Open in Colab
(https://colab.research.google.com/github/christianhbye/bayesian-analysis/blob/main/Project1_part1_288.ipynb)

Project 1 - Part 1

Linear Algebra and Optimization

This notebook is arranged in cells. Texts are usually written in the markdown cells, and here you can use html tags (make it bold, italic, colored, etc). You can double click on this cell to see the formatting.

The ellipsis (...) are provided where you are expected to write your solution but feel free to change the template (not over much) in case this style is not to your taste.

Hit "Shift-Enter" on a code cell to evaluate it. Double click a Markdown cell to edit.

Write your partner's name here (if you have one).

Imports

In [1]:

```
import numpy as np
from scipy.integrate import quad
#For plotting
import matplotlib.pyplot as plt
%matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.colors import LogNorm
```

Mounting Google Drive locally

Mount your Google Drive on your runtime using an authorization code.

Note: When using the 'Mount Drive' button in the file browser, no authentication codes are necessary for notebooks that have only been edited by the current user.

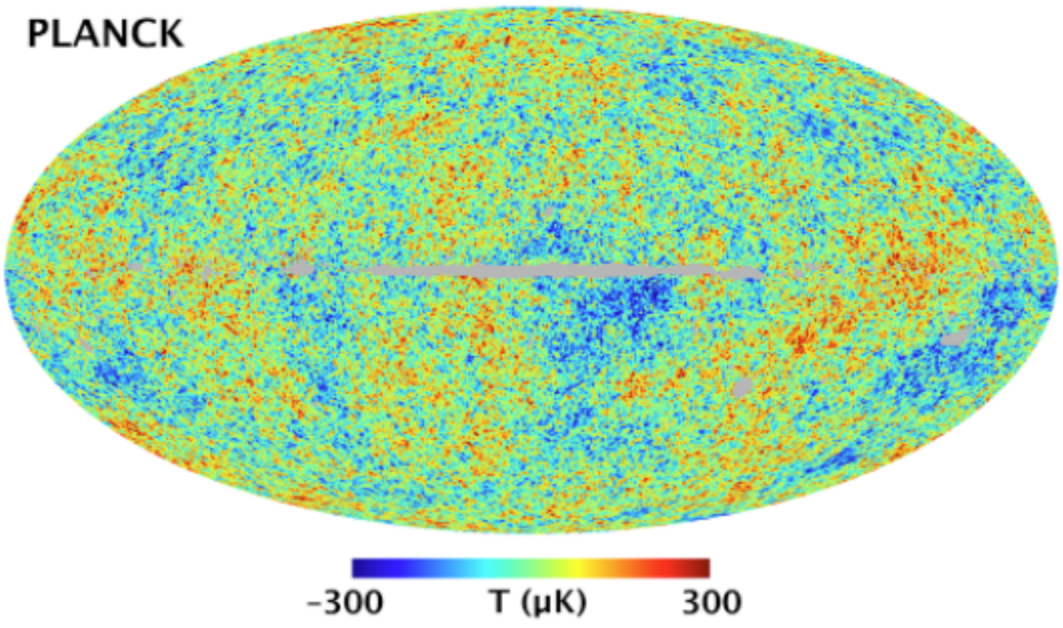
In [2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Problem 1 - Planck analysis cont'd

Planck is the third-generation space telescope, following COBE and WMAP, and it aims to determine the geometry and content of the Universe by observing the cosmic microwave background radiation (CMB), emitted around 380,000 years after the Big Bang. Permeating the whole universe and containing information on the properties of the early Universe, the CMB is widely known as the strongest evidence for the Big Bang model.

Measuring the spectrum of the CMB, we confirm that it is very close to the radiation from an ideal blackbody, and flunctuations in the spectrum are very small. Averaging ocer all locations, its mean temperature is $2.725K$, and its root mean square temperature fluctuation is $\langle(\frac{\delta T}{T})^2\rangle^{1/2} = 1.1 \times 10^{-5}$ (i.e. the temperature of the CMB varies by only ~ 30 μK across the sky).



Suppose you observe the fluctuations $\delta T/T$. Since we are taking measurements on the surface of a sphere, it is useful to expand $\delta T/T$ in spherical harmonics (because they form a complete set of orthogonal functions on the sphere):

$$\frac{\delta T}{T}(\theta,\phi) = \sum_{l=0}^{\infty}\sum_{m=-l}^{l} \mathrm{a}_{lm}\mathrm{Y}_{lm}(\theta,\phi)$$

In flat space, we can do a Fourier transform of a function $f(x)$ as $\sum_k \mathrm{a}_k \mathrm{e}^{ikx}$ where k is the wavenumber, and $|\mathrm{a}_k|$ determines the amplitude of the mode. For spherical harmonics, instead of k, we have l, the number of the modes along a meridian, and m, the number of modes along the equator. So l and m determine the wavelength ($\lambda = 2\pi/l$) and shape of the mode, respectively.

In cosmology, we are mostly interested in learning the statistical properties of this map and how different physical effects influence different physical scales, so it is useful to define the correlation function $C(\theta)$ and split the CMB map into different scales.

Suppose that we observe $\delta T/T$ at two different points on the sky. Relative to an observer, they are in direction $\hat{n}$ and $\hat{n}'$ and are separated by an angle θ given by $cos\theta = \hat{n} \cdot \hat{n}'$ Then, we can find the correlation function by multiplying together the values of $\delta T/T$ at the two points and average the product over all points separated by the angle θ.

$$C(\theta)^{TT} = \Big\langle \frac{\delta T}{T}(\hat{n})\frac{\delta T}{T}(\hat{n}')\Big\rangle_{\hat{n}\cdot\hat{n}'=cos\theta}$$

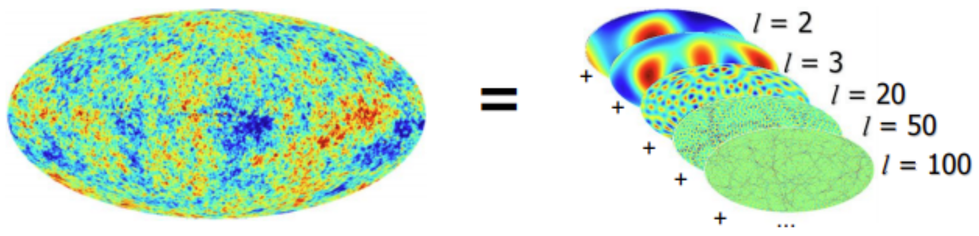
The above expression is specific to the temperature fluctuations, but we can also do a similar analysis for the polarization map of the CMB. (The CMB is polarized because it was scattered off of free electrons during decoupling.) We decompose the polarization pattern in the sky into a curl-free "E-mode" and grad-free "B-mode."

However, the CMB measurements (limited by the experiment resolution and the patch of sky examined) tell us about $C(\theta)$ over only a limited range of angular scales. (i.e. the precise values of $C(\theta)$ for all angles from $\theta = 0$ to $\theta = 180^\circ$ is not known.) Hence, using the expansion of $\delta T/T$ in spherical harmonics, we write the correlation function as:

$$C(\theta) = \frac{1}{4\pi}\sum_{l=0}^{\infty}(2l+1)C_l P_l(cos\theta)$$

where P_l are the Legendre polynomials.

So we break down the correlation function into its multipole moments C_l, which is the angular power spectrum of the CMB.



Remember that $\lambda = 2\pi/l$. So C_l measures the amplitude as a function of wavelength. ($C_l = \frac{1}{2l+1}\sum_{m=-l}^{l}|\mathrm{a}_{lm}|^2$). In this problem, we will consider the E-mode power spectrum $C_l^{EE} = \frac{1}{2l+1}\sum_{m=-l}^{l}|\mathrm{a}_{lm}^E|^2$

THe CMB angular power spectrum is usually expressed in terms of $D_l = l(l+1)C_l/2\pi$ (in unit of μK^2) because this better shows the contribution toward the variance of the temperature fluctuations.

Cosmologists built a software called "cosmological boltzmann code" which computes the theoretical power spectrum given cosmological parameters, such as the Hubble constant and the baryon density. Therefore, we can fit the theory power spectrum to the measured one in order to obtain the best-fit parameters.

Here, we consider six selected cosmological parameters, $\vec{\theta} = [\theta_1, \theta_2, \ldots, \theta_6] = [H_0, \Omega_b h^2, \Omega_c h^2, n_s, A_s, \tau]$. ($H_0$ = Hubble constant, $\Omega_b h^2$ = physical baryon density parameter, $\Omega_c h^2$ = physical cold dark matter density parameter, n_s = scalar spectral index, A_s = curvature fluctuation amplitude, τ = reionization optical depth.). We provide you with the measured CMB E-mode power spectrum from Planck Data Release 2. Then, assuming a simple linear model of the CMB power spectrum (i.e. assuming its respose to those parameters are linear), we estimate the best-fit values of $\vec{\theta}$ using linear algebra and Gauss-Newton optimization and plot their 1-σ, 2-σ confidence regions.

Then, how do we build a linear model of the theory power spectrum? Suppose a very simple scenario where you wish to determine the best-fit value of H_0 assuming all the other parameters are already known (so $\theta = H_0$ in this case). The measurements from WMAP (the CMB satellite which preceded Planck and consequently had a lower resolution) estimate that $H_0 = 73.2$. You take it as your starting value θ_{ini} and compute the theory power spectrum there using the Boltzmann code. You also compute the derivative of D_l with respect to θ at θ_{ini}. Then, you can estimate the power spectrum as you perturb θ around θ_{ini}:

$$D_l^{model}(\theta = \theta_{ini} + \delta\theta) = D_l^{model}(\theta_{ini}) + \frac{\partial D_l}{\partial \theta}\Big|_{\theta=\theta_{ini}} \delta\theta.$$

From Planck, you get the measured D_l and error σ_l, so you can find the best-fit value of H_0 which minimizes $\chi^2 = ((D_l^{measured} - D_l^{model})/\sigma_l)^2$. Also note that you expect the best-fit values from Planck will be close to WMAP estimate, so the above linear model is a valid approximation.

