STATIC-QUARK POTENTIAL CALCULATION

Giles Strong

IST QCD Course by Pedro Bicudo & Nuno Cardoso

Summary presentation - 06/12/18

PROBLEM DESCRIPTION

OVERVIEW

- The aim is to compute the value of the static quark potential at any distance
- Can discretise the gluon fields as link variables between fixed points in space-time (a lattice)
- The potential between two quarks is then the trace of the product of a closed loop of link variables
- The potential should be gauge invariant, therefore the loop can be computed across the lattice and averaged
- Loops of different sizes may be computed to get the potential at different distances, and then a fit can be performed

EXPERIMENT & CODE

https://github.com/GilesStrong/LatticeQCD_IST2018/tree/1.0

CONFIGURATION READING

- XTensor library used for lattice configuration storage and tensor mathematics
- Lattice stored as 4-dimensional array
  - Each dimension corresponds to a coordinate on the lattice (3 spatial, 1 temporal)
  - Each element (a vertex) consists of an array of four SU(3) matrices corresponding to links between lattice vertices in each direction
  - Each SU(3) matrix contains nine complex numbers
- Configurations read in from binary files by initialising empty lattice and looping over elements and reading in values from the binary file
- Each "SU(3)" matrix is tested to ensure it really is unitary, with a tolerance of $1e^{-10}$

```cpp
void Lattice::readConfig(std::string configName) {
    /*Read inconfiguration from file*/

    if (_verbose == "load") std::cout << "Reading configuration from: " << configName << "\n";
    std::ifstream filein(configName.c_str(), std::ios::in | std::ios::binary);

    double real, imaginary;
    std::complex<double> det;

    //Lattice iteration
    for (size_t t = 0; t < _shape[3]; t++) { //Loop over t
        for (size_t z = 0; z < _shape[2]; z++) { //Loop over z
            for (size_t y = 0; y < _shape[1];y++) { //Loop over y
                for (size_t x = 0; x < _shape[0]; x++) { //Loop over x

                    direction tmp_direction;
                    //Directional SU(3) iteration
                    for (size_t d = 0; d < 4; d++) { //Loop through SU(3) matrices
                        if (_verbose == "load") std::cout << "\nSU(3) matrix at lattice point (" << x << ", " << y << ", " << z << ", " << t << ") in " << Lattice::getDim(d) << " direction:\n";

                        su3Matrix tmp_su3Matrix;
                        //Elements of SU(3) matrix
                        for (size_t a = 0; a < 3; a++) { //Loop through columns of SU(3) matrix
                            for (size_t b = 0; b < 3; b++) { //Loop through rows of SU(3) matrix
                                filein.read((char*)&real, 8);
                                filein.read((char*)&imaginary, 8);
                                if (_verbose == "load") std::cout << "(" << a << ", " << b << "): " << real << " + " << imaginary << "*i\n";
                                tmp_su3Matrix(a, b) = std::complex<double>{real, imaginary};
                            }
                        } //Elements

                        det = xt::linalg::det(tmp_su3Matrix);
                        if (_verbose == "load") std::cout << "Det = " << det << "\n";
                        if (!doubleCompare(det.real(), 1.0).first | !doubleCompare(det.imag()+1, 1.0).first) {
                            std::cout << "Matrix at (" << x << ", " << y << ", " << z << ", " << t << ") in " << Lattice::getDim(d) << " direction is not unitary\n";
                            std::cout << "Relative distances are: " << doubleCompare(det.real(), 1.0).second << " and " << doubleCompare(det.imag()+1, 1.0).second << "\n";
                            throw std::runtime_error("Non-unitary matrix");
                        }

                        tmp_direction(d) = tmp_su3Matrix;
                        if (_debug == "load") throw std::runtime_error("Debug mode: Only print one SU(3) matrix");

                    } //SU(3) matrices
                    _config(x,y,z,t) = tmp_direction;
                }
            }
        }
    } //Lattice points

    filein.close();
}
```

