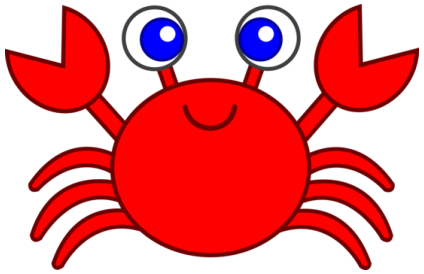
The CRAB algorithm

(and the Nelder-Mead algorithm)

The CRAB algorithm

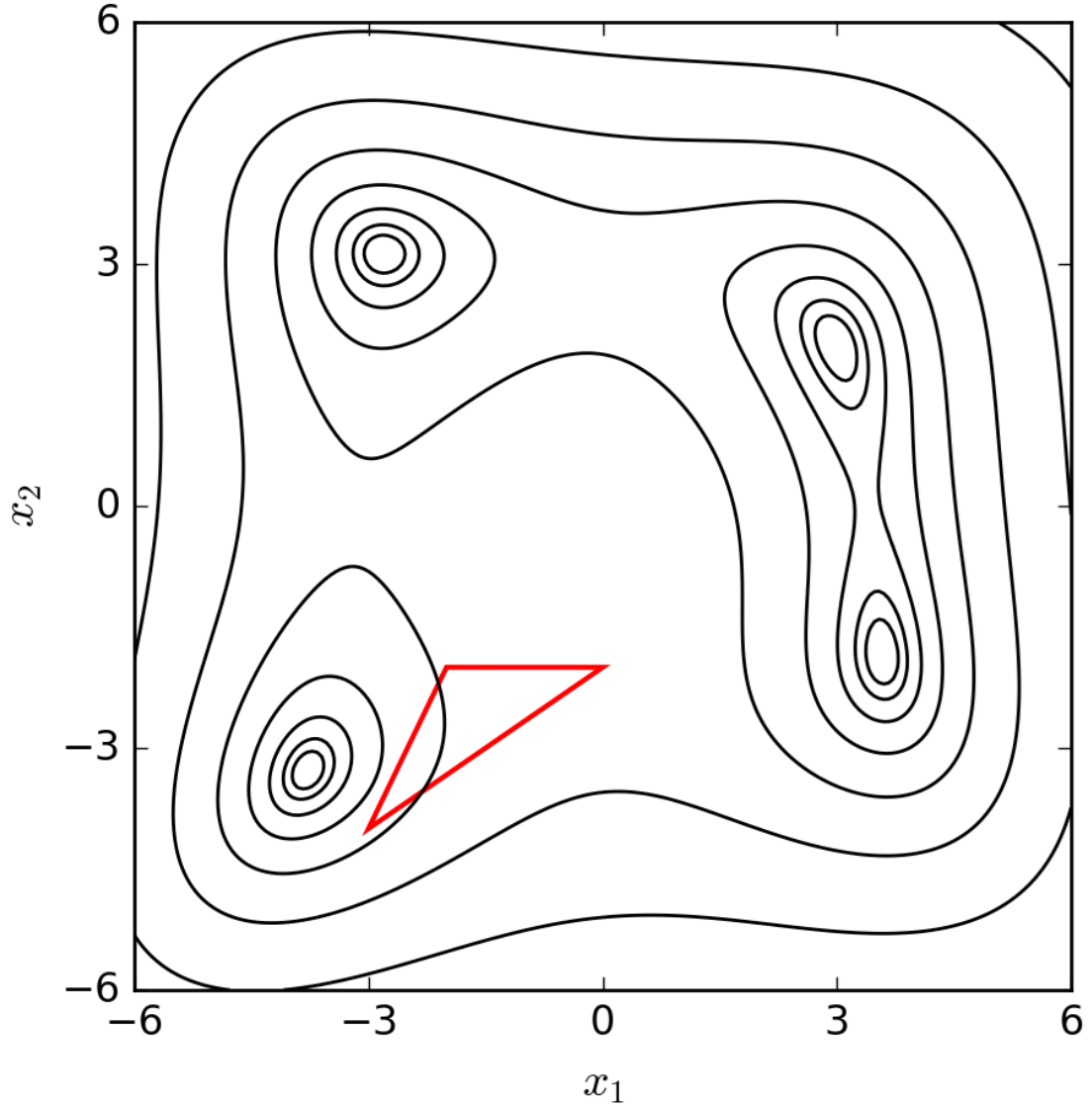


- CRAB: Constrained RAndom Basis optimization of a system with Hamiltonian $\hat{H}$ with a time-dependent control field $\Gamma(t)$ [1]
- As always, construct an error function (e.g. including the projection of the final and desired states, constraining field amplitudes, etc.)
- Start with an initial guess for the control field $\Gamma_0(t)$ (usually $\Gamma_0(t) = 1$)
  - Modify the control field $\Gamma(t) \rightarrow \Gamma_0(t)g(t)$
  - $g(t) = 1 + \lambda(t) \sum_n A_n \sin(\omega_n t) + B_n \cos(\omega_n t)$
  - We see an *envelope* function $\lambda(t)$ that, e.g., ensures smooth turn-on and turn-off (i.e. a Gaussian or $\sin^2 t$ function)
- Cast the optimization now as a local optimization of vectors $\vec{A}$ and $\vec{B}$
  - Can choose either random ω_n or pick ω_n based on knowledge of the problem at hand (e.g. avoiding/including resonances)
  - Use well-established local optimization methods (e.g. Nelder-Mead optimization) to solve the problem
  - You can choose other bases! Fourier is typically a good start, though.

[1] https://arxiv.org/abs/1103.0855

The Nelder-Mead algorithm

- The CRAB algorithm is really just a way to parameterize your control, but you have to do the optimization with something—commonly the Nelder-Mead method
- This is known as a *simplex* method, where we define (in N dimensions) $N + 1$ vertices, then the simplex works its way to the minimum.
- Gradient-free, local search method



How to Nelder-Mead

- Step -1:
  - Define your initial simplex spread $\{\vec{x}_j\}$ throughout your search space, evaluate functional J for each point