Now, we can similarly build a simple linear model power spectrum with six parameters. We take $\vec{\theta}_{ini}$ as an estimate of the cosmological parameters from WMAP data (https://lambda.gsfc.nasa.gov/product/map/dr2/params/lcdm_wmap.cfm (https://lambda.gsfc.nasa.gov/product/map/dr2/params/lcdm_wmap.cfm)).

$$D_l^{model}(\vec{\theta} = \vec{\theta}_{ini} + \delta\vec{\theta}) = D_l^{model}(\vec{\theta}_{ini}) + \sum_{i=1}^{6} \frac{\partial D_l}{\partial \theta_i}\Big|_{\vec{\theta}=\vec{\theta}_{ini}} \delta\theta_i$$

So you can find the best-fit values of the above six cosmological parameters ($\vec{\theta}_{best-fit}$) which minimizes

$$\chi^2(\vec{\theta}) = \sum_{l=l_{min}}^{l_{max}} \Big(\frac{D_l^{measured} - D_l^{model}(\vec{\theta})}{\sigma_l}\Big)^2$$

(i.e. when $\vec{\theta} = \vec{\theta}_{best-fit}$, χ^2 is minimized.)

References :

http://carina.fcaglp.unlp.edu.ar/extragalactica/Bibliografia/Ryden_IntroCosmo.pdf
(http://carina.fcaglp.unlp.edu.ar/extragalactica/Bibliografia/Ryden_IntroCosmo.pdf)

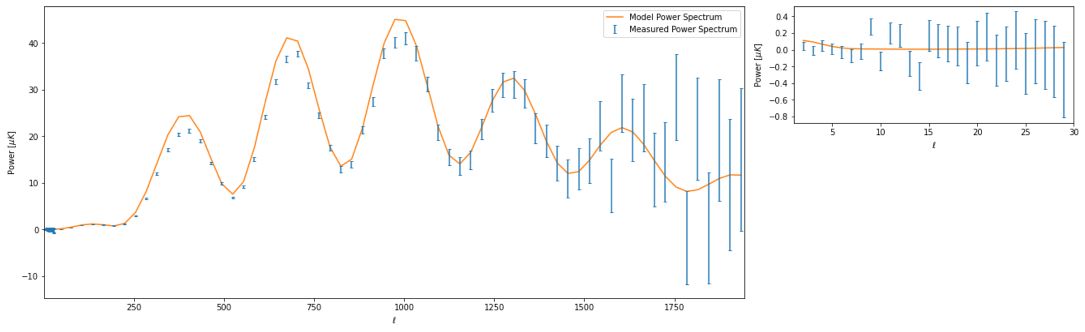
The below cell defines $l, D_l^{measured}, \sigma_l, \vec{\theta}_{ini}, D_l^{model}(\vec{\theta}_{ini}), \frac{\partial D_l}{\partial \theta_i}\Big|_{\vec{\theta}=\vec{\theta}_{ini}}$ (In this problem, we only consider the CMB E-mode power spectrum, so D_l refers to D_l^{EE}.)

Here, we set $l_{min} = 2, l_{max} = 2000$, and we have 92 l-bins in this range (For $2 \leq l < 30$, the power spectra are not binned ($\Delta l = 1$), and for $30 \leq l < 2000$, they are binned, and the bin size is $\Delta l = 30$). We obtain the measured and model power spectrum in that 92 l-bins.

In [3]:
```
# Load Data

# Measured power spectra from Planck
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_Pro
# l (same for all model and measured power spectrum)
ell = data[:,0]
# D_l^EE (measured)
EE_measured = data[:,1]
# and error sigma_l
error_EE_measured = data[:,2]

# initial estimate of the parameters (\theta_{ini}) - from https
H0     = 73.2
ombh2  = 0.02229
omch2  = 0.1054
ns     = 0.958
As     = 2.347e-9
tau    = 0.089

theta_ini = np.array([H0, ombh2, omch2, ns, As, tau])


# Model power spectra given \theta_{ini} (calculated at the same
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_Pro
# D_l^EE (model)
EE_model = data[:,1]

# Derivative of the power spectra at \theta = \theta_{ini} (calc
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_Pro
# Derivative of D_l^EE with respect to six parameters
# ([theta1, theta2, theta3, theta4, theta5, theta6] = [H_0, \Ome
deriv_DlEE_theta1 = data[:,1]
deriv_DlEE_theta2 = data[:,2]
deriv_DlEE_theta3 = data[:,3]
deriv_DlEE_theta4 = data[:,4]
deriv_DlEE_theta5 = data[:,5]
deriv_DlEE_theta6 = data[:,6]
```

1. Plot the measured power spectrum with errorbar. Also, plot the model power spectrum on top, by interpolating between the data points. You should find that the data from Planck does not fit to the model very well. To better see the low-l measurements, also plot both spectra in the range $2 \leq l < 30$. Remember that the power spectra D_l have units of μK^2. Don't forget to label all plots.

In [4]:
```
low_ell_ix = np.where(ell < 30)[0]

fig = plt.figure(figsize=(12, 5))
ax1 = fig.add_axes([0., 0., 1., 1.])
ax1.errorbar(
    ell,
    EE_measured,
    yerr=error_EE_measured,
    fmt="none",
    capsize=2,
    label="Measured Power Spectrum",
)
ax1.plot(ell, EE_model, label="Model Power Spectrum", zorder=0)
ax1.set_xlim(ell.min(), ell.max()+10)
ax1.legend()

ax2 = fig.add_axes([1.07, 0.6, .4, .4])
ax2.errorbar(
  ell[low_ell_ix],
  EE_measured[low_ell_ix],
  yerr=error_EE_measured[low_ell_ix],
  fmt="none",
  capsize=2,
)
ax2.plot(ell[low_ell_ix], EE_model[low_ell_ix], zorder=0)
ax2.set_xlim(ell[low_ell_ix].min()-1, ell[low_ell_ix].max()+1)

plt.setp([ax1, ax2], ylabel="Power [$\mu K$]", xlabel="$\ell$")
plt.show()
```



2. Using the techniques from linear algebra (normal equations, SVD, etc), find the best-fit cosmological parameters ($\vec{\theta}_{best-fit}$). Print $\vec{\theta}_{best-fit}$.

Take a look at the undergrad version for hints

We need to minimize

$$\chi^2(\vec{\theta}) = \sum_{l=l_{min}}^{l_{max}} \left(\frac{D_l^{measured} - D_l^{model}(\vec{\theta})}{\sigma_l}\right)^2 = \sum_{l=l_{min}}^{l_{max}} \left(\frac{D_l^{measured} - D_l^{model}(\overrightarrow{\theta_{\text{ini}}}) -}{\sigma_l}\right.$$

Thus, we are fitting for the linear parameters $\{\delta\theta_i\}_{i=1}^{6}$.

Define the following variables:

$$a_i \equiv \delta\theta_i$$
$$b_l \equiv \frac{D_l^{measured} - D_l^{model}(\overrightarrow{\theta_{\text{ini}}})}{\sigma_l}$$
$$A_{li} \equiv \frac{1}{\sigma_l}\frac{\partial D_l}{\partial \theta_i}\Big|_{\vec{\theta}=\vec{\theta}_{ini}}$$

Minimizing χ^2 is thus equivalent to minimizing $|\mathbf{A}\cdot\mathbf{a}-\mathbf{b}|^2$.

We will solve this by decomposing $\mathbf{A}$ with SVD. In Homework 3, we showed that the solution is then given by:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$
$$\mathbf{a} = \sum_{c=1}^{M}\left(\frac{\mathbf{U}_c\cdot\mathbf{b}}{\mathbf{S}_{c,c}}\right)\mathbf{V}_c,$$

where the subscript c denotes column c.

The covariance between parameters in this case is:
$$\mathrm{Cov}(a_j, a_k) = \sum_{i=1}^{M}\frac{V_{ji}V_{ki}}{S_{ii}^2}.$$

We note that the parameters of interest are $\theta_i = \theta_{\text{ini,i}} + \delta\theta_i$. Thus, after obtaining the $\delta\theta_i$ we can simply add $\theta_{\text{ini,i}}$ to recover the best-fit parameters. The covariances do not change, since $\theta_{\text{ini,i}}$ are constants.

In [66]:

```python
b = (EE_measured - EE_model) / error_EE_measured
A = np.array([
    deriv_DlEE_theta1,
    deriv_DlEE_theta2,
    deriv_DlEE_theta3,
    deriv_DlEE_theta4,
    deriv_DlEE_theta5,
    deriv_DlEE_theta6,
]).T
A /= error_EE_measured[:, None]

# SVD
U, S, Vt = np.linalg.svd(A)
V = Vt.T

# best fit parameters (delta theta)
M = len(S)
a = np.zeros(M)
for i in range(M):
  a += (U[:, i] @ b / S[i]) * V[:, i]

# variance
s = (V / S) ** 2   # summand
sigma_a = np.sqrt(s.sum(axis=1))

# add back theta_ini
theta_best = a + theta_ini

params = [
    "H0", "ombh2", "omch2", "ns", "As", "tau"
]

print("Best fit parameters:")
for i, p in enumerate(params):
  print(f"{p} = {theta_best[i]:.3g} +- {sigma_a[i]:.3g}")
```

```
Best fit parameters:
H0 = 68.9 +- 2.77
ombh2 = 0.0236 +- 0.00117
omch2 = 0.114 +- 0.00433
ns = 0.978 +- 0.0136
As = 2.18e-09 +- 6.36e-11
tau = 0.0702 +- 0.0122
```

We can define $\chi^2(\vec{\theta}) = \vec{r}(\vec{\theta})^T\ \vec{r}(\vec{\theta})$ where $\vec{r} = \frac{D_l^{model}\ (\vec{\theta})\ -D_l^{measured}}{\sigma_l}$. (so $\vec{r}$ is a vector of length 92.)

