Non-Perturbative QFT Problem Sheet - Question 4

Matthew Hawes

April 21, 2017

Abstract

This question is in two sections, firstly a description of the algorithm used, followed by the results of the simulations. All code was written in c++ and is available at `www.github.com\mghawes`. This was my first foray into the world of c++ so some of the implementations may not be perfect. All image rights belong to the author.

1 Algorithm

The algorithm, implemented in c++, relies heavily on the concept of Object-Oriented programing to improve readability as well as speed. The general principal in this case being to create objects with methods and attributes that can be initialized once and then iterated on very efficiently. The key point being that we do as much setup as possible *before* beginning to perform the metropolis updates. The simplest example of this is the abstraction of the lattice into *links* and *staples*, which will now describe as a primer.

1.1 Lattice

The most efficient way to store a $U(1)$ lattice is to store the phases θ_l of the group elements $U_l = \mathrm{e}^{i\theta_l}$ that live on the links l. In memory these links will exist as just a contiguous block, or array, of `double` values. Thus, we have to prescribe some mapping from the $2+1$D lattice to the elements of the 1D array. The mapping chosen is to split the array into three blocks with the first third containing only x directed links and the 2nd and 3rd containing

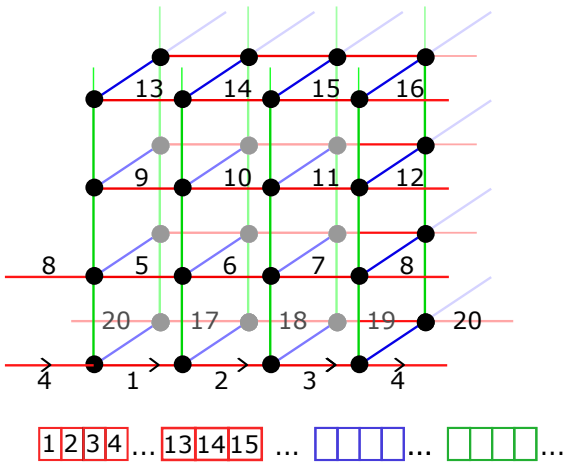


Figure 1: 'Unraveling' the links of the periodic lattice into a 1D array depicted beneath the lattice.

y and t links. Within each of these blocks we 'unravel' the lattice into a 1D array by moving first along the x direction, then moving along y and finally along z. This is depicted in Fig 1 and realised by the equation displayed below.

$$i = \mu n/3 + x \,\%\, n_x + n_x \left(y \,\%\, n_y + n_y \left(t \,\%\, n_t\right)\right) \tag{1}$$

Where i is the 1D index, $\mu \in \{0, 1, 2\}$ is the direction of the link and (x, y, t) is the location of the link in the $n = n_x \times n_y \times n_t$ lattice. Note that $\%$ represents remainder division, in order to enforce the periodic boundary conditions.[1]

With this mapping in place it is possible to obtain the value of any link specified by (μ, x, y, t) and indeed the values of the links connected to it. The neighbouring links can be accessed by incrementing the coordinates along the appropriate directions and recalculating the index. With this in place we can define our lattice as an array of link objects that each point to a different lattice value. The utility of this class based approach will become clear as we consider the implementation of the metropolis update.

[1]Strictly speaking this equation would fail if directly implemented into c++ code as it doesn't properly handle $x, y, t \in \mathbb{Z}^-$. This can easily be done by instead using $(x+n_x)\%n_x$ to extend the range of validity down to $x = -n_x$

1.2 Metropolis

Building on the class model of Section 1.1 we can define an update method of the link class that takes a specified random phase θ_l' and attempts to update the link $U_l = \mathrm{e}^{i\theta_l}$ to this new phase. The random phase is chosen from a gaussianly distributed set of phases that is symmetric about zero and sufficiently large in order to satisfy reversibility and ergodicity. The variance of the distribution can be tuned to obtain around a 50% update acceptance rate, this corresponds to $\sigma \approx 0.7$.

The randomness required by the probabilistic nature of the Metropolis algorithm is provided several pseudo-random number generators. Specifically the *Mersenne Twister* algorithm, seeded with different clock values for each required random sequence. These random sequences are then mapped to different distributions, e.g, Gaussian or Uniform.

