Soliton Solutions of the Complex Ginzburg Landau Equation

By Max Proft
u5190335

Contents

Abstract

Abstract goes here.

Acknowledgements

Chuck in some very cliché acknowledgements here.

Chapter 1

Introduction to the CGLE

The Complex Ginzburg Landau equation (CGLE) is a differential equation that describes a range of quantum phenomena, such as ——————-. It is typically written in the following form:

$$i\psi_t + \frac{D}{2}\psi_{xx} + |\psi|^2\psi + \nu|\psi|^4\psi = i\delta\psi + i\epsilon|\psi|^2\psi + i\beta\psi_{xx} + i\mu|\psi|^4\psi$$

ψ refers to the normalised field (normalised with respect to what?)
$D = \pm 1$. if $D = 1$ then normal. This means that as the optical frequency increases, the group velocity decreases. If $D = -1$ then the group velocity dispersion is anomalous, which is the opposite.
$\beta > 0$ is parabolic gain and $\beta < 0$ effectively means that sharp spikes are filtered out. (This doesn't make sense - shouldn't it be the other way around?) causes spectral filtering
δ is the linear gain/loss depending on whether it is positive/negative.
ϵ is nonlinear gain which can arise for a variety of reasons, such as saturable absorption (absorbtion decreases/reflectivity increases as intensity increases.)
μ and ν, when negative, represents the saturation of the NL gain and NL refractive index respectively.

Chapter 2

Derivations of the CGLE

2.1 For a Quantum System

2.2 For a Classical System

Chapter 3

Types of solitons

There are many types of solitons, however an explicit expression for the function usually cannot be found.

While many explicit solutions for the schrodinger equation have been found, with additional nonlinearity means that finding explicit solutions to the CGLE is often not possible.

(Jia-Min Zhu and Zheng-Yi Ma (2007),Pierre Hillion (2012), Mihalache and Panoiu (1992), Yan-Ze Peng and Krishnan (2007) = Schrodinger equation)

We can classify solitons depending on their behaviour. Stationary solitons maintain a constant amplitude. Pulsating solitons are another type, and can be categorised into subsets such as "plain pulsating", "exploding" and creeping.

Plain pulsating solitons are periodic and don't exhibit any significant features, as given in figure 3.1. Exploding solitons are characterised by a sharp jumps in their energy profile, as given in figure 3.2, with the 2D intensity given in figure 3.3. Other solitons will, instead of exploding outwards, give a high intensity spike, as in figure 3.5 whose energy increases by ¿10 times because of the spike As parameters are varying, unusual behaviour can also be seen. For example, with the set of parameters (———), when varying ϵ, periodic doubling can be seen, as in figure **??**. As the parameter c is increased, the the height of the solitons changes to a period 2 then a period 4 cycle.

We also get solitons that creep sideways as they oscillate. An example is given in figure ——- of a soliton that creeps to the right forever. Another example is in figure 3.6

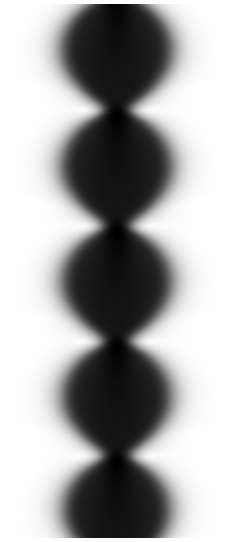

Figure 3.1: The 2D plot of a plain pulsating soliton with parameters .....

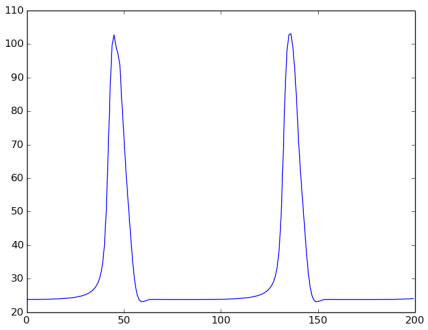

Figure 3.2: The profile of an exploding soliton with parameters .....

Figure 3.3: The 2D intensity of an exploding soliton with parameters .....

Figure 3.4: The 2D intensity of a high amplitude soliton with parameters .....