We then compute the gradient and the Hessian of χ^2 to apply the Gauss-Newton method. (This is a linear least squares problem, so using the Gauss-Newton method is equivalent to using the normal equations in this case - yes, you may think that it is silly to use the Gauss-Newton here! So we are expected to reach minimum just after one iteration. The Jacobian J is identical to the design matrix A.)

The jth component of the gradient is:

$$(\nabla\chi^2(\vec{\theta}))_j = 2\sum_l r_l(\vec{\theta})\frac{\partial r_l}{\partial\theta_j}(\vec{\theta})$$

where $\frac{\partial r_l}{\partial \theta_j} = \frac{1}{\sigma_l}\frac{\partial D_l}{\partial \theta_j}\Big|_{\vec{\theta}=\vec{\theta}_{ini}}$

Now, the Jacobian matrix $J(\vec{\theta})$ is:

$$\begin{bmatrix} \frac{\partial r_{l_{min}}}{\partial \theta_1}(\vec{\theta}) & \dots & \frac{\partial r_{l_{min}}}{\partial \theta_6}(\vec{\theta}) \\ \dots. & \dots. & \dots. \\ \frac{\partial r_{l_{max}}}{\partial \theta_1}(\vec{\theta}) & \dots & \frac{\partial r_{l_{max}}}{\partial \theta_6}(\vec{\theta}) \end{bmatrix}$$

The gradient of χ^2 can be written as:

$$\nabla \chi^2(\vec{\theta}) = 2J(\vec{\theta})^T \vec{r}(\vec{\theta})$$

Similarly, the (i,j)th component of the Hessian matrix of χ^2 is given by:

$$\frac{\partial^2(\chi^2)}{\partial \theta_i \partial \theta_j}(\vec{\theta}) = 2\sum_l \frac{\partial r_l}{\partial \theta_i}(\vec{\theta})\frac{\partial r_l}{\partial \theta_j}(\vec{\theta})$$

(Here, $\frac{\partial^2 r_l}{\partial \theta_i \partial \theta_j} = 0$)

Because our model power spectrum is linear, we can write the Hessian matrix simply as $H(\vec{\theta}) = 2J(\vec{\theta})^T J(\vec{\theta})$.

Then, using Newton's method, we can find $\vec{\theta}$ which minimizes χ^2:

$$\vec{\theta}^{(k+1)} = \vec{\theta}^{(k)} - \left(J(\vec{\theta}^{(k)})^T J(\vec{\theta}^{(k)})\right)^{-1} J(\vec{\theta}^{(k)})^T \vec{r}(\vec{\theta}^{(k)})$$

We have a simple linear model in this case, so we are expected to reach the minimum after one step.

3. Using the Gauss-Newton optimization, find the best-fit parameters ($\vec{\theta}_{best-fit}$). Iterate until you reach the minimum. (Yes, you are expected to converge to the minimum after one step. So show that this is the case indeed.) Does your result agree with Part 2?

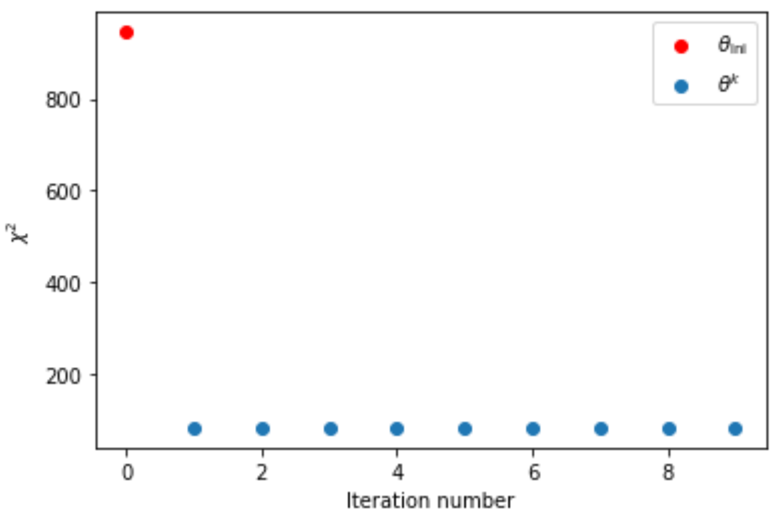
It appears from the formula above that we need to compute the Jacobian matrix for each iteration (at each θ^k). However, we note that the elements of the Jacobian are of the form $\frac{1}{\sigma_l}\frac{\partial D_l}{\partial \theta_j}\Big|_{\vec{\theta}=\vec{\theta}^k}$ and that D_l is a linear model. Hence, the derivative is constant (by definition) and the Jacobian does not change with θ.

In [67]:

```python
class GaussNewton:
  def __init__(self, niter=10):
    # compute the ,matrix that updates theta
    self.jacobian = A.copy()
    self.update_matrix = (
        np.linalg.inv(self.jacobian.T @ self.jacobian) @ self.ja
    )
    self.theta_ini = theta_ini  # initial params
    self.model_ini = EE_model  # model at theta_ini
    self.spec_measured = EE_measured  # measured spectrum
    self.sigma_l = error_EE_measured
    self.niter = niter  # max. number of iterations
    self.theta = np.empty((self.niter+1, len(self.theta_ini)))
    self.residuals = np.empty((self.niter+1, len(self.jacobian))

  def ee_model_theta(self, theta):
    """
    Evaluate the EE model at given theta, using the linear model
    """
    delta_theta = theta - self.theta_ini
    delta_model = (self.jacobian * error_EE_measured[:, None]) @
    return self.model_ini + delta_model

  def res(self, theta):
    """
    Compute r vector
    """
    m = self.ee_model_theta(theta)
    r = (m - self.spec_measured) / self.sigma_l
    return r


  def update_theta(self, theta_prev, res_prev):
    """
    Updates theta with Gauss-Newton optimization given theta and
    residual vector r.
    """
    theta_new = theta_prev - self.update_matrix @ res_prev
    return theta_new

  def run(self):
    """
    Run the GN optimization
    """
    self.theta[0] = self.theta_ini
    for k in range(1, self.niter+1):
      r = self.res(self.theta[k-1])
      self.residuals[k-1] = r
      self.theta[k] = self.update_theta(self.theta[k-1], r)
```

In [68]:
```
# run the GaussNewton algorithm
gn = GaussNewton(niter=10)
gn.run()

# compute and plot chi squared vs iteration
chi_sq = np.sum(gn.residuals**2, axis=1)
plt.figure()
plt.scatter(0, chi_sq[0], c="red", label="$\\theta_{\mathrm{ini}
plt.scatter(np.arange(len(chi_sq[1:]))+1, chi_sq[1:], label="$\\
plt.legend()
plt.xlabel("Iteration number")
plt.ylabel("$\\chi^2$")
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:6:
```



In [69]:
```
print(np.allclose(gn.theta[1], gn.theta[2:]))
```

```
True
```

We see that χ^2 reaches its minimum at the first iteration in the plot above and that θ^k remains unchanged after the first step. This shows that the algorithm converges in one step.

In [70]:

```
# best fit parameters
gn_theta_best = gn.theta[1]

# the parameters are the same as before
assert np.allclose(gn_theta_best, theta_best)

print("Best fit parameters:")
for i, p in enumerate(params):
  print(f"{p} = {gn_theta_best[i]:.3g}")
```

```
Best fit parameters:
H0 = 68.9
ombh2 = 0.0236
omch2 = 0.114
ns = 0.978
As = 2.18e-09
tau = 0.0702
```

We can compute the covariance matrix as $\left(J(\vec{\theta})^T J(\vec{\theta})\right)^{-1}$. (Remember that the covariance matrix in the normal equations is $(A^T\ A)^{-1}$ where A is the design matrix.) From this, we can plot 1-d and 2-d constraints on the parameters. (See Fig. 6 in Planck 2015 paper https://arxiv.org/pdf/1502.01589v3.pdf (https://arxiv.org/pdf/1502.01589v3.pdf))

1-d constraint (corresponding to the plots along the diagonal in Fig. 6, Planck 2015 paper):

First, the ith diagonal element of the covariance matrix correspond to $\sigma(\theta_i)^2$. Then, we can plot 1-d constraints on the parameter θ_i assuming a normal distribution with mean = $(\vec{\theta}_{best-fit})_i$ and variance = $\sigma(\theta_i)^2$.

2-d constraint (off-diagonal plots in Fig. 6, Planck 2015 paper):

Consider two parameters θ_i and θ_j from $\vec{\theta}$. Now marginalize over other parameters - in order to marginalize over other parameters, you can simply remove those parameters' row and column from the full covariance matrix. (i.e. From the full covariance matrix, you know the variance of all six parameters and their covariances with each other. So build a smaller dimension - 2 x 2 - covariance matrix from this.) - and obtain a 2×2 covariance matrix:

$$\mathrm{C_{ij}} = \begin{pmatrix} \sigma(\theta_i)^2 & \mathrm{Cov}(\theta_i, \theta_j) \\ \mathrm{Cov}(\theta_i, \theta_j) & \sigma(\theta_j)^2 \end{pmatrix}$$

Now, we can plot the 2-dimensional confidence region ellipses from this matrix. The lengths of the ellipse axes are the square root of the eigenvalues of the covariance matrix, and we can calculate the counter-clockwise rotation of the ellipse with the rotation angle:

$$\phi = \frac{1}{2}\arctan\Big(\frac{2 \cdot \mathrm{Cov}(\theta_i, \theta_j)}{\sigma(\theta_i)^2 - \sigma(\theta_j)^2}\Big) = \arctan(\frac{\overrightarrow{v_1}(y)}{\overrightarrow{v_1}(x)})$$

where $\overrightarrow{v_1}$ is the eigenvector with the largest eigenvalue. So we calculate the angle of the largest eigenvector towards the x-axis to obtain the orientation of the ellipse.

Then, we multiply the axis lengths by some factor depending on the confidence level we are interested in. For 68%, this scale factor is $\sqrt{\Delta\chi^2} \approx 1.52$. For 95%, it is $\sqrt{\Delta\chi^2} \approx 2.48$.

