{2.19}$$

dove p e z sono operatori. Sostituendo

$$|\psi(t)\rangle = e(z,t)|e\rangle + g(z,t)|g\rangle \tag{2.20}$$

nell'equazione di Schrödinger si ottiene la coppia di equazioni differenziali

$$\begin{aligned} i\hbar\dot{e}(z,t) &= \frac{p^2}{2M}e(z,t) + \hbar\Omega\cos(kz)g(z,t) - \hbar\Delta e(z,t)\,, \\ i\hbar\dot{g}(z,t) &= \frac{p^2}{2M}g(z,t) + \hbar\Omega\cos(kz)e(z,t)\,. \end{aligned} \tag{2.21}$$

Per Δ grandi rispetto alla banda dello stato eccitato (quindi anche $\Delta \gg \Omega, \omega_r$), e fissando gli atomi inizialmente allo stato fondamentale, si può eliminare adiabaticamente lo stato eccitato:

$$i\hbar\dot{g}(z,t) = -\frac{\hbar^2}{2M}\frac{\partial^2 g(z,t)}{\partial z^2} + \frac{\hbar\Omega^2}{\Delta}\cos^2(kz)g(z,t)\,. \tag{2.22}$$

Se poniamo

$$g(z,t) = \sum_{m=-\infty}^{\infty} g_m(t)e^{imkz} \tag{2.23}$$

e usiamo $\cos^2(kz) = 1/2 + 1/4(e^{2ikz} + e^{-2ikz})$, otteniamo che per ogni m

$$i\hbar\dot{g}_m = \hbar\left(\omega_r m^2 + \frac{\Omega^2}{2\Delta}\right)g_m + \frac{\hbar\Omega^2}{4\Delta}(g_{m+2} + g_{m-2})\,. \tag{2.24}$$

Questa equazione abbina stati di impulso pari con altri stati pari oppure stati dispari con dispari. Ciò è dovuto dal fatto che le transizioni Raman comportano sempre variazioni di impulso quantizzate multiple di $\Delta p = 2\hbar k$, ovvero due volte l'impulso del fotone. Cercheremo quindi soluzioni aventi tutti i termini dispari nulli oppure soluzioni con termini pari nulli.

Dalla Eq. (2.24) fissando le condizioni agli stati iniziali $g_{-n} = 1$ e $g_m = 0$ per $m \neq -n$ si ottengono le equazioni (sottraendo una costante $n^2\hbar\omega_r + \hbar\Omega^2/2\Delta$ dalla scala di energia)

$$i\hbar\dot{g}_{-n+2k} = \begin{cases} \frac{\hbar\Omega^2}{4\Delta}(g_{-n+2k+2} + g_{-n+2k-2}) & \text{per } k = 0, n \\ 4k(k-n)\hbar\omega_r g_{-n+2k} + \frac{\hbar\Omega^2}{4\Delta}(g_{-n+2k+2} + g_{-n+2k-2}) & \text{per } k \neq 0, n \end{cases} \tag{2.25}$$

Considerando la frequenza di Rabi ad inviluppo Gaussiano

$$\Omega(t) = \Omega_0 e^{-t^2/2\tau^2}\,, \tag{2.26}$$

dove Ω_0 è la frequenza di picco mentre τ è la durata dell'impulso, si può ricavare la frequenza di Rabi efficace sviluppata in serie di potenze

$$\frac{\Omega_{eff}}{\omega_r} = \frac{1}{8^{n-1}[(n-1)!]^2}\left(\frac{\Omega}{\omega_r}\right)^n \left| 1 - \sum_{j=1} \alpha_n^{(2j)} \left(\frac{|\Omega|}{\omega_r}\right)^{2j} - \sum_{j=1} \beta_n^{(j)} \left(\frac{\dot{\Omega}}{\omega_r \Omega}\right)^{2j} + \ldots \right|\,. \tag{2.27}$$

Se l'integrale della frequenza di Rabi efficace è uguale a π, detto "π-impulso", tutta la popolazione atomica si troverà nello stato finale; mentre se è $\pi/2$, detto "impulso $\pi/2$", lo sarà solamente metà di essa.
Per i $\pi/2$-impulsi si avrà quindi la condizione

$$\int_{-\infty}^{\infty} \Omega_{eff}(t')dt' = \frac{\pi}{2}\,, \tag{2.28}$$

Mentre per i π-impulsi si avrà

$$\int_{-\infty}^{\infty} \Omega_{eff}(t')dt' = \pi\,. \tag{2.29}$$

Queste due condizioni permettono di determinare i parametri della curvatura della Gaussiana.

2.4 Elementi del sistema analitico basato su teorema adiabatico

Il teorema adiabatico, formulato da Max Born e Vladimir Fock nel 1928 [10], enuncia che "Un sistema fisico rimane nel suo autostato istantaneo se una perturbazione data agisce su di esso abbastanza lentamente e se c'è un salto tra l'autovalore associato ed il

resto dello spettro della Hamiltoniana."
Nel caso di un atomo a due livelli immerso in un campo esterno vale l'approssimazione adiabatica quando l'interazione graduale con il campo permette il passaggio dallo stato fondamentale allo stato eccitato e viceversa.

Applicando impulsi luminosi che seguano il teorema adiabatico, la Hamiltoniana potrà essere espressa come segue.
La Hamiltoniana avrà termini di struttura identica per tutti i valori di quasi-impulso p, a parte un termine in Doppler shift che tratteremo come una perturbazione di primo ordine, per via della sua dipendenza dal quasi-impulso non banale. La Hamiltoniana al solo ordine zero che agisce sul sottospazio di Hilbert $\mathscr{H}_{p\alpha}$ è

$$H_\alpha(p) = \sum_{n\in\mathbb{Z}_\alpha} \left[\hbar\omega_r n^2 \hat{\sigma}_{n,n}(p) + \frac{\hbar\Omega}{2}\left(e^{2i\phi_L}\hat{\sigma}_{n+2,n}(p) + H.c.\right)\right], \tag{2.30}$$

dove $\hat{\sigma}_{n,m}(p) := |n\hbar k + p\rangle\langle m\hbar k + p|$ è il prodotto ket-bra degli autostati dell'impulso e ϕ_L è la fase del laser che viene assunta costante. Con l'introduzione di una nuova base di autostati simmetrici ed antisimmetrici $|p, n, \pm\rangle$, viene distinta in $H_\alpha = H_+ + H_-$. Distinguendo inoltre tra gli stati di ordine N pari con tutte le componenti dispari nulle e gli stati dispari con tutte le componenti pari nulle si giungerà alle quattro formule delle Hamiltoniane.
Le Hamiltoniane che agiscono sui sottospazi di Hilbert pari $\mathscr{H}_{pe\pm}$ sono, per $n \neq 0$,

$$\begin{aligned} H_{e-} &= \sum_{n\in\mathbb{N}_e} \left[\hbar\omega_r n^2 \hat{\sigma}^-_{n,n} + \frac{\hbar\Omega}{2}(\hat{\sigma}^-_{n+2,n} + H.c.)\right] \\ H_{e+} &= \sum_{n\in\mathbb{N}_e} \left[\hbar\omega_r n^2 \hat{\sigma}^+_{n,n} + \frac{\hbar\Omega}{2}(\hat{\sigma}^+_{n+2,n} + H.c.)\right] + \frac{\hbar\Omega}{\sqrt{2}}(\hat{\sigma}^+_{2,0} + H.c.) \end{aligned} \tag{2.31}$$

e quelle che agiscono sui sottospazi dispari $\mathscr{H}_{po\pm}$ sono

$$H_{o\pm} = \sum_{n\in\mathbb{N}_o} \left[\hbar\omega_r n^2 \hat{\sigma}^\pm_{n,n} + \frac{\hbar\Omega}{2}(\hat{\sigma}^\pm_{n+2,n} + H.c.)\right] \pm \frac{\hbar\Omega}{2}(\hat{\sigma}^\pm_{1,1}). \tag{2.32}$$

nelle quali $\hat{\sigma}^\pm_{n,m}(p) := |p, n, \pm\rangle\langle p, m, \pm|$.

Integrandole si può ottenere la fase dinamica del fascio $\theta^{dyn}_{N\pm} = \tau\omega_r x_{N\pm}(\Omega_0)$ dove $x_{N\pm}(\Omega_0)$ è un parametro adimensionale dipendente dalla forma esatta dell'impulso laser

$$x_{N\pm}(\Omega_0) = \int_{-\infty}^{+\infty} d\zeta \left(\frac{E_{N\pm}(\zeta\tau)}{\hbar\omega_r} - N^2\right) \tag{2.33}$$

in cui $E_{N\pm}(t)$ è l'autovalore dell'energia al tempo t, mentre $\zeta = t/\tau$ è un tempo adimensionale in scala rispetto alla durata dell'impulso laser.
La correzione alla fase per effetti di Landau-Zener è

$$\theta_{N\pm}^{LZ} = \frac{y_{N\pm}(\Omega_0)\Omega_0^2}{256(N-1)^3\omega_r^3\tau} \tag{2.34}$$

dove il parametro adimensionale $y_{N\pm}(\Omega_0)$ è

$$y_{N\pm}(\Omega_0) = \int_{-\infty}^{+\infty} d\zeta \left(\frac{\partial_\zeta \Omega(\zeta)}{\Omega_0}\right)^2 \times \frac{64q_{N\pm}^2(N-1)^3[N^2 - e_{N\pm}(w) + w\partial_w e_{N\pm}(w)]^2}{\left\{[N^2 - e_{N\pm}(w)]^2 + 4q_{N\pm}^2 w^2\right\}^{5/2}} \tag{2.35}$$

dove $q_{N\pm}$ è un parametro introdotto per quadrare rispetto ai dati numerici, mentre $w(\zeta) = \Omega(\zeta)/2\omega_r$.

La durata dell'impulso laser viene ricavata come

$$\tau(\Theta, \Omega_0) = \frac{\Theta}{2x_N(\Omega_0)\omega_r}\left(1 + \sqrt{1 - \frac{x_N(\Omega_0)y_N(\Omega_0)\Omega_0^2}{64(N-1)^3\Theta^2\omega_r^2}}\right) \tag{2.36}$$

dove $x_N(\Omega_0) = x_{N+}(\Omega_0) - x_{N-}(\Omega_0)$ e $y_N(\Omega_0) = y_{N+}(\Omega_0) - y_{N-}(\Omega_0)$. $\Theta = \pi, \pi/2$ è la fase differenziale desiderata.