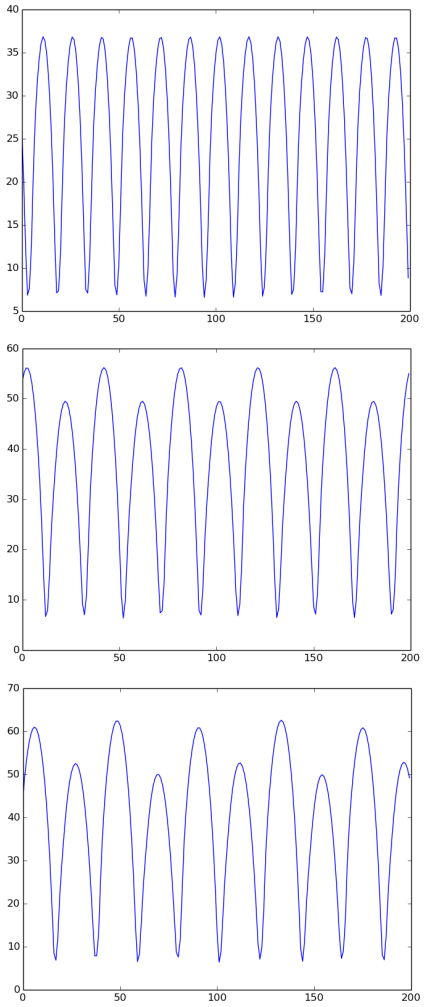

Figure 3.5: The energy profile of 3 solitons, with periodic doubling being seen. Parameters ....., for these images, the actual domain is from 0 to 100.

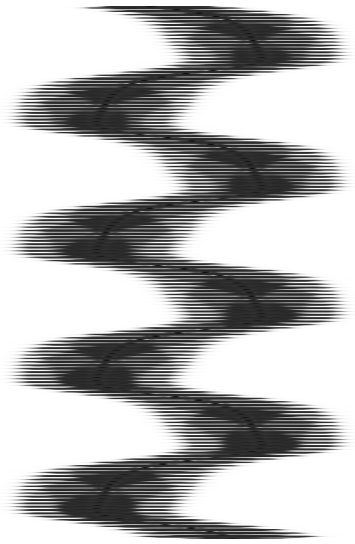

Figure 3.6: A creeping, pulseating soliton that is also periodic. Note that the weird patterning inside the image is from aliasing. params ———

Chapter 4

Mathematical Theory Behind Solving the CGLE

4.1 A Comparison of Different Methods

A finite difference method was used initially to get it all going. This is bad because you do $\delta\psi/\delta x$, so over a small time period, the $\delta\psi$ term can lose several digits of precision, not good. There are a number of other ways for this to be solved though.

For the nonlinear schrodinger equation, (Taha and Ablowitz) compared various methods for solving this. While under certain conditions, some other methods were found to be better than the split step fourier method, overall it was found to be a faster algortihm. Given that the CGLE is an extension of the NLSE, this means that this should also be a reasonably good method to do this part as well. This has previously been done by other authors as well, (Wang, Zhang).

4.2 Split Step Fourier and Runge-Kutta Method

4.2.1 Linear evolution (Fourier)

Here we are solving the equation:

$$\frac{d\psi}{dt} = A\frac{d^2\psi}{dx^2} + B\psi$$

Defining the fourier transform by:

$$\hat{\psi}(f) = \int_{-\infty}^{\infty} \psi(x) e^{-i2\pi x f} dx$$

By taking the fourier transform of both sides, the equation becomes:

$$\frac{d\hat{\psi}}{dt} = C(-2\pi if)^2 D\hat{\psi}(f) + \hat{\psi}(f) = (C - D4\pi^2 f^2)\hat{\psi}(f)$$

$$\implies \hat{\psi}_t(f) = e^{(C-D4\pi^2 f^2)t}\hat{\psi}_0(f)$$

And so upon inverse fourier transforming, we get the solution to this DE, where $\hat{\psi}_t(f)$ is the fourier transform at time t. The Fourier transform can be done numerically with a discrete fourier transform, as described in a later subsection.

4.2.2 Nonlinear evolution (Runge-Kutta)

Here we are solving the equation:

$$\frac{d\psi}{dt} = C|\psi|^2\psi + D|\psi|^4\psi$$

The runga kutta method allows you to numerically solve a 1st order differential equation. which are of the form $\frac{dy}{dt} = f(t, y)$. Similar to Euler's method (which is basically just doing a derivative from first principles), except that the runge kutta method has better convergence than this. (it goes as h^5 as opposed to Euler's Method which is h^2, the backward euler formula, h^2, or improved euler formula which goes as h^3. Here h is the step size.

The steps for doing this are as follows: Step 1: define $f(t, y) = f(y) = C|y|^2 y + D|y|^4 y$ Step 2: Have an initial value for t and y, and step size and number of steps Step 3: define the following:

$$K_1 = f(t, y) = f(y)$$

$$K_2 = f(t + 0.5h, y + 0.5 * h * K_1) = f(y + h * K_1/2)$$

$$K_3 = f(t + 0.5h, y + 0.5 * h * K_2) = f(y + h * K_2/2)$$

$$K_4 = f(t + h, y + h * K_3) = f(y + h * K_3)$$

Step 4: we then progress forwards t and y in the following way:

$$t \to t + h$$

$$y \to y + (K_1 + 2K_2 + 2K_3 + K_4) * h/6$$

4.2.3 The relationship between a Fourier Transform and a Discrete Fourier Transform:

Below works for periodic boundary conditions, or if you assume that the function goes to zero at the boundary. For ease of proving the relationship, we assume the latter.