4. Plot 1-d and 2-d constraints on the parameters. For 2-d plot, show 68% and 95% confidence ellipses for each pair of parameters. You can arrange those subplots in a triangle shape, as in Fig. 6, Planck 2015 (https://arxiv.org/pdf/1502.01589v3.pdf).

In [71]:
```
# covariance matrix according to formula above
cov = np.linalg.inv(gn.jacobian.T @ gn.jacobian)
```

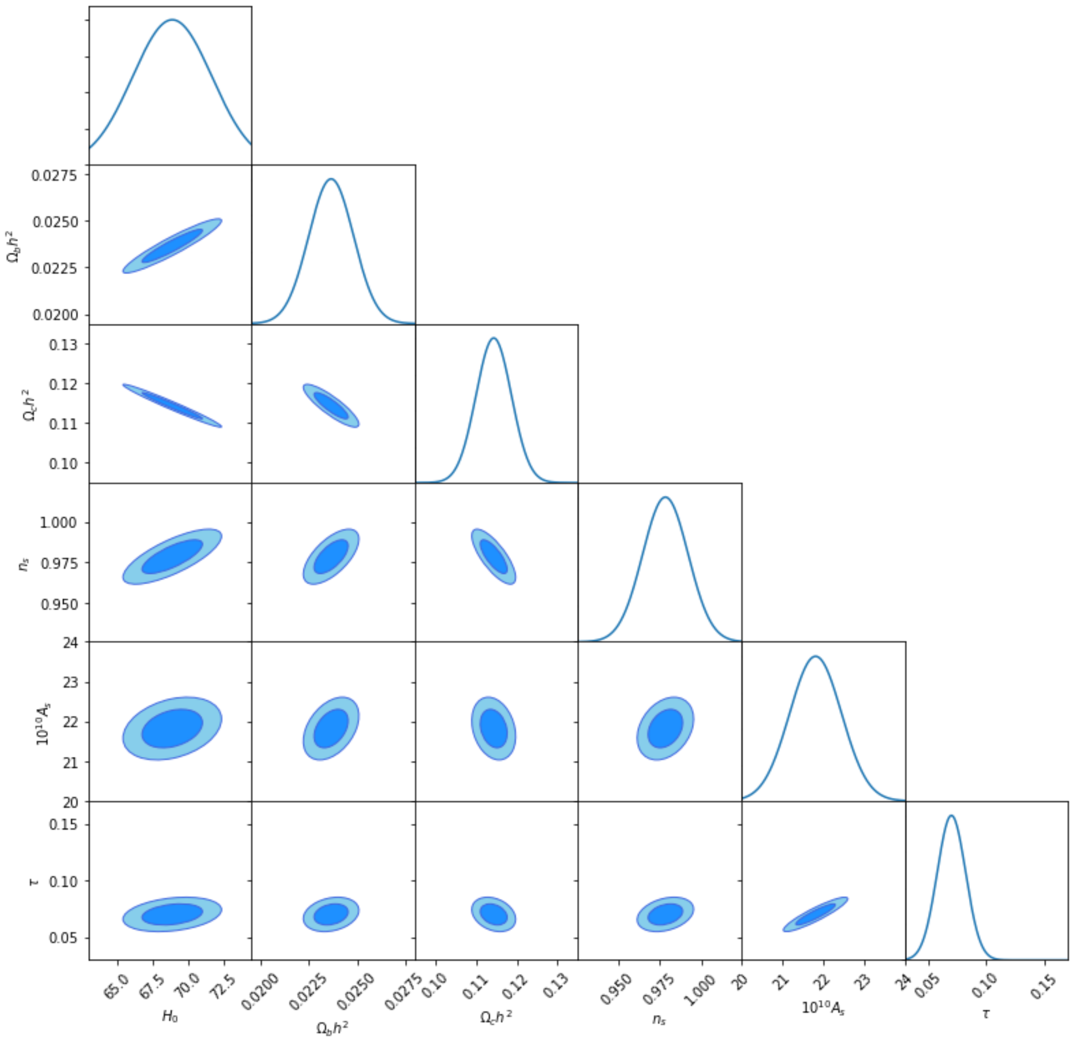
In [72]:

```
# we reuse the code from last week


"""
Note: We rescale A_s by 10^10 since it has a very small value. T
plotting a bit nicer. We can do this since none of the probabili
shapes are changed (we've only done linear transformations to ar
probability distributions)
"""

AS_RESCALE = 1e10

from matplotlib.patches import Ellipse

def gauss(x, mean, variance):
  """
  Normal distribution given mean and variance
  """
  a = 1 / np.sqrt(2 * np.pi * variance)
  z = (x - mean)**2 / variance
  return  a * np.exp(-z / 2)

params_latex = [
    "$H_0$",
    "$\\Omega_b h^2$",
    "$\\Omega_c h^2$",
    "$n_s$",
    "$10^{10}A_s$",
    "$\\tau$",
]



def plot_1d(ax, mean, variance, xmin, xmax, color=None, label=No
  """
  Helper function to plot 1d normal distribution
  """
  xgrid = np.linspace(xmin, xmax, num=200)
  y = gauss(xgrid, mean, variance)
  y /= y.max()  # divide out amplitude in plot since units are n
  ax.plot(xgrid, y, color=color, label=label)
  ax.set_xlim(xmin, xmax)
  ax.set_ylim(0., 1.1)

def confidence(cov_matrix, interval):
  """
  Construct confidence ellipse from cov matrix and confidence in
  """
  eigvec, eigval, u = np.linalg.svd(cov_matrix)
  # Semimajor axis (diameter)
  semimaj = np.sqrt(eigval[0])
  # Semiminor axis (diameter)
  semimin = np.sqrt(eigval[1])
  # theta
  theta = np.arctan2(eigvec[0, 1], eigvec[0, 0])

  if interval == 0.68:
    factor = 1.52
  elif interval == 0.95:
    factor = 2.48
  semimaj *= factor
  semimin *= factor

  return semimaj, semimin, theta


def corner(best_params, cov_matrix, bounds, cov_matrix2=None):
  """
  Make corner plot given a covariance matrix. Optionally, constr
  a 2nd covariance matrix can be overplotted.

  Parameters
  ----------
  best_params : dict
    The best fit parameters (mean of normal distribution). Key i
    name, value is the best fit value.

  cov_matrix : np.ndarray
    Covariance matrix.

  cov_matrix2 : np.ndarray
    2nd covariance matrix.
  """
  NROWS = NCOLS = len(params) + 1
  fig = plt.figure(figsize=(15, 15))
  axs = []  # diagonal axes

  # plot the 1d constraints along the diagonal
  for i, par in enumerate(best_params):
    if i == 0:
      ax = fig.add_subplot(NROWS, NCOLS, i * (NCOLS + 1) + 1)
    else:
      ax = fig.add_subplot(NROWS, NCOLS, i * (NCOLS + 1) + 1, sh
    axs.append(ax)

    mean = best_params[par]
    var = cov_matrix[i, i]

    if par == "As":
      mean *= AS_RESCALE  # see note on top of cell
      var *= AS_RESCALE ** 2

    # parameter bounds
    xmin, xmax = bounds[par]
    plot_1d(ax, mean, var, xmin, xmax, color="C0", label="PLANCK

    if cov_matrix2 is not None:
      var2 = cov_matrix2[i, i]
      if par == "As":
        var2 *= AS_RESCALE ** 2
      plot_1d(ax, mean, var2, xmin, xmax, color="C3", label="No

      if i == 0:
        ax.legend(loc="upper right", bbox_to_anchor=(3., 1.), fo

    # labels and ticks
    ax.locator_params(axis="x", nbins=4, min_n_ticks=3)
    plt.setp(ax.get_yticklabels(), visible=False)
    if i == len(best_params)-1:
```

```
      ax.set_xlabel(params_latex[i])
      visible = True
    else:
      visible = False
    plt.setp(ax.get_xticklabels(), visible=visible, rotation=45)


  # 2d constraints
  covMs = [cov_matrix]
  if cov_matrix2 is not None:
    covMs.append(cov_matrix2)

  # ellipse colors
  ELL_FACE_COLORS = ["dodgerblue", "skyblue", "firebrick", "ligh
  ELL_EDGE_COLORS = ["royalblue", "red"]

  for i in range(NROWS-1):
    for j in range(i+1, NCOLS-1):

      ax = fig.add_subplot(NROWS, NCOLS, j * NCOLS + i + 1, shar
      # set limits on y axes to be same as x axis of the same pa
      ypar = list(best_params.keys())[j]
      ymin, ymax = bounds[ypar]
      ax.set_ylim(ymin, ymax)
      ax.locator_params(axis="y", nbins=4, min_n_ticks=3)

      # best parameter values
      p1 = list(best_params.values())[i]
      p2 = list(best_params.values())[j]

      if i == 4:  # As, see note on top of cell
        p1 *= AS_RESCALE
      if j == 4:
        p2 *= AS_RESCALE

      for k, cov in enumerate(covMs):  # loop over cov matrices
        # 2x2 cov matrix
        cov_22 = np.zeros((2, 2))
        cov_22[0, 0] = cov[i, i]
        cov_22[0, 1] = cov_22[1, 0] = cov[i, j]
        cov_22[1, 1] = cov[j, j]

        if i == 4:  # As, see note on top of cell
          cov_22[0, 0] *= AS_RESCALE ** 2
          cov_22[0, 1] *= AS_RESCALE
          cov_22[1, 0] *= AS_RESCALE

        if j == 4:
          cov_22[1, 1] *= AS_RESCALE ** 2
          cov_22[0, 1] *= AS_RESCALE
          cov_22[1, 0] *= AS_RESCALE

        semimaj, semimin, theta = confidence(cov_22, .68)
        fc = ELL_FACE_COLORS[2*k]
        ec = ELL_EDGE_COLORS[k]
        ell = Ellipse(
            xy=[p1, p2],
            width=semimaj,
            height=semimin,
            angle=theta*180/np.pi,
            facecolor=fc,
            edgecolor=ec,
        )

        semimaj, semimin, theta = confidence(cov_22, .95)
        fc = ELL_FACE_COLORS[2*k+1]
        ell2 = Ellipse(
            xy=[p1, p2],
            width=semimaj,
            height=semimin,
            angle=theta*180/np.pi,
            facecolor=fc,
            edgecolor=ec,
        )

        # add ellipses
        ax.add_patch(ell2)
        ax.add_patch(ell)

      if i == 0:
        ax.set_ylabel(params_latex[j])
        visible = True
      else:
        visible = False
      plt.setp(ax.get_yticklabels(), visible=visible)

      if j == NCOLS-2:
        ax.set_xlabel(params_latex[i])
        visible = True
      else:
        visible = False
      plt.setp(ax.get_xticklabels(), visible=visible, rotation=4

  plt.subplots_adjust(wspace=0, hspace=0)
  plt.show()
```