La correzione al primo ordine del detuning doppler è

$$\langle +, N, p|Z_\alpha|-, N, p\rangle = 2N\tau^2\omega_r^2 z_{N,\Theta}(\Omega_0)e^{i\Theta/2}\,, \tag{2.37}$$

dove il parametro $z_{N,\Theta}(\Omega_0)$ è

$$z_{N,\Theta}(\Omega_0) \approx -i\int_{-\infty}^{+\infty} d\zeta\, \zeta \frac{E_{N+}(\zeta\tau) - E_{N-}(\zeta\tau)}{\hbar\omega_r} e^{i(\Theta(\zeta\tau) - \Theta/2)}\,. \tag{2.38}$$

Capitolo 3

Il Lavoro Svolto

3.1 Approccio

Per produrre i loro grafici il gruppo di Siemß et al. ha sviluppato [11] codice nel linguaggio Wolfram Mathematica che simuli il sistema quantistico. In questa sezione si illustra il procedimento tramite cui questo codice è stato tradotto in ambiente Python con l'obiettivo di porre a confronto i risultati di questo modello analitico con termini derivanti dal modello sviluppato da Müller et al.

Mathematica è un sistema di algebra computazionale per il calcolo simbolico e numerico. Ovvero si basa sulla definizione di simboli come funzioni supportate dal sistema o come equazioni contenenti parametri anche non definiti.
La principale difficoltà incontrata deriva dalla differenza di approccio logico adottata nel calcolo simbolico rispetto ad ambienti di programmazione orientati agli oggetti.
Infatti in ambiente Mathematica è possibile definire parametri con dipendenza simbolica da variabili prive di valore assegnato, mentre in ambiente Python è richiesto un valore numerico su cui applicare le operazioni desiderate.
Si è reso quindi necessario esprimere tali parametri come funzioni, in questo modo però si generano nuove difformità perché non è più possibile applicare le stesse operazioni matematiche disponibili per numeri, vettori o matrici.
Invece è risultato immediato il processo di integrazione della Hamiltoniana, in quanto la funzione di risoluzione della funzione di Schrödinger presente nella libreria Qutip ha permesso la trattazione diretta della dipendenza dal tempo all'interno della matrice stessa.
In tutti gli altri casi è stato necessario sviluppare codice apposito per trattare la variazione dei parametri in ingresso, tipicamente un doppio ciclo for per elaborare ogni singolo elemento della matrice.

Inoltre anche rimanendo nello stesso ambito di Python, in alcuni casi si sono dovute

definire interfacce per fare dialogare le diverse librerie Scipy, Qutip e Matplotlib, in particolare nella gestione dei dati.
In particolare non è stato possibile fare convergere l'integrazione numerica di funzioni continue con la libreria Scipy. Quindi si è reso necessario l'uso dell'integrazione per intervalli al fine di ottenere la convergenza dei parametri $x_{N\pm}$, $y_{N\pm}$, τ_{dyn}, τ_{LZ} e $z_{N,\Theta}$.

3.2 Procedimento

In primo luogo è stata codificata come funzione la frequenza di Rabi ad inviluppo Gaussiano $\Omega(t) = \Omega_0 \exp(-t^2/2\tau^2)$ dove sia Ω_0 sia τ sono state espresse in unità di ω_r.

Poi la Hamiltoniana espressa nelle equazioni (2.31) e (2.32) è stata definita come una matrice diagonale a blocchi in uno spazio di Hilbert non più infinito, ma troncato ad un ordine sufficientemente maggiore rispetto all'ordine di diffrazione. Le dimensioni scelte sono selezionate in modo tale da non avere cambiamenti significativi nei risultati all'aumento delle medesime.
Gli autostati finali dell'impulso e le energie $E_{N\pm}(t)$ ad essi connesse sono stati ricavati per integrazione del sistema su scala temporale che comprende l'intera durata dell'interazione con l'impulso laser. Anche questo intervallo temporale è stato preso in modo tale da non avere cambiamenti considerevoli dei risultati al suo incremento.
Calcolando gli autostati a $\Delta p = 0$ al variare di Ω_0 e τ si è tracciata la funzione di fedeltà $F_{\Theta,0}(\Omega_0, \tau)$ della distribuzione di popolazione ottenuta, rispetto a quella dovuta all'impulso ideale $\Theta = \pi, \pi/2$.

$$F_{\Theta,0}(\Omega_0, \tau) = lim_{\Delta p \to 0} \left| \langle \psi^{ideal}_{out,\Theta} | \psi_{out} \rangle \right|^2 \tag{3.1}$$

Dalle energie sono stati ricavati inoltre vari parametri:
$x_{N\pm}(\Omega_0)$ parametro adimensionale associato alla fase dinamica (2.33);
$z_{N,\Theta}(\Omega_0)$ parametro adimensionale associato al detuning doppler (2.37);
e due formulazioni per la durata temporale dell'impulso laser secondo (2.36):
τ_{dyn} calcolata a priori dalle correzioni di fase Landau-Zener ponendo il termine $y_N = 0$
τ_{LZ} comprensiva delle correzioni di fase Landau-Zener.

Infine è stato rianalizzato il sistema per tracciare nuovi grafici della funzione di fedeltà $F_{\Theta,0}(\Omega_0, \tau)$ fissando la stessa scala temporale e gli stessi intervalli di Ω_0 e τ. Le Fig. 3.2 ottenute rappresentano la fedeltà degli impulsi laser, ricavata evolvendo la Hamiltoniana di partenza (2.30) a $p = 0$, nel passaggio dallo stato $|N\hbar k\rangle$ allo stato $|-N\hbar k\rangle$.

3.3 Risultati

La funzione corrispondente alla frequenza di Rabi Gaussiana ha un comportamento appropriato.
Le Hamiltoniane dei sottospazi simmetrici ed antisimmetrici (2.31) e (2.32) sono state generate correttamente rispetto a [1, (30)].
Al momento gli autostati dell'impulso non combaciano con i valori attesi.
Non è noto se l'incongruenza dei dati derivi da un errore di traduzione in Python delle formule, oppure da limiti intrinseci di questo ambiente di sviluppo.
Di conseguenza non è stato possibile verificare il codice per i parametri $x_{N\pm}$, τ_{dyn}, τ_{LZ} e $z_{N,\Theta}$.

Si è tentato di riprodurre i grafici di fedeltà in [1, Fig. 1] per $\Delta p = 0$ e $\Theta = \pi$ ottenendo le Fig. reffig:siemssmirror. In esse si osservano pattern ondulatori diversi da quelli attesi. Inoltre si evidenziano comportamenti anomali come per ordine Bragg $N = 4$, in cui si ha una zona di uniforme alta fedeltà per valori di τ bassi.
Come confronto sono state prodotte le Fig. 3.2 che mostrano distribuzioni più simili a quelle desiderate, ma non ottengono completa corrispondenza con i dati previsti.
Analogamente sono state prodotte le Fig. 3.3 per il caso di beam-splitter $\Theta = \pi/2$. Anche la loro distribuzione non rispecchia i valori attesi.

Ciò significa che ulteriori analisi saranno necessarie per poter giungere alla sintesi desiderata.

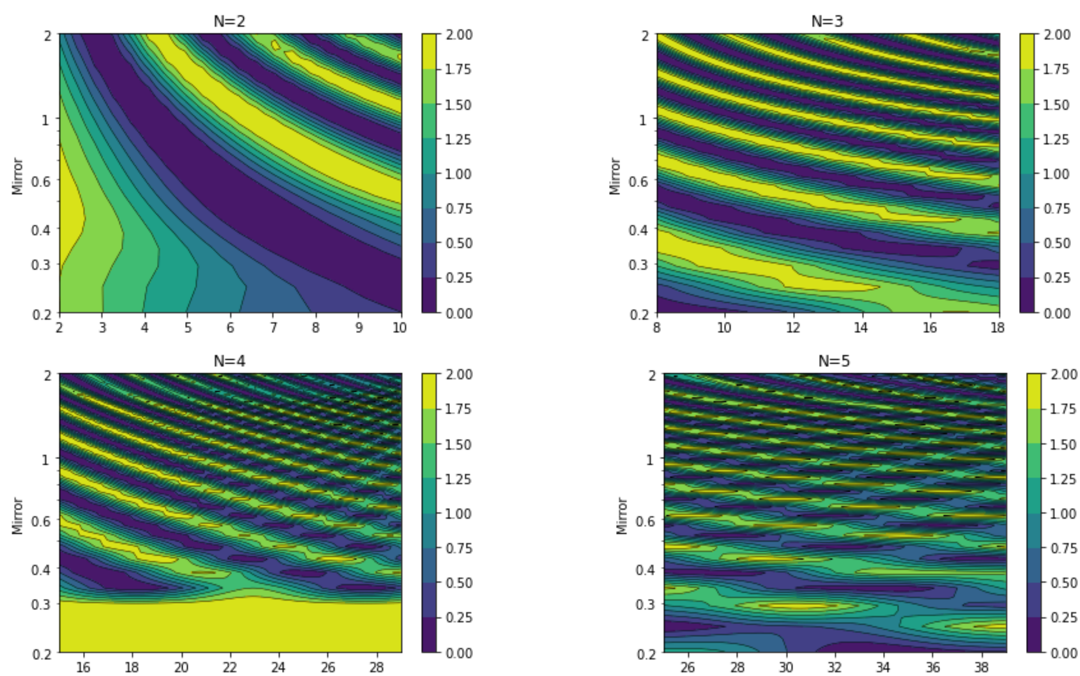

Figura 3.1: Fedeltà $F_{\pi,0}(\Omega_0, \tau)$ degli stati dell'impulso finali determinate numericamente integrando le Hamiltoniane (2.31) e (2.32) per un singolo π-impulso Gaussiano in funzione di Ω_0 su asse x e τ su asse y.