- Step 0:
  - Rank the vertices such that $J(\vec{x}_0) \leq J(\vec{x}_1) \leq \cdots \leq J(\vec{x}_N)$

How to Nelder-Mead

- Each iteration:
  - Calculate the *centroid* of all points except the worst: $\vec{x}_M = \frac{1}{N}\sum_{j=0}^{N-1} \vec{x}_i$
  - Compute the *reflected* point—reflect the worst point about the centroid

$$\vec{x}_r = \vec{x}_M + \alpha(\vec{x}_M - \vec{x}_N)$$

  - If $J(\vec{x}_1) < J(\vec{x}_r) < J(\vec{x}_{N-1})$, replace $\vec{x}_N$ with $\vec{x}_r$ and go back to Step 0
  - Elseif $J(\vec{x}_r) < J(\vec{x}_1)$, expand in the direction of $\vec{x}_r$ by computing $\vec{x}_e = \vec{x}_M + \gamma(\vec{x}_r - \vec{x}_M)$
    - If $J(\vec{x}_e) < J(\vec{x}_r)$, replace $\vec{x}_N$ with $\vec{x}_e$ and go to Step 0, else replace $\vec{x}_N$ with $\vec{x}_r$ and goto 0

How to Nelder-Mead

- Each iteration:
  - Calculate the *centroid* of all points except the worst: $\vec{x}_M = \frac{1}{N}\sum_{j=0}^{N-1} \vec{x}_i$
  - Compute the *reflected* point—reflect the worst point about the centroid

$$\vec{x}_r = \vec{x}_M + \alpha(\vec{x}_M - \vec{x}_N)$$

  - Elseif $J(\vec{x}_r) \geq J(\vec{x}_{N-1})$, contract
    - If $J(\vec{x}_r) > J(\vec{x}_N)$, $\vec{x}_c = \vec{x}_M + \rho(\vec{x}_r - \vec{x}_M)$, if $J(\vec{x}_c) < J(\vec{x}_r)$, replace $\vec{x}_N$ with $\vec{x}_C$, goto 0, else *shrink*
    - Elseif $J(\vec{x}_r) \leq J(\vec{x}_N)$, $\vec{x}_c = \vec{x}_M + \rho(\vec{x}_N - \vec{x}_M)$, if $J(\vec{x}_c) < J(\vec{x}_N)$, replace $\vec{x}_N$ with $\vec{x}_C$, goto 0, else *shrink*
  - Shrink: replace all points except $\vec{x}_0$ with $\vec{x}_j = \vec{x}_0 + \sigma\left(\vec{x}_j - \vec{x}_0\right)$, goto 0

