assn2

April 29, 2025

[2]:
```
:dep argmin
:dep argmin-math
:dep nalgebra
:dep plotters = { version = "0.3", features = ["evcxr"] }
```

[3]:
```
use argmin::core::{CostFunction, Executor, Gradient};
use argmin::solver::linesearch::condition::ArmijoCondition;
use argmin::solver::linesearch::BacktrackingLineSearch;
use argmin::solver::neldermead::NelderMead;
use argmin::solver::quasinewton::LBFGS;
use argmin_math::Error;
use nalgebra::{dmatrix, dvector, DMatrix, DVector};
use plotters::prelude::*;
```

1 Optimization with Equality Constraints

1.1 Is Lagrangian Needed?

A Lagrangian is not needed here because x_1 and x_2 are not free variables. x_1 is definitely 1, and we can solve the second constraint to show that $x_1 = 2$. Therefore, the minimization is performed on one free variable x_3 without constraints, which we can minimize directly.

[4]:
```
// Define the objective function with penalty
struct F1;

impl CostFunction for F1 {
    type Param = Vec<f64>;
    type Output = f64;

    fn cost(&self, x: &Self::Param) -> Result<Self::Output, Error> {
        let x1 = x[0];
        let x2 = x[1];
        let x3 = x[2];

        let objective = x1 + x2 + 2.0 * x3.powi(2);
        // Add penalty for the constraints
        let penalty = 1e12 * ((x1 - 1.0).powi(2) + (x1.powi(2) + x2.powi(2) - 1.
↪0).powi(2));
```

```
        let objective = objective + penalty;
        Ok(objective)
    }
}

// Separate the original f1(x) (without penalty)
fn f1(x: &Vec<f64>) -> f64 {
    x[0] + x[1] + 2.0 * x[2].powi(2)
}

let init_param = vec![
    vec![1.0, 0.0, 0.0],
    vec![0.0, 1.0, 0.0],
    vec![0.0, 0.0, 1.0],
];

// Set up the solver
let solver = NelderMead::new(init_param);

// Run the solver
let res = Executor::new(F1, solver).run()?;

let best_param = res.state().param.clone().unwrap();
println!(
    "Best solution: {:?}",
    best_param.iter().map(|x| (x * 100.0).round() / 100.0).collect::<Vec<f64>>()
);
println!(
    "Value of f1: {:.2}",
    (f1(&best_param) * 100.0).round() / 100.0
);
```

```
Best solution: [1.0, -0.0, 0.0]
Value of f1: 1.00
```

1.2 Basic Lagrangian

To compute the Lagrangian here, we get the derivatives. The original expression with the Lagrangian multiplier is

$$\mathcal{L}(x_1, x_2, \lambda) = 2x_1^2 + x_2^2 + \lambda(x_1 + x_2 - 1)$$

Taking the derivative with respect to each parameter

$$\frac{\partial \mathcal{L}}{\partial x_1} = 4x_1 + 0 \cdot x_2 + 1 \cdot \lambda = 0$$

$$\frac{\partial \mathcal{L}}{\partial x_2} = 0 \cdot x_1 + 2x_2 + 1 \cdot + \lambda = 0$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 1 \cdot x_1 + 1 \cdot x_2 + 0 \cdot \lambda = 1$$

These are the equations we are minimizing.

Converted to matrix form, we get

$$\begin{bmatrix} 4 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

[5]:
```rust
fn lagrangian(a: DMatrix<f64>, rhs: DVector<f64>) -> DVector<f64> {
    a.lu()
        .solve(&rhs)
        .expect("System should have unique solution")
}

let data: DMatrix<f64> = dmatrix![
    4.0, 0.0, 1.0;
    0.0, 2.0, 1.0;
    1.0, 1.0, 0.0;
];
// === Lagrangian solution ===
let rhs1 = DVector::from_vec(vec![0.0, 0.0, 1.0]);
let rhs1_05 = DVector::from_vec(vec![0.0, 0.0, 1.05]);

let solution1 = lagrangian(data.clone(), rhs1);
let solution1_05 = lagrangian(data.clone(), rhs1_05);

let x1_rhs1 = solution1[0];
let x2_rhs1 = solution1[1];
let x1_rhs1_05 = solution1_05[0];
let x2_rhs1_05 = solution1_05[1];

println!(
    "Lagrangian solution (rhs = 1): x1 = {:.4}, x2 = {:.4}",
    x1_rhs1, x2_rhs1
);
println!(
    "Lagrangian solution (rhs = 1.05): x1 = {:.4}, x2 = {:.4}",
    x1_rhs1_05, x2_rhs1_05
);
```

```
Lagrangian solution (rhs = 1): x1 = 0.3333, x2 = 0.6667
Lagrangian solution (rhs = 1.05): x1 = 0.3500, x2 = 0.7000
```

We can verify this solution with the gradient-based numerical optimizer L-BFGS.