Loop through all lattice points

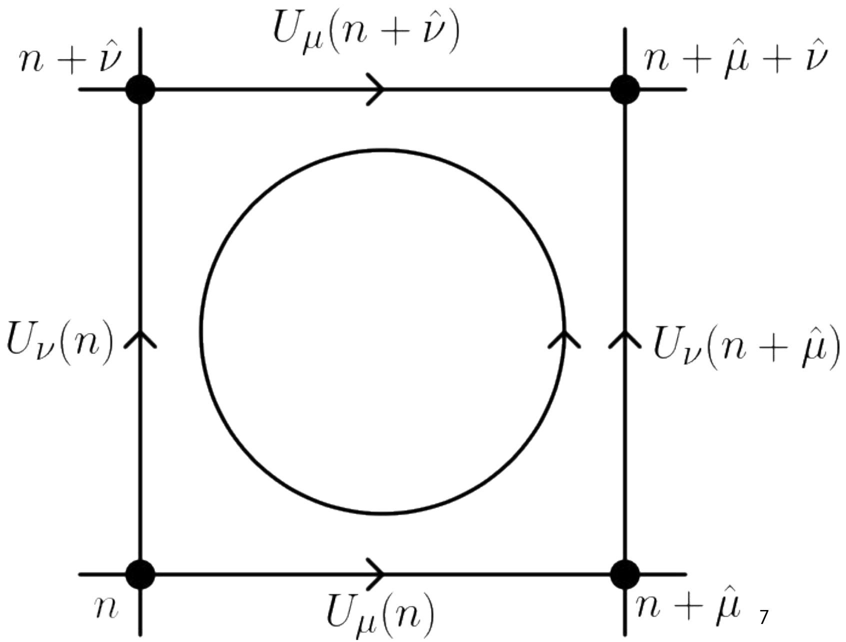
Read in vertex

Read in SU(3) matrix

Ensure unitarity

PLAQUETTE CALCULATION

- Simplest loop to calculate
- Move in 2-D plane of lattice along loop of size 1 lattice spacing
- Take dot product of link matrices
  - Take complex conjugate for opposite direction
- Value of the plaquette is the real part of the trace of the dot product divided by 3 (3 colours)



```cpp
size_t modulo(int a, int b) {
    /*Modulo that also works with negaive numbers*/
    return (a%b+b)%b;
}

std::array<size_t, 4> Lattice::movePoint(std::array<size_t, 4> point, size_t direction, int amount) {
    /*Move position in grid whilst respecting peiodic boundaries*/
    if (_verbose == "movePoint") {
        std::cout << "\nCurrect point: " << point[direction] << " and moving " << amount << " steps\n";
        std::cout << "Candidate point is: " << static_cast<int>(point[direction])+amount << "\n";
        std::cout << "Periodic boundary conditions means new point is: " << modulo(static_cast<int>(point[direction])+amount, _shape[direction]) << "\n";
    }
    point[direction] = modulo(static_cast<int>(point[direction])+amount, _shape[direction]);
    return point;
}
```

Enforce periodic boundary of lattice

```cpp
std::complex<double> Lattice::calcPlaquette(std::array<size_t, 4> point, std::pair<size_t, size_t> plane) {
    /*Calculate value of plaquette at specified starting gridpoint and 2D plane*/
    if (_verbose == "calcPlaquette") std::cout << "\nCalculating plaquette at (" << point[0] << "," << point[1] << "," << point[2] << "," <<

    //Link in mu direction at point
    su3Matrix u = _config[point][plane.first];
    if (_verbose == "calcPlaquette") std::cout << "U matrix:\n" << u << "\n At point: (" << point[0] << "," << point[1] << "," << point[2] <

    //Link in nu direction at point+mu
    std::array<size_t, 4> tmp_point = movePoint(point, plane.first, 1);
    su3Matrix v = _config[tmp_point][plane.second];
    if (_verbose == "calcPlaquette") {
        std::cout << "V matrix:\n" << v << "\n At point: (" << tmp_point[0] << "," << tmp_point[1] << "," << tmp_point[2] << "," << tmp_poir
    }

    //Reverse of link in mu direction at point+nu
    tmp_point = movePoint(point, plane.second, 1);
    su3Matrix uprime = xt::conj(xt::transpose(_config[tmp_point][plane.first]));

    if (_verbose == "calcPlaquette") {
        std::cout << "U prime matrix:\n" << _config[tmp_point][plane.first] << "\nconjugate transpose:\n" << uprime << "\n At point: (" << t
    }

    //Reverse of link in nu direction at point
    su3Matrix vprime = xt::conj(xt::transpose(_config[point][plane.second]));
    if (_verbose == "calcPlaquette") std::cout << "V prime matrix:\n" << _config[point][plane.second] << "\nconjugate transpose:\n" << vprim

    //Compute plaquette

    su3Matrix product = xt::linalg::dot(u, xt::linalg::dot(v, xt::linalg::dot(uprime, vprime)));
    if (_verbose == "calcPlaquette") std::cout << "Plaquette product:\n" << product << "\n";
    std::complex<double> trace = xt::sum(xt::diagonal(product))[0];
    if (_verbose == "calcPlaquette") std::cout << "Plaquette trace: " << trace << "\n\n";

    if (_debug == "calcPlaquette") throw std::runtime_error("Debug mode: Only try one product");

    return trace/3.;
}
```