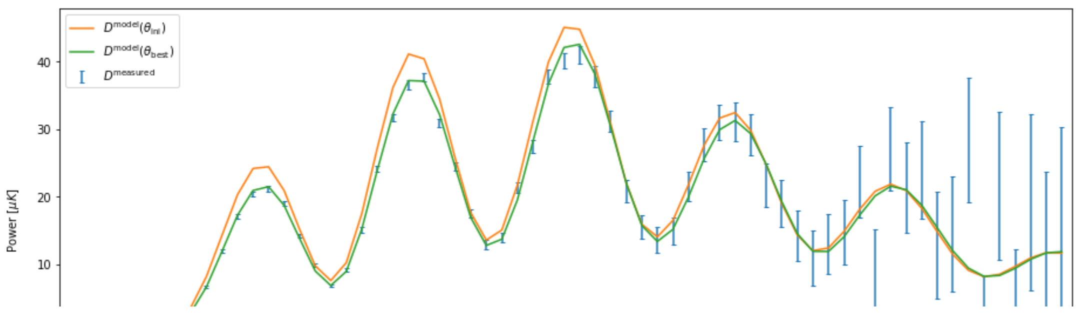
In [77]:

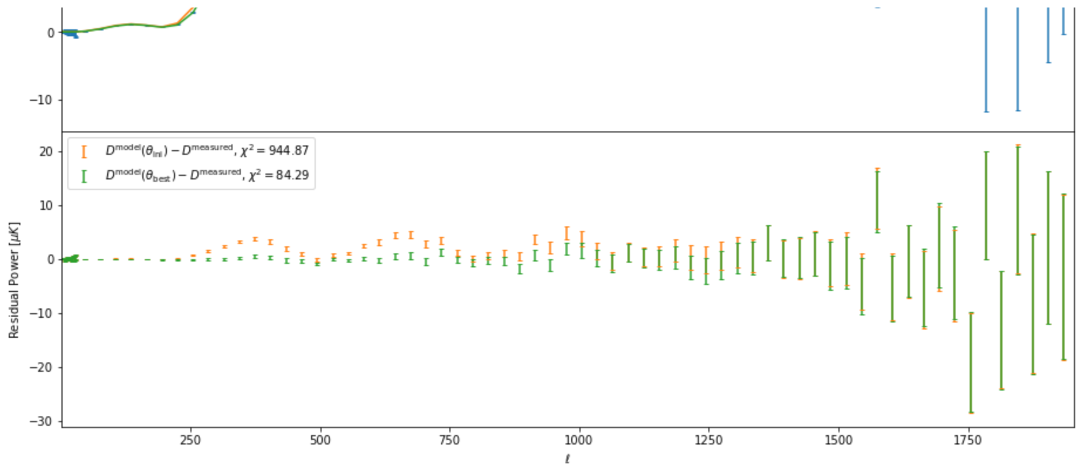
```
# make a dictionary with theta_best from the gauss newton optimi
best_params = {}
for i in range(len(params)):
  key = params[i]
  value = gn_theta_best[i]
  best_params[key] = value

# parameter bounds to plot, aiming to (somewhat) match fig6 in t
BOUNDS = {
    "H0": (63., 74.4),
    "ombh2": (0.0195, 0.028),
    "omch2": (0.095, 0.135),
    "ns": (0.926, 1.024),
    "As": (20, 24),  # the paper plots the log
    "tau": (0.03, 0.17),
}

corner(best_params, cov, BOUNDS)
```



5. Plot $D_l^{model}(\vec{\theta} = \vec{\theta}_{ini})$ and $D_l^{model}(\vec{\theta} = \vec{\theta}_{best-fit})$ as well as $D_l^{measured}$ with errorbar. Show that with the best-fit parameters you obtained, the model power spectrum fits better to the measured data.

In [102]:
```
# D^model at best fit theta
best_model = gn.ee_model_theta(gn_theta_best)

fig = plt.figure(figsize=(12, 5))
ax1 = fig.add_axes([0., 0., 1., 1.])
ax1.errorbar(
    ell,
    EE_measured,
    yerr=error_EE_measured,
    fmt="none",
    capsize=2,
    label="$D^{\mathrm{measured}}$",
)
ax1.plot(ell, EE_model, label="$D^{\mathrm{model}} (\\theta_{\ma
ax1.plot(ell, best_model, label="$D^{\mathrm{model}} (\\theta_{\
ax1.set_xlim(ell.min(), ell.max()+20)
ax1.set_ylabel("Power [$\mu K$]")
ax1.legend()

ax2 = fig.add_axes([0., -.7, 1., .7])
ax2.errorbar(
    ell,
    EE_model - EE_measured,
    yerr=error_EE_measured,
    fmt="none",
    capsize=2,
    c="C1",
    label=(
        "$D^{\mathrm{model}} (\\theta_{\mathrm{ini}}) - D^{\math
        f", $\chi^2 = {chi_sq[0]:.2f}$"
    ),
)
ax2.errorbar(
    ell,
    best_model - EE_measured,
    yerr=error_EE_measured,
    fmt="none",
    capsize=2,
    c="C2",
    label=(
        "$D^{\mathrm{model}} (\\theta_{\mathrm{best}}) - D^{\mat
        f", $\chi^2 = {chi_sq[1]:.2f}$"
    ),
)
ax2.set_xlim(ell.min(), ell.max()+20)
ax2.set_ylabel("Residual Power [$\mu K$]")
ax2.set_xlabel("$\ell$")
ax2.legend()

plt.show()
```





Problem 2 - Optimization

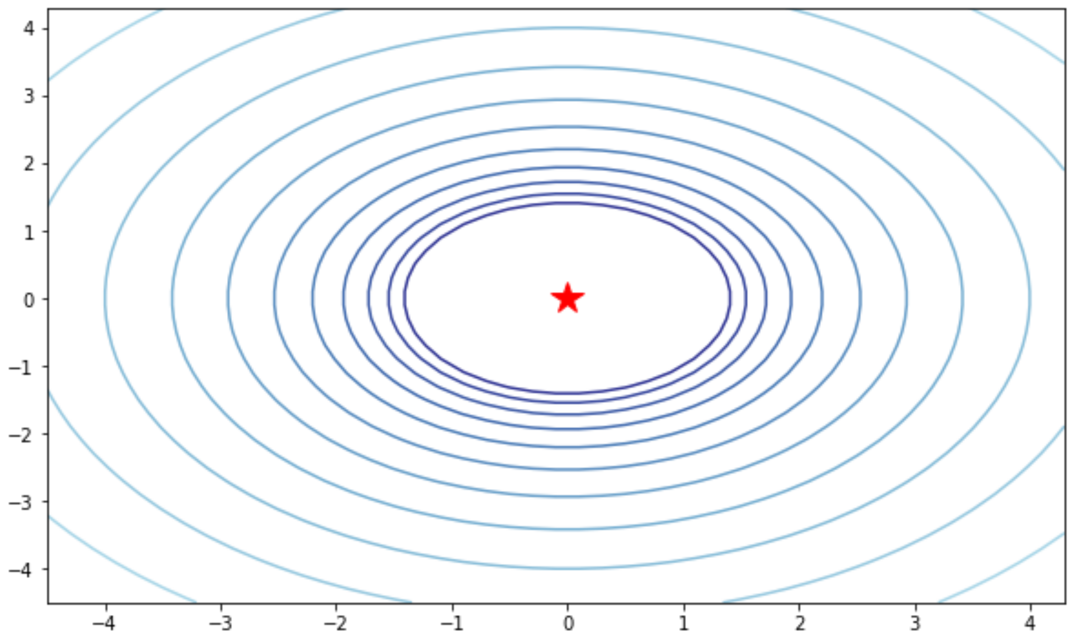
Consider the surface $z = ax^2 + by^2 - 1$. Let $a = 1$ and $b = 1$ and make the contour plot. The global minimum of this surface is at $(x, y) = (0, 0)$ (marked by a star).

In [103]:
```
def minima_surface(x,y,a,b):
    return a*x**2+b*y**2-1
```

In [104]:
```
x, y = np.meshgrid(np.arange(-4.5, 4.5, 0.2), np.arange(-4.5, 4.
fig, ax = plt.subplots(figsize=(10, 6))
z=minima_surface(x,y,1,1)
ax.contour(x, y, z, levels=np.logspace(0, 5, 35), norm=LogNorm()
ax.plot(0,0, 'r*', markersize=18)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4:
  after removing the cwd from sys.path.
```

Out[104]:
```
[<matplotlib.lines.Line2D at 0x7f330ed8f3d0>]
```



Now, let us visualize what different gradient descent methods do. We will be especially interested in trying to understand how various hyperparameters. especially the learning rate η, affect our performance.

We denote the parameters by θ and the energy function we are trying to minimize by $E(\theta)$.