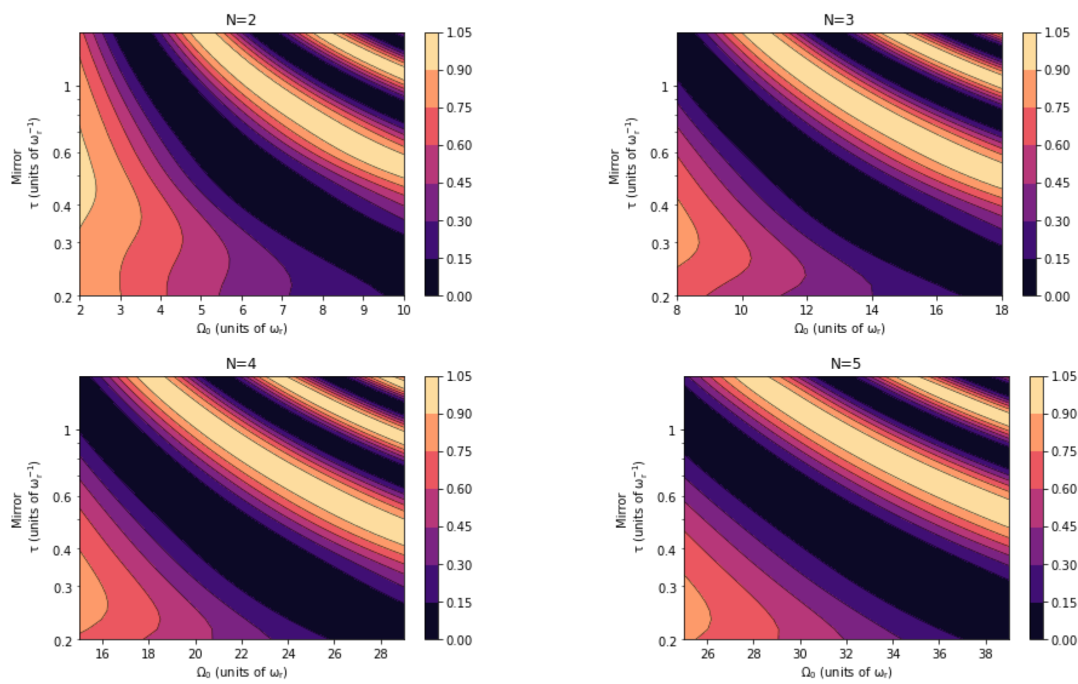

Figura 3.2: Fedeltà $F_{\pi,0}(\Omega_0, \tau)$ degli stati dell'impulso finali determinate numericamente integrando la Hamiltoniana (2.30) per un singolo π-impulso Gaussiano in funzione di Ω_0 su asse x e τ su asse y.

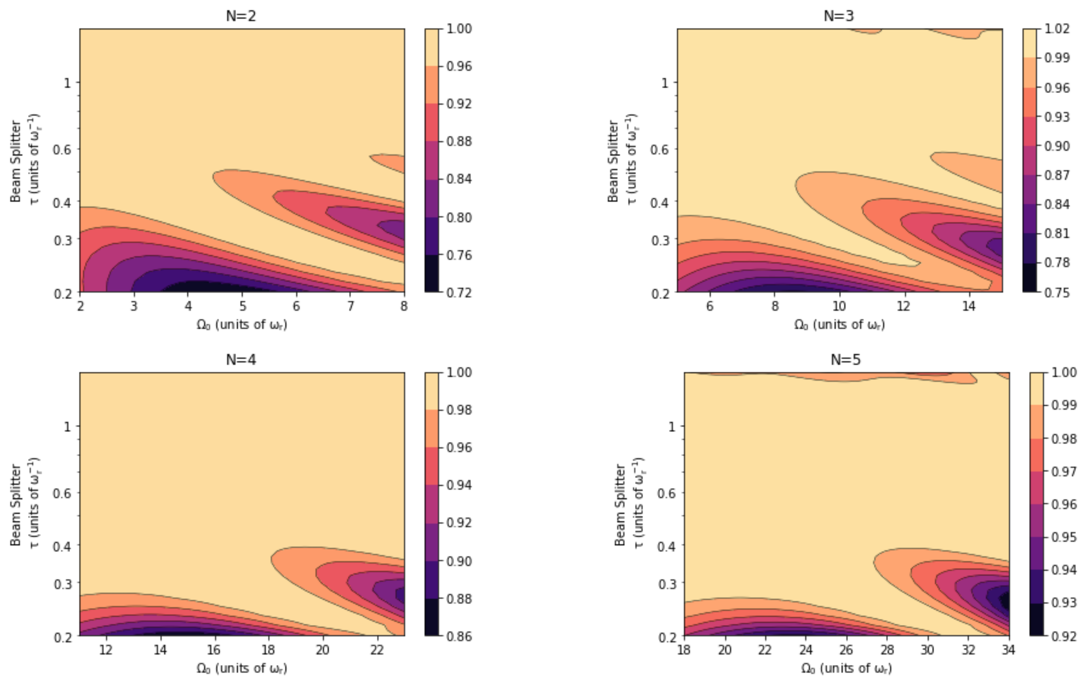

Figura 3.3: Fedeltà $F_{\pi/2,0}(\Omega_0, \tau)$ degli stati dell'impulso finali determinate numericamente integrando la Hamiltoniana (2.30) per un singolo impulso $\pi/2$ Gaussiano in funzione di Ω_0 su asse x e τ su asse y.

Appendice A

Ambiente hardware e software

Il sistema operativo su cui si è poggiato l'ambiente di sviluppo è Windows 10, l'hardware sottostante è composto da CPU Intel Core i5 7400, 4 core, 8 GB di RAM. L'ambiente di sviluppo è Python 3.8.5 con piattaforma Anaconda 3 come gestore della compatibilità dei pacchetti installati completato da Spyder 4.1.5 come IDE. Ulteriori librerie utilizzate sono Numpy 1.19.2 per la gestione degli array, SciPy 1.5.2 per la integrazione dei parametri, QuTiP 4.5.2 per la gestione degli oggetti quantistici e MatPlotLib 3.3.2 per la produzione di grafici.
Mathematica 12.2 è stato utilizzato per la lettura e l'interpretazione del codice prodotto da Siemß et al.

Appendice B

Sorgenti Python

A seguito sono trascritti i tre moduli che compongono il codice sviluppato:

- SiemssModel.py
- p0Fidelity.py
- MuellerFidelity.py

Il primo consiste nella modellazione basata sul teorema adiabatico tratta da [11].
Il secondo modulo ricava la fedeltà degli autostati secondo la Hamiltoniana eqn:hamiltoniansimple e genera i grafici in Figura 3.2.
Il terzo è un modello secondo la teoria analitica di Müller con autostati nel regime di Bragg per raffronto con i precedenti risultati.

SiemssModel.py

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
import math
import qutip.parallel as par
from qutip import * #qobj, fileio
import scipy.integrate as integrate
import matplotlib as mpl

# *** C1 Mathematica: Analytic Pulse Parameters ***

# Initial laser phase  L = 0
# Planck constant    hbar = 1

# * Pulse Parameters *

# General pulse duration. Identical for all pulses
# t1-t0 -> duration of the pulse to be well within the scattering limit

# TODO: fix time scale
#alpha = 20         # scaling factor for initial/final time as per Siemss calculation
alpha = 3          # scaling factor for initial/final time
t0 = - alpha * 1.1 # initial time
t1 =   alpha * 1.1 # final time

# Parameters for solving integrals and differential equations
tsteps = 4 #10000 # number of subdivisions of the timeline
tlist = np.linspace(t0,t1, tsteps) # timeline
opts = Options()
opts.nsteps = 50000#500000 # max number of iterations for the solver


Nmax = 0 # number of Hilbert space dimensions
Nbragg  = 0 # Bragg scattering order

# *  Rabi Frequency as a Gaussian pulse, function of time *
def GaussianRabiFreq(t,args):
    # expressed in units of recoil frequency r
    omega0 = args["PeakRabi"]
```

```
    tau = args ["PulseLength"]

    omega = omega0 * np.exp(- 0.5 * (t / tau) **2 )
    return omega

# * d/d Partial derivative of the Rabi Frequency as a Gaussian pulse *
def Derivate_GaussianRabiFreq(zeta, args):
    # zeta = := t / tau
    omega0 = args["PeakRabi"]

    omega = -zeta * omega0 * np.exp(- 0.5 * (t / tau) **2 )
    return omega

# * w = /2r  *
def w(zeta, args):
    # expressed in units of recoil frequency r
    # zeta = := t / tau
    omega0 = args["PeakRabi"]

    omega = omega0 * np.exp(- 0.5 * zeta**2 ) / 2
    return omega


# * Symmetric/Antisymmetric (S/A) Hamiltonians *
#   status: complete
def DetermineHamiltonian(sym, n, RabiFreq = GaussianRabiFreq):
    length = math.trunc(n / 2 + 1) # number of elements in the main diagonal of the H
    a=   np.zeros((length,)) #a = np.array([]) # main diagonal, elements not dependant
    a1 = np.zeros((length,)) # main diagonal, elements DEPENDANT on RabiFreq
    b =  np.zeros((length - 1,)) # nearby diagonals, elements DEPENDANT on RabiFreq

    if sym == 's' :
        # Hamiltonian in symmetric subspace
        if(n % 2 == 0) :
            # even n

            for k in range(0, length) :
                a[length-1 - k] = (k*2)**2

            if (n>2):
                b[0 : length - 2] = 1/2
```

```
        b[length-2] = 1 / math.sqrt(2)

    else:
        # odd n

        for k in range(0, length) :
            if k==0 :
                a[length-1]  = 1
                a1[length-1] = 1/2
            else:
                a[length-1 - k]  = (2*k + 1)**2

        b[0 : length-1] = 1/2


elif sym == 'a':
    # Hamiltonian in asymmetric subspace

    if(n % 2 == 0) :
        # even n

        for k in range(0, length) :
            a[length-1 - k] = (k*2)**2

        if (n>2):
            b[0 : length - 2] = 1/2


    else:
        # odd n

        for k in range(0, length) :
            if k==0 :
                a[length-1]  = 1
                a1[length-1] = -1/2
            else:
                a[length-1 - k]  = (2*k + 1)**2

        b[0 : length-1] = 1/2
```

```
    else:
        print("ERROR: The symmetry of the Hamiltonian MUST be either 's' or 'a'")
        return

    H0 = np.diag(np.append(a, [0] * (n - length)))
    H1 = np.diag(np.append(a1, [0] * (n - length))) + np.diag(np.append(b, [0] * (n -

    H = [Qobj(H0), [Qobj(H1), RabiFreq]]

    return H

# * List of Eigenstates in timeline*
#   TODO: tweak to make sure atoms don't have energy in higher level eigenstates
#   status: incomplete
def DetermineEigenstates(sym, H, args, RabiFreq = GaussianRabiFreq, isDebug = False):

    highestLevel = math.trunc((Nmax - Nbragg)/2)

    # Initial basis
    rho0 = basis(Nmax, Nbragg)

    # As defined by Mathematica codebase:
    """
    if sym == 's' :
        # in symmetric subspace
        rho0 = basis(Nmax, highestLevel) / math.sqrt(2)
    elif sym == 'a':
        # in asymmetric subspace
        rho0 = - basis(Nmax, highestLevel) / math.sqrt(2)
    else:
        print("The symmetry of of the eigenstates MUST be either 's' or 'a'")
        return
    """

    if isDebug : print('initial basis:',rho0.data)

    if isDebug : progress=True
    else : progress = None

    output = mesolve(H, rho0, tlist, c_ops= None, e_ops= None, args= args, options= o
```

```
    if isDebug :
        #print('length list eigenstates:',len(output.states),'| size eigenstates at t