Extract link matrices from starting point

Move 1 unit in plane
Get next link

Moving opposite to link direction;
Take conjugate transpose

Compute production of links
Take trace
Average over number of colours

Return both real & imaginary parts

WILSON-LOOP CALCULATION

- Simple extension to plaquette calculation
- Compute loops of arbitrary size (spatial & temporal)
- But, one dimension of the loop plane must be time

```cpp
std::complex<double> Lattice::calcWilsonLoop(std::array<size_t, 4> point, size_t spatialDimension, size_t R, size_t T) {
    /*Compute latice loop starting at given point with spatial width r and temporal width t in given spatial direction*/
    if (_verbose == "calcWilsonLoop") std::cout << "\nCalculating Wilson loop at (" << point[0] << "," << point[1] << "," << point[2] << "," << point[3] << ") in " << ge

    su3Matrix product({{1,0,0},{0,1,0},{0,0,1}});
    su3Matrix tmp;

    //Reverse link in temporal direction
    for (size_t i = 0; i < T; i++) {
        tmp = xt::conj(xt::transpose(_config[point][3]));
        product = xt::linalg::dot(tmp, product);
        if (_verbose == "calcWilsonLoop") std::cout << "Reverse temporal link " << i << " at point: (" << point[0] << "," << point[1] << "," << point[2] << "," << point[
        point = movePoint(point, 3, 1);
    }

    //Reverse link in spatial direction
    for (size_t i = 0; i < R; i++) {
        tmp = xt::conj(xt::transpose(_config[point][spatialDimension]));
        product = xt::linalg::dot(tmp, product);
        if (_verbose == "calcWilsonLoop") std::cout << "Reverse spatial link " << i << " at point: (" << point[0] << "," << point[1] << "," << point[2] << "," << point[3
        point = movePoint(point, spatialDimension, 1);
    }

    //Link in temporal direction
    for (size_t i = 0; i < T; i++) {
        point = movePoint(point, 3, -1);
        product = xt::linalg::dot(_config[point][3], product);
        if (_verbose == "calcWilsonLoop") std::cout << "Temporal link " << i << " at point: (" << point[0] << "," << point[1] << "," << point[2] << "," << point[3] << ")
    }

    //Link in spatial direction
    for (size_t i = 0; i < R; i++) {
        point = movePoint(point, spatialDimension, -1);
        product = xt::linalg::dot(_config[point][spatialDimension], product);
        if (_verbose == "calcWilsonLoop") std::cout << "Spatial link " << i << " at point: (" << point[0] << "," << point[1] << "," << point[2] << "," << point[3] << ")
    }

    //Compute trace
    if (_verbose == "calcWilsonLoop") std::cout << "Wilson loop product:\n" << product << "\n";
    std::complex<double> trace = xt::sum(xt::diagonal(product))[0];
    if (_verbose == "calcWilsonLoop") std::cout << "Wilson loop trace: " << trace << "\n\n";

    if (_debug == "calcWilsonLoop") throw std::runtime_error("Debug mode: Only try one product");

    return trace/3.;
}
```