[6]:
```rust
struct F2 {
    rhs: f64,
}

impl CostFunction for F2 {
    type Param = Vec<f64>;
    type Output = f64;

    fn cost(&self, x: &Self::Param) -> Result<Self::Output, Error> {
        let x1 = x[0];
        let x2 = x[1];
        let objective = 2.0 * x1.powi(2) + x2.powi(2);
        let penalty = 1e8 * (x1 + x2 - self.rhs).powi(2);
        Ok(objective + penalty)
    }
}

impl Gradient for F2 {
    type Param = Vec<f64>;
    type Gradient = Vec<f64>;

    fn gradient(&self, x: &Self::Param) -> Result<Self::Gradient, Error> {
        let x1 = x[0];
        let x2 = x[1];

        // Gradient of objective
        let grad_obj_x1 = 4.0 * x1;
        let grad_obj_x2 = 2.0 * x2;

        // Gradient of penalty
        let common_penalty_grad = 2.0 * 1e2 * (x1 + x2 - self.rhs);

        Ok(vec![
            grad_obj_x1 + common_penalty_grad,
            grad_obj_x2 + common_penalty_grad,
        ])
    }
}

fn f2(x: &Vec<f64>) -> f64 {
    2.0 * x[0].powi(2) + x[1].powi(2)
}


// === Gradient-based numerical optimization ===
let init_param1 = vec![1.0 / 3.0, 2.0 / 3.0];
let init_param2 = vec![0.35, 0.70];

let linesearch = BacktrackingLineSearch::new(ArmijoCondition::new(0.2)?);
let solver1 = LBFGS::new(linesearch.clone(), 5);
let solver2 = LBFGS::new(linesearch, 5);
let res1 = Executor::new(F2 { rhs: 1.0 }, solver1)
    .configure(|state| state.param(init_param1))
    .run()?;

let res2 = Executor::new(F2 { rhs: 1.05 }, solver2)
    .configure(|state| state.param(init_param2))
    .run()?;

let best1 = res1.state().param.clone().unwrap();
let best2 = res2.state().param.clone().unwrap();

println!(
    "Gradient solution (rhs = 1): x1 = {:.4}, x2 = {:.4}",
    best1[0], best1[1]
);
println!(
    "Gradient solution (rhs = 1.05): x1 = {:.4}, x2 = {:.4}",
    best2[0], best2[1]
);

println!("\nf* (rhs = 1): {:.4}", f2(&best1));
println!("f* (rhs = 1.05): {:.4}", f2(&best2));
println!("Difference: {:.6}", f2(&best2) - f2(&best1));
```

```
Gradient solution (rhs = 1): x1 = 0.3333, x2 = 0.6667
Gradient solution (rhs = 1.05): x1 = 0.3500, x2 = 0.7000

f* (rhs = 1): 0.6667
f* (rhs = 1.05): 0.7350
Difference: 0.068333
```

1.3 Max Expected Return with Target Risk

1.3.1 Lagrangian Solution

We can solve this analytically.

We start with the Lagrangian equation with two multipliers, λ_1 and λ_2.

$$\mathcal{L}(\omega_1, \omega_2, \lambda_1, \lambda_2) = \rho_1\omega_1 + \rho_2\omega_2 + \lambda_1(\sigma_1^2\omega_1^2 + \sigma_2^2\omega_2^2 + 2\rho_{12}\sigma_1\sigma_2\omega_1\omega_2 - \sigma_T^2) + \lambda_2(\omega_1 + \omega_2 - 1)$$

We can simplify this to a single quadratic equation, since we are functionally solving for only one variable, ω_1.

First we substitute

$$\omega_2 = 1 - \omega_1$$

into the squared risk constraint to get

$$\sigma_1^2\omega_1^2 + \sigma_2^2(1-\omega_1)^2 + 2\rho_{12}\sigma_1\sigma_2\omega_1(1-\omega_1) = \sigma_T^2$$

Then we expand and rearrange to get a quadratic form:

$$= (\sigma_1^2 + \sigma_2^2 - 2\rho_{12}\sigma_1\sigma_2)\omega_1^2 + (-2\sigma_2^2 + 2\rho_{12}\sigma_1\sigma_2)\omega_1 + \sigma_2^2$$

which can be solved by the quadratic formula to get

$$\omega_1^* = \frac{(\sigma_2^2 - \rho_{12}\sigma_1\sigma_2) \pm \sqrt{(\sigma_2^2 - \rho_{12}\sigma_1\sigma_2)^2 - (\sigma_1^2 + \sigma_2^2 - 2\rho_{12}\sigma_1\sigma_2)(\sigma_2^2 - \sigma_T^2)}}{\sigma_1^2 + \sigma_2^2 - 2\rho_{12}\sigma_1\sigma_2}$$