When to stop the Nelder-Mead/CRAB

- You can always stop when your functional value reaches a certain threshold J_{min}
- Likewise, you can compute the size of your simplex and stop when it reaches a certain threshold (look up Cayley-Manger determinant)
- Your initial simplex matters, so you can always restart a few times and see if you do better (or try and uniformly sample your search space)
- Note that if you are coupling NM to CRAB, you can always re-roll with a new set of frequencies (call this dressed CRAB or dCRAB)

$$g_{new}(t) = g_{old} + \Gamma_0(t)\left(1 + \lambda(t)\sum_{n'} A_{n'}\sin(\omega_{n'}t) + B_{n'}\cos(\omega_{n'}t)\right)$$

GROUP, or gradient-CRAB

- The GROUP method [1] combines CRAB with a gradient-based optimizer in that it uses the chopped randomized basis, but it calculates gradients in that basis $\{f_n(t)\}$

$$u(t) = u_0(t) + \lambda(t) \sum_{k=1}^{M} c_k f_k(t)$$

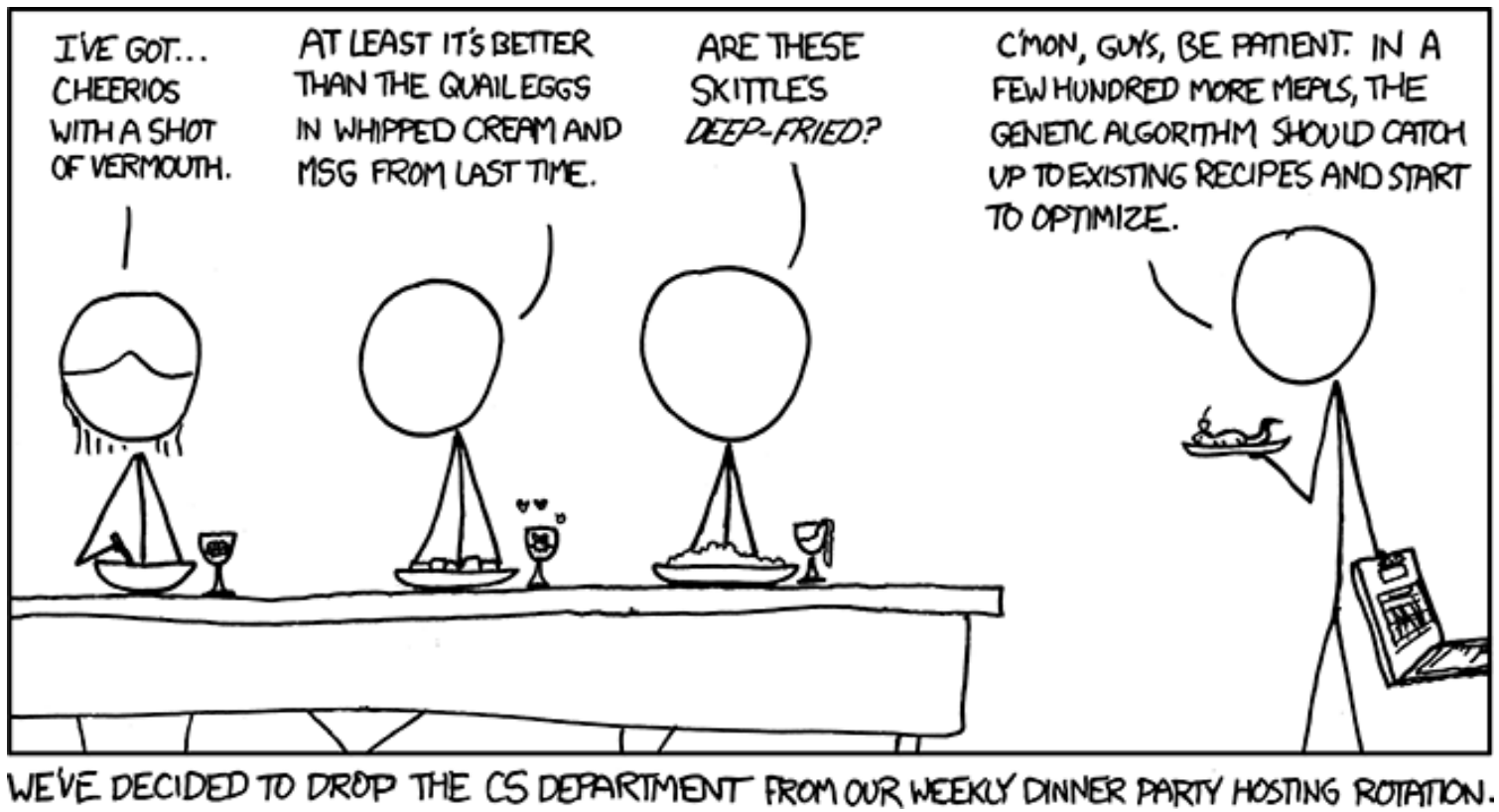
- Solution very similar to GRAPE but the gradient is a bit more difficult to calculate