The uniform distribution is required for the main operation in the Metropolis update. This is the acceptance of a link update with probability

$$\tilde{P}(U_l \to U_l') = \begin{cases} \exp\left\{-\beta\left[S(U_l') - S(U_l)\right]\right\}, & S(U_l') > S(U_l) \\ 1, & S(U_l') < S(U_l). \end{cases} \tag{2}$$

Using the uniform random number $r \in [0, 1)$ we can implement this acceptance by simply checking for $r < \tilde{P}(U_l \to U_l')$. This occurs when the `link::update()` method is called on a link. This method requires the *change* in the action, which is where the 'staples' come in. The action is defined as

$$S(\{U_l\}) = \sum_{\{U_p\}} (1 - \mathrm{Re}\{U_p\}), \tag{3}$$

$$\text{where } U_p = \prod_{\{U_l\} \in U_p} U_l, \tag{4}$$

where $\{U_p\}$ is the set of all plaquette operators. Each U_p product is formed from an ordered traversal of the square on the lattice that define the plaquette, see Fig 2. As our gauge field is abelian there is no issue of product ordering, however the direction of traversal for each link is important. When a link is traversed in the negative direction the conjugate field is used in the product.

From this definition of the action we can see that if only one link is changed then the change in the action is localised to only those plaquettes

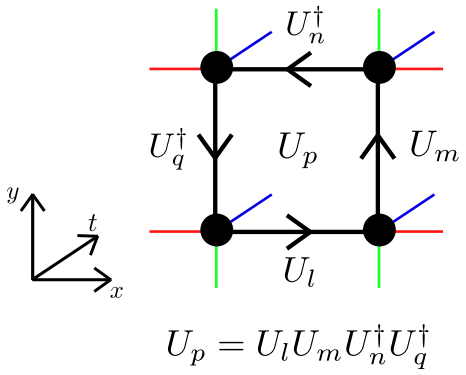


Figure 2: Pictorial representation of the plaquette operator U_p.

containing the changed link. Note also that the overall sense of plaquette traversal, i.e. clockwise or anticlockwise, is irrelevant as we take the real part. Thus it is convenient to define the changed link to be traversed in the forward direction, see Fig 3. This suggest the notion of staples $\{\xi_l\}$ belonging to each link that will be required by the link during a Metropolis update. Mathematically, from (3),

$$\Delta S = \sum_{\{\xi_i\}} \cos(\theta_l' + \xi_i) - \cos(\theta_l + \xi_i). \tag{5}$$

Where the staples $\{\xi_i\}$ are appropriately signed sums of staple phases as shown in Fig 3.

Returning to our notion of link objects we can see that although it is possible to calculate the index of all the staple links every time we update, this is very inefficient. By defining staples as class member objects that point to links and have the appropriate signs for calculating ξ_i already assigned we can economise our effort. By doing so the only operations needed to retrieve a staple value are a series of three dereferences followed by three additions.[2]

This class structure also provides a nice way of defining operations on the links as class methods. We also gain a degree of encapsulation, though this is not true encapsulation as multiple objects access the same links. To

[2]I don't believe that there is any way of further minimising the effort of getting the link values.

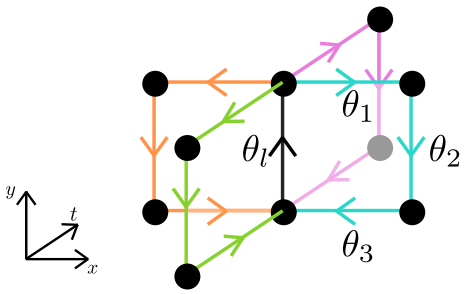


Figure 3: Pictorial representation of the concept of a staple. In each case the phase θ is the phase parameterising the U operator that lives on the link, $U = \mathrm{e}^{i\theta}$.

summarise, the diagram in Fig 4 displays a simplified pictorial representation of the class structure.

1.3 Masses

The smallest glueball masses in the theory are calculated using correlation functions $C(t)$ defined as

$$C(t) = \langle \Phi(n_t)^\dagger \Phi(0) \rangle \underset{n_t \to \infty}{\propto} \mathrm{e}^{-amn_t}. \tag{6}$$

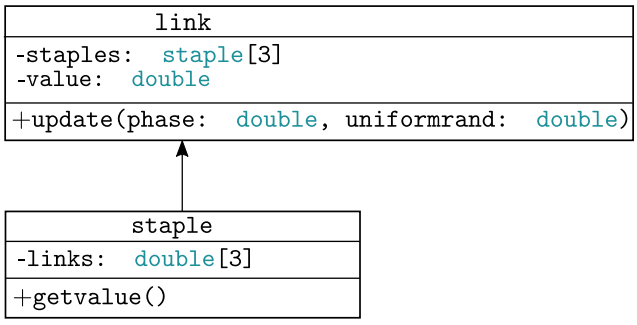


Figure 4: Simplified 'UML like' class diagram of the lattice and staple classes

Where the expectation is an average over field configurations generated using the Metropolis update method. The idea being to create a sequence of field configurations where each configuration differs by an update attempt on *all* links, the ordering is not important. Then measure $\Phi(n_t)^\dagger\Phi(0)$ for each configuration and average this quantity over the sequence to get $C(t)$.