Extract SU(3) links as before

But keep running product of matrices; must work backwards around loop

Product already computed

Take trace

Average over number of colours

Return both real & imaginary parts

WILSON-LOOPS OVER LATTICE

- Previous function computed Wilson loop of:
  - Arbitrary spatial and temporal size
  - Arbitrary spatial direction
  - Arbitrary position in lattice
- Now want to compute all Wilson loops of:
  - Specified spatial and temporal size
  - In all spatial directions
  - At all points in the lattice
- Only interested in the real part of the mean of their traces

```cpp
double Lattice::calcOverallMeanWilsonLoopMP(size_t R, size_t T) {
    /*Calculate mean of all Wilson loops of spatial width r and temporal width t across entire lattice with multi processing*/
    double sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (size_t t = 0; t < _shape[3]; t++) { //Loop over t
        for (size_t z = 0; z < _shape[2]; z++) { //Loop over z
            for (size_t y = 0; y < _shape[1]; y++) { //Loop over y
                for (size_t x = 0; x < _shape[0]; x++) { //Loop over x
                    for (size_t i = 0; i < 3; i++) { //Direction iteration
                        sum += calcWilsonLoop({x,y,z,t}, i, R, T).real();
                    }
                }
            }
        }
    }

    return sum/(_shape[0]*_shape[1]*_shape[2]*_shape[3]*3);
}
```

Parallelised loop across lattice and directions
Accumulate sum of real parts of traces

Divide by number of vertices and directions

VARIABLE SIZE WILSON-LOOPS

- Can now get mean of specific size Wilson loop across entire lattice
- Inter-quark potential requires loops of various sizes
- Extend loop to compute mean Wilson loop for a range of lengths and widths and record results

```cpp
void runWilsonExperimentMP(Lattice* config, std::string name) {
    /*Loop over range of R and T values and compute mean of corresponding Wilson loops*/
    std::ofstream outFile;
    outFile.precision(50);
    outFile.open(name);
    outFile << "R,T,Mean\n";

    double mean;
    for (size_t R = 1; R <= config->getShape()[0]/2; R++) {
        for (size_t T = 1; T <= config->getShape()[3]/4; T++) {
            if (verbose != "") std::cout << "(R, T) = " << R << ", " << T << ", mean = ";
            mean = config->calcOverallMeanWilsonLoopMP(R, T);
            outFile << R << "," << T << "," << mean << "\n";
            if (verbose != "") std::cout << mean << "\n";
        }
    }

    outFile.close();
}
```

Loop over range of sizes of Wilson loop

Compute mean Wilson loop and write to file

MANY CONFIGURATIONS

- Now have mean Wilson loops for a range of spatial and temporal sizes
- But, this is only for one possible lattice configuration
- O(1000) different configurations were provided
- Use batch system to analyse all configurations simultaneously

```python
def make_job_file(uid, input_file, output_dir):
    """Build and submit analysis job."""
    output_file = output_dir + str(uid) + '.csv'

    cmd = "./bin/main.exe "
    cmd += "-i " + input_file
    cmd += " -o " + output_file

    job_name = "analysis_" + str(uid) + ".job"
    job_file = open(job_name, "w")
    job_file.write("echo Beginning\ job\n")
    job_file.write("module load gcc-5.4\n")
    job_file.write("export PATH=/lstore/cms/giles/programs/bin:$PATH\n")
    job_file.write("export LD_LIBRARY_PATH=/lstore/cms/giles/programs/lib64\n")
    job_file.write("cd " + SOFTDIR + "\n")
    job_file.write("echo Paths\ set\n")
    job_file.write(cmd + "\n")
    job_file.close()

    sub = "qsub " + job_name
    print("Submitting: " + sub)
    os.system(sub)


if __name__ == "__main__":
    parser = optparse.OptionParser(usage=__doc__)
    parser.add_option("-i", "--input_dir", dest="input_dir",
                      default="/lstore/cms/giles/configs/confs_b6.2_bin/",
                      action="store", help="Directory of configs")
    parser.add_option("-n", "--N", dest="n", action="store", default=-1,
                      help="Number of files to run")
    parser.add_option("-o", "--output_dir", dest="output_dir", action="store",
                      default='Output/', help="Output directory")
    opts, args = parser.parse_args()

    samples = glob.glob(opts.input_dir + '*.bin')
    print('Running over {} of {} samples found'.format(opts.n, len(samples)))
    if opts.n > 0:
        samples = samples[0:int(opts.n)]

    for i, sample in enumerate(samples):
        make_job_file(i, sample, opts.output_dir)
```