and

$$\omega_2^* = 1 - \omega_1^*$$

where optimal return $R_P(\sigma_T)$ is

$$R_P(\sigma_T) = \rho_1\omega_1^* + \rho_2\omega_2^*$$

[7]:
```
fn R_p(
    sigma1: f64,
    sigma2: f64,
    rho12: f64,
    sigma_t: f64,
    rho1: f64,
    rho2: f64,
) -> Result<(f64, f64, f64), String> {
    let a = sigma1.powi(2) + sigma2.powi(2) - 2.0 * rho12 * sigma1 * sigma2;
    let b = sigma2.powi(2) - rho12 * sigma1 * sigma2;
    let c = sigma2.powi(2) - sigma_t.powi(2);

    let discriminant = b.powi(2) - a * c;
    if discriminant < 0.0 {
        return Err("No real solution exists for the given inputs.".to_owned());
    }

    let sqrt_discriminant = discriminant.sqrt();

    // Calculate the two possible solutions for  1
    let omega1_1 = (b + sqrt_discriminant) / a;
    let omega1_2 = (b - sqrt_discriminant) / a;

    // Choose the solution that satisfies the constraints (e.g., 0 <=  1 <= 1)
    let omega1 = if (0.0..=1.0).contains(&omega1_1) {
        omega1_1
    } else if (0.0..=1.0).contains(&omega1_2) {
        omega1_2
    } else {
        return Err("No valid solution for  1 in the range [0, 1].".to_owned());
    };

    let omega2 = 1.0 - omega1;

    // Calculate the optimal return
    let rp = rho1 * omega1 + rho2 * omega2;

    Ok((omega1, omega2, rp))
}
```

1.3.2 Realized Efficient Frontier

[8]:

```
let rho1 = 0.05;
let rho2 = 0.10;
let sigma1 = 0.10;
let sigma2 = 0.20;
let rho12 = -0.5;

let sigma_t_values: Vec<f64> = (4..=60).map(|x| x as f64 / 200.0).collect();
let mut rp_values = Vec::new();

for sigma_t in &sigma_t_values {
    match R_p(sigma1, sigma2, rho12, *sigma_t, rho1, rho2) {
        Ok((_, _, rp)) => {
            rp_values.push(rp);
        }
        Err(e) => {
            //println!("Error: {}", e);
            rp_values.push(0.0); // Push a default value in case of error
            continue;
        }
    };
}

let root = evcxr_figure((800, 600), |root| {
    root.fill(&WHITE).unwrap();
    let mut chart = ChartBuilder::on(&root)
        .caption("Efficient Frontier", ("sans-serif", 20))
        .margin(10)
        .x_label_area_size(30)
        .y_label_area_size(30)
        .build_cartesian_2d(0.0..0.30, 0.0..0.12)
        .unwrap();

    chart.configure_mesh().draw().unwrap();

    chart
        .draw_series(LineSeries::new(
            sigma_t_values.into_iter().zip(rp_values),
            &RED,
        ))
        .unwrap()
        .label("Efficient Frontier")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &RED));

    chart
        .configure_series_labels()
        .border_style(BLACK)
        .draw()
        .unwrap();

    Ok(())
});

root.evcxr_display()
```

[8]: ()

1.3.3 Min Variance with Target Expected Return

We're using the same quadratic formula for σ_P^2 as before.

We can solve for the weights linearly with respect to ρ_T.

$$\rho_1\omega_1 + \rho_2(1 - \omega_1) = \rho_T$$
$$\omega_1^* = \frac{\rho_T - \rho_2}{\rho_1 - \rho_2}$$
$$\omega_2^* = 1 - \omega_1$$

And we minimize σ_P with respect to these inputs.

[9]:
```
fn sigma_p(
    sigma1: f64,
    sigma2: f64,
    rho12: f64,
    rho1: f64,
    rho2: f64,
    rho_t: f64,
) -> Result<(f64, f64, f64), String> {
    let covar12 = rho12 * sigma1 * sigma2;
    let a = sigma1.powi(2) + sigma2.powi(2) - 2.0 * covar12;
    let b = -2.0 * (sigma2.powi(2) - covar12);
    let c = sigma2.powi(2);

    let denom = rho1 - rho2;
    if denom.abs() < 1e-10 {
        return Err("Division by zero: rho1 and rho2 are too close.".to_owned());
    }

    // Calculate omega1 using the target return
    let omega1 = (rho_t - rho2) / denom;
    let omega2 = 1.0 - omega1;

    // Ensure solution is valid
    if !(0.0..=1.0).contains(&omega1) || !(0.0..=1.0).contains(&omega2) {
        return Err("No valid solution for  1,  2 in the range [0, 1].".
 ↪to_owned());
    }

    // Calculate the minimal achievable variance
    let variance = a * omega1.powi(2) + b * omega1 + c;
    if variance < 0.0 {
        return Err("Negative variance encountered, no real solution.".
 ↪to_owned());
    }

    let sigma_p = variance.sqrt();

    Ok((omega1, omega2, sigma_p))
}
```