Defining the Fourier transform as:

$$\tilde{\psi}(f) = \int_{-\infty}^{\infty} \psi(x) e^{-i2\pi xf} dx \approx \int_{0}^{L} \psi(x) e^{-i2\pi xf} dx$$

Let $\Delta x = L/N$ for some integer N.

$$\approx \sum_{n=0}^{N-1} \psi(n\Delta x) e^{-i2\pi n\Delta xf} \Delta x$$

With the same arguments, we get the inverse Fourier transform to be:
($\Delta f = (f_1 - f_0)/N$, for some values of f_1 and f_2)

$$\psi(x) = \int_{-\infty}^{\infty} \tilde{\psi}(f) e^{i2\pi xf} df \approx \int_{f_0}^{f_1} \tilde{\psi}(f) e^{i2\pi xf} df \approx \sum_{m=0}^{N-1} \tilde{\psi}(f_0 + m\Delta f) e^{i2\pi n\Delta xf_0} e^{i2\pi xm\Delta f} \Delta f$$

Let $\psi(n\Delta x) = \psi_n$ and $\tilde{\psi}(f_0 + m\Delta f) = \tilde{\psi}_m$ and let $f_0 = -\pi/\Delta x$

$$\tilde{\psi}_m \approx \Delta x \sum_{n=0}^{N-1} \psi_n e^{-i2\pi n\Delta xf_0} e^{-i2\pi nm\Delta x\Delta f}$$

$$\psi_n \approx \Delta f e^{i2\pi n\Delta xf_0} \sum_{m=0}^{N-1} \tilde{\psi}_m e^{i2\pi nm\Delta x\Delta f}$$

Since the definition of the discrete Fourier transform is

$$\tilde{A}_m = \sum_{n=0}^{N-1} A_n e^{-i2\pi nm/N}$$

$$A_n = \frac{1}{N} \sum_{m=0}^{N-1} \tilde{A}_m e^{i2\pi nm/N}$$

We now have a way to compute (inverse) Fourier transforms given a set of data that is a sufficiently good approximation of the initial function.

We want $f_1 - f_0 = 1/\Delta x$ so that the frequency difference corresponds to the difference between adjacent elements. This means we get $\Delta f = 1/(N\Delta x)$. We then choose to take $f_0 = \frac{-1}{2\Delta x}$ is chosen so it satisfies the Nyquist limit.

$$\tilde{\psi_m} \approx \Delta x \sum_{n=0}^{N-1} \psi_n e^{i\pi n} e^{-i2\pi nm/N} = \Delta x \sum_{n=0}^{N-1} \psi_n (-1)^n e^{-i2\pi nm/N}$$

$$\psi_n \approx \frac{e^{-i\pi n}}{N\Delta x} \sum_{m=0}^{N-1} \tilde{\psi}_m e^{i2\pi nm/N} = \frac{(-1)^n}{N\Delta x} \sum_{m=0}^{N-1} \tilde{\psi}_m e^{i2\pi nm/N}$$

4.2.4 Combining these operations

With the split step fourier method, we assume that the linear and nonlinear terms do not affect each other over short timescales. This means that we can propegate forwards according to the linear evolution equation, and then propegate forwards according to the nonlinear evolution equation, and we should get a good approximation for the solution to the original differential equation. This can be represented as the following, where $\hat{N}$ refers to the nonlinear evolution operator, and $\hat{L}$ refers to the linear evolution operator.

$$\psi(t) = e^{\hat{N}(t)} e^{\hat{L}(t)} \psi(0)$$

It turns out, however, that a more stable way of doing this evolution is with the following (notice that the two outside terms are both evolution operators for half of the time period):

$$\psi(t) = e^{\hat{N}(t/2)} e^{\hat{L}(t)} e^{\hat{N}(t/2)} \psi(0)$$

Baker Campbell Hausdoff Lie Product Formula Trotter Product Formula

4.3 Testing the Code

I compared with mathematica and got the same answer with initial conditions —— and step size —— for both the linear and NL parts.
Also I subbed in exact solns to the NLSE and it matched.
Upon decreasing the step size, if the solution doesn't change significantly, we expect a very similar answer.