Save job instructions
Submit job to batch queue

Loop over each sample

RESULTS & ANALYSIS

https://github.com/GilesStrong/LatticeQCD_IST2018/blob/1.0/Analysis/Config_Analysis_Final.ipynb

POTENTIAL AT SET DISTANCE

- Can compute the field strength V at a set distance R by computing
  - $V(R) = \ln\left(\frac{W(R)_t}{W(R)_{t+1}}\right)$
  - Where $W(R)_t$ is the value of a Wilson loop of spatial size R and temporal size t
- This should decrease with t and eventually plateau
- $V(R)$ is the value at the plateau

JACKKNIFE RESAMPLING

- Could simply average *V(R)* over all results for lattice configurations
  - But uncertainties are likely to still be too large
- Instead, can use jackknife (leave one out) resampling to compute the uncertainty

```
def jackknife(in_vals):
    vals_sum = np.sum(in_vals)
    if vals_sum != vals_sum: #Contains NaNs
        vals = [i for i in in_vals if i == i]
        vals_sum = np.sum(vals)

    else:
        vals = in_vals

    jk = np.zeros_like(vals)
    n = len(vals)

    for i in range(n):
        jk[i] = (vals_sum-vals[i])
    jk /= n-1

    mean = np.mean(jk)
    std = np.sqrt((len(jk)-1)*np.sum((jk-mean)**2)/len(jk))
    return mean, std
```