1.3.4 Realized Efficient Frontier

[10]:
```
// Generate target return values (from 5% to 10%)
let rho_t_values: Vec<f64> = (10..=20).map(|x| x as f64 / 200.0).collect();
let mut sigma_p_values = Vec::new();

for rho_t in &rho_t_values {
    match sigma_p(sigma1, sigma2, rho12, rho1, rho2, *rho_t) {
        Ok((_, _, sigma_p)) => {
            sigma_p_values.push(sigma_p);
        }
        Err(e) => {
            println!("Error: {}", e);
            // Push a default value (0.0) in case of error
            sigma_p_values.push(0.0);
        }
    };
}

let root = evcxr_figure((800, 600), |root| {
    root.fill(&WHITE).unwrap();
    let mut chart = ChartBuilder::on(&root)
        .caption("Minimum Variance vs Target Return", ("sans-serif", 20))
        .margin(10)
        .x_label_area_size(30)
        .y_label_area_size(30)
        .build_cartesian_2d(0.0..0.30, 0.0..0.12)
        .unwrap();

    chart.configure_mesh().draw().unwrap();

    chart
        .draw_series(LineSeries::new(
            sigma_p_values.into_iter().zip(rho_t_values),
            &BLUE,
        ))
        .unwrap()
        .label("Minimum Variance Curve")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &BLUE));

    chart
        .configure_series_labels()
        .border_style(BLACK)
        .draw()
        .unwrap();

    Ok(())
});

root.evcxr_display()
```

```
[10]: ()
```

Both plots show a range of expected returns that are analytically solved for between 5% and 10%. The graphs are identical, except for the fact that the max return graph only shows the smaller variance when there are two possible variances for a single return. The reversed min variance graph shows both.

2 Optimization with Inequality Constraints

2.1 Convex Problem

The Lagrangian is

$$\mathcal{L}(x_1, x_2, \lambda_1, \lambda_2) = (x_1 - 2)^2 + 2(x_2 - 1)^2 + \lambda_1(x_1 + 4x_2 - 3) + \lambda_2(x_2 - x_1)$$

The derivatives are

$$\frac{\partial \mathcal{L}}{\partial x_1} = 2(x_1 - 2) + \lambda_1 - \lambda_2 = 0$$
$$\frac{\partial \mathcal{L}}{\partial x_2} = 4(x_2 - 2) + 4\lambda_1 + \lambda_2 = 0$$

We'll consider the first constraint being active, and the second one being inactive. That means the first constraint must be equal to 0, $\lambda_1 \geq 0$, and $\lambda_2 = 0$.

Then we can solve for λ_1 using the derivatives as a systems of equations

$$\lambda_1 = -2(x_1 - 2)$$
$$x_2 = 1 - \lambda_1$$

and substituting in the active constraint

$$x_1 + 4(1 - \lambda_1) = 3$$
$$x_1 = -1 + 4\lambda_1$$

which is substituted back into the original.

$$\lambda_1 = -2(-1 + 4\lambda_1 - 2)$$
$$\lambda_1 = \frac{2}{3}$$

$$x_1 = -1 + 4\left(\frac{2}{3}\right) = \frac{5}{3}$$
$$x_2 = 1 - \frac{2}{3} = \frac{1}{3}$$

This satisfies primal feasibility:

$$\frac{5}{3} + 4\left(\frac{1}{3}\right) - 3 = 0$$
$$\frac{1}{3} - \frac{5}{3} \leq 0$$

and dual feasibility:

$$\frac{2}{3} > 0$$
$$0 = 0$$

which means the optimal point is:

$$(x_1, x_2) = \left(\frac{5}{3}, \frac{1}{3}\right)$$

[11]:
```
struct F4;

impl CostFunction for F4 {
    type Param = Vec<f64>;
    type Output = f64;

    fn cost(&self, x: &Self::Param) -> Result<Self::Output, Error> {
        let x1 = x[0];
        let x2 = x[1];

        let objective = (x1 - 2.0).powi(2) + 2.0 * (x2 - 1.0).powi(2);

        // Constraint 1: x1 + 4x2   3 → penalty if > 3
        let g1 = (x1 + 4.0 * x2 - 3.0).max(0.0);

        // Constraint 2: x2 - x1   0 → penalty if > 0
        let g2 = (x2 - x1).max(0.0);

        let penalty = 1e6 * (g1.powi(2) + g2.powi(2));

        Ok(objective + penalty)
    }
}

impl Gradient for F4 {
    type Param = Vec<f64>;
    type Gradient = Vec<f64>;

    fn gradient(&self, x: &Self::Param) -> Result<Self::Gradient, Error> {
        let x1 = x[0];
        let x2 = x[1];

        // Objective gradient
        let grad_x1 = 2.0 * (x1 - 2.0);
        let grad_x2 = 4.0 * (x2 - 1.0);

        // Constraint gradients (only active if violated)
        let mut penalty_grad_x1 = 0.0;
        let mut penalty_grad_x2 = 0.0;

        // g1 = x1 + 4x2 - 3
        let g1 = x1 + 4.0 * x2 - 3.0;
        if g1 > 0.0 {
            penalty_grad_x1 += 2.0 * 1e6 * g1;
            penalty_grad_x2 += 8.0 * 1e6 * g1;
        }
```

```rust
        // g2 = x2 - x1
        let g2 = x2 - x1;
        if g2 > 0.0 {
            penalty_grad_x1 += -2.0 * 1e6 * g2;
            penalty_grad_x2 += 2.0 * 1e6 * g2;
        }

        Ok(vec![
            grad_x1 + penalty_grad_x1,
            grad_x2 + penalty_grad_x2,
        ])
    }
}

fn f4(x: &Vec<f64>) -> f64 {
    (x[0] - 2.0).powi(2) + 2.0 * (x[1] - 1.0).powi(2)
}

let init_param = vec![1.0, 1.0]; // Initial guess inside feasible region

let linesearch = BacktrackingLineSearch::new(ArmijoCondition::new(0.2)?);
let solver = LBFGS::new(linesearch, 5);

let result = Executor::new(F4, solver)
    .configure(|state| state.param(init_param))
    .run()?;

let best = result.state().param.clone().unwrap();

println!("Optimal solution: x1 = {:.4}, x2 = {:.4}", best[0], best[1]);
```

```
Optimal solution: x1 = 1.6667, x2 = 0.3333
```

