Introduction

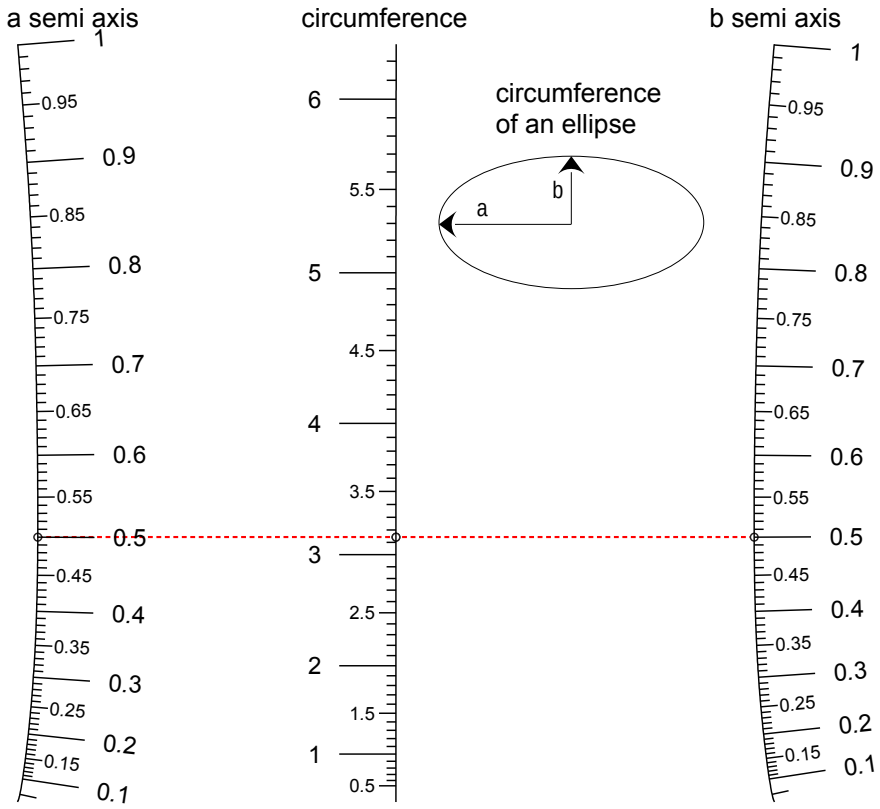
Nomograms are useful for providing a graphical understanding of how 3 variables relate to each other.

Nomogen makes a nomogram quick & easy to create by auto-generating the scale lines from a given function of 2 variables.

Simply enter your formula, along with the max & min values, then nomogen generates a nomogram, possibly (probably) non linear.

- It uses Python to calculate the nomogram numerically using standard python (SciPy) libraries, and
- writes the nomogram to a pdf using pynomo libraries

The scale lines are often called axes and a nomogram with M axes, N of them curved is known as a Class M, Genus N nomogram. Thus nomogen attempts to auto-generate class 3, genus 3 nomograms.



Note that there is a need to recognise that some formulas cannot have a nomogram.

Also nomogen works successfully for a very large class of formulas, but cannot always be guaranteed to succeed.

Using nomogen

Assume you have a python3 setup with pynomo, etc., as described in reference [1].

Write a python function of 2 arguments, corresponding to the left and right hand scales, with the return value corresponding to the middle scale.

You may need to solve your equation numerically if it is an implicit function. See colebrookf.py for an example.

Notes:

1. The function should be smooth, with no steps or kinks (i.e. the function should be differentiable and continuous)
2. At the end points of its specified range the function is evaluated a small distance outside this range, so the function must be valid slightly beyond this range.
   For example, suppose the function contains a square root, and the nomogram plots this function from zero to some maximum. To evaluate the function at zero, the function needs to be evaluated at some number delta < 0, which is impossible for a square root.

nomogen determines the equations of the scale line for you, but you still need to arrange the nomogram layout for things like the nomogram paper size, title, etc. This is described in the pynomo documentation. Also, Ron Doerfler's excellent essay [2] is definitely worth checking out for instructions on setting up the nomogram.

Nomogen uses your equation to derive and set up the determinant equations, so there's no need to do any of that.

The axes are represented by a series of carefully chosen points - which are eventually turned into polynomials - so nomogen needs to know the number of points to use. The more straight the axes, and the more evenly spaced the intervals, the fewer points needed. Use trial and error, starting with a degree of about 7.

Fewer points make the calculations faster, more points are more accurate.

An axis, can be configured for 'linear smart' if the scale is approximately evenly spaced. If the high values on the scale are bunched together more tightly, the scale type might be better set to 'log' or 'log smart'. Nomogen plots the log of these scale variables, so the nomogram might be generated more quickly and accurately.

Note that some functions are not accurately representable as a nomogram, so reducing the range of one or more scales should improve accuracy.

When a nomogram is calculated, nomogen reports an estimate of the tolerance. If the tolerance is larger than 0.1mm, it is shown on the nomogram.

Normally, the outer scales of the nomogram are fixed to the corners of the defined nomogram area.

Sometimes a better nomogram results when the scale line end points are free from being tied to the corners of the nomogram area. This is allowed by the 'muShape' parameter – this shapes the nomogram by trying to expand the axes so they can be read more easily. See the pendulum.py program for an example.

There's lots of reference examples included with nomogen.

Step by Step Example

This example shows how to set up a nomogram using nomogen and that even complicated equations can be used to generate a nomogram quickly and easily.

Let's take example (xv) from Allcock & Jones, ref [3], the relative load on a retaining wall.

Details are here:

https://babel.hathitrust.org/cgi/pt?id=mdp.39015000960115&view=1up&seq=128

The formula to use is:

$$(1+L)h^2 \; - \; Lh(1+p) \; - \; \frac{1}{3}(1-L)(1+2\,p) \; = \; 0$$

with

$$0.5 \; \leq \; L \; \leq \; 1$$

$$0.5 \; \leq \; p \; \leq \; 1$$

thus

$$0.75 \; \leq \; h \; \leq \; 1 \quad .$$

The nomogram is to fit inside a 10 cm x 15 cm rectangle.

Use example file test1.py as a template, copy it to, say, myproj.py and edit that.

Write a new function **AJh(L, p)** replacing (or perhaps ignoring) the existing w(u,v). A beginners tutorial is given in ref [2].

The function above cannot be rearranged to give a value for h (it's a so-called implicit function of h), but it is a function of the form

$Ah^2 \; + \; Bh \; + \; C \; = \; 0$, from which h can be found using the quadratic formula.