The two masses are the $J^{PC} = 0^{--}$ and 0^{++} masses, $m_{0^{--}}$ and $m_{0^{++}}$. These are calculated from the operators

$$\Phi_{m_{0^{++}}}(n_t) = \sum_{\bar{n}} \left[\mathrm{Re}\{\tilde{U}_p(\bar{n}, n_t)\} - \langle \mathrm{Re}\{\tilde{U}_p\}\rangle \right] \tag{7}$$

$$\text{and } \Phi_{m_{0^{--}}}(n_t) = \sum_{\bar{n}} \mathrm{Im}\{\tilde{U}_p(\bar{n}, n_t)\}, \tag{8}$$

where $\bar{n} = (x, y)$ and the $\tilde{U}_p$ are plaquettes lying in the xy plane only. The sum over all spatial sites is akin to the zero momentum fourier mode and so these operators project onto states with zero momentum. The choice of real and imaginary parts is used to pick out the desired charge conjugation symmetry. The effect of charge conjugation on a continuum gauge field is $A^\mu \to -A^\mu$. Thus, via the definition of a Wilson loop operator as

$$\mathrm{Tr}\left(\mathcal{P}\exp\left(i \oint_\gamma A_\mu \mathrm{d}x^\mu\right)\right), \tag{9}$$

this corresponds to a reversal of the path direction in the lattice theory. Forming operators that project exclusively onto either the + or − symmetry sectors can thus be achieved by taking Re or Im of the plaquette, as depicted in Fig 5. The subtraction of $\langle \mathrm{Re}\{\tilde{U}_p\}\rangle$ in $m_{0^{++}}$ is a removal of the vacuum expectation value from the operator.

Following in the same vein as previously, we now consider how to abstract this operator into our object oriented picture. Each time we generate a field configuration the minimal amount of processing that must be done to measure $\Phi(n_t)^\dagger\Phi(0)$ for all t is to obtain the link values and calculate the plaquette sum. Thus the natural object is a `twall` or *T-wall* with class member `plaq` objects lying in a single constant t plane. The `plaq` objects each point to the relevant four links and possess a `plaq::getvalue()` method to return U_p. This is depicted in Fig 6 and Fig 7.

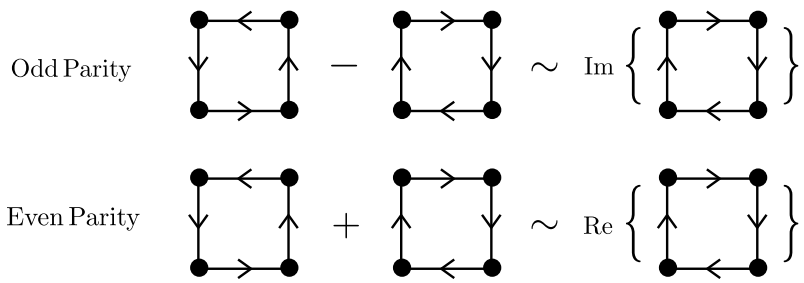


Figure 5: Forming operators that project onto separate symmetry sectors.