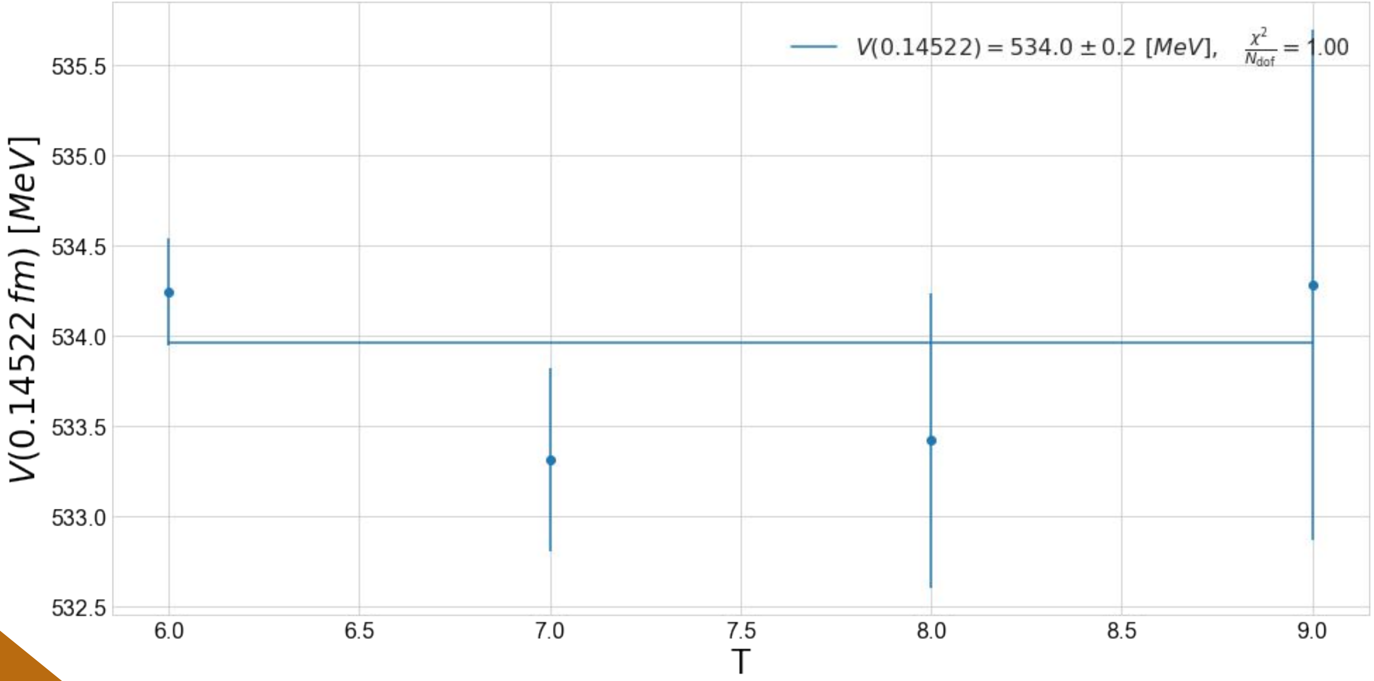
Create resampled data

Compute mean & standard deviation of resampled data

FIELD-STRENGTH FITTING

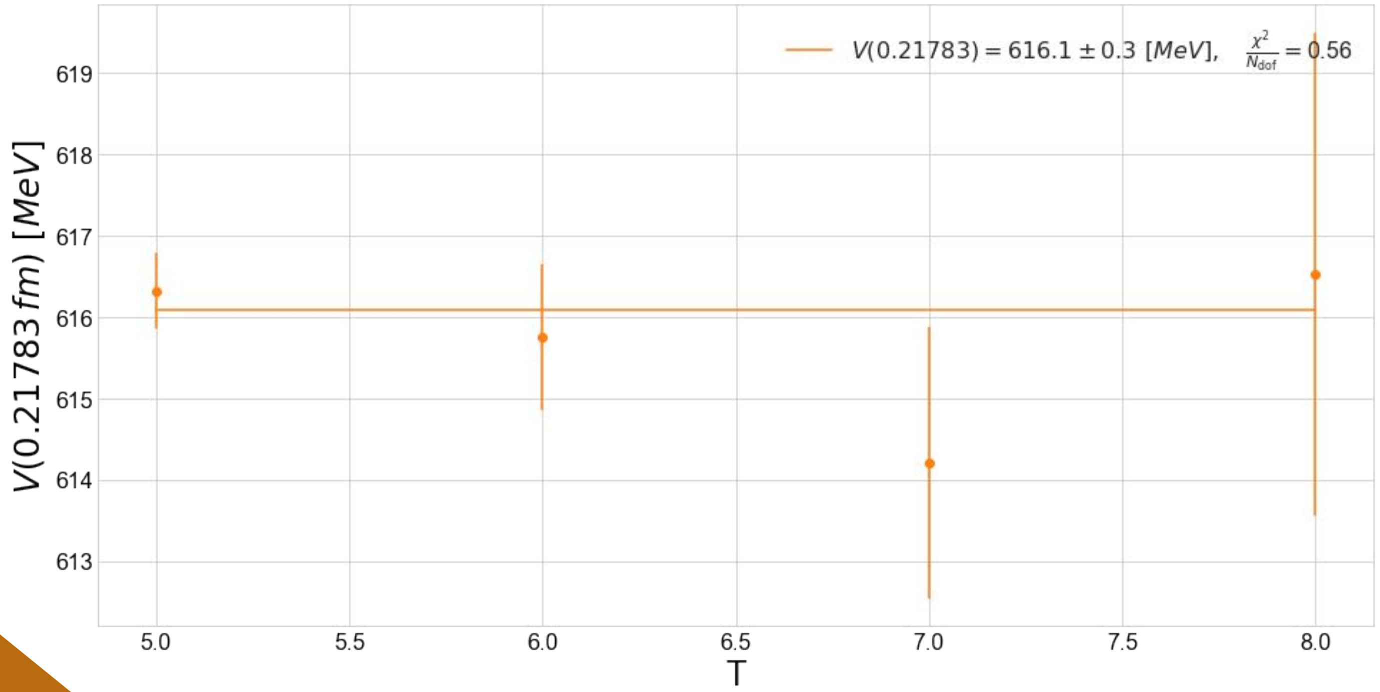
- Can now fit to plateaus, but results still display fluctuations
  - Must restrict considered points to improve fit; use reduced χ^2 as a guide
- Only four R values are able to be considered for fitting
- Substitute in lattice spacing parameter: $a = 0.07261\ fm$

