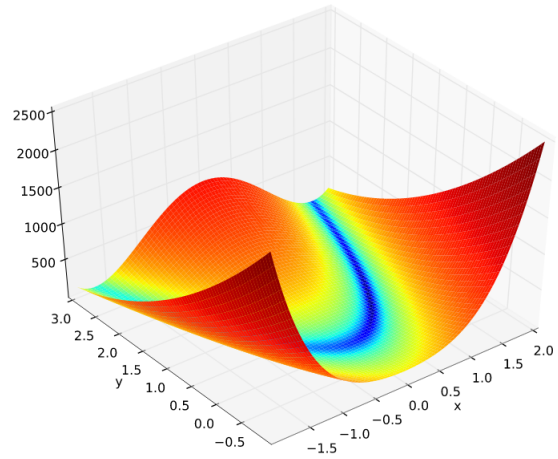
Deterministic Optimisation for Rosenbrock's function

Marcell Veiner (1619878)

February 2022

1 Introduction and Prolem Statement

This report has been written to compare some optimisation methods for the the Rosenbrock (or Rosenbrock's banana) function, which is non-convex function used for performance testing of optimisation algorithms [8]. The general function takes the form of Eq 1, which is known to have a global minimum at (a, a^2), for which the function has a root. This minimum is inside a long, parabolic shaped flat valley, which makes finding this point difficult.



$$f(x,y) = (a-x)^2 + b(y-x^2)^2 \tag{1}$$

Figure 1: Rosenbrock Function with $a = 1, b = 100$ (left) [1], and the Generic Formula (right)

The function will be optimised with gradient descent, conjugate gradient, and if time allows with the Levenberg–Marquardt method [5] when $a = 1, b = 100$. For this, we need to compute the gradient of the function with this parametrisation.

$$\nabla f = \begin{bmatrix} 400x^3 - 400xy + 2x - 2 \\ 200y - 200x^2 \end{bmatrix} \tag{2}$$

Similarly, the Hessian of f is:

$$H_f = \begin{bmatrix} 1200x^2 - 400y + 2 & -400x \\ -400x & 200 \end{bmatrix} \tag{3}$$

and the determinant of H_f is $det(H_f) = 240000x^2 - 80000y + 400 - 160000x^2 = 400(200x^2 - 200y + 1)$, which is positive in the region $y < x^2 + \frac{1}{200}$, note that in this case $1200x^2 - 400y + 2$ is also positive. Hence f is convex in any convex subregion of $y < x^2 + \frac{1}{200}$.

To classify the point $(1,1)$ we may check that $\nabla f(1,1)$ indeed is the zero vector. Then calculating the Hessian at $(1,1)$ we get:

$$H_f(1,1) = \begin{bmatrix} 802 & -400 \\ -400 & 200 \end{bmatrix}$$

whose eigenvalues are positive ($\lambda_1 = 1001.60, \lambda_2 = 0.39$), therefore we know that $(1,1)$ is a minimum.

2 Methods

The code for the project is very simplistic and can be found in the `src` folder. The `Rosenbrock.c` script contains the function and its gradient, namely:

```c
#include <math.h>

double rosenbrock(double * x){
  return pow((1 - x[0]), 2) + 100*pow((x[1] - x[0]*x[0]), 2);
}

void rosenbrock_grad(double * x, double * grad){
  grad[0] = -400*x[0]*x[1] + 400*x[0]*x[0]*x[0] + 2*x[0] - 2;
  grad[1] = 200*x[1] - 200*x[0]*x[0];
}
```

Listing 1: Rosenbrock.c

As one can see the functions operate on pointers and not two doubles x, y, which was done to make the code extensible for higher dimensional functions, and we will see that the rest of the code is written with that in mind too.

2.1 Gradient Descent

The optimisers can be found in the `Deterministic.c` script. The gradient descent code is concise and short, with the only complications because of the support of higher dimensions and other cases. In particular, the function takes a function pointer as an input, which calculates the gradient of the function to be minimised. Because `x` can be more than 2 dimensional, the updates need to be done with a for loop, even if in our case it executes only twice. The `calc_error` function simply computes the norm of the difference of `x` and `x_prev`. The convergence is controlled via a maximum number of iterations, and the epsilon parameter, checking the size of our steps.

```
double * gradient_descent(void (*grad_func)(double *, double *), double * x, int n,
    ...){
  // Init omitted

  while (error > epsilon && i < max_iter){
    for (j = 0; j < n; j++) x_prev[j] = x[j];

    // Update
    grad_func(x, grad);
    for (j = 0; j < n; j++) x[j] -= gamma * grad[j];

    // Get error
    error = calc_error(x, x_prev, n);

    // Log
    for (j = 0; j < n; j++) fprintf(fp, "%f, ", x[j]);
    fprintf(fp, "%f \n", error);
    i++;
  }

  return x;
}
```