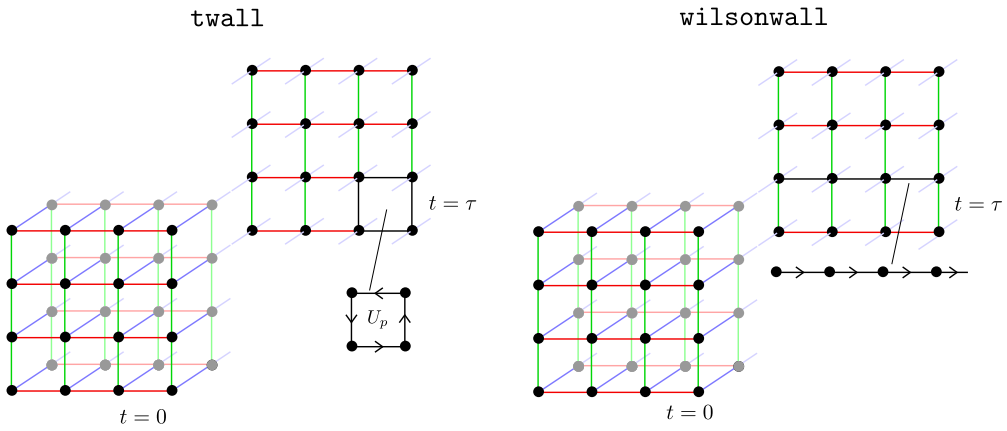


Figure 6: Pictorial representation of the 'wall' operators. Note that here the plaquettes formed at the edge of the lattice by the periodic boundary conditions are not shown.

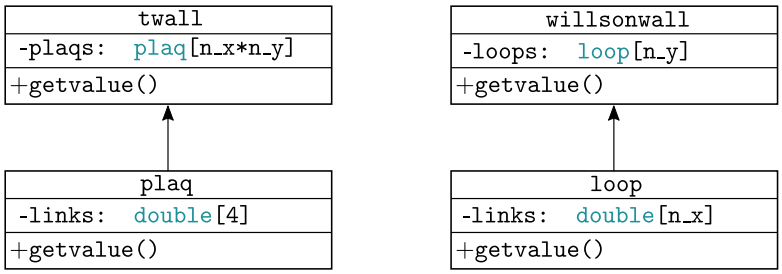


Figure 7: Simplified 'UML like' class diagram of the 'wall' classes

1.4 String Tension

The string tension is calculated similarly by considering infinite Wilson loop operators constructed from operators that wrap around the whole lattice,

$$\Phi_{Wilson}(n_t) = \sum_{n_y} \prod_{n_x=1}^{n_x=L_x} U_{\hat{x}}(n_x, n_y, n_t). \tag{10}$$

As previously, this suggests the notion of `wilsonwall` objects defined at a given t, containing L_y `loop` objects that each point to L_x links forming a loop. This is depicted in Fig 6 and Fig 7. The confining tension $a^2\sigma$ is calculated using the 'Nambu-Goto' formula

$$E_f(l = aL_x) = \sigma l \left(1 - \frac{\pi}{3\sigma l^2}\right)^{\frac{1}{2}}. \tag{11}$$

Which in the limit of large l becomes

$$E_f(l) \underset{l\to\infty}{=} \sigma l - \frac{\pi}{6l} + O\left(\frac{1}{l^3}\right). \tag{12}$$

1.5 Code Overview

Now that we have our principal objects of the algorithm it remains only to describe how they are pieced together. `main()` begins by setting up the relevant files for output and timers to record runtime. After this, the lattice and operator objects can be initialized and instantiated to their appropriate values. This involves using member routines to calculate where the appropriate links are and set pointers to these. For example, perform the calculation of which staples are connected to what link and where in memory those link values of the staples are actually stored.

After this setup we run $N_{equilib} = 20000$ Metropolis sweeps through the lattice in order to reach an equilibrium configuration. Whilst doing so we monitor the acceptance rate of link updates to ensure we are at around 50%. Additionally we can measure the runtime and make extrapolations about the completion time. After equilibrating we can begin to make measurements, which is most easily explained in the following pseudocode. For brevity we only show the measurement of one correlator.

```
for seq_no in range(N_sequences):
    correlatoravg = [0]*n_t
    acceptances = 0
    for fieldconfig in range(N_configs):
        for link in range(N_lattice):
            acceptances += lattice[link].update()
        for T in range(n_t):
            phi_T = twallarray[T].getvalue()
            for t in range(n_t):
                phi_tplusT = twallarray[(t+T)%n_t].getvalue()
                correlatoravg[t] += phi_tplusT*phi_T
        correlatoravg = correlatoravg/N_mes
        print('seq'+seq_n+'='correlatoravg)
```

This code also shows how we may use the imposed periodicity of our lattice to improve the quality of our data. We do so by measuring $\Phi(t+T)\Phi(T)$ and sweeping over T. A great deal is absent from the above code but it gives a good idea of the skeleton of the algorithm. We have not mentioned an 'average plaquette' class. This is because the ability to obtain plaquette values is built into the link and staple classes.