        #print('\nket t0\n', output.states[0])
        #print('ket t=0\n', output.states[math.trunc(len(output.states)/2)].data)
        #print('ket t1\n', output.states[len(output.states)-1],'\n')

        plot_fock_distribution(output.states[math.trunc(len(output.states)/2)], title
        plot_fock_distribution(output.states[-1],title= r"$t = t1, n = %i$" % Nbragg)

        final_state_a = np.zeros([len(tlist),], dtype=np.complex)
        for t in range(len(tlist)):
            final_state_a[t]=output.states[t][highestLevel]
        #print("possible final state",highestLevel,"at t0:",final_state_a[0])

        final_state_b = np.zeros([len(tlist),], dtype=np.complex)
        for t in range(len(tlist)):
            final_state_b[t]=output.states[t][Nbragg]
        #print("possible final state",Nbragg,"at t0:",final_state_b[0])
        final_state_a = final_state_a * np.conj(final_state_a)
        final_state_b = final_state_b * np.conj(final_state_b)

        fig, axes = plt.subplots(1, 1, figsize=(10,6))
        axes.plot(tlist, np.real(final_state_a), label="$Highest Level %i$" % highestL
        axes.plot(tlist, np.real(final_state_b), label="$Bragg Order %i$" % Nbragg)
        axes.legend(loc=0)
        axes.set_xlabel('Time')
        axes.set_ylabel('Occupation probability')
        if sym == 's':
            axes.set_title('Final state oscillations |{},+⟩'.format(Nbragg))
        elif sym == 'a':
            axes.set_title("Final state oscillations |{},-⟩".format(Nbragg))

    return output.states

# * Eigenvalues *
#   status: eigenenergy formula is unclear
def DetermineEigenvalue(t_in, H, args, EStates, RabiFreq = GaussianRabiFreq, isDebug =
    # Definition of of eigenvalues/eigenenergies (cf. Eqs.(41))

    # index of highest occupied eigenstate (corresponding to parameter "len" in Mathem
```

```
highestLevel = math.trunc((Nmax - Nbragg)/2)

if np.ndim(t_in) == 0:

    # t_in is a scalar value

    # check t_in is within parameters
    if((t_in < t0) or (t_in > t1)) :
        print('ERROR: scalar t_in',t_in,'in DetermineEigenvalue outside the range
        # set to default
        t_in = 0

    # detemine index of eigenstate that matches the time t_in
    t_idx = round((t_in-t0)/(t1-t0) * len(EStates))
    if t_idx >= len(EStates) :
        t_idx = len(EStates)-1
        print('-Warning-  in DetermineEigenvalue t_idx was clamped to ', t_idx)
    elif t_idx < 0 :
        t_idx = 0
        print('-Warning-  in DetermineEigenvalue t_idx was clamped to ', t_idx)

    #check that t_in aligns with the eigenstate[t_idx] within a 0.001 margin
    if (abs(t_in - (((t_idx)/len(EStates)*(t1-t0))+t0)) > 0.001) :
        print('-Warning-  in DetermineEigenvalue t_in is', round(t_in, 4),' while

    # eigenvector in t=t_in
    rho_t = EStates[t_idx]

    # Hamiltonian in t=t_in
    H_t = H[0] + RabiFreq(t_in, args) * H[1][0]

    """
    # Hamiltonian projection on eigenvector
    a = (H_t * rho_t)

    # Eigenergy in t=t_in with projection on basis
    E_t = np.zeros((Nmax,), dtype='complex')
    for d in range(Nmax):
        denom = rho_t.data.todense()[d][0]
        if (np.real(denom) == 0):
            denom = 1 + np.imag(denom)*1j
```

```
            if (np.imag(denom) == 0):
                denom = np.real(denom) + 1j
            E_t[d] = a.data.todense()[d][0] / denom
        E_t = np.sort_complex(E_t)
        """

        if isDebug :
            E_t = H_t * rho_t
            print('\nH|n⟩ = E|n⟩ =\n',E_t.data)

            E_t = np.linalg.eig(H_t)
            print('E =\n',E_t)

        # Eigenenergies in t=t_in without solver
        Eauto = H_t.eigenenergies()

        if isDebug :
            #print('\nSolved SE eigenenergies |',len(E_t)-highestLevel-1,'"highest le
            print('\nOUTPUT: Qutip function eigenenergies(), ',len(Eauto)-highestLevel

        return Eauto[-highestLevel-1]
    else:

        # t_in is an array of times

        # assuming the array t_in is a truncated tlist

        if isDebug : print("Eigenenergies' time list of",len(t_in),"elements")

        # list of eigenvectors in t=t_in
        #rho_list = np.array(EStates[:len(t_in)])

        # list of Hamiltonians in t=t_in
        H_list = [ H[0] + RabiFreq(_t, args) * H[1][0] for _t in t_in ]

        # Eigenenergies in t=t_in without sesolver
        E_list = np.array([ H_t.eigenenergies()[-highestLevel-1] for H_t in H_list ])

        if isDebug :
            #print('\nSolved SE eigenenergies |',len(E_t)-highestLevel-1,'"highest le
            print('\nOUTPUT: Qutip function eigenenergies() E(t0)={}'.format(E_list[0
```

```
        return E_list

# * Definition of dynamic phase parameter xN+/xN- *
#   status: incomplete, check the output with the tables
def xIntegral(sym, H, args):
    # Definition according to Eq. (46)
    # Planck constant = 1, wr as unit

    # calculating eigenstates
    EStates = DetermineEigenstates(sym, H, args)

    # eigenvalue to integrate
    def xInt(zeta, args):
        E = DetermineEigenvalue(zeta, H, args, EStates= EStates)
        return E - Nbragg**2

    # continuous integration
    #x = integrate.quad(xInt, t0, t1, args= args, epsabs=1e-02, epsrel=1e-02, limit= 

    # fixed steps integration
    xInt_list = xInt(tlist,args)
    x = integrate.simps(xInt_list, tlist, even= 'first')

    return x

# * Definition of LZ phase parameter yN+/yN- *
#   status: complete
#   TODO: align the output with the y tables
def yIntegral(sym, omega0, fct):
    # Definition according to Eq. (B8)
    # omega0 = 0 : peak Rabi frequency for Gaussian pulse.
    # fct : Pre-diagonalization parameter to account for ACStark shifts. See qN in Ap

    # args : omega0 as an additional argument for many functions
    args = {"PeakRabi" : omega0}

    # Parameters dependant on the Bragg order N:
    q=0
    # lowEval : Pre-diagonalization eigenvalue. See Appendix B Eq.(B5), Table II.
    lowEval = lambda zeta,args: print("lowEval in yIntegral undefined")
```

```
# derivLE : partial derivative of lowEval in w
derivLE = lambda zeta,args: print("derivLE in yIntegral undefined")


if sym == 's' :
    # y in symmetric subspace

    # Setting q, lowEval and derivLE
    if(Nbragg == 2):

        q = math.sqrt(2)
        def lowEval_n2(zeta,args):
            return 0

        def derivLE_n2(zeta,args):
            return 0

        lowEval = lowEval_n2
        derivLE = derivLE_n2

    elif(Nbragg == 3):

        q = 1
        def lowEval_n3(zeta, args):
            return 1 + w(zeta, args)

        def derivLE_n3(zeta,args):
            return 1

        lowEval = lowEval_n3
        derivLE = derivLE_n3

    elif(Nbragg == 4):

        q = fct
        def lowEval_n4(zeta, args):
            return 2 + math.sqrt(4 + 2 * w(zeta, args)**2)

        def derivLE_n4(zeta, args):
            return 2 * w(zeta, args) / math.sqrt(4 + 2 * w(zeta, args)**2)
```

```
        lowEval = lowEval_n4
        derivLE = derivLE_n4

    elif(Nbragg == 5):

        q = fct
        def lowEval_n5(zeta, args):
            return 5 + (w(zeta, args)/2) + math.sqrt(16 - 4*w(zeta, args) + (5 * 

        def derivLE_n5(zeta, args):
            return 1/2 + (-4 + 5*w(zeta, args)/2) / (2 * math.sqrt(16 - 4*w(zeta,

        lowEval = lowEval_n5
        derivLE = derivLE_n5

    else:
        print("Error: yIntegral has unexpected Nbragg value", Nbragg)
        return 0

elif sym == 'a' :
    # y in antisymmetric subspace

    # Setting q, lowEval and derivLE
    if(Nbragg == 2):

        q = 0
        def lowEval_n2(zeta,args):
            return 0

        def derivLE_n2(zeta,args):
            return 0

        lowEval = lowEval_n2
        derivLE = derivLE_n2

    elif(Nbragg == 3):

        q = 1
        def lowEval_n3(zeta, args):
            return 1 - w(zeta, args)
```

```
        def derivLE_n3(zeta,args):
            return -1

        lowEval = lowEval_n3
        derivLE = derivLE_n3

    elif(Nbragg == 4):

        q = 1
        def lowEval_n4(zeta, args):
            return 4

        def derivLE_n4(zeta, args):
            return 0

        lowEval = lowEval_n4
        derivLE = derivLE_n4

    elif(Nbragg == 5):

        q = 1
        def lowEval_n5(zeta, args):
            return 5 - (w(zeta, args)/2) + math.sqrt(16 + 4*w(zeta, args) + (5 * 

        def derivLE_n5(zeta, args):
            return -1/2 + (4 + 5*w(zeta, args)/2) / (2 * math.sqrt(16 + 4*w(zeta,

        lowEval = lowEval_n5
        derivLE = derivLE_n5

    else:
        print("Error: yIntegral has unexpected Nbragg value", Nbragg)
        return 0
else:
    print("The symmetry of the yIntegral MUST be either 's' or 'a'")
    return 0

# Numerical calculation of Y integral function:
def Integrand(z, a):
    return (64 * q**2 *(Nbragg - 1)**3) * (Nbragg**2 - lowEval(z,a) + w(z,a) * de
def yInt(zeta, args):
```

```
        omega0 = args["PeakRabi"]
        return (Derivate_GaussianRabiFreq(zeta, args) / omega0)**2 * Integrand(zeta, 

    #print("\nOmega0: ",omega0.round(4)," Integrand: ",Integrand(0,args).round(4)," y

    # Integration in zeta = t / tau
    # continuous integration
    #YS = integrate.quad(yInt, t0, t1, args= args, epsabs=1e-02, epsrel=1e-02, limit=

    # fixed steps integration
    yInt_list = yInt(tlist, args)
    YS = integrate.simps(yInt_list, tlist, even= 'first')

    return YS

# Analytically inferred pulse duraction for adiabatic dynamics (cf. Eq. (51) for yN(0
#   status: integral not converging
def TauDyn(diff_phase, m, Hs, Ha, args):
    # diff_phase =  : target differential phase.
    # m =           m: natural number multiplied by 2.
    # Hs =         H+: symmetric Hamiltonian of bragg order Nbragg
    # Ha =         H-: asymmetric Hamiltonian of bragg order Nbragg
    # args =     args: extra arguments for the Hamiltonian