$V(0.14522\ fm)$



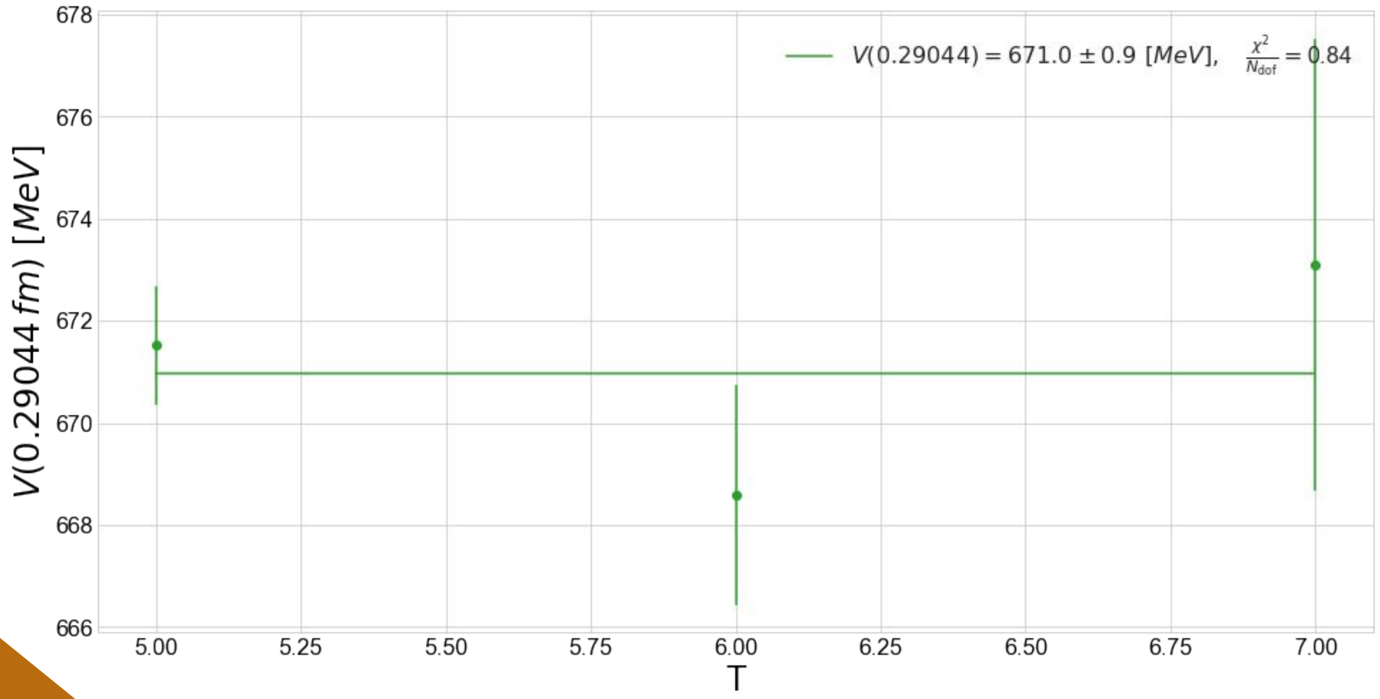
$V(0.14522) = 534.0 \pm 0.2$ $[MeV]$, $\frac{\chi^2}{N_{\mathrm{dof}}} = 1.00$
535.5
535.0
534.5
534.0
533.5
533.0
532.5
$V(0.14522\ fm)$ $[MeV]$
6.0
6.5
7.0
7.5
8.0
8.5
9.0
T

V(0.21783 fm)