There are a few final points that deserve mention, particularly concerning runtimes and possible speed ups. I believe the class based structure gives us a good base of speed to work from. However, I do not know this for certain. I also think a lot of it will come down to compiler implementation, which is a bit too low level for a virginal c++ programmer. Despite this, one obvious significant speedup is that of parallelisation, which for this algorithm is trivial to implement. One can simply run multiple instances of the code, started at slightly staggered times so as to obtain different clock times for the random seed. In this way we are simply multi-processing and there is no complication of thread-safety, as with multi-threading. Upon completion is then the small matter of amalgamating the data outputs. Table 1 displays the average runtimes for all lattice sizes.

Putting this all together means that the total runtime for all $\beta = 2.0, 2.1, 2.2, 2.3$ on all lattices falls in at just less than a day of computing time. Which I'm actually rather pleased with!

Lattice Size	$18 \times 18 \times 24$	$22 \times 22 \times 36$	$28 \times 28 \times 40$
Runtime (hrs:mins)	0:52	1:33	2:57

Table 1: Total runtimes for calculations at a *single* β parallelised into 5 processes. Each process outputs 6 subsequences of 10000 field configurations each, giving us 30 sequences in total. This includes Metropolis update time and time taken to make measurements.

2 Results

At this point our simulations have produced files containing the four desired quantities for different values of β on different lattice sizes. As stated previously, the field configurations used were split at runtime into $n = 30$ subsequences so as to obtain error estimates for the various values. We begin by describing the process of calculating these error estimates, all of which was implemented in python because it's much easier than c++!

2.1 Errors[3]

Each subsequence gives us a measurement 'x_i' for each of our 'quantities', for example $C_{m_{0^{++}}}(t = 4)$ on $28 \times 28 \times 40$ at $\beta = 2.2$. Thus we can use unbiased estimators to form data points with errors

$$\bar{x} = \frac{1}{n} \sum_i x_i, \tag{13}$$

$$\bar{\sigma}_x^2 = \frac{1}{n-1} \sum_i (x_i - \bar{x})^2. \tag{14}$$

We do not know the underlying distribution's standard deviation, in fact we don't know its distribution at all! However, given that the x_i measurements themselves are in fact an arithmetic mean over $N = 10000$ field configurations we may use the central limit theorem to say that the x_i are in fact Gaussianly distributed. There is a point of contention here about the fact that the central limit theorem requires the sum of *independent* random variables. Strictly speaking the field configurations are certainly not independent as they are generated from each other in sequence. Thus we would fully expect there to

[3]My understanding of statistics is perhaps not as good as it ought to be and as you will see I have a couple of cautious concerns about what I've done in this section.

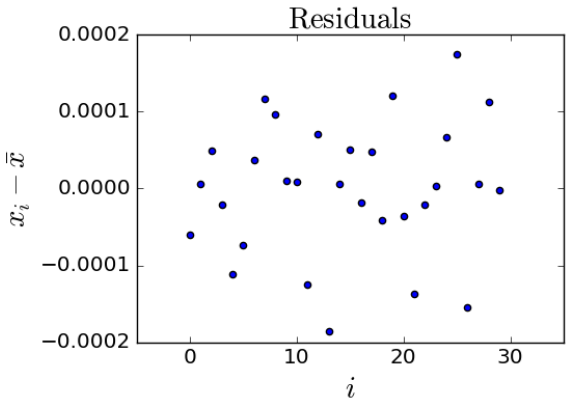


Figure 8:

be some correlation, but I imagine it is probably not so relevant given the sufficiently large n and perhaps some appeal to ergodicity. Moving on...

When sampling from a Gaussian distribution with unknown standard deviation we must use the t-distribution for our standard errors.

$$x = \bar{x} \pm t_{n,\alpha} \frac{\sigma_x}{\sqrt{n}} \tag{15}$$

Where this specifies the error for a $(1-\alpha) \times 100\%$ confidence interval, in this case I chose the seemingly standard 95% interval. The value of $t_{n,\alpha}$ is chosen from t-distribution tables. Before we completely discard all information about our n field configurations it is good to first check that our assumption of no correlations between simulations is indeed valid and that our residuals are random. The plot in Fig 8 suggests we are correct in our assumption of no inter-sequence correlation. This obviously doesn't violate the presence of intra-sequence correlation.

2.2 Fitting

Now that we have standard errors on all our measurements it is possible to fit our results. Obviously this is only necessary for our correlators $C(t)$ satisfying $C(t) \underset{t\to\infty}{=} c\mathrm{e}^{-aEn_t}$ and not the average plaquette.