$$\frac{\partial J}{\partial c_k} = \frac{\partial J}{\partial u_n} \frac{\partial u_n}{\partial c_k}$$

- This is a nice solution for cases where you know a reasonable basis set but have an analytic problem

[1] https://arxiv.org/abs/1802.07509

The genetic algorithm



The genetic algorithm, a recipe

- Initialize a population of M "individuals" $\{\vec{x}_m\}$, this is your first generation
- These can, for example, be the Fourier amplitudes $\tilde{f}(\omega)$ of a time-varying signal $f(t)$ and an envelope function $\lambda(t)$, such that

$$x(t) = \lambda(t)\mathcal{F}\{\tilde{f}(\omega)\}$$

- As with CRAB, Step 0 involves evaluating your functional over each individual $\{J(\vec{x}_m)\}$ and ranking them from best to worst $J(\vec{x}_0) \leq J(\vec{x}_1) \leq \cdots \leq J(\vec{x}_N)$. This is called your *fitness*.

The genetic algorithm, a recipe

- Typically, in a genetic algorithm, you keep the N_{best} best individuals (survival of the fittest) and drop the N_{worst} worst individuals.
- Then, to generate the $N - N_{best}$ remaining individuals for the next generation, choose from the $N - N_{worst}$ remaining individuals and, at random, choose from four different sets of operations
  - One-point crossover
  - Two-point crossover
  - Creep
  - Mutation

The mutation operations

- Given two random individuals $\vec{y} = \{y_0, y_1, \dots y_N\}$ and $\vec{z} = \{z_0, z_1, \dots, z_N\}$
- One-point crossover: choose a random index $i_1 \in (0, N)$

$$\vec{c} = \{y_0, y_1, \dots y_{i_1}, z_{i_1+1}, \dots z_N\}$$

- Two-point crossover: choose two random indices $i_1, i_2 \in (0, N), i_2 > i_1$

$$\vec{c} = \{y_0, y_1, \dots y_{i_1}, z_{i_1+1}, \dots z_{i_2}, y_{i_2+1}, \dots, y_N\}$$