First, consider a $\mathbf{simple\ gradient\ descent\ method}$. In this method, we will take steps in the direction of the local gradient. Given some parameters θ, we adjust the parameters at each iteration so that

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta E(\theta_t)$$

where we have introduced the learning rate η_t that controls how large a step we take. In general, the algorithm is extremely sensitive to the choice of η_t. If η_t is too large, then one can wildly oscillate around minima and miss important structure at small scales. If η_t is too small, then the learning/minimization procedure becomes extremely slow. This raises the natural question: What sets the natural scale for the learning rate and how can we adaptively choose it?

1. Run gradient descent on the surface $z = x^2 + y^2 - 1$. Start from an inital point $(-2, 4)$ and plot trajectories for $\eta = 0.1, 0.5, 1$. Take 100 steps. Do you see different behaviors that arise as η increases? (The trajectory converges to the global minima in multiple steps for small learning rates (η = 0.1). Increasing the learning rate to 1 causes the trajectory to oscillate around the global minima before converging.) Make sure to lable each plot.

Hint:

In this problem, $\theta = (x, y)$, and $E(\theta) = x^2 + y^2 - 1$, i.e. find (x, y) which minimizes $x^2 + y^2 - 1$. Hence, $\nabla_\theta E(\theta) = (2x, 2y)$.

$$(x, y)_{t+1} = (x, y)_t - \eta_t \cdot (2x, 2y)_t$$

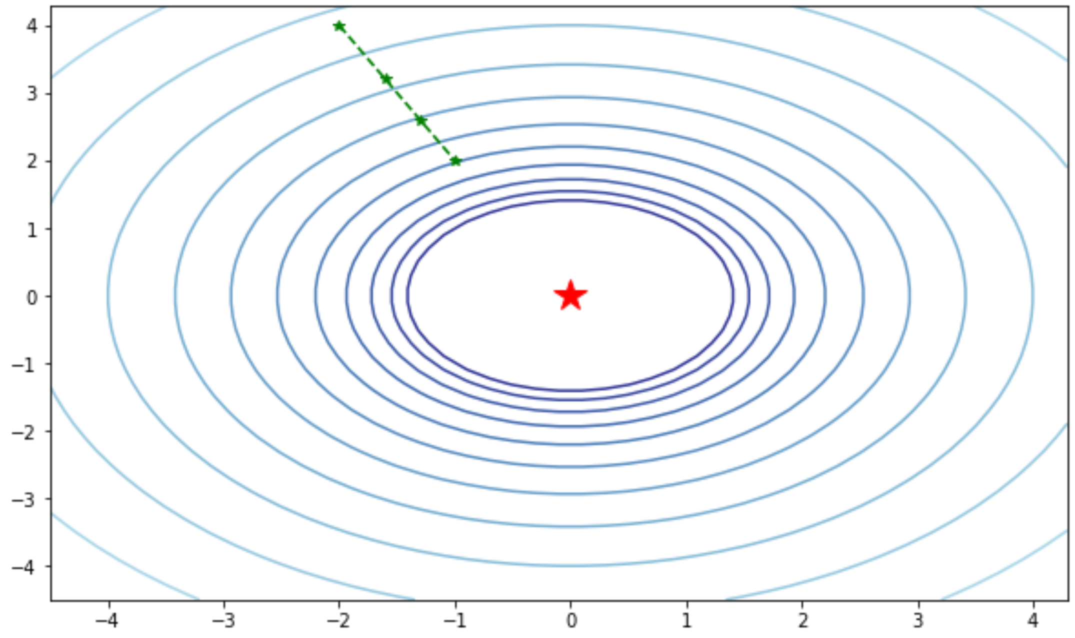
It is given that $(x, y)_{t=0} = (-2, 4)$.

Suppose that you take 3 steps and obtain $(x, y)_{t=1} = (-1.6, 3.2), (x, y)_{t=2} = (-1.3, 2.6), (x, y)_{t=3} = (-1.0, 2.0)$. Then, you can plot this trajectory in the following way:

In [105]:

```
x, y = np.meshgrid(np.arange(-4.5, 4.5, 0.2), np.arange(-4.5, 4.
fig, ax = plt.subplots(figsize=(10, 6))
z=minima_surface(x,y,1,1)
ax.contour(x, y, z, levels=np.logspace(0, 5, 35), norm=LogNorm()
ax.plot(0,0, 'r*', markersize=18)

Nsteps = 3
trajectory = np.zeros([Nsteps+1,2])
trajectory[0,:] = [-2,4]; trajectory[1,:] = [-1.6,3.2]; trajecto

xs=trajectory[:,0]
ys=trajectory[:,1]
ax.plot(xs,ys, 'g--*')

plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: 
  after removing the cwd from sys.path.
```



In this problem, the simple gradient descent method is defined by

$$(x, y)_{t+1} = (x, y)_t - \eta_t \cdot (2x, 2y)_t = (1 - 2\eta_t) \cdot (x, y)_t$$

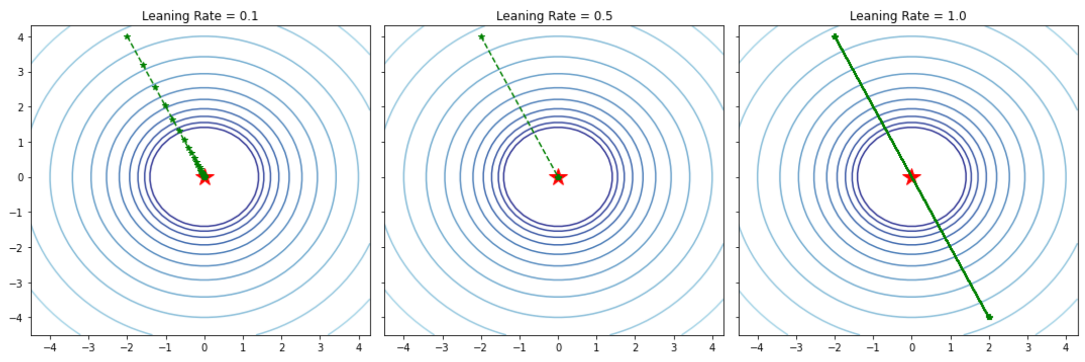
In [128]:

```
def grad(theta):
  """
  Gradient of the surface defined above
  """
  return 2 * theta

def simple_gd(initial_theta, gradient, learning_rate, niter):
  """
  Do simple gradient descent given iniital set of parameters, gr
  learning rate, and number of iterations

  Gradient is a function that returns the gradient at the given
  """
  thetas = np.empty((niter+1, len(initial_theta)))
  thetas[0] = initial_theta
  for i in range(niter):
    thetas[i+1] = thetas[i] - learning_rate * gradient(thetas[i]
  return thetas
```

In [129]:

```
x, y = np.meshgrid(np.arange(-4.5, 4.5, 0.2), np.arange(-4.5, 4.
fig, axs = plt.subplots(figsize=(15, 5), ncols=3, sharex=True, s
z = minima_surface(x, y, 1, 1)
levels = np.logspace(0, 5, 35)

for i, eta in enumerate([.1, .5, 1.]):  # loop over learning rat
  ax = axs[i]
  ax.contour(x, y, z, levels=levels, norm=LogNorm(), cmap="RdYlB
  ax.plot(0,0, 'r*', markersize=18)
  trajectory = simple_gd((-2, 4), grad, eta, 100)
  xs = trajectory[:,0]
  ys = trajectory[:,1]
  ax.plot(xs,ys, "g--*")
  ax.set_title(f"Leaning Rate = {eta:.1f}")

plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:8:

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:8:

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:8:
```



We see that a lot of steps are needed when the learning rate is 0.1, especailly as we approach the minimum. When the learning rate is 0.5, the convergence happens in just one step (the steps are marked by green stars). When the learning rate is 1., the best fit values jumps between two extremes and do not converge at the minimum.

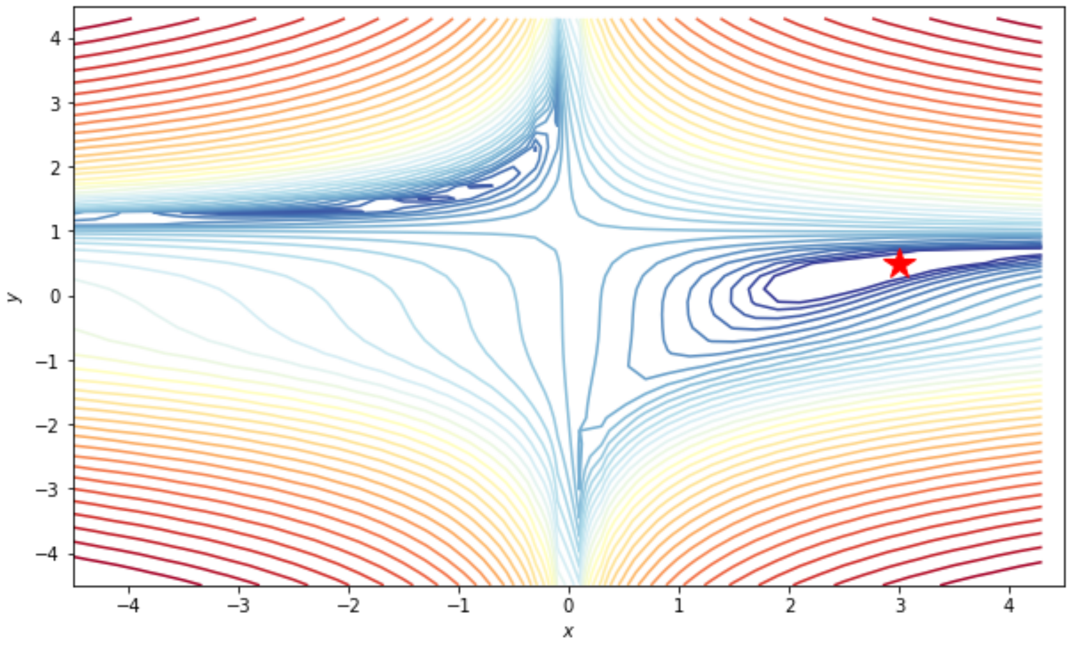
Next, take Beale's Function, a convex function often used to test optimization problems of the form:

$$z = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.6250 - x + xy^3)^2$$

The global minimum of this function is at $(3, 0.5)$ (marked by a star).