Assigning values for the maximum and minimum of each of the variables gives a code fragment something like this:

```
def AJh(L,p):
    a = 1+L
    b = -L*(1+p)
    c = -(1-L)*(1+2*p)/3
    h = ( -b + math.sqrt(b**2 - 4*a*c) ) / (2*a)
    return h

Lmin = 0.5; Lmax = 1.0;
pmin = 0.5; pmax = 1.0;
hmin = AJh(Lmax, pmin);
```

```
hmax = AJh(Lmin, pmax);
NN = 3
```

The variable NN is the number of points needed to define the scales or axes. It can be tricky to get right, but 3 works well for this example.

Now define the scales as you would normally do with pynomo. **L** and **p** are the left and right scales and **h** (the function value) is in the middle. For each scale, set the minimum & maximum values, and (optionally) a title. For this example, leave the other definitions as they are:

```
left_axis = {,'u': {}, 'v': {},
    'u_min': Lmin,
    'u_max': Lmax,
    'title': r'$L$',
    'scale_type': 'linear smart',
    'tick_levels': 3,
    'tick_text_levels': 2
}

right_axis = {
    'u_min': pmin,
    'u_max': pmax,
    'title': r'$p$',
    'scale_type': 'linear smart',
    'tick_levels': 3,
    'tick_text_levels': 2 ,
}

middle_axis = {
    'u_min': hmin,
    'u_max': hmax,
    'title': r'$h$',
    'scale_type': 'linear smart',
    'tick_levels': 3,
    'tick_text_levels': 2
}
```

Note that when the title text sits between the '$' characters, as in `r'$myTitle$'`, the text is printed in maths mode. This prints in italic text and enables a wider range of characters, like Greek letters, and subscripts and superscripts. For normal Roman text, omit the `'$'` characters. The r character in `r'..'` implies a raw string (i.e. python does not interpret escape codes), so escape codes are passed directly to the TeX print formatter.

It is not always obvious how to add text, so it's worth checking ref [2] beforehand.

In the `main_params` section, set the nomogram height and width, in cm, and add a title:

```
main_params = {
    'filename': __file__.endswith(".py") and __file__.replace(".py", ".pdf")
or "nomogen.pdf",
    'paper_height': 15, # units are cm
    'paper_width': 10,
    'title_x': 4.5,
    'title_y': 1.5,
    'title_str':r'example',
    'block_params': [block_params0],
    'transformations': [('scale paper',)],
    'npoints': NN
}
```

The parameters 'title_x' & 'title_y' are the x and y positions of the title, measured in cm from the bottom left of the nomogram.

Finally, give nomogen the function defined above:

```
Nomogen(AJh, main_params);  # generate nomogram for AJh function
```

That's it!

Appendix 1 below shows the complete python program with the relevant parts highlighted.

Now run the program, and a neatly generated nomogram will be in myproj.pdf- see Figure 2 below.

The pynomo documentation and ref[2] describe lots of other options for setting up and fine tuning the nomogram. See also the example Ajh.py.

Compare this with the calculations shown in the example in ref [3], which needs several pages of algebra and geometric theory to derive the curves for the nomogram. There's no need for that with nomogen, just write the function and set the parameters.

While generating the nomogram, nomogen prints the following:

```
calculating the nomogram ...
using 3 Chebyshev points
AJh costs:  24, eAcc 8.64e-07, eDer 1.59e-06, 6.58e-06, cost improvement is 10%
cost function is 1.00e-06, Optimization terminated successfully.
max derivative error is 0.0013
checking solution every  1 mm,
alignment error is estimated at less than  0.07 mm
putting v scale ticks on left side
putting w scale ticks on right side
printing  AJh.pdf  ...
```

The essential information is the alignment error, shown highlighted above. It is defined as the maximum distance between any index line and the expected reading on the middle scale line (the line labelled *h* for this nomogram).

Figure 1 below shows a measurement from this nomogram. The points **A** and **B** correspond to some values of *L* & *p* and *h* corresponds to the value given by the nomogram's formula. Since this nomogram has been derived numerically, h will not necessarily lie exactly on the index line **AB**. The alignment error, e, is the distance from the index line to the expected value h, and we now know that e will always be less than 0.07mm, thus showing that this nomogram easily accurate enough for practical use.

Some more information on alignment errors is given on page 7 below.

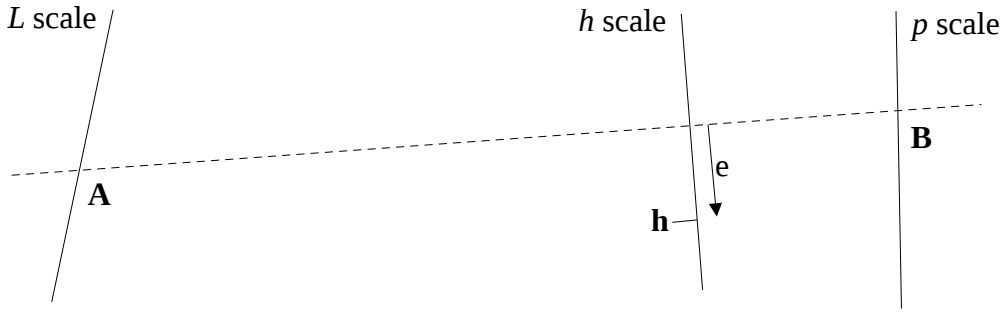
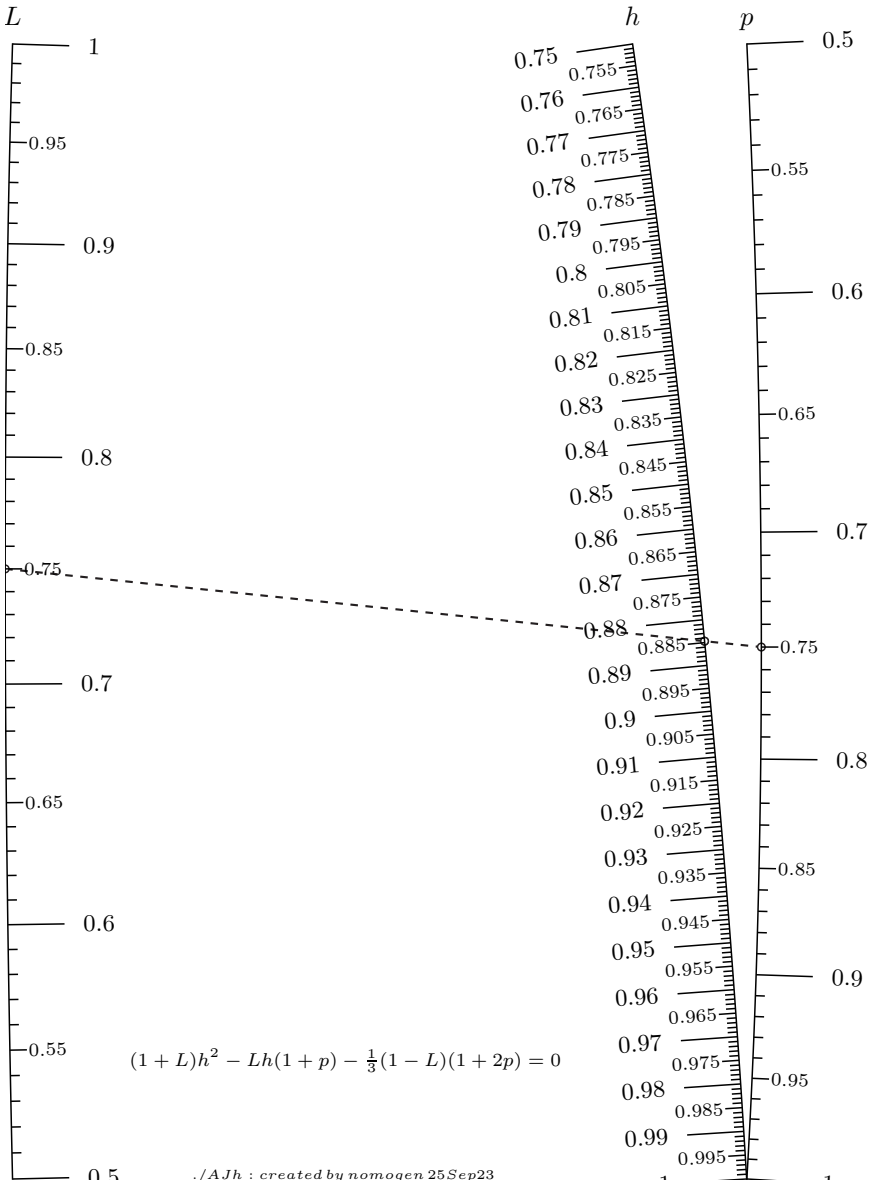