All fitting and further analysis was performed in OriginPro. We expect to see some discrepancy between the data and the fits for various reasons, including finite size effects and inherent fluctuations from the Monte Carlo

approach. Figure 9 displays the fitting process for the $m_{0^{--}}$ and $m_{0^{++}}$ masses. In each case the decision was made to fit the curves to either

$$y = a\mathrm{e}^{bx}, \tag{16}$$

$$\text{or } y = y_0 + A\mathrm{e}^{R_0 x}. \tag{17}$$

The offset was chosen as some of the smaller lattices especially displayed an offset that I assume is due to the aforementioned finite size. This is the reason for not just taking the logarithm and fitting that linearly.

As a result of the imposed periodic boundary conditions it is only valid to fit to the first half of the lattice.[4]

Figure 9 shows exceptionally tight error bars for the $m_{0^{++}}$ and the same is true of the Wilson loops. This is present in all lattice sizes at all β which is somewhat concerning, but the fits seem good? Once again, moving on...

After applying this fitting process to all simulations we obtain the value of 'E' for each of our correlators. In the case of the masses this is exactly what was required to be calculated and the results are listed in Table 2. The tables list the value of $m_{0^{--}}$, $m_{0^{++}}$, and the average plaquette as this has been available for a while. The errors listed here are the 95% confidence interval bounds for the fitting parameters, determined by Origin. This calculation takes into account the errors on the individual points. The fitting routine itself also makes use of the error bars, using them inversely as weights in the fitting.

2.3 Further Analysis

Although the values of the masses are obtainable directly from the exponent we must use the aforementioned Nambu-Goto formula (12) to obtain the string tension. This is done very similarly to the previous fitting, only this time linear. Figure 10 displays the fitting process and Table 3 displayed the results and their errors. Again these errors are the 95% confidence interval bounds for the fit parameters.

Though technically we are now done we conclude by taking a look at the beta dependence of the masses in Fig 11.

[4] I suppose you could fit both halves separately and then combine them to decrease the error but as the correlators are symmetric in I wondered if this would constitute an over fitting of the data.