2.2 Nonlinear Problem

Constraints are

$$
\begin{aligned}
g_1(x) &= -x_1 \leq 0 \\
g_2(x) &= -x_2 \leq 0 \\
g_3(x) &= 2 - x_1 x_2 \leq 0
\end{aligned}
$$

This is concave maximization, so we can take the opposite sign for convex minimization:

$$\min_{x_1, x_2} \tilde{f}(x) = x_1^2 + x_1 x_2 + 3x_2^2 - 5$$

Lagrangian is

$$\mathcal{L}(x_1, x_2, \lambda_1, \lambda_2, \lambda_3) = x_1^2 + x_1 x_2 + 3x_2^2 - 5 + \lambda_1(-x_2) + \lambda_2(-x_2) + \lambda_3(2 - x_1 x_2)$$

And derivatives are

$$\frac{\partial \mathcal{L}}{\partial x_1} = 2x_1 + x_2 - \lambda_1 - \lambda_3 x_2 = 0$$
$$\frac{\partial \mathcal{L}}{\partial x_2} = x_1 + 6x_2 - \lambda_2 - \lambda_3 x_1 = 0$$

When considering candidates for KKT, we can disregard active constraints on g_1 and g_2, because in either case, $x_1 x_2 = 0$, resulting in $g_3(x) = 2 \leq 0$ which is false.

The only possible active constraint is g_3. Following the same systems and substitutions methods as previously:

We will restate our derivatives without λ_1 or λ_2:

$$2x_1 + x_2(1 - \lambda_3) = 0$$

$$x_1(1 - \lambda_3) + 6x_2 = 0$$

And we also know that x_1 and x_2 are related by $x_1 x_2 = 2$. So we can substitute

$$x_2 = \frac{2}{x_1}$$

into

$$2x_1 + \frac{2}{x_1}(1 - \lambda_3) = 0$$
$$x_1^2 = \lambda_3 - 1$$

$$x_1(1 - \lambda_3) + 6 \cdot \frac{2}{x_1} = 0$$
$$x_1^2 = \frac{12}{\lambda_3 - 1}$$

So we can set these equal,

$$(\lambda_3 - 1)^2 = 12$$
$$\lambda_3 = 1 \pm 2\sqrt{3}$$

and $\lambda_3 \geq 0$ so

$$\lambda_3 = 1 + 2\sqrt{3}$$

and substituting back

$$x_1^2 = 1 + 2\sqrt{3} - 1$$

$$x_1 = \sqrt{2\sqrt{3}} \approx 1.8612$$

(must be positive because of g_1)

$$x_2 = \frac{2}{\sqrt{2\sqrt{3}}} \approx 1.0746$$

And because none of the other cases can be satisfied, this local minimum (maximum) is known to be the global minimum (maximum).

[12]:
```rust
struct F5;

impl CostFunction for F5 {
    type Param = Vec<f64>;
    type Output = f64;

    fn cost(&self, x: &Self::Param) -> Result<Self::Output, Error> {
        let x1 = x[0];
        let x2 = x[1];

        let objective = -5.0 + x1.powi(2) + x1 * x2 + 3.0 * x2.powi(2);

        // Constraint 1: x1 >= 0 → penalty if < 0
        let g1 = (-x1).max(0.0);

        // Constraint 2: x2 >= 0 → penalty if < 0
        let g2 = (-x2).max(0.0);

        // Constraint 3: x1 * x2 >= 2 → penalty if < 2
        let g3 = (2.0 - x1 * x2).max(0.0);

        let penalty = 1e6 * (g1.powi(2) + g2.powi(2) + g3.powi(2));

        Ok(objective + penalty)
    }
}

impl Gradient for F5 {
    type Param = Vec<f64>;
    type Gradient = Vec<f64>;

    fn gradient(&self, x: &Self::Param) -> Result<Self::Gradient, Error> {
        let x1 = x[0];
        let x2 = x[1];
        // Objective gradient
        let grad_x1 = 2.0 * x1 + x2;
        let grad_x2 = x1 + 6.0 * x2;

        // Constraint gradients (only active if violated)
        let mut penalty_grad_x1 = 0.0;
        let mut penalty_grad_x2 = 0.0;

        // g1 = -x1
        let g1 = -x1;
        if g1 > 0.0 {
            penalty_grad_x1 += -2.0 * 1e6 * g1;
        }

        // g2 = -x2
        let g2 = -x2;
        if g2 > 0.0 {
            penalty_grad_x2 += -2.0 * 1e6 * g2;
        }

        // g3 = 2 - x1 * x2
        let g3 = 2.0 - x1 * x2;
        if g3 > 0.0 {
            penalty_grad_x1 += -2.0 * 1e6 * g3 * x2;
            penalty_grad_x2 += -2.0 * 1e6 * g3 * x1;
        }

        Ok(vec![
            grad_x1 + penalty_grad_x1,
            grad_x2 + penalty_grad_x2,
        ])
    }
}

// Objective function without penalty
fn f5(x: &Vec<f64>) -> f64 {
    -5.0 + x[0].powi(2) + x[0] * x[1] + 3.0 * x[1].powi(2)
}

let init_param = vec![1.5, 1.5]; // Reasonable initial guess inside feasible␣
 ↪region

let linesearch = BacktrackingLineSearch::new(ArmijoCondition::new(0.2)?);
let solver = LBFGS::new(linesearch, 5);

let result = Executor::new(F5, solver)
    .configure(|state| state.param(init_param))
    .run()?;

let best = result.state().param.clone().unwrap();

println!("Optimal solution: x1 = {:.4}, x2 = {:.4}", best[0], best[1]);
```

```
Optimal solution: x1 = 1.8612, x2 = 1.0746
```