Figure 1

Figure 2: Example Nomogram



The Lemmon Equation

In this example we show again how nomogen can easily create a nomogram from a very difficult equation and introduce the use of log scales. Also we'll see how to examine alignment errors in more detail and discuss briefly the number of points needed to define the axes.

The Lemmon equation describes the compressibility of hydrogen as a function of temperature and pressure:

$$Z(p,T) = 1 + \sum_{i=1}^{9} a_i \left(\frac{100\,K}{T}\right)^{b_i} \left(\frac{p}{1\,MPa}\right)^{c_i}$$

Reference [4] describes the details, including the values of the constants (all 27 of them!).

The equation is very complex, but is relatively easy to program into nomogen.

We need to set NN, this time to 7, and choose a custom footer string in the main parameter block:

```
'footer_string': r'Lemmon equation'
```

The complete code is shown in Appendix 2 below, with the described code changes highlighted. Additionally, other changes as described in the previous example need to be added.

The resulting nomogram is shown in Figure 4 below.

log scales

Notice that the Z axis has its higher values closer together than the lower values – this is a strong hint that a log scale should be used for this axis. Checking that Z is never zero or less, we see that a log scale can indeed be specified for the Z axis and this is shown in the code.

Nomogen will report an error if a log scale is specified when the corresponding axis contains a zero or negative range.

Alignment Errors

We saw above that nomogen reports the maximum alignment error. Nomogen optionally will export a file of alignment error measurements if the block parameters contains this line:

```
'LogAlignment': True
```

Nomogen measures the alignment error for all combinations of points at every millimetre along the temperature and pressure scales and saves the results in the file lemmon.error.py. Other nomograms will have a corresponding name. This file can be used by other utility programs to analyse the error data. The example program show_error.py uses this data to display a scatter plot of the errors as shown in Figure 3 below. Note that once again the alignment errors are well within the limits required for a practical nomogram.

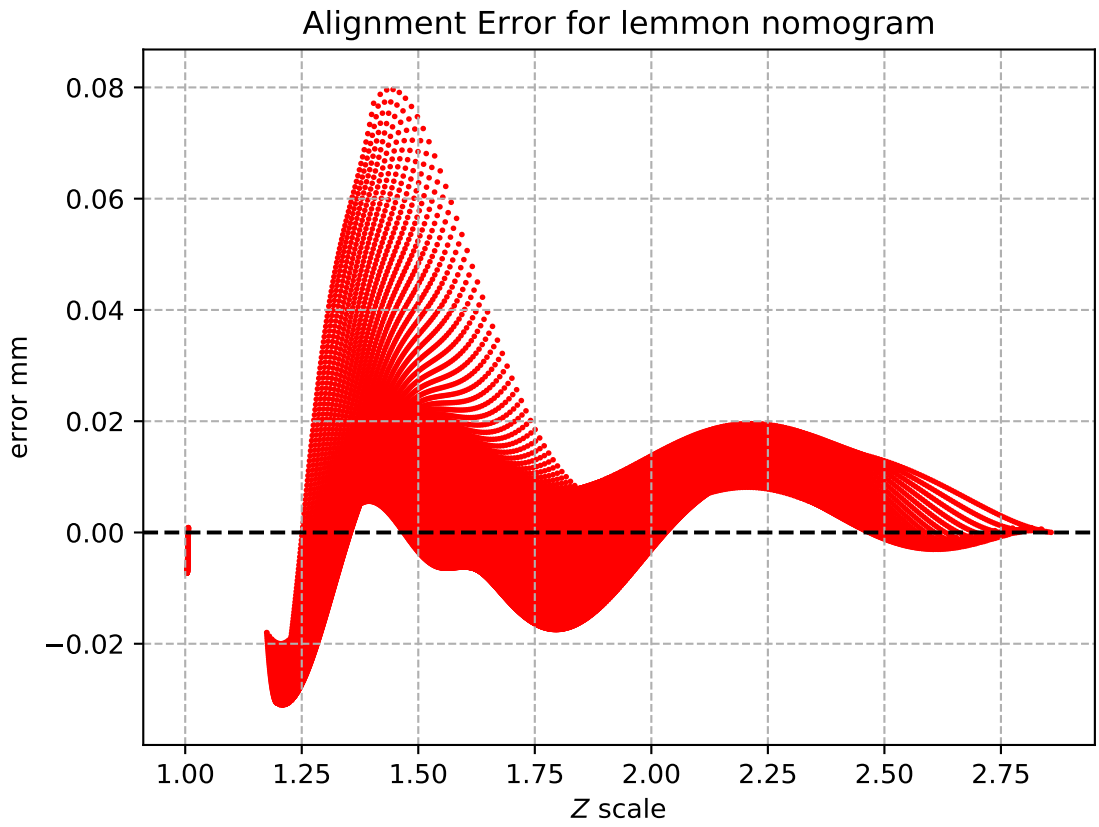


Figure 3: Alignment errors for Lemmon nomogram

Number of points

The geometry of this nomogram looks very similar to the example on page 3 above, so why does it need 7 points rather than 3?