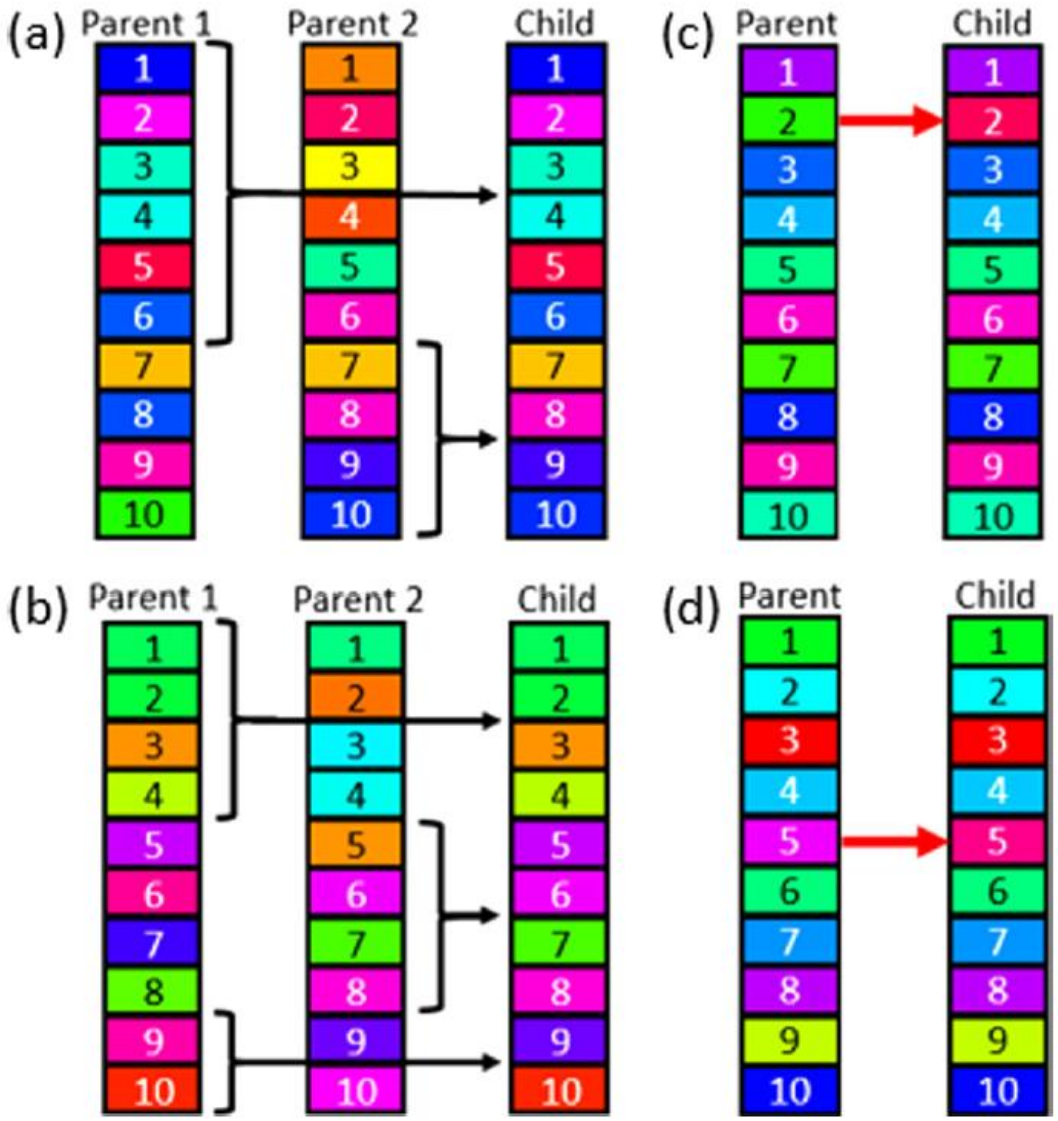
- Creep: choose a random individual $\vec{y}$ random index $i_1 \in [0, N]$, a random number $r_C \in (0,1]$ and a (pre-defined, unchanging) *creep rate* C

$$\vec{c} = \{y_0, y_1, \dots y_{i_1} + C(0.5 + r_C), y_{i_1+1}, \dots, y_N\}$$

- Mutation: choose a random individual $\vec{y}$ random index $i_1 \in [0, N]$, a random number $r_M \in (0, M]$ given a (pre-defined, unchanging) *mutation rate* M