In [119]:
```
def beales_function(x,y):
    f=np.square(1.5-x+x*y)+np.square(2.25-x+x*y*y)+np.square(2.6
    return f

def contour_beales_function():
    #plot beales function
    x, y = np.meshgrid(np.arange(-4.5, 4.5, 0.2), np.arange(-4.5
    fig, ax = plt.subplots(figsize=(10, 6))
    z=beales_function(x,y)
    ax.contour(x, y, z, levels=np.logspace(0, 5, 35), norm=LogNo
    ax.plot(3,0.5, 'r*', markersize=18)

    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')

    ax.set_xlim((-4.5, 4.5))
    ax.set_ylim((-4.5, 4.5))

    return fig,ax
```

In [120]:
```
fig,ax =contour_beales_function()
plt.show()
```



One problem with gradient descent is that it has no memory of where it comes from. This can be an issue when there are many shallow minima in our landscape. If we make an analogy with a ball rolling down a hill, the lack of memory is equivalent to having has no inertia or momentum (i.e. completely overdamped dynamics). Without momentum, the ball has no kinetic energy and cannot climb out of shallow minima. Then, we can add a memory or momentum term to the stochastic gradient descent term above:

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta E(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_t$$

with $0 \leq \gamma < 1$ called the momentum parameter. When $\gamma = 0$, this reduces to ordinary gradient descent, and increasing γ increases the inertial contribution to the gradient. From the equations above, we can see that typical memory lifetimes of the gradient is given by $(1-\gamma)^{-1}$. For $\gamma = 0$ as in gradient descent, the lifetime is just one step. For $\gamma = 0.9$, we typically remember a gradient for ten steps. We call this method $\textbf{gradient descent with momentum}$.

In stochastic gradient descent, with and without momentum, we still have to specify a schedule for tuning the learning rates η_t as a function of time. In the context of Newton's method, this presents a number of dilemmas. The learning rate is limited by the steepest direction which can change depending on where in the landscape we are. To circumvent this problem, ideally our algorithm would take large steps in shallow, flat directions and small steps in steep, narrow directions. Second-order methods accomplish this by calculating or approximating the Hessian and normalizing the learning rate by the curvature. However, this is very computationally expensive for extremely large models. Ideally, we would like to be able to adaptively change our step size to match the landscape without paying the steep computational price of calculating or approximating Hessians.

Recently, a number of methods have been introduced that accomplish this by tracking not only the gradient but also the second moment of the gradient. These methods include AdaGrad, AdaDelta, RMS-Prop, and ADAM.

In $\textbf{ADAM}$, we keep a running average of both the first and second moment of the gradient and use this information to adaptively change the learning rate for different parameters. In addition to keeping a running average of the first and second moments of the gradient, ADAM performs an additional a bias correction to account for the fact that we are estimating the first two moments of the gradient using a running average (denoted by the hats in the update rule below). The update rule for ADAM is given by (where multiplication and division are understood to be element wise operations below)

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta})$$
$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$$
$$\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$$
$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1}$$
$$\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}},$$

where β_1 and β_2 set the memory lifetime of the first and second moment and are typically take to be 0.9 and 0.99 respectively, and η_t is a learning rate typically chosen to be 10^{-3}, and $\epsilon \sim 10^{-8}$ is a small regularization constant to prevent divergences.

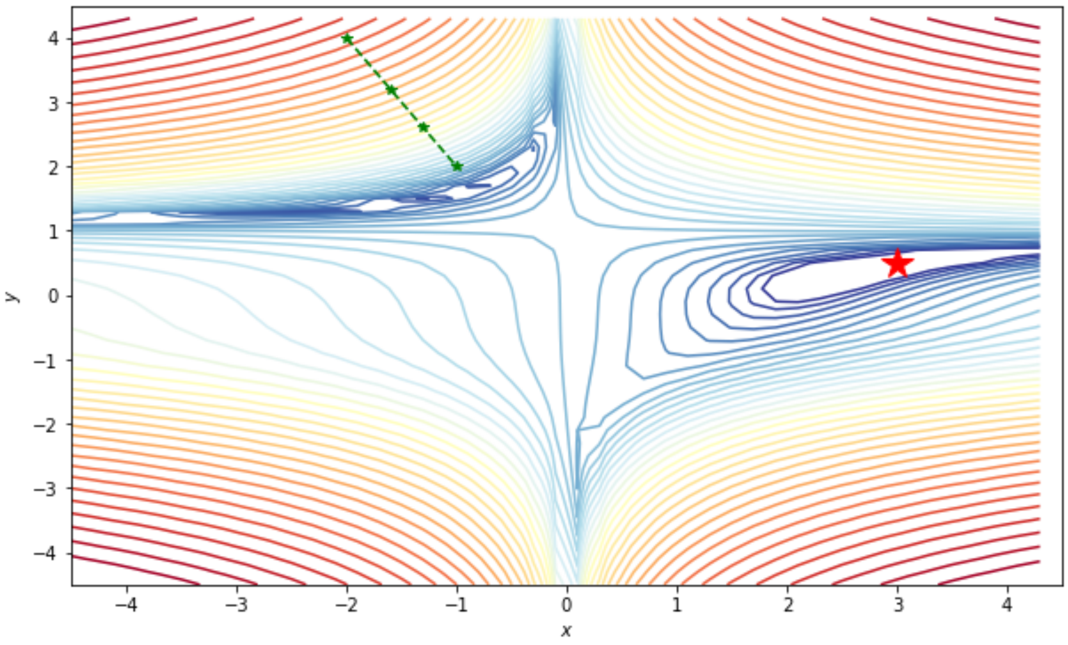
2. Take Beale's Function. We will use 3 different methods, gradient descent with and without momentum and ADAM, to find the minimum starting at different initial points: $(x, y) = (4, 3), (-2, -4), (-1, 4)$*. Take* 10^4 *steps. Set the learning rate for gradient* *descent (both with and without momentum) and ADAM to 10^{-6} and 10^{-3}, respectively. Plot trajectories. Make sure to lable each plot.*

Hint:

Here, θ = (x,y), and
$E(\theta) = E(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.6250 - x + xy^3)^2$
Hence, $\nabla_\theta E(\theta) = (\nabla_x E(x,y), \nabla_y E(x,y))$.

Suppose that you take 3 steps and obtain
$(x,y)_{t=1} = (-1.6, 3.2), (x,y)_{t=2} = (-1.3, 2.6), (x,y)_{t=3} = (-1.0, 2.0)$. Then, you can plot this trajectory in the following way:

In [121]:
```
fig, ax=contour_beales_function()

Nsteps = 3
trajectory = np.zeros([Nsteps+1,2])
trajectory[0,:] = [-2,4]; trajectory[1,:] = [-1.6,3.2]; trajecto

xs=trajectory[:,0]
ys=trajectory[:,1]
ax.plot(xs,ys, 'g--*')

plt.show()
```



In this case, the gradient is given by:

$$\nabla_\theta E(\theta) = (\nabla_x E(x,y), \nabla_y E(x,y))$$

$$\nabla_x E(x,y) = 2(1.5 - x + xy)(-1 + y) + 2(2.25 - x + xy^2)(-1 + y^2) + 2(2.6$$

$$\nabla_x E(x,y) = (2y^6 + 2y^4 - 4y^3 - 2y^2 - 4y + 6)x + 5.25y^3 + 4.5y^2$$

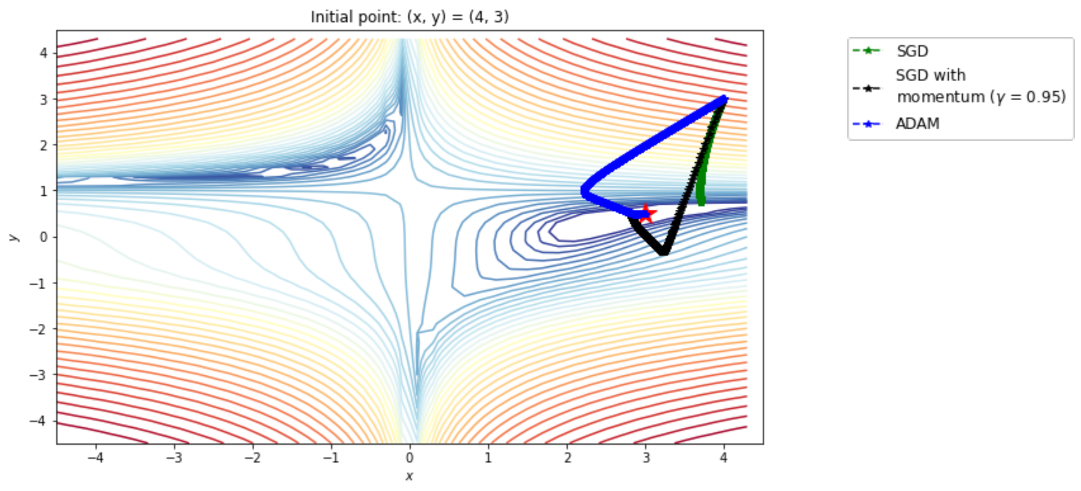
$$\nabla_y E(x,y) = 2(1.5 - x + xy)x + 2(2.25 - x + xy^2)2xy + 2(2.6250 -$$