The answer is that the distances along the axes are not so close to linear. Looking carefully at the pressure scale for example, the distance between the values is greater at higher pressures. The pressure indicated by the arrows labelled ‘a’ & ‘b’ are respectively about 14MPa more than the minimum pressure and the same amount less than the maximum pressure – so the two distances for the same pressure change are very different.

The 7 points for this example are carefully placed at fixed values of T, P & Z on the axes. The large dots on pressure axis show this.

Note that the dots are fixed to the value of the pressure, not to a position. Nomogen’s algorithms jiggle the position of the points around until the nomogram is accurate then smoothly join the points to make the axes.

So the points control not only the locus of the axes, but also the scale along the lines. They need to force the axis to bend to the correct path and stretch or shrink the scales to make the nomogram accurate.

If the distances between the values is constant, few points are needed (as shown in the earlier example), but if the distances vary, more points are needed as we see in this example.

A suitable value for NN can be obtained through rial & error. Here's a few hints:

- Start with, say, 7 points
- if the accuracy is good, try reducing the number of points to speed up generation of the nomogram
- if the accuracy is not good enough try using more points to increase accuracy
- if increasing the number of points doesn't doesn't improve accuracy try reducing the range of the scales
- if axes need to be a little smoother try reducing the number of points

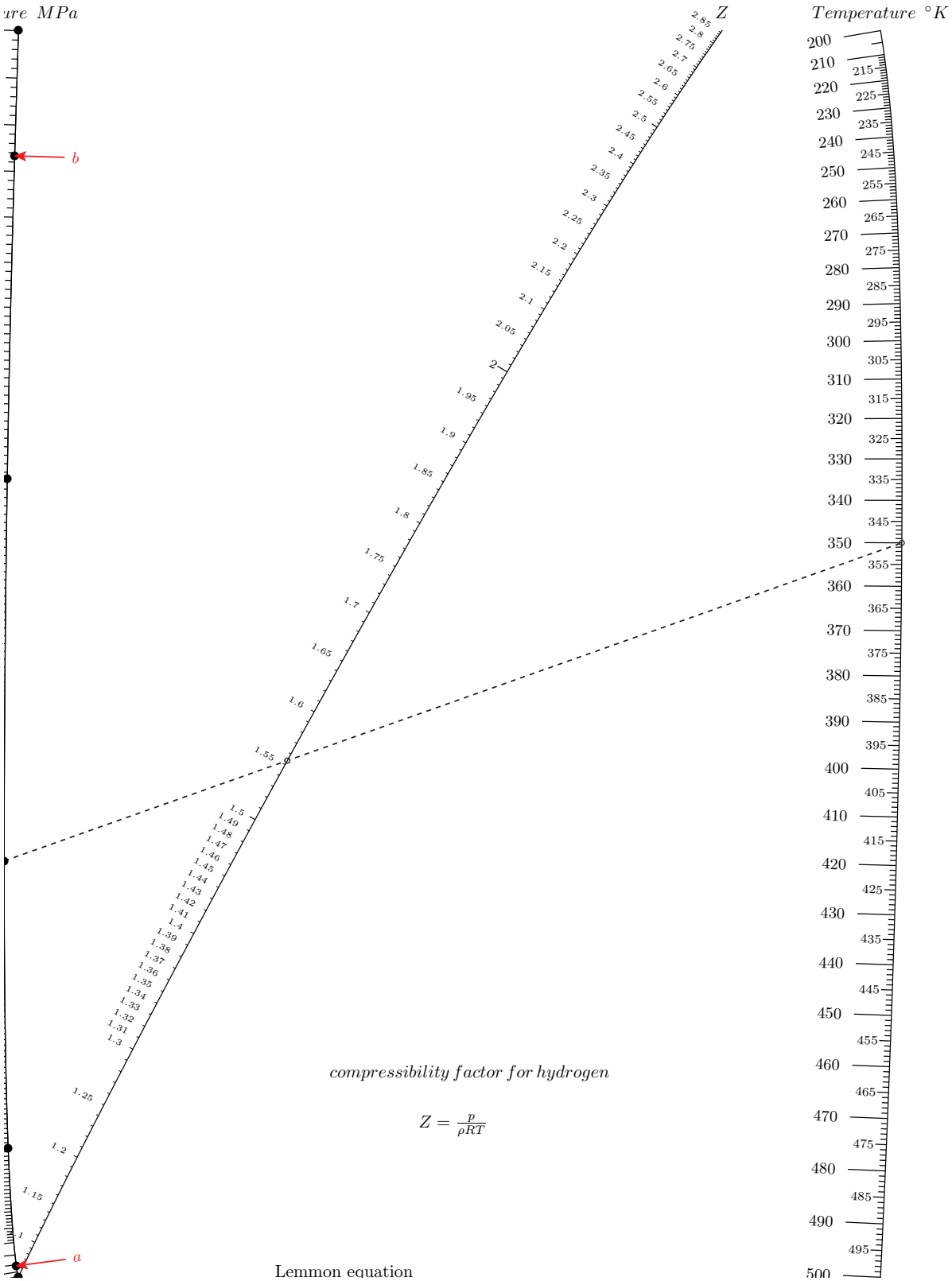


Figure 4: Nomogram for the Lemmon equation

Hiking Equation

This example introduces more advanced features like dual scales and relaxes the requirement that the outer axes are tied to the corners of the printed area. We also see that the scales cannot extent up to the limit of their valid range.

This equation comes from reference [5] - energy expended when walking in hilly terrain and is

$$EE = 1.44 + 1.94S^{0.43} + 0.24S^{4} + 0.34SG\left(1-1.05^{1-1.11^{(G+32)}}\right)$$

As before, use an existing example as a template and start by defining the equation and the ranges of the variables.

```
"""
return energy expended W/kg of body mass
S is  walking speed km/hr
G is  gradient %
"""


def EE(S,G):

    # S is km/hr, need m/s
    S *= 10.0 / 36.0
    t1 = S
    t1 = t1**0.43
    t1 = t1 * 1.94
    t2 = 0.24*S**4
    t3 = 0.34*S*G*(1-1.05**(1-1.11**(G+32)))

    return 1.44 + t1 + t2 + t3



# range for speed km/hr
Smin = 0.1 * 36 /10        # 0.1 m/s -> km/hr
Smax = 3 * 36 /10          # 3 m/s -> km/hr

# range for slope
Gmax = +25
Gmin = -Gmax

# range for energy expended
EEmin = 1.44 #EE(Smin, Gmin)
EEmax = EE(Smax, Gmax)
```

A good value for the number of points for the polynomials is 15. More details about finding this value are given below. The relevant code is

```
NN = 15
```

The walking speed axis will have both metric and imperial scales, and the expended energy axis will have a scale for W kg^{-1} and calories per hour for a person weighing 155 lb.