Listing 2: Gradient Descent

2.2 Conjugate Gradient

The conjugate gradient is significantly longer, as we need to calculate the α and β terms, and the conjugate direction. The stepsize α is found using backtrack search (see Listing 3) and β is calculated using Fletcher - Reeves [6].

```
double backtrack_search(...){
  // Init omitted
  double fx0 = func(x0);
  for (j = 0; j < n; j++){
    x[j] = x0[j];
    prod += grad[j] * dx[j];
  }

  while((func(x) > fx0 + alpha * epsilon * prod) && i < max_iter){
    epsilon *= beta;
    for (j = 0; j < n; j++) x[j] = x0[j] + epsilon * dx[j];
    i++;
  }
  return epsilon;
}
```

Listing 3: Backtrack Search

The functions again support functions of higher order, and therefore operate on pointers. The convergence is checked similarly to gradient descent, but using the norm of the gradient instead of the error, nevertheless the error is computed for logging purposes.

```c
double * conjugate_gradient(...){
  // Init omitted

  // Iteration 0
  grad_func(x, grad);
  for (j = 0; j < n; j++) dx[j] = -grad[j];
  alpha = backtrack_search(func, grad, x, dx, n, 0.5, max_iter);
  for (j = 0; j < n; j++) {
    x[j] += alpha * dx[j];
    s[j] = dx[j];
  }

  while (norm > epsilon && i < max_iter){
    for (j = 0; j < n; j++) {
      x_prev[j] = x[j];
      grad_prev[j] = grad[j];
    }

    // Calculate Steepest Direction
    grad_func(x, grad);
    for (j = 0; j < n; j++) dx[j] = -grad[j];

    // Get beta - RF
    beta = 0;
    for (j = 0; j < n; j++){
       beta += grad[j] * grad[j];
       temp += grad_prev[j] * grad_prev[j];
     }
    beta /= temp;

    // Update Conjugate Direction
    for (j = 0; j < n; j++) s[j] = dx[j] + beta * s[j];

    // Get alpha
    alpha = backtrack_search(func, grad, x, s, n, 0.5, max_iter);

    // Update Position
    for (j = 0; j < n; j++) x[j] += alpha * s[j];

    // Log
    error = calc_error(x, x_prev, n);
    for (size_t j = 0; j < n; j++) fprintf(fp, "%f, ", x[j]);
    fprintf(fp, "%f, ", error);

    // Conditions
    norm = calc_norm(grad, n);
    fprintf(fp, "%f \n", error);
    i++;
  }
  return x;
}
```

Listing 4: Conjugate Gradient

2.3 Main

The code can be compiled with gcc, using `gcc main.c -o main -lm` to attach the required math library. The main takes the parameters visible on Table 1, with certain default values.

Parameter	Functionality	Range	Default
mode	Selects the optimiser, i.e gradient descent (1) vs conjugate gradient (0).	0, 1	1
x	Starting x coordinate.	$\mathbb{R}$	-1.5
y	Starting y coordinate.	$\mathbb{R}$	-1
gamma	Gradient descent learning rate, not used for conjugate gradient.	$[0, 1]$	0.00125
epsilon	Convergence check, if ε distance from criterion.	$[0, 1]$	0.00001
max_iter	Maximum number of iterations if no convergence.	$\mathbb{N}$	`INT_MAX`
path	Path for logging.	-	"logs/rosenbrock_gd.csv"

Table 1: Parameters in main.c

3 Results

The results were obtained by using the default parameters, i.e. starting from the point $(-1.5, -1)$ and an epsilon of 0.00001, with no maximum number of iterations (in practice). The gradient descent required 6,854 iterations to converge, while conjugate gradient only needed 1,040. Both algorithms correctly converged to the global minimum at $(1, 1)$. The intermediate points are visualised on Figure 3. The blue points denote the seed.