    # calculating eigenstates
    EStates_s = DetermineEigenstates('s', Hs, args)
    EStates_a = DetermineEigenstates('a', Ha, args)

    def EnerDiffer(t):
        #print('Nmax:',Nmax,'\nn',Nbragg,'\nHs',Hs,'\nrho0', basis(Nmax, Nbragg))
        Es = DetermineEigenvalue(t, Hs, args, EStates= EStates_s)
        Ea = DetermineEigenvalue(t, Ha, args, EStates= EStates_a)
        return Es - Ea

    numerator = diff_phase + m * 2 * math.pi

    # continuous integration
    #denominator = integrate.quad(EnerDiffer, t0, t1, epsabs=1e-02, epsrel=1e-02, lim

    # fixed steps integration
    eed = EnerDiffer(tlist)
```

```
    denominator = integrate.simps(eed, tlist, even= 'first')

    return numerator/denominator

# * Landau-Zener phases according to Eqs. (49) & (B8) *
#   status: complete but needs testing
def LZPhase(yInt, args, fct = 1):
    om0 = args["PeakRabi"]
    tau = args["PulseLength"]

    return yInt * om0**2 /(256 * (Nbragg - 1)**3 * tau)

# * Analytically inferred pulse duration Eq. (51) *
#   status: complete but needs testing
def TauLZ(diff_phase, m , Hs, Ha, args, fct = 1):
    # diff_phase =  : target differential phase.
    # m =            m: natural number multiplied by 2.
    # Hs =          H+: symmetric Hamiltonian of bragg order Nbragg
    # Ha =          H-: asymmetric Hamiltonian of bragg order Nbragg
    # args =     args: extra arguments for the Hamiltonian
    # fct: Pre-diagonalization parameter to account for AC-Stark shifts. See q_N in A

    # calculating eigenstates
    EStates_s = DetermineEigenstates('s', Hs, args)
    EStates_a = DetermineEigenstates('a', Hs, args)

    # calculating Y parameters
    y_s = yIntegral('s', args["PeakRabi"], fct)
    y_a = yIntegral('a', args["PeakRabi"], fct= 1)

    # To solve Eq.(50) for (0) we set =1 here
    args_tau1 = {"PeakRabi" : args["PeakRabi"], "PulseLength" : 1}

    # calculating Landau-Zener phases
    phiLZ_s = LZPhase(y_s, args_tau1, fct)
    phiLZ_a = LZPhase(y_a, args_tau1)
    # Note: LZ phase in asymmetric subspace is neglected (fct=1) as it is suppressed

    def EnerDiffer(t):
        #print('Nmax:',Nmax,'\nn',Nbragg,'\nHs',Hs,'\nrho0', basis(Nmax, Nbragg))
        Es = DetermineEigenvalue(t, Hs, args, EStates= EStates_s)
```

```
        Ea = DetermineEigenvalue(t, Ha, args, EStates= EStates_a)
        return Es - Ea

    # continuous integration
    #a = integrate.quad(EnerDiffer, t0, t1, epsabs=1e-02, epsrel=1e-02, limit= opts.n

    # fixed steps integration
    eed = EnerDiffer(tlist)
    a = integrate.simps(eed, tlist, even= 'first')

    p = (diff_phase + m * 2 * math.pi) / a

    q = (phiLZ_s - phiLZ_a) / a
    # Note: solution (p/2) - math.sqrt( ... ) not physically relevant
    return (p/2) + math.sqrt((p/2)**2 - q)

# * Perturbative treatment of Doppler detuning following Eq.(C5) *
#   status: needs testing
def zIntegral(Hs, Ha, args):
    # diff_phase =  : target differential phase.
    # m =            m: natural number multiplied by 2.
    # Hs =          H+: symmetric Hamiltonian of bragg order Nbragg
    # Ha =          H-: asymmetric Hamiltonian of bragg order Nbragg
    # args =      args: extra arguments for the Hamiltonian

    # calculating eigenstates
    EStates_s = DetermineEigenstates('s', Hs, args)
    EStates_a = DetermineEigenstates('a', Hs, args)

    def EnerDiffer(t):
        #print('Nmax:',Nmax,'\nn',Nbragg,'\nHs',Hs,'\nrho0', basis(Nmax, Nbragg))
        Es = DetermineEigenvalue(t, Hs, args, EStates= EStates_s)
        Ea = DetermineEigenvalue(t, Ha, args, EStates= EStates_a)
        return Es - Ea

    # continuous integration
    """
    def PhyRel(tf, args):
        # tf: final time for integration interval

        tau = args["PulseLength"]
```

```
        return tau * integrate.quad(EnerDiffer, t0, tf, epsabs=1e-02, epsrel=1e-02, l

    # final PhyRel
    phyRel_f = PhyRel(t1, args)

    def Integral(zeta2):
        return math.e ** (1j * (PhyRel(zeta2, args) - phyRel_f/2)) * zeta2 * EnerDiff

    zInt = 1j * integrate.quad(Integral, t0, t1, epsabs=1e-02, epsrel=1e-02, limit= o
    """

    # fixed steps integration

    def PhyRel(idx_tf, args):
        # idx_tf: intex of final time for integration interval

        tau = args["PulseLength"]
        return tau * integrate.simps( EnerDiffer(tlist[:idx_tf]), tlist[:idx_tf], eve

    # final PhyRel
    phyRel_f = PhyRel(len(tlist)-1, args)

    # building array for integrand
    integrand = np.zeros([len(tlist),], dtype=np.complex)
    for idx_zeta in range(len(tlist)):
        zeta = tlist[idx_zeta]
        integrand[idx_zeta] = math.e ** (1j * (PhyRel(idx_zeta, args) - phyRel_f/2))

    zInt = 1j * integrate.simps(integrand, tlist, even= 'first')


    return zInt

# * previous version of fidelity used by Mathematica codebase *
#
def Fidelity_old(diff_phase, symmstate, asymstate, isNorm = False, isDebug = False):
    # diff_phase : diffraction phase (theta) determining if the pulse is a Beam-Split
    # symmstate,asymstate : final eigenstates s_N(0,)(t1), a_N(0,)(t1)
    # isNorm : if flagged the function will output a normalized fidelity (used in Fig

    if (diff_phase == math.pi/2) :
```

```
        # * Beam-Splitter *

        if isNorm == False :
            # BS fidelity
            fid = (abs(symmstate)**2 + abs(asymstate)**2)/2 - np.imag(np.conj(symmsta

        else :
            # Normalized fidelity :
            #   losses from the subspace (s_N(0,)(t1),a_N(0,)(t1)) are masked out
            fid = 1/2 - (np.imag(np.conj(symmstate) * asymstate))/(abs(symmstate)**2 

    elif(diff_phase == math.pi) :
        # * Mirror *

        if isNorm == False :
            # M fidelity
            fid = (abs(symmstate + asymstate)**2)/2

        else :
            # Normalized fidelity :
            #   losses from the subspace (s_N(0,)(t1),a_N(0,)(t1)) are masked out
            fid = (abs(symmstate + asymstate)**2)/ (2 * abs(symmstate)**2 + abs(asyms

    else:
        print('ERROR: the diffraction phase in Fidelity() MUST be either pi or half-p

    if isDebug : print('fidelity:', fid)

    return fid

# * fidelity F(0,) represented in Fig. 1 *
#
def Fidelity(diff_phase, s_i, s_f, isNorm = False, isDebug = False):
    # diff_phase : diffraction phase (theta) determining if the pulse is a Beam-Split
    # symmstate,asymstate : final eigenstates s_N(0,)(t1), a_N(0,)(t1)
    # isNorm : if flagged the function will output a normalized fidelity (used in Fig

    if (diff_phase == math.pi/2) :
        # * Beam-Splitter *

        fid = abs(s_i)**2 + abs(s_f)**2
```

```
    elif(diff_phase == math.pi) :
        # * Mirror *

        fid = abs(s_f)**2

    else:
        print('ERROR: the diffraction phase in Fidelity() MUST be either pi or half-p

    if isDebug : print('fidelity:', fid)

    return fid

def PulseParameters(N,phase):

    global Nmax
    global Nbragg


    RabiFreqMin = 0 # temporary values
    RabiFreqMax = 0 # "
    # TODO: risistemare la RabiFreqVar
    #RabiFreqVar = 0.01 # increment in Rabi frequency
    RabiFreqVar = 0.4

    TauMin = 0.2
    TauMax = 2
    TauSteps = 40

    if  (N == 2) :
        # Bragg order N=2, Truncation nmax = 6
        Nbragg = 2
        Nmax = 6

        if phase == 'BS' :

            # * Beam Splitter *

            # Rabi interval
            RabiFreqMin = 2
            RabiFreqMax = 8
```

```
        #RabiFreqVar = 0.01

    else:

        # * Mirror *

        # Rabi interval
        RabiFreqMin = 2
        RabiFreqMax = 10
        #RabiFreqVar = 0.1

elif(N == 3) :
    # Bragg order N=3, Truncation nmax = 7
    Nbragg = 3
    Nmax = 7

    if phase == 'BS' :

        # * Beam Splitter *

        # Rabi interval
        RabiFreqMin = 5
        RabiFreqMax = 15
        #RabiFreqVar = 0.05

    else:

        # * Mirror *

        # Rabi interval
        RabiFreqMin = 8
        RabiFreqMax = 18
        RabiFreqVar = 0.15

elif(N == 4) :
    # Bragg order N=4, Truncation nmax = 8
    Nbragg = 4
    Nmax = 8

    if phase == 'BS' :
```

```
        # * Beam Splitter *

        # Rabi interval
        RabiFreqMin = 11
        RabiFreqMax = 23#24
        #RabiFreqVar = 0.05
    else:

        # * Mirror *

        # Rabi interval
        RabiFreqMin = 15
        RabiFreqMax = 29
        RabiFreqVar = 0.15

elif(N == 5) :
    # Bragg order N=52, Truncation nmax = 11
    Nbragg = 5
    Nmax = 11

    if phase == 'BS' :

        # * Beam Splitter *

        # Rabi interval
        RabiFreqMin = 20
        RabiFreqMax = 34
        RabiFreqVar = 0.1# 0.05

    else:

        # * Mirror *

        # Rabi interval
        RabiFreqMin = 25
        RabiFreqMax = 39
        RabiFreqVar = 0.15

else : return

# X var
```

```
    RabiFreqList = np.linspace(RabiFreqMin, RabiFreqMax, int((RabiFreqMax-RabiFreqMin