Start with this axis having a metric scale (the other scale will be added later):

```
# km/hr scale of the left axis
left_axis = {
    'tag': 'left',            # link to alternative scale
    'u_min': Smin,
```

```
    'u_max': Smax,
    'title': r'walking \thinspace speed',
    'extra_titles':[
        {'dx':-2.5,
         'dy':-0.0,
         'text':r'$\small km \thinspace hr^{-1}$',
         'width':5,
         }],
    'scale_type': 'linear smart',
    'tick_levels': 6,
    'tick_text_levels': 3,
    'tick_side': 'left',
}
```

Notice the ‘tick_side’ field which puts the metric scale on the left hand side of the axis, and the ‘tag’ field which will be used to link this axis to the imperial scale. There is also an ‘extra_titles’ field to label the metric scale on the axis.

Put the gradient scale on the right hand axis. This is very similar to the other examples above, but it will not have a dual scale so there is no tag parameter.

```
right            # u scale_axis = {
    'u_min': Gmin,
    'u_max': Gmax,
    'title': r'$gradient \thinspace \%$',
    'title_x_shift': 0.6,
    'scale_type': 'linear smart',
    'tick_levels': 5,
    'tick_text_levels': 3,
    'tick_side': 'left',
}
```

The middle axis will have dual scales. Firstly set up the W kg^{-1} scale here, the other scale will be added later.

```
middle_axis = {
    'tag': 'middle',            # link to alternative scale
    'u_min': EEmin,
    'u_max': EEmax,
    'title': r'$Expended \thinspace energy$',
    'title_x_shift': -1.3,
    'extra_titles':[
        {'dx':-2.5,
         'dy':-0.0,
         'text':r'$\small Wkg^{-1}$',
         'width':5,
         }],
    'scale_type': 'linear smart',
    'tick_levels': 5,Figure 1: Example Nomogram
    'tick_text_levels': 3,
    'tick_side': 'left',
}
```

As before, we have used the `'tag'`, `'tick_side'` and `'extra_titles'` fields.

Dual scales

Before delving further into dual scales, we need to recall that pynomo constructs a nomogram from a series of blocks, where each block implements an equation. Each block consists of one or more axes, each of which represents a variable.

The examples above defined an axis for each of the three variables, assembled these axes into a block that represents an equation, and constructed the nomogram from that single block.

To implement the mph scale on the left axis we need to create a new type 8 block that holds another single axis for the mph scale that overlays the km/h axis.

The axis needs the same tag as the kph scale axis, `'left'` in this case, and with ticks on the right because the km/hr scale is on the left.

This axis uses mph values, so set `'u_min'` & `'u_max'` to mph values matching the min & max values on the km/hr axis, and define `'align_func'` which is a scaling function that defines how mph values align with the km/hr values.

The resulting code should look something like this:

```
# mph scale for left axis

km_per_mile  = 1.609344
left_axis_mph = {
    'tag': 'left',
    'u_min': left_axis['u_min'] / km_per_mile,
    'u_max': left_axis['u_max'] / km_per_mile,
    'extra_titles':[
        {'dx':-0.1,
         'dy':0.0,
     'text':r'$\small mph$'
         }],
    'align_func': lambda m: m*km_per_mile,
    'scale_type': 'linear smart',
    'tick_levels': 5,
    'tick_text_levels': 3,
    'tick_side': 'right'
}
```

Finally put this axis into a type_8 block, like this:

```
block_1_params={
    'block_type':'type_8',
    'f_params': left_axis_mph,
    'isopleth_values':[['x']],
```

```
}
```

Repeat this process for the energy axis then assemble all the blocks into the nomogram by extending `'block_params'` in the main_params definition:

```
'block_params': [block_params0, block_1_params, block_2_params],
```

The resulting code is shown in Appendix 3 on page 24.

Here we overlaid two separate type_8 (i.e. single scale) nomograms over the original nomogram to create the dual scales.

Alternatively, we could overlay a single type_9 nomogram over the original nomogram as is done in the piedra.py example.

Removing constraints on the outer axes

Normally, nomogen fixes the ends of the outer axes to the corners of the printed area to make the nomogram as large as possible.

Alternatively, nomogen can calculate the axes to be as long as possible, with index lines as close to perpendicular as possible to reduce parallax errors.

This is enabled with the ‘muShape’ parameter, and code like the following is added to the main parameter block:

```
  'muShape': 1,
```

If the `'muShape'` parameter is not present or has the value 0 the ends of the scale lines are tied to the corners of the defined nomogram area.there is a need to recognise

Sometimes this option produces a better nomogram, other times it just takes longer to make a nomogram that almost the same.

when speed goes to zero

The nomogram would fail to be generated if the minimum speed was set to zero. To keep iterating the momogram toward smaller errors, the optimisation algorithm needs to evaluate energy around values S=0, including at values less than 0. Note that the equation contains the term $S^{0.43}$ which cannot be evaluated for S<0.

The solution is to change the range of S so that the min is slightly greater than 0.

Interpreting the nomogram

Now let’s take a minute to see what the nomogram tells us.

We see that the speed and energy axes cross when the speed is zero. In other words, you are standing on a hillside, and using your “at rest” energy, independent of what the gradient might be.

Of course, we already know this, but we can use this knowledge to confirm that the equations represented by the nomogram match our understanding.

Looking more closely, at a gradient of, say, -4% you use the same energy as with a gradient at -20%. At the shallower gradient, you are using energy to maintain your speed, while at the steeper gradient you are using energy to stop accelerating faster.

Further, at a gradient of -10% you are using minimum energy. For any walking speed less than about 5 kmh^{-1} (3 mph), you are using no more than “at rest” energy, with gravity is providing all the walking energy.

So, on the one hand, we see that nomograms help us understand the equations a little better. On the other hand, we could see problems immediately if the equations contained an error.

This makes nomograms an important verification tool for experimentally derived equations.