3 Mean-Variance Optimization

3.1 Full Implementation Code

[27]:
```
struct InvestmentUniverse {
    cov: DMatrix<f64>,
    inv_cov: DMatrix<f64>,
    ones: DVector<f64>,
    rho: DVector<f64>,
    rho0: f64,
}

impl InvestmentUniverse {
    pub fn new(cov: DMatrix<f64>, rho: DVector<f64>, rho0: f64) -> Self {
        InvestmentUniverse {
            inv_cov: cov
                .clone()
                .try_inverse()
                .expect("Covariance matrix is not invertible"),
            ones: DVector::from_element(rho.len(), 1.0),
            cov,
            rho,
            rho0,
        }
    }
    pub fn gmv(&self) -> DVector<f64> {
        let w_min_var = &self.inv_cov * &self.ones;
        let v = (self.ones.transpose() * &self.inv_cov * &self.ones)[(0, 0)];
        w_min_var / v
    }
    pub fn min_var_target(&self, target_return: f64) -> DVector<f64> {
        let a = (self.rho.transpose() * &self.inv_cov * &self.rho)[(0, 0)];
        let b = (self.rho.transpose() * &self.inv_cov * &self.ones)[(0, 0)];
        let c = (self.ones.transpose() * &self.inv_cov * &self.ones)[(0, 0)];

        let lambda = -2.0 * (c * target_return - b) / (a * c - b.powi(2));
        let gamma = -2.0 * (a - b * target_return) / (a * c - b.powi(2));

        -0.5 * &self.inv_cov * (lambda * &self.rho + gamma * &self.ones)
    }
    pub fn two_fund_theorem(&self, target_return: f64) -> DVector<f64> {
        let rho_1 = self.rho[0];
        let rho_avg_23 = (self.rho[1] + self.rho[2]) / 2.0;

        let w_min_var_1 = self.min_var_target(rho_1);
        let w_min_var_avg_23 = self.min_var_target(rho_avg_23);

        let alpha = (target_return - rho_avg_23) / (rho_1 - rho_avg_23);

        alpha * w_min_var_1 + (1.0 - alpha) * w_min_var_avg_23
    }

    pub fn tangent_portfolio(&self) -> DVector<f64> {
        let excess_rho = &self.rho - self.rho0 * &self.ones;
        let inv_cov_excess = &self.inv_cov * &excess_rho;

        let denominator = self.ones.dot(&inv_cov_excess);
        inv_cov_excess / denominator
    }

    pub fn one_fund_theorem_return(&self, target_return: f64) -> f64 {
        let tangent_portfolio = self.tangent_portfolio();
        let tangent_return = (self.rho.transpose() * &tangent_portfolio)[(0,␣
 ↪0)];
        (target_return - tangent_return) / (self.rho0 - tangent_return)
    }

    pub fn one_fund_theorem_volatility(&self, target_volatility: f64) -> f64 {
        let tangent_portfolio = self.tangent_portfolio();
        let tangent_volatility =
            ((tangent_portfolio.transpose() * &self.cov *␣
 ↪&tangent_portfolio)[(0, 0)]).sqrt();

        1.0 - tangent_volatility / target_volatility
    }
}
let iu = InvestmentUniverse::new(
    dmatrix![
        1.0, 0.2, 0.1;
        0.2, 1.1, 0.3;
        0.1, 0.3, 2.0
    ],
    dvector![4.27, 0.15, 2.85],
    0.75,
);
```

3.2 Global Minimal Variance Portfolio

[14]:
```
println!("{:.4}", iu.gmv());
```

```
0.4495
0.3564
0.1941
```

3.3 Minimum Variance Portfolio

[15]:
```
println!("{:.4}", iu.min_var_target(iu.rho[0]));
```

```
 0.8231
-0.0930
 0.2699
```

[16]:
```
println!("{:.4}", iu.min_var_target((iu.rho[1] + iu.rho[2]) / 2.0));
```

```
0.2297
0.6208
0.1496
```

3.4 Two-Fund Theorem

[17]:
```
println!("{:.4}", iu.two_fund_theorem(4.0));
```

```
 0.7653
-0.0235
 0.2582
```