$$\nabla_y E(x,y) = (6y^5 + 4y^3 - 6y^2 - 2y - 2)x^2 + (15.75y^2 + 9$$

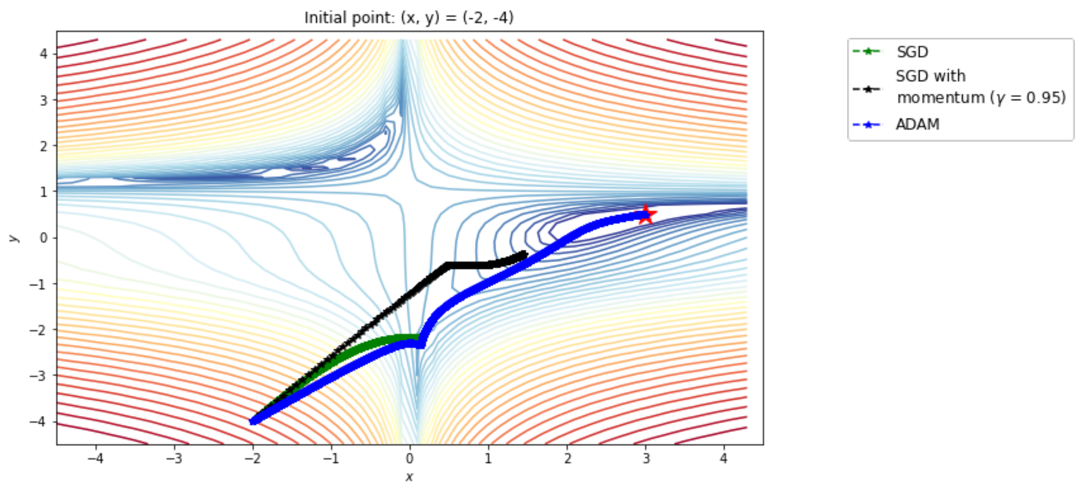
In [153]:

```
def grad_beale(theta):
  """
  Gradient of the Beale's function
  """
  x, y = theta
  gradx = (2*y**6 + 2*y**4 - 4*y**3 - 2*y**2 - 4*y + 6)*x
  gradx += 5.25*y**3 + 4.5*y**2 + 3*y - 12.75
  grady = (6*y**5 + 4*y**3 - 6*y**2 - 2*y - 2)*x**2 + (15.75*y**
  return np.array([gradx, grady])


def momentum_gd(initial_theta, gradient, learning_rate, niter, g
  """
  Gradient descent with momentum
  """
  thetas = np.empty((niter+1, len(initial_theta)))
  thetas[0] = initial_theta
  velocity = 0  # initial velocity
  for i in range(niter):
    velocity = gamma * velocity + learning_rate * gradient(theta
    thetas[i+1] = thetas[i] - velocity
  return thetas


def adam(
  initial_theta,
  gradient,
  learning_rate,
  niter,
  beta1=.9,
  beta2=.99,
  epsilon=1e-8,
):
  thetas = np.empty((niter+1, len(initial_theta)))
  thetas[0] = initial_theta
  m = 1  # iniital m
  s = 1  # initial s
  for i in range(niter):
    g = gradient(thetas[i])
    m = beta1 * m + (1-beta1) * g
    s = beta2 * s + (1-beta2) * g**2
    mhat = m / (1-beta1)
    shat = s / (1-beta2)
    thetas[i+1] = thetas[i] - learning_rate * mhat / np.sqrt(sha
  return thetas
```

In [171]:
```
INIT = (4, 3)
NSTEPS = int(1e4)
MOMENTUM_GAMMA = 0.95  # we tried a few, this gets closest of th

fig, ax = contour_beales_function()

# simple gradient descent
trajectory = simple_gd(INIT, grad_beale, 1e-6, NSTEPS)
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'g--*', label="SGD")

# with momentum
trajectory = momentum_gd(INIT, grad_beale, 1e-6, NSTEPS, MOMENTU
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'k--*', label=f"SGD with \nmomentum ($\gamma$ =

# adam
trajectory = adam(INIT, grad_beale, 1e-3, NSTEPS)
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'b--*', label="ADAM")

ax.legend(bbox_to_anchor=([1.45, 1.]), fontsize=12)
ax.set_title(f"Initial point: (x, y) = {INIT}")
plt.show()
```

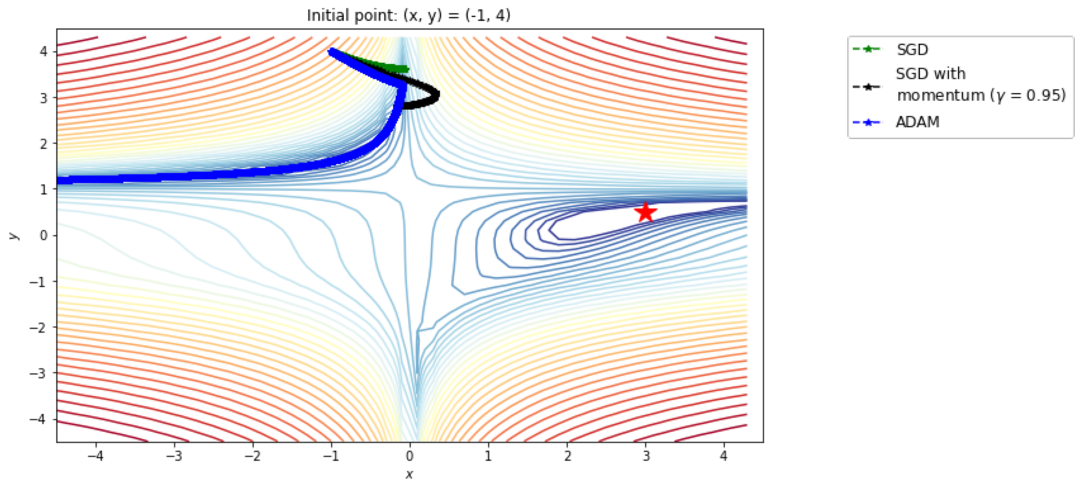


In [172]:
```
INIT = (-2, -4)

fig, ax = contour_beales_function()

# simple gradient descent
trajectory = simple_gd(INIT, grad_beale, 1e-6, NSTEPS)
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'g--*', label="SGD")

# with momentum
trajectory = momentum_gd(INIT, grad_beale, 1e-6, NSTEPS, MOMENTU
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'k--*', label=f"SGD with \nmomentum ($\gamma$ =

# adam
trajectory = adam(INIT, grad_beale, 1e-3, NSTEPS)
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'b--*', label="ADAM")

ax.legend(bbox_to_anchor=([1.45, 1.]), fontsize=12)
ax.set_title(f"Initial point: (x, y) = {INIT}")
plt.show()
```



In [173]:

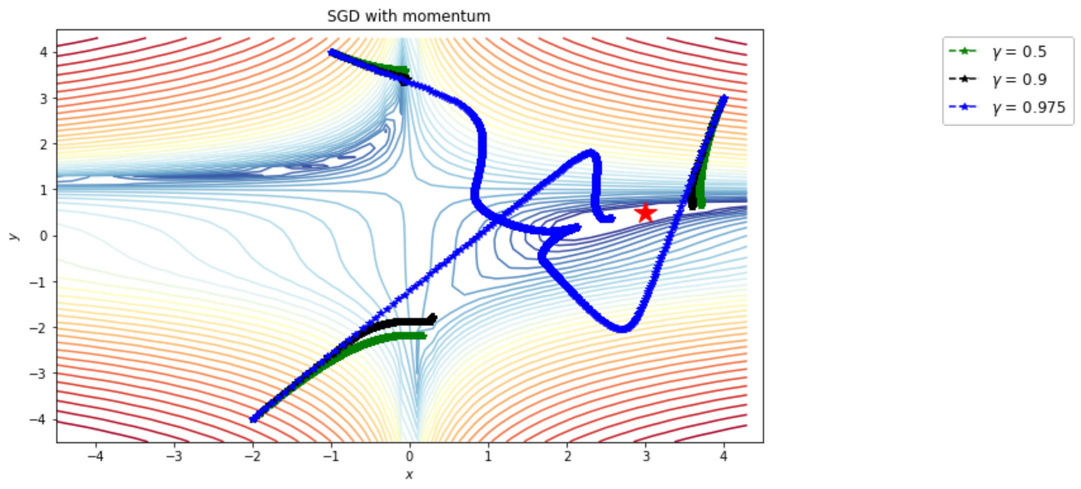
```
INIT = (-1, 4)


fig, ax = contour_beales_function()

# simple gradient descent
trajectory = simple_gd(INIT, grad_beale, 1e-6, NSTEPS)
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'g--*', label="SGD")

# with momentum
trajectory = momentum_gd(INIT, grad_beale, 1e-6, NSTEPS, MOMENTU
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'k--*', label=f"SGD with \nmomentum ($\gamma$ =

# adam
trajectory = adam(INIT, grad_beale, 1e-3, NSTEPS)
xs = trajectory[:,0]
ys = trajectory[:,1]
ax.plot(xs, ys, 'b--*', label="ADAM")

ax.legend(bbox_to_anchor=([1.45, 1.]), fontsize=12)
ax.set_title(f"Initial point: (x, y) = {INIT}")
plt.show()
```



We note that ADAM worked the best overall for this problem, but it could not find the minimum when the starting point was too bad (in the last case). Simple gradient descent with momentum converged after fine tuning of the mometum parameter (see this below). ADAM would likely be able to as well if we tuned the 7 parameters (beta1, beta2, epsilon, initial s, initial m, learning rate, number of steps). This surface clearly shows the limitations of gradient descent without momentum.

In [192]:

```
fig, ax = contour_beales_function()



inits = [(4, 3), (-2, -4), (-1, 4)]
gammas = [0.5, 0.9, 0.975]
ls = ["g--*", "k--*", "b--*"]

for j, i in enumerate(inits):
  for k, g in enumerate(gammas):
    trajectory = momentum_gd(i, grad_beale, 1e-6, NSTEPS, g)
    xs = trajectory[:,0]
    ys = trajectory[:,1]
    if j == 0:
      ax.plot(xs, ys, ls[k], label=f"$\gamma$ = {g}")
    else:
      ax.plot(xs, ys, ls[k])

ax.legend(bbox_to_anchor=([1.45, 1.]), fontsize=12)
ax.set_title("SGD with momentum")
plt.show()
```


SGD with momentum
γ = 0.5
γ = 0.9
γ = 0.975
x
y