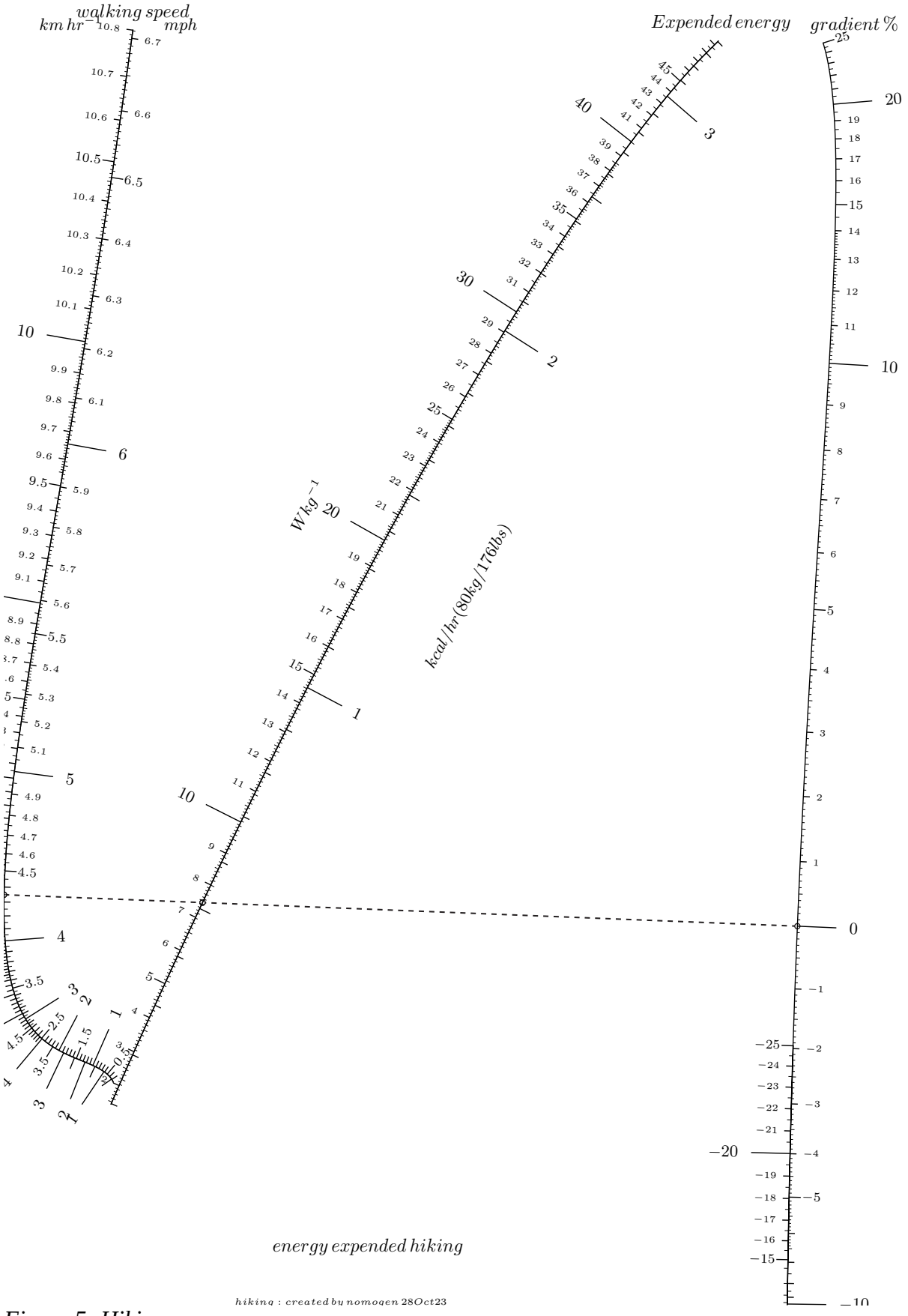


Figure 5: Hiking nomogram

Other features

Here we describe features not covered above.

Anamorphosis

Anamorphosis is the stretching and shrinking of the values along the scale lines – we have already seen an example of this with the use of log scales.

This is important because nomogen uses polynomials and there are some functions that polynomials cannot represent in a practical manner. For example exponentials grow much faster than polynomials. Of course a polynomial of very high degree can represent such a function over a limited range in a nomogram, at a cost of taking much longer to generate the nomogram.

One solution is to provide nomogen with with a function and its inverse so the plot is approximately linear. In the student-t nomogram the v axis extends to infinity, which of course, cannot be implemented by a finite polynomial. If we pre-scale this axis with, say, the arctan function then a range that extended to infinity now extends to $\pi/2$ and this can be implemented by a polynomial.

Nomogen Reference

Nomogen adds these parameters to the pynomo main_params block:

Name	description
`npoints`	Number of points used to approximate the axes. If this is missing or is not an integer a default value of 9 is used.
`footer_string`	Nomogram configuration string printed at the bottom of the printed nomogram. If missing, a default text containing the date is used.
`muShape`	A non zero value for this parameter frees the nomogram from the corners of the unit square. Instead the shape of the nomogram is made as tall and wide as possible while staying within its allotted bounds. The value is ignored. Try this if some axes create large parallax errors because they are almost parallel to the index lines. Sometimes this creates a better looking nomogram, sometimes it doesn't.
`LogAlignment`	If this parameter has a non-zero value then nomogen writes alignment errors to a file in addition to generating a nomogram. If the nomogram file is called xxx.py, then the error file will be called xxx.error.py.

Nomogen adds these parameters to an axis parameter block:

`anamorphosis`[1]	`eg 'anamorphosis': {'func': math.atan, 'inv': math.tan}`

1 Currently, this is an experimental feature - meaning that it might be modified or replaced in the future.

Appendix 1

```
#!/usr/bin/python3

# nomogen example program

import sys

sys.path.insert(0, "..")

import math

from nomogen import Nomogen
from pynomo.nomographer import Nomographer


########################################
#
#  this is the target function,
#  - the limits of the variables
#  - the function that the nonogram implements
#
#  format is m = m(l,r), where l, m & r are respectively the values
#                        for the left, middle & right hand scales
########################################

################################################
#
# from Allcock & Jones, example xv, p112..117
# "the classical example of a nomogram consisting of three curves ..."
# Load on a retaining wall
# (1+L)*h**2 - L*h*(1+p) - (1-L)*(1+2*p)/3 == 0

def AJh(L, p):
    a = 1 + L
    b = -L * (1 + p)
    c = -(1 - L) * (1 + 2 * p) / 3
    h = (-b + math.sqrt(b ** 2 - 4 * a * c)) / (2 * a)
    return h

Lmin = 0.5;
Lmax = 1.0;
pmin = 0.5;
pmax = 1.0;
hmin = AJh(Lmax, pmin);
hmax = AJh(Lmin, pmax);

################################################################
#
# nr Chebyshev nodes needed to define the scales
# a higher value may be necessary if the scales are very non-linear
# a lower value increases speed, makes a smoother curve, but could introduce
errors
NN = 3

################################################
#
# definitions for the scales for pyNomo
# dictionary with key:value pairs
```

```
left_axis = {
    'u_min': Lmin,
    'u_max': Lmax,
    'title': r'$L$',
    'scale_type': 'linear smart',
    'tick_levels': 3,
    'tick_text_levels': 2
}

right_axis = {
    'u_min': pmin,
    'u_max': pmax,
    'title': r'$p$',
    'scale_type': 'linear smart',
    'tick_levels': 3,
    'tick_text_levels': 2
}

middle_axis = {
    'u_min': hmin,
    'u_max': hmax,
    'title': r'$h$',
    'scale_type': 'linear smart',
    'tick_levels': 3,#!/usr/bin/python3


block_params0 = {
    'block_type': 'type_9',
    'f1_params': left_axis,
    'f2_params': middle_axis,
    'f3_params': right_axis,
    'transform_ini': False,
    'isopleth_values': [[(left_axis['u_min'] + left_axis['u_max']) / 2, \
                         'x', \
                         (right_axis['u_min'] + right_axis['u_max']) / 2]]
}

main_params = {
    'filename': __name__ == "__main__" and (
            __file__.endswith(".py") and __file__.replace(".py", "") or
"nomogen") or __name__,
    'paper_height': 15,  # units are cm
    'paper_width': 10,
    'title_x': 4.5,
    'title_y': 1.5,
    'title_box_width': 8.0,
    'title_str': r'$(1+L)h^2 - Lh(1+p) - {1 \over 3} (1-L)(1+2p) = 0$',
    'block_params': [block_params0],
    'transformations': [('scale paper',)],
    'npoints': NN
}

print("calculating the nomogram ...")
Nomogen(AJh, main_params)  # generate nomogram for AJh function

main_params['filename'] += '.pdf'
print("printing ", main_params['filename'], " ...")
Nomographer(main_params)
```