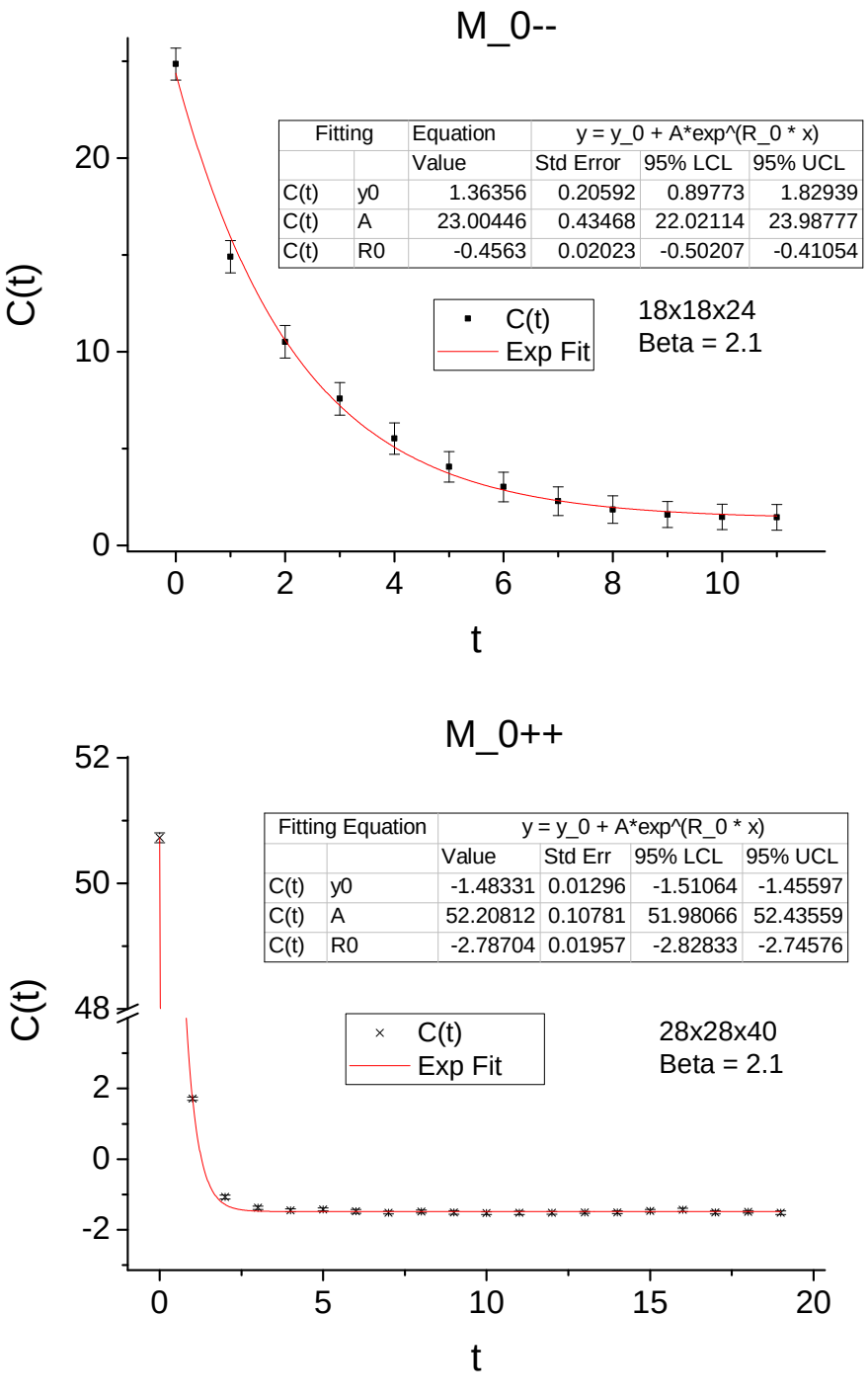


Figure 9: Masses exponential fitting.

$m_{0^{++}}$	$18 \times 18 \times 24$	$22 \times 22 \times 36$	$28 \times 28 \times 40$
2.0	2.680 ± 0.085	2.690 ± 0.112	2.666 ± 0.050
2.1	2.790 ± 0.061	2.785 ± 0.045	2.787 ± 0.041
2.2	2.880 ± 0.079	2.874 ± 0.053	2.875 ± 0.045
2.3	2.978 ± 0.050	2.974 ± 0.060	2.976 ± 0.029

$m_{0^{--}}$	$18 \times 18 \times 24$	$22 \times 22 \times 36$	$28 \times 28 \times 40$
2.0	0.546 ± 0.040	0.448 ± 0.038	0.550 ± 0.025
2.1	0.456 ± 0.046	0.385 ± 0.032	0.378 ± 0.026
2.2	0.392 ± 0.051	0.289 ± 0.029	0.0326 ± 0.026
2.3	0.352 ± 0.056	0.272 ± 0.031	0.267 ± 0.023

Avg. Plaq.	$18 \times 18 \times 24$	$22 \times 22 \times 36$	$28 \times 28 \times 40$
2.0	0.80594 ± 7.3e-5	0.81256 ± 0.005	0.80599 ± 3.5e-5
2.1	0.81878 ± 5.8e-5	0.81876 ± 4.3e-5	0.81875 ± 3.9e-5
2.2	0.82955 ± 6.1e-5	0.82949 ± 4.7e-5	0.82948 ± 3.5e-5
2.3	0.83873 ± 6.8e-5	0.83868 ± 3.7e-5	0.83850 ± 3.2e-5

Table 2: Mass data from fits

β	String Tension
2.0	0.191 ± 0.041
2.1	0.183 ± 0.058
2.2	0.176 ± 0.022
2.3	0.168 ± 0.002

Table 3: String tensions for different β.