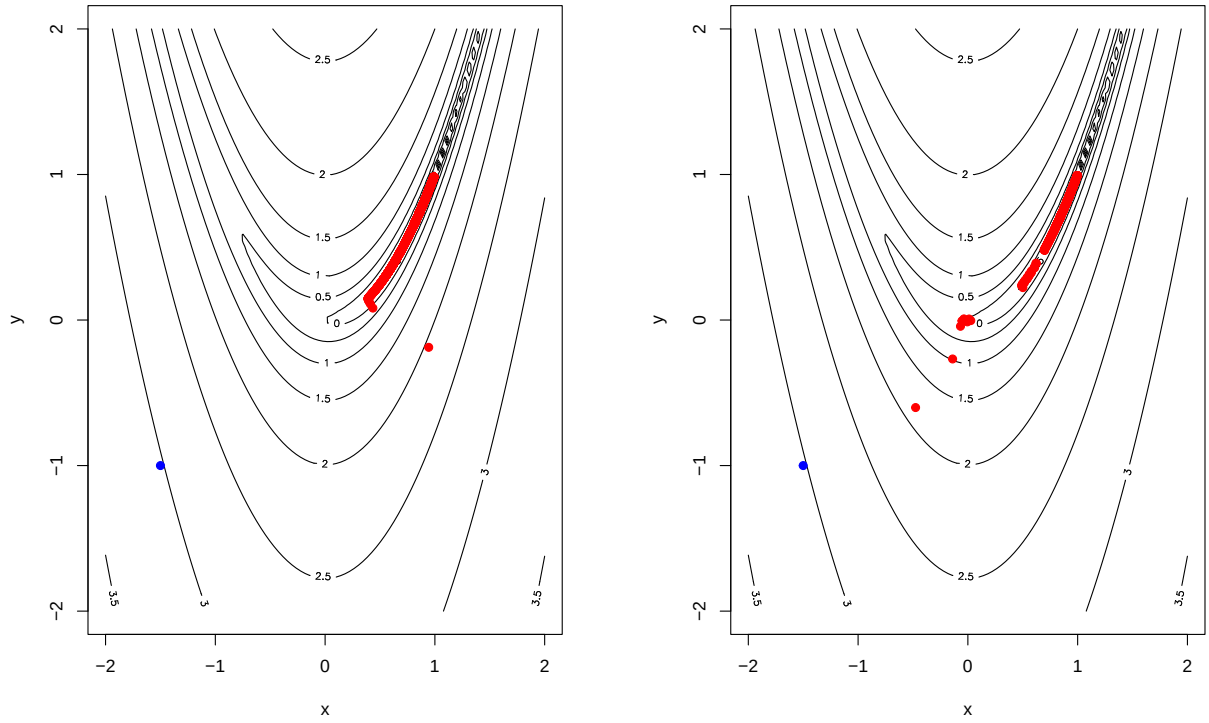


Figure 2: Intermediate Points of Gradient Descent (left), and Conjugate Gradient (right)

Looking at the error and norm (Figure 3), we can see that both algorithms took large steps in the beginning, and then smaller and smaller as it got closer to the target.