Appendix 2

```
#!/usr/bin/python3

# Lemmon equation
# "Revised Standardized Equation for Hydrogen Gas Densities for Fuel Consumption
Applications"
# Journal of Research of the National Institute of Standards and Technology,
2008 Nov-Dec; 113(6): 341–350.
# Eric W. Lemmon, Jacob W. Leachman and Marcia L. Huber

import sys

sys.path.insert(0, "..")

from nomogen import Nomogen
from pynomo.nomographer import Nomographer

import math

# Constants associated with the density equation for normal hydrogen
# i       ai             bi     ci
# 1  0.058 88460        1.325   1.0    # log alignment errors
    # If this is missing or False then alignment error logs are disabled
    'LogAlignment': True,

# 2 −0.061 361 11       1.87     1.0
# 3 −0.002 650 473      2.5      2.0
# 4  0.002 731 125      2.8      2.0
# 5  0.001 802 374      2.938   2.42
# 6 −0.001 150 707      3.14    2.63
# 7  0.958 852 8 × 10−4  3.37   3.0
# 8 −0.110 904 0 × 10−6  3.75   4.0
# 9  0.126 440 3 × 10−9  4.0     5.0

a = [0.05888460, -0.06136111, -0.002650473, 0.002731125, 0.001802374,
     -0.001150707, 0.9588528E-4, -0.1109040E-6, 0.1264403E-9]

b = [1.325, 1.87, 2.5, 2.8, 2.938, 3.14, 3.37, 3.75, 4.0]

c = [1.0, 1.0, 2.0, 2.0, 2.42, 2.63, 3.0, 4.0, 5.0]

M = 2.01588  # Molar Mass, g/mol
R = 8.314472  # Universal Gas Constant,  J/(mol · K)


# p pressure in MPa
# T degrees K
def Z(p, T):
    s = 1
    for i in range( len(a) ):
        s = s + a[i] * (100 / T) ** b[i] * p ** c[i]
    return s


pmin = 1
pmax = 200
Tmin = 200
Tmax = 500
Zmin = 1
Zmax = Z(pmax, Tmin)
```

```
# Test points for validating
# T(K) p(MPa)     Z        ρ (mol/1)
# 200    1    1.00675450   0.59732645
# 300    10   1.05985282   3.78267048
# 400    50   1.24304763   12.09449023
# 500    200  1.74461629   27.57562673
# 200    200  2.85953449   42.06006952

correct = True
if not math.isclose(Z(1, 200), 1.00675450, abs_tol=5e-09):
    print("Z(1,200) fails")
    correct = False

if not math.isclose(Z(10, 300), 1.05985282, abs_tol=5e-09):
    print("Z(10,300) fails")
    correct = False

if not math.isclose(Z(50, 400), 1.24304763, abs_tol=5e-09):
    print("Z(50,400) fails")
    correct = False

if not math.isclose(Z(200, 500), 1.74461629, abs_tol=5e-09):
    print("Z(200,500) fails")
    correct = False

if not math.isclose(Z(200, 200), 2.85953449, abs_tol=5e-09):
    print("Z(200,200) fails")
    correct = False

if not correct:
    # print(  Z(1,200), Z(10,300), Z(50,400), Z(200,500), Z(200,200) )
    sys.exit("test points for Z(p,T) failed")


##################################################################
#
# nr Chebyshev nodes needed to define the scales
# a higher value may be necessary if the scales are very non-linear
# a lower value increases speed, makes a smoother curve, but could introduce
errors
NN = 7


################################################
#
# definitions for the scales for pyNomo
# dictionary with key:value pairs

left_axis = {
    'u_min': pmin,
    'u_max': pmax,
    'title': r'Pressure MPa',
    'scale_type': 'linear smart',
    'tick_levels': 3,
    'tick_text_levels': 2,
}

right_axis = {
    'u_min': Tmin,
    'u_max': Tmax,
```

```
    'title': r'Temperature $ ^\circ $K',
    'scale_type': 'linear smart',
    'tick_levels': 3,
    'tick_text_levels': 2,
}

middle_axis = {
    'u_min': Zmin,
    'u_max': Zmax,
    'title': r'$Z$',
    'scale_type': 'log smart',
    'tick_levels': 6,
    'tick_text_levels': 5,
}

block_params0 = {
    'block_type': 'type_9',
    'f1_params': left_axis,
    'f2_params': middle_axis,
    'f3_params': right_axis,
    'transform_ini': False,

    # log alignment errors
    # If this is missing or False then alignment error logs are disabled
    'LogAlignment': True,

    'isopleth_values': [[(left_axis['u_min'] + left_axis['u_max']) / 2, \
                         'x', \
                         (right_axis['u_min'] + right_axis['u_max']) / 2]]
}

main_params = {
    'filename': __name__ == "__main__" and (
                __file__.endswith(".py") and __file__.replace(".py", "") or
"nomogen") or __name__,
    'paper_height': 25,  # units are cm
    'paper_width': 18,
    'title_x': 9.0,
    'title_y': 3.0,
    'title_box_width': 8.0,
    'title_str'Finally : r'$\Large Z = {p \over {\rho R T}}$',
    'extra_texts': [
        {'x': 6,
         'y': 4,
         'text': r'compressibility factor for hydrogen',
         'width': 10,
         }],
    'block_params': [block_params0],
    'transformations': [('scale paper',)],
    'npoints': NN,

    # text to appear at the foot of the nomogram
    # make this null string for nothing
    # a default string will appear if this is omitted
    'footer_string': r'Lemmon equation'
}

print("calculating the nomogram ...")
Nomogen(Z, main_params)  # generate nomogram for the function
main_params['filename'] += '.pdf'
print("printing ", main_params['filename'], " ...")
Nomographer(main_params)
```