    # Y var
    TauList = np.linspace(TauMin, TauMax, TauSteps)

    return RabiFreqList, TauList

# * generate data for a fidelity plot *
#
def MapFidelity(N, phase):

    RabiFreqList,TauList = PulseParameters(N, phase)

    print('x points:',np.shape(RabiFreqList))
    print('y points:',np.shape(TauList))


    # save-file name
    file_name = "Fidelity Data/"
    file_name += "N={}_nmax={}_".format(Nbragg, Nmax)
    file_name += phase
    file_name += ".dat"

    # S/A Hamiltonians
    H_s = DetermineHamiltonian("s", Nmax, GaussianRabiFreq)
    H_a = DetermineHamiltonian("a", Nmax, GaussianRabiFreq)

    # target diffraction phase
    if phase == 'BS' : diff_phase = math.pi/2
    elif phase == 'Mi' : diff_phase = math.pi

    # Z var
    FidList = np.zeros((len(TauList),len(RabiFreqList)))
    print('z progress:')

    for i in range(len(RabiFreqList)):
        for k in range(len(TauList)) :
            pos = {"PeakRabi" : RabiFreqList[i], "PulseLength" : TauList[k]}

            symmstate = DetermineEigenstates('s', H_s, args= pos, isDebug= False)[-1]
            asymstate = DetermineEigenstates('a', H_a, args= pos, isDebug= False)[-1]
```

```
            FidList[k,i] = Fidelity_old(diff_phase, symmstate, asymstate, isDebug= Fa

        print(i+1,'/',len(RabiFreqList),'...')

    print('z points:',np.shape(FidList))

    # store z in .dat
    print('saved in',file_name)
    file_data_store(file_name, FidList, sep="\t", numtype='real')
                                # FidList.T
    # return x,y,z
    return RabiFreqList, TauList, FidList#.T

def SaveFidelities(diff_phase) :
    for N in range(2,5+1):
        print("\n\n * mapping N={}, phase={} *\n".format(N,diff_phase))
        MapFidelity(N, diff_phase)
    return


def ContourPlot(x, y, z, label=""):

    fig, axs = mpl.pyplot.subplots()
    cs = axs.contourf(x, y, z, cmap=mpl.cm.purples)

    mpl.rcParams['lines.linewidth'] = 0.5
    plt.rcParams['axes.formatter.min_exponent'] = 2
    axs.contour(cs, colors='k')
    axs.set_title('N={}'.format(Nbragg))
    plt.yscale('log')
    plt.ylabel(label)
    plt.figure(figsize=(8, 6),dpi=300)
    cbar = fig.colorbar(cs)

    #fig.set_size_inches(18.5, 10.5, forward=True)

    return

def ImportData(N, phase, extra=""):
    RabiFreqList, TauList = PulseParameters(N, phase)
```

```
    file_name = 'Fidelity Data/N={}_nmax={}_{}{}.dat'.format(Nbragg, Nmax, phase, ext
    print('loading',file_name)
    z = file_data_read(file_name, sep="\t")
    print('import data loaded!')
    return RabiFreqList, TauList, z

# * Parameters to create tables *
#   status: missing file export and parameters for N>2
def AnalyticPulseParameters(N):
    # Set parameters
    prfx = ""

    global Nmax
    global Nbragg

    if N == 2 :
        # Bragg order N=2, Truncation nmax = 6
        Nbragg = 2
        Nmax = 6
        prfx = "N=2_nmax=6_"

        # * Beam Splitter *
        DiffPhase_BS = math.pi/2
        RabiFreqMin = 2
        RabiFreqMax = 8
        # TODO: risistemare la RabiFreqVar
        #RabiFreqVar = 0.01 # increment in Rabi frequency
        RabiFreqVar = 0.5
        RabiFreqList = np.linspace(RabiFreqMin, RabiFreqMax, int(RabiFreqMax / RabiFr
        m0 = 0  # First Bragg resonance
        fct = 1 # See qN in Appendix B and Table II.


        H_s = DetermineHamiltonian("s", Nmax, GaussianRabiFreq)
        H_a = DetermineHamiltonian("a", Nmax, GaussianRabiFreq)


        omega0=2.0 #for omega0 in RabiFreqList:

        tau_dyn = TauDyn(DiffPhase_BS, m0, H_a, H_s, args= {"PeakRabi" : omega0, "Pul
```

```
        x_s = xIntegral('s', H_s, args= {"PeakRabi" : omega0, "PulseLength" : 1})
        x_a = xIntegral('a', H_a, args= {"PeakRabi" : omega0, "PulseLength" : 1})

        y_s = yIntegral('s', omega0, fct)
        y_a = yIntegral('a', omega0, fct)


        for omega in RabiFreqList:

            # Pulse parameters according to Eq.(51) for first Bragg resonance
            tLZ = TauLZ(DiffPhase_BS, m0, H_s, H_a, args= {"PeakRabi" : omega, "Pulse

            """
            # Pulse parameters according to Eq.(51) for multiple Bragg resonances
            for m in range(4):
                TauLZ(DiffPhase_BS, m, H_s, H_a, args= {"PeakRabi" : omega, "PulseLen
            """


        print('\n\n')

        print('N:',Nbragg,'Omega0:', np.round(omega0, 4), '\ttau_dyn:', np.round(tau_

        print('N:',Nbragg,'Omega0:', np.round(omega0, 4), '\tx_s:', np.round(x_s, 4),

        print('N:',Nbragg,'Omega0:', np.round(omega0, 4), '\ty_s:', np.round(y_s, 4),

        print('N:',Nbragg,'Omega:', np.round(omega, 4), '\ttau_LZ:', np.round(tLZ, 4)

        print('\n\n')

    #file_data_store('expect.dat', output_data.T, numtype="real")

# Method for easier reading of the Hamiltonian
def PrintH(name, H_in, w=1):


    H0 = H_in[0].data.todense()
    H1 = H_in[1][0].data.todense() * 2 * w
    H = np.real(np.around(H0 + H1,2))
```

```
    print(' -',name, '| N max:',Nmax,'-')
    print(H,'\n')


# * Build & Save datasets: *

#SaveFidelities("Mi")

aX,aY,aZ = MapFidelity(5, "Mi")

#aX,aY,aZ = MapFidelity(2, "BS")

# * Plot a dataset: *

#aX,aY,aZ = ImportData(2,"Mi")
ContourPlot(aX, aY, aZ, "Mirror")

#aX,aY,aZ = ImportData(2,"BS")
#ContourPlot(aX, aY, aZ, "Beam Splitter")


# * Try Mathematica data as input *
"""
in_arr = np.genfromtxt('Siemss - Data/Fig1/N=2_nmax=6_numAmplitudes.dat', delimiter='
                       dtype=[('omega0', '<f8'), ('tau', '<f8'), ('s_N', np.complex12

print(in_arr,'\n\n',in_arr.dtype )
x = in_arr['omega0']
y = in_arr['tau']
s = in_arr['s_N']
a = in_arr['a_N']
print('a',np.shape(a))
z= np.reshape(x,(1000,236))
print("final: ",x[-1,1],", next: ",x[0,2])
print("final: ",x[1,-1],", next: ",x[2,0])

"""

# * Test Hamiltonians and Eigenstates *
"""
# Setting some temporary values
```

```
omega0 = 5
tau = 1

for k in range(2,5+1):
    # setting Nbragg and Nmax
    Nbragg = k

    if  (Nbragg == 2) : Nmax = 6
    elif(Nbragg == 3) : Nmax = 7
    elif(Nbragg == 4) : Nmax = 8
    elif(Nbragg == 5) : Nmax = 11
    else              : Nmax = Nbragg + 4

    args = {"PeakRabi" : omega0, "PulseLength" : tau}

    # calculating Hamiltonians
    Hs = DetermineHamiltonian("s", Nmax, GaussianRabiFreq)
    #PrintH("Hs", Hs)
    Ha = DetermineHamiltonian("a", Nmax, GaussianRabiFreq)
    #PrintH("Ha", Ha)

    # calculating eigenstates and eigenvalues
    print('\n+++ |N,+⟩   N Bragg:',Nbragg,',   N max:',Nmax, '+++')
    EState_s = DetermineEigenstates('s', Hs, args , isDebug=False)
    #E_s = DetermineEigenvalue(0, Hs, args, EState_s, isDebug=True)

    print("final state s",EState_s[-1][Nbragg])

    print('\n--- |N,-⟩   N Bragg:',Nbragg,',   N max:',Nmax, '---')
    EState_a = DetermineEigenstates('a', Ha, args , isDebug=False)
    #E_a = DetermineEigenvalue(0, Ha, args, EState_a, isDebug=True)

    print("final state a",EState_a[-1][Nbragg])
"""