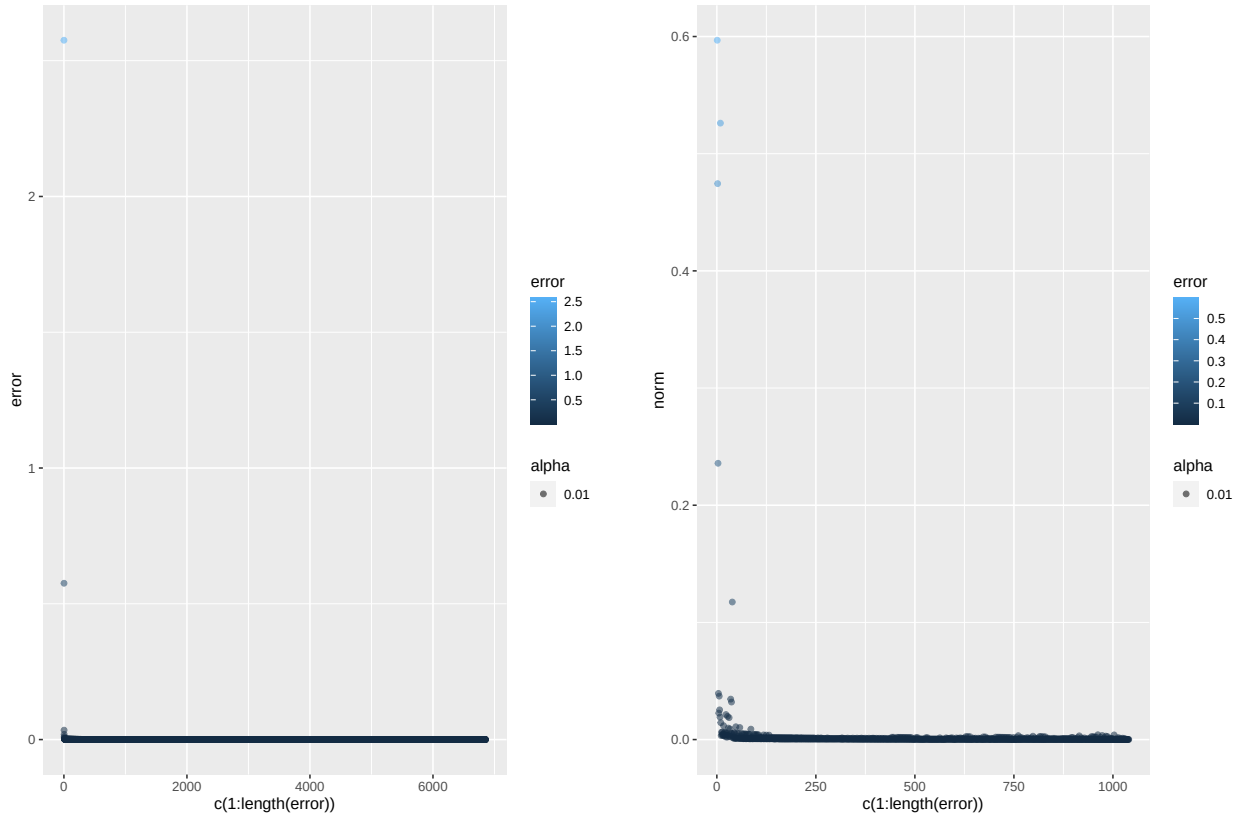


Figure 3: Gradient Descent Error (left), and Conjugate Gradient Norm (right) over Iterations

The final point for gradient descent was $(0.991124, 0.982291)$, while for conjugate gradient it settled on the point $(0.996188, 0.992341)$. It is clear that overall conjugate gradient was superior, however, both algorithms found the correct minimum. The seed also proved to be good in the sense that the differences in the algorithms could be observed, but not significant enough to obtain diverging results.

4 Further Work & Remarks

This was only a short glimpse into non-convex optimisation, and much more experimentation could be carried out. For example, initially the goal was to also explore the Levenberg–Marquardt method, which sadly had to be cut due to the strict time constraints at the end of the term, and some unexpected deadlines. One could further experiment with the effect of learning rate γ on gradient descent, and stating from different seeds.

Similarly, there are other versions for line search to try for conjugate gradient such as the golden section search [2] and Wolfe line search [3]. Moreover, the variable β can also be calculated in a number of ways such as Polak–Ribière [7], Hestenes-Stiefel [9], and Dai–Yuan [4], all of which would have been a nice addition given more time, even though they were not asked for in the assignment in any way.

References

[1] Wikipedia: Rosenbrock function. `https://www.wikiwand.com/en/Rosenbrock_function`. Accessed: 06/02/2022.

[2] Yen-Ching Chang. N-dimension golden section search: Its variants and limitations. In *2009 2nd International Conference on Biomedical Engineering and Informatics*, pages 1–6. IEEE, 2009.

[3] Yu-Hong Dai and Cai-Xia Kou. A nonlinear conjugate gradient algorithm with an optimal property and an improved wolfe line search. *SIAM Journal on Optimization*, 23(1):296–320, 2013.

[4] Yu-Hong Dai and Yaxiang Yuan. A nonlinear conjugate gradient method with a strong global convergence property. *SIAM Journal on optimization*, 10(1):177–182, 1999.

[5] Ethan Eade. Gauss-newton/levenberg-marquardt optimization. *Tech. Rep.*, 2013.

[6] Reeves Fletcher and Colin M Reeves. Function minimization by conjugate gradients. *The computer journal*, 7(2):149–154, 1964.

[7] Elijah Polak and Gerard Ribiere. Note sur la convergence de méthodes de directions conjuguées. *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, 3(R1):35–43, 1969.

[8] HoHo Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, 1960.

[9] Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Standards*, 49:409–435, 1952.