Appendix 3

```
#!/usr/bin/python3

"""
energy expended while hiking, accounting for slope and speed
use dual scales for speed & energy axes

https://getpocket.com/explore/item/this-is-how-many-calories-you-burn-on-a-hilly-hike

 https://pubmed.ncbi.nlm.nih.gov/30973477/

"""

import sys

sys.path.insert(0, "..")

from nomogen import Nomogen
from pynomo.nomographer import Nomographer


########################################
#
#  this is the target function,
#  - the function that the nonogram implements
#  - add the limits of the variables below
#
#  format is m = m(l,r), where l, m & r are respectively the values
#                        for the left, middle & right hand scales
#
#
########################################

"""
return energy expended W/kg of body mass
S is  walking speed km/hr
G is  gradient %
"""


def EE(S,G):

    # S is km/hr, need m/s
    S *= 10.0 / 36.0
    t1 = S
    t1 = t1**0.43
    t1 = t1 * 1.94
    t2 = 0.24*S**4
    t3 = 0.34*S*G*(1-1.05**(1-1.11**(G+32)))

    return 1.44 + t1 + t2 + t3



# range for speed km/hr
Smin = 0.1 * 36 /10        # 0.1 m/s -> km/hr
Smax = 3 * 36 /10          # 3 m/s -> km/hr
```

```
# range for slope
Gmax = +25
Gmin = -Gmax

# range for energy expended
EEmin = 1.44 #EE(Smin, Gmin)
EEmax = EE(Smax, Gmax)


#####################################################################
#
# nr Chebyshev nodes needed to define the scales
# a higher value may be necessary if the scales are very non-linear
# a lower value is faster, makes a smoother curve,
#     but could be less accurate
NN = 15


################################################
#
# definitions for the scales for pyNomo
# dictionary with key:value pairs

# km/hr scale of the left axis
left_axis = {
    'tag': 'left',            # link to alternative scale
    'u_min': Smin,
    'u_max': Smax,
    'title': r'walking  speed',
    'extra_titles':
        {'dx':-2.5,
         'dy':-0.0,
         'text':r'$\small km \thinspace hr^{-1}$',
         'width':5,
         }],
    'scale_type': 'linear smart',
    'tick_levels': 6,
    'tick_text_levels': 3,
    'tick_side': 'left',
}

right_axis = {
    'u_min': Gmin,
    'u_max': Gmax,
    'title': r'gradient \%',
    'title_x_shift': 0.6,
    'scale_type': 'linear smart',
    'tick_levels': 5,
    'tick_text_levels': 3,
    'tick_side': 'left',
}

middle_axis = {
    'tag': 'middle',            # link to alternative scale
    'u_min': EEmin,
    'u_max': EEmax,
    'title': r'Expended  energy',
    'title_x_shift': -1.3,
    'extra_titles':[
        {'dx':-2.5,
         'dy':-0.0,
         'text':r'$\small Wkg^{-1}$',
         'width':5,
```

```
         }],
    'scale_type': 'linear smart',
    'tick_levels': 5,
    'tick_text_levels': 3,
    'tick_side': 'left',
}

# assemble the above 3 scale lines into a block
block_params0 = {
    'block_type': 'type_9',
    'f1_params': left_axis,
    'f2_params': middle_axis,
    'f3_params': right_axis,

    # the isopleth connects the mid values of the outer scale lines
    # edit this for different values
    'isopleth_values': [[7, 'x', 0]]
}


######## the second scales ##############


# mph scale for left axis

km_per_mile = 1.609344
left_axis_mph = {
    'tag': 'left',
    'u_min': left_axis['u_min'] / km_per_mile,
    'u_max': left_axis['u_max'] / km_per_mile,
    'extra_titles':[
        {'dx':-0.1,
         'dy':0.0,
         'text':r'$\small mph$',
         }],
    'align_func': lambda m: m*km_per_mile,
    'scale_type': 'linear smart',
    'tick_levels': 5,
    'tick_text_levels': 3,
    'tick_side': 'right'
}

block_1_params={
    'block_type':'type_8',
    'f_params': left_axis_mph,
    'isopleth_values':[['x']],
}



# calorie scale for middle axis

# 1 nutrition calorie Cal   = 4186.80  joules J
# 1 kg = 2.20462262 lbs
# kilo-calories per hour for 70kg hiker

# weight of hiker in kg & lbs
wkg = 80
wlbs = round(80*2.20462)

watts_per_calph = 4186.80*1000/wkg/3600
```

```
middle_axis_cal = {
    'tag': 'middle',
    'u_min': middle_axis['u_min'] / watts_per_calph,
    'u_max': middle_axis['u_max'] / watts_per_calph,
    'title':r'$\small kcal/hr ({}kg/{}lbs)$'.format(wkg,wlbs),
    'title_distance_center': 2.0,
    'title_draw_center': True,
    'align_func': lambda c: c*watts_per_calph,
    'scale_type': 'linear smart',
    'tick_levels': 5,
    'tick_text_levels': 1,
    'tick_side': 'right'
}

block_2_params={
    'block_type':'type_8',
    'f_params': middle_axis_cal,
    'isopleth_values':[['x']],
}


# the nomogram parameters
main_params = {
    'filename': __name__ == "__main__" and (
                __file__.endswith(".py") and __file__.replace(".py", "") or
"nomogen") or __name__,
    'paper_height': 24,  # units are cm
    'paper_width': 16,
    'title_x': 7.0,
    'title_y': 1.0,
    'title_box_width': 8.0,
    'title_str': r'energy expended hiking',
    'block_params': [block_params0, block_1_params, block_2_params],
    'transformations': [('scale paper',)],

    'npoints': NN,

    # instead of forcing the ends of the axes to the corners of the unit square,
    # nomogen can shape the nomogram to minimise parallax errors
    # uncomment the following line to select this option
    ‘muShape': 0,

    # text to appear at the foot of the nomogram
    # make this null string for nothing
    # a default string will appear if this is omitted
    # 'footer_string': (use default)
}

print("calculating the nomogram ...")
Nomogen(EE, main_params)  # generate nomogram for EE function


main_params['filename'] += '.pdf'
print("printing ", main_params['filename'], " ...")
Nomographer(main_params)
```

References

1. PyNomo documentation
2. https://deadreckonings.files.wordpress.com/2009/07/creatingnomogramswithpynomo.pdf
3. Allcock and Jones: The Theory and Practical Construction of Computation Charts
4. Eric W. Lemmon, Jacob W. Leachman and Marcia L. Huber: “Revised Standardized Equation for Hydrogen Gas Densities for Fuel Consumption Applications”, Journal of Research of the National Institute of Standards and Technology, 2008 Nov-Dec; 113(6): 341–350.
   https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4652867/
5. https://pubmed.ncbi.nlm.nih.gov/30973477/

   see also “ This Is How Many Calories You Burn on a Hilly Hike”: Alex Hutchinson, https://www.outsideonline.com/outdoor-adventure/hiking-and-backpacking/how-many-calories-burned-hiking/