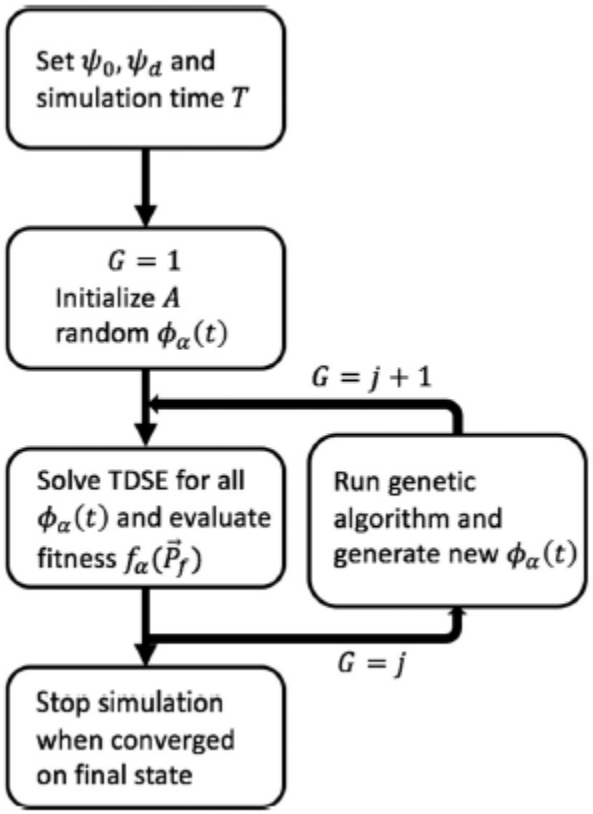
$$\vec{c} = \{y_0, y_1, \dots, r_M, y_{i_1+1}, \dots, y_N\}$$

The mutation options, graphically



The genetic algorithm recipe

- Once you have generated your new set of N individuals, go to Step 0 and repeat.
- Convergence can be slow, and the higher N is, the worse it will be
- This method is *gradient- free* and *global*
- Depends heavily on how you initialize your set, but non-deterministic regardless