# * Test x,y,tau as integrals (for N=2 only) *

#AnalyticPulseParameters(2)
```

p0Fidelity.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed May  5 17:44:38 2021

@author: Gioele
"""

import numpy as np
import matplotlib.pyplot as plt
import math
from qutip import *
import scipy.integrate as integrate
from matplotlib import ticker, cm
import matplotlib.pyplot as plt
import matplotlib as mpl

alpha = 3          # scaling factor for initial/final time
t0 = - alpha * 1.1 # initial time
t1 =   alpha * 1.1 # final time
tsteps = 4 # number of subdivisions of the timeline
tlist = np.linspace(t0,t1, tsteps) # timeline

opts = Options()
opts.nsteps = 50000 # max number of iterations for the solver

Nmax = 2
Nbragg = 2
pos0 = 0.0
k=1

# *  Rabi Frequency as a Gaussian pulse, function of time *
#   Eq. 60
def GaussianRabiFreq(t,args):
    # expressed in units of recoil frequency r
    omega0 = args["PeakRabi"]
    tau = args ["PulseLength"]

    omega = omega0 * np.exp(- 0.5 * (t / tau) **2 )
    return omega
```

```
# * Hamiltonian of atom/wave system as a sum on the truncated Hilbert-space *
#
def DetermineHamiltonian(N):
    # Using Eq. (23a) in Siemss
    # w_r = 1
    # hbar = 1
    # phy_L = 0

    H0list = []
    H1list = []

    for n in range(-N+1,N,2):
        # position of n-order state
        idx_n = math.trunc(pos0 + n/2)

        # h0 = sigma_n,n * n^2
        H0list.append( projection(N,idx_n,idx_n) * n**2 )
        #print("H0",H0list[n])

        # h1 = (sigma_n+2,n)/2
        if (idx_n < N-1):
            H1list.append( (projection(N,idx_n+1,idx_n) + projection(N,idx_n+1,idx_n)
            #print("H1",H1list[n])

    H0 = sum(H0list)
    H1 = sum(H1list)
    H = [Qobj(H0), [Qobj(H1), GaussianRabiFreq]]
    return H

# * Eigenstates |+Nhk> and |-Nhk> *
#
def Estates(H,rho0, args, isDebug = False):

    fin_states = mesolve(H, rho0, tlist, c_ops= None, e_ops= None, args= args, option

    if isDebug : plot_fock_distribution(rho0, title= r"$t = t0, n = %i$" % Nbragg)
    if isDebug : plot_fock_distribution(fin_states,title= r"$t = t1, n = %i$" % Nbrag

    # return |+Nhk> and |-Nhk>
    return fin_states[math.trunc(pos0 - Nbragg/2)], fin_states[math.trunc(pos0 + Nbra
```

```
# * fidelity F(0,) represented in Fig. 1 *
#
def Fidelity(diff_phase, s_i, s_f, isDebug = False):
    # diff_phase : diffraction phase (theta) determining if the pulse is a Beam-Split
    # s_i, s_f : final eigenstates |+Nk> and |-Nk>

    if (diff_phase == math.pi/2) :
        # * Beam-Splitter *

        fid = abs(s_i)**2 + abs(s_f)**2

    elif(diff_phase == math.pi) :
        # * Mirror *

        fid = abs(s_f)**2

    else:
        print('ERROR: the diffraction phase in Fidelity() MUST be either pi or half-p

    if isDebug : print('fidelity:', fid)

    return fid


# * generate data for a fidelity plot *
#
def MapFidelity(N, phase, extra=""):

    RabiFreqList,TauList = PulseParameters(N, phase)

    print('x points:',np.shape(RabiFreqList))
    print('y points:',np.shape(TauList))


    # save-file name
    file_name = "Fidelity Data/"
    file_name += "Simple_N={}_nmax={}_".format(Nbragg, Nmax)
    file_name += phase + extra + ".dat"
```

```
    # Hamiltonian
    H = DetermineHamiltonian(Nmax)

    # Initial basis
    rho0 = basis(Nmax, math.trunc(pos0 + Nbragg/2))# Nbragg)

    # target diffraction phase
    if phase == 'BS' : diff_phase = math.pi/2
    elif phase == 'Mi' : diff_phase = math.pi

    # Z var
    FidList = np.zeros((len(TauList),len(RabiFreqList)))
    print('z progress:')

    for i in range(len(RabiFreqList)):
        for k in range(len(TauList)) :

            PositionInPlot = {"PeakRabi" : RabiFreqList[i], "PulseLength" : TauList[k

            s_pos,s_neg = Estates(H, rho0, PositionInPlot)
            FidList[k,i] = Fidelity(diff_phase, s_pos, s_neg, isDebug= False)

        print(i+1,'/',len(RabiFreqList),'...')

    print('z points:',np.shape(FidList))

    # store z in .dat
    print('saved in',file_name)
    file_data_store(file_name, FidList, sep="\t", numtype='real')

    # return x,y,z
    return RabiFreqList, TauList, FidList#.T

# * x,y parameters for fidelity plots *
#
def PulseParameters(N,phase):

    global Nmax
    global Nbragg
    global pos0
```

```
RabiFreqMin = 0 # temporary values
RabiFreqMax = 0 # "
#RabiFreqVar = 0.01 # increment in Rabi frequency
RabiFreqVar = 0.2

TauMin = 0.2
TauMax = 1.5
TauSteps = 80

if  (N == 2) :
    # Bragg order N=2, Truncation nmax = 6
    Nbragg = 2
    Nmax = 7

    if phase == 'BS' :

        # * Beam Splitter *

        # Rabi interval
        RabiFreqMin = 2
        RabiFreqMax = 8
        #RabiFreqVar = 0.01

    else:

        # * Mirror *

        # Rabi interval
        RabiFreqMin = 2
        RabiFreqMax = 10
        #RabiFreqVar = 0.1

elif(N == 3) :
    # Bragg order N=3, Truncation nmax = 7
    Nbragg = 3
    Nmax = 8

    if phase == 'BS' :

        # * Beam Splitter *
```

```
        # Rabi interval
        RabiFreqMin = 5
        RabiFreqMax = 15
        #RabiFreqVar = 0.05

    else:

        # * Mirror *

        # Rabi interval
        RabiFreqMin = 8
        RabiFreqMax = 18
        RabiFreqVar = 0.15

elif(N == 4) :
    # Bragg order N=4, Truncation nmax = 8
    Nbragg = 4
    Nmax = 9

    if phase == 'BS' :

        # * Beam Splitter *

        # Rabi interval
        RabiFreqMin = 11
        RabiFreqMax = 23#24
        #RabiFreqVar = 0.05
    else:

        # * Mirror *

        # Rabi interval
        RabiFreqMin = 15
        RabiFreqMax = 29
        RabiFreqVar = 0.15

elif(N == 5) :
    # Bragg order N=52, Truncation nmax = 11
    Nbragg = 5
    Nmax = 12
```

```
        if phase == 'BS' :

            # * Beam Splitter *

            # Rabi interval
            RabiFreqMin = 18# 20
            RabiFreqMax = 34# 34
            RabiFreqVar = 0.1# 0.05

        else:

            # * Mirror *

            # Rabi interval
            RabiFreqMin = 25
            RabiFreqMax = 39
            RabiFreqVar = 0.15

    else : return

    pos0=(Nmax-1)/2

    # X var
    RabiFreqList = np.linspace(RabiFreqMin, RabiFreqMax, int((RabiFreqMax-RabiFreqMin

    # Y var
    TauList = np.linspace(TauMin, TauMax, TauSteps)

    return RabiFreqList, TauList

# * plot fidelity as function of peak Rabi freq. and pulse duration *
#
def ContourPlot(x, y, z, label="", fig=None, axs=None):

    if(fig == None):
        fig, axs = mpl.pyplot.subplots()

    cs = axs.contourf(x, y, z, cmap= cm.magma)

    mpl.rcParams['lines.linewidth'] = 0.5
```

```
    plt.rcParams['axes.formatter.min_exponent'] = 2
    params = {'mathtext.default': 'regular' }
    plt.rcParams.update(params)
    axs.contour(cs, colors='k')
    axs.set_title('N={}'.format(Nbragg))
    plt.yscale('log')
    plt.ylabel(label + '\n  (units of $\omega_r^{-1}$)')
    plt.xlabel('$\Omega_0$ (units of $\omega_r$)')
    plt.figure(figsize=(8, 6),dpi=300)
    cbar = fig.colorbar(cs)

    #fig.set_size_inches(18.5, 10.5, forward=True)

    return

def MultiPlot(phase):
    fig, axs = mpl.pyplot.subplots()

    if (phase == "BS"):
        for N in range(2,5+1):
            aX,aY,aZ = ImportData(N, phase)
            ContourPlot(aX,aY,aZ, 'Beam Splitter', fig,axs)
    elif (phase == "Mi"):
        for N in range(2,5+1):
            aX,aY,aZ = ImportData(N, phase)
            ContourPlot(aX,aY,aZ, 'Mirror', fig,axs)
    return

# * load .dat file containing the fidelities and generates the corresponding Rabi fre
#
def ImportData(N, phase, extra=""):
    RabiFreqList, TauList = PulseParameters(N, phase)
    file_name = 'Fidelity Data/Simple_N={}_nmax={}_{}{}.dat'.format(Nbragg, Nmax, pha
    print('loading',file_name)
    z = file_data_read(file_name, sep="\t")
    print('import data loaded!')
    return RabiFreqList, TauList, z


# * Generate all Beam Splitter plots *
"""
```

```
for N in range(2,5+1):
    aX,aY,aZ = MapFidelity(N, "BS")
    #aX,aY,aZ = ImportData(N, "BS")
    ContourPlot(aX,aY,aZ, 'Beam Splitter')
"""
# * Generate all Mirror plots *
""""""
for N in range(2,5+1):
    aX,aY,aZ = MapFidelity(N, "Mi")
    #aX,aY,aZ = ImportData(N, "Mi")
    ContourPlot(aX,aY,aZ, 'Mirror')
"""

Nmax = 6
Nbragg = 2
pos0 = (Nmax-1)/2

# Hamiltonian
H = DetermineHamiltonian(Nmax)

print(H[1][0])

# Initial basis
rho0 = basis(Nmax, math.trunc(pos0+Nbragg/2))

PositionInPlot = {"PeakRabi" : 6, "PulseLength" : 0.5}

s_pos,s_neg = Estates(H, rho0, PositionInPlot, isDebug = True)

PositionInPlot = {"PeakRabi" : 4, "PulseLength" : 1}

s_pos,s_neg = Estates(H, rho0, PositionInPlot, isDebug = True)
"""
```

MuellerFidelity.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue May  4 16:51:44 2021

@author: Gioele
"""

import numpy as np
import matplotlib.pyplot as plt
import math
from qutip import *
import scipy.integrate as integrate
from matplotlib import ticker, cm
import matplotlib.pyplot as plt
import matplotlib as mpl

alpha = 20          # scaling factor for initial/final time
t0 = - alpha * 1.1 # initial time
t1 =   alpha * 1.1 # final time


Nbragg = 2

# *  Rabi Frequency as a Gaussian pulse, function of time *
#   Eq. 60
def GaussianRabiFreq(t,args):
    # expressed in units of recoil frequency r
    omega0 = args["PeakRabi"]
    tau = args ["PulseLength"]

    omega = omega0 * np.exp(- 0.5 * (t / tau) **2 )
    return omega

# * d/dt time derivative of the Rabi Frequency as a Gaussian pulse *
def Derivate_GaussianRabiFreq(t, args):
    # expressed in units of recoil frequency r
    omega0 = args["PeakRabi"]
    tau = args ["PulseLength"]
```

```
    omega = -t * omega0 * np.exp(- 0.5 * (t / tau) **2 ) / tau**2
    return omega

#   Eq. 61
def omega_eff(t, args) :
    omega0 = args["PeakRabi"]
    tau = args ["PulseLength"]

    om_eff = omega0**Nbragg * np.exp(- 0.5 * (t / tau) **2 ) / (8**(Nbragg-1) * (math
    return om_eff

# Effective Rabi freq. as the first 3 terms of a Taylor series
#   Eq. 68
def omega_eff_approx(t, args):

    sum_term = np.zeros([4,])
    for k in range(4):
        sum_term[k] = alpha(Nbragg,2*(k+1)) * abs(GaussianRabiFreq(t, args))**(2*(k+1
        #print("sum_term order",k+1,":",sum_term[k])

    om_eff = omega_eff(t,args) * (1 - np.sum(sum_term) - beta_signed(Nbragg) * (Deriv

    #print("Om_eff:",omega_eff(t,args),"t1:",1,", t2:",- np.sum(sum_term),", t3:",- b

    return om_eff

# Eigenstate |-Nbragg⟩ *
#   Eq. 18
def g_neg(t, args) :
    g = math.cos( integrate.quad(omega_eff,t0,t,args=args)[0] / 2)
    return g