3.5 Two-Fund Theorem Efficient Frontier

[18]:
```
let target_returns: Vec<f64> = (0..=65).map(|x| x as f64 / 10.0).collect();
let mut weights = Vec::new();
let mut volatilities = Vec::new();

for target_return in &target_returns {
    let w = iu.two_fund_theorem(*target_return);
    let volatility = ((w.transpose() * &iu.cov * &w)[(0, 0)]).sqrt();
    weights.push(w);
    volatilities.push(volatility);
}

let root = evcxr_figure((800, 600), |root| {
    root.fill(&WHITE).unwrap();
    let mut chart = ChartBuilder::on(&root)
        .caption("Mean-Variance Efficient Frontier", ("sans-serif", 20))
        .margin(10)
        .x_label_area_size(30)
        .y_label_area_size(30)
        .build_cartesian_2d(0.5..1.5, 0.0..6.5)
        .unwrap();

    chart.configure_mesh().draw().unwrap();

    chart
        .draw_series(LineSeries::new(
            volatilities.into_iter().zip(target_returns),
            &BLUE,
        ))
        .unwrap()
        .label("Efficient Frontier")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &BLUE));

        for i in 0..3 {
            chart
                .draw_series(PointSeries::of_element(
                    vec![((iu.cov[(i, i)]).sqrt(), iu.rho[i])],
                    5,
                    &RED,
                    &|c, s, st| {
                        return EmptyElement::at(c)
                            + Circle::new((0, 0), s, st.filled())
                            + Text::new(format!("S{i}"), (10, 0),␣
↪("sans-serif", 15));
                    },
                ))
                .unwrap();
        }

    chart
        .configure_series_labels()
        .border_style(BLACK)
        .draw()
        .unwrap();

    Ok(())
});

root.evcxr_display();
```

4 Riskless Asset

4.0.1 Tangent Portfolio

[19]:
```
println!("{:.4}", iu.tangent_portfolio());
```

```
  1.1297
 -0.4619
  0.3321
```

4.0.2 New Efficient Frontier

[20]:
```
let target_returns: Vec<f64> = (0..=65).map(|x| x as f64 / 10.0).collect();
let mut weights = Vec::new();
let mut volatilities = Vec::new();

for target_return in &target_returns {
    let w = iu.two_fund_theorem(*target_return);
    let volatility = ((w.transpose() * &iu.cov * &w)[(0, 0)]).sqrt();
    weights.push(w);
    volatilities.push(volatility);
}

let w_tangent = iu.tangent_portfolio();
let rp_tangent = w_tangent.dot(&iu.rho);
let sigma_tangent = ((w_tangent.transpose() * &iu.cov * &w_tangent)[(0, 0)]).
  ↪sqrt();
```

```rust
let w_f_values: Vec<f64> = (-10..=12).map(|x| x as f64 / 10.0).collect();
let mut cml_sigmas = Vec::new();
let mut cml_returns = Vec::new();

for w_f in &w_f_values {
    let sigma_p = (1.0 - w_f) * sigma_tangent;
    let rho_p = w_f * iu.rho0 + (1.0 - w_f) * rp_tangent;

    cml_sigmas.push(sigma_p);
    cml_returns.push(rho_p);
}

let root = evcxr_figure((800, 600), |root| {
    root.fill(&WHITE).unwrap();
    let mut chart = ChartBuilder::on(&root)
        .caption("Mean-Variance Efficient Frontier", ("sans-serif", 20))
        .margin(10)
        .x_label_area_size(30)
        .y_label_area_size(30)
        .build_cartesian_2d(0.5..1.5, 0.0..6.5)
        .unwrap();

    chart.configure_mesh().draw().unwrap();

    chart
        .draw_series(LineSeries::new(
            volatilities.into_iter().zip(target_returns),
            &BLUE,
        ))
        .unwrap()
        .label("Efficient Frontier")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &BLUE));

    // Plot Tangent Portfolio
    chart
        .draw_series(PointSeries::of_element(
            vec![(sigma_tangent, rp_tangent)],
            5,
            &RED,
            &|c, s, st| {
                return EmptyElement::at(c)
                    + Circle::new((0, 0), s, st.filled())
                    + Text::new("Tangent Portfolio", (10, 0), ("sans-serif",␣
 ↪15));
            },
        ))
        .unwrap()
        .label("Tangent Portfolio")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &RED));

    // Plot Capital Market Line
    chart
        .draw_series(LineSeries::new(
            cml_sigmas.into_iter().zip(cml_returns),
            &RED,
        ))
        .unwrap()
        .label("Capital Market Line")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &RED));

    for i in 0..3 {
        chart
            .draw_series(PointSeries::of_element(
                vec![((iu.cov[(i, i)]).sqrt(), iu.rho[i])],
                5,
                &RED,
                &|c, s, st| {
                    return EmptyElement::at(c)
                        + Circle::new((0, 0), s, st.filled())
                        + Text::new(format!("S{i}"), (10, 0), ("sans-serif",␣
↪15));
                },
            ))
            .unwrap();
    }

    chart
        .configure_series_labels()
        .border_style(BLACK)
        .draw()
        .unwrap();

    Ok(())
});

root.evcxr_display();
```

The efficient frontier does intersect with the one obtained with risky asset only once, at the tangency point. This is because the tangency point represents the maximized Sharpe ratio, which is the minimium of the convex portfolio.