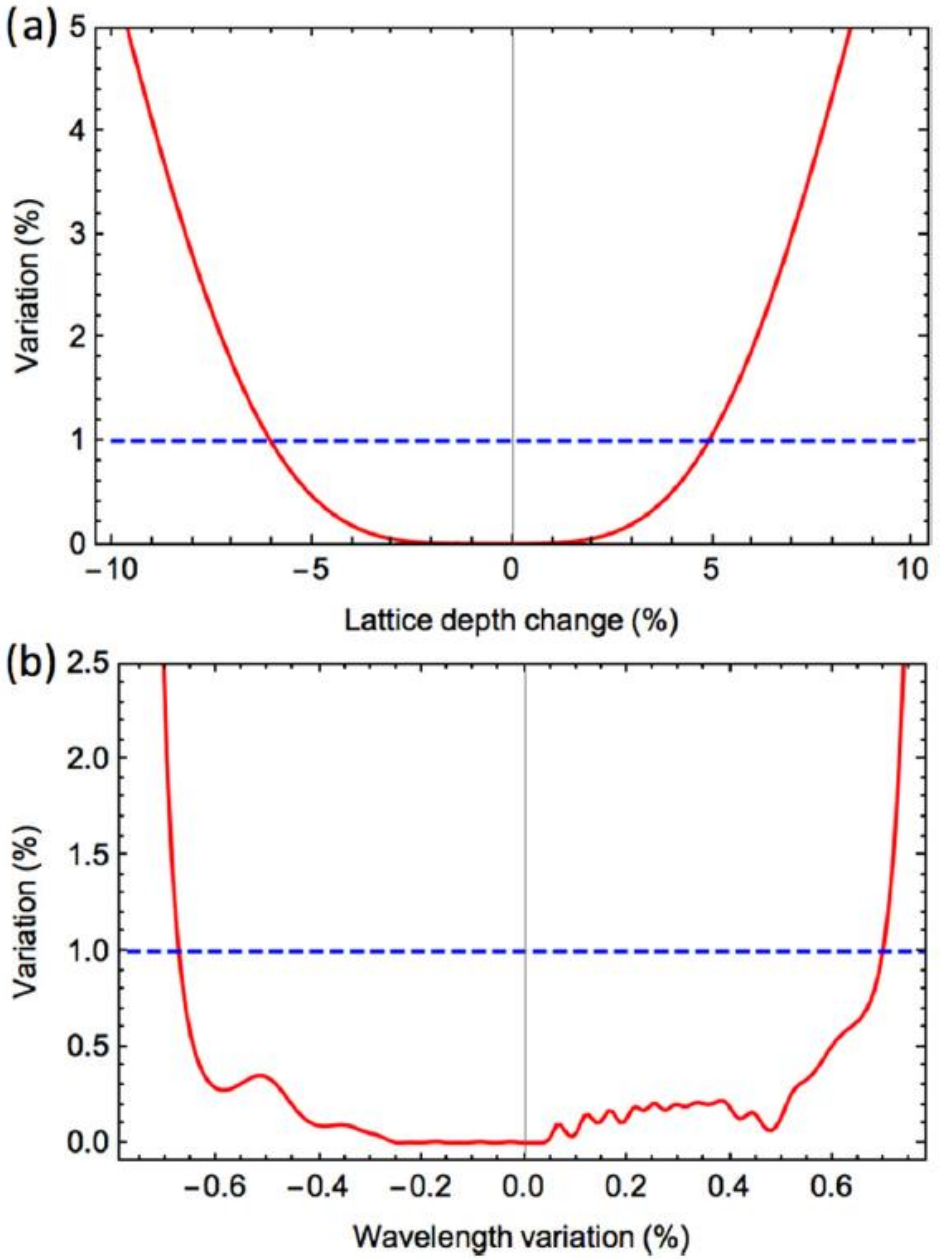


Robust control

(and why this is not a well-defined question)

Robust control in practice

- So, applying classical robust control techniques to quantum is…not easy. What do we do?
- Two types of robustness:
  - Robustness to variations in the control
  - Robustness to variations in the other system parameters
- There are a few ways to tackle this problem:
  - Simple parameter scans
  - Monte-Carlo sampling
  - Adding noise to your controls
- Simple parameter scans are straightforward, but often oversimplified…



Monte Carlo methods

- Monte Carlo methods are excellent ways to simulate complex systems (e.g. quantum many-body problems)
- Basic idea: randomly sample your parameter space, then perform some sort of computation on that parameter space
- With enough samples, your mean will approach the true expected value of what you are trying to determine.
- Convergence is roughly $1/\sqrt{N}$ for N samples

Monte Carlo for robust control: a recipe

- Solve your control problem, find a good control $\vec{u}$
- Determine your uncertain parameters and how they vary (is it uniform, or Gaussian, or…)
- Randomly sample from these parameters (within their distributions) and determine how your transfer fidelity changes as you run your system with these modified parameters
- The spread in F will determine the robustness of your controller (lower spread = more robust)
- As an aside: you can run some optimization algorithms with this variation built-in to your system. Some will succeed, others will fall on their face

Doing better than Monte Carlo

- You can improve on Monte Carlo by making use of *quasi-Monte Carlo* sampling, where your sampling is not truly random, but more uniformly distributed
- Use a low-discrepancy sequence like the Sobol sequence
- Reduces convergence to $1/N$
- There are some drawbacks, mainly that care must be taken with, e.g., Gaussian distribution of variables (Basu and Owen, SIAM J. Numerical Analysis **54**, 3, (2016))

Other important bits to note

- If you use traditional Monte Carlo sampling, you can try *Latin hypercube sampling*
  - Like having N rooks on a chessboard that do not threaten each other
- Note also that random sampling on a sphere is tricky!
  - In 2D differential surface element is $rd\phi \times r\sin(\phi)\, d\theta$
  - Sample more closer to the equator than the poles!
  - This does **not** generalize nicely in arbitrary higher dimensions!

Adding noise to a control

- With a time-dependent control, one can check robustness by simply adding noise to the function
- One straightforward method: add Gaussian white noise to the control
  - Additive Gaussian white noise (AGWN) is uniform in frequency across the entire system (i.e. there is no correlation between the noise at t and $t + \delta t$) and samples from a Gaussian distribution with zero average
- How much noise to add? That's largely a question for your experimentalist friends
- You can also add a more sophisticated noise model (e.g. 1/f noise) but AGWN takes care of a lot of common cases