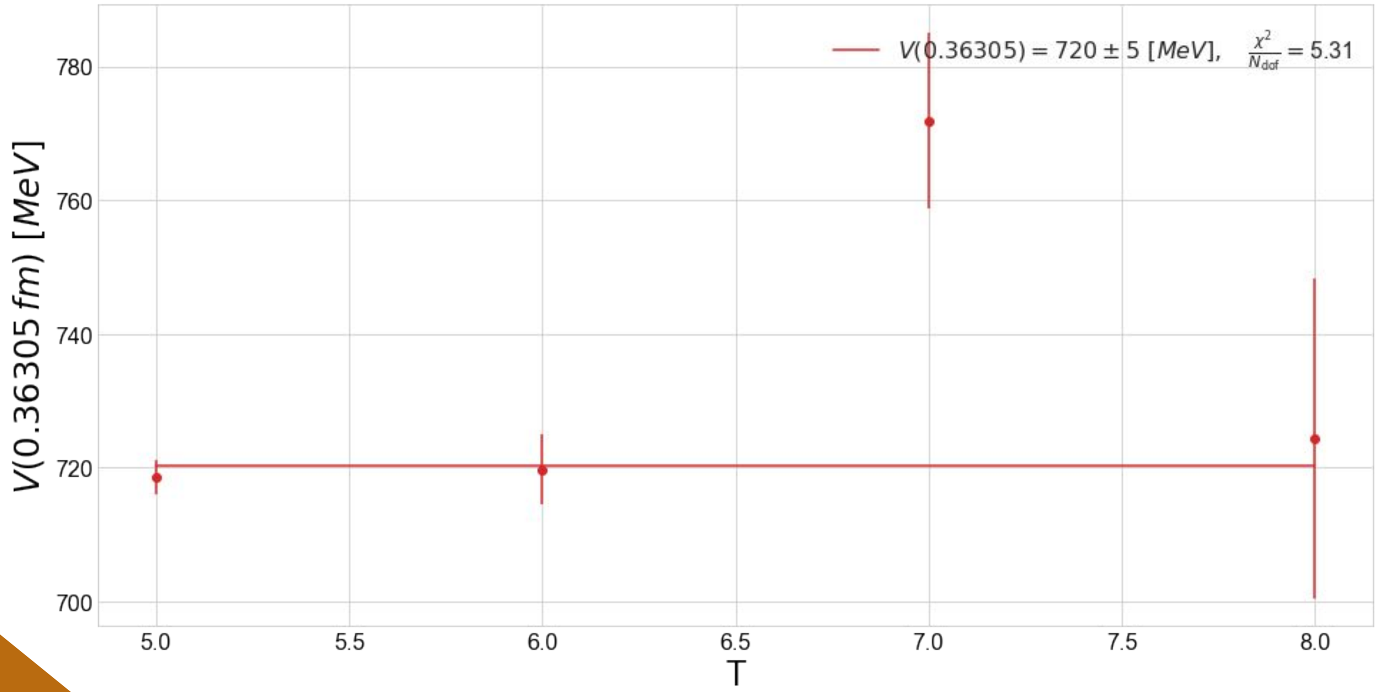
$V(0.21783) = 616.1 \pm 0.3$ $[MeV]$, $\frac{\chi^2}{N_{dof}} = 0.56$
619
618
617
616
615
614
613
$V(0.21783\,fm)$ $[MeV]$
5.0
5.5
6.0
6.5
7.0
7.5
8.0
T

V(0.29044 fm)



$V(0.29044) = 671.0 \pm 0.9$ $[MeV]$, $\frac{\chi^2}{N_{dof}} = 0.84$
678
676
674
672
670
668
666
$V(0.29044\, fm)$ $[MeV]$
5.00
5.25
5.50
5.75
6.00
6.25
6.50
6.75
7.00
T

$V(0.36305\ fm)$



$V(0.36305) = 720 \pm 5\ [MeV]$, $\frac{\chi^2}{N_{\rm dof}} = 5.31$
780
760
740
720
700
$V(0.36305\ fm)\ [MeV]$
5.0
5.5
6.0
6.5
7.0
7.5
8.0
T

STATIC-QUARK POTENTIAL FIT

- We now know the inter-quark potential at 4 set separation distances
- Can fit to these points in order to be able calculate potential at any distance
- Following Gattringer & Lang, the static potential should be of the form:
  - $V(r) = \frac{A}{r} + B + \sigma r$
  - Where σ is referred to as the *string tension*
  - Expected to have a value of 900 MeV/fm

STATIC-QUARK POTENTIAL FIT