4.0.3 P_3

[31]:
```
let p3 = iu.one_fund_theorem_return(7.0);
println!("One-fund theorem risk-free weight (target return 7%): {}", p3);
```

```
One-fund theorem risk-free weight (target return 7%): -0.26228586091033607
```

4.0.4 P_4

[32]:
```
let p4 = iu.one_fund_theorem_volatility(2.0);
println!("One-fund theorem risk-free weight (target volatility 2%): {}", p4);
```

```
One-fund theorem risk-free weight (target volatility 2%): 0.3864225191860321
```

[34]:
```
let target_returns: Vec<f64> = (0..=65).map(|x| x as f64 / 10.0).collect();
let mut weights = Vec::new();
let mut volatilities = Vec::new();

for target_return in &target_returns {
    let w = iu.two_fund_theorem(*target_return);
    let volatility = ((w.transpose() * &iu.cov * &w)[(0, 0)]).sqrt();
    weights.push(w);
    volatilities.push(volatility);
}

let w_tangent = iu.tangent_portfolio();
let rp_tangent = w_tangent.dot(&iu.rho);
let sigma_tangent = ((w_tangent.transpose() * &iu.cov * &w_tangent)[(0, 0)]).
  ↪sqrt();


let w_f_values: Vec<f64> = (-10..=12).map(|x| x as f64 / 10.0).collect();
let mut cml_sigmas = Vec::new();
let mut cml_returns = Vec::new();

let p3_sigma = (1.0 - p3) * sigma_tangent;
let p3_rho = p3 * iu.rho0 + (1.0 - p3) * rp_tangent;
let p4_sigma = (1.0 - p4) * sigma_tangent;
let p4_rho = p4 * iu.rho0 + (1.0 - p4) * rp_tangent;


for w_f in &w_f_values {
    let sigma_p = (1.0 - w_f) * sigma_tangent;
    let rho_p = w_f * iu.rho0 + (1.0 - w_f) * rp_tangent;

    cml_sigmas.push(sigma_p);
    cml_returns.push(rho_p);
}
```

```
let root = evcxr_figure((800, 600), |root| {
    root.fill(&WHITE).unwrap();
    let mut chart = ChartBuilder::on(&root)
        .caption("Mean-Variance Efficient Frontier", ("sans-serif", 20))
        .margin(10)
        .x_label_area_size(30)
        .y_label_area_size(30)
        .build_cartesian_2d(0.5..2.0, 0.0..8.0)
        .unwrap();

    chart.configure_mesh().draw().unwrap();

    chart
        .draw_series(LineSeries::new(
            volatilities.into_iter().zip(target_returns),
            &BLUE,
        ))
        .unwrap()
        .label("Efficient Frontier")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &BLUE));

    // Plot Tangent Portfolio
    chart
        .draw_series(PointSeries::of_element(
            vec![(sigma_tangent, rp_tangent)],
            5,
            &RED,
            &|c, s, st| {
                return EmptyElement::at(c)
                    + Circle::new((0, 0), s, st.filled())
                    + Text::new("Tangent Portfolio", (10, 0), ("sans-serif",␣
↪15));
            },
        ))
        .unwrap()
        .label("Tangent Portfolio")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &RED));

    // Plot Capital Market Line
    chart
        .draw_series(LineSeries::new(
            cml_sigmas.into_iter().zip(cml_returns),
            &RED,
        ))
        .unwrap()
        .label("Capital Market Line")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &RED));

    chart
        .draw_series(PointSeries::of_element(
            vec![(p3_sigma, p3_rho)],
            5,
            &GREEN,
            &|c, s, st| {
                return EmptyElement::at(c)
                    + Circle::new((0, 0), s, st.filled())
                    + Text::new("P3", (10, 0), ("sans-serif", 15));
            },
        ))
        .unwrap()
        .label("P3")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &GREEN));


    chart
        .draw_series(PointSeries::of_element(
            vec![(p4_sigma, p4_rho)],
            5,
            &GREEN,
            &|c, s, st| {
                return EmptyElement::at(c)
                    + Circle::new((0, 0), s, st.filled())
                    + Text::new("P4", (10, 0), ("sans-serif", 15));
            },
        ))
        .unwrap()
        .label("P4")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 10, y)], &GREEN));

    for i in 0..3 {
        chart
            .draw_series(PointSeries::of_element(
                vec![((iu.cov[(i, i)]).sqrt(), iu.rho[i])],
                5,
                &RED,
                &|c, s, st| {
                    return EmptyElement::at(c)
                        + Circle::new((0, 0), s, st.filled())
                        + Text::new(format!("S{i}"), (10, 0), ("sans-serif",␣
↪15));
                },
            ))
            .unwrap();
    }

    chart
        .configure_series_labels()
        .border_style(BLACK)
        .draw()
        .unwrap();

    Ok(())
});

root.evcxr_display();
```