# Eigenstate |+Nbragg⟩ *
#   Eq. 18
def g_pos_old(t, args) :
    g = -1j * math.sin( integrate.quad(omega_eff,t0,t,args=args)[0] / 2)
    return g


# Eigenstate |+Nbragg⟩ with integral approximated *
#   Eq. 18
```

```
def g_pos(t, args) :
    omega0 = args["PeakRabi"]
    tau = args ["PulseLength"]

    g = -1j * math.sin( omega0 * tau * math.sqrt(2 * math.pi) / 2)
    return g

#   Eqs. A1
def alpha(N,order):
    a=0
    if  (order == 2):
        a = (N+2)/(2**4 * (N**2 - 1)**2)

    elif(order == 4):
        if (N == 2):
            a = -11141/7077888
        else:
            a = - (N+4)*(4*N**5 - 15*N**4 - 32*N**3 + 12*N**2 + 64*N + 111) / (2**11

    elif(order == 6):
        if (N == 2):
            a = 1086647/9555148800
        elif (N == 3):
            a = -872713/1087163596800
        else:
            a = (N+6) * (4*N**10 - 45*N**9 + 76*N**8 + 846*N**7 + 484*N**6 - 3960*N**

    elif(order == 8):
        if (N == 2):
            a = - 20778032863/2254342434324480
        elif (N == 3):
            a = 16738435813/272747603165840000
        elif (N == 4):
            a = - 218963004049/2301307901706240000000
        else:
            polyn = 32*N**19 - 720*N**18 + 4998*N**17 + 11487*N**16 - 257532*N**15 -
            denom = 24 * 2**21 * (N**2 -1)**8 * (N**2 -4)**4 * (N**2 -9)**2 * (N**2 -

            a= -(N+8)*polyn/denom

    else: print('ERROR: alpha of order {} is not determined'.format(order))
```

```
    return a

#   Eqs. A2
def beta(N,order):

    Hn = sum(1/k for k in range(1,N))

    if (order == 1):
        b = Hn * 1j / 4

    else:
        sum_term = 0
        for k in range (2,N-1):
            Hk =  sum(1/i for i in range(1,k))
            sum_term += (N-k-1) * Hk / (k * (N-k))

        b = ((1-2/N)*Hn**2 - sum_term)/16

    return b

#   Eq. 69
def beta_signed(N):
    return -abs(beta(N,1))**2 / 2 + beta(N,2)


# * fidelity F(0,) represented in Fig. 1 *
#
def Fidelity(diff_phase, s_i, s_f, isDebug = False):
    # diff_phase : diffraction phase (theta) determining if the pulse is a Beam-Split
    # s_i, s_f : final eigenstates |+Nk> and |-Nk>

    if (diff_phase == math.pi/2) :
        # * Beam-Splitter *

        fid = abs(s_i)**2 + abs(s_f)**2

    elif(diff_phase == math.pi) :
        # * Mirror *

        fid = abs(s_f)**2
```

```
    else:
        print('ERROR: the diffraction phase in Fidelity() MUST be either pi or half-p

    if isDebug : print('fidelity:', fid)

    return fid

def PulseParameters(N,phase):

    global Nbragg


    RabiFreqMin = 0 # temporary values
    RabiFreqMax = 0 # "
    #RabiFreqVar = 0.01 # increment in Rabi frequency
    RabiFreqVar = 0.2

    TauMin = 0.6
    TauMax = 2
    TauSteps = 80

    if  (N == 2) :
        # Bragg order N=2, Truncation nmax = 6
        Nbragg = 2

        if phase == 'BS' :

            # * Beam Splitter *

            # Rabi interval
            RabiFreqMin = 2
            RabiFreqMax = 8
            #RabiFreqVar = 0.01

        else:

            # * Mirror *

            # Rabi interval
            RabiFreqMin = 2
```

```
        RabiFreqMax = 10
        #RabiFreqVar = 0.1

elif(N == 3) :
    # Bragg order N=3, Truncation nmax = 7
    Nbragg = 3

    if phase == 'BS' :

        # * Beam Splitter *

        # Rabi interval
        RabiFreqMin = 5
        RabiFreqMax = 15
        #RabiFreqVar = 0.05

    else:

        # * Mirror *

        # Rabi interval
        RabiFreqMin = 8
        RabiFreqMax = 18
        RabiFreqVar = 0.15

elif(N == 4) :
    # Bragg order N=4, Truncation nmax = 8
    Nbragg = 4

    if phase == 'BS' :

        # * Beam Splitter *

        # Rabi interval
        RabiFreqMin = 11
        RabiFreqMax = 23#24
        #RabiFreqVar = 0.05
    else:

        # * Mirror *
```

```
            # Rabi interval
            RabiFreqMin = 15
            RabiFreqMax = 29
            RabiFreqVar = 0.15

    elif(N == 5) :
        # Bragg order N=52, Truncation nmax = 11
        Nbragg = 5

        if phase == 'BS' :

            # * Beam Splitter *

            # Rabi interval
            RabiFreqMin = 12# 20
            RabiFreqMax = 29# 34
            RabiFreqVar = 0.2# 0.05

        else:

            # * Mirror *

            # Rabi interval
            RabiFreqMin = 25
            RabiFreqMax = 39
            RabiFreqVar = 0.15

    else : return

    # X var
    RabiFreqList = np.linspace(RabiFreqMin, RabiFreqMax, int((RabiFreqMax-RabiFreqMin

    # Y var
    TauList = np.linspace(TauMin, TauMax, TauSteps)

    return RabiFreqList, TauList

def MapMullerFidelity(N, phase):

    RabiFreqList,TauList = PulseParameters(N, phase)
```

```
    print('x points:',np.shape(RabiFreqList))
    print('y points:',np.shape(TauList))


    # save-file name
    file_name = "Fidelity Data/"
    file_name += "N={}_Muller_".format(Nbragg)
    file_name += phase
    file_name += ".dat"

    # target diffraction phase
    if phase == 'BS' : diff_phase = math.pi/2
    elif phase == 'Mi' : diff_phase = math.pi

    # Z var
    FidList = np.zeros((len(TauList),len(RabiFreqList)))
    print('z progress:')

    for i in range(len(RabiFreqList)):
        for k in range(len(TauList)) :
            pos = {"PeakRabi" : RabiFreqList[i], "PulseLength" : TauList[k]}

            s_f = g_neg(t1, pos)
            s_i = g_pos(t1, pos)
            #      [k,-i-1]
            FidList[k,i] = Fidelity(diff_phase, s_i, s_f, isDebug= False)

        print(i+1,'/',len(RabiFreqList),'...')

    print('z points:',np.shape(FidList))

    # store z in .dat
    print('Not yet saving',file_name)
    #print('saved in',file_name)
    #file_data_store(file_name, FidList, sep="\t", numtype='real')
                             # FidList.T
    # return x,y,z
    return RabiFreqList, TauList, FidList#.T

def ContourPlot(x, y, z, label=""):
```

```
    fig, axs = mpl.pyplot.subplots()
    cs = axs.contourf(x, y, z, cmap=mpl.cm.viridis)

    mpl.rcParams['lines.linewidth'] = 0.5
    plt.rcParams['axes.formatter.min_exponent'] = 2
    axs.contour(cs, colors='k')
    axs.set_title('N={}'.format(Nbragg))
    plt.yscale('log')
    plt.ylabel(label)
    plt.figure(figsize=(8, 6),dpi=300)
    cbar = fig.colorbar(cs)

    #fig.set_size_inches(18.5, 10.5, forward=True)

    return

def ImportData(N, phase, extra=""):
    RabiFreqList, TauList = PulseParameters(N, phase)
    file_name = 'Fidelity Data/Muller_N={}_{}{}.dat'.format(Nbragg, phase, extra)
    print('loading',file_name)
    z = file_data_read(file_name, sep="\t")
    print('import data loaded!')
    return RabiFreqList, TauList, z


# * Beam Splitter *
"""
for N in range(2,5+1):
    aX,aY,aZ = MapMullerFidelity(N, "BS")
    ContourPlot(aX,aY,aZ, 'Beam Splitter')

# * Mirror *
"""
for N in range(2,2+1):
    aX,aY,aZ = MapMullerFidelity(N, "Mi")
    ContourPlot(aX,aY,aZ, 'Mirror')
""""""
```

Bibliografia

[1] Jan-Niclas Siemß, Florian Fitzek, Sven Abend, Ernst M. Rasel, Naceur Gaaloul, Klemens Hammerer, "Analytic theory for Bragg atom interferometry based on the adiabatic theorem", Phys. Rev. A 102, 033709 (2020)

[2] Holger Müller, Sheng-wey Chiow, and Steven Chu, "Atom-wave diffraction between the Raman-Nath and the Bragg regime: Effective Rabi frequency, losses, and phase shifts", Phys. Rev. A 77, 023609 (2008)

[3] Peter Asenbaum, Chris Overstreet, Minjeong Kim, Joseph Curti, Mark A. Kasevich, "Atom-Interferometric Test of the Equivalence Principle at the 10^{-12} Level", Phys. Rev. Lett. 125, 191101, (2020)

[4] C. Keller, J. Schmiedmayer, A. Zeilinger, T. Nonn, S. Dürr, and G. Rempe, "Adiabatic following in standingwave diffraction of atoms", Appl. Phys. B 69, 303 (1999).

[5] Yoshio Torii, Yoichi Suzuki, Mikio Kozuma, Toshiaki Sugiura, Takahiro Kuga, Lu Deng and E. W. Hagley, "Mach-Zehnder Bragg interferometer for a Bose-Einstein condensate", Phys. Rev. A 61, 041602(R) (2000).

[6] Emile Mathieu, "Mémoire sur le mouvement vibratoire d'une membrane de forme elliptique", Journal de Mathématiques Pures et Appliquées (2) 13, 137 (1868)

[7] Clarence Zener, "Non-Adiabatic Crossing of Energy Levels", Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, 137, 696 (1932)

[8] S. S. Szigeti, J. E. Debs, J. J. Hope, N. P. Robins, and J. D.Close, "Why momentum width matters for atom interferometry with Bragg pulses", New J. Phys. 14, 023009 (2012).

[9] Daniel A. Steck, "Quantum and Atom Optics", available online at http://steck.us/teaching (revision 0.12.6, 23 April 2019).

[10] M. Born and V. A. Fock, "Beweis des Adiabatensatzes", Zeitschrift für Physik A. 51 (3–4): 165–180. (1928)

[11] J.-N. Siemß and K. Hammerer, "Data and scripts for figures 1 to 9", Zenodo (2020).