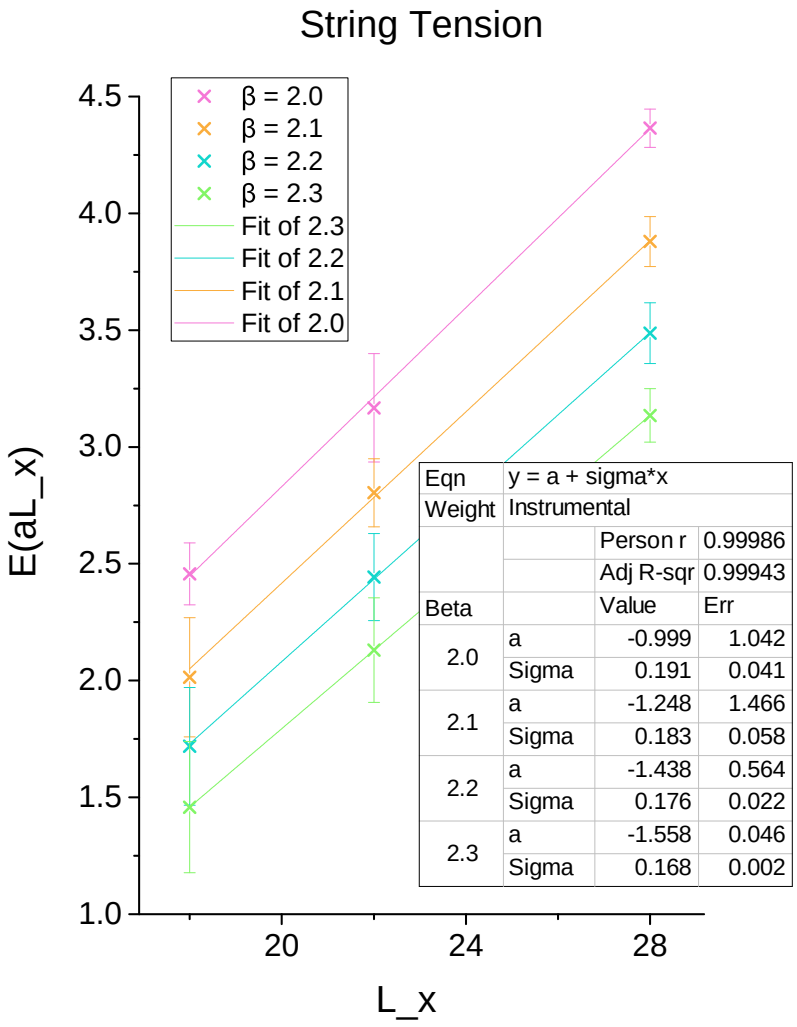
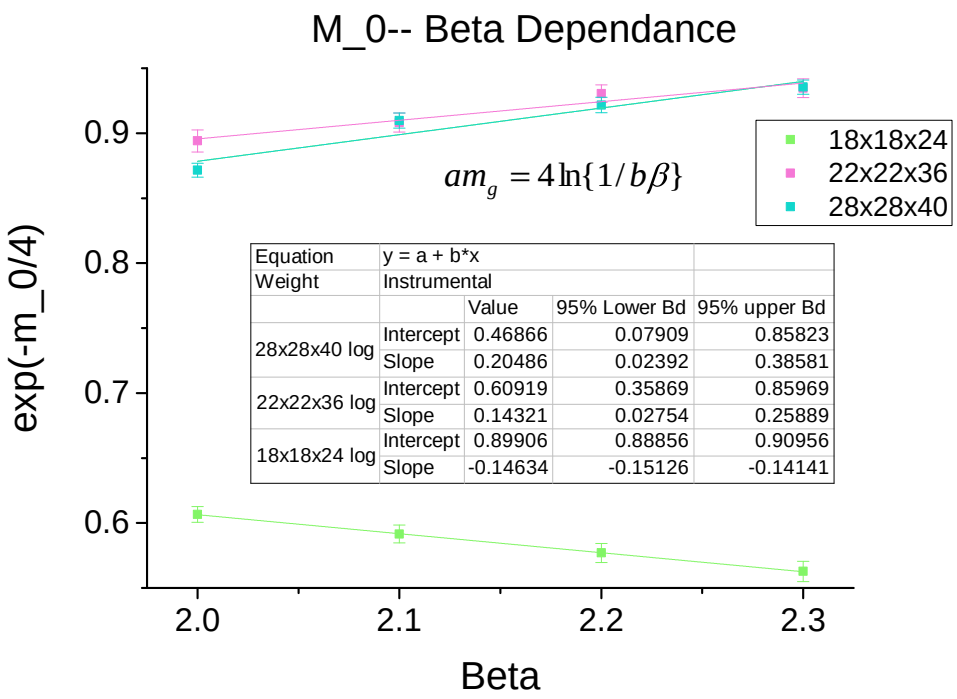


Figure 10: Wilson loop correlator data used to calculate the string tension σa^2 via the Nambu-Goto formula (12)



Equation	y = a + b*x			
Weight	Instrumental			
		Value	95% Lower Bd	95% upper Bd
28x28x40 log	Intercept	0.46866	0.07909	0.85823
	Slope	0.20486	0.02392	0.38581
22x22x36 log	Intercept	0.60919	0.35869	0.85969
	Slope	0.14321	0.02754	0.25889
18x18x24 log	Intercept	0.89906	0.88856	0.90956
	Slope	-0.14634	-0.15126	-0.14141

Figure 11: β dependences of the masses for different lattice sizes. Masses are first exponentiated so that we can linear fit to $am_g = 4\ln(1/b\beta)$. The fits are nicely linear but a zero intercept falls just outside the 95% confidence limits, especially for the smaller lattices.