Additionally, as previously done, there are a number of papers which give interesting solutions to the CGLE, and we can reproduce these solitons.

Chapter 5

Machine Learning Theory

5.1 Different Types of Machine Learning

I want to enter in an image, and get it to tell me if it is oscillating/etc.
I want another machine learning algorithm to predict the required precision, and wait time before recording the output.
Monte Carlo and gradient descent to find new solns.

5.2 Linear Regression

Former Title: Different Options for the Machine Learning Algorithm
Another Title Removed: A detailed look at neural networks

Step 1: Define a cost function:
Suppose we have input parameters $\vec{x}_i$ with the true value given by y_i. For linear regression, we choose $\vec{\theta}$ such that our prediction for the value of y_i is $\vec{\theta} \cdot \vec{x}_i$. We want to vary the vector $\vec{\theta}$ such that the predictions match as best as possible. This is typically done by providing a cost function that is lowest when the predictions match the true value best. A typical cost function is (least squares):

$$J_\theta = \sum_i \left(\vec{\theta} \cdot \vec{x}_i - y_i \right)^2$$

Suppose we have two parameters, we might get an image as shown below. The minimum of this graph give θ which causes the best predictions. Typically the algorithm to find the minimum is done by continually taking the steepest path. However the algorithm may end up finding a local minima, as in the subsequent figure. Octave has lots of fancy functions that can do this really efficiently, and can try to avoid local minima.

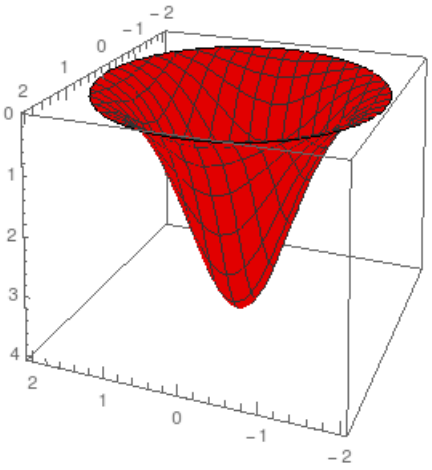

Figure 5.1: Gradient descent example

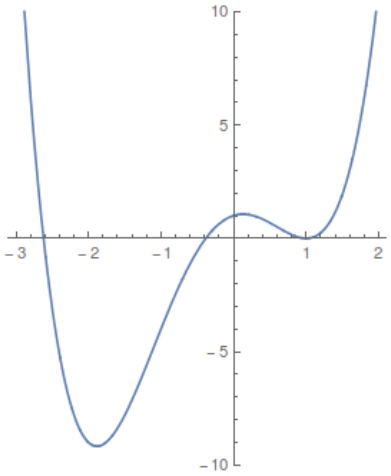

Figure 5.2: Gradient descent example

We can increase the number of input features. One feature we want to add is an intercept term, which will equal 1 for every training example. Other features we might want to add, for example, would be the polynomial features $\{x_1^2, x_1x_2, x_1x_3, ...\}$. However if we add in these extra features, we are faced with another issue, and that is overfitting. If we have a high degree polynomial, it becomes much easier to overfit the data. In order to prevent this, we can adjust the cost function to:

$$J_\theta = \frac{1}{2m}\sum_i \left(\vec{\theta}\cdot\vec{x}_i - y_i\right)^2 + \frac{\lambda}{2m}|\vec{\theta}|^2$$

By doing this, if several of the elements of $\vec{\theta}$ are large, then the cost will increase, and so only the important features will remain large. However for this to work we must

normalise the data, and in my program this is done by transforming each feature $x_i^{(j)}$ in the following manner, where the i indicates that it is the i^{th} training example, and the j index refers to the j^{th} component of the vector. Below $\mu^{(j)}$ refers to the mean of the j^{th} component of all the training examples, and $\sigma^{(j)}$ the standard deviation

$$x_i^{(j)} \rightarrow \frac{x_i^{(j)} - \mu^{(j)}}{\sigma^{(j)}}$$

This ensures that all of the features are on the same scale, and so the elements of θ are penalised equally by the cost function.

5.3 Training and Testing the Algorithm

Tried initially with only linear terms, then expanded to polynomial terms with regularisation and it was able to fit data with polynomial features $(0.5x_1 + x_2^3)$. This game me the confidence that the algorithm was working correctly, and so I could go ahead and train the algorithm on real data with confidence.

My algorithm went up to 4th order polynomial terms, and with 6 different parameters being varied, this results in a total of 210 features.

Chapter 6

The result from much work

6.1 The sorts of things that I found

Conclusion

Appendix A

Appendix B

Bibliography

[1] Name *Title*, Publishing info, etc

[2] Another Guy *title